UNIVERSITÀ DEGLI STUDI DI TORINO
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Magistrale in Informatica

Riassunto di

Database Multimediali

Enrico Mensa

Basato sulle lezioni di: *Maria Luisa Sapino*

Indice

Capitolo 1

Introduzione

1.1 Media e multimedia

Che cosa è un *media*?
Possiamo vedere il media come una maniera di comunicare informazioni in modo compatto. Pensiamo ad una fotografia: con un colpo d'occhio istantaneo è possibile capire molto della situazione o dell'oggetto che è rappresentato dalla figura. Fra i media più comuni troviamo il testo strutturato (XML)/non strutturato (un foglio di giornale), immagini (GIF, TIFF, JPG), video (MJPG, MPEG, AVI), audio (MP3, FLAC) e 3D media (VRML, X3D).

Quando più media concorrono all'obiettivo comune di fornire informazioni in modo compatto abbiamo un *multimedia*. Il telegiornale è un esempio lampante: abbiamo il giornalista (audio) che descrive alcune immagini (video) mentre in sovraimpressione scorrono ulteriori notizie (testo).
Un'altra definizione importante è quella di *hypermedia*, che definisce dei particolari multimedia che prevedono l'interazione esplicita dell'utente durante il loro utilizzo (delle slide, ad esempio). La caratteristica peculiare dell'hypermedia è il fatto che uno stesso hypermedia usato da utenti diversi può portare a esiti differenti (poiché la decisione dell'utente è fondamentale e quindi può far variare il risultato finale dell'interazione).

Il nostro obiettivo è quello di *smembrare* questi media/multimedia in modo da poterli memorizzare e successivamente interrogare, ovvero essere in grado di eseguire query come ''Trova tutte le mie foto'' piuttosto che ''Trova tutte le foto con un sorriso''.

1.2 Tre aspetti fondamentali

Una volta definito cosa sia un media, cerchiamo di capire quali sono le sfide che dobbiamo superare per svolgere il nostro mestiere di databasisti del multimedia.

Fra i tanti, tre degli aspetti da tenere sotto considerazione:

- Eterogeneità semantica.
- Eterogeneità delle risorse.
- Interattività e personalizzazione.

Prima di parlare dettagliatamente di ciascuno di questi punti consideramo questo esempio tramite il quale sarà facile contestualizzare le problematiche che discuteremo.

Immedesimiamoci nella crew di una stazione di polizia che deve condurre una investigazione. Disporremo di: dati video (telecamera di sorveglianza), dati audio (intercettazioni), fotografie (telecamere di sorveglianza e foto segnaletiche), documenti di testo (report dei testimoni), dati formalizzati (transazioni bancarie), dati geografiche (tracce GPS). Capiamo ora cosa s'intende con ''eterogeneità delle risorse'' : tutte le informazioni di cui disponiamo, anche se di natura estremamente diversa, devono poter essere correlate al fine di rispondere a query eventualmente anche molto complicate. Per esempio potremmo chiederci: ''Trova tutti i record di ogni criminale che assomigliano a quello nell'immagine `surv_img.png` e che hanno eseguito un trasferimento di più di 50.000 euro negli ultimi cinque mesi. Si restituiscano anche i record dei complici dei suddetti criminali''.

1.2.1 Eterogeneità semantica

Dimensioni gerarchiche. Sotto il cappello eterogeneità semantica mettiamo il trattamento di dati di tipo *spaziale* e di tipo *temporale*. Queste due tipologie sono in realtà simili poiché condividono una natura gerarchica, ovvero se diciamo ''negli scorsi 10 minuti ha piovuto'' intendiamo anche che ''oggi'' ha piovuto e ovviamente che ''quest'anno ha piovuto''. Allo stesso modo dire che troviamo in piemonte implica che ci troviamo in Italia e quindi in Europa. Per poter ottenere inferenze di questo tipo abbiamo bisogno di diversi strumenti:

- Un *modello* che rappresenti i dati.
- Una *specifica* che classifichi i dati.
- Una architettura di *indirizzamento* che permetta un veloce reperimento dei dati (sotto l'aspetto fisico).
- Un sistema di *retrieval* che effettui il ranking e ci restituisca i dati interessanti.
- Metodi di visualizzazione che siano efficaci ed user-friendly.

Dipendenza dall'utente e dal contesto. Ulteriore aspetto da considerare per quanto riguarda l'eterogeneità semantica è la dipendenza che intercorre fra ciò che deve fare il nostro sistema rispetto alle scelte dell'utente e la conformazione del contesto. Dire che una montagna è alta è infatti un concetto relativo all'utente che la pronuncia: anche i concetti di liscio, frastagliato, luminoso, ecc. sono tutti soggettivi e quindi è impossibile avere una risposta esatta (ecco perché ci serve un sistema di ranking).
È dunque evidente che non possiamo avere una risposta *esatta*, ma possiamo sfruttare una categorizzazione fuzzy. Lo stesso discorso vale anche per l'aspetto del contesto: il concetto di ricchezza ad esempio è dipendente dalla ricchezza di chi ci sta intorno. A tutte queste imprecisione si aggiunge anche l'eventuale imprecisioni degli strumenti che adoperiamo per reperire e studiare i dati (introduciamo noi stessi dell'imprecisione in questo caso).

Per questi motivi l'interazione con un database multimediale è di tipo *iterativo*: man mano i risultati vengono affinati avvicinandosi (si spera) sempre più al risultato desiderato.

Diversi livelli di qualità possibili. Certi prodotti possono esistere in versioni di diversa qualità (pensiamo alla risoluzione di una immagine). Evidentemente la scelta della qualità sarà dettata da un trade-off che prevede come contropartita una esecuzione più veloce.

1.2.2 Eterogeneità fisica

Fra le problematiche relative all'eterogeneità fisica comprendiamo i costi di storage (trascurabili oggigiorno), di delivery (vogliamo che due media vengano trasmessi contemporaneamente, e.g. un file video) e ovviamente i costi di processing.
Ottenere maggiore precisione in un risultato ha un costo, dunque bisogna valutare attentamente di cosa abbiamo realmente bisogno. Una delle tecniche adottate nello streaming è la *graceful degradation*, per cui il video trasmesso non si ferma all'improvviso ma man mano diminuisce il numero di frame e la loro qualità per permettere una trasmissione costante anche a fronte di una minore portanza della rete.

1.2.3 Interazione

L'ultimo aspetto da considerare è quello dell'interazione. Riportando l'esempio dello streaming di prima, un utente non si accorgerà di un drop dei frame se questi continuano ad essere visualizzati sequenzialmente ogni 100ms, mentre sopra questa soglia l'utente inizia a notare lo stacco fra i frame. Per evitare che questo accada è possibile adottare politiche di *prefetching* (mentre si carica un dato se ne caricano anche altri probabilmente correlati) e di *resource allocation*.

Evidentemente la tipologia di utente che sfrutta la base di dati è un aspetto fondamentale da tenere in considerazione: in un database medico, ad esempio, a seconda che sia un chirurgo, un inferimere o un paziente ad interrogare la base di dati, dovremo saper dare la risposta più utile (quindi priva di informazioni che l'utente non può capire).
Non da meno è il problema del *semantic gap*: il pensiero non può essere rappresentato con oggettività e perfezione, dunque ci sarà sempre una distanza fra l'utente e la macchina.

1.3 Cos'è un modello dei dati

Abbiamo parlato, trattando il problema dell'eterogeneità semantica, della necessità di delinare un modello per trattare i nostri dati. Ma cos'è un modello? Dicesi modello un insieme di vincoli che descrivono la struttura ed il comportamento dei dati. Nel modello relazionale, ad esempio, la struttura è tabellare mentre il comportamento è regolato dal linguaggio SQL.

Il ruolo del modello

Un buon modello deve poter:

- Descrivere il dato (modello *concettuale*, e.g. modello ER).
- Permettere la memorizzazione dei dati (modello *fisico*, e.g. file sequenziali).
- Permettere la validazione e la ridondanza dei dati (modello *logico*).
- Permettere il *recupero dei dati* (operazioni di confronto, indicizzazione, processing delle query).

Si ha dunque da un lato, il modello fisico che descrive come i dati devono essere memorizzati, e dall'altro un modello concettuale che descrive i concetti della realtà. In mezzo troviamo il modello logico che funge da intermediario e permette il mapping fra i concetti della realtà e la loro rappresentazione fisica.

1.4 Le query

Facciamo ora un excursus delle possibili query che siamo interessati a risolvere. Vediamo le varie tipologie di query ordinandole per *complessità*.

Query su metadati. La prima tipologia è la più classica, le query per *metadati*. Questa tipologia è l'unica che prevede la possibilità di un risultato esatto (preciso). Si tratta infatti di una interrogazione del tipo ''Tutti i quadri di Picasso'' ma non sfruttiamo le immagini per risolvere l'interrogazione bensì i metadati ad esso associati (un semplice confronto fra stringhe nel campo `autore` della fotografia).

Query per esempio. Decisamente più interessanti (per il nostro studio) sono le query per esempio. Viene fornita una certa `immagine.png` e si richiede di trovare immagini che siano:

- Esattamente uguali a quella fornita (precisione esatta).
- Parzialmente uguali alla foto fornita: si introduce quindi il concetto di *similarità*, ovvero si cercano foto che abbiano le stesse caratteristiche di quella fornita (esempio di query:''Foto con questo viso''). Chiaramente una certa foto potrà essere giudicata più o meno simile secondo alcune caratteristiche (quindi potremo stilare un ranking fra le varie foto del database).

È anche possibile fornire una descrizione testuale anziché una immagine di esempio.

Object query. Una tipologia di query molto più raffinata. Possiamo distinguere tre tipi di somiglianza che vogliamo esprimere:

- *Similarità visuale*: vengono confrontate le forme contenute nelle immagini senza interpretarle in alcun modo. La presenza di forme simili (e nello stesso posto) di una fotografia porta ad un maggiore livello di somiglianza.
- *Similarità semantica*, ovvero il contenuto della fotografia viene interpretato e quindi è interrogabile. Ad esempio, una bottiglia di acqua Lete e una di acqua San Benedetto possono apparire visivamente molto diverse, ma sono semanticamente uguali (due bottiglie).
- *Similarità spaziale*: uno stesso oggetto potrebbe apparire in due fotografie ma in posizioni diverse. Vorremmo però cogliere questo aspetto e quindi trascendere dalla posizione spaziale degli oggetti catturati nella fotografia.

1.5 Cos'è un'immagine

Entriamo nel vivo del discorso cercando di capire cosa sia una immagine; scegliamo di parlare delle immagini poiché sono uno dei multimedia più diffusi.

Il nostro intento è quello di trovare una forma di rappresentazione che mantenga le informazioni contenute nel multimedia ma allo stesso tempo sia interrogabile e non occupi troppo spazio. Ovviamente ci sarà un trade-off fra queste caratteristiche. Quando il database acquisisce un nuovo dato ne genera quindi un *surrogato* che verrà mantenuto nel database e usato nella computazione. Solo al momento della restituzione dei risultati dal surrogato (se selezionato come risultato) si risalirà al dato iniziale affinché l'utente possa visualizzarlo. Cerchiamo dunque un buon surrogato per una immagine.

Immagine come matrice bidimensionale. Ad ogni pixel viene associato un elemento della matrice che ne riporterà le caratteristiche cromatiche (ad esempio un codice RGB). Questo tipo di rappresentazione è poco conveniente per diverse ragioni:

- L'occupazione di spazio è elevata.
- È impossibile effettuare confronti solamente fra matrici di dimensioni uguali (quindi fra immagini della stessa dimensione),
- È impossibile effettuare query per esempio con similarità parziale o più complesse (il confronto è fatto pixel a pixel, dunque se lo stesso oggetto si trova in due posizioni diverse non viene rilevato come compresente nelle due immagini).

Immagine come collezione di oggetti correlati spazialmente. Potremmo rappresentare l'immagine come una serie di oggetti che hanno una descrizione visuale e semantica. Tra loro gli oggetti verranno legati da un rapporto di spazialità (come sono posizionati nell'immagine).

Immagine come vettore di features. Si scelgono diverse caratteristiche salitenti dell'immagine (e.g. l'istogramma dei colori) e si sintetizzano in un unico vettore. Ogni caratteristica è detta *feature* dunque il vettore che le rappresenta tutte è detto *feature vector*. La scelta di una feature è fonadamentale: mantenere informazioni inutili è costoso e vogliamo evidentemente evitarlo. Questa ultima modalità è la più usata ed è quella che studieremo dettagliatamente in seguito.

1.6 La necessità di un nuovo database

Cerchiamo di capire perché si renda necessario sviluppare un nuovo sistema di database piuttosto che sfruttare quelli già esistenti. Delineiamo quindi, prima di tutto, quale sia il nostro obiettivo e cosa cerchiamo di ottenere.

1.6.1 Di cosa abbiamo bisogno

Un database di immagini è:

- Una collezione di immagini (in termini di storage).
- Un entità in grado di risolvere query (mapping fra il linguaggio di query e il modello dei dati) e di restituire i risultati in maniera ordinata per rilevanza (ranking).
- Un sistema di visualizzazione che è in grado di mostrare il risultato delle query agli utenti.

Da questi tre punti capiamo che abbiamo bisogno di:

- Trattare immagini di grandi dimensioni (ed i loro surrogati).
- Gestire proprietà visuali (capacità di image processing) e semantiche (gestione di metadati e feedback dell'utente).
- Engine per la valutazione della similarità (quindi nuove strutture per gli indici e supporti per fare ranking).
- Linguaggio di query adatto a specificare le richieste dell'utente.

1.6.2 Features papabili

Abbiamo già detto che si rende necessario scegliere delle caratteristiche salienti (dette features) secondo le quali costruire il nostro surrogato (il vettore di features). Le features considerabili sono (in ordine di complessità di calcolo):

- Distribuzione del colore (quindi istogramma dei colori).
- Linee di separazione (nette, tratteggiate, sottili, spesse).
- Trama (rugosità, uniformità, tessitura). Ad esempio una foglia d'erba ed una foglia di acero si possono distinguere dalla trama ''a fili'' tipica dell'erba rispetto alla trama più lineare della foglia.
- Segmenti dell'immagine (forma del blocco, colore e posizione spaziale).
- Oggetti (caratteristiche visuali e semantiche).
- Metadati (inseriti tendenzialmente a mano dall'utente).

1.6.3 Esempi di query

Riportiamo quindi alcune delle query che saremmo interessati a risolvere:

- Trova tutte le immagini create da Jhon (query su metadati).
- Trova tutte le immagini simili a `img.png` (query per esempio).
- Trova le prime cinque immagini più simili a `img.png` (query cosiddetta *top-K* con $k = 5$).
- Trova tutte le foto che contengono una forma simile a `img.png` (query per esempio).
- Trova tutte le immagini soleggiate (è necessaria una conoscenza semantica).
- Trova tutte le immagini che contengono una macchina.
- Trova tutte le immagini che contengono due oggetti, il cui primo assomiglia a `img.png` ed il secondo è una macchina. Il primo oggetto si trova alla destra del secondo oggetto. Si restituisca la semantica di questi due oggetti.

Dunque, se la query consistesse di questa immagine:

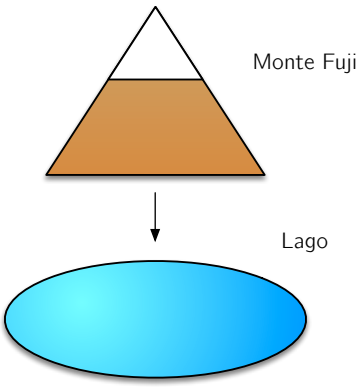


Figura 1.1: Esempio di query.

staremmo dicendo che desideriamo una fotografia dove ci sia una montagna (di quella forma) e ci sia un lago (di quella forma) e che la montagna sia spazialmente sopra al lago.

In un linguaggio SQL-like potremmo vedere la query in questo modo:

```
1 SELECT image P, object obj1, object, obj2
2 where P contains obj1
```

```
    and P contains obj2
    and obj1.semantical_property is_like "mountain"
    and obj1.image_property image_match "FujiMountain.png"
    and obj2.semantical_property is_like "lake"
    and obj2.image_property image_match "lakeImageSample.png"
    and obj1.position is_above obj2.position
```

Listing 1.1: Traduzione in pseudo query.

Si noti l'uso dell'operatore `is_like`, un operatore evidentemente fuzzy. Sarà dunque possibile confrontare ogni fotografia nel database con questa descrizione alla ricerca del best match.

Potremmo quindi assegnare questi punteggi:

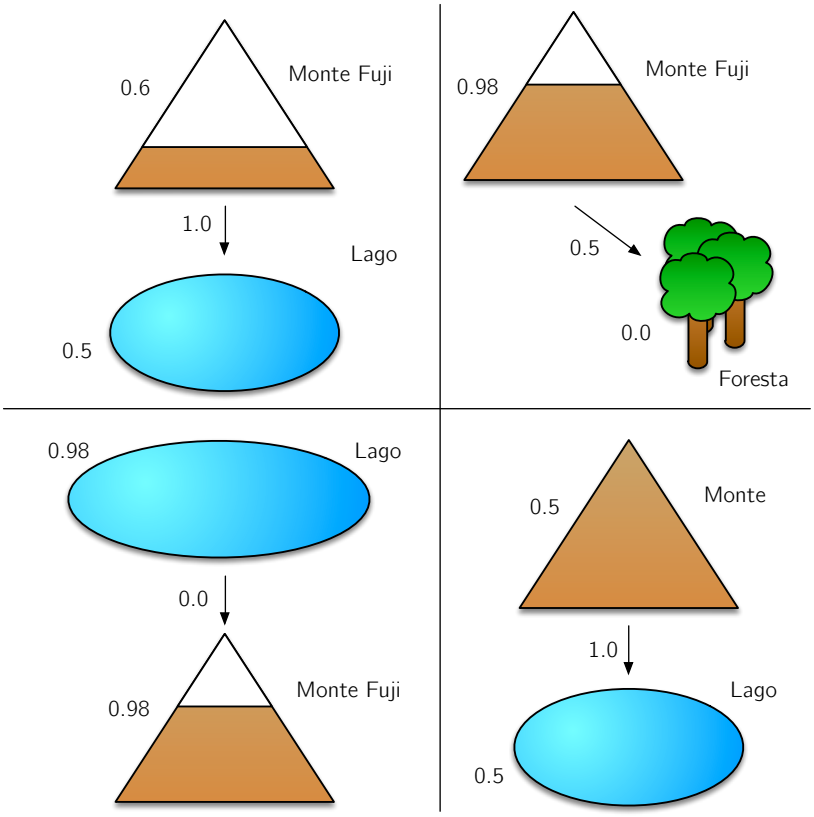


Figura 1.2: Esempio di valutazione di similarità.

Combinare i punteggi. Risulterà poi necessario combinare i vari punteggi per ottenere un valore unico di similarità. A seconda di quale *combination function* si sceglie, avremo risultati molto diversi. Alcune delle possibilità sono:

- *max*(*punteggi*), che la semantica dell'OR (basta un punteggio alto per avere similarità alta).
- *min*(*punteggi*), che è la semantica dell'AND (il punteggio più basso è quello che definisce la similarità, quindi vogliamo tutti i punteggi alti).
- Pesare ogni punteggio a seconda delle preferenze dell'utente.

1.6.4 L'inadeguatezza dei modelli preesistenti

Facciamo ora una rapida carrelata dei vari modelli esistenti e, mettendo a confronto quanto offerto rispetto ai nostri obiettivi, capiamo perché abbiamo bisogno di una soluzione ad hoc per adempiere al nostro goal.

Il modello relazionale

Come sappiamo il modello relazionale è perfetto per le applicazioni di business poiché i dati che vengono mantenuti sono piccoli (testuali tendenzialmente) e sono direttamente visualizzabili nel momento del retrieval. Abbiamo un approccio dichiarativo (si dice cosa si vuole e non come si vuole ottenerlo) e l'aggiornamento dei dati è estremamente agevolato (query di `UPDATE`). Abbiamo anche molti moduli per la gestione della concorrenza direttamente implementati all'interno del DBMS.

Problematiche. Purtroppo i sistemi relazionali non sono adatti per adempiere ai nostri scopi. Infatti:

- I dati relativi alle immagini sono difficili da mantenere all'interno del database.
- Non esiste alcun sistema di comparazione fra immagini.
- È impossibile effettuare matching parziale (le query sono esatte).
- Conseguentemente al punto precedente non esiste ranking.
- Mancano le primitive per poter avere una computazione più performante.

Una soluzione potrebbe essere quella di implementare in un linguaggio ospite le varie funzioni di elaborazione dei multimedia e quindi di ranking, ma questo è, come intuibile, estremamente costoso e poco automatizzato.

Un'altra grossa mancanza consta nell'impossibilità di rappresentare la semantica degli oggetti rappresentati nei media e quindi l'assenza di relazioni gerarchiche/aggregative. Più precisamente avremmo bisogno di:

- Ereditarietà gerarchica (relazioni di tipo *PART-OF*), per esempio una ruota è parte di una macchina.

- Ereditarietà aggregativa (relazioni di tipo *IS-A*), per esempio un cane è un mammifero.

Object Data Models

Grazie a questi modelli possiamo mappare entità su strutture dati e comportamenti su funzioni. Dunque, una relazione è descritta come riferimento ad oggetto oppure come entità singola. Sono dunque risolti i problemi in merito alle relazioni gerarchiche e aggregative che sono invece ben trattate da questi modelli. La possibilità di definire metodi personalizzati rende poi l'implementazione delle funzioni di ranking e di similarità estremamente semplici (ma ancora troppo manuali).

Problematiche. Anche questi modelli soffrono di problematiche: c'è molto overhead (ed è difficile ottimizzare), manca il matching parziale e il processing delle query è guidato dai costi e non dalla similarità (cioè se avessimo ranking potremmo mostrare i primi 10 risultati e calcolare gli altri in seguito, ottimizzando così i costi).

Object Relational Database

Questo tipo di database fonde i punti positivi dei due modelli di cui abbiamo appena parlato, ma ovviamente continuano a soffrire delle stesse problematiche comuni: manca il partial match, non esiste ranking e il processing delle query è guidato dai costi e non dalla similarità.

1.6.5 La soluzione finale

Non resta quindi che sviluppare il nostro sistema database ''pescando'' le varie funzionalità dai modelli preesistenti e da discipline originariamente nate in altri campi di ricerca. Più precisamente prendiamo:

- Dai deductive database (particolari database guidati dalla logica ed in grado di risolvere query booleane):
  - Capacità di inferenza.
- Database fuzzy/probabilistici (la differenza fra i due consta nel fatto che i database probabilistici hanno sempre somma delle probabilità associate agli eventi pari a 1 mentre in quelli fuzzy non è detto che sia così):
  - Nulla è vero o falso.
  - I risultati non sono esatti.
- Database spaziali/temporali:

  - Modelli vettoriali.
  - Query di range (dato un oggetto nel punto x, ottenere gli oggetti nel raggio di 50km).
  - Query visuali e dichiarative.

- Data mining (trovare regole o informazioni non banali mantenute implicitamente all'interno di grandi moli di dati):
  - Trovare pattern.
  - Rilevare feedback degli utenti.

1.7 Rappresentazione digitale di suoni ed immagini

Concludiamo la nostra introduzione parlando di come è possibile trasformare un media (prenderemo in esame i casi di audio e immagini) anologico in un media rappresentabile dalla nostra macchina e quindi digitale.
I modelli che sono stati sviluppati sono tutti basati sul principio di *imitare quel che fa il cervello*: se il computer processa le immagini così come fa il cervello, non avremo un overhead di lavoro che non sarà in grado di percepire alcun umano e allo stesso tempo riusciremo a catturare più facilmente il significato semantico dei media che troviamo.

1.7.1 L'esempio dell'audio

Come sappiamo l'audio è un media analogico, una funzione continua nel tempo. Per poter passare dall'analogico al digitale compiamo una operazione di *camponamento* (per discretizzare l'asse x) e una operazione di *quantizzazione* (per discretizzare l'asse y). Spiegheremo dettagliatamente in cosa constano queste fasi successivamente.
Il punto fondamentale è che l'audio colto da un microfono è una variazione di pressione nell'aria, poi rappresentata in maniera *analoga* su un supporto (ecco il perché del nome analogico). La rappresentazione digitale dell'audio è invece una *approssimazione* della sua versione analogica.

1.7.2 Conversione analogico-digitale

Andiamo quindi a capire come avviene la conversione dall'analogico al digitale (abbiamo già accennato alle fasi di campionamento e quantizzazione).

Prima di tutto: quali vantaggi porta la digitalizzazione?

1. Le tecniche di decodifica sono efficenti: è possibile avere alta qualità.
2. È possibile copiare il dato originale senza alterazione del supporto.

3. È possibile effettuare compressione (e quindi risparmiare storage).

Le fasi della conversione

Partiamo dal nostro segnale analogico (sull'asse delle x c'è il tempo mentre sull'asse delle y c'è la nostra misura):

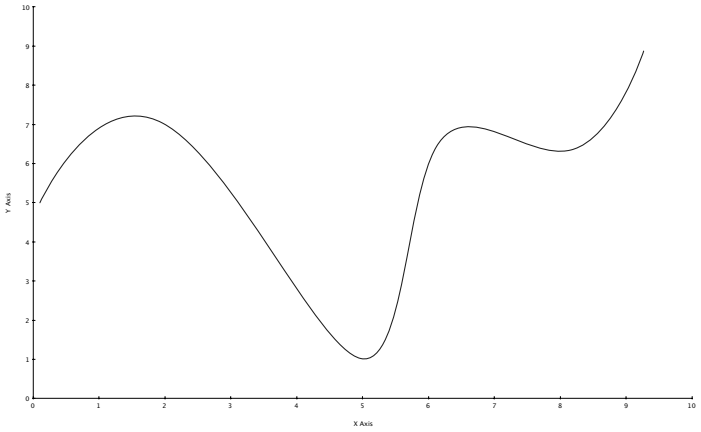


Figura 1.3: Un segnale analogico (funzione continua).

La fase di campionamento. La fase di campionamento consiste nel prendere un intervallo temporale t a piacere, e ad ogni t, $t * 2$, $t * 3$, ecc. si considera il valore della funzione. A questo punto da una funzione continua con infiniti punti ne otteniamo una discreta, con un numero di punti tanto più alto quanto è piccolo il valore di t:

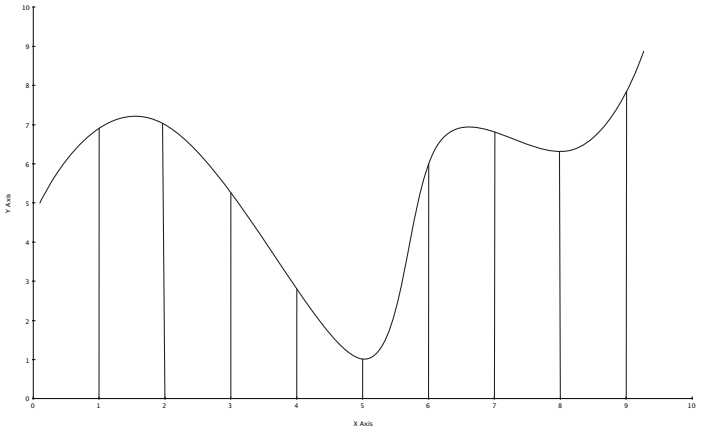


Figura 1.4: Campionamento con $t = 1$.

Abbiamo ora discretizzato l'asse delle x.

La fase di quantizzazione. Come è noto il computer sfrutta una rappresentazione binaria. Ma avendo noi campionato a partire da una funzione continua, potremmo potenzialmente avere moltissimi diversi valori sulle y, anche se discostati fra loro di pochissimo. Compito della fase di quantizzazione è quello di definire degli intervalli sull'asse y entro i quali rappresentaranno l'insieme di valori di x che variano nell'intorno di quell'intervallo. Definiamo quindi gli intervalli sulle ordinate:

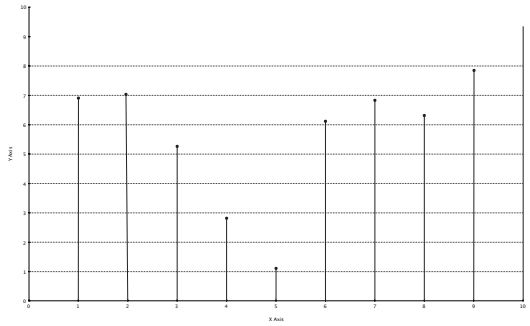


Figura 1.5: Si è scelto un intervallo $y = 1$.

Dopodiché allineiamo tutti i valori contenuti nell'intervallo in questione su un unico valore (diciamo il punto medio dell'intervallo):

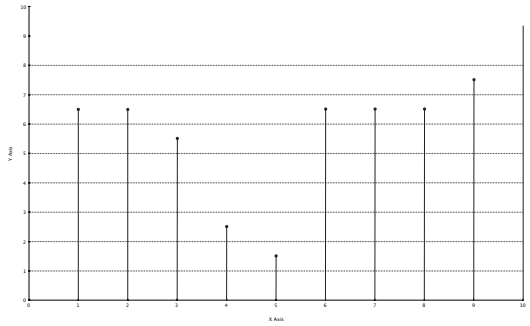


Figura 1.6: Punti nello stesso intervallo diventano punti di uguale valore.

Non resta che tracciare la nostra funzione discreta, approssimazione dell'analogica (Figura 1.7).
È evidente che più è piccolo l'intervallo di campionamento, più precisa sarà la misura (ma più onerosa la memorizzazione del file). Al contrario, più bit usiamo per campionare e maggiore sarà il numero di combinazioni di simboli che potremo usare, quindi avremo più intervalli nei quali categorizzare i nostri valori ovvero avremo maggiore pecisione.

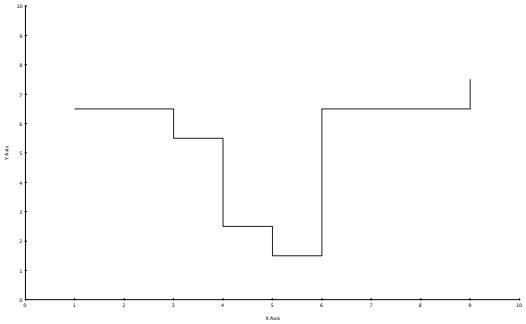


Figura 1.7: Funzione discreta.

1.7.3 Immagini: il modello umano

Trattare una immagine è più complesso di trattare un file audio: dobbiamo ad esempio considerare il fatto che uno stesso oggetto da più prospettive abbia caratteristiche visive diverse (in termine di colori o di numero di pixel magari), inoltre c'è sempre da considerare il fattore della lumonosità: uno stesso oggetto investito da diversi fasci di luce può risultare cromaticamente diverso.
Il problema del nostro modello è quindi quello di dover trovare feature che trascendano da queste caratteristiche e che siano il più generali possibili: è molto complesso.
Ma la maggiore differenza fra il modello umano e quello delle feature è il fatto che il cervello sia in grado di *creare le proprie feature* in modo dinamico: non sono imposte a monte da nessuno. Facebook e Google stanno lavorando in questa direzione, ovvero cercano di creare un sistema che capisca da solo quali siano le feature più descrittive e più interessanti.

In ogni caso il nostro obiettivo è sempre quello di comprimere l'informazione per poterla memorizzare. La scelta su cosa comprimere all'interno di una immagine ricadrà ovviamente su quelle caratteristiche che l'uomo non può vedere o percepire. Ancora una volta, quindi, la prima domanda da porsi è: ''come si comporta l'occhio umano''?

Percezione visiva e visione foveale

L'analisi della scena visiva (ad esempio l'osservazione di un quadro o di un panorama) è strettamente associata alla *visione foveale*. L'occhio scansiona la parte osservata sfruttando movimenti rapidi (detti *movimenti saccadici*) alternati a *fissazioni*. Gli studi dimostrano che le fissazioni non sono distribuite in modo uniforme ma ci sono zone che vengono fissate più spesso rispetto ad altre. Giacché l'unico momento in cui il cervello può acquisire informazione visiva è quando c'è una fissazione (e non durante un movimento saccadico) è interessante capire su quali punti si fissi l'occhio, o più precisamente studiare:

- La distribuzione spaziale delle fissazioni (quali sono le zone più interessanti, cioè più informative).
- La durata delle fissazioni (ci può dire quanta informazione locale si trova in una certa zona fissata).
- La sequenza delle fissazioni (ci permette di studiare la successione temporale dell'analisi dell'informazione visiva fatta dal sistema nervoso centrale).

Da questa analisi capiamo che può essere interessante, ad esempio, usare più istogrammi su una immagine sola e poi mediare le informazioni ottenute da ogni istogramma. Parleremo in seguito di questa tecnica.

1.7.4 Codificare le immagini

Nel mondo reale le immagini sono un insieme continuo di due fattori: *luce* e *colore*. Ancora una volta, dovendo trattare una informazione continua, ricorreremo al campionamento e alla quantizzazione. Il modo più intuitivo di compiere queste operazioni è quello di sovrapporre ad una immagine un reticolo di punti, ottenendo così una grafica *bitmap* (o *raster*). Ad ogni elemento della griglia (detto *pixel*) viene associato un colore che è l'intensità media della cella considerata.

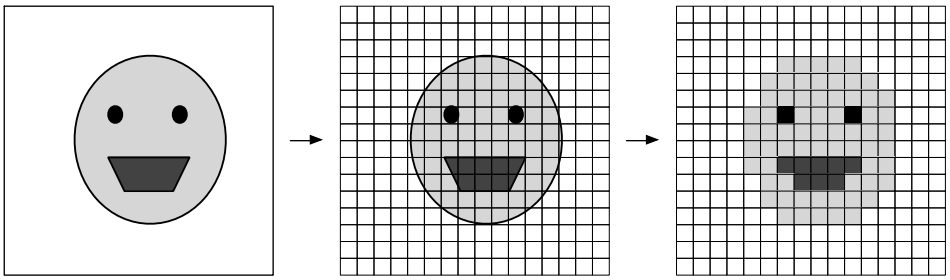


Figura 1.8: Rappresentazione in pixel.

La descrizione dei colori

Il colore è descrivibile tramite tre attributi percettivi:

1. La *tinta* (hue), che riferisce ad una famiglia di colori (tutti i rossi ad esempio). L'occhio umano è in grado di distinguere 250 tinte diverse.
2. La *saturazione* (saturation), che indica quanto è pura (non mescolata al bianco) la tinta. Ad esempio blu e celeste hanno la stessa tinta ma sono a saturazione diversa. L'apparato umano è in grado di distinguere circa 100 saturazioni differenti.

3. La *chiarezza* (lightness), la quantità totale di luce che il colore emette rispetto ad un oggetto ugualmente illuminato che appare bianco nella scena.

Come vedremo, ci sono diversi modelli in grado di rappresentare il colore. Li studieremo nel dettaglio in seguito.

Capitolo 2

Features, vettori e distanze

2.1 Il modello vettoriale

Entriamo nel vivo del discorso introducendo un modello fondamentale: il modello vettoriale. Come sappiamo una immagine è astrabile come una matrice (in due dimensioni quindi): l'idea è quella di ottenere un vettore di *features* (caratteristiche principali) che funga da surrogato al posto della matrice (che come abbiamo già discusso è molto scomoda). È importante capire che in un solo vettore rappresentiamo tutte le features, una dopo l'altra (almeno dal punto di vista astratto, poi in pratica i confronti saranno fatti feature per feature su frazioni del vettore).
Quindi se abbiamo ad esempio un vettore di feature con tre componenti, possiamo rappresentare le immagini del database su uno spazio vettoriale in tre dimensioni:

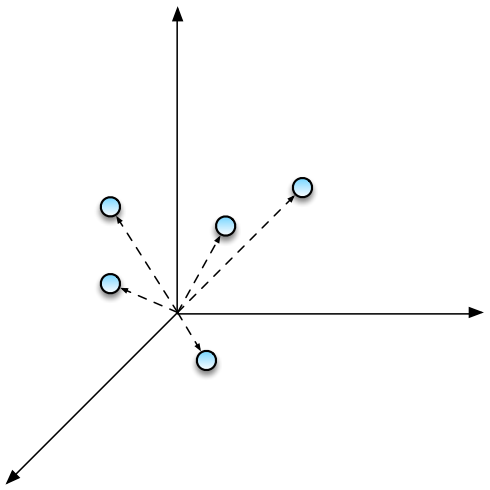


Figura 2.1: Cinque immagini, rappresentate dal loro vettore di features.

Capiamo dunque che il concetto di similarità fra vettori è equiparabile alla ''vicinanza'' fra questi. Ad esempio considerando un certo vettore query (in verde nell'immagine) potremmo fare una query per range:

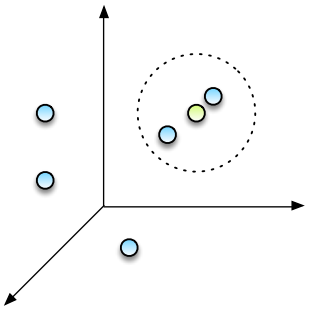


Figura 2.2: Query per range.

2.1.1 Il concetto di distanza

Formalizziamo la nostra intuizione: dati due vettori $A = [a_1, a_2, a_3]$ e $B = [b_1, b_2, b_3]$, come facciamo a capire quanto sono differenti?

La distanza euclidea

L'approccio più inutitivo è quello della distanza, come abbiamo già detto. Calcoliamo quindi la *distanza euclidea*:

$$\Delta(A, B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2} \tag{2.1}$$

A questo punto, avendo due vettori (diciamo B e C) è facile capire quali di questi sia più simile ad un altro vettore A:

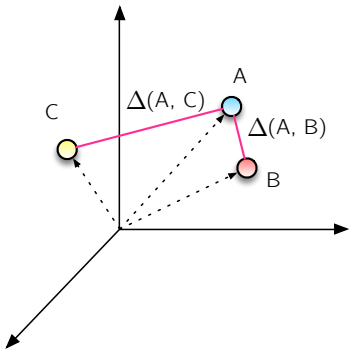


Figura 2.3: B è più simile ad A di quanto lo sia C.

Si ha quindi una correlazione fra *bassa distanza* e *somiglianza*.

Quello che andiamo a fare è una differenza componente a componente, quindi questo tipo di confronto ha senso solamente in situazioni ''fair''. Se ad esempio confrontiamo due istogrammi dei colori (che contengono dunque delle frequenze) possiamo tranquillamente confrontare una immagine 10×10 con una 100×100 poiché le frequenze non sono condizionate dal numero di pixel dell'immagine. Ma se invece di un istogramma con frequenze manteniamo il numero di occorrenze di pixel di un certo colore, allora indubbiamente l'immagine 10×10 sarà sempre estremamente diversa da quella 100×100 anche se, per assurdo, fossero una l'ingrandimento dell'altra.

2.1.2 La cosine similarity

Consideriamo questo esempio: Stando alla distanza euclidea, B è evidente-

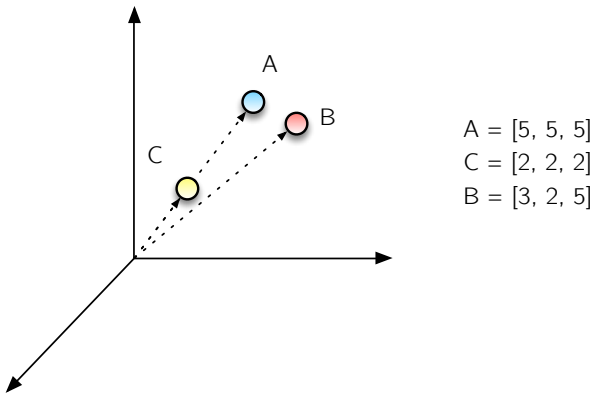


Figura 2.4: Tre vettori di features.

mente più simile ad A rispetto a quanto lo sia C. Però notiamo anche che c'è una somiglianza fra il vettore A ed il vettore C nei termini di composizione delle componenti. In entrambi i vettori tutte le componenti sono uguali. Questo fatto è catturabile misurando il coseno fra A ed E: quando l'angolo fra due vettori è 0, il coseno è 1, quindi la similarità è massima (supponendo che vari da 0 a 1). Definiamo secondo questo principio la *cosine similarity*:

$$cos(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{|\,\vec{x}\,||\,\vec{y}\,|} \tag{2.2}$$

Dove il prodotto scalare fra $\vec{x}$ e $\vec{y}$ è calcolato come:

$$\text{dot product} \qquad \vec{x} \cdot \vec{y} = \sum_{i=1}^{n} x_i \cdot y_i$$

2.1.3 Generalizzando: le misure metriche

Abbiamo visto due diverse misure si similarità. Qual è la migliore? Ovviamente dipende dall'applicazione che ne dobbiamo fare. In generale, però, vogliamo avere misure di similarità che facciano parte della famiglia delle cosiddette *misure metriche*. Le misure metriche godono infatti di particolari proprietà (che stiamo per introdurre) che rendono possibili (e sensate) molte delle operazioni che facciamo sui vettori.

Assiomi delle misure metriche

Una qualsiasi funzione d che esprima una distanza deve soddisfare i seguenti assiomi (con la lettera s rappresentiamo un vettore):

- *Self-minimality*: $d(s, s) = 0$, cioè la distanza fra un oggetto e se stesso deve essere 0.
- *Minimality*: $d(s_1, s_2) \geq d(s_1, s_1)$, cioè la distanza fra due oggetti qualsiasi è maggiore o uguale alla distanza fra un oggetto e se stesso.
- *Simmetry*: $d(s_1, s_2) = d(s_2, s_1)$.
- *Triangular inequality*: $d(s_1, s_2) + d(s_2, s_3) \geq d(s_1, s_3)$.

Sebbene tutti questi assiomi siano fondamentali, la disuguaglianza triangolare è particolarmente importante poiché ci permette di potare molte delle scelte all'interno dell'albero ontologico di un database (ne parleremo in dettaglio): in sostanza ci risparmia di percorrere molte strade sull'albero escludendone una buona quantità ogni volta che compiamo una scelta.

2.1.4 Generalizzando: metriche di Minkoswki

Esiste una intera famiglia di metriche (di cui la distanza euclidea è una elemento) che sono dette le *metriche di Minkoswki*. Tali metriche vengono indicati come $Li - metric$ dove i è un certo indice che varia da 1 a infinito. La definizione generale di una metrica di minkowski è la seguente:

$$Li - metric : d = (dX^i + dY^i)^{\frac{1}{i}} \tag{2.3}$$

Abbiamo quindi che più l'indice della metrica è alto, più le distanze vengono attutite. Fra le più importanti metriche di Minkoswki troviamo la $L1$ e la $L2$, così definite:

$$L1 - metric : d = (dX + dY)$$
$$L2 - metric : d = (dX^2 + dY^2)^{\frac{1}{2}}$$

Come avevamo anticipato, $L2$ è la nostra distanza euclidea. $L1$ è invece detta distanza di Manhattan (per via della struttura a pianta rettangolare della città).

Altra interessante metrica è la $L(infinity)$, definita come massimo fra X e Y:

$$L(infinity) : d = max\{X, Y\} \tag{2.4}$$

Vediamo ora una rappresentazione grafica di $L1$ ed $L2$, giusto per chiarire le idee (in rosso rappresentiamo come viene calcolata la distanza):

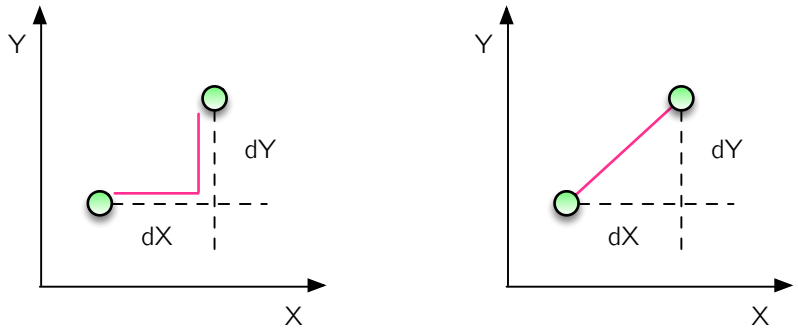


Figura 2.5: Rispettivamente la metrica $L1$ ed una metrica $L2$.

2.2 Le migliori features

Durante il corso del nostro viaggio troveremo altre metriche, ma per ora ci fermiamo qui. Parliamo invece delle caratteristiche salienti che contraddistinguono una immagine: le avevamo già definite *features*. Gli aspetti fondamentali di una immagine sono sostanzialmente:

- Istogramma dei colori.
- Texture.
- Bordi.
- Forme.
- Oggetti.
- Oggetti (in senso semantico).

Sono parecchie: quali sono le più interessanti? Ovvero, quali feature vogliamo includere nel nostro vettore delle features (il surrogato di ogni immagine)?
Per poter rispondere alla domanda dobbiamo introdurre una definizione: una buona feature è una feature che sia *significativa* e che ci permetta di *differenziare* il più possibile un oggetto da un altro. Abbiamo quindi due concetti:

- Feature significativa: è una feature che identifica eventi rari. Ad esempio un'eclisse può caratterizzare una giornata molto più di quanto un'alba possa fare.
- Feature differenziante: è una caratterisca pecularie, che quindi distingue molto un oggetto da un altro.

Se ad esempio dobbiamo caratterizzare un umano, dire che respira è sicuramente una feature poco rilevante (ogni umano respira). Un altro parametro importante da considerare è quanto la feature possa interessare l'utente: se molte query richiedono un'analisi su una certa caratteristica di una foto allora potrebbe essere saggio introdurre una feature che rappresenti quella caratteristica.
Studiamo ora nel dettaglio le due caratteristiche di significatività e di differenziazione che una feature può possedere.

2.2.1 Significatività: l'entropia

In teoria dell'informazione si dice che un evento è più significante se porta più informazioni, inoltre un evento che ha alta probabilità di accadere porta con sé poche informazioni (ricordiamo l'esempio dell'eclisse). Esiste dunque una correlazione secondo la quale alta frequenza corrisponde a poca informazione e a bassa frequenza corrisponde molta informazione.
La misura adatta per studiare questo fattore è chiamata *entropia* ed è definita come segue:

$$H(X) = \sum_{x \in \mathbb{A}_X} P(x) \log \left(\frac{1}{P(x)} \right) \tag{2.5}$$

Quindi se ad esempio abbiamo due eventi equiprobabili $\mathbb{A}_{\mathbb{X}} = \{a, b\}$ con $P(a) = 0.5$ e $P(b) = 0.5$ avremo che l'entropia vale:

$$H(X) = P(a) \log \left(\frac{1}{P(a)} \right) + P(b) \log \left(\frac{1}{P(b)} \right) = 0.15 + 0.15 = 0.30$$

Abbiamo quindi una entropia $H > 0$, questo indica la presenza di informazione. I due eventi infatti sono equiprobabili, quindi nessuno dei due accade ''quasi sempre'', c'è incertezza.
Ma se noi avessimo $P(a) = 1$ e $P(b) = 0$ allora otterremmo:

$$H(X) = P(a) \log \left(\frac{1}{P(a)} \right) + P(b) \log \left(\frac{1}{P(b)} \right) = 0 + 0 = 0$$

Quando $H = 0$ sappiamo che non c'è nessuna informazione (infatti sappiamo già a priori che accadrà sempre l'evento a).

2.2.2 La differenziazione

Per quanto riguarda la differenziazione, diamo una definizione intuitiva grazie a questo disegno:

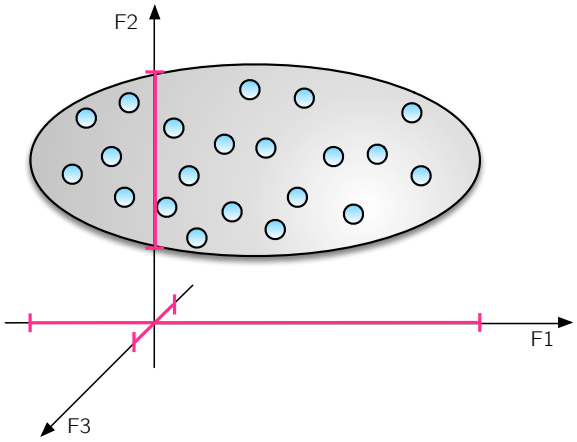


Figura 2.6: Una serie di feature vectors con tre componenti.

Quale delle tre feature (cioè delle tre componenti) è più differenziante? È sufficiente considerare una certa porzione ξ della feature per poter notare un fatto interessante:

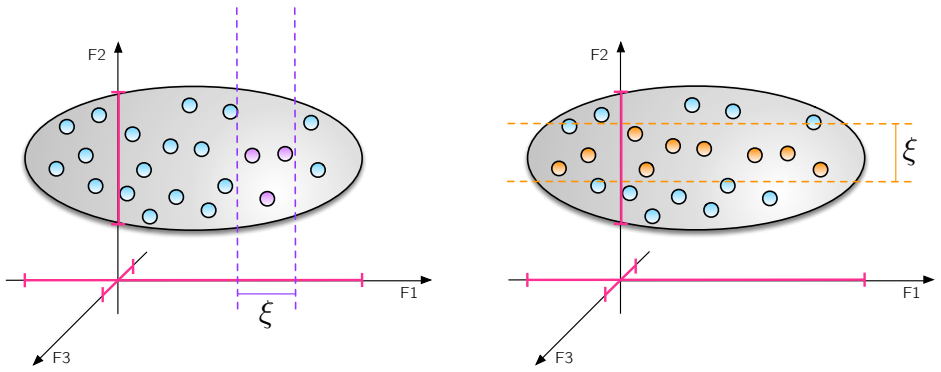


Figura 2.7: Consideriamo una porzione ξ per $F1$ ed $F2$.

Come vediamo, nella feature $F1$ in una porzione ξ otteniamo un numero minore di elementi rispetto a quanto accade in $F2$. Questo significa che $F1$ è *più differenziante*, ovvero $F1$ deve variare di più rispetto a quanto debba variare $F2$ per poter considerare due elementi ''simili''.

2.3 Ridurre il numero di features

Mettiamoci ora nella situazione in cui abbiamo un numero limitato di features possibili: quali rimuovere?

2.3.1 Perché un numero limitato di features

La scelta di features banali non è soltanto inutile, è addirittura *controproducente*, poiché ogni feature poco interessante''appiattirà'' le differenze effettive che sono invece rilevabili da altre features. Problema non da meno, se abbiamo molte feature sarà sempre più difficile trovare oggetti simili (se consideriamo ad esempio le persone come: colore dei capelli, colore delle scarpe, nome dei genitori, età dei genitori, ecc. sarà sempre più difficile trovarne due simili): al limite avremo tutti oggetti distinti e non simili fra loro. Ulteriore fattore da considersi è il costo di mantenimento degli indici: mantenere un indice ha un costo e potrebbe superare quello della ricerca sequenziale se vi sono troppe features. Nel caso di indici si può notare da test empirici che il numero di features suggeribile è al massimo quindici. Approfondiremo in seguito tutte queste problematiche.

2.3.2 Primo approccio: principle component analysis

Abbiamo già detto che una feature ''non buona'' è una feature che non differenzia e ha bassa entropia. Ma non è sempre così facile capire quali sono le features che differenziano di più, o meglio, potremmo non averne una soltanto. Consideriamo infatti questo caso:

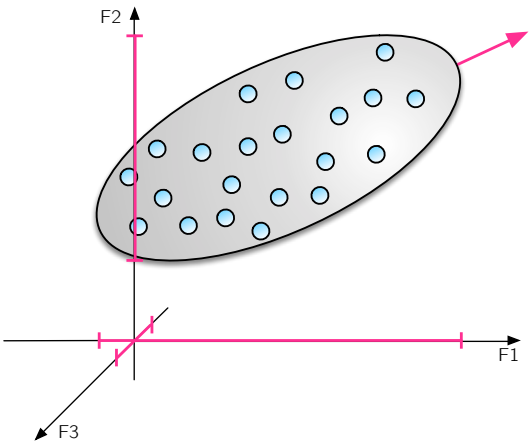


Figura 2.8: Nessuna feature è intuitivamente più differenziante.

Come vediamo la direzione di differenziazione è costituita da $F1$ e da $F2$ insieme, non da uno dei due soltanto. Se fossimo costretti ad eliminare una feature, dunque, quale sceglieremmo (supponiamo che F3 sia molto differenziante, anche non è rappresentata nel disegno)?
Potremmo trasformare lo spazio generando una nuova feature $F4$ ottenuta da $F1$ ed $F2$. Un po' come usare il colore verde per rappresentare sia il giallo che il blu (che lo compongono). La direzione di $F4$ è detta *direzione della massima componente*, che è quella che ha massima *variabilità* (cioè in cui i dati variano maggiormente fra loro).
Tutto il processo che ci porta da uno spazio (quindi un insieme di feature) ad un altro con un numero minore di componenti (cioè di features) perdendo la quantità minima di informazione è detto *principle component analysis* o anche *Karhunene-Loeve Transform*: una trasformazione lineare che scorrela in maniera ottimale gli input.

Eigendecomposition

Abbiamo detto che la direzione dei nuovi vettori segue la componente con massima *variabilità* (cioè in cui i dati variano maggiormente fra loro). Proprio questo concetto ci fa capire che potremmo sfruttare la *matrice di covarianza* per sintetizzare le informazioni relative alla variabilità. Poi, lavorando sulla matrice di covarianza, potremmo estrarne gli autovettori per costruire un nuovo spazio (usando però meno autovalori rispetto a quante fossero le feature prima). Saremo guidati nella scelta di quali autovettori non considerare grazie all'autovalore ad essi associati: più sarà piccolo l'autovalore, minore sarà il contributo informativo portato dalla dimensione (l'autovalore) ad esso associato.

La matrice di covarianza. Iniziamo con l'introdurre la matrice di covarianza. La matrice di covarianza è una matrice A di dimensione $n \times n$ (dove n è il numero di features considerate) e per cui per ogni elemento $a_{i,j}$ è calcolato come segue:

$$a_{i,j} = cov(i,j) = \frac{1}{n} \cdot \sum_{k} (k[i] - \mu_i)(k[j] - \mu_j)$$

Come vediamo all'esterno abbiamo diviso per il numero delle features, poiché stiamo facendo una media aritmetica. A numeratore infatti abbiamo una somma, che, per ogni oggetto k-esimo nel database calcola il valore di quell'oggetto per la feature i-esima (con notazione $k[i]$) e della feature j-esima (quindi $k[j]$) e a questi sottrae rispettivamente il valore medio (quindi calcolato su tutto il database) sulla feature i-esima (μ_i) e quello sulla feature j-esima (μ_j).

Ogni elemento quindi calcola quanto è correlata la variazione della feature i-esima rispetto a quella j-esima (e lo fa sfruttando lo discostamento della media di ogni valore assunto sulle due feature dagli oggetti del database).

Gli autovettori della matrice di covarianza. Ricordiamo che un autovettore di una matrice A è un vettore $\vec{v}$ tale per cui vale la formula:

$$A\vec{v} = \vec{v}\lambda$$

dove λ è l'autovalore associato all'autovettore $\vec{v}$.

Gli autovettori godono inoltre di diverse proprietà, fra le quali alcune molto interessanti per la nostra applicazione. Infatti, data una matrice quadrata A di dimensione $n \times n$, sei suoi autovalori $\lambda_1, ..., \lambda_k$ (con ovviamente $k \leq n$) sono distinti (cioè non coincidenti) allora i suoi autovettori $\vec{v}_1, ..., \vec{v}_k$ sono linearmente indipendenti (ovvero nessuno di loro è combinazione lineare di uno o più degli altri).
Ricordiamo che due vettori $\vec{v}_k$ e $\vec{v}_q$ sono linearmente indipendenti se combinandoli linearmente l'unico modo di ottenere il vettore vuoto è assegnarvi coefficienti a_k e a_q pari a 0 (cioè tutti i coefficienti assegnati ai vettori devono essere nulli). Ovvero:

$$\vec{0} = a_k\vec{v}_k + a_q\vec{v}_q$$

è vero solo se $a_k = a_q = 0$ (se infatti non fosse vero allora potremmo scrivere $a_k\vec{v}_k = -a_q\vec{v}_q$ e quindi i due vettori sarebbero combinazione lineare l'uno dell'altro e più non linearmente indipendenti).

Tornando al nostro problema, se noi calcoliamo gli autovettori della matrice di covarianza A e poi vediamo quanti di questi sono distinti (diciamo ve ne siano k), potremo trasformare il nostro sistema di riferimento in uno basato su quegli autovettori e useremmo ''solo'' k dimensioni invece di n. Evidentemente però se le feature di partenza sono già scorrelate fra loro otterremo $k = n$ e quindi non potremmo eliminare alcuna dimensione.

L'esempio del parallelepipedo. Prima di proseguire con il nostro ragionamento facciamo un piccolo esempio. Abbiamo un parallelepipedo che è descritto da quattro feature: altezza, lunghezza, profondità e volume. Evidentemente il volume è una misura ridondante (poiché dipendente dalle altre tre) e quindi con la nostra analisi sulla matrice di covarianza troveremmo al massimo 3 autovalori distinti (al massimo poiché non possiamo escludere a priori ulteriori dipendenze fra altezza lunghezza e profondità). A questo punto potremmo usare solamente $k = 3$ dimensioni (invece di 4) per costruire il nostro spazio vettoriale e *non avremmo perso alcuna informazione*: ciò che abbiamo tolto era ridondanza a tutti gli effetti.

Eigendecomposition. Continuiamo la nostra discussione. Per ora sappiamo che se siamo fortunati e vi sono feature dipendenti da altre siamo in grado di eliminare la ridondanza (quindi abbiamo un approccio non-lossy). Ma è evidente che potremmo desiderare di ridurre comunque il numero delle feature essendo disposti a perdere un po' di informazione. Ovviamente gradiremmo che questa informazione perduta fosse *minima*. L'algoritmo di eigendecomposition, fortunatamente, ci viene incontro. Infatti la matrice A viene scomposta in tre matrici:

$$A = Q \cdot \Lambda \cdot Q^T =$$

$$\begin{pmatrix} q_{1,1} & \cdots & q_{1,k} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ q_{n,1} & \cdots & q_{n,k} \end{pmatrix} \cdot \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \lambda_k \end{pmatrix} \cdot \begin{pmatrix} q_{1,1} & \cdots & \cdots & \cdots & q_{1,n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ q_{k,1} & \cdots & \cdots & \cdots & q_{k,n} \end{pmatrix}$$

La prima matrice è Q, di dimensione $n \times k$. Sulle righe abbiamo quindi le vecchie features mentre sulle colonne abbiamo i k autovettori. Quindi ogni colonna contiene i coefficienti da assegnare alle vecchie feature in modo da ottenere l'autovettore corrispondente (o in altre parole ogni riga indica la composizione lineare della feature nei termini di tutti gli autovettori della matrice). L'insieme dei k autovettori (le colonne) ci darà quindi la nuova base sulla quale fondare il nostro database. Si noti che ogni colonna di questa matrice è *ortonormale*, ovvero è ortogonale (indipendente) e ha norma$_2$ pari ad 1.
La matrice Λ (di dimensione $k \times k$) è una matrice assai particolare poiché è costituita da tutti zero tranne sulla diagonale dove abbiamo i k autovalori *ordinati in maniera crescente* lungo di essa. Quindi, stando alla notazione adottata prima, abbiamo che $\lambda_1 > \lambda_2 > ... > \lambda_k$. Si noti che gli autovettori della matrice di covarianza sono sempre reali.
La terza matrice è la trasposta di Q.

Esiste un fatto che ci permette di sfruttare le proprietà della matrice Λ per risolvere il problema che ci eravamo posti: infatti vale sempre che quando una matrice A descrive la variazione di dati (come è nel nostro caso con la matrice di covarianza) allora un certo autovalore λ_i indicherà la quantità di contributo che l'autovettore $\vec{v}_i$ ad esso associato porta all'interno del sistema. Ovvero: ad autovalori piccoli corrispondono autovettori che sintetizzano features che hanno poca variabilità e che quindi *possiamo sacrificare*. Perciò se dobbiamo scegliere quali dei k autovettori non includere nella base, opteremo per quelli con autovalori più piccoli.

Riassumendo. In definitiva quando vogliamo ridurre il numero di feature calcoliamo la matrice di covarianza e successivamente la eigendecomposition: se siamo fortunati avremo già un k sufficientemente basso e quindi potremo costruire direttamente il nuovo spazio usando tutti i k autovettori (non c'è perdita di informazione), altrimenti saremo costretti a rimuovere ulteriori autovettori (scegliendo quelli ''a fondo'' della matrice Q poiché sono quelli con autovalore minore) e quindi avremo una perdita di informazione (seppur minimizzata dalla scelta astuta di quali autovettori non considerare).

Quanti autovalori dovremmo mantenere?

Abbiamo appena capito come ridurre a piacere il nostro spazio avendo sempre la garanzia di eliminare ''il meno peggio''. Ma qual è il numero di autovettori da mantenere ottimale? Ci sono diverse possibili filosofie:

- *Autovalore medio*: si mantengono solamente quegli autovettori il cui autovalore è maggiore o uguale alla media degli autovalori di tutta la matrice.
- *Kaiser rule*: si mantengono solamente quegli autovettori il cui autovalore è maggior o uguale ad 1.
- *Analisi parallela*: si fa una analisi più raffinata. Si genera infatti una matrice di covarianza casuale e se ne calcolano gli autovettori. Dopodiché si calcola una funzione cumulativa dei vari autovalori cioè $f(i) = \sum_{k=1}^{i} \lambda_i$ e la si plotta. Si procede allo stesso modo con la matrice di covarianza originale e poi si considera l'intersezione fra le due funzioni.
- *Scree test*: si considera la successione degli autovalori e si cerca un punto dal quale la variazione fra un autovalore ed il suo precedente è più ampia rispetto a quanto accadeva prima (ad esempio in una successione 0.9, 0.8, 0.7, 0.3, ... non considereremo gli autovettori associati ad autovalori pari a 0.3 o minore).
- *Variance explained*: si mantiene un numero di dimensioni tali per cui il 95% della varianza iniziale è rappresentato.

2.3.3 Secondo approccio: compattezza del database

Un altro approccio, decisamente più semplice rispetto allo studio della matrice di covarianza, sfrutta il concetto secondo il quale un database compatto è un database *meno interessante* rispetto a uno che non lo è altrettanto. La *compattezza* del database è definita come segue:

$$comp(D) = \sum_{i \neq j} similarity(o_i, o_j)$$

Calcolata per ogni oggetto i e j del database. Molto semplicemente stiamo calcolando la somma delle similarità di tutte le coppie di oggetti presenti nel database.
L'idea che un database compatto sia meno interessante risulta evidente se consideriamo questa figura:

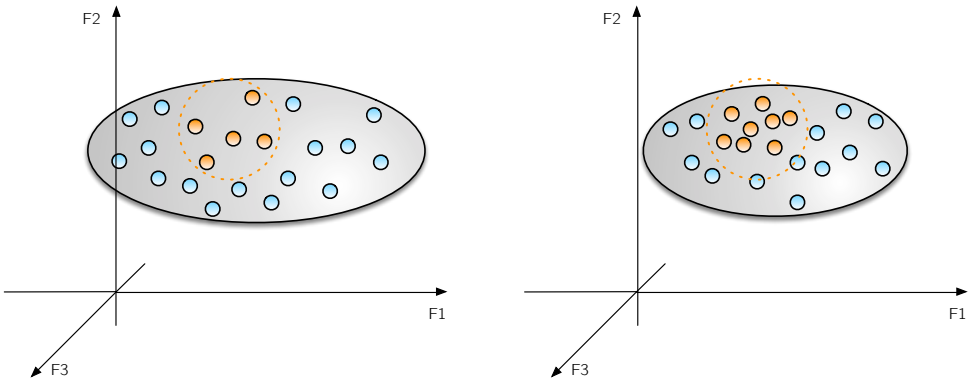


Figura 2.9: Un database compatto avvicina di più i suoi oggetti.

È quindi evidente che il concetto di compattezza sia il duale di quello di differenziazione.

Dovendo quindi scegliere quale (o quali, se iteriamo il processo) feature rimuovere potremmo provare, per ogni feature, a rimuoverla e a calcolare la compattezza del database. La feature la cui eliminazione porterà a un minor aumento di compatezza sarà quella scelta per essere esclusa dallo spazio.

2.3.4 Trasformazioni: miss vs false hit

Abbiamo parlato prima del fatto che le trasformazioni che effettuiamo sfruttando le informazioni forniteci dall'eigendecomposition possano essere eventualmente lossy. Questo fatto ci fornisce l'occasione di parlare della *preservazione della distanza*: quando rimuoviamo una feature che non era composizione lineare delle altre rimuoviamo informazioni, e dal punto di vista geometrico proiettiamo i nostri vettori su un altro piano. Questa proiezione fa sì che le distanze relative fra i vettori *vengano modificate*. Lo stesso discorso vale per gli angoli.
È evidente che questo fatto non sia molto gradito, dato che le nostre misure di similarità si basano sulla distanza (euclidea, ad esempio) oppure sugli angoli (cosine similarity).
Dovendo però comunque rimuovere delle informazioni (e quindi deformare le distanze e gli angoli), qual è il miglior modo di farlo? È meglio allontanare gli oggetti fra loro oppure avvicinarli? Nel primo caso predisporremmo il sistema a

dei miss che non avrebbero duvuto esserci mentre nel secondo agevoleremmo l'occorrerre di false hits.

La risposta è: *meglio i false hit*. Quando accade un false hit sostanzialmente consideriamo interessante un oggetto che altrimenti sarebbe stato ignorato. Ma non c'è reale perdita di informazione, infatti, durante il postprocessing (avremo quindi abbiamo estratto un numero decisamente ridotto di oggetti) possiamo facilmente sfruttare la feature che avevamo eliminato per verificare la reale similarità fra la query e l'oggetto recuperato (ma questa operazione la compiamo su un numero molto più piccolo di oggetti).
Nel caso di miss, invece, perdiamo a tutti gli effetti l'informazione: i due oggetti saranno diversi per sempre.

Capitolo 3

Le immagini

3.1 Modificare il dominio

Le tecniche di compressione che abbiamo studiato fino a questo momento sono molto generali e sono applicabili a molti domini diversi.
Concentriamoci ora sul dominio delle immagini e vediamo quali tecniche sono disponibili per poter comprimere i nostri dati e quindi poterci lavorare con maggiore facilità.

3.1.1 Perché ridurre i costi

Cerchiamo di capire il motivo per cui si renda *necessario* ridurre le dimensioni dei nostri dati e non possiamo trattarli in formato raw.

Image processing vs image analysis

Distinguiamo innanzi tutto i due importanti concetti di *image processing* e di *image analysis*. Nel primo caso il risultato della computazione è ancora una immagine, ma modificata (ad esempio con i bordi più accentuati) rispetto a quella di partenza. Le operazioni di analisi invece constano spesso nella produzione di un risultato sintetizzato che ''riassume'' una certa caratteristica dell'immagine, ma si tratta di dati numerici e confrontabili.
Fra le operazioni di processing troviamo:

- Rotazione.
- Traslazione.
- Cut&Paste.
- Ridimensionamento.
- Affinamento dei bordi.

- Sfocamento.

Le operazioni di analysis sono quelle sulle quali si basano i processi di estrazioni delle feature:

- Istogramma dei colori.
- Texture.
- Bordi.
- Forme.
- Semantica associata alle forme.

Spesso queste operazioni richiedono un impiego *convoluto* di filtri, ovvero non si analizzano i singoli pixel bensì una zona di pixel (in modo da capire la correlazione fra un pixel e i suoi vicini). Questo fa sì che se abbiamo una immagine di dimensione $N \times M$ e esaminiamo una finestra di pixel di dimensione $w \times w$ per la nostra analisi avremo un costo molto alto: $N \times M \times w \times w$.

Lo storage

Se abbiamo una immagine di dimensione 1920x1080 ed ogni bit è codificato in RGB (lo studieremo nel dettaglio a breve) allora per ogni pixel abbiamo 3 byte, per un totale di 6MB: una dimensione proibitiva se pensiamo di avere migliaia di immagini del genere da dover trattare simultaneamente.

Soluzioni possibili per la compressione

I due problemi appena esposti ci fanno capire che si renda necessario comprimere i nostri dati in qualche modo. Vi sono due approcci generali:

- Modificare la codifica: molto spesso le codifiche contengono ridondanza; l'intenzione è quella di rimuoverla senza perdere informazione (tecnica non lossy).
- Effettuare una compessione lossy: le immagini possono contenere dettagli che l'occhio umano non è in grado di riconoscere. Si possono dunque usare tecniche ci permettono di spostarci di dominio (passiamo dal dominiio spaziale a quello della frequenza) così da poter identificare le zone dell'immagini più dettagliate e che quindi possono essere sacrificate durante la compressione (tecnica DCT). Alternativamente possiamo approssimare i colori sfruttando un numero minore di bit per codificarli (tecnica del *median cut*).

Si noti che la compressione risulta utile non solo per motivi di spazio, ma diminuendo le fini differenze fra le immagini aumenterà la similarità e quindi sarà possibile fare un retrieval più corposo e fedele (relativamente a ciò che vede l'uomo).

3.1.2 Alcune tecniche non-lossy

Accenniamo ad alcune tecniche non-lossy. Gli esempi che tratteremo sono sul testo ma è facile riprodurre la tecnica sul dominio delle immagini.

Huffman coding (statico/dinamico). Si effettua analisi del testo e si determinano i caratteri più frequenti. Dopodiché si usano meno bit per codificare quei caratteri rispetto agli altri. Per esempio potremmo codificare gli spazi con due byte soltanto invece che con tre. Questa tecnica necessita di una distribuzione non uniforme dei valori per poter funzionare (ma come ha dimostrato Zipf questo non è un problema nel dominio del testo).

Aritmetic coding. Funziona in modo del tutto simile ai codici di Huffman con la differenza che invece di sostituire un valore per ogni carattere, l'intero messaggio viene codificato in un singolo numero n (con $0.0 \leq n \leq 1.0$).

LSW coding. Indici di diversa lunghezza vengono associati a *porzioni* di testo. In pratica è una versione applicata su porzioni su testo dell'Huffman coding. Abbiamo un vocabolario (inizialmente contiene le codifiche ASCII di tutti i singoli caratteri). Dopodiché, tramite un processo iterativo, si esamina il testo. Consideriamo ad esempio la parola `storia`. La prima lettera presa in considerazione è la `s`, che è già presente nel vocabolario, dunque avanziamo senza fare nulla. A questo punto consideriamo la stringa `st`: non è presente nel nostro vocabolario e quindi la aggiungendo associandoci il primo indice libero (256, dato che 255 erano i primi caratteri in codice ASCII). Si procede in questo modo su tutto il testo, ottenendo quindi una codifica numerica per porzioni di testo (speranzosamente ripetute).
Il problema di questa tecnica è che ogni immagine ha una compressione propria (un suo vocabolario).

3.1.3 Il median cut (lossy)

Il median cut è una clusterizzazione ad hoc basata sulle tonalità di colore presenti nel database: per esempio se abbiamo una base di data popolata da foto di mare avremo una grossa predominanza del blu e delle sue diverse sfumature.

Il concetto di clusterizzazione è molto semplice: invece di avere una scala di 255 rossi, 255 verdi e 255 blu si sceglie di averne un numero minore e quindi la rappresentazione passa dai 3 byte di partenza ad un numero arbitrariamente più piccolo di bit (al minimo uno per colore). I cluster sono quindi insiemi che raggruppano colori inizialmente diversi secondo un principio di somiglianza (ad esempio potremmo avere 4 cluster per il rosso: [0-64], [65-128], [129-192], [193-256]), rendendoli ai fini della nostra analisi del tutto uguali (con un valore

scelto arbitrariamente nell'intervallo rappresentato, tipicamente, la mediana). Nell'esempio di prima quindi, tutti i rossi da 0-64 saranno rappresentati da una tonalità pari a 32.
Può essere però interessante essere in grado di distribuire in maniera non uniforme i bit tramite i quali rappresentiamo i cluster, ovvero avere ''sensibilità'' diverse sui tre colori RGB.
Il median cut è quindi un sistema più sofisticato del clustering normale, poiché gli intervalli entro i quali i colori vengono categorizzati non sono uniformi né prestabiliti, ma sono data-driven (quindi si deve avere un database sufficientemente statico).

Esecuzione del median cut

Vediamo ora una esecuzione dell'algoritmo di median cut. Ricordiamo che questo algoritmo è applicato per *generare il surrogato* di una immagine: al momento del retrieval verrà restituita l'immagine originale. Inoltre si rammenti che la codifica che andremo ad imporre (in termini di numeri di bit) dovrà essere comune a tutte le immagini del database (anche a quelle fornite come query, quindi).
Diciamo di voler passare da una codifica RGB (8 bit ogni colore per un totale di 24 bit) ad una codifica di 6 bit totali (due per colore quindi quattro tonalità ciascuno). Il numero di bit a cui vogliamo passare è il parametro dell'algoritmo, quindi si può applicare per valori a piacere. Si procede come segue:

Consideriamo il colore **rosso**:

1. Si calcola la mediana sui valori del rosso (m_R) del database. In questo modo abbiamo che per ogni pixel di ogni immagine la probabilità di avere una tonalità di rosso maggiore a m_R è uguale quella di averla minore di m_R. Supponiamo che il valore di m_R trovato sia pari a 140.

2. Se dunque una immagine avrà valore di rosso $R > m_R$ avrà il **primo** bit impostato ad 1, altrimenti sarà impostato a 0.

Consideriamo poi il colore **verde**:

1. Si calcola la mediana sui valori del verde del database: ($m_G = 200$).

2. Se dunque una immagine avrà valore di verde $G > m_G$ avrà il **secondo** bit impostato ad 1, altrimenti sarà impostato a 0.

Consideriamo poi il colore **blu**:

1. Si calcola la mediana sui valori del blu del database: ($m_B = 160$).

2. Se dunque una immagine avrà valore di verde $B > m_B$ avrà il **terzo** bit impostato ad 1, altrimenti sarà impostato a 0.

A questo punto abbiamo definito tre bit (siamo a metà strada) e in totale 8 combinazioni entro le quali piazzare le nostre immagini. Si procede assegnando gli ulteriori tre bit nello stesso modo, ma questa volta calcoleremo le mediane sui singoli intervalli!
Quindi per definire il **quarto** bit:

- Consideriamo l'intervallo di rosso $[0 - 140]$ e ne calcoliamo la mediana $m_{R(2)'}$, diciamo valere 52: qualunque immagine che abbia una tonalità di $R > m_{R(2)'}$ avrà combinazione di bit `0**1@@`, altrimenti avrà combinazione di bit `0**0@@`.

- Consideriamo l'intervallo di rosso $[141 - 256]$ e ne calcoliamo la mediana $m_{R(2)''}$, diciamo valere 110: qualunque immagine che abbia una tonalità di $R > m_{R(2)''}$ avrà combinazione di bit `1**1@@`, altrimenti avrà combinazione di bit `1**0@@`.

Procediamo con il **quinto** bit:

- Consideriamo l'intervallo di verde $[0-200]$ e troviamo $m_{G(2)'} = 70$, dunque una immagine per cui $G > m_{G(2)'}$ avrà combinazione di bit `*0**1@`, altrimenti avrà combinazione di bit `*0**0@`.

- Consideriamo l'intervallo di verde $[201 - 256]$ e troviamo $m_{G(2)''} = 220$, dunque una immagine per cui $G > m_{G(2)''}$ avrà combinazione di bit `*1**1@`, altrimenti avrà combinazione di bit `*1**0@`.

Terminiamo con il **sesto** bit:

- Consideriamo l'intervallo di blu $[0 - 160]$ e troviamo $m_{B(2)'} = 100$, dunque una immagine per cui $B > m_{B(2)'}$ avrà combinazione di bit `**0**1`, altrimenti avrà combinazione di bit `**0**0`.

- Consideriamo l'intervallo di blu $[151 - 256]$ e troviamo $m_{B(2)''} = 190$, dunque una immagine per cui $B > m_{B(2)''}$ avrà combinazione di bit `**1**1`, altrimenti avrà combinazione di bit `**1**0`.

NB: con un asterisco intendiamo una qualsiasi configurazione (0 o 1) del bit (sono bit che non concernono il colore trattato) mentre con la chiocciola indichiamo i bit non ancora assegnati.

Insomma abbiamo costruito una sequenza $RGBRGB$ dove ogni bit specifica man mano una definizione più fine delle tonalità di questi tre colori. Nessuno, comunque, ci vieta di comporre le iterazioni in modo differente e quindi se vogliamo un ampio dettaglio su blu potremmo avere una sequenza sbilanciata $RGBB$: il blu ha due bit di definizione mentre il rosso e il verde soltanto uno.

Sfruttando i dati di prima illustriamo il risultato della conversione di due pixel:

- 201 075 013 viene convertito in 100100.
- 027 141 032 viene convertito in 000010.

Ai fini del calcolo della similarità i valori di colore impostati per ogni cluster (o sotto cluster) saranno la mediana dell'intervallo in cui ci si trova.

Tutti i valori delle mediane devono essere memorizzati per poter convertire ogni immagine fornita come query nel vocabolario del database, così da renderla confrontabile con le altre immagini.

3.1.4 La Discrete Cosine Transform (DCT)

La DCT è, di per sé, una trasformazione non-lossy ed invertibile. L'uso che ne facciamo noi porterà però ad una perdita di informazioni (ed è lo stesso che se ne fa nella compressione JPEG).
Come abbiamo già accennato scopo della DCT è la trasformazione del nostro dominio da un dominio delle ampiezze (cioè ogni pixel rappresenta l'intensità di colore) ad un dominio delle frequenze. Se visualizziamo dunque la nostra immagine come un vettore costituito dalla concatenazione di ogni riga di pixel e vi applichiamo la DCT, otteniamo un nuovo vettore dove a sinistra abbiamo le basse frequenze e a destra le alte frequenze.

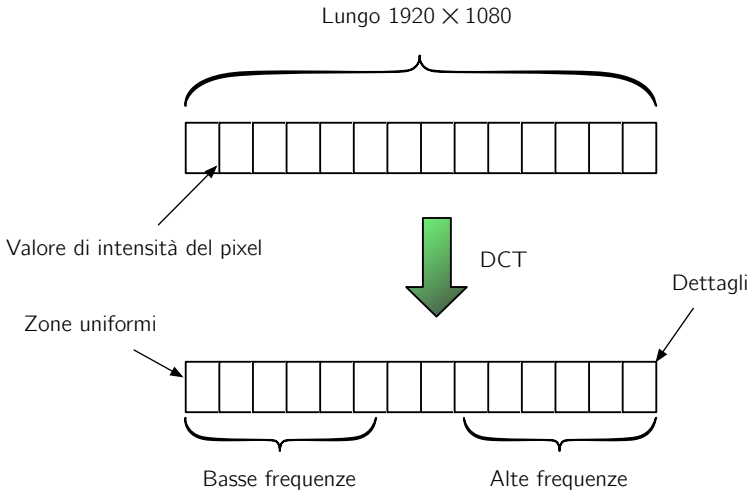


Figura 3.1: Applicazione della DCT per una immagine 1920×1080.

Come vediamo dalla figura questo tipo di trasformazione è molto interessante perché ci permette di individuare le zone di ampio dettaglio dell'immagine: sono quelle con frequenza più alta. Possiamo dunque lavorare su queste per ridurre la dimensione dell'immagine.

La DCT è una trasformazione molto buona poiché preserva la distanze euclidee e non è sensibile a shift dell'immagine (le frequenze son sempre quelle).

Esempio di applicazione della DCT

Prendiamo la famosa immagine di *Lena*, usata in moltissimi benchmark. Di questa immagine estriamo lo spettro di grigio (cioè la convertiamo in bianco e nero) poiché ai nostri scopi non interessa il colore di per sé ma la sua intensità.

Figura 3.2: Lena Söderberg, playmate del novembre 1972.

Poi definiamo un dominio delle frequenze dove mapperemo quelle di Lena:

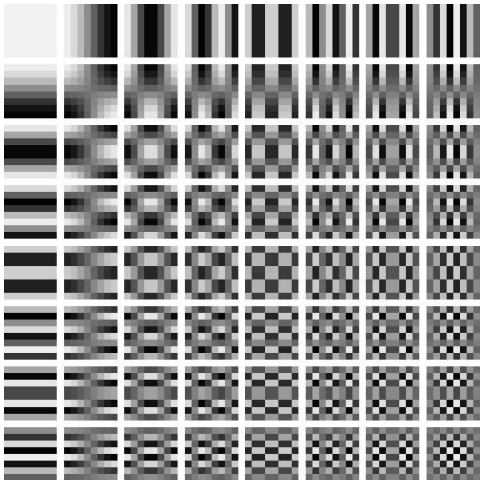


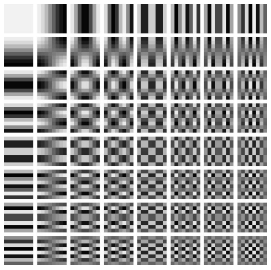
Figura 3.3: Dominio delle frequenze.

Abbiamo scelto un dominio 8×8. Vediamo verticalmente abbiamo un aumento delle frequenze sulle y mentre orizzontalmente abbiamo un aumento delle frequenze su x. È dunque evidente che la zona di maggiore dettaglio sia quella in basso a destra, dove sia la x che la y sono ad alta frequenza (e infatti abbiamo una scacchiera perfetta: la variazione è altissima).

Uso della DCT. Procediamo con l'applicazione della DCT, sfruttando la formula (apposita per il dominio):

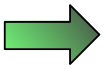
$$F(u,v) = \frac{C(u)C(v)}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} = cos\left(\frac{(2i+1)u\pi}{16}\right) + cos\left(\frac{(2j+1)v\pi}{16}\right) f(i,j)$$

dove il termine $f(i,j)$ proviene dal dominio dei pixel (l'immagine di Lena). Otteniamo quindi:



```
 70  70 100  70  87  87 150 187
 85 100  96  79  87 154  87 113
100  85 116  79  70  87  86 196
136  69  87 200  79  71 117  96
161  70  87 200 103  71  96 113
161 123 147 133 113 113  85 161
146 147 175 100 103 103 163 187
156 146 189  70 113 161 163 197
```

DCT

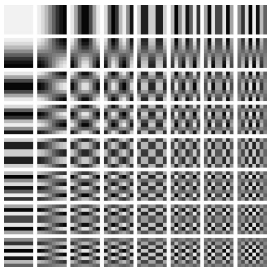


```
-80 -40  89 -73  44  32  53  -3
-135 -59 -26   6  14  -3 -13 -28
 47 -76  66  -3 -108 -78  33  59
 -2  10 -18   0  33  11 -21   1
 -1  -9 -22   8  32  65 -36  -1
  5 -20  28 -46   3  24 -30  24
  6 -20  37 -28  12 -35  33  17
 -5 -23  33 -30  17  -5  -4  20
```

Figura 3.4: Applicazione della DCT.

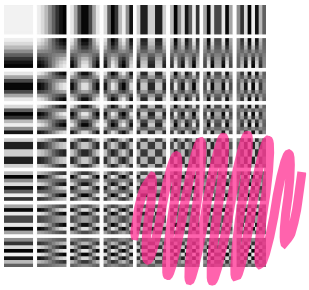
Ricordiamo che fino ad ora non abbiamo perso alcuna informazione: la trasformata è reversibile.

Rouding. Introduciamo ora la parte lossy. Come abbiamo in detto, in basso a destra vi sono i valori rappresentanti il maggior dettaglio. Li impostiamo a zero (a seconda del livello di compressione desiderato potremo regolare la propagazione degli zeri) e otteniamo una nuova matrice nel dominio delle frequenze:



-80	-40	89	-73	44	32	53	-3
-135	-59	-26	6	14	-3	-13	-28
47	-76	66	-3	-108	-78	33	59
-2	10	-18	0	33	11	-21	1
-1	-9	-22	8	32	65	-36	-1
5	-20	28	-46	3	24	-30	24
6	-20	37	-28	12	-35	33	17
-5	-23	33	-30	17	-5	-4	20

Rouding



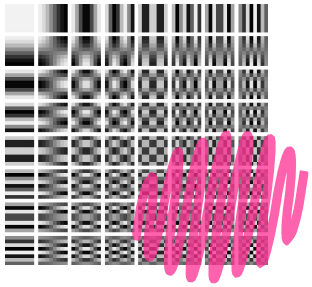
-80	-44	90	-80	48	40	51	0
-132	-60	-28	0	26	0	0	-55
42	-78	64	0	-120	-57	0	56
0	17	-22	0	51	0	0	0
0	0	-37	0	0	109	0	0
0	-35	55	-64	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figura 3.5: Approssimazione tramite rounding.

Tornare al dominio delle intensità. Non resta che tornare al dominio delle intensità per avere la nostra immagine compressa. Usiamo la DCT inversa:

$$\tilde{f}(i,j) = \frac{C(u)C(v)}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} = cos\left(\frac{(2i+1)u\pi}{16}\right) + cos\left(\frac{(2j+1)v\pi}{16}\right) F(u,v)$$

e otteniamo:



-80	-44	90	-80	48	40	51	0
-132	-60	-28	0	26	0	0	-55
42	-78	64	0	-120	-57	0	56
0	17	-22	0	51	0	0	0
0	0	-37	0	0	109	0	0
0	-35	55	-64	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

DCT
Inversa

70	60	106	94	62	103	146	176
85	101	85	75	102	127	93	144
98	99	92	102	74	98	89	167
132	53	111	180	55	70	106	145
173	57	114	207	111	89	84	90
164	123	131	135	133	92	85	162
141	159	169	73	106	101	149	224
150	141	195	79	107	147	210	153

Figura 3.6: Otteniamo l'immagine approssimata.

Una perdita non così grave

Notiamo che questa tecnica, seppur lossy, deforma le distanze in modo non grave. Infatti, le distanze reciproche vengono *dimiunite* e questo creerà eventualmente dei false-hit ma mai dei miss (per maggiori dettagli si riveda la Sezione 2.3.4).

3.2 Studio della feature: istogramma dei colori

Iniziamo adesso lo studio dettagliato ed implementativo delle feature più importanti. La prima feature che introduciamo è l'*istogramma dei colori*, la feature più intuitiva che possiamo pensare. Partiamo da una forma primordiale di istogramma e man mano ne esamineremo le caratteristiche cercando di migliorarlo. Ogni pixel dell'immagine ha un colore, dunque facciamo un listato dei colori presenti nell'immagine e li mettiamo sulle ascisse mentre sulle ordinate mettiamo la quantità di occorrenze di quel colore (quanti pixel sono colorati di quel colore).

3.2.1 Diverse rappresentazioni del colore

Esistono diversi modelli per rappresentare i colori. Avere più modelli è utile poiché alcuni di questi riflettono in maniera più esatta la percezione dell'umano.

Codifica RGB (Red, Green, Blue). Ad ogni pixel sono associati tre byte, ognuno dei quali rappresenta valori da 0 a 255 che indicano la quantità di Rosso, Verde o Blu all'interno del pixel.

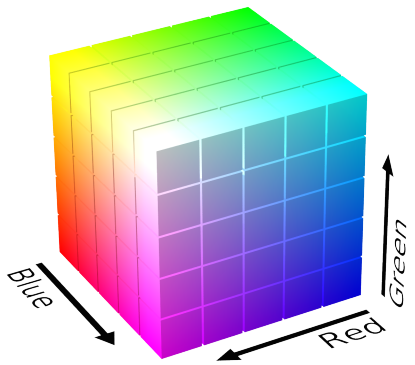


Figura 3.7: Rappresentazione del modello RGB.

Codifica HSV (Hue, Saturation, Value). Abbiamo accennato in passato al fatto che la luminosità possa essere un elemento di disturbo nella ricerca di

similarità fra colori (a seconda dell'illuminazione un colore può apparire molto diverso). Questo tipo di codifica ci permette di separare la tonalità (hue) dalla sua saturazione (quanto è pura, ovvero quanto bianco c'è) e dalla sua dalla luminosità (value). Il valore di tonalità viene assegnato come grado, e ne vediamo la corrispondenza in questa figura:

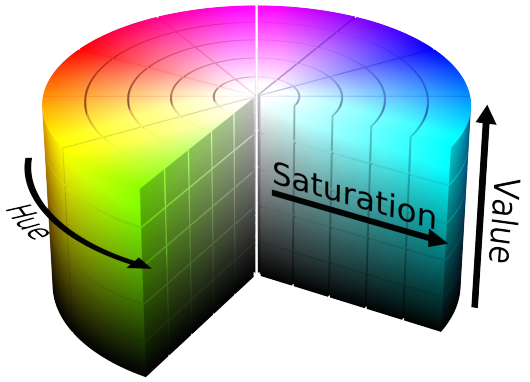


Figura 3.8: Rappresentazione del modello HSV.

Codifica YUV (Luminance, Red-Cyan, Magenta-Green). Questo modello risulta estremamente interessante poiché la componente Y (espressa in scala di grigio) è la componente *più informativa* che vi sia all'interno di una immagine e quindi quando si applicano compressioni (ad esempio il median cut) si prediligerà effettuarle sulle componenti U e V, mantenendo sempre intatta la Y (nel caso del median cut assegneremo più bit a Y). Lo vediamo bene da questa immagine:

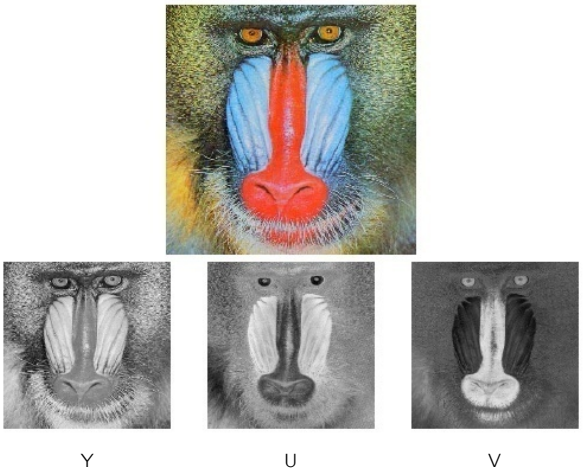


Figura 3.9: La componente Y è la più informativa.

La codifica YUV è *equivalente* a quella RGB in termini di informazioni, infatti esiste una trasformazione lineare con cui passare da una all'altra:

$$Y = 0.299R + 0.587G + 0.144B$$
$$U = 0.492(B - Y)$$
$$V = 0.877(R - Y)$$

Quello che fa la codifica YUV è quindi modificare lo spazio dell'RGB nel seguente modo:

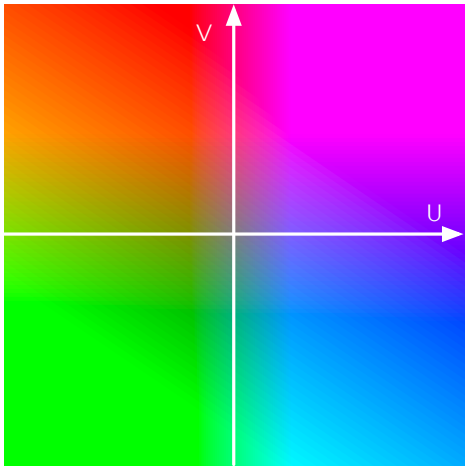


Figura 3.10: Rappresentazione del modello YUV.

Codifica YIQ. Studi empirici hanno dimostrato che l'occhio umano è più sensibile al range dell'arancione-blue rispetto a quello del viola-verde, come vediamo dal nostro solito esempio della scimmia:

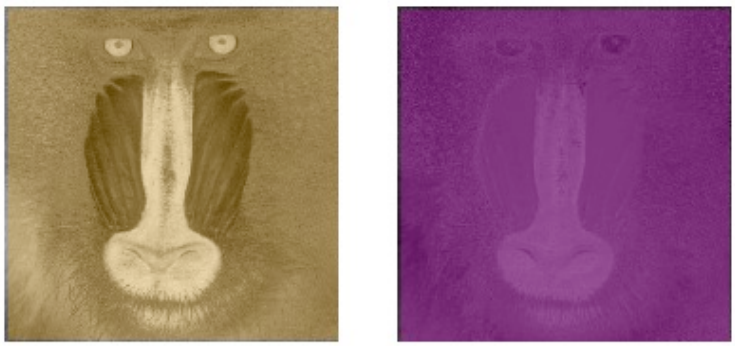


Figura 3.11: La seconda immagine porta pochissima informazione.

Da qui l'idea di un nuovo modello che espliciti la componente viola-verde (in

modo da poterla martoriare con i nostri algoritmi lossy): la codifica YIQ. Il nostro spazio risulta dunque rotato come possiamo vedere da questa immagine:

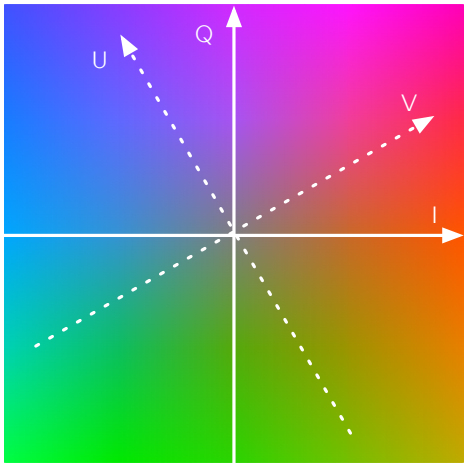


Figura 3.12: Rappresentazione del modello YIQ (rotazione di YUV).

3.2.2 Esempio di istogramma

Abbiamo già descritto, in teoria, come creae un istogramma. Evidentemente però siamo stati poco realistici: è infatti estremamemente probabile che una immagine figuri moltissimi colori diversi. Per questo si scelgono dei *rappresentanti* (dei cluster quindi) entro i quali andare a inserire i nostri colori effettivi. Ecco quindi un esempio realistico di istogramma:

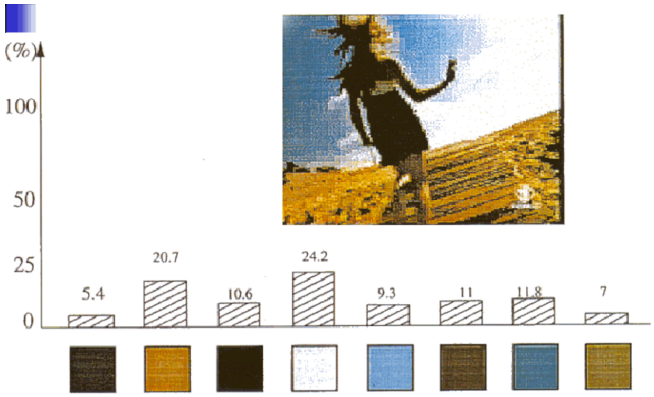


Figura 3.13: Un istogramma dei colori.

3.2.3 Il problema della località

Uno dei problemi più grandi degli istogrammi è il fatto che non abbiano una accezione locale, ma bensì globale. Per capire di cosa stiamo parlando osserviamo questo esempio:

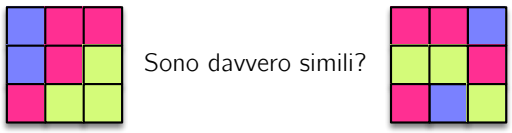


Figura 3.14: Stando all'istogramma dei colori queste due immagini sono identiche.

Capiamo dunque che la *località* è un concetto importante e deve essere preso in considerazione. Il problema è risolvibile effettuando istogrammi su porzioni più piccole dell'immagine e poi facendo il confronto su questi:

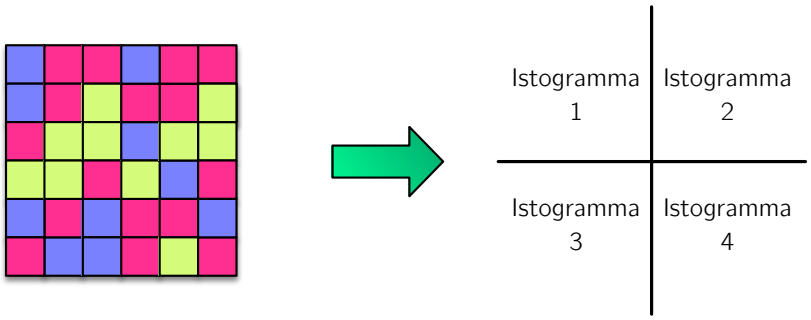


Figura 3.15: I quattro istogrammi descriveranno più fedelemente il colore.

3.2.4 Similarità ed istogrammi: l'associazione fra colori.

Un ulteriore problema (questa volta più difficile da risolvere) è il fatto che la similarità fra colori all'interno di un istogramma non venga considerata. Ad esempio il giallo è simile all'arancio: ma questo fattore non è conteggiato, quindi una immagine tutta blu e una tutta rossa sono distanti tanto quanto una tutta arancio ed una tutta gialla.
Due delle tecniche per confrontare due istogrammi (che non tengono conto dell'associazione fra i colori) sono la distanza euclidea e la intersection similarity. Ricordiamo la formula della prima:

$$\sqrt{(b_1 - b_2)^2 + (g_1 - g_2)^2 + (p_1 - p_2)^2 + (r_1 - r_2)^2 + \ldots}$$

Abbiamo usato le lettere b, g, p e r per indicare il valore dell'istogramma relativo al blue, green, purple e red. Il pedice invece ci dice se la misura appartiene all'istogramma della figura 1 o della figura 2.

La intersection similarity. Il concetto dietro alla intersection similarity è molto semplice: si considera ''quanto c'è in comune'' fra due immagini. Ovviamente il concetto di condivisione è equivalente al minimo fra due valori sull'istogramma (se una immagine ci sono 40 pixel rossi e nell'altra 20, in comune ve ne saranno 20 soltanto). La formula risulta quindi essere:

$$\frac{min(b_1, b_2) + min(g_1, g_2) + min(p_1, p_2) + min(r_1, r2)}{b_2 + g_2 + p_2 + r_2}$$

Nel caso in cui le due immagini abbiano pari dimensioni il denominatore può essere costituito da tutti i valori sull'istogramma di una delle due figure qualsiasi. È evidente che questa similarità sia utilizzabile solamente quando non si ha un istrogramma delle frequenze ma bensì delle occorrenze.

3.2.5 La distanza euclidea completa

Andiamo finalmente a risolvere il problema dell'associazione dei colori parlando della distanza euclidea completa. Definiamo con $\vec{x}$ e $\vec{y}$ due istogrammi sottoforma di vettori, ognuno di lunghezza n. Dopodiché definiamo la distanza euclidea come segue:

$$d^2 = \sum_{i,j=1}^{n} a_{ij}(x_i - y_i)(x_j - y_j)$$

Il fattore a_{ij} è detto *cross talk factor* ed è stabilito a priori: esiste quindi un cross talk factor per ogni coppia possibile di colori. Questo fattore varia da 0 a 1 e ci dice quanto due colori sono correlati (ad esempio il grigio ed il nero avranno fattore molto alto). Questa formula è riconducibile alla distanza euclidea classica; infatti assegniamo un crosstalk in modo che dato un colore l'unico a cui questo è correlato sia se stesso:

$$a_{ij} = \begin{cases} 1 & \text{se } i = j \\ 0 & \text{se } i \neq j \end{cases}$$

avremo una somma euclidea classica.

La distanza euclidea completa è quindi una tecnica ottimale ma è computazionalmente assai costosa: $O(N^2)$. Esiste un modo di sfruttare la distanza euclidea indirettamente, essendo la nostra soluzione ottimale. Prima di introdurre questo fatto, parliamo della distanza quadratica.

3.2.6 Distanza quadratica

Partiamo dall'esatto opposto della soluzione ottima: la soluzione meno costosa e più stupida che si possa considerare. Diciamo di avere un istogramma con

solamente tre colori, R G e B. Il valore corrispondente sarà calcolato come media dei rossi, dei verdi e dei blu presenti nell'immagine, in formule:

$$R_{avg} = \frac{1}{n}\sum_{i=1}^{n} R(p_i)$$
$$G_{avg} = \frac{1}{n}\sum_{i=1}^{n} G(p_i)$$
$$B_{avg} = \frac{1}{n}\sum_{i=1}^{n} B(p_i)$$

dove con $R(p_i)$ si intende il valore associato al rosso nel pixel i-esimo (e analogamente per $G(p_i)$ e $B(p_i)$). A questo punto il nostro vettore di feature non sarà altro che un vettore $\hat{x}$ così definito:

$$\hat{x} = (R_{avg}, G_{avg}, B_{avg})$$

Non resta che definire la distanza quadratica fra due vettori come:

$$\begin{aligned} d^2_{avg}(\hat{x}, \hat{y}) &= (\hat{x} - \hat{y})^T(\hat{x} - \hat{y}) \\ &= (R_{avg\_x} - R_{avg\_y})^2 + (G_{avg\_x} - G_{avg\_y})^2 + (B_{avg\_x} - B_{avg\_y})^2 \end{aligned}$$

Abbiamo inserito una trasposta per poter effettuare la moltiplicazione.

3.2.7 Un ottimo compromesso

Qual è l'utilità della distanza quadratica? Come può avvicinarci al nostro obiettivo di una formula computazionalmente ammissibile ma che funzioni sufficientemente bene? Ci viene in contro una fondamentale disequazione:

$$d^2_{avg}(\hat{x}, \hat{y}) \leq c_n \cdot d^2_{hist}(\hat{x}, \hat{y})$$

dove con d^2_{hist} abbiamo indicato la distanza euclidea completa. Abbiamo quindi che la distanza quadratica è sempre una *sottostima* della distanza euclidea completa (a meno di una costante c_n dipendente dalla dimensione dell'istogramma).
L'utilità di questa disequazione è lampante: calcolare la distanza tramite distanza quadratica (molto veloce) può portarci ad avere false hits ma *mai* miss! Potremo dunque ottenere un sottoinsieme di fotografie simile abbastanza ristretto e a quel punto potremo applicare la distanza euclidea completa per raffinare definitivamente i risultato. Inoltre, siamo in grado di usare delle tolleranze per esprimere diversi livelli di precisione.

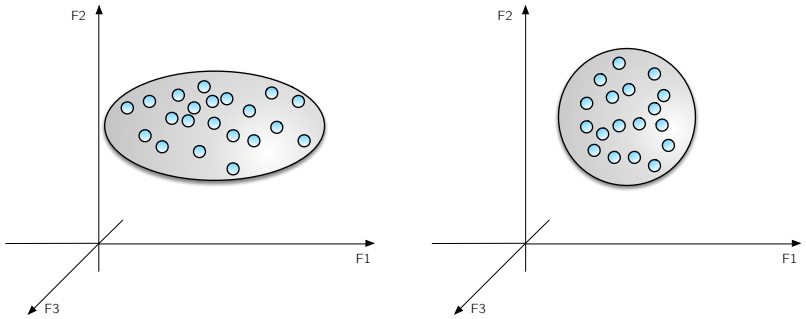


Figura 3.16: Due database con conformazioni diverse.

3.2.8 Distanza di Mahalonobis

La distanza di Mahalonobis risulta utile quando, dato un database, giungono due nuove immagini ma possiamo includerne solo una: quale scegliere? Consideriamo questi due database:
Come vediamo nel primo database vi sono alcuni feature più differenzianti di altre, mentre nel secondo database ogni feature è differenziante quanto le altre. Ora simuliamo l'arrivo dei due dati nei due diversi database:

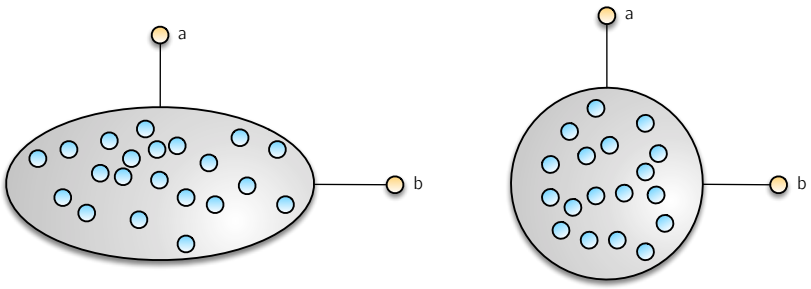


Figura 3.17: Possiamo solo scegliere un elemento fra *a* e *b*.

Supponiamo di aver calcolato la distanza fra i nuovi *a* e *b* e il resto del database e supponiamo la distanza *d* sia risultata paritaria in tutte e quattro le misure. Nel secondo database scegliere *a* o *b* è assolutamente irrilevante, poiché c'è regolarità fra i dati (la scelta di uno dei due porterà in ogni caso ad uno sbilanciamento a favore di $F1$ o di $F2$). Ma nel primo database invece la scelta è rilevante: il punto più papabile sembrerebbe essere *b* poiché *preserva il rapporto fra gli assi*. Quindi a fronte di una distanza *d* uguale, uno dei due dati è più interessante dell'altro. Vorremmo essere in grado di cogliere questo aspetto: ci viene in contro la distanza di Mahalonobis, definita come segue:

$$\Delta_{Mah}(\vec{x}, \vec{y}) = \sqrt{(\vec{x} - \vec{y})^T S^{-1} (\vec{x} - \vec{y})}$$

dove con S^{-1} intendiamo l'inversa della matrice di covarianza, che quindi ci serve per correggere il valore della distanza tenendo conto di quanto discusso sopra. Verrà dunque ridotto l'impatto laddove c'è alta varianza (ovvero il punto che si trova allineato alla massima varianza sarà considerato più simile al database e quindi verrà incluso poiché con valore di Δ_{Mah} maggiore).
Si noti che che se al posto di S usassimo una matrice identità $\mathbb{1}$ ci ricondurremmo alla formula della distanza quadratica (in generale della distanza euclidea, dipende se usiamo array che sono ottenuti come media dei colori oppure no). Se invece usiamo la matrice di similarità (A) otteniamo la distanza euclidea completa (poiché all'interno della matrice di similarità sono mantenuti i coefficienti di cross-talk).

3.3 Studio della feature: le textures

Con le textures si intendono le trame assunte dall'immagine. Siamo interessati a riconoscerle per poter, ad esempio, essere in grado di distinguere il verde di una foglia d'erba (distribuito come linee fine e ripetute) rispetto al verde di una foglia. Al fine di accentuare le linee di demarcazione della texture l'analisi viene effettuata con immagini codificate in YUV (si lavora sulla componente Y).

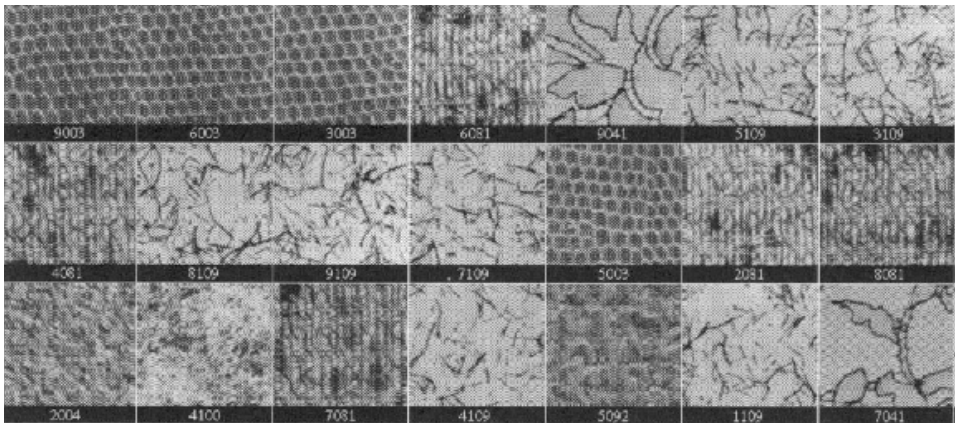


Figura 3.18: Alcune textures.

È evidente che si renda necessaria una analisi *convolutiva*, ovvero che si debbano considerare più pixel contemporaneamente. È infatti essenziale capire la correlazione che v'è fra più pixel adiacenti fra loro. La contestualizzazione dei pixel può essere fatta, ad esempio, seguendo modelli frattale, ma noi ci concentreremo sullo studio di tecniche più affini al modello studiato fin ora (quindi il modello vettoriale).

3.3.1 Ottenere il gradiente: filtri di Sobel

Diciamo di voler prendere in analisi questa zona di pixel e ci concentriamo sul pixel bordato di viola:

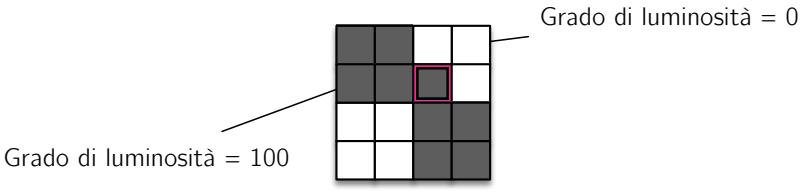


Figura 3.19: Una zona di pixel.

Siamo interessati a capire come si comporta la *variazione* di luminosità sia sull'asse delle x che sull'asse delle y. Crediamo dunque due filtri di Sobel (matrici) di dimensione 3×3, uno per ogni asse:

$$\text{Asse } x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \text{Asse } y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Ora non resta che applicare i filtri, ovvero sovrapporre la zona presa in analisi (un'area 3×3 centrata sul pixel che stavamo analizzando) ai due filtri e moltiplicare, elemento per elemento, il valore di luminosità con il peso riportato nel filtro. Dopodiché si sommano tutti i valori all'interno del filtro per ottenere i due valori di variazione.
Capiamo ora il perché della conformazione di quei valori nei filtri: se osserviamo il filtro per le ascisse, ad esempio, vediamo che andando direttamente a destra (seguendo l'asse) c'è un peso 2, se invece andiamo in diagonale ma sempre verso destra abbiamo un peso 1 (è più piccolo poiché non segue esattamente l'asse). Muoversi sull'asse verticale risulta irrilevante (ci sarà l'altro filtro a calcolarlo) mentre se la variazione è opposta all'asse consideriamo dei valori negativi (-2 se direttamente opposti all'asse e -1 se in diagonale). Quindi l'applicazione dei filtri non farà altro che indicarci in che direzione si ''muove'' l'intensità di colore. Nel nostro esempio otteniamo due matrici:

$$dx = \begin{bmatrix} -100 & 0 & 0 \\ -200 & 0 & 0 \\ 0 & 0 & 100 \end{bmatrix} = -200 \qquad dy = \begin{bmatrix} 100 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -200 & -100 \end{bmatrix} = -200$$

Ora che abbiamo ottenuto una coppia di valori (dx, dy) possiamo costruire il nostro *vettore del gradiente* (il vettore che indica la variazione di intensità).

Tale vettore avrà:

$$\text{Lunghezza: } \sqrt{(dx)^2 + (dy)^2}$$
$$\text{Angolo: } tan^{-1}\left(\frac{dx}{dy}\right)$$

Abbiamo quindi un vettore con angolo pari alla direzione della variazione (l'arcotangente) e lunghezza pari all'intensità della variazione di luminosità.

3.3.2 L'istogramma dei gradienti

Ovviamente il gradiente è calcolabile per ogni pixel (nell'esempio di prima avremmo dovuto applicare quindi i due filtri 16 volte ciascuno), dunque se ne creerà un'istogramma fondamentale per circoscrivere zone di variazione simile.
Evidentemente, essendo i valori ottenuti da questi calcoli possibilmente molti e molto distanti (vi sono infiniti numeri naturali), potremmo avere un vocabolario incredibilmente vasto. Sarà quindi necessaria una fase di quantizzazione in modo da avere un numero limitato di gradienti sull'asse delle x dell'istogramma.

Vi sono poi alcuni ulteriori accorgimenti che devono essere presi per poter avere un istogramma significativo.

Rimozione del rumore. Come abbiamo già detto i valori dei gradienti possono essere moltissimi: i valori per cui la frequenza è molto bassa verranno quindi rimossi ed interpretati come *rumore*.

Messa a fuoco. Se fra pixel adiacenti si ha un gradiente di uguale direzione e intensità simile (diciamo uno con intensità α e uno β con $\alpha > \beta$) allora si manterrà solamente il gradiente con α (ovviamente aumentandone di una unità la frequenza). L'effetto sull'immagine è quello di avere una messa a fuoco; accade qualcosa del genere:

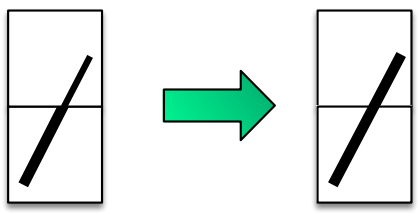


Figura 3.20: Effetto della messa a fuoco.

Dimensione del pattern. Volendo è possibile includere una terza componente al gradiente, indicante la distanza minima che c'è fra il pixel a cui è associato il vettore e il primo pixel che presenta la stessa texture. Questo ulteriore parametro ci permette di individuare la *dimensione del pattern* (la grana del pattern).

3.4 Studio della feature: riconoscimento dei contorni

Il riconoscimento dei contorni è necessario per poter delinare zone chiuse (regioni) all'interno dell'immagine e poi, eventualmente, individuare delle forme (lo vedremo in seguito). Il principio base è quello di cercare *zone omogenee*. Già questa definizione risulta imprecisa: possiamo cercare omogeneità di colore oppure di texture: sarà il relevance feedback ad aiutarci a capire cosa è più rilevante per l'utente. Esistono diverse tecniche per il riconoscimento dei contorni, vediamole una per una nel dettaglio.

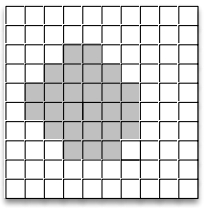


Figura 3.21: Esempio di immagine con una singola regione.

3.4.1 Individuare la regione

Iniziamo con il cercare la regione contigua, poi studieremo come rappresentarla.

Primo approccio: clutering fra pixel

La prima tecnica è estremamente semplice: di pixel simili vengono opportunamente raggruppati in cluster (si clusterizza per similarità). Pixel dello stesso cluster vengono quindi considerati facente parte di una stessa regione. Evidentemente se vi sono pixel molto distanti fra loro, il cluster verrà scisso in più parti (tante quante sono le ''zone'' di pixel spazialmente distanti).

Secondo approccio: regiongrowing

Viene scelto un pixel a caso (eventualmente si può guardare l'istogramma alla ricerca dei gradienti/colori più frequenti e si sceglie fra quelli) e poi si espande la regione per somiglianza col pixel scelto. Ci si fermerà quando si ha somiglianza da un lato e non somiglianza dall'altro.

Terzo approccio: watershed transformation

Per l'applicazione questa tecnica si fa uso di una metafora.
Prima di tutto l'immagine viene modificata e si colorano in modo diverso le zone a seconda del gradiente. Si assegna un colore scuro ai punti con *basso gradiente* e un colore *chiaro* ai punti con alto gradiente. In sostanza stiamo dicendo che le zone con minore variazione nel loro intorno sono più scure, mentre le zone chiare sono quelle dove la variazione è ampia.
Si interpreta ora l'immagine come una topologia: le zone scure rappresentano montagne mentre le zone chiare sono avvallamenti. A questo punto supponiamo di versare un liquido (a partire dalle montagne) e si studia il percorso che tale liquido segue: quel percorso rappresenta un bordo.

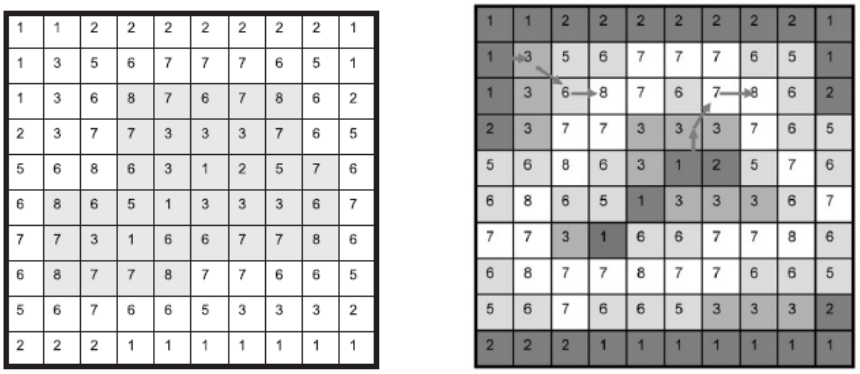


Figura 3.22: Esempio di esecuzione di watershed.

Nella prima immagine vediamo una regione centrale (che vogliamo indiduare), inoltre sono riportati i valori dei gradienti. La seconda immagine mostra la colorazione basata sul gradiente e mostra anche il fenomeno dello watershed: il liquido è stato versato in due punti diversi; come vediamo la tendenza è quella di accumularsi nella valle che ricalca abbastanza fedelmente il bordo della regione dell'immagine iniziale.

Nel caso si individuino dei percorsi non chiusi, si cerca di chiuderli ''al meglio''.

3.4.2 Rappresentare il bordo

Ora che abbiamo trovato la regione dobbiamo rappresentarla in qualche modo, così da essere in grado di compararla ad altre regioni.

Primo approccio: chaining

Questo approccio prevede l'uso di un protocollo comune secondo il quale ogni direzione intorno ad un certo pixel è codificata con un valore numerico. Ad

esempio potremmo scegliere questa configurazione:

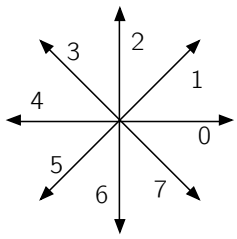


Figura 3.23: Protocollo di codifica delle direzioni.

Dopodiché si procede semplicemente scegliendo un punto del bordo (anche qui si adotta un protocollo per poter scegliere sempre lo stesso punto in ogni bordo, ad esempio il punto con minima x e minima y) e si percorre il perimetro del bordo assegnando man mano i valori codificati delle direzioni seguite:

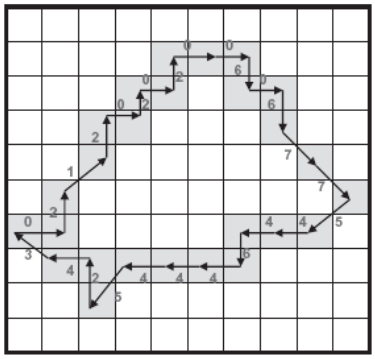


Figura 3.24: Esempio di esecuzione del chaining.

Abbiamo ripreso l'esempio della regione individuata prima con il watershed e abbiamo percorso, partendo dal pixel in basso a sinistra, tutto il perimetro. Se riportiamo ogni direzione scelta in una stringa otteniamo:

021202022262677544644452 43

A questo punto il problema della comparazione dei bordi è trasformato in un semplice confronto fra stringhe. Seppure questa soluzione sia computazionalmente attrente è troppo semplicistica: due immagini identiche ma scalate o ruotate diversamente avranno una stringa totalmente diversa (scalare una immagine significa eliminare le piccole protuberanze e quindi eliminare del tutto alcuni passi).

Secondo approccio: rappresentazione vettoriale

Il problema è risolto tramite l'impiego di una tecnica differente secondo la quale i bordi sono rappresentati da una serie di rette (vettori) interpolanti.
Anche in questo caso si sceglie un pixel di partenza secondo una qualche convenzione e poi si esamina il pixel adiacente (rispetto al bordo) e lo si collega tramite un vettore. Poi, si considera il pixel successivo (adiacente al secondo) e si prova a collegarlo usando una retta interpolatrice per i tre pixel: supponendo di avere una tolleranza ϵ, se l'errore di interpolazione (calcolato come distanza media fra ogni pixel e il vettore) è minore di ϵ allora accettiamo questo nuovo vettore e proseguiamo, altrimenti creiamo un nuovo vettore (quindi è come se ricominciassimo da capo) e lasciamo quello precedente così com'è. Procedendo in questo modo otterremo qualcosa del genere:

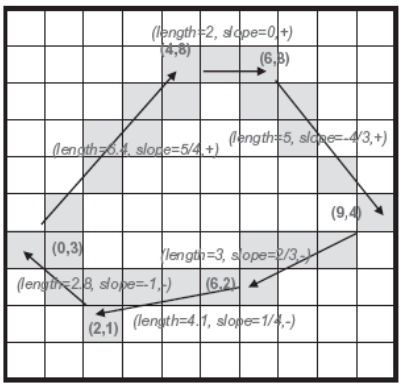


Figura 3.25: Rappresentazione vettoriale del bordo.

Ovvero una serie di vettori con direzione e lunghezza tali da approssimare il più fedelmente possibile (rispetto alla ϵ impostata) il bordo individuato. Il numero di vettori che troveremo dipenderà chiaramente dalla tolleranza. Chiaramente la lunghezza del vettore è la distanza euclidea fra il pixel iniziale e quello finale.

I problemi di rotazione e di scalatura sono ora gestiti:

- Nel caso di rotazione è sufficiente ruotare ogni vettore della differenza γ e ritroviamo il bordo.
- Nel caso di scalatura ogni vettore avrà un certo fattore di scala γ rispetto al suo corrispondente vettore non scalato.

Terzo approccio: uso delle serie temporali

Un ulteriore approccio vede l'utilizzo di serie temporali. Una serie temporale è utile quando si vuole rappresentare l'andamento di una qualche misura nel tempo. Sulle ascisse abbiamo quindi un andamento regolare ($t, 2t, 3t$, ecc.)

mentre sulle ordinate abbiamo le rilevazioni in quel dato tempo.
Nel nostro caso abbiamo bisogno di una serie temporale per le x e una per le y della nostra immagine. Partiamo dal punto centrale della figura e, definito un certo angolo θ che sarà l'unità della serie temporale (la t dell'esempio di prima), tracciamo una serie di rette. Più precisamente tracciamo una retta ogni θ fino a completare l'angolo giro. Quindi se $\theta = 6$ avremo una retta a 6 gradi, una a 12, una a 18, ecc.). Quando una retta incontra un pixel (un punto) che noi sapevamo essere del bordo, calcoliamo il valore di x nel punto (e lo inseriamo nella serie temporale su x) e poi calcoliamo la y (e lo inseriamo nella serie temporale su y). Al termine del procedimento avremo due serie temporali che potremo utilizzare per fare confronti (sfruttando algoritmi appositi, come quello del warping che vedremo in seguito).

3.4.3 La trasformata di Hough

Il nostro intento è quello di trovare una formula analitica che descriva il nostro bordo. Per motivi di semplicità supporremo di voler rappresentare una retta ma il discorso è ampliabile a qualsiasi figura che si voglia rappresentare (ma come vedremo il metodo è particolarmente costoso, quindi si impiega solamente per trovare figure regolari/con pochi lati).
Supponiamo quindi di conoscere la forma di cui vogliamo trovare l'equazione (una spezzata, descrivibile da una retta) e di conoscere tutti i punti che la compongono. Come sappiamo l'equazione della retta in forma esplicita è:

$$y = mx + q$$

dove m è il coefficente angolare e q è l'intercetta. Questa formulazione è utile quando si hanno sia m che q e si vuole trovare una soluzione per y fissato un certo x oppure viceversa.
Se ci trovassimo nel mondo del continuo sarebbe sufficiente prendere due punti fra quelli che possediamo e creare un sistema con due equazioni, fare qualche sostituzione e trovare le incognite m e q. Ma non ci troviamo nel mondo del continuo, le x e y sono associate ai pixel che sono ovviamente discreti e questo significa che *potrebbe non esserci soluzione*. Vediamo dunque cosa possiamo fare.

L'equazione per la retta in forma esplicita *non è utile* poiché per noi le incognite sono il coefficiente angolare e l'intercetta, non le coordinate x ed y. Effettuiamo per questo una *trasformazione di spazio* (ecco perché la tecnica è detta ''trasformata'') e otteniamo quindi:

$$q = y - mx$$

Adesso i termini noti sono x e y e le nuove incognite sono q ed m.

L'idea del voting

Come fare a trovare la coppia (m, q) più appropriata? Intuitivamente vorremmo ''quella che soddisfa di più'' i punti, o meglio, quella che soddisfa più punti possibili. Per poter trovare quale coppia sia, usiamo il sistema del *voting*. Istanziamo una matrice dei voti dove sulle colonne riportiamo un range di possibili m e sulle righe mettiamo un range di possibili q. Poi istanziamo le nostre equazioni per ogni x e per ogni y e proviamo ogni combinazione di m e di q alla ricerca di quelle coppie che soddisfano l'equazione presa in esame: per ogni coppia(m_i, q_j) che soddisfa una equazione andiamo a fare un +1 nella cella i, j corrispondente della matrice che apparirà dunque così:

$$V = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & 0 & \cdots & 3 \\ 1 & 2 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \end{pmatrix} \begin{matrix} q_1 \\ q_2 \\ \vdots \\ q_s \end{matrix}$$
$$m_1 m_2 \cdots m_k$$

La coppia scelta risulterà quella con più voti, in questo caso quindi (q_k, m_2). Nel caso in cui si abbia una parità fra più coppie abbiamo una indicazione del fatto che vi sia una linea ''ramificata'' (una spezzata) e che quindi si divide in più parti.

Rappresentazione più effiace: non con le coordinate polari

La spiegazione appena riportata una grande problematica: per linee verticali il coefficente angolare risulterebbe essere infinito, inoltre è difficile definire una matrice finita per uno spazio potenzialmente molto grande definito dalle m e dalle q. Per questi motivi non si utilizza l'equazione appena illustrata ma si adopera l'equazione in *coordinate polari*, ovvero:

$$\ell = x cos\theta + y sin\theta$$

dove ℓ è la distanza tra l'origine e la linea mentre θ è l'angolo fra il vettore e l'asse delle x. Lo spazio (ℓ, θ) così ottenuto è decisamente più trattabile poiché i valori in cui variano ℓ e θ sono finiti (rispettivamente la dimensione dell'immagine e 2π). Se tali valori dovessero comunque essere estremamente vari (e quindi la matrice dei voti essere molto grande) si procederà ad una quantizzazione.

3.4.4 L'algoritmo SIFT

Ciò che abbiamo considerato per ora sono tutte feature globali che caratterizzano quindi una immagine nella sua totalità. L'algoritmo *SIFT* (Scale-invariant

feature transform) ci permette di ottenere informazioni puntuali. Vogliamo, ad esempio, essere in grado di riconoscere parti comuni di fotografie (al di la di scala, rotazione, illuminazione, ecc.). Consideriamo ad esempio queste immagini:

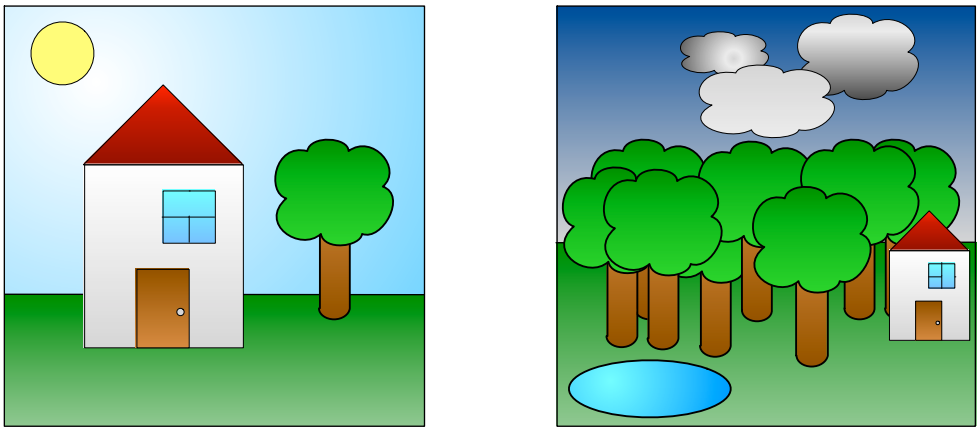


Figura 3.26: Due immagini dove appare la stessa casa.

Come vediamo sono molto diverse fra loro dal punto di vista dell'istogramma dei colori (quindi molto distanti globalmente) ma la casetta è praticamente identica (a parte la scala). SIFT è quindi un algoritmo che si concentra su determinati punti dell'immagine (detti *keypoint*) alla ricerca di zone locali ''interessanti''. Nel nostro caso uno dei keypoint sarebbe la punta del tetto della casetta.
Esiste una versione di SIFT sviluppata apposta per il rilevamento dei tratti somatici caratterizzanti di un viso (e quindi usato per il confronto di visi) ed è detta ASIFT.

Overview dei passi

Facciamo una piccola overview dei passi effettuati da SIFT che andremo a dettagliare in seguito.

1. Scale-space extrama detection: si cercano punti di interesse operando a diverse scale (operazione di *smoothing*).
2. Keypoint localization: fra i punti trovati al passo 1 si identificano i più robusti (si elimina il rumore e si eliminano i punti mal localizzati).
3. Orientation assignment: l'orientamento dell'immagine è normalizzato (indipendenza dalla rotazione).
4. Keypoint descriptor computation: vengono generati i descrittori per i keypoints (particolari vettori che caratterizzano i punti in modo che siano identificabili).

Passo 1: scale-space extrema detection

Il nostro obiettivo è quello di riconoscere forme uguali anche a scale diverse. Si effettua quindi una operazione di sfocamento (anche detta blur o smoothing) usando un filtro gaussiano:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{\sigma^2}}$$

Tale filtro è applicato per ogni pixel ma in maniera convoluttiva (cioè basandosi anche sui pixel intorno a quello preso in esame). Quindi si usa:

$$L(x, y, \sigma) = G(x, y, \sigma) \otimes I(x, y)$$

dove $\otimes$ è il simbolo di convoluzione. L'effetto finale è quello di fare una sorta di media pesata fra un pixel e quelli intorno a lui, ma dato che usiamo una funzione gaussiana il peso sarà guidato dalla distanza del pixel centrale rispetto ai suoi adiacenti: pixel più distanti da quello preso in esame peseranno di meno rispetto a quelli più vicini (e ovviamente peseranno meno rispetto al pixel stesso).

Questa operazione è dunque compiuta su ogni pixel (in maniera convoluttiva, ricordiamo) dell'immagine usando un certo parametro σ. Abbiamo quindi ottenuto una versione σ-sfocata dell'immagine iniziale. Questa operazione è ripetuta molte volte con σ diversi, in generale a passo k (quindi σ, $k\sigma$, $2k\sigma$, ecc). Ora abbiamo una lista di immagini man mano più sfocate (all'aumentare di σ aumenta la sfocatura).
Non resta che considerare le differenze fra due sfocature adiacenti:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

ed esaminarle:

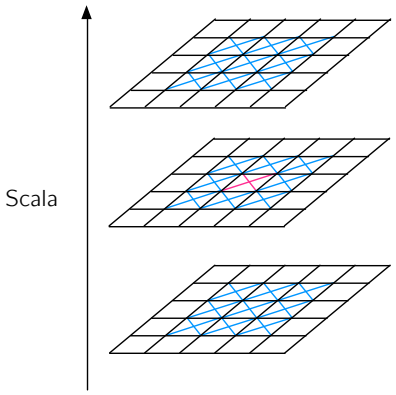


Figura 3.27: Confronto con i 26 vicini.

Come vediamo ogni pixel è confrontato con i suoi 26 vicini (8 intorno a lui alla stessa scala, 9 alla scala superiore e 9 alla scala inferiore). Il confronto avviene in termini di gradiente, quindi i punti che vengono individuati come *massimi locali* o *minimi locali* sono punti di interesse. Stiamo infatti parlando di un cambio di gradiente e quindi di un probabile punto di interesse. Se andiamo a riprendere l'immagine della casetta vediamo che il tetto della casa era in effetti un punto che corrisponde a questa descrizione, quindi, un ottimo candidato keypoint.

Passo 2: keypoint localization

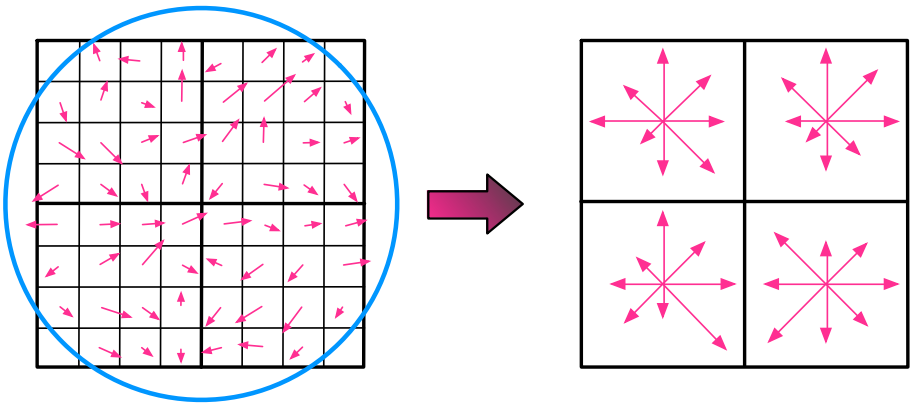
Andiamo a pulire l'insieme dei keypoint trovato al passo precedente andando a rimuovere i keypoint con poco contrasto (vengono interpretati come rumore) e quelli mal localizzati. Un punto mal localizzato è un punto che non porta informazioni interessanti, ad esempio su una linea retta avremo molti punti con uguale variazione (alta) di gradiente ma tali punti saranno tutti uguali ed allineati, quindi poco interessanti (in questa analisi, s'intende). I due estremi della retta saranno invece più rilevanti poiché tutti i suoi vicini hanno gradiente diverso.

Passo 3: orientation assignment

Per ogni keypoint trovato si crea un istogramma di gradienti ad ogni scala σ dove il keypoint si trova (anche in questo caso nella creazione dell'istogramma si usa una funzione Gaussiana che pesa maggiormente i pixel vicini rispetto ai distanti. Il parametro di qeusta funzione è 1.5σ, con il σ della scala a cui stiamo lavorando). Ora non resta che orientare in maniera coerente le immagini basandoci sull'istogramma: ad esempio potremmo ruotare verso l'alto il gradiente massimo. Nel caso in cui i picchi sull'istogramma siano più di uno il keypoint viene duplicato e mantenuto con le due diverse orientazioni (per non perdere nessun risultato eventuale).

Passo 4: orientationkeypoint descriptor computation

Non resta che formalizzare quanto scoperto in merito al keypoint in modo da poter usare quei dati per identificare un certo punto. Si considera dunque un'area (tipicamente 16×16) intorno ad un keypoint e si ottiene un array di gradienti (tipicamente 4×4) che riassuma le informazioni sui gradienti intorno a quel keypoint.
Gli array terranno conto dell'orientation impostata al passo 3 (per poter confrontare i vari punti). Nell'esempio a seguire abbiamo ottenuto da un'area 8×8 un descrittore 2×2:



Gradienti intorno al keypoint Descrittore del keypoint

Figura 3.28: Generazione del descrittore per un keypoint.

3.4.5 Dai contorni alle forme

Una volta definite diverse regioni all'interno di una immagine, possiamo caratterizzarla tramite la sua *segmentazione*. La *segmentazione* di una immagine I è un set $\{R_1, ..., R_n\}$ di regioni tali che:

- $R_1 \cup ... \cup R_n = I$ (cioè l'unione fra le regioni sia l'immagine per intero).
- $R_i \cap R_j \ \forall i, j$ con $i \neq j$ (cioè ogni regione sia disgiunta dalle altre).
- $H(R_i) = true \ \forall i$ (H è una funzione boolena che ci dice se la zona è omogenea, quindi vogliamo che ogni regione sia uniforme).
- Se R_i ed R_j sono adiacenti, allora $H(R_i \cup R_j) = false \ \forall i, j$ con $i \neq j$ (le regioni sono massimali, ovvero, vogliamo che regioni vicine siano disomogenee).

Ogni forma estratta sarà a sua volta caratterizzabili da feature locali, fra cui:

- Rotondità.
- Rapporto fra asse verticale ed asse orizzontale (se è una regione più lunga che larga ad esempio).
- Orientamento della dimensione con maggiore valore.

3.5 Query spaziali

Mediante l'uso di query spaziali siamo quindi interessati a lavorare sulla posizione relativa delle feature, a che angolazione si trovano, qual è la distanza fra loro, ecc. Una delle applicazioni più classica delle query spaziali è la fingerprint detection.

3.5.1 Minimum Bounding Rectangles

Consideriamo questi oggetti:

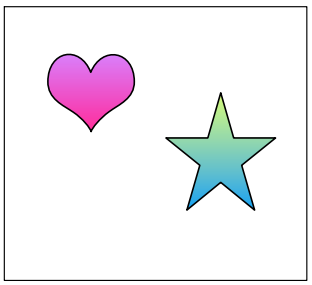


Figura 3.29: Due oggetti nello spazio.

L'idea più banale che ci può venire in mente per studiare la posizione di questi oggetti è quella di tracciare un rettangolo tangente ai bordi delle due figure, per poi sfruttare le coordinate dei vertici di questi rettangoli al fine di determinare la posizione reciproca degli oggetti. Questa tecnica prende il nome di *Minimum Bounding Rectangles* (MBR in breve). Otteniamo così: È ora molto

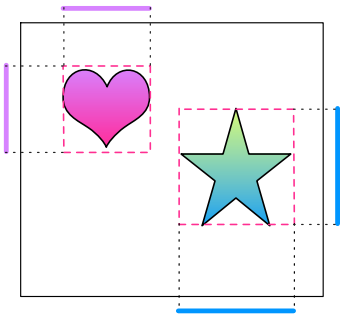


Figura 3.30: In rosso, i bounding box.

semplice rispondere alla domanda ''Quale dei due oggetti si trova a sinistra? '': evidentemente il cuore è alla sinistra della stella.
Ma se ci chiedessimo quale dei oggetti si trovi sopra all'altro, la risposta sarebbe di certo meno immediata.

3.5.2 Plane Sweep

L'MBR è una tecnica poco costosa ma troppo semplicistica. Consideriamo infatti la Figura 3.31. Come vediamo, nel caso di figure poco regolari l'MBR spreca molto spazio, rendendo i confronti possibilmente errati. Nell'esempio vediamo un oggetto che risulta sovrapposto ad un altro anche se in realtà è solo adiacente.

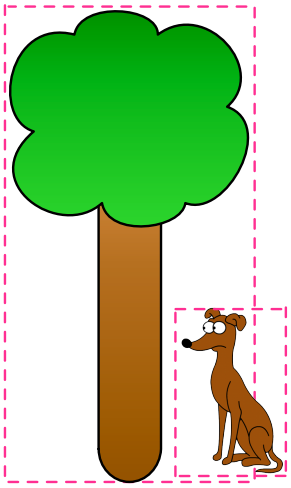


Figura 3.31: L'MBR è estremamente impreciso.

Si adotta per questo motivo una tecnica più precisa, chiamata *plane sweep* (letteralmente spazzamento di piani) che fa passare sull'immagine una serie di piani a diverse angolazioni cercando i punti di tangenza con i bordi della figura. Si determinano così gli *event point*, che sarebbero i punti di tangenza. La tecnica produce quindi questo risultato:

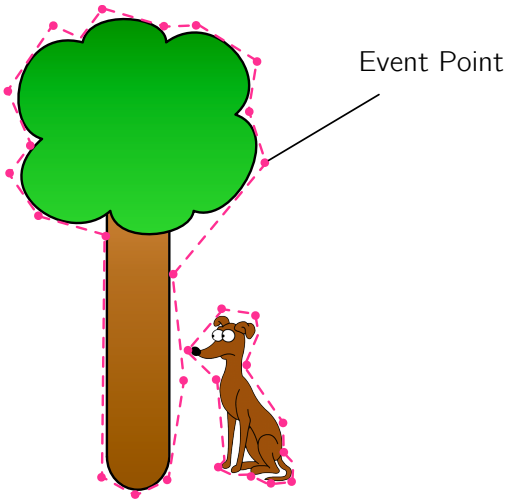


Figura 3.32: Applicazione dei plane sweep.

3.5.3 Lavorare con le 2D string

Il risultato di una operazione di plane sweep è quindi una serie di event point distribuiti sul piano. Per poter rappresentare la figura si rende necessario tra-

durre queste posizioni spaziali in qualche modo: l'idea è quella di ottenere due stringhe (una per asse) che descrivano l'orientamento dei punti. Le operazioni da compiere sono quindi 2:

1. Assegnare un label ad ogni punto (vedremo dopo come assegnarli coerentemente).
2. Definire (per ogni asse) un ordinamento fra i punti e rappresentarlo come stringa.

Le relazioni di ordinamento fra punti che useremo sono:

- = (stessa posizione).
- < (alla sinistra di).
- : (nello stesso set) [relazione poco usata].

Ricapitolando, si prende l'immagine, si spazza tramite plane sweep, ad ogni event point si assegnano dei label e poi si creano i due ordinamenti propri della immagine che ci saranno utili per effettuare i confronti.

Quindi, per esempio, dati questi punti (a cui abbiamo già assegnato i label):

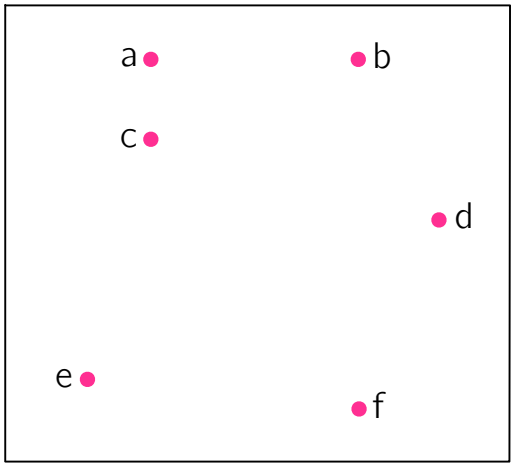


Figura 3.33: Serie di punti ottenuti tramite plane sweep.

faremo un ordinamento orizzontale (da sinistra verso destra) e verticale (dall'alto verso il basso) che ci darà come risultato:

Ordinamento orizzontale : `e<a=c<f=b<d`

Ordinamento verticale : `b=a<c<d<e<f`

Per far sì che questa tecnica funzioni, si rendono necessari due accorgimenti. Vediamoli nel dettaglio.

Simmetria: break the tie. La simmetria in senso matematico prevede che a=b sia uguale ad b=a. Ma giacché dovremo fare un confronto fra stringhe, si rende necessario *break the tie*, e quindi decidere chi dei due deve essere il primo event point ad apparire nell'uguaglianza. È evidente che tale scelta deve essere uguale per ogni immagine.
Si deve dunque scegliere un protocollo, ad esempio, il primo punto è quello che ha coordinata più grande sull'asse che non si sta considerando (è il protocollo usato nell'esempio di prima).

Assegnare coerentemente i label. Chiaramente una scelta coerente dei label è fondamentale per far sì che il confrontro fra stringhe sia sensato. Per coerente si intende che qualsiasi sia la disposizione delle figure all'interno di una immagine, i punti uguali devono essere contrassegnati con gli stessi label: Per ottenere questo risultato si usano tecniche quali quelle già viste durante lo

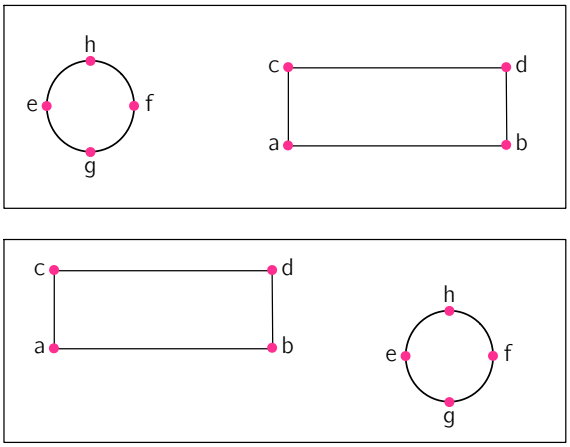


Figura 3.34: Assegnamento coerente dei label.

studio dell'algoritmo SIFT, cioè, si confrontano i descrittori degli event point per capire se sono lo stesso punto. Ricordiamo che i descrittori portano molte informazioni e dunque sono in grado di caratterizzare in modo sufficientemente preciso un punto.

Confronto tramite edit distance

Il problema di confronto fra figure si è ora ridotto ad un mero controllo fra stringhe. Ad esempio dovendo confrontare le due immagini in Figura 3.35 confronteremo semplicemente le quattro stringhe caratterizzanti le figure contenute nelle due immagini.
La distanza fra stringhe è quantificata tramite *edit distance*, cioè la quantità di operazioni richieste per modificare una stringa in modo da renderla identica a quella con cui la sto confrontando. Le operazioni possibili sono:

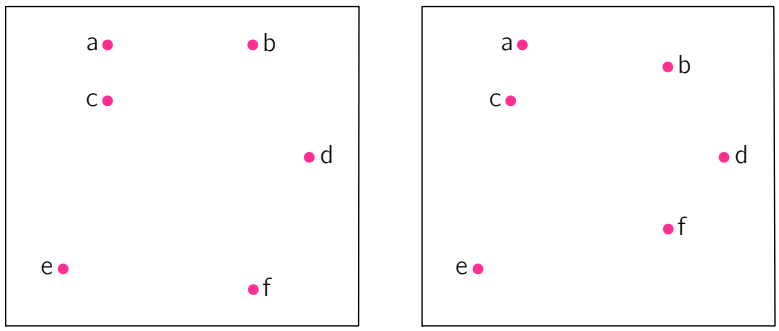


Figura 3.35: Stringhe caratterizzanti le forme di due immagini.

- *Add*: si aggiunge una lettera.
- *Delete*: si rimuove una lettera.
- *Swap*: lettere adiacenti vengono invertite di posizione.

Ad ogni operazione è attribuito un costo: minore sarà il costo necessario per modificare una stringa in un'altra e maggiore sarà la somiglianza fra due.
La similarità tramite edit distance permette anche una discreta personalizzazione del calcolo. Se ad esempio siamo più interessati ad avere allineamenti

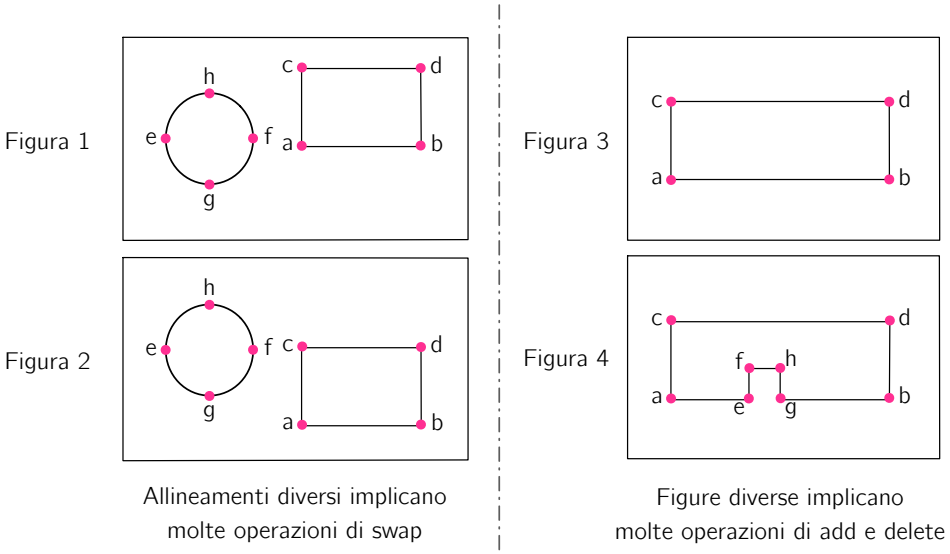


Figura 3.36: Pesando le operazioni caratterizziamo la distanza.

simili fra figure piuttosto che esattamente le stesse figure renderemo l'operazione di *swap* più costosa. Se invece siamo interessati ad avere forme perfette

e non ci importa della loro posizione, renderemo più costose le operazioni di *add* e *delete*.

3.5.4 θR string: risolvere problemi di rotazione e traslazione

La rappresentazione 2D string soffre di due annose problematiche: a fronte di variazione di scala, traslazione e rotazione, due figure per il resto identiche generano stringhe molto diverse e quindi risultano diverse.
È necessario innanzi tutto modificare il modo in cui la stringa viene generata. Inizialmente individuiamo il *punto di massimo carico* della figura, ovvero il punto con coordinata x la media fra le x dei punti del bordo e come y la media delle y dei punti del bordo.
Consideriamo il punto di massimo carico come l'origine di un vettore che spazza la figura ogni θ gradi. Quando il vettore incide contro un punto, viene memorizzato il raggio R (la distanza fra il punto e l'origine) oltre che ovviamente il θ a cui si trova il vettore. Quando troviamo più punti incidenti su uno stesso vettore dovremo anche qui effettuare un break the tie (ad esempio ordiniamo l'uguaglianza basandoci sull'R).

Il problema della rotazione è risolto poiché è sufficiente *allineare il punto di partenza* delle due figure (cioè entrambe devono partire dallo stesso punto) per poi poter confrontare la stringa. Tale operazione è effettuabile ancora una volta ricorrendo ai descrittori di un punto: si parte da una delle due immagini e si cerca il primo punto, appena questo è stato trovato si cerca nell'altra immagine il punto più simile e si parte a spazzare la figura da quel punto. In questo modo gli assi risultano allineati.

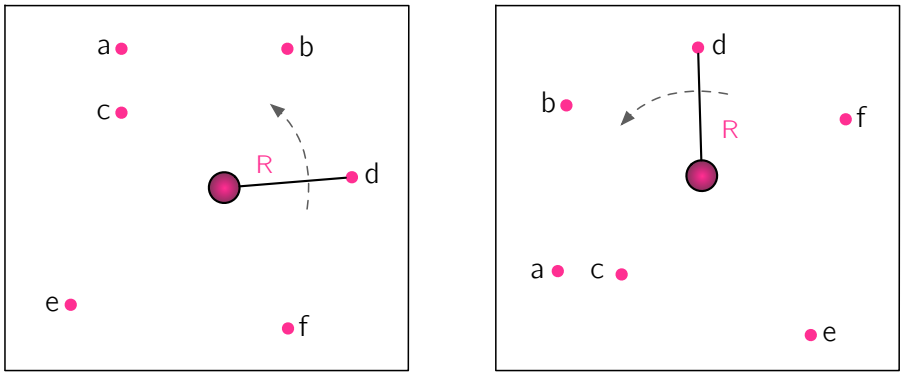


Figura 3.37: I due punti di partenza sono allineati.

Purtroppo il problema sulla scala rimane, ma d'altronde due figure scalate diversamente in immagini diverse cambiano effettivamente la loro posizione reciproca.

3.5.5 2D estring

Esistono anche versioni alternative del 2D string che invece di considerare i punti come a loro stanti considerano degli intervalli. A questo punto il compare è a livello di intervalli (quindi di frazioni spaziali) e vi sono molti più operatori: contain, meet, begin, end, overlap, equal, less than, inverse.

Capitolo 4

Altri multimedia

4.1 I video

Il video è un tipo multimedia che racchiude al suo interno una componente audio ed una video. Le due componenti sono ovviamente correlate fra loro da un legame di tipo temporale (sono sincronizzate).
L'effetto ''video'' è ottenuto basandosi sulla scoperta dei fratelli Lumière, ovvero che una serie di immagini in sequenza, se riprodotte ad una velocità sufficiente, danno l'illusione del movimento fluido. Un video è quindi in sostanza una serie di fotogrammi, detti *frame*. L'occhio umano non rileva la transizione fra un frame e l'altro quando ve ne sono almeno 30$al secondo$ (30fps in breve).

4.1.1 La struttura temporale

Capiamo dunque che analizzare un video sarebbe estremamente costoso se considerassimo un'analisi frame per frame (un video di 30 minuti contiene circa 54.000 frame). Si adotta per questo motivo questa suddivisione:

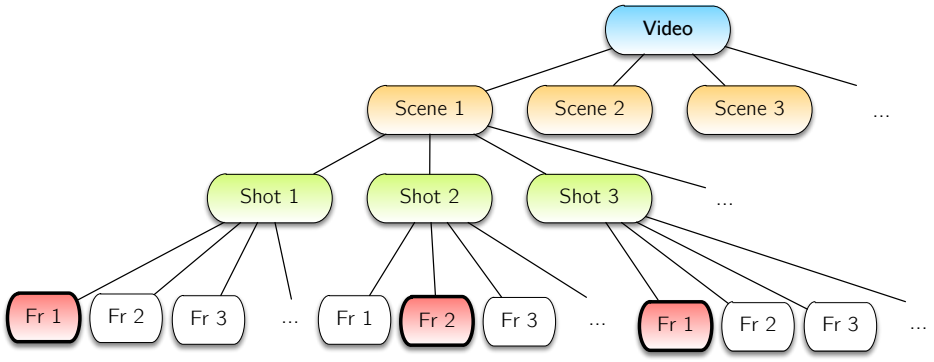


Figura 4.1: Struttura temporale di un video.

Un video è costituito da una serie di *scene* che rappresentano unità semantiche omogenee; ad esempio potremmo avere la scena di un uomo che mangia ad un bar, piuttosto che di un gatto che insegue un topo.
Ogni scena è identificata da una serie di *shot* ed il cambio di shot avviene quando la telecamera cambia angolazione. Questo fa sì che all'interno di uno stesso shot l'insieme dei frame sia molto omogeneo. Scendendo ancora di livello, ogni shot è identificato da uno dei frame che lo compongono, detto *keyframe*.
Questa semplificazione ci permette di avere un frame rappresentativo degli altri per ogni shot e quindi di migliorare di molto il costo computazionale. Quando una query è eseguita si prendono quindi in considerazione i keyframe, e, una volta fatta una scrematura a questo livello, si va eventualmente a verificare il predicato sull'insieme dei frame rappresentati dai singoli keyframe selezionati.

L'intento è quello di avere una rappresentazione che ci indichi ''ogni volta che cambia qualcosa'': tale cambiamento avviene a livello di shot, quindi, di keyframe.

4.1.2 Motion e problematiche

Il tipo di query che dovremo trattare saranno ora di tipologia diversa, dovremo infatti chiederci ''se qualcosa accade nel tempo''. Il nostro intento è dunque quello di correlare le informazioni fra un frame ed il suo adiacente, per capire come si muovo gli oggetti all'interno del video. Lo studio del *motion* (il movimento) presuppone però di saper distinguere fra tre macro tipologie di movimenti che possono accadere in un video:

- Movimento della camera.
  - Zoom (variazione di focus).
  - Tilting (variazione verticale).
  - Panning (variazione orizzontale).
  - Tracking (variazione orizzontale trasversale).
  - Booming (variazione verticale trasversale).
  - Dollying (variazione orizzontale e laterale, si gira intorno ad un obiettivo).
- Movimento degli oggetti (quello che siamo interessati a rilevare).
- Movimento dettato dal cambio di scena.

È estremamente difficile comprendere se il muoversi degli oggetti all'interno di una scena sia dettato dall'effettivo agire dei personaggi oppure se è un motion dettato dal movimento della camera. Per ottenere queste informazioni si ricorre spesso a metadati (ad esempio gli script del regista).

4.1.3 Object tracking e object identity

Non tratteremo questo argomento nel dettaglio, ma l'obiettivo dell'analisi di un video si può riassumere nell'object tracking e nell'object identity. Questi due aspetti rendono possibile l'identificazione di un certo oggetto al fine di individuarne i movimenti all'interno del video.
Si tratta quindi di saper riconoscere un oggetto a diverse angolazioni e poi di interpretare il suo movimento anche tenendo conto dello spostamento della camera.

Figura 4.2: Esempio di computazione in seguito all'object tracking.

4.1.4 Il tempo e la sua rappresentazione

Come abbiamo già detto, il tempo è uno dei fattori fondamentale di un video ed è materia di interrogazione da parte di una query. Per questo motivo dobbiamo in qualche modo rappresentarlo, dobbiamo rappresentare gli eventi su una linea temporale e dobbiamo anche capire come specificare query relative al tempo. Si noti che la nozione di tempo è fondamentale: se in uno stadio sentiamo il boato del pubblico e poi la palla entra in rete, probabilmente non è stato fischiato un rigore. Mentre invece se i due eventi sono all'inverso abbiamo un goal seguito dal festeggiamento dei tifosi.
La rappresentazione del tempo è trattata nella Sezione 4.2. Per ora fingiamo di sapere come funzioni tale rappresentazione e parliamo delle query su video.

4.1.5 Query su video

Esistono diverse tipologie di query. Vediamone una carrellata (in ordine di complessità).

Query su oggetti. In questo tipo di query si richiede che una serie di oggetti siano presenti nel video, in qualsiasi ordine e con qualsiasi frequenza. Risolvere queste query è semplice: dopo aver capito la semantica delle parole espresse nella query è sufficiente procedere di keyframe in keyframe alla ricerca degli oggetti richiesti.

Figura 4.3: Query su oggetti.

Query su frame. Si chiede ora che esistano frame *distinti* ove gli oggetti siano posizionati come indicati dalla freccia. Abbiamo quindi una richiesta di tipo *spaziale*, (non ancora temporale) sui singoli frame, che possono quindi susseguirsi in un qualsiasi ordine.

Figura 4.4: Query su frame.

Simple action query. Nelle simple action query si richiede che un certo susseguirsi di frame accada entro un determinato tempo (espresso in numero di frame).

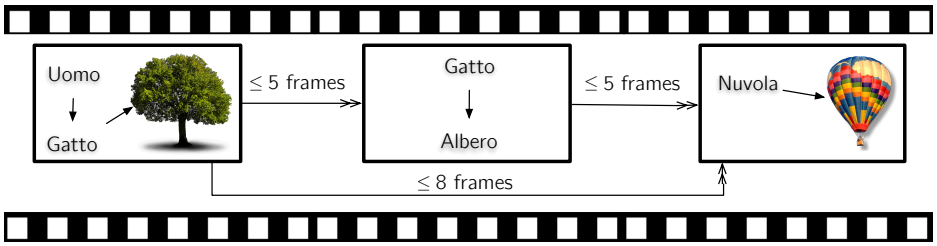


Figura 4.5: Simple action query.

In questo esempio è richiesto che vi sia un frame con un uomo (più in alto rispetto ad un gatto il quale si trova a sinistra di un albero), dopodiché entro 5 frame deve esserci un gatto sopra ad un albero ed entro altri cinque frame una nuvola deve trovarsi a sinistra della mongolfiera. Tutto deve però accadere entro 8 frame. Anche in questo caso l'analisi è effettuata sui keyframe e nel caso in cui ci sia il match degli oggetti ma non della loro posizione spaziale si può andare a vedere se in qualche altro frame gli oggetti sono piazzati come richiesto dalla query.

Composite action query. Le composite action query permettono di definire gruppi diversi di frame che si devono succedere all'interno del video. Ad esempio potremmo avere:

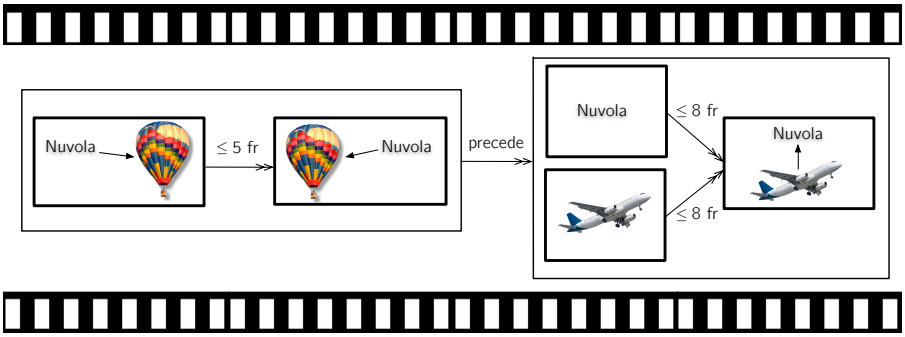


Figura 4.6: Composite action query.

Si sta chiedendo che una nuvola passi dalla sinistra alla destra di una mongolfiera entro 5 frame. In seguito (un qualsiasi momento successivo) si richiede che vi siano due frame (eventualmente lo stesso) con una nuvola ed un aereo e che entro 8 frame dalla comparsa di ognuno dei due ve ne sia uno dove la nuvola appare sopra all'areo.

4.2 Il tempo

4.2.1 Due modelli per il tempo

Il tempo è un fattore fondamentale in molti tipi di multimedia. Si è quindi reso necessario creare un formalismo che fosse in grado di rappresentare il tempo (sempre al fine di poter fare dei confronti). Così come per lo spazio, possiamo definire il tempo secondo due diversi modelli. Possiamo avere una *rappresentazione ad istanti* raffrontabili fra loro tramite tre tipi di relazione [<, =, >], oppure in termini di *intervalli* (avendo in questo modo 13 tipi di relazioni).

4.2.2 Algebra ad intervalli

Iniziamo parlando del secondo modello.

Relazioni fra intervalli

Ecco una tabella che riporta tutte le 13 possibili relazioni fra intervalli:

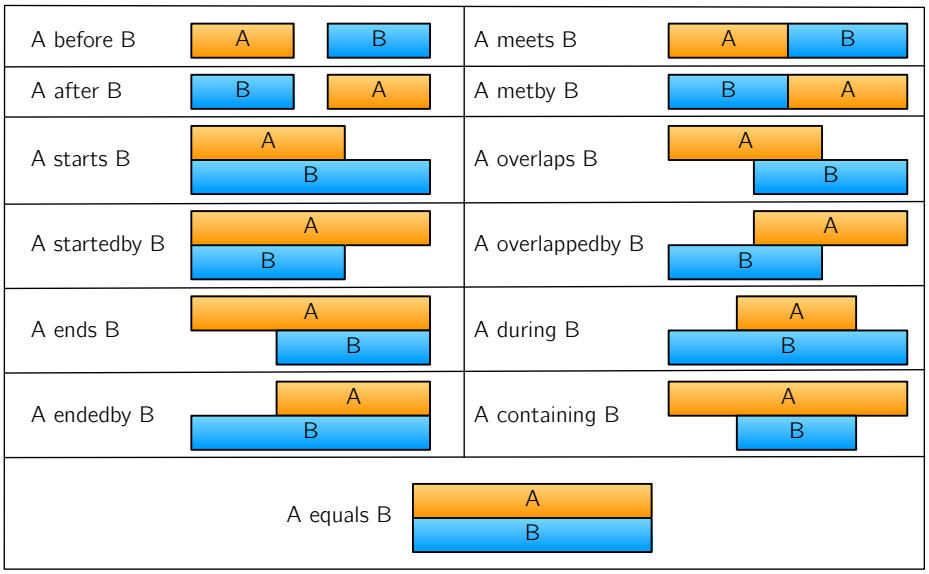


Figura 4.7: Tabella delle relazioni fra intervalli

Come vediamo le relazioni sono in realtà 6 più l'equivalenza e poi si hanno le altre 6 inverse.

Assiomi, definizioni e regole

Chiaramente questo modello dispone anche di assomi e definizioni.

Se I_1, I_2, I_3 sono intervalli diversi fra loro, allora possiamo dire che:

$$before(I_1, I_2) \ \wedge \ before(I_2, I_3) \Rightarrow before(I_1, I_3) \quad \text{(transitività)}$$

$$meets(I_1, I_2) \wedge during(I_2, I_3) \Rightarrow$$
$$overlaps(I_1, I_3) \ \vee \ during(I_3) \ \vee meets(I_1, I_3)$$

È poi possibile definire ulteriori predicati come ad esempio il predicato *in* (nel senso di inclusione propria):

$$in(I_1, I_2) \Leftrightarrow during(I_1, I_2) \ \vee \ starts(I_1, I_2) \ \vee \ ends(I_1, I_2)$$

Infine, si può capire se un certo predicato p o q è mantenuto all'interno di un intervallo tramite queste regole (i è un intervallo):

$$holds(p, I) \Leftrightarrow \forall i \ (in(i, I) \Rightarrow holds(p, i))$$
$$holds(and(p, q), I) \Leftrightarrow holds(p, I) \wedge holds(q, I)$$
$$holds(or(p, q), I) \Leftrightarrow \forall i \ (in(i, I) \Rightarrow not(holds(p, i)))$$

4.2.3 Minimal labeling problem

È evidente che adottando un'algebra di questo tipo siano molte le informazioni inferibili. Ma questo può comportare un problema quando si vanno a studiare le correlazioni fra tutti gli intervalli temporali contenuti ad esempio in un video. Questo problema, detto *minimal labeling problem*, è NP completo. Ma se la KB di partenza è definita in maniera astuta (non ci sono OR nelle regole) allora esistono algoritmi in grado di risolvere il problema in tempo $O(N^3)$. Inoltre, si tende a mantenere l'informazione minima (dalla quale è possibile inferire tutto) nella KB così da non dover maneggiare enormi moli di regole/fatti.

4.2.4 Algebra puntuale

Alternativalmente al modello ad intervalli possiamo usare quello puntuale. L'unità base è quindi l'istante invece che l'intervallo e gli unici tre operatori sono prima, dopo ed uguale (al posto dei 13 operatori sugli intervalli). L'impostazione è molto simile a quella già vista per trattare lo spazio.
D'altronde ogni intervallo può essere descritto come una coppia di istanti st ed et. Quindi, *nel caso di assenza di disgiunzioni*, vale l'importante equivalenza:

$$\text{Algebra ad intervalli} \ = \ \text{Algebra puntuale}$$

Se ne deduce che ogni relazione definita per l'algebra ad intervalli sia traducibile nei termini di st ed et (quindi di algebra puntuale). Per esempio:

$$during(A, B) = st(A) > st(B) \wedge et(A) < et(B) \wedge$$
$$st(A) < et(A) \wedge st(B) < st(B)$$

4.2.5 Object Composition Petri Net

Per quanto concerne l'evoluzione temporale di un certo media, usiamo una rappresentazione molto adottata nei sistemi per il trattamento delle informazioni: le *Petri Net* (opportunamente modificate). Introduciamo prima le reti di petri ''classiche'' e poi vediamone l'applicazione nel nostro campo.

Le Petri Net sono un modello adottato per rappresentare situazioni di concorrenza. Più precisamente una Petri Net è un grafo bipartito con nodi di tipologia *places* e di tipologia *transition* (per questo sono anche dette reti P/T); ogni

arco collega necessariamente nodi di tipologia diversa.
Tramite nodi places rappresentiamo delle risorse mentre i nodi transition indicano delle operazioni che fanno uso dei places da cui sono puntati. Ecco quindi un esempio di rete P/T:

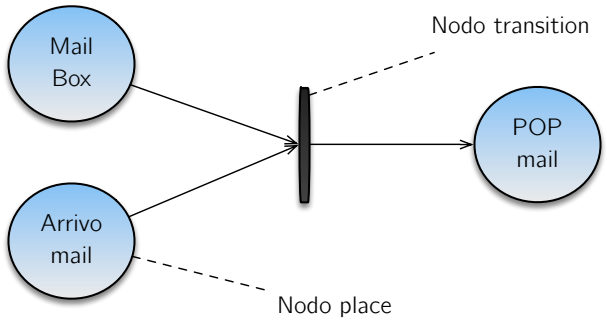


Figura 4.8: Esempio di P/T net.

Il flow è regolato tramite la presenza di *token* che si trovano nei places. Più precisamente, una transition viene *attivata* (in modo non deterministico) quando ogni place ad essa collagata dispone di un token. La transizione consuma dunque i token dai places ''precondizioni'' (indietro nel grafo) e ripone un token in ogni place a cui è collegata (in avanti nel grafo). In questo modo si genera un flusso di token a partire dall'inizio del grafo verso la fine.
Ogni place può avere anche più di un token, e nel caso si voglia ristabilire un token in seguito al suo uso è possibile creare un ciclo dal nodo transition a quello place di partenza.

Noi adoperiamo le reti di Petri assimilando al concetto di place una certa porzione del multimedia ed al concetto di transition il passaggio fra una porzione e l'altra. Questo tipo di reti sono dette *Object Composition Petri Nets*, OCPNs in breve. Quindi, ad esempio, un servizio del telegiornale può essere rappresentato come segue:

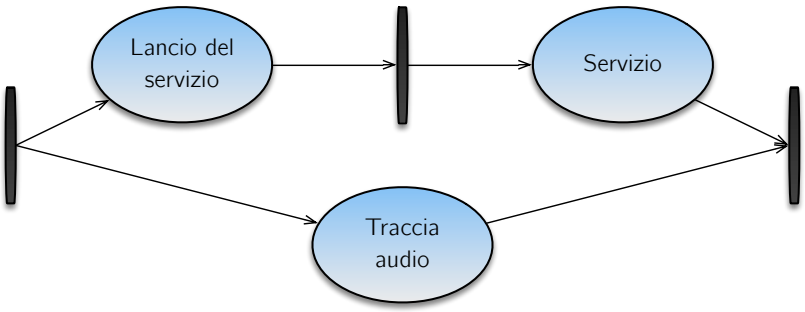


Figura 4.9: Object Composition Petri Net per un telegiornale.

Confronto fra OCPNs

Le OCPNs si prestano moto bene al confronto (motivo per cui le abbiamo scelte come modello). Consideriamo ad esempio questa query (in alto) e questi risultati (in basso): Notiamo che:

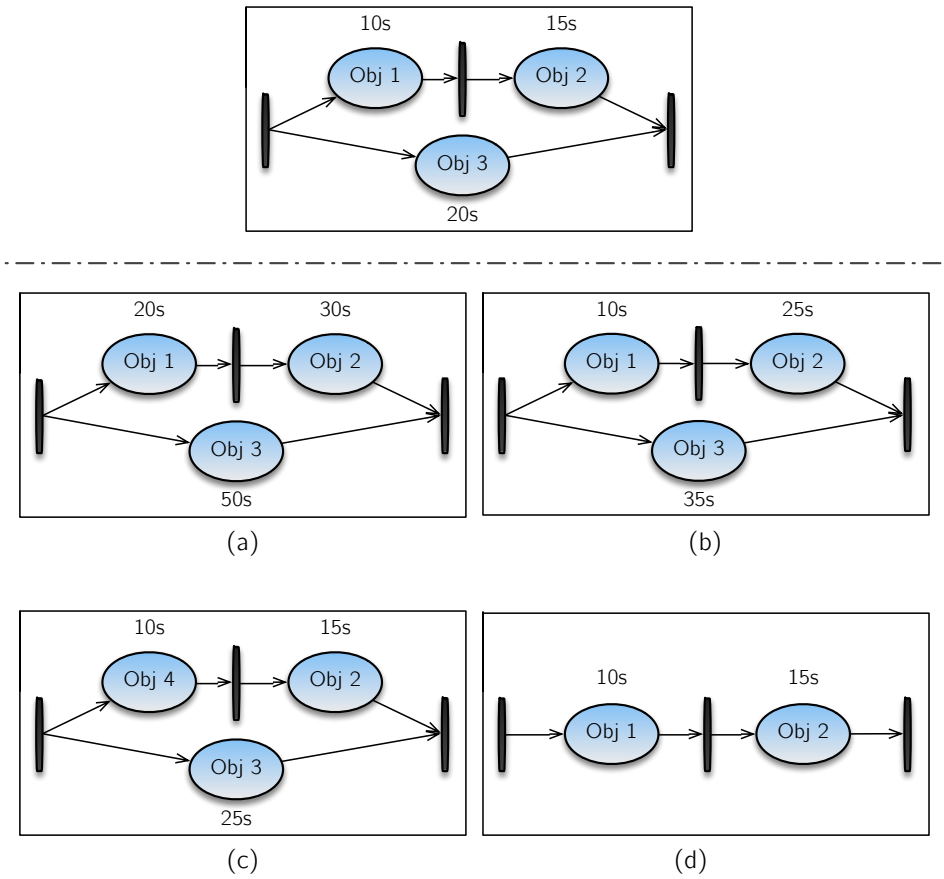


Figura 4.10: Confronto fra OCPNs.

- (a) ha match esatto dal punto di vista degli oggetti e nel caso si voglia mantenere lo stesso rapporto temporale fra le parti è ottimo (cosine similarity ad 1).
- (b) ha match esatto dal punto di vista degli oggetti e si discosta poco dai tempi esplicitati nella query (distanza euclidea bassa).
- (c) ha perfetta corrispondenza temporale ma uno degli oggetti non è quello specificato nella query. La similarità sarà dunque dipendente dalla similarità fra *Obj* 1 ed *Obj* 4.

- (d) si tratta dello stesso multimedia ma senza audio. Si noti che usando un approccio minimo (cioè la minima similarità fra ogni elemento) abbiamo risultato 0 (massima differenza).

4.3 Il testo

È il multimedia più antico ed è spesso presente in altri multimedia. Vediamo ora come trattarlo.

4.3.1 Tipologie di testo

Distinguiamo innanzi tutto in tre tipologie di testo (ed elenchiamo alcune delle caratteristiche proprie):

- Testo non strutturato (testo libero): si può compiere un'analisi stastitica per cercare di rappresentare il testo (si noti che ''a sbatte contro b'' e ''b sbatte contro a'' sono statisticamente identiche seppur non lo siano semanticamente). Quando si effettua una query si tiene conto delle occorrenze esatte della query, delle somiglianze semantiche (''mamma'' e ''madre'') e di quelle sintattiche (''tavolo'' e ''tavola'').

- Testo semi strutturato (XML, SGML): tendenzialmente le query vengono ridotte ad un confronto fra le gerarchie.

- Testo strutturato (linguaggi di programmazione).

4.3.2 Il testo come collezione di keywords

Giacché le query che andiamo ad eseguire sono fondamentalmente del tipo ''data una keyword trova tutti i documenti che la contengono'' la rappresentazione più naturale è quella di associare ad ogni documento del *corpus* (così si chiama l'insieme dei documenti nel database) un vettore che è la lista invertita di tutti i termini che contiene. Questo tipo di rappresentazione non è però sufficiente per rispondere a query che contengono suffissi di keyword (dovrà essere ampliata).
Quindi ad ogni documento è associato un surrogato che è il *vocabolario* (l'insieme delle keyword possibili) e ad ogni keyword di ogni surrogato è associata la frequenza del termine all'interno del documento cui il surrogato appartiene.

Ogni documento è composto da parole più significative rispetto ad altre. Nella fattispecie, gli articoli o le preposizioni sono presenti in ogni testo e sono quindi poco differenzianti. Tali termini vengono detti *stop words* e *non vengono inclusi* nei surrogati poiché poco differenzianti (e appiattirebbero le differenze fra i termini).

Preprocessing

È dunque necessaria una fase di *preprocessing* che consta di:

- Rimozione delle stop words.
- Stemming/Lemmatizzazione: vengono identificate le radici dei termini, quindi ad esempio `mangio`, `mangiare` e `mangiato` diventano `mang`.
- Phrasing: vengono identificati i termini composti (ad esempio `Torri Gemelle` ha un significato diverso dai termini `torri` e `gemelle` considerati separatamente. Più precisamente si ha un phrasing quando due termini, se affiancati, hanno una semantica differente rispetto a quando sono soli. Si rende quindi necessario uno studio della co-occorrenza.

4.3.3 La legge di Zipf

Zipf ha effettuato degli studi in merito alla distribuzione delle parole costituenti il dizionario inglese e ha scoperto una distribuzione *power-law*, come quella di Pareto. Abbiamo quindi una coda pesata sulla quale sono distribuiti la maggior parte dei termini della lingua e poi alcuni di essi, gli estremamente più frequenti, si trovano in un picco. Per eliminare le stop words non dovremo far altro che ''tagliare'' quelle parole che si trovano nel picco iniziale della curva.

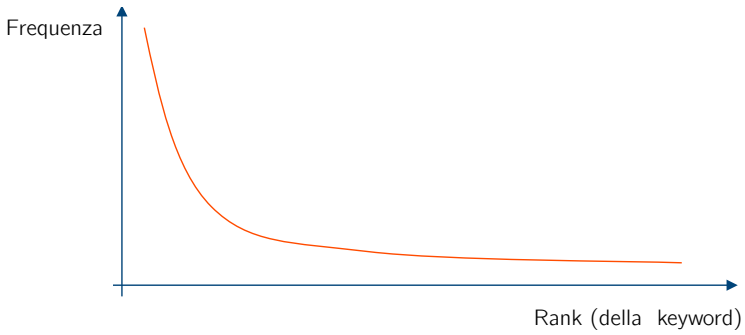


Figura 4.11: Distribuzione Zipfiana.

La frequenza del k-esimo termine più frequente è quindi $\left(\frac{1}{k}\right)^{\theta}$ volte la keyword più frequente, cioè:

$$freq(k_i) = \frac{1}{r_i \ln(1.78 \cdot V)}$$

dove V è la dimensione del vocabolario.

4.3.4 La rappresentazione vettoriale: tf-idf

Come già per le immagini, se immaginiamo che ogni componente (keyword) della lista invertita associata ad un documento sia una dimensione, possiamo avere una rappresentazione vettoriale dei documenti.
Dato quindi un documento d_i nel corpus, avremo un surrogato del tipo:

$$d_i =< w_i[1], w_i[2], ..., w_i[n] >$$

Ogni peso associato alla keyword sarà 0 se la keyword non esiste e un certo valore maggiore di zero se la keyword è presente. Determinare quindi il valore di ogni w_i è fondamentale per avere una misura di similarità sensata.

Term Frequency

La prima idea, la più naturale, è che un termine molto presente in un testo sia più importante di uno meno presente. Perciò il peso $w_i[j]$ per il documento i assumerà valore pari alla frequenza del termine j nel documento i. Questo valore è detto tf (term frequency):

$$tf = \frac{n}{K}$$

dove n è il numero di occorrenze del termine studiato mentre K è il numero totale di termini nel documento.

Quanto differenzia il termine

Un secondo aspetto che vogliamo cogliere è quanto un termine sia proprio del documento, ovvero, quanto differenzi rispetto all'interno database. Se la frequenza di un certo termine è alta ma è alta in ogni documento, allora probabilmente il termine non sarà così significativo (rispetto ad altri meno frequenti nel corpus e che hanno frequenza relativa più bassa nel documento). Si usa per questo la misura idf (inverse document frequency):

$$idf = \log\left(\frac{N}{m}\right)$$

dove N è il numero totale di documenti nel corpus e m è la quantità di documenti dentro i quali la keyword appare.

Combinare le misure

Poiché entrambi gli aspetti risultano fondamentali, la misura adottata (ovvero il peso assunto da un termine nel documento) verrà calcolato usando la $tf\text{-}idf$:

$$tf\text{-}idf = tf \cdot idf = \frac{n}{K} \cdot \log\left(\frac{N}{m}\right)$$

In realtà per motivi computazionali non si manterrà questo valore direttamente nei surrogati, ma si preferisce mantenere un vettore che contiene tutti gli *idf* separatamente, così da poterlo aggiornare più agevolmente all'inserimento di un nuovo documento. Quando poi si dovrà effettuare una query, i valori mantenuti nei surrogati (solo la tf quindi) verranno moltiplicati col corrispondente idf.

Eventualmente può essere utile normalizzare la idf:

$$norm\_tf\text{-}idf = tf \cdot idf = \frac{n}{K} \cdot \frac{\log\left(\frac{N}{m}\right)}{\max idf}$$

4.3.5 Eseguire le query (variante di Salton&Buckley)

Quando si effettua una query su testo è possibile specificare:

- Una keyword soltanto.
- Una lista di keyword, ad esempio ''teoria informazioni entropa''.
- Un documento.

Evidentemente il secondo punto è un caso particolare del terzo, quindi abbiamo soltanto due casistiche.
Nel caso di singola ricerca di una keyword stiamo in sostanza cercando la massima entropia su quel termine, dunque il retrieval è immediato (si prendono i massimi $tf\text{-}idf$ per quel termine).
Ma nel caso in cui si fornisca un documento allora si deve effettuare il calcolo della tf per ogni termine e combinarla con la idf del database al fine di ottenere un surrogato confrontabile con gli altri già nel database. Una soluzione alternativa è stata proposta da Salton&Buckley e mira a far pesare il fatto che i termini contenuti nel documento-query siano molto importanti e quindi la loro presenza all'interno di un altro documento debba essere più rilevante. Perciò la $tf\text{-}idf$ per il documento viene calcolata usando questa variante:

$$tf\text{-}idf_{SB} = \left(0.5 + 0.5 \cdot \frac{\frac{n}{K}}{\max freq}\right) \log\left(\frac{N}{m}\right)$$

In questo modo ogni termine presente nel documento-query ha una $tf\text{-}idf$ di almeno 0.5 (sommato alla effettiva frequenza del termine) e quindi risulta ''più importante''. La variante è applicata su tutti i documenti ma ovviamente solo sui termini che appaiono nella query (in modo che lo 0.5 venga attribuito in modo sensato).
Chiariamo questo fatto con un piccolo esempio. Diciamo di avere un documento-query che contenga due termini L ed R. Poi abbiamo tre documenti nel database per cui la tf (classica) su L ed R ci da (per semplicità mostriamo

l'esempio usando solo la tf, ma il concetto sarebbe identico avendo anche il fattore idf):

- doc_a:
  - $tf(R) = 0.30$
  - $tf(L) = 0$
  - Similarità totale: $0.30 + 0 = 0.30$
- doc_b:
  - $tf(R) = 0$
  - $tf(L) = 0.20$
  - Similarità totale: $0.20 + 0 = 0.20$
- doc_c:
  - $tf(R) = 0.05$
  - $tf(L) = 0.05$
  - Similarità totale: $0.05 + 0.05 = 0.10$

Come vediamo, solamente nel doc_c appaiono tutte e due le keyword della query ma il suo valore di similarità è molto basso. Se invece ricalcoliamo le tf seguendo la variante di Salton&Buckley otteniamo:

- doc_a:
  - $tf(R) = 0.50 + 0.50 \cdot 0.30$
  - $tf(L) = 0$
  - Similarità totale: $0.65 + 0 = 0.65$
- doc_b:
  - $tf(R) = 0$
  - $tf(L) = 0.50 + 0.50 \cdot 0.20$
  - Similarità totale: $0.60 + 0 = 0.60$
- doc_c:
  - $tf(R) = 0.50 + 0.5 \cdot 0.05$
  - $tf(L) = 0.50 + 0.5 \cdot 0.05$
  - Similarità totale: $0.52 + 0.52 = 1.04$

Come si vede, quando la frequenza di un termine è zero perdiamo il contributo di 0.5 e questo pesa fortemente sul calcolo finale. Infatti, la similarità per il doc_c è ora molto più alta (è maggiore di uno poiché i calcoli non sono normalizzati).

4.3.6 Dipendenza fra termini: distanza semantica

Il modello vettoriale assume che i termini siano independenti fra loro... ma lo sono veramente? Ci troviamo in una situazione simile a quella già vista con i colori: avevamo un istogramma che considerava separatamente ogni colore e quindi abbiamo introdotto la distanza Euclidea completa al fine di considerare anche questo fattore. Analogamente ora cerchiamo una misura che sia più fine e che tenga conto dei fattori di correlazioni fra le keywords.

Relazioni semantiche

Oltre alle somiglianze sintattiche (che siano fra prefissi oppure calcolate tramite edit distance) esistono anche delle relazioni di tipo semantico. Le relazioni semantiche si suddividono in due grandi gruppi: dipendenti dal corpus (devono essere ricalcolate per ogni database) ed indipendenti dal corpus (sono relazioni proprie della lingua).

- Relazioni semantiche indipendenti dal corpus:
  - Sinonimie (diversa sintassi ma stessa semantica).
  - Polisemia (un termine ha più di un significato).
  - Iponimia (k_1 è un iponimo di k_2 [chiamato iperonimo] se ogni k_1 è anche un k_2, cioè l'iponimo ha un campo semantico meno esteso rispetto all'iperonimo).
  - Iperonimia (l'inverso della iponimia).
- Relazioni semantiche dipendenti dal corpus:
  - Co-occorrenza di termini.

Calcolo della distanza semantica

Vogliamo quindi tenere da conto la correlazione semantica fra i termini e saper ad esempio quantificare:

- dist(man, woman)
- dist(man, child)
- dist(man, human)

Sono tutti termini molto simili, ma come quantificare questa somiglianza? Si ricorre ad un modello di *distanza semantica*, il cui calcolo può risultare più o meno complesso. Più precisamente, una volta stabilita una tassonomia di dominio è possibile calcolare la distanza semantica attraverso diversi parametri che portano ad un affinamento sempre migliore del concetto di distanza semantica.

Consideriamo questa tassonomia ed andiamo a studiare differenti definizioni di distanza semantica: Si noti che l'impiego di una tassonomia può essere utile

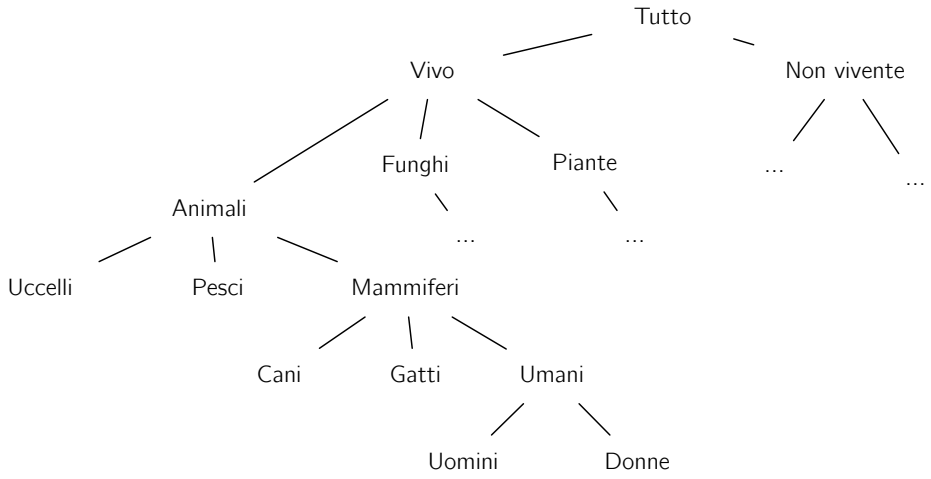


Figura 4.12: Una tassonomia delle keywords.

anche per disambiguare le polisemie: la pesca (intesa come frutto) apparirà in un contesto che presenterà termini molto diversi rispetto a quelli trovabili durante l'uso del termine pesca (intesa come sport). Avendo quindi in mano i termini che appaiono nel documento sarà facile capire di quale pesca si sta parlando, sarà infatti sufficiente un controllo con i termini vicini nella tassonomia ed avremo disambiguato.

Distanza come numero di archi. L'idea più banale (ma anche meno descrittiva) è quella di, dati due termini nella tassonomia, calcolare il numero di archi fra loro e considerare tale valore come distanza fra i due termini:

$$\text{sim(pesce, uomo)} = \text{numero di archi fra pesce ed uomo}$$

Quindi, per esempio, la distanza fra ''pesce'' ed ''uomo'' è di 4.

Distanza di Resnick. La definizione appena introdotta manca di considerare i fattori relativi alla co-occorrenza dei termini e anche la quantità informativa di ogni termine.

$$\text{sim(pesce, uomo)} = \max\{inf_{content}(ca(pesce, uomo))\}$$

Abbiamo che $inf_{content}()$ è una funzione che calcola $\log(1/f)$ dove f è la frequenza del parametro. Il contenuto informativo è quindi assimilato al logaritmo dell'inverso della frequenza con cui quel termine appare. La funzione $ca()$ invece calcola i common ancestors (antenati comuni) dei suoi parametri.

In questo esempio quindi avremmo $ca = \{animali, vivo, tutto\}$.
Ricapitolando: per ogni antenato comune viene calcolato il contributo informativo e poi se ne prende il massimo.

Distanza di Richardson. Questa definizione risulta ancora più sofisticata e cerca di cogliere tutti gli aspetti possibili dei due termini. Ogni arco viene pesato attraverso l'impiego di tre parametri:

- Densità della gerarchia nella zona dove si trova l'arco: il senso è che se la gerarchia è molto densa ci sono molte sfaccettature di concetto e quindi la distanza reciproca fra i termini è poca. Al contrario se abbiamo pochi nodi la differenza fra un nodo e l'altro è molto grande. Quindi ad alta densità corrisponde un basso peso.
- Altezza a cui si trova l'arco: concetto simile a quello esposto nel punto precedente. Un arco di alto livello specificherà concetti più ampi e quindi maggiormente diversi fra loro.
- Il contenuto informativo di ciascuno dei due estremi dell'arco. Tale contenuto è calcolato sempre usando la formula introdotta prima ed il valore assunto dall'arco sarà la differenza fra i contenuti informativi dei due estremi.

Dopo aver pesato tutti gli archi fra i due termini sarà sufficiente calcolare la distanza sull'albero (ovviamente sfruttando i pesi appena introdotti).

4.3.7 Dipendenza fra termini: concept vector space

Quanto definito fino ad ora funziona perfettamente ma è scomodo: il calcolo del cammino può essere dispendioso e sarebbe più interessante poter includere direttamente in ogni termine (nodo) tutte le informazioni relative alla sua correlazione semantica con gli altri nodi della tassonomia. Si vuole dunque ottenere un vettore (detto *concept vector*) che rappresenti il concetto contestualizzato nella tassonomia in cui si trova. Consideriamo questa tassonomia geografica:
Abbiamo un totale di 9 nodi. Vorremmo che ad ognuno di essi fosse associato un vettore di 9 elementi che descriva la correlazione di ogni termine con tutti gli altri. L'algoritmo che ci permette di ottenere questo risultato è chiamato *CP/CV* (concept propagation/concept vectors) ed è un algoritmo iterativo. Si parte configurando ogni vettore con un valore di 1 per il nodo stesso e di 0 per tutti gli altri. Dopodiché ogni nodo propaga i propri valori verso l'alto e verso il basso, modificando gli array dei suoi antenati e figli (a distanza uno). La propagazione viene regolata tramite un parametro α, detto coefficiente di propagazione. L'algoritmo termina quando i valori tendono a non muoversi più.

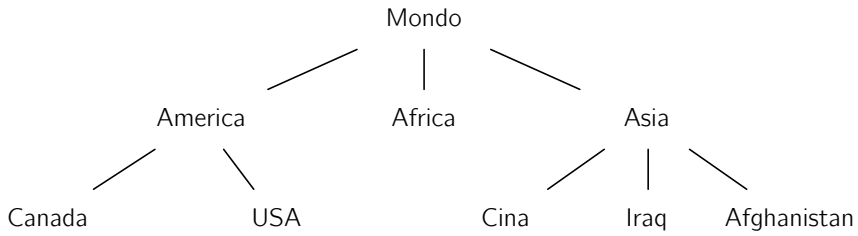


Figura 4.13: Tassonomia geografica.

La disparità impostata a inizio algoritmo (settando ad 1 il campo relativo al nodo stesso nel concept vector) farà sì che un concetto della tassonomia sia sempre più ''se stesso'' che altre cose (quella componente rimarrà quella più alta).
L'applicazione dell'algoritmo fornisce questo risultato:

	world	Asia	Africa	America	Afghanistan	Iraq	China	Canada	US
$\vec{cv}_{world}$	0.450	0.169	0.141	0.158	0.018	0.018	0.018	0.021	0.021
$\vec{cv}_{Asia}$	0.052	0.469	0.006	0.006	0.156	0.156	0.156	0.0003	0.0003
$\vec{cv}_{Africa}$	0.100	0.012	0.873	0.012	0.0006	0.0006	0.0006	0.0007	0.0007
$\vec{cv}_{America}$	0.057	0.007	0.007	0.520	0.0003	0.0003	0.0003	0.204	0.204
$\vec{cv}_{Afghanistan}$	0.004	0.100	0.0002	0.0002	0.872	0.012	0.012	0	0
$\vec{cv}_{Iraq}$	0.004	0.100	0.0002	0.0002	0.012	0.872	0.012	0	0
$\vec{cv}_{China}$	0.004	0.100	0.0002	0.0002	0.012	0.012	0.872	0	0
$\vec{cv}_{Canada}$	0.006	0.0003	0.0003	0.165	0	0	0	0.806	0.023
$\vec{cv}_{US}$	0.006	0.0003	0.0003	0.165	0	0	0	0.023	0.806

Figura 4.14: Risultati dell'algoritmo CP/CV sulla tassonomia d'esempio.

Possiamo notare come ad esempio i valori dell'africa siano poco condizionati poiché questa non ha figli. Possiamo anche notare che all'aumentare del cammino fra due termini la loro correlazione diminuisce. Teoricamente non è possibile avere correlazione 0 (se i termini fanno parte della tassonomia sono in qualche senso, anche minimo, correlati), ma quando il valore è molto piccolo si può approssimare alla non correlazione.

4.3.8 Dipendenza fra termini: coefficiente di correlazione

Per quanto riguarda la correlazione termine a termine all'interno del corpus è possibile calcolare il coefficiente di correlazione. Siano i ed j due diversi termini; il loro coefficiente di correlazione c è definito come:

$$c_{i,j} = \frac{n_{i,j}}{n_i + n_j - n_{i,j}}$$

dove $n_{i,j}$ è la probabilità che i due termini appaiano nello stesso documento. La formula è quindi un classico casi favorevoli su casi totali: dividiamo la probabilità che i due termini appaiano insieme per la probabilità che i appaia

nel documento (da solo o insieme a j), più quella che j appaia nel documento (da solo oppure con i) e sottriamo quella che appaiano insieme (al fine di ottenere la probabilità che appiano separatamente).
L'uso di questo coefficiente è analogo a quello che abbiamo adottato per il coefficiente di cross talk per i colori.

4.4 Altri modelli

Accenniamo ad altri tre modelli per rappresentare/calcolare la similarità del testo.

4.4.1 Extended boolean model

Il modello booleano esteso permette di capire quale fra due termini sia più generale dell'altro. Ad esempio, ''animali da compagnia a pelo lungo'' e ''animali domestici'': qual è il più generale? Qual è il più specifico?
È importante capire che i due concetti *non sono uno il contrario dell'altro*, quindi non è vero che un termine è generale tanto più non è specifico e viceversa.
Il modello è quindi utilizzabile in due modi diversi: per comprendere la specificità di un termine o per comprenderne la generalità. Ma non è possibile da uno dei due risultati derivare l'altro, bisogna applicare da capo il metodo due volte per ottenere la risposta a entrambi i quesiti.
Il modello si basa su una semplice metafora:

- Se un oggetto è caratterizzato nei termini di OR delle sue caratteristiche, allora è un oggetto *generico* (ad esempio se dico che un veicolo o è a motore o è a pedali, sto parlando di un veicolo molto generico).
- Se un oggetto è caratterizzato nei termini di AND delle sue caratteristiche, allora è un oggetto *specifico* (ad esempio se dico che un veicolo è a motore e a pedali, sto parlando di un veicolo molto specifico).

Se rappresentassimo su un grafico i due concetti ''motore'' e ''pedali'' allora all'origine avremmo il vettore [0, 0], in altre parole ''no pedali e no motore'' che applicando De Morgan ci dice $\neg(pedali \vee motore)$. Ma giacché la semantica che abbiamo attribuito all'OR è quella della generalità (e c'è un not davanti), stiamo dicendo che l'origine è il massimo di *non generalità*, quindi un oggetto sarà tanto più generale quanto è *distante dall'origine*.
Allo stesso modo, potremmo considerare il vettore [1, 1] che corrisponde a ''pedali e motore'', ovvero $pedali \wedge motore$. Abbiamo quindi che il punto [1,1] rappresenta la massima specificità. Quindi, tanto più siamo vicini al punto [1,1] e tanto più il nostro concetto sarà *specifico*.
Riassumiamo quanto detto in queste due figure:

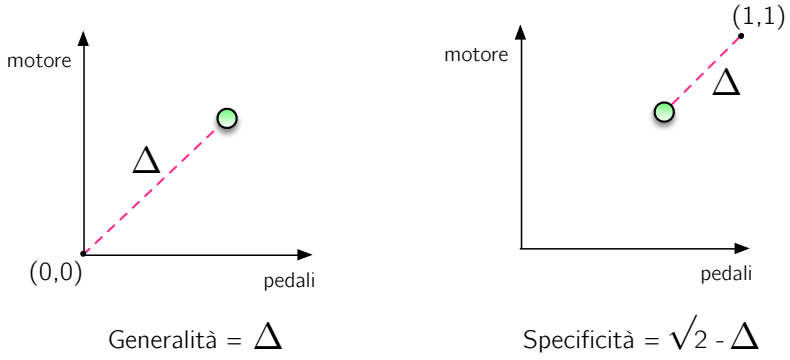


Figura 4.15: Rappresentazione grafica dell'extended boolean model.

4.4.2 Modelli probabilistici

Data una query ed un documento si stima la probabilità che l'user troverà il risultato interessante. Si assume quindi che esista un set ideale di cui dobbiamo stimare in qualche modo le proprietà.
Torneremo a parlare di questo modello quando tratteremo il relevance feedback.

4.4.3 Modello fuzzy

Il modello fuzzy associa ad ogni query un insieme fuzzy che contiene i dati più plausibilmente in grado di soddisfare l'utente. I vari insiemi possibili non sono però disgiunti (ed è la caratteristica che contraddisingue i fuzzy set dai modelli probabilistici).
Ad esempio: data una query con vari termini k_i e considerato un documento d_j, qual è la plausibilità che tale documento venga incluso nell'insieme fuzzy della query? La risposta è calcolabile come segue:

$$u_{i,j} = 1 - \prod_{k_l \in d_j} (1 - c_{i,l})$$

Quello che facciamo con la produttoria è scorrere l'intero documento d_j e per ogni keyword in esso contenuta (k_l) calcoliamo la co-occorrenza fra la keyword della query k_i e la keyword k_l (in simboli $c_{i,l}$). Facendo poi 1 - tale valore otteniamo la probabilità che k_l e k_i non co-occorrano nel documento e dunque la moltiplicazione di tutte queste probabilità ci da la probabilità che nessuna chiave contenuta nel documento co-occorra con quella della query. Ma all'esterno della produttoria abbiamo ancora 1 - e quindi in totale abbiamo la probabilità che *qualche chiave* co-occorra con k_i.
Anche questo modello verrà sfruttato in seguito (per il processing delle query).

Capitolo 5

Trovare gli oggetti

Ora che conosciamo quali feature possiamo considerare in fase di progettazione del database, vogliamo introdurre delle tecniche per memorizzare tali oggetti e renderli recuperabili nel minor tempo possibile. Data quindi una query e un set di oggetti nel database, come *troviamo gli oggetti* che matchano?

5.1 Tre grandi tecniche

Esistono tre grandi tecniche complementari fa loro per adempiere al nostro obiettivo di rendere effiacace l'uso delle feature:

- Classification.
- Clustering.
- Indexing.

Le tecniche usate nella realtà sfruttano i principi delle tre appena elencate in modo da combinarne i punti forti ed ottenere il miglior risultato possibile.

Ci concentreremo principalmente sull'indexing (prima del testo e poi di oggetti multimediali in generale) ma faremo anche alcuni accenni alla classificazione e alla clusterizzazione.

5.1.1 Classification

Una *classe* è un insieme di oggetti che condividono *proprietà comuni*. Perciò, una volta stabilita una o più proprietà è possibile scorrere l'intero database e inserire ogni oggetto in una classe diversa a seconda delle proprietà di cui si sta tenendo conto. Se ad esempio vogliamo classificare per età stiamo dicendo che la proprietà rilevante è appunto l'età di ogni individuo e quindi avremo la classe dei 20enni, dei 30enni, ecc. Ogni oggetto apparterrà o meno ad una classe a seconda della sua proprietà. Vediamo questo esempio:

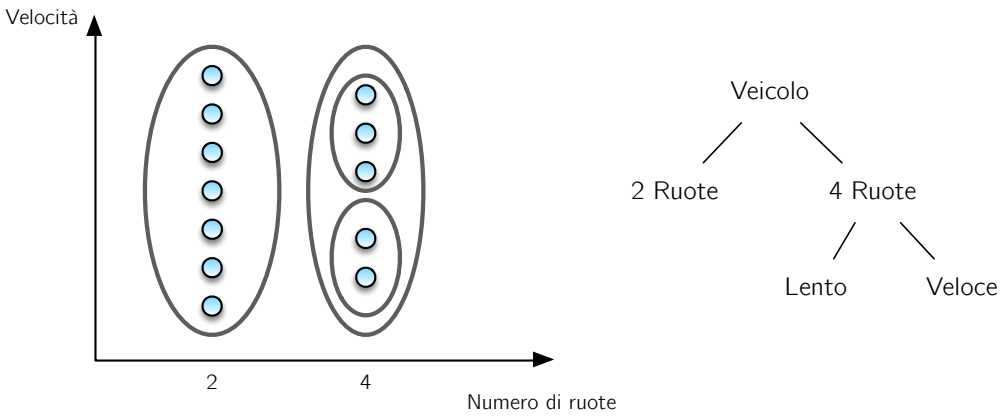


Figura 5.1: Esempio di classificazione.

Abbiamo classificato i veicoli secondo due proprietà (in ordine): prima il numero di ruote e poi la velocità a cui possono andare.

5.1.2 Clustering

Il *clustering* è una tecnica che raggruppa gli oggetti per *somiglianza*. Tutti gli oggetti appartenenti ad un cluster sono quindi *omogenei fra loro*. È importante capire la differenza fra classificazione e clustering: nel primo caso le proprietà sono definite a monte mentre nel secondo l'omogeneità è di carattere globale e non puntuale. Non avremo quindi un cluster di persone di età maggiore di 20 anni, ma avremo un cluster di persone di età simile fra loro, qualunque essa sia. Il criterio di clustering è dunque indotto dai dati e dal contesto in cui si trovano. Vediamo questo esempio:

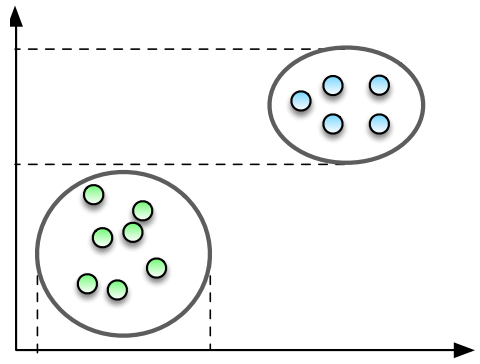


Figura 5.2: Esempio di clustering.

Come vediamo sono presenti due cluster. Tanto più grande sarà la distanza

fra i cluster e tanto più saranno disomogenei gli elementi fra i due insiemi (l'omogeneità interna è invece rappresentata dall'ampiezza del cluster).
Avere cluster il più omogenei è fondamentale in fase di ricerca: ad ogni cluster viene associato un *rappresentante* che verrà sfruttato per fare i confronti in sede di interrogazione. Si capisce dunque che più il rappresentante sarà rappresentativo (cioè omogeneo rispetto agli altri elementi del cluster) e più l'interrogazione potrà basarsi su dati precisi e quindi restituire una risposta migliore.

5.1.3 Clustering vs classification

Come abbiamo già detto il cluster e la classification sono tecniche diverse e possono portare a risultati eventualmente differenti:

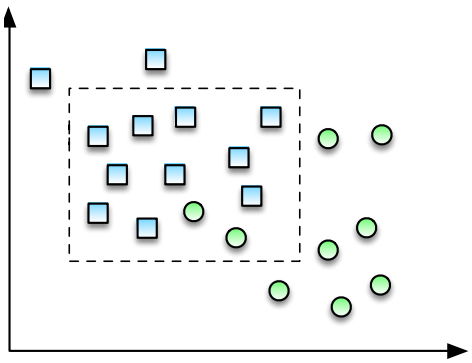


Figura 5.3: Clustering vs classification.

In questo esempio abbiamo rappresentato la classificazione in due insiemi (con simboli diversi) e il clustering con una linea tratteggiata: come vediamo nello stesso cluster sono inclusi oggetti di classi differenti. Questo è un fatto naturale se pensiamo che la semantica delle due tecniche è simile ma comunque diversa.

5.1.4 Indexing

L'indexing si basa su specifiche strutture dati che sfruttano la loro morfologia per guidare la ricerca dell'oggetto evitando di intraprendere strade che sappiamo già essere sbagliate (si evita quindi una lettura sequenziale). L'indice di un libro, ad esempio, ci permette di evitare di leggere tutto il testo ma grazie ad una consultazione veloce ci rimanda direttamente alla pagina di interesse.
La struttura tipica sulla quale gli indici si basano è l'albero. Vediamo un esempio di albero nella Figura 5.4.

5.2 Tipiche strutture ad indice (alberi)

Un *albero* è un grafo diretto connesso dove ogni nodo (tranne la radice) possiede di un arco entrante al massimo.

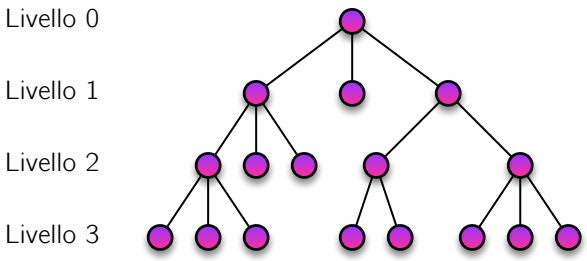


Figura 5.4: Esempio di albero.

Quando si parla di alberi vi sono alcune caratteristiche fondamentali che possono interessarci:

- Alberi *bilanciati*: sono alberi che hanno tutte le foglie sullo stesso livello. Si può definire bilanciato anche un albero che ha tutte le foglie sullo stesso livello accettando però alcune foglie ad un livello superiore/inferiore (quindi con tolleranza di un livello). Tale definizione è necessaria quando la struttura non è sufficientemente piena per poter completare il livello più basso per intero.

- Alberi k-ari: ogni nodo ha al massimo k figli uscenti (tale parametro è detto *fanout*).

È fondamentale conoscere questi parametri poiché dato un albero k-ario bilanciato con N nodi possiamo conoscerne l'altezza, cioè $O(\log_k(N))$. L'altezza dell'albero è sua volta un parametro fondamentale poiché molte delle strutture a indice (non tutte quelle che vedremo, però) permettono di arrivare ai dati cercati tramite un path dell'albero (dalla radice alle foglie).

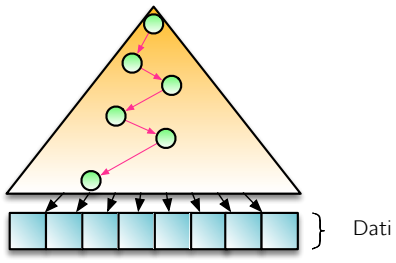


Figura 5.5: Albero come indice.

Dunque il valore dell'altezza risulta essere il costo di interrogazione. Si noti che anche la dimensione delle pagine è un parametro fondamentale: se le pagine sono molto capienti potremo mettere più puntatori (elementi) nello stesso nodo e quindi abbassare l'altezza dell'albero riducendo di fatto i costi di ricerca.

Facciamo ora una carrellata delle varie tipologie di alberi usabili per indicizzare i nostri oggetti. Su alcuni andremo più velocemente (sono indici molto conosciuti) mentre su altri ci soffermeremo di più.

5.2.1 Search tree

Un albero di ricerca è un albero bilanciato che mantiene un valore in ogni suo nodo. Tale valore permette di suddividere lo spazio di ricerca in due parti: poiché i dati sono ordinati, scendere da un lato o dall'altro esclude automaticamente tutti i valori nel ramo non selezionato.

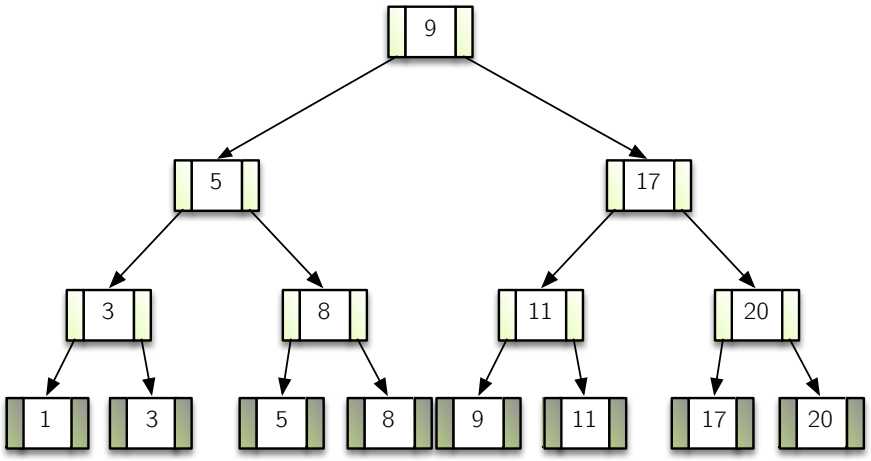


Figura 5.6: Esempio di search tree.

Il problema fondamentale dei search tree sono i costi di mantenimento: aggiungere e rimuovere nodi è costoso poiché è importante mantenere la struttura sia coerente e sia bilanciata. L'assenza di bilanciamento può portare ad avere un ramo degenere e ridurre i costi di ricerca a $O(N)$ (correre una lista in pratica).

5.2.2 B-tree

I B-tree sono alberi i cui nodi (tutti tranne la radice) devono soddisfare due condizioni:

- Condizione di *riempimento*: $n/2$.
- Condizione di *massima capacità*: n.

Vediamo un esempio di B-tree con $n = 4$.

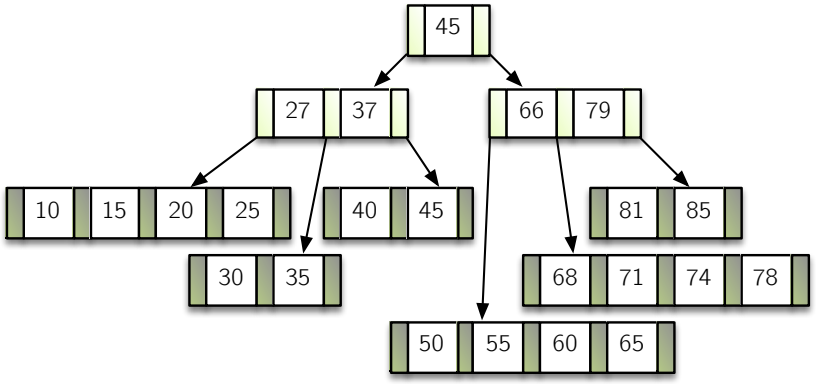


Figura 5.7: Esempio di B-Tree.

Ogni nodo dispone quindi di un numero di informazioni che varia da $n/2$ fino ad un massimo di n.

5.3 Strutture per lo scan del testo

Diciamo di avere una query composta da alcune keyword: vorremmo recuperare tutti i documenti in cui quelle keyword appaiono.
Le strutture che abbiamo appena visto possono aiutarci in questo compito: possiamo infatti popolare i nostri alberi con i termini e poi restituire una *lista invertita*, cioè una lista di documenti in cui quel termine appare. Possiamo sfruttare quindi i B-Tree che abbiamo appena visto, oppure l'hashing oppure ancora particolari alberi detti *trie*. Studiamo nel dettaglio quest'ultimi, precisamente, definiamo i *trie per prefissi* e gli *alberi per suffissi*.

5.3.1 Prefix trie

Introduciamo questa struttura dati mentre ne vediamo l'applicazione per la memorizzazione dei prefissi delle parole contenute in un documento.

Definito un alfabeto di simboli possibili che le parole possono contenere (diciamo le 21 lettere dell'alfabeto italiano, per esempio), definiamo ogni nodo del trie come una lista di 21 puntatori, uno per ogni simbolo. Supponiamo di avere soltanto il nodo radice (senza puntatori valorizzati) e di voler indicizzare tutti i documenti che contengono la parola ''cane'': sarà sufficiente creare quattro nodi e collegarli in modo da poter ricomporre la parola con una visita in profondità del trie. Al fondo di questa catena di nodi (alla lettera ''e'' dell'ultimo nodo) andremo ad inserire la lista invertita dei documenti ove ''cane'' compare.

In questo esempio vediamo un trie che indicizza i documenti contenenti le parole ''cane'' e ''canestro'':

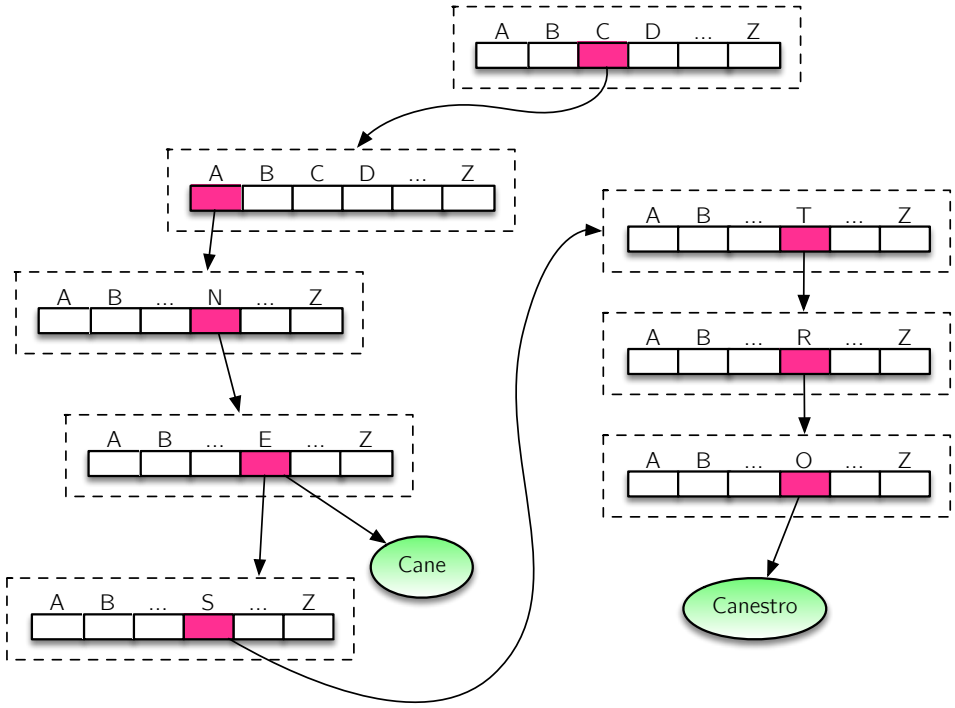


Figura 5.8: Esempio di trie.

Come vediamo la cella corrispondente alla lettera ''e'' ha un doppio puntatore: uno per la lista invertita della parola ''cane'' e uno per proseguire l'indicizzazione di ''canestro''.
La struttura è detta *prefix trie* poiché è necessario conoscere il prefisso della parola (fino a quanto sappiamo ora ci serve tutta, ma come stiamo per vedere può bastare il prefisso) per poterla cercare all'interno del trie.

Esiste anche una versione ottimizzata di trie (detta *Patricia trie*) che, nel caso in cui non ci siano conflitti di più parole (cioè prefissi comuni) su un path, invece di memorizzare l'intero path comprime la strada nell'ultimo nodo e linka direttamente alla lista invertita (nell'esempio di sopra avremmo compresso gli ultimi tre nodi nel quart'ultimo e avremmo linkato direttamente a ''canestro'').

Il costo di ricerca all'interno del trie è potenzialmente pari alla lunghezza della parola più lunga del database.

5.3.2 Suffix tree

Non sempre cerchiamo parole per prefisso, oppure non conosciamo esattamente il prefisso del termine che vogliamo individuare, oppure ancora non esiste l'inizio della stringa (ad esempio in una sequenza di DNA). In questi casi si adottano le ricerche su *suffix trie*.
Il suffix tree potrebbe essere costruito basandoci su tutti i possibili suffissi della stringa che vogliamo indicizzare, ma tali suffissi sono potenzialmente tanti quanti le lettere della stringa stessa:

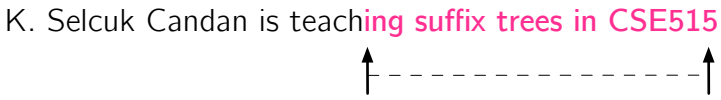


Figura 5.9: A partire da ogni lettera definiamo un suffisso.

Questo di per sé non rappresenta un problema, ma per semplicità noi consideremo valido un suffissoche parte con una parola di senso compiuto (quindi il numero di suffissi è pari al numero di parole nella stringa):

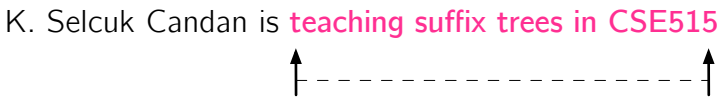


Figura 5.10: A partire da ogni parola definiamo un suffisso.

Consideriamo quindi questa stringa d'esempio ed andiamo a costruire il suffix tree corrispondente prima di tutto numerando ogni suffisso (non consideriamo le stop word):

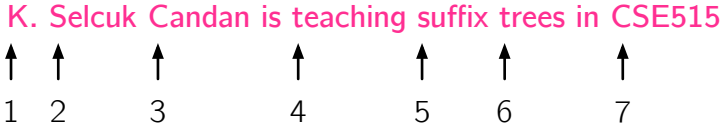


Figura 5.11: Numeriamo i suffissi.

Poi a partire dalla root definiamo tanti figli quanti sono i prefissi disgiunti dei suffissi che vogliamo indicizzare (quindi in questo caso 4). Poi, con una tecnica simile a quella già vista per i prefix trie, andiamo a costruire l'indice col fine di discriminare le sovrapposizioni fra i suffissi e linkare a fondo albero i suffissi per intero. Si noti che l'ordinamento dei figli non è casuale: se infatti mettiamo i vari nodi in ordine alfabetico possiamo poi rappresentare le foglie dell'albero come un array ed effettuarvi una ricerca binaria per un retrieval ancora più veloce.

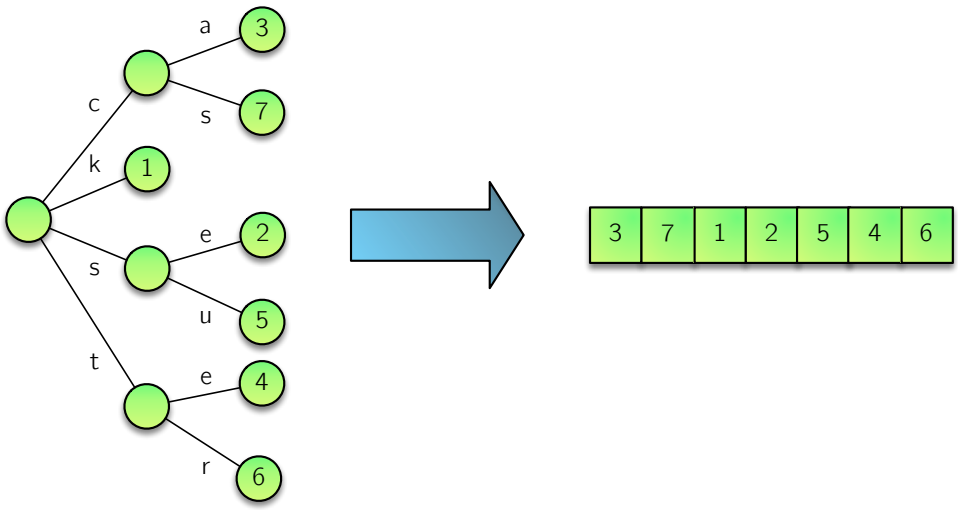


Figura 5.12: Suffix tree e array corrispondente.

In figura vediamo il suffix tree generato per indicizzare la nostra stringa d'esempio (anche questo è ottimizzato come Patricia trie, cioè non si porta dietro path senza sovrapposizioni ma linka direttamente al suffisso quando può).
Se ora cerchiamo ''Candan'' (una parola a metà stringa) seguiamo il ramo per la ''c'', poi quello per la ''a'' e ci viene ritornato 3, puntatore al suffisso ''Candan is teaching suffix trees in CSE515''.

5.3.3 Senza prefissi e senza suffissi

Mettiamoci ora in una situazione ancora più difficile: diciamo di cercare una serie di lettere all'interno di un testo, ma non sono né il prefisso né il suffisso di una parola. Il nostro obiettivo è quello di individuare tutte le occorrenze della parola *scorrendo una sola volta il file* in maniera sequenziale.

Se procedessimo in maniera naive (brute force) cercando un pattern lungo M in un testo lungo N avremmo un costo pari ad $O(M \cdot N)$, questo perché durante la ricerca faremmo una sorta di *convoluzione*, leggendo le lettere da blocchi di M avanzando di una lettera alla volta: di fatto rileggiamo molte volte le stesse lettere.

Dovremmo cercare, invece, di sfruttare le informazioni lette man mano anche se non ci hanno portato al match con la keyword. Per fare questo si *costruisce una struttura sulla keyword nella query*, con l'intento di formalizzare i pattern ricorrenti all'interno della query stessa. Questa tecnica è chiamata Knuth-Morris-Pratt (dal nome degli autori) e garantisce una complessità di $O(N)$, cioè di uno scan sequenziale soltanto.

Prendiamo questo esempio:

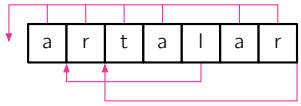


Figura 5.13: Keyword espressa nella query.

Usiamo ora questo sistema per individuare le due occorrenze di ''artalar'' all'interno del documento:

`brartalartalar`

I passi sono molto semplici (con α indichiamo a quale punto dello schema di sopra siamo arrivati e con ξ indichiamo a che punto siamo del documento):

- Passo 1: $\alpha =$ `a`, $\xi =$ `b`. `a` $\neq$ `b`, quindi procediamo aumentando il puntatore sul documento ma continuiamo a cercare `a`.
- Passo 2: $\alpha =$ `a`, $\xi =$ `r`. `a` $\neq$ `b`, quindi procediamo aumentando il puntatore sul documento ma continuiamo a cercare `a`.
- Passo 3: $\alpha =$ `a`, $\xi =$ `a`. `a` $=$ `a`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `r`).
- Passo 4: $\alpha =$ `r`, $\xi =$ `r`. `r` $=$ `r`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `t`).
- Passo 5: $\alpha =$ `t`, $\xi =$ `t`. `t` $=$ `t`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `a`).
- Passo 6: $\alpha =$ `a`, $\xi =$ `a`. `a` $=$ `a`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `l`).
- Passo 7: $\alpha =$ `l`, $\xi =$ `l`. `l` $=$ `l`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `a`).
- Passo 8: $\alpha =$ `a`, $\xi =$ `a`. `a` $=$ `a`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema (ora cerchiamo `r`).
- Passo 9: $\alpha =$ `r`, $\xi =$ `r`. `r` $=$ `r`, quindi procediamo aumentando sia il puntatore sul documento che quello sullo schema.

Durante l'esecuzione del passo 10 capiamo il vantaggio portato dal nostro sistema. Infatti, su ξ abbiamo una `t` e giacché il riconoscimento di `artalar` è concluso, dovremmo ricominciare da capo (riconoscimento che fallirebbe poiché `t`, non è `a`). Ci perderemmo dunque una occorrenza della keyword,

poiché quella appena rilevata condivide le ultime due lettere con la prossima da rilevare. Ma grazie allo schema non ricominciamo da capo; c'è un puntatore a fondo schema che ci riporta sulla keyword, precisamente ci prima della `t` (ci permette quindi di leggerne una) e spostarci di nuovo in attesa della `a` (abbiamo riconosciuto quindi le due lettere `ar`, poi la `t` e ora siamo pronti al passo 11 per trovare una `a`).
In questo modo si legge il documento una volta soltanto ma si garantisce di trovare ogni occorrenza della keyword. Se invece avessimo usato l'approccio brute force avremmo comunque rilevato entrambe le occorrenze ma avremmo ogni volta provato di lettera in lettera a procedere di M e vedere se la keyword aveva match con la finestra di M lettere considerate al momento. Dovendo fare questa operazione spostando la finestra di un carattere alla volta e avendo N caratteri, avremmo avuto il costo $O(M \cdot N)$ di cui abbiamo parlato prima.

Usare più keyword

Esiste la possibilità di creare trie appositi (Aho-Corasick trie) anche per query con più di un termine, ad esempio vediamo questa per due termini (''artalar'' e ''tall''):

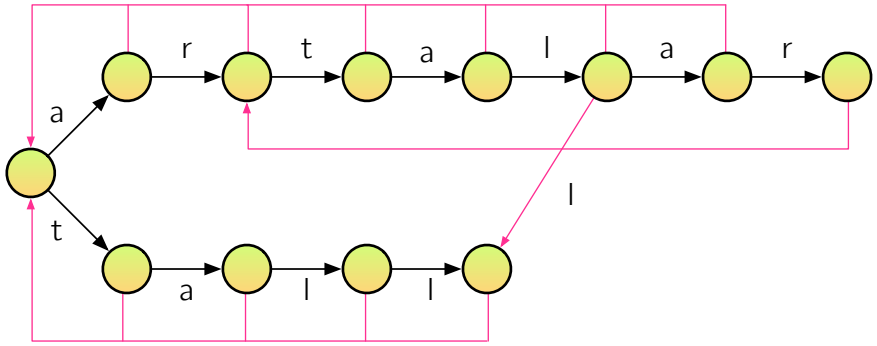


Figura 5.14: Esempio di Aho-Corasick trie.

Il funzionamento è del tutto analogo rispetto a quanto spiegato in precedenza, ma il trie tiene ovviamente conto delle proprietà strutturale di entrambe le keyword. Il costo computazionale è anche qui $O(N)$.

5.3.4 Accettare il mismatch

Tutte le tecniche viste fin ora richiedevano *esattezza* della keyword rispetto quanto richiesto. Ma se volessimo invece imporre una certa tolleranza? Ovvero, se volessimo trovare tutte le parole corrispondenti a ''casa'' concedendo la possibilità di commettere un errore di battitura? L'inesattezza delle query o dei dati è molto frequente, basti pensare alle lettere accentate che vengono comunemente scambiate con la lettera affiancata all'apostrofo.

Andiamo a costruire un automa che sia tollerante in questo senso, partendo da uno che non tolleri nessun errore ed incrementalmente aggiungiamo tolleranze di tipo diverso.
Vogliamo riconoscere la parola `selcuk`.

Fino a 0 errori. Non tolleriamo errori.

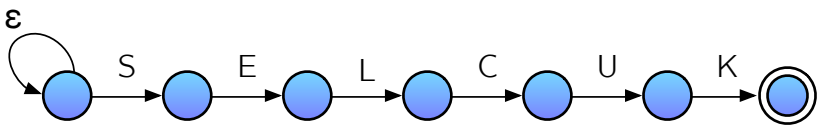


Figura 5.15: Tolleranza: 0 errori.

Fino a 1 inserimento. Tolleriamo un errore di inserimento.

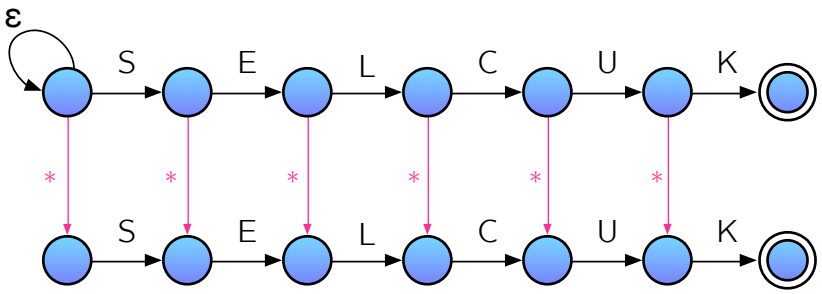


Figura 5.16: Tolleranza: 1 inserimento.

Fino a 1 inserimento/sostituzione. Tolleriamo un errore di inserimento o di sostituzione.

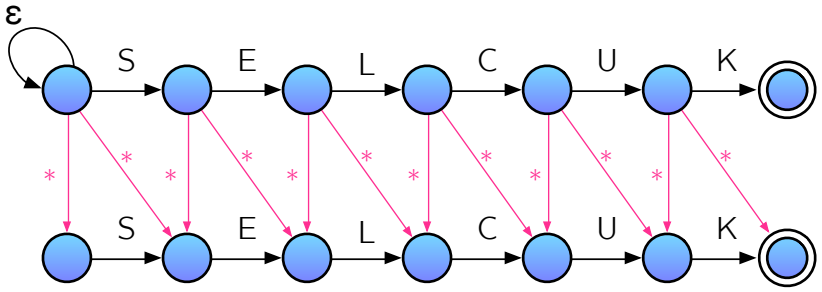


Figura 5.17: Tolleranza: 1 inserimento o sostituzione.

Fino a 1 inserimento/sostituzione/cancellazione. Tolleriamo un errore di inserimento o di sostituzione o di cancellazione.

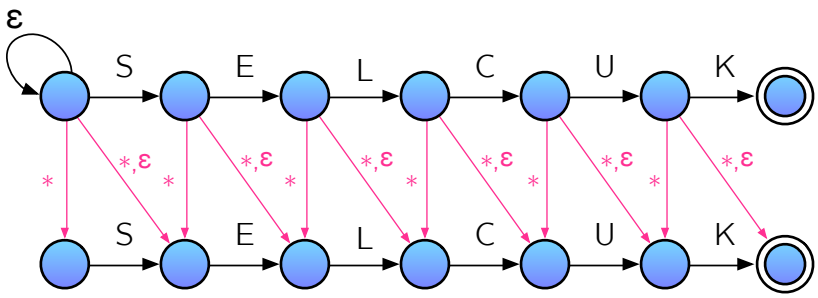


Figura 5.18: Tolleranza: 1 inserimento o sostituzione o cancellazione.

Aggiungendo le epsilon-mosse abbiamo permesso gli errori di cancellazione, aggiungendo le star-mosse in verticale abbiamo permesso l'inserimento ed inserendole in diagonale abbiamo permesso la sostituzione (nel senso di inserimento di un carattere estraneo). Tutto questo per una volta sola, però. Se vogliamo permettere questi errori per n volte dovremo replicare l'automa n volte (oltre alla prima). Quindi, volendo permettere due errori (di inserimento, sostituzione o cancellazione) useremo questo automa:

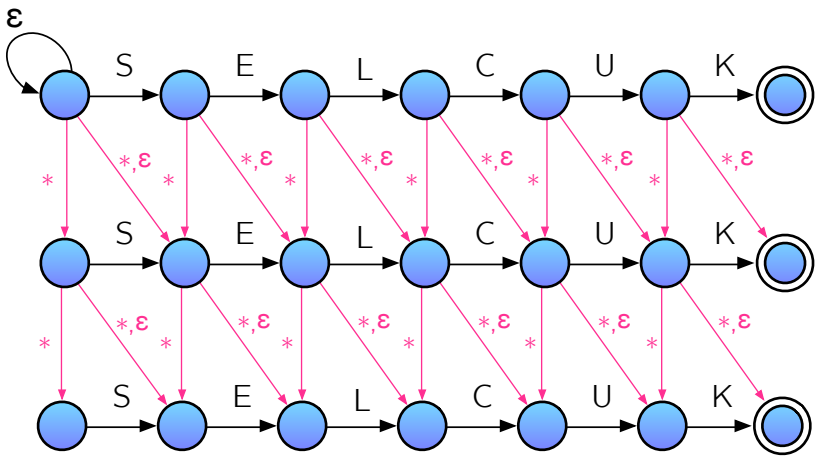


Figura 5.19: Tolleranza: 2 inserimenti o sostituzioni o cancellazioni.

5.3.5 Approximate string matching

Un'altra metodologia per fare matching di stringhe ''approssimativamente'' uguali, è l'approximate string matching. Date due stringhe, possiamo calcolarne la distanza tramite questo semplice algoritmo ricorsivo.

Definiamo come $C[i,j]$ il numero di errori (la distanza) che c'è fra $P_{1...i}$ e $T_{1...j}$ dove P e T sono due stringhe e i e j sono indici sui caratteri. Quindi stiamo dicendo che $C[i,j]$ indica il numero di operazioni da compiere (siano queste cancellazioni, sostituzioni o inserimenti) per far sì che i primi i caratteri di P siano uguali ai primi j caratteri di T. Dato questo formalismo, vediamo il passo base ed il passo ricorsivo dell'algoritmo.

Passo base. Abbiamo:

- $C[0,j] = j$
- $C[i,0] = i$

Banalmente stiamo dicendo che la differenza fra una stringa lunga 0 e una lunga i è sicuramente i (tutte le insert degli i caratteri) e analogamente per la j.

Passo ricorsivo. Abbiamo una condizione da verificare:

```
1  if(P_i = T_j) {
2    C[i, j] = C[i-1, j-1]
3  } else {
4    C[i,j] = 1 + min{C[i-1, j], C[i, j-1], C[i-1, j-1] }
5  }
```

Listing 5.1: Algoritmo di approximate string matching.

L'if controlla se i due caratteri sui due indici siano uguali: se lo sono, allora il costo per avere sia il carattere i di P che il carattere j di T uguali è zero (lo sono già), dunque la distanza è la stessa del passo precedente.
Se invece l'if non è vero, allora abbiamo una differenza (quindi facciamo un +1) e poi dobbiamo aggiungerci il costo precedente. Tale calcolo si effettua prendendo il minimo fra $C[i-1,j]$ che rappresenta il costo di inserimento, $C[i,j-1]$ che rappresenta la cancellazione e fra $C[i-1,j-1]$ che rappresenta la sostituzione. È una operazione poco intuitiva, ma facendo l'esempio dell'inserimento, si considera $C[i-1,j]$ poiché questo rappresenta il fatto che vi sia un carattere in più (j) in T rispetto a quanto siamo arrivati in P (dove siamo ancora ad $i-1$). Cioè la differenza fra P e T è di un inserimento, dunque effettuiamo la chiamata ricorsiva per conoscere quella distanza. Si procederà quindi con le chiamate ricorsive fino ad arrivare al caso base.

Il costo risulta essere $O(M \cdot N)$.

5.3.6 Signature files

L'ultima tecnica che vediamo per lo scan di file alla ricerca di keywords è una tecnica che si basa sulle *signature files*, particolari codifiche in bit associate ai

documenti. L'idea di questo metodo è molto simile a quanto si fa coi filtri di bloom.
L'intenzione è quella di *limitare il numero di documenti* da leggere sequenzialmente, quindi esiste la possibilità di false hit (che devono essere quindi post processati). Il vantaggio è che la tecnica è incredibilmente veloce, quindi fornisce una scrematura a poco costo. Tecniche di questo tipo vengono definite ''quick and dirty'' in riferimento al fatto che il risultato sarà ancora da ripurile. Vediamo dunque come funziona questo sistema.

Generare la signature

Abbiamo una funzione di hash (H) che data una parola (contenuta in un documento o arrivata da una query) la traduce in una serie di bit. Parametri fondamentali (come scopriremo) sono il numero di bit in cui una keyword è tradotta e quanti di questi sono a 1; per il momento supponiamo siano valori fissi.
Si prende dunque ogni documento del database e si fa passare ognuno dei termini in esso contenuti dentro la funzione di hash. Si otterrà quindi un grosso insieme di blocchi di bit, uno per ogni keyword. Tutti questi blocchi vengono messi in bit-OR (quindi uno ad uno), ottenendo quindi una sola sequenza di bit lunga esattamente quanto tutte quelle che abbiamo generato con la funzione di hash.
Vediamo un piccolo esempio (un documento con sole due parole):

```
1 Kasim
2 Selcuk
```

Listing 5.2: Esempio di documento.

Ogni parola del documento è fatta passare nella funzione di hash H e si ottiene:

```
1 H(Kasim)  -->   00110101
2 H(Selcuk) -->   10010110
```

Listing 5.3: Esempio di documento

Facendo l'OR bit a bit, si ottiene:

```
1   00110101
2   10010110
3   --------
4   10110111
```

Listing 5.4: OR bit a bit per ottenere la signature.

Il risultato così ottenuto è detto *signature del file*. Ogni file ha una signature propria (ovviamente dipendente dai termini che contiene) che viene interrogata nel momento in cui si esegue una query.

Eseguire le query

Nel momento in cui si esegue una query (diciamo per semplicità che contenga un termine solo), anche quel termine viene fatto passare dentro H, ottenendo così una sequenza di bit che possiamo sfruttare insieme ad una signature di un file per capire se quel termine è contenuto o meno nel file cui la signature appartiene.
Il confronto si fa tramite AND della signature e del risultato della funzione di hash sul termine della query: se l'operazione di AND non fa variare la sequenza associata alla keyword della query, allora il termine *può* essere nel documento. Ma quello che è interessante è che se l'AND invece modifica la sequenza associata alla keyword della query, certamente quella keyword non è nel documento.
Vediamo sull'esempio di prima, proviamo a mettere in AND l'hash di Kasim e la signature del file e otteniamo:

```
1  00110101 AND
2  10110111
3  ————————
4  00110101
```

Listing 5.5: AND per verificare la presenza della keyword.

La sequenza è ancora la `H(Kasim)`: allora Kasim *può* far parte del documento (e noi sappiamo che ne fa parte).

Regolare dimensione e ratio

È evidente che all'aumentare del numero di bit a 1 nella signature, accetteremo più termini (fino al caso degenere in cui accettiamo tutti se ci sono tutti 1). Dobbiamo quindi dimensionare correttamente la signature e ratio di bit a 1 per evitare di avere moltissimi false hit. Andiamo a ragionare con qualche formula. Iniziamo col definire:

- ℓ il numero di bit a 1.
- B la dimensione della signature.

Possiamo quindi dire che se ci sono b keywords nel documento, la probabilità che un bit della signature sia settato a 1 è pari a:

$$1 - \left(1 - \frac{1}{B}\right)^{b\ell}$$

La formula è stata ottenuta seguendo questo ragionamento:

- Se abbiamo B bit e ne vogliamo settare uno ad 1, allora la probabilità di settare un determinato bit (preciso) è $\frac{1}{B}$.
- Quindi la probabilità che un determinato bit *non* venga settato ad uno è $1 - \frac{1}{B}$ (come si vede stiamo accettando le riestrazioni, ergo potremmo avere meno di ℓ bit settati a 1, ma non è un problema).
- Ma giacché ne dobbiamo settare ℓ, vi sono ℓ estrazioni e quindi la probabilità che un bit rimanga a 0 dopo tutte le estrazioni è $\left(1 - \frac{1}{B}\right)^{\ell}$.
- Poiché vi sono b keywords nel documento e la signature ha un bit a 0 solo se *nessuna* keyword ha il bit in quella posizione ad 1, allora l'evento di prima deve succedere b volte, quindi abbiamo $\left(1 - \frac{1}{B}\right)^{b\ell}$. Abbiamo appena descritto la probabilità che, fissata una posizione, nessun bit in nessuna keyword sia ad 1 in quella posizione e quindi sia a 0 nella signature
- Se quindi facciamo $1 - \left(1 - \frac{1}{B}\right)^{b\ell}$ abbiamo la probabilità che fissata una posizione almeno un bit in una keyword sia a 1 in quella posizione e quindi risulti 1 anche nella signature.

Tramite alcuni passaggi analitici, è possibile trasformare la formula di prima in una versione approssimata più trattabile:

$$1 - \left(1 - \frac{1}{B}\right)^{b\ell} \approx 1 - e^{-b\alpha}$$

con:

$$\alpha = \frac{\ell}{B}$$

La definizione di α è interpretabile come la percentuale di bit a 1 della signature(numero di bit a 1 sul totale di bit). Ora, avevamo detto di avere ℓ bit a 1 e la formula che abbiamo descrive la probabilità che 1 bit della signature sia a 1. Quindi tale probabilità va presa ancora ℓ volte:

$$(1 - e^{-b\alpha})^{\ell}$$

Possiamo sostituire usando la definizione di α e otteniamo:

$$(1 - e^{-b\alpha})^{\alpha B}$$

Vogliamo chiaramente *minimizzare* questo valore (che è il numero di 1), quindi essendo un esponenziale minimizziamo la sua base (ne faccio la derivata zero, poi studio i massimi ed i minimi) ed ottengo che tale funzione è minimizzata per:

$$\alpha = \frac{\ln(2)}{B}$$

Quindi, sfruttando ancora una volta la definizione di α otteniamo finalmente il valore ottimale di ℓ:

$$\ell = B\frac{\ln(2)}{b}$$

Quando questo vincolo è rispettato, allora abbiamo che la probabilità di false hit è $\frac{1}{2^\ell}$. Come vediamo la formula evidenza un tradeoff tra la lunghezza della signature e il numero di false hit.
Sperimentalmente si è notato che un buon α è 0.5.

5.4 Strutture per multimedia: una sola dimensione?

Prima di introdurre delle strutture che possano memorizzare surrogati (cioè strutture multidimensione), proviamo a vedere se esiste un modo di sfruttare le strutture dati che abbiamo studiato fin ora per rappresentare qualcosa di multidimensionale.
È evidente che questo sia impossibile (abbiamo studiato tutte strutture monodimensionali) ma ciò che si può fare è ridurre le n dimensioni dei surrogati in una dimensione soltanto, così che siano poi indicizzabili dalle strutture monodimensionali che conosciamo.

Ancora una volta, ricordiamo che scendere di dimensione implica perdere informazioni (amenoché una delle dimensioni non fosse ridondante); ma è possibile perdere meno informazioni possibili se la riduzione ad uno spazio solo viene fatta in maniera intelligente.
L'intento è quindi quello di matenere gli oggetti che erano vicini nello spazio k-dimensionale ancora vicini nello spazio monodimensionale. Andiamo ora a elencare una serie di idee possibili (discutendole una per una). Per motivi grafici partiremo sempre da 2 dimensioni ma chiaramente è possibile comprimere qualsiasi spazio k-dimensionale seguendo queste metodologie.

Tutte le tecniche si fondano comunque sul principio di *space filling curve*, cioè di attraversare lo spazio k-dimensionale di oggetto in oggetto e di fornire un ordinamento fra questi (un ordinamento è ovviamente monodimensionale).

Row order/column order

La prima idea, la più banale: sfruttiamo l'ordinamento riga/colonna.

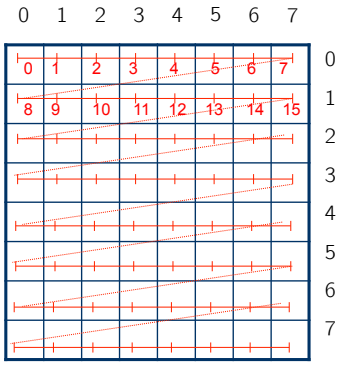


Figura 5.20: Spaziamo lo spazio seguendo l'ordinamento riga/colonna.

Questa soluzione è estremamente sbilanciata: i punti (0,1) e (0,2) che erano a distanza 1 nelle 2D, sono ancora a distanza 1 nella monodimensione, ma (0,0) e (1,0) hanno una distanza di 8 mentre prima erano distanti 1! In sostanza abbiamo preservato in maniera perfetta una dimensione mentre abbiamo storpiato enormemente l'altra. Lo stesso errore lo ritroviamo al ''cambio di riga'', fra i punti (0,7) e (1,0) (che prima distavano $\sqrt{15}$ e ora distano 1).
Cerchiamo qualcosa di più equilibrato.

Row prime order/column prime order

Muoviamoci ora senza effettuare il salto in diagonale, ma zigzagando per la zona.

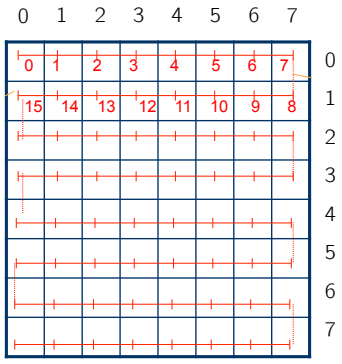


Figura 5.21: Abbiamo risolto alcuni problemi, ma non tutti.

Abbiamo risolto qualcosa: ora un ''cambio di riga'' su due non è problematico, cioè, non c'è più problema a 7-8 ma lo ritroviamo a 0-15 dove non c'è la congiunzione dello zigzagare.

Cantor diagonal order

Proviamo allora a seguire un andamento diagonale su tutto lo spazio. Ora

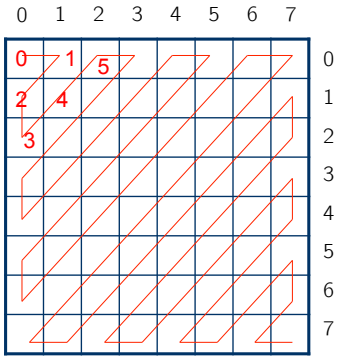


Figura 5.22: Problemi fra le diagonali.

il problema delle distanze si presenta lungo le diagonali (fra una diagonale e l'altra, precisamente) e non più sul cambio di riga. Non va ancora bene!

Z-order curve (Hilbert curve)

Usiamo una curva di Hilbert, che riproduce un pattern di zeta (una sorta di frattale).

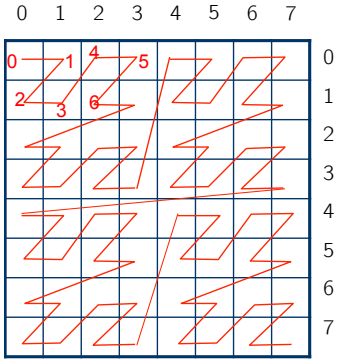


Figura 5.23: Ottima soluzione, facile computazione.

Questa soluzione è decisamente interessante anche se presenta ancora qualche difetto: la soluzione migliore la vedremo nel prossimo paragrafo (la curva di Peano-Hilbert). Nella pratica si adotta però la Z-order curve per motivi di implementazione. Il calcolo dello spazio è infatti estremamente semplice. Riguardando l'immagine sopra, ad esempio, se vogliamo tradurre il punto (1, 2) non dovremo far altro che tradurli entrambi in binario (rispettivamente 001 e 010) e poi effettuare bit shuffling (prendiamo un bit da una coordinata e un

bit dall'altro finché non li abbiamo presi tutti), ottenenendo così 000110 che è proprio la codifica di 6, cioè il punto corrispondente nella monodimensione.

Altro elemento fondamentale è il fatto che le ricerche per range sono facilitate dalla struttura; potremmo usare un trie:

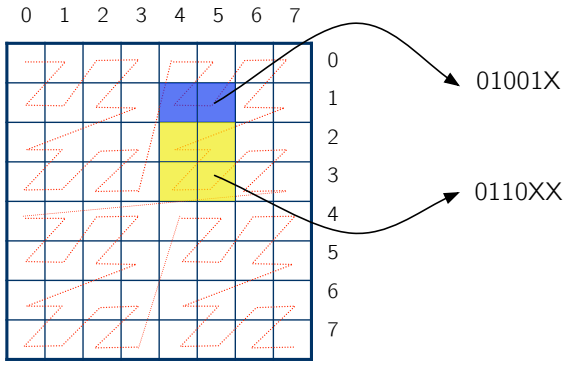


Figura 5.24: Le query per range sono semplici da eseguire.

Peano-Hilbert curve

Per completezza riportiamo anche la curva che genera la minor quantità di errore, la Peano-Hilbert:

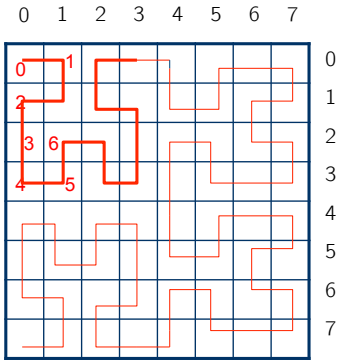


Figura 5.25: Soluzione perfetta ma di difficile computazione.

La traduzione dalla k-dimensione alla 1D è però molto più complessa, quindi si preferisce la Z-order curve.

5.5 Strutture multidimensionali per multimedia

Vediamo ora delle strutture ''proprie'' per rappresentare i multimedia in tutte le loro dimensioni senza doverle comprimere in una soltanto. Abbiamo quindi

uno spaizio N-dimensionale con M punti (surrogati) al suo interno. Diponiamo poi una funzione di distanza che ci dice quanto due punti sono simili fra loro. Le strutture che andremo a studiare sono:

1. Point quadtrees
2. MX quadtrees
3. kD trees
4. R-trees
5. TV-trees (Telescopic Vector)
6. X-trees

Le prime tre strutture sono basate su punti per definire regioni, la quarta direttamente sulle regioni, la quinta simula l'effetto di un telescopio e l'ultima estende i TV-trees risolvendone alcune problematiche.
Studieremo poi indici per indicizzare intervalli (perfetti quindi per il tempo).

Ricordiamo che le query che siamo intenzionati a risolvere sono query esatte, top-K oppure di range.

5.5.1 Approccio generale

L'approccio generale che useremo è sempre lo stesso e sfrutta la metafora che già abbiamo imparato con gli alberi (quelli che prima erano valori ora sono regioni):

- La struttura è gerarchica, ogni nodo rappresenta una regione.
- Scendere di livello significa muoversi in una sotto-regione o in un altra, quindi ogni nodo definisce anche delle sotto-regioni.
- È possibile aggiungere oggetti (punti, surrogati) alle regioni.
- È possibile splittare regioni piene per fare spazio ad altri oggetti.
- È possibile percorrere la struttura in fase di retrieval risparmiando così una lettura sequenziale: la struttura deve essere corretta nell'escludere certe strade (garanzia che non ci siano risultati interessanti).

5.5.2 Grid file: una base da cui partire

Prima di parlare delle strutture dati che abbiamo elencato prima, introduciamo una forma di indicizzazione davvero banale sulla quale però si fondano i concetti di molte delle tecniche che vedremo.

Il grid file ha una funzione simile all'indice di un libro suddiviso in capitoli: lo spazio viene scisso in regioni (eventualmente non regolari) ciascuna delle quali mantiene un certo numero di surrogati. Le regioni sono suddivise secondo la distribuzione dei dati; ad ogni cella corrisponde una pagina in memoria. Ci sarà dunque una struttura che mappa ogni cella su un certo indirizzo di memoria.

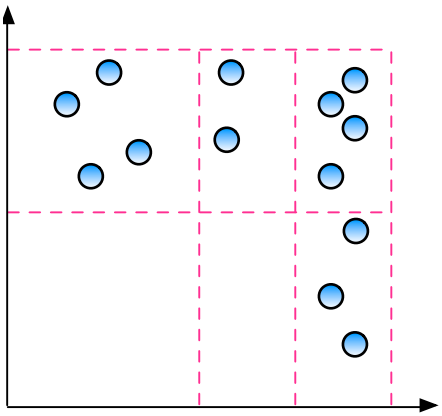


Figura 5.26: Esempio di grid file (dimensione pagina: 4 elementi)

Questo tipo di suddivisione (molto semplice) permette una facile risoluzione delle query ma soffre di alcune mancanze: le pagine in memoria hanno un numero limitato di elementi e dunque sarà possibile memorizzare un numero limitato di surrogati in ogni cella (ma il numero di surrogati in ogni cella è guidato dalla similarità e dunque non controllabile); inoltre nel caso in cui i dati siano non uniformemente distribuiti avremo un grosso spreco di memoria (le celle associate a regioni vuote sono sprecate).

5.5.3 Suddivisioni arbitrarie?

Assumendo di avere uno spazio bidimensionale possiamo trovare molti modi di suddividerlo: quadrati regolari, triangoli, rettangoli, ecc. Ad esempio se usassimo delle rette (ogni retta divide lo spazio in due parti) per rappresentare lo spazio potremmo ottenere qualcosa del genere:

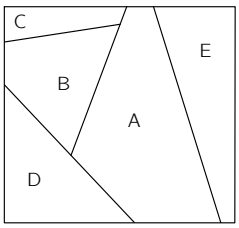


Figura 5.27: Esempio di suddivisione arbitraria.

La computazione risulterebbe però estremamente complessa, così come la manutenzione della struttura. Si preferiscono per questi motivi approcci che suddividano lo spazio nel modo *più regolare possibile*.

5.5.4 Point quadtrees

La prima struttura multi dimensionale che andiamo a studiare sono i point quadrees. Trattasi di una struttura ad albero caratterizzata da alcune peculiarità:

- Ogni nodo rappresenta *implicitamente* una regione.
- Ogni nodo cointene *esplicitamente* un punto (surrogato) che funge da nome.
- La root rappresenta l'intero spazio.
- Le sotto-regioni di un nodo sono ottenute dividendo la regione che esso rappresenta in quattro porzioni (''quadrants''), che vengono definite tracciando una linea orizzontale e una verticale con intersezione nel punto che identifica il nodo.
- Ogni nodo ha quindi quattro figli che puntano ai quattro ''quadrants''.

Andiamo ora a vedere come la struttura viene costruita, come è possibile modificarla e come è impiegabile per risolvere query esatte, top-K e per range.

Costruzione di un point quadtree

Iniziamo introducendo un nodo, la root, che rappresenta tutta la regione:

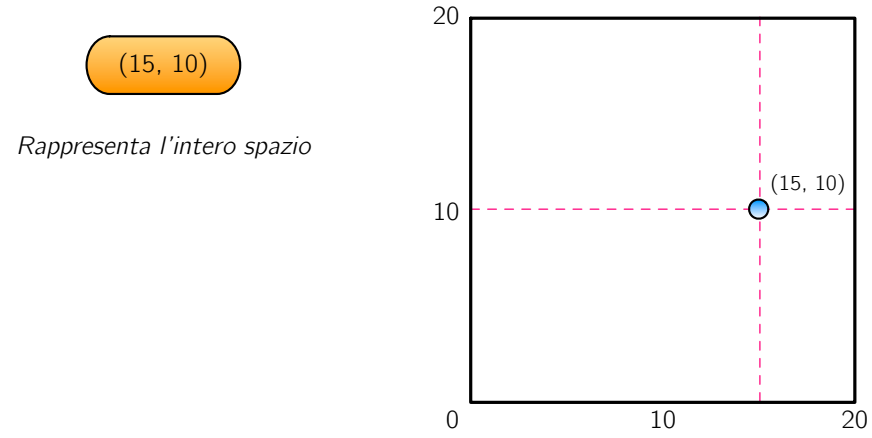


Figura 5.28: Il primo nodo diventa root.

Come vediamo l'introduzione del nodo ha portato alla suddivisione dell'intero

spazio (che esso rappresenta) in quattro parti, al momento vuote. Procediamo con l'aggiunta di un ulteriore nodo:

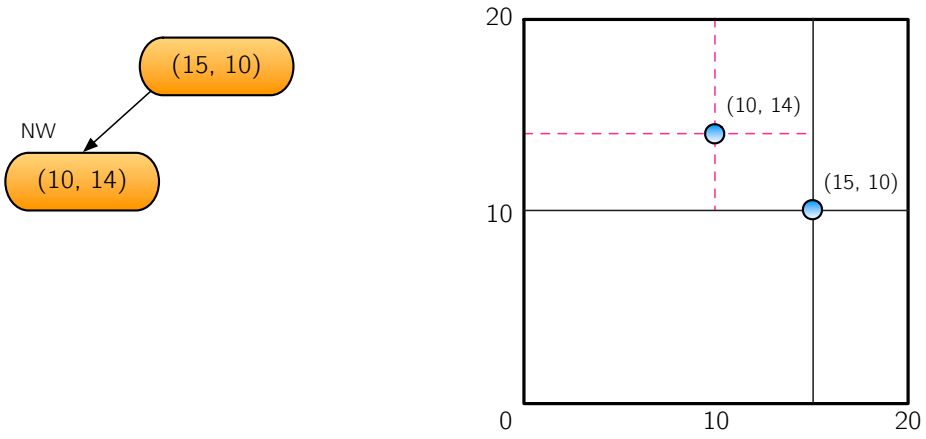


Figura 5.29: Un nodo è aggiunto a nord-ovest della radice.

Giacché il punto `(10, 14)` si trova a nord-ovest rispetto alla radice dell'albero, è stato creato un nodo etichettato come `NW` al quale è stato appeso il nuovo punto (che a sua volta suddivide la regione in quattro parti e la rappresenta per intero). Aggiungiamo ancora un nodo:

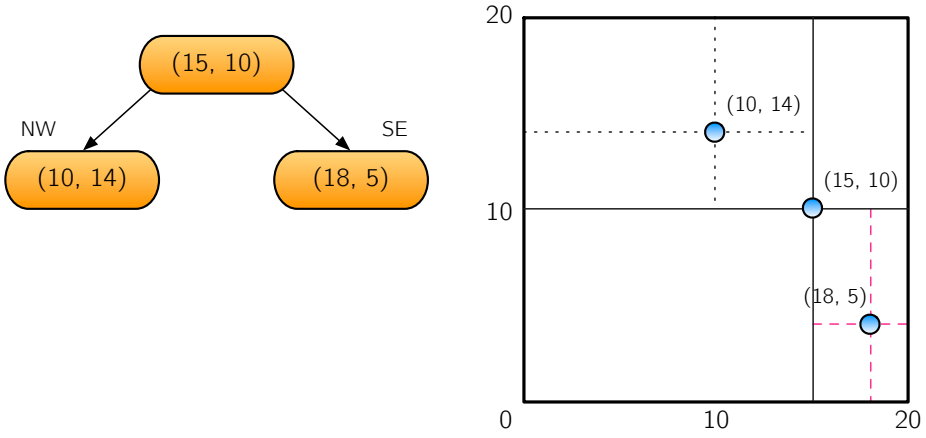


Figura 5.30: Un nodo è aggiunto a sud-est della radice.

Questa volta il punto si trova a sud-est rispetto alla radice, dunque il nodo è stato aggiunto con il label `SE` ed ha suddiviso quella regione in altre quattro, rappresentandola però per intero. Aggiungiamo un ulteriore nodo (Figura 5.31); il nodo `(2, 12)` si troverebbe a `NW` rispetto alla radice, ma giacché quella posizione è già occupata da `(10, 14)` si scende nelle regioni che quel

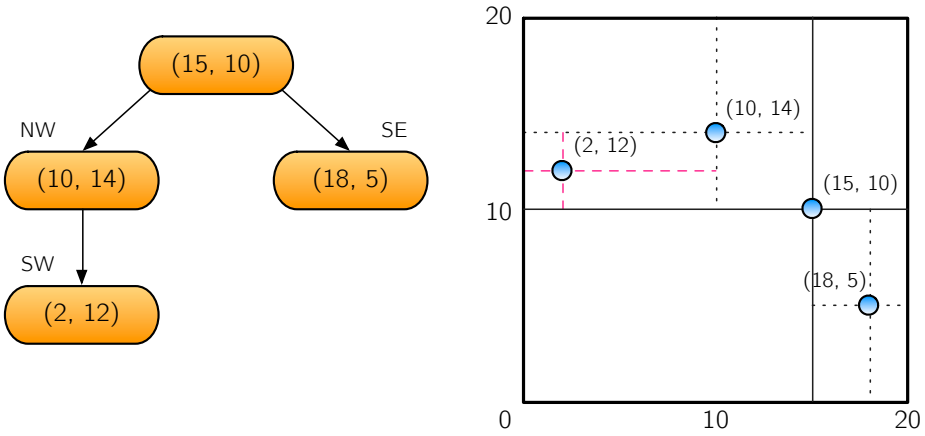


Figura 5.31: Un ulteriore nodo è aggiunto.

nodo definisce e quindi si piazza il nuovo nodo *relativamente* al nodo a `NW` della root. Perciò abbiamo un nodo a `SW` rispetto a `(10, 14)`.

Considerazioni in merito alla struttura. È evidente che la forma finale della struttura sia fortemente legata all'ordine con il quale i nodi vengono inseriti nell'albero. Un ulteriore fatto che si può notare è il grande spreco di puntatori: già in questo piccolo esempio ogni nodo ne ha quattro (16 totali) ma di questi ne usiamo soltanto tre. Questo fattore aumenta con l'aumentare delle dimensioni.

Analisi dei costi computazionali. Supponendo di avere un point quadtree con N nodi al suo interno, l'altezza massima sarà N e il caso peggiore di inserimento/ricerca sarà quindi N.

Cancellazione di un nodo

La cancellazione è un'operazione molto critica nei point quadtrees, infatti rimuovere un nodo significa de-contestualizzare tutto il sotto albero da esso definito. È quindi necessario scegliere (dal sottoalbero) un nodo che sostituisca quello appena eliminato ed il criterio di scelta sarà ''il nodo meno peggio'' in termini di modifica dell'albero. Chiaramente più la cancellazione avviene in alto nell'albero e più sarà critica la ricostruzione della struttura.

I costi di cancellazione sono così critici che talvolta si preferisce segnare semplicemente i nodi come *morti* mantendoli però all'interno della struttura per permettere comunque il raggiungimento dei sottonodi. Quando il numero di nodi morti supera una certa soglia, si potrà ricostruire l'intero albero da capo.

Vediamo un esempio di cancellazione sulla struttura che abbiamo costruito poc'anzi (abbiamo eliminato la root):

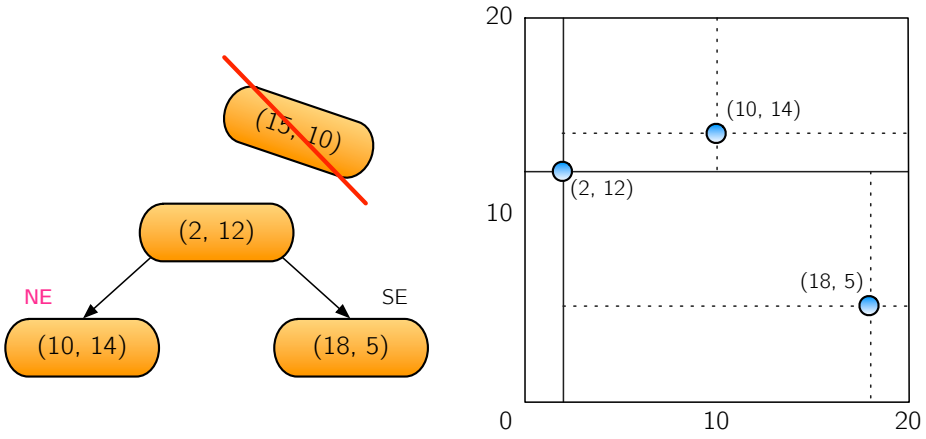


Figura 5.32: Esempio di cancellazione su point quadtrees.

Come è indicato in figura, il sostituito scelto è stato `(2, 12)` e per questo il nodo `(10, 14)` ha cambiato il suo label.

Esecuzione di query top-K (con $K = 1$)

Dato una certa query Q (che definirà ovviamente un punto nello spazio) si vogliono trovare i k nodi più vicini. Per semplicità concentriamoci su un esempio $K = 1$.

Un nuovo assioma. Ricordiamo gli assomi per le metriche in spazi di una dimensione (Sezione 2.1.3) e li estendiamo con un nuovo assioma per il nostro spazio bidimensionale:

$$d(x, R) = min\{d(x, y) | y \text{ si trova in } R\}$$

Stiamo quindi dicendo la la distanza fra un certo punto x e una regione R è calcolabile come la minima distanza fra x ed un y che si trovi nella R, cioè, è la distanza fra il punto in R più vicino ad x ed x.
Grazie a questo nuovo assioma siamo in grado di eseguire l'algoritmo per risolvere query top-K.

Sketch dell'algoritmo. Durante l'esecuzione dell'algoritmo vengono mantenute due variabili:

- `bestdist` che mantiene la distanza minima trovata fin'ora (inizializzata a ∞).

- `bestSOL` che mantiene il nodo che si trova a `bestdist`, cioè la migliore soluzione fin'ora (inizializzata a NIL).

Appurato che ogni nodo N definisce una regione $N.reg$, l'algoritmo compie questi passi:

1. L'algoritmo parte visitando la radice.
2. Ogni volta che viene visitato un nodo N, viene esaminato il punto che è associato al nodo. Se $d(\texttt{Q}, \texttt{N.point}) < \texttt{bestdist}$ (se la distanza fra la query e il punto mantenuto nel nodo è minore di `bestdist`) allora `bestdist` e `bestSOL` vengono aggiornate. Altrimenti, continua l'esplorazione.

Durante l'esecuzione possiamo però *potare* alcuni rami dell'albero: questo è possibile se e solo se $d(\texttt{Q}, \texttt{N.reg}) > \texttt{bestdist}$. La cosa è abbastanza naturale: se il punto più vicino alla query della regione è già più distante di quanto lo sia la nostra `bestdist`, allora la regione (quindi il sottoalbero associato al nodo che la implicitamente la rappresenta) non potrà contenere nulla di interessante.

Si tenga presente della grande differenza che c'è fra il confrontare la distanza fra la query e *la regione* e confrontare la distanza fra la query ed il *punto* di un nodo! Possiamo potare solamente se la distanza fra la query e l'intera regione è maggiore di quella che abbiamo già, ma non possiamo potare basandoci solo sul punto: di fatto all'interno della regione rappresentata dal nodo *potrebbe* esserci una distanza migliore.

Esempio d'esecuzione. Riprendiamo l'albero che abbiamo costruito prima e proviamo ad eseguire la query `Q = (9, 14)` (usiamo una coda a priorità per mantenere i nodi ancora da esplorare, ordinati per distanza crescente).

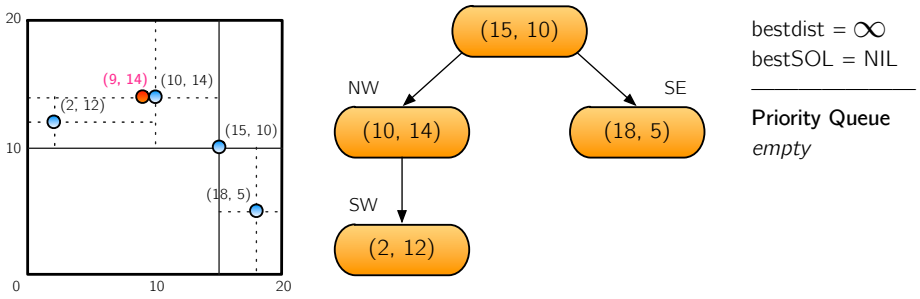


Figura 5.33: Point quadtree su cui stiamo per eseguire una query top-K.

Iniziamo dunque chiedendoci se la distanza fra la regione rappresentata dalla root e la query è minore di infinito (al fine di capire se la regione è da esplorare o meno):

$$d(\texttt{Q}, \texttt{root.reg}) = 0 < \infty = \texttt{bestdist}$$

chiaramente lo è, poiché la root rappresenta tutto lo spazio e la query vi è inclusa (infatti la distanza è 0). Giacché la root è da esplorare, viene inserita nella coda.

Preleviamo un nodo dalla coda (l'unico che c'è, root) e per prima cosa verifichiamo se il punto che quel nodo rappresenta migliora la nostra soluzione:

$$d(\texttt{Q}, \texttt{root.point}) = \sqrt{(9-15)^2 + (14-10)^2} = \sqrt{52}$$

Giacché $\sqrt{52} < \infty$ (l'attuale valore di `bestdist`), aggiorniamo sia `bestdist` che `bestSOL`.

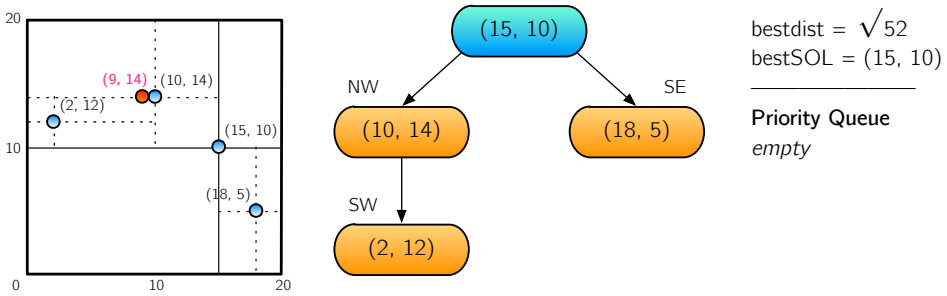


Figura 5.34: Visitiamo il nodo `root` (in blu).

Fermiamoci un attimo per capire quanto appena accaduto: si sono succeduti due passi profondamente diversi sebbene simili: prima abbiamo considerato la *regione* della root e abbiamo cercato di capire se tale regione fosse interessante o meno (confrontandola con `bestdist` ma *senza modificarne il valore*). Il nodo è stato così messo in priority queue.
A quel punto abbiamo estratto dalla priority queue il nodo e abbiamo lavorato sul punto che rappresenta, verificando se si trattasse o meno di una soluzione migliore di quella che avevamo. Essendo `bestdist` a infinito, la distanza fra `Q` e `root` è risultata migliore e quindi abbiamo aggiornato le due variabili `bestdist` e `bestSOL`.

Adesso ripetiamo questi due passi partendo dalla prima operazione, cioè, capiamo quale nodo deve essere esplorato.
Per fare questo esaminiamo tutti i figli del nodo sul quale ci troviamo e se promettenti (nel senso di distanza fra regione e query minore della `bestdist` attuale) le inseriamo nella priority queue (nel caso di prima ci trovavamo al passo iniziale, quindi c'era solo root e l'uso della coda di priorità è stato inutile).

Calcoliamo quindi le due distanze:

- Per `NW`: $d(\texttt{Q}, \texttt{root.NW.reg}) = 0$ (poiché `Q` si trova nella regione). $0 <$ `bestdist`, quindi la regione merita d'essere esaminata (è possibile che

contega nodi in grado di migliorare la nostra soluzione attuale).
Viene inserita in priority queue con priorità 0 (la distanza appena calcolata).

- Per `SE`: $d(\texttt{Q}, \texttt{root.SE.reg}) = \sqrt{52}$. Tale valore è stato calcolato come distanza fra `Q` ed un certo punto (α, β) che è il punto appartenente alla regione `SW` più vicino possibile alla query. Chiaramente questo calcolo dipende dalla posizione reciproca della query e della regione presa in esame.
  Ad esempio in questo caso (si guardi la figura), (α, β) è il punto in alto a sinistra della regione, cioè proprio `(15, 10)`. Ma la distanza fra `Q` e quel punto l'abbiamo già calcolata al passo prima, è proprio $\sqrt{52}$. Si ponga attenzione al fatto che *il calcolo non è sempre così semplice, dipende dalla posizione reciproca fra la query e la regione*.
  Grazie all'assioma che abbiamo definito prima, sappiamo che se la distanza appena calcolata non è minore di quella che avevamo in `bestdist` allora la regione (quindi il sottoalbero) è potabile.
  In questo caso, $\sqrt{52}$ non è maggiore di `bestdist` (sono uguali), quindi la regione non merita d'essere esaminata (viene potata) e perciò non viene inserita nella priority queue.

La situazione attuale è quindi:

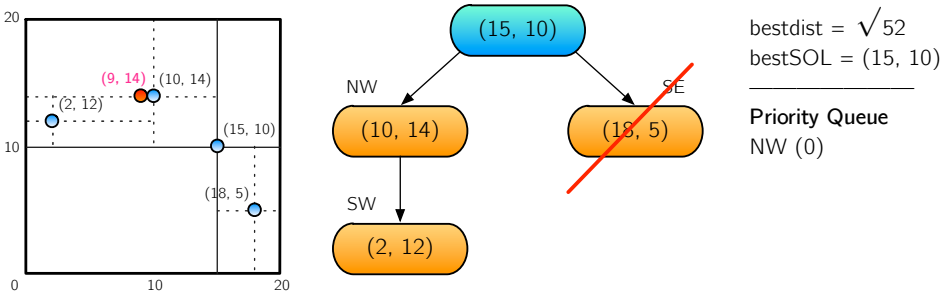


Figura 5.35: Potiamo il sottoalbero con radice `root.SW`.

Ora, peschiamo un nuovo nodo dalla priority queue.

Consideriamo il punto ad esso associato `(10, 14)` e vediamo se è in grado di migliorare la nostra soluzione. Calcoliamo la sua distanza dalla query:

$$d(\texttt{Q}, \texttt{root.NW.point}) = \sqrt{(9-10)^2 + (14-14)^2} = 1$$

Giacché $1 < \sqrt{52} = 7.2$ (l'attuale valore di `bestdist`), aggiorniamo le nostre variabili `bestdist` e `bestSOL`:

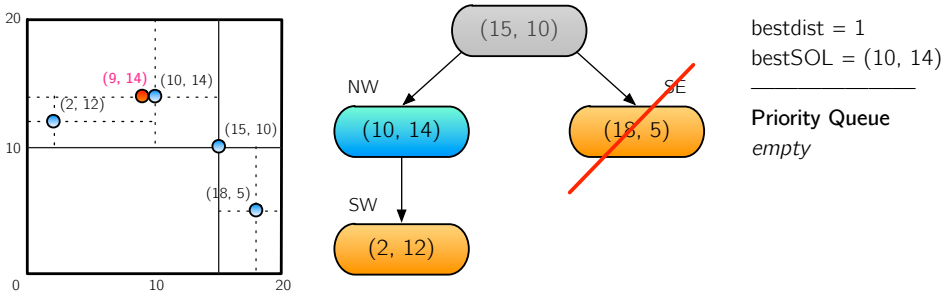


Figura 5.36: Visitiamo il nodo `root.NW` (in blu).

Ora che il nodo è stato esaminato, dobbiamo considerare i suoi figli alla ricerca di altre regioni interessanti:

- Figlio `NW`: $d(\texttt{Q}, \texttt{root.NW.NW.reg}) = 0$ (poiché `Q` si trova nella regione). $0 <$ `bestdist`, quindi la regione merita d'essere esaminata (è possibile che contega nodi in grado di migliorare la nostra soluzione attuale). Viene inserita in priority queue con priorità 0 (la distanza appena calcolata).

Viene a questo punto pescato l'unico nodo nella priority queue e viene esaminato punto ad esso associato.

$$d(\texttt{Q}, \texttt{root.NW.NW.point}) = \sqrt{(9-2)^2 + (14-16)^2} = \sqrt{53}$$

Giacché $7.3 = \sqrt{53} > 1$ (l'attuale valore di `bestdist`), non dobbiamo aggiornare le variabili `bestdist` e `bestSOL`.

L'algoritmo termina qui poiché *l'albero è finito*, con il risultato che il nodo più vicino alla query è `(10,14)` (risultato che è facile verificare esser vero grazie alla figura).

Esecuzione di query top-K generiche

Abbiamo visto l'esempio dell'esecuzione di una query top-K con $K = 1$. Se vogliamo generalizzare l'algoritmo, non dobbiamo far altro che apportare alcuni lievi modifiche alla procedura appena studiata:

- `bestdist` e `bestSOL` diventano due array, ordinando gli elementi dal meno distante al più distante.
- I confronti su `bestdist` e `bestSOL` vengono effettuati sull'elemento più distante (a fondo vettore quindi).
- Quando un elemento viene inserito in `bestdist` e `bestSOL` viene inserito in ordine.

Esecuzione di query per range

Le query per range usano un sistema molto simile a quello appena visto, ma invece di usare `bestSOL` e `bestdist` si mantiene una lista delle soluzioni `SOL` e si sfrutta il raggio fornito nella query per individuare i punti accettati.
Più precisamente:

- Data una query `Q` ed un raggio `R` si richiede di trovare tutti i punti dell'albero che sono a distanza al massimo `R` da `Q`.
- La query definisce quindi una zona circolare di raggio `R` centrata in `Q`, cioè $C(\texttt{Q}, \texttt{R})$.
- Il nodo N viene potato se la sua regione $N.reg$ non ha intersezioni con $C(\texttt{Q}, \texttt{R})$ (eventuamente si può usare un bouding box intorno al cerchio per semplicità).

Nell'esempio di prima, avendo un raggio $R = 3$ con la solita query $Q = (9, 14)$, avremmo eseguito:

1. La regione della `root` (tutto lo spazio) ha intersezione con la query, quindi la prendiamo in considerazione.
2. Peschiamo la `root` dalla coda ed esaminiamo il nodo ad essa associato: $d(\texttt{Q}, \texttt{root.point}) > R = 3$, quindi il nodo `root` non è soluzione.
3. Esaminiamo i figli di `root` e vediamo quali regioni intersecano il cerchio definito dalla query. Abbiamo:
   - Per `NW`: $d(\texttt{Q}, \texttt{root.NW.reg}) = 0$ (il nodo è incluso in quella regione), e giacché $0 < R = 3$ la regione è presa in considerazione (messa in coda con priorità 0).
   - Per `SE`: $d(\texttt{Q}, \texttt{root.SE.reg}) = \sqrt{52}$ (calcolato sempre con distanza fra il punto più vicino alla query della regione e la query stessa), e giacché $\sqrt{52} > R = 3$ la regione non è presa in considerazione (il ramo è potato).
4. Viene pescato un nodo dalla priority queue (l'unico presente è `root.NW`), si calcola la distanza fra quel nodo e la query: $d(\texttt{Q}, \texttt{root.NW.point}) = 1$. Giacché $1 < R = 3$ il nodo `root.NW` è aggiunto all'insieme delle soluzioni, quindi `SOL = {(10, 14)}`.
5. Vengono presi in considerazione i figli del nodo dove ci troviamo, l'unico che c'è è `root.NW.NW`. Vediamo se la regione che rappresenta è interessante: $d(\texttt{Q}, \texttt{root.NW.NW.reg}) = 0$ (il punto si trova in quella regione). La distanza appena trovata è minore di $R = 3$ e quindi `root.NW.NW` è incuso nella priority list.

6. Viene pescato il nodo `root.NW.NW` dalla priority list e si calcola la distanza fra il punto che rappresenta e la query: $d(\texttt{Q}, \texttt{root.NW.NW.point}) = \sqrt{53}$ e giacché $\sqrt{53} > R = 3$ il punto non viene incluso nella soluzione.
7. L'albero è finito, la soluzione si trova nella lista `SOL = {(10, 14)}`.

Problemi dei point quadtree

I point quadtree soffrono di alcune problematiche:

- La cancellazione è problematica e lenta.
- L'albero può essere fortemente sbilanciato.
- Le dimensioni fra le regioni possono essere drammaticamente diverse.

L'insieme di questi fattori rende la previsione del tempo di esecuzione delle query di range e top-K impredicibile.

5.5.5 MX quadtrees

Cerchiamo di risolvere le problematiche dei point quadtrees introducendo gli *MX quadtrees*. MX sta per *matrix* e capiamo subito il perché di questo nome:

- L'intero spazio è diviso in una matrice $2^n \times 2^n$ (quindi le celle sono una quantità che è potenza di due).
- Ogni regione è ottenuta dividendo con una linea verticale ed una orizzontale il centro della regione.
- Solamente le foglie sono valorizzate (mantengono un punto).

Vediamo un esempio di spazio vuoto suddiviso in regioni per MX quadtree:

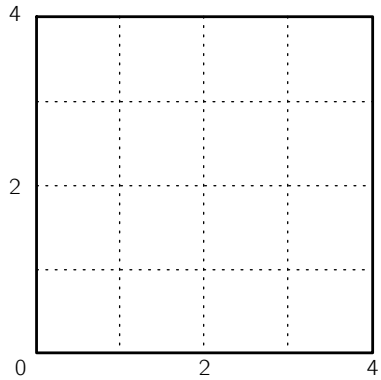


Figura 5.37: Esempio di spazio vuoto con $n = 2$.

Lo spazio generato è 4×4, poiché è un $2^n \times 2^n$ con $n = 2$.

Costruzione di un MX quadtree

Costruiamo ora un MX quadtree insieme. La tecnica è analoga a quella studiata per i point quadtrees, ma questa volta non possiamo inserire alcun punto nei nodi (le regioni sono prestabilite fin dall'inizio!). Iniziamo con l'inserire un punto `(1, 4)`:

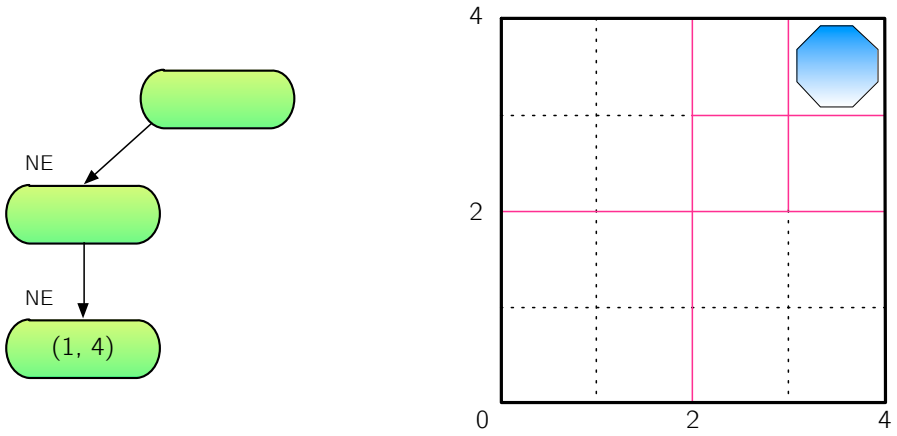


Figura 5.38: Insert del punto `(1,4)`.

Come vediamo, la struttura si è espansa fino a rappresentare la cella (1, 4) dove il punto si trova.

Inseriamo ora altri due punti:

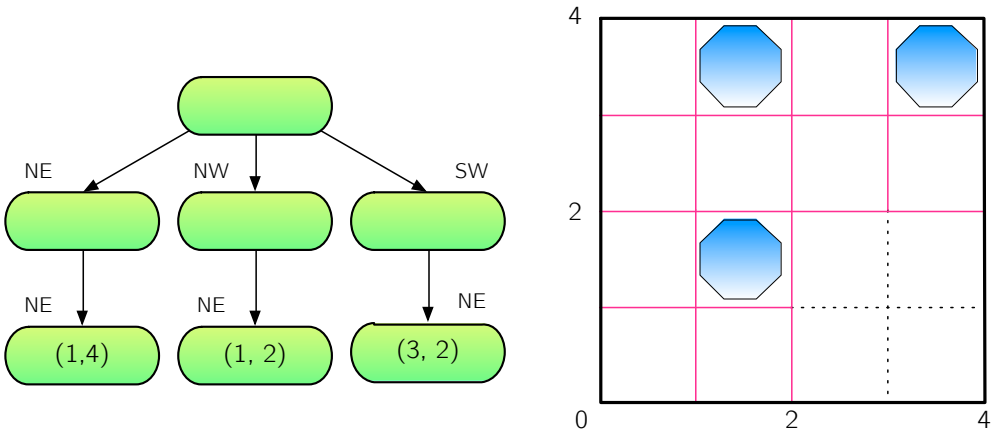


Figura 5.39: Inseriamo i punti `(1,2)` e `(3, 2)`.

Come vediamo la tecnica è analoga e leggermente più semplice rispetto a prima, poiché le regioni sono definite a monte.

Considerazioni in merito alla struttura. Possiamo quindi fare le seguenti considerazioni:

- Ogni nodo rappresenta una regione.
- Un nodo a livello j rappresenta una regione di dimensione $2^{n-j} \times 2^{n-j}$ (infatti il root a livello 0 rappresenta la regione totale $2^n \times 2^n$).

Analisi dei costi computazionali. L'inserzione e la ricerca hanno costo $O(n)$, cioè logaritmico nel numero di nodi e lineare nel numero di livelli (si noti che è n e non N!).

Cancellazione di un nodo

La cancellazione subisce un netto miglioramento rispetto a quanto abbiamo visto nei point quadrees. Ora non si deve far altro che raggiungere il nodo (che sarà senz'altro una foglia) ed eliminarlo (costo logaritmico). Nel caso in cui in seguito ad una cancellazione il nodo padre rimanga senza figli, si procede alla cancellazione anche di quest'ultimo. Questo processo è detto *collapsing*.

Query di top-K e di range su MX quadtree

Le query di top-K e di range si svolgono in modo del tutto analogo rispetto a quanto studiato per i point quadtrees.

5.5.6 PR quadtrees

Purtroppo gli MX quadtree soffrono di una grande problematica: i dati devono essere discretizzati a monte per poterli indicizzare dunque non c'è rappresentazione nel continuo dei vari punti. I *PR quadtrees* (di cui facciamo solamente un accenno) cercano un compromesso fra i point quadtrees e gli MX quadtrees: la struttura è quindi indipendente dall'ordine di inserzione dei nodi e la cancellazione è facile (come era negli MX quadtrees) e lo spazio considerato è continuo (come nei point quadtree). Ne vediamo un esempio in Figura 5.40.

5.5.7 KD trees

Tutti i quadtree visti fin ora soffrono di tre problematiche collegate fra loro; più precisamente data una certa dimensione k dello spazio si ha che:

- Ogni nodo richiede k confronti durante la ricerca per scegliere quale strada perseguire.
- Ogni foglia contiene k puntatori a null.
- I nodi diventano sempre più grossi all'aumentare di k.

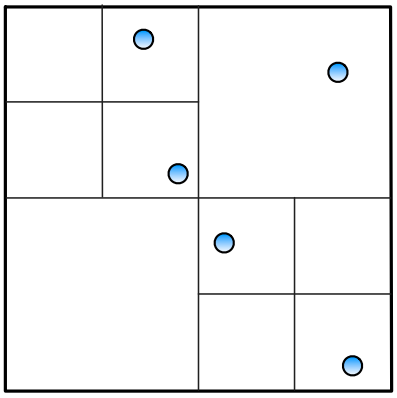


Figura 5.40: Esempio di PR quadtree.

Questo accade poiché ogni nodo ha in generale 2^k puntatori a figli con l'evidente rischio di avere molti pointer a null. È quindi evidente che all'aumentare delle dimensioni le problematiche aumentino con andamento *esponenziale*: questo fatto non ci piace per nulla.

I *KD trees* pongono rimedio a queste problematiche, definendo alberi i cui nodi hanno *sempre due figli* (si tratta quindi di alberi binari): in questo modo il numero di puntatori che ogni nodo possiede è sempre e solo due, quale che sia la dimensione dello spazio indicizzato dall'albero.

Definizione dell'albero

Un KD-tree è quindi usato per immagazzinare punti K-dimensionali (nella forma $x_0, ..., x_{K-1}$). L'albero si configura in questo modo:

1. Assumendo che la radice sia a livello 0, un nodo a livello i discrimina sulla dimensione $i \mod K$.

2. Ogni nodo spezza la regione ad esso associata in due parti, suddivise seguendo la dimensione associata al livello cui il nodo si trova.

Capiremo meglio cosa intendiamo in seguito.
Nota bene: negli esempi a seguire vedremo applicazioni con $K = 2$ (quindi parlemo di 2D trees) per motivi di semplicità e di rappresentazione grafica altrimenti incomprensibile.

Un 2D tree è un albero binario tale che:

- Se N è un nodo che si trova a livello *pari*, allora per ogni nodo M nel suo sottoalbero sinistro varrà la condizione :

$$M.xcoord < N.xcoord$$

e per ogni nodo P del suo sottoalbero destro varrà la condizione:

$$P.xcoord \geq P.xcoord$$

- Se N è un nodo che si trova a livello *dispari*, allora per ogni nodo M nel suo sottoalbero sinistro varrà la condizione :

$$M.ycoord < N.ycoord$$

e per ogni nodo P del suo sottoalbero destro varrà la condizione:

$$P.ycoord \geq P.ycoord$$

Quello che stiamo dicendo è che i nodi a livello pari discriminano sulle x (tracciano una linea implicita verticale) mentre i nodi a livello dispari discriminano sulle y (tracciano una linea implicita orizzontale). È quello che avevamo espresso nel primo dei due punti di inizio paragrafo: se infatti abbiamo in generale k dimensioni, al livello i discrimineremo sulla dimensione $i \mod k$.

Vediamo ora la costruzione di un 2D tree:

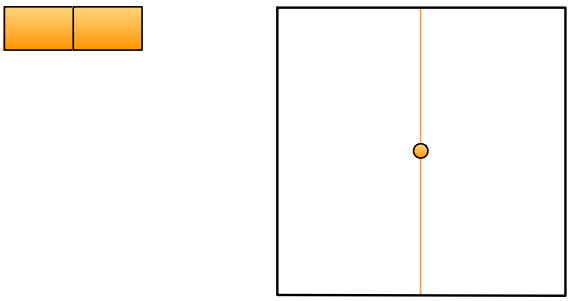


Figura 5.41: Un punto è inserito all'interno del 2D tree.

Essendo il punto inserito a livello di radice (cioè a livello 0), ed essendo il livello 0 un livello pari, il nodo discrimina sulle x: il nodo fraziona la regione che rappresenta (tutto lo spazio) in due parti, tramite un segmento verticale passante per il punto (non rappresentato numericamente all'interno dell'albero per semplicità).

Proseguiamo ora inserendo un ulteriore punto (Figura 5.42).
Giacché *il valore delle* x del nuovo punto è maggiore rispetto al valore delle x della radice, il nuovo punto viene istanziato nel sottoalbero *destro* della radice: non si considera quindi il valore delle y nel processo di selezione della strada da seguire. Siamo ora a livello 1, un livello dispari, dunque il nodo discrimina sulle y: la regione rappresentata dal nodo è frazionata in due parti tramite un segmento orizzontale passante per il punto.

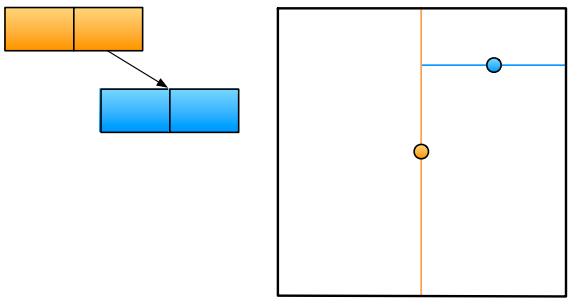


Figura 5.42: Un altro punto è inserito all'interno del 2D tree.

Si aggiunge un ulteriore punto (Figura 5.43): ripartiamo dalla radice e dobbiamo nuovamente effettuare un confronto sulle x ignorando le y: supponiamo che questa volta la x del nuovo punto sia minore rispetto alla x della radice: scendiamo a sinistra e piazziamo il nuovo nodo. Come prima, siamo a livello 1 e quindi discriminiamo sulle y.

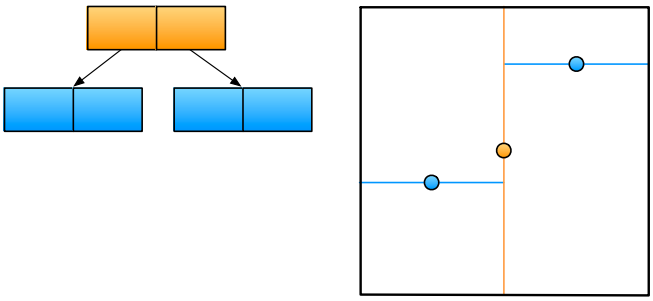


Figura 5.43: Un altro punto è inserito all'interno del 2D tree.

Aggiungiamo un punto ancora (Figura 5.44):

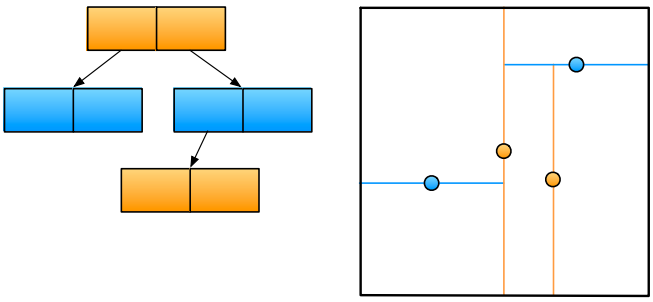


Figura 5.44: Un altro punto è inserito all'interno del 2D tree.

Ancora una volta siamo partiti dalla radice ed abbiamo considerato la sola x: la x del nostro punto è risultata maggiore dunque ci siamo spostati a destra. Poi, essendo a livello dispari, abbiamo considerato la y del nodo e l'abbiamo

confrontata con la y del nuovo punto, scoprendo che la prima è maggiore della seconda: il nodo è quindi stato collegato a sinistra del nodo. Siamo nuovamente ad un livello pari quindi il nuovo nodo discriminerà secondo le x (abbiamo tracciato la solita linea verticale passante per il punto).

Aggiungiamo un ulteriore punto (Figura 5.45):

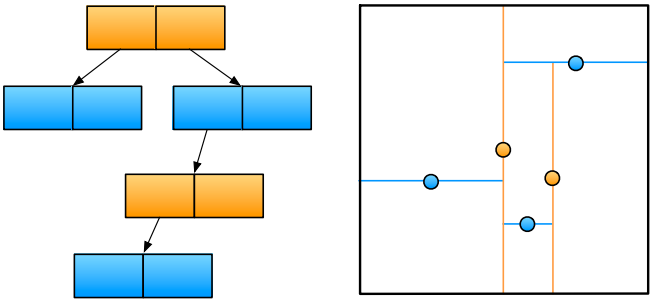


Figura 5.45: Un altro punto è inserito all'interno del 2D tree.

Abbiamo confrontato la x del nuovo punto con la x della radice, scoprendo che la prima è maggiore della seconda e per questo siamo scesi a destra. Poi, abbiamo confrontato la y del nuovo punto con quella del nodo e abbiamo visto che era minore e per questo siamo scesi a sinistra. Abbiamo poi considerato la x del nuovo punto e la x del nodo: abbiamo visto che la prima è minore della seconda e quindi siamo scesi a sinistra, dove il nuovo nodo è stato appeso. A questo livello (dispari) discriminiamo sulle y quindi è stata aggiunta una linea orizzontale passante per il punto.

Confronto con point quadtree

Osserviamo questa figura:

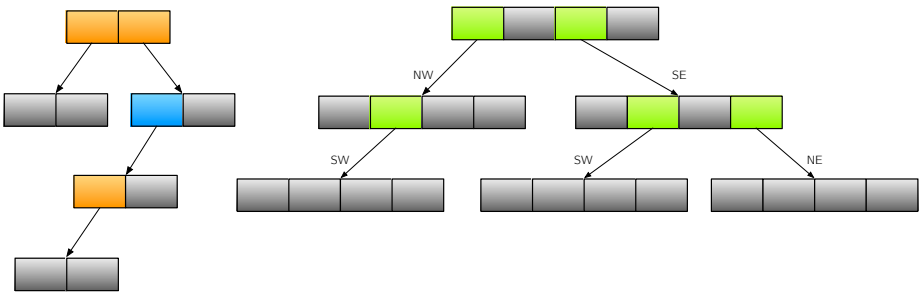


Figura 5.46: Confronto fra 2D tree e point quadtree.

Si tratta di due alberi che rappresentano gli stessi punti. A destra abbiamo un point quadtree mentre a sinistra abbiamo un 2D tree ed abbiamo indicato in grigio i puntatori a null. Come è evidente, la nuova struttura è molto più

conveniente dal punto di vista dello spazio anche se richiede un livello in più per rappresentare gli stessi punti (è quindi meno discriminante).

Un esempio concreto

Si riporta ora un esempio valorizzato di 2D tree:

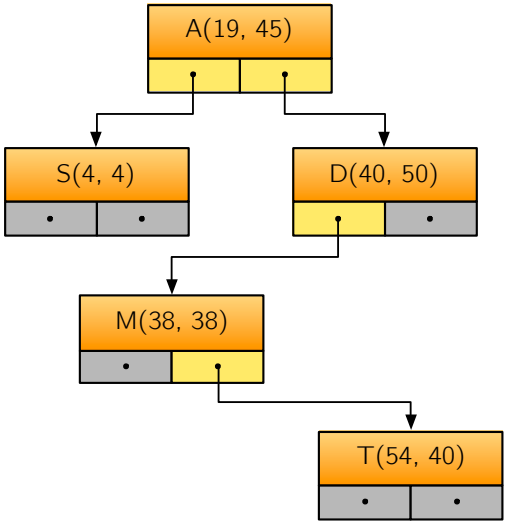


Figura 5.47: Un 2D tree.

Ricerca top-K e di range

Le ricerche top-K e di range funzionano esattamente come già studiato per i point quadtree.

Note finali

I problemi relativi alla cancellazione che avevamo visto con i point quadtree permangono, così come il rischio di albero degenere.
D'altro canto queste strutture sono molto semplici da implementare e (abbiamo visto) occupano uno spazio relativamente ridotto.

5.5.8 R-trees

Cambiamo ora approccio e poniamoci l'obiettivo di indicizzare *regioni*: fino ad ora abbiamo indicizzato punti a cui abbiamo associato regioni, mentre ora il nostro intento è la vera e propria memorizzazione di regioni. Si noti che quando diciamo ''memorizzare regioni'' intendiamo sia la memorizzazione di

regioni come *dati* (cioè vere e proprie superfici di cui vogliamo tenere traccia) ma possiamo anche considerare le regioni come spettro di variabilità di una certa dimensione (tale spettro definito da dati puntuali). Più precisamente, potremmo prendere due dimensioni (perso ed altezza delle persone, ad esempio) e rappresentarle su un grafico: se suddividiamo il grafico in diverse porzioni, allora avremmo delle regioni *semantiche*, ad esempio una regione dove vi sono i ragazzi alti 1,70 e che pesano 80 kg.
Nella nostra applicazione, comunque, penseremo a regioni vere e proprie.

Gli *R-trees* nascono inizialmente per rappresentare rettangoli bidimensionali, ma sono facilmente generalizzabili a spazi k-dimensionali (noi ci limiteremo allo studio della versione basilare).
Gli R-tree prendono molto dai fratelli B-tree, infatti, per ogni nodo è possibile memorizzare un minimo di $N/2$ figli ed un massimo di N figli. Nei casi reali, N sarà tale da riempire una pagina di memoria (ad ogni nodo è associata una pagina), e sarà quindi abbastanza grande (anche se nei nostri esempi per le solite ragioni di rappresentazione useremo degli N più piccoli).
Le caratteristiche dell'R-tree sono quindi:

- L'albero è bilanciato.
- Ogni nodo ha un numero di figli compreso fra un minimo di $N/2$ ed un massimo di N.
- I dati (le regioni) sono nelle foglie.
- Ogni nodo rappresenta implicitamente una regione.
- La root rappresenta l'intero spazio ed ha almeno due figli (amenoché non sia una foglia).

A differenza dei B-tree però abbiamo che:

- Gli spazi rappresentati da due o più fratelli possono essere *non disgiunti*.
- L'unione degli spazi rappresentati da tutti i figli di un nodo *non ricopre* lo spazio rappresentato dal nodo padre.

Esempio di R-tree

Costruiamo insieme un R-tree. Partiamo da quattro regioni, avendo un $N = 4$ (Figura 5.48).
Come vediamo le quattro regioni sono comprese in un nodo unico (poiché $N = 4$). Il nodo radice è quindi il bounding box che racchiude tutti i nodi in esso inclusi (lo vediamo tratteggiato in figura).

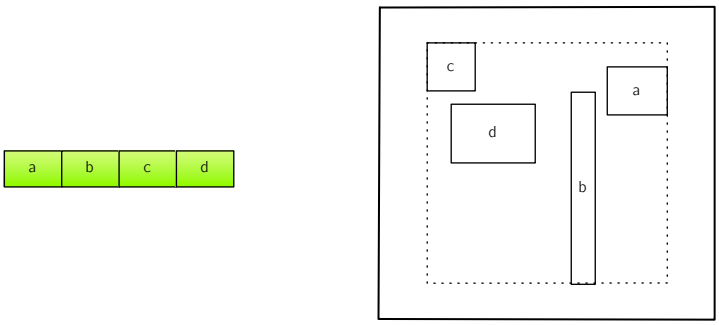


Figura 5.48: Quattro regioni rappresentate in un R-tree.

Rendiamo le cose interessanti aggiungendo una nuova regione:

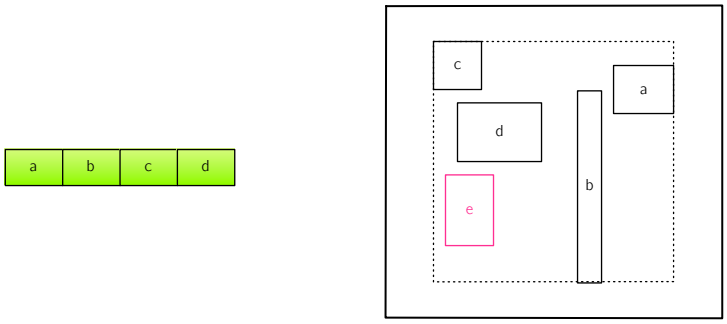


Figura 5.49: Si aggiunge una regione: ma la radice è piena.

Come vediamo la nuova regione non può essere inclusa nella radice che è piena: è necessario splittare la radice. Procediamo in questo modo:

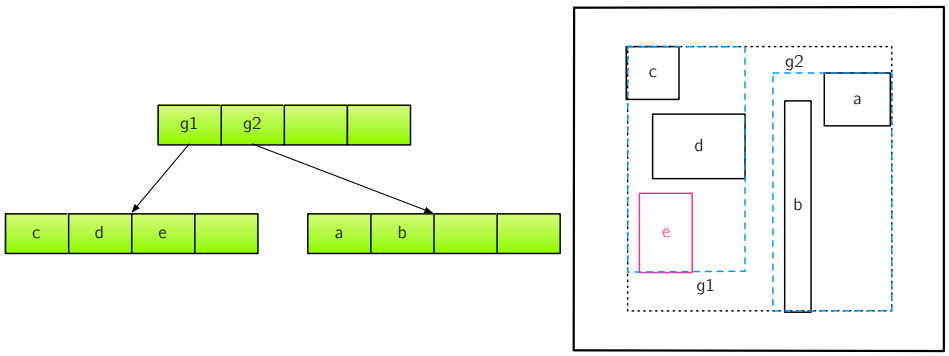


Figura 5.50: Si splitta la radice per fare spazio alla nuova regione.

Sono state create due nuove regioni e sono state inserite a livello di radice, mentre le altre ragioni sono state ripartite all'interno delle due nuove regioni. Il principio secondo cui abbiamo redistribuito le regioni è quello di *minimizzare*

le dimensione dei bounding box. Il calcolo della bounding box è NP-hard, quindi tendenzialmente si prendono i vertici delle due regioni più distanti e si considerano come ''poli d'attrazione'': intorno ad essi si creeranno le due regioni che saranno quindi sufficentemente separate.

Il problema dell'overlapping

Purtroppo gli R-Tree soffrono di una grave problematica relativa all'overlapping delle regioni. Diciamo infatti di avere un $N = 3$ e consideriamo questa configurazione di regioni:

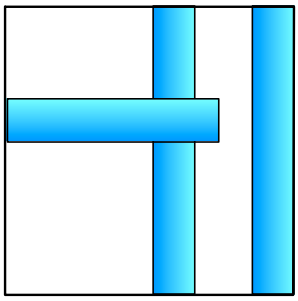


Figura 5.51: Alcune regioni rappresentate da un R-tree con $N = 3$.

Diciamo che giunga una nuova regione e che quindi non ci basti più il nodo radice: ancora una volta dovremo splittare. Ma come?

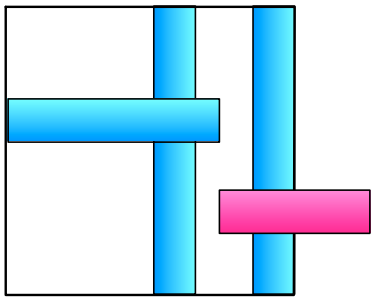


Figura 5.52: Come splittiamo?.

Esistono due principi secondo i quali potremmo andare ad effettuare lo split:

- Minimizzare l'overlap fra le bounding box: è evidente che avere bounding box sovrapposte implichi esplorare entrambi i puntatori che le rappresentano in fase di ricerca, per poi scoprire che magari l'elemento non era presente in nessuna delle due. Si vuole evitare questa situazione.
- Minimizzare l'area totale: avere tanti spazi vuoti all'interno delle bouding box significa effettuare ricerche per oggetti che *starebbero* in una certa

zona, che però essendo vuota non è interessante. Riuscire a eliminare le zone vuote è quindi di primaria importanza.

Ecco quindi l'applicazione dei due diversi split:

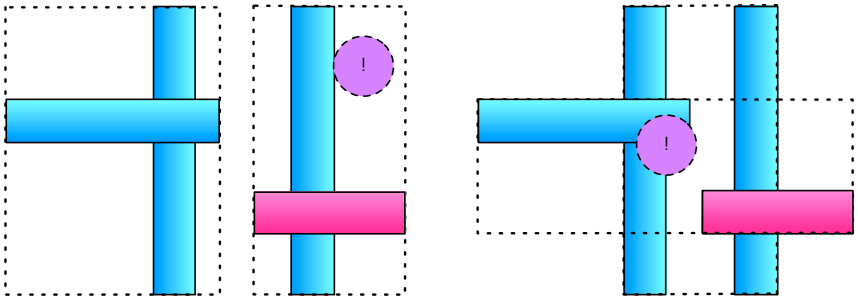


Figura 5.53: Due modi di splittare.

Nell'immagine a sinistra abbiamo minimizzato le bounding box, lasciando però molto spazio vuoto al loro interno. Nell'immagine a destra abbiamo invece minimizzato l'area lasciando però delle zone sovrapposte che quindi richiederanno l'accesso a due pagine diverse di memoria (ricordiamo che ad ogni nodo è associata una pagina in memoria).

Operazioni sugli R-tree

Le operazioni di inserzione, eliminazione e ricerca sono del tutto simili a quelle già accennate per i B-tree.

R^+-trees

Come abbiamo detto gli overlap sono un grave problema: gli R^+-tree mirano a risolverlo. Consideriamo questo R-tree:

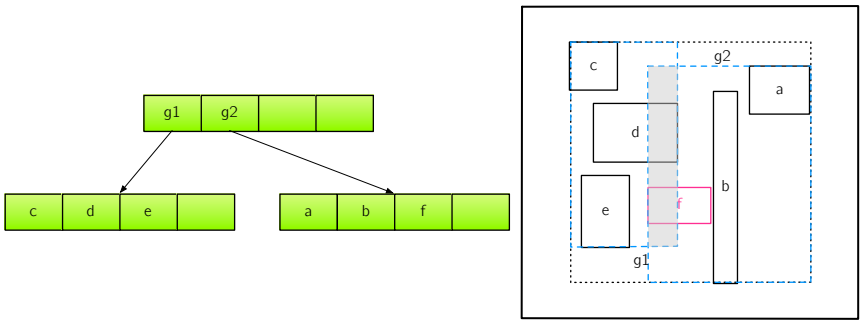


Figura 5.54: Un R-tree che presenta overlap (a causa della regione f).

Gli R^+-tree cercano di risolvere il problema scindendo *la regione che crea*

l'overlap in due parti, in modo da non avere più delle bouding box sovrapposte. Chiaramente questo approccio andrà ad aumentare mano a mano il numero di regioni (e si renderà comunque necessario memorizzare l'informazione relativa al fatto che le due regioni facevano parte di una regione inizialmente unica).

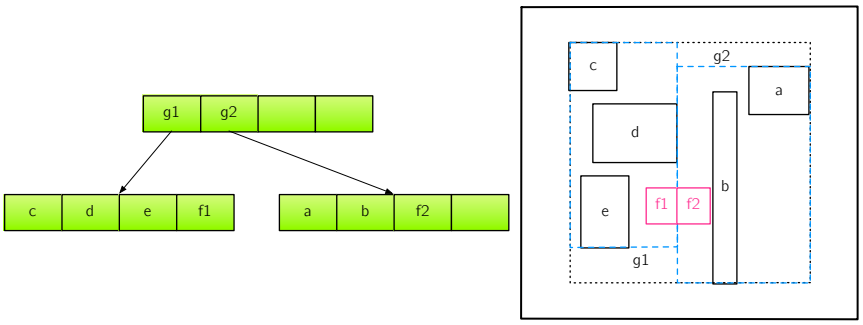


Figura 5.55: Si scindono le regioni per eliminare l'overlap.

Si aggiunge quindi complessità nella struttura al fine di ridurre l'overlap e quindi il costo di ricerca/inserzione/eliminazione.

5.5.9 X-trees

Gli X-tree sono particolari varianti degli R-tree che mirano ancora una volta a risolvere il problema dell'overlapping. Quando infatti il numero delle dimensioni aumenta notevolmente, l'overlapping è particamente certo.
Si introduce per questa ragione il concetto di *supernodo*: quando lo split comporta la creazione di regioni in overlapping, si preferisce non splittare ma assegnare una nuova pagina di memoria al nodo che quindi potrà continuare ad espandersi (superando il limite delle N regioni massime ed acquisendo una capacità di $2N$). Nel caso in cui anche questo nodo si debba riempire, si raddoppierà nuovamente lo spazio ottenendo così nodi man mano più grandi.

Questa tecnica sembra folle ma non lo è: i supernodi (diciamo di dimensione $2N$) fanno sempre accesso a due pagine, ma almeno *sappiamo essere solo due*. Con overlapping crescenti e nodi singoli potremmo avere molte più pagine da dover caricare durante la ricerca di una determinata regione.

5.5.10 TV-trees

I *TV-trees* (Telescopic Vector Trees) forniscono un nuovo approccio al problema dell'indexing e si basano su alcune delle idee già viste quando abbiamo parlato della classificazione. Abbiamo già detto molte volte (e studieremo dopo nel dettaglio) che abbiamo a che fare con la *dimensionality course*: più aumentano le dimensioni più le strutture si rendono ingestibili ed inutilizzabili.

L'idea di fondo è che non tutte le features sono importanti in egual modo: se dunque le ordinassimo dalla più rilevante alla meno rilevante, potremmo trovare una maniera per ridurre i descrittori dei nodi. Più precisamente, possiamo contrarre ed espandere la quantità di feature mantenute nei nodi a seconda delle necessità, col fine ultimo di ridurre al minimo la dimensione dei nodi e della struttura in generale. Si ha quindi che il *fanout non è necessariamente uguale per ogni nodo*. Consideriamo questa tassonomia:

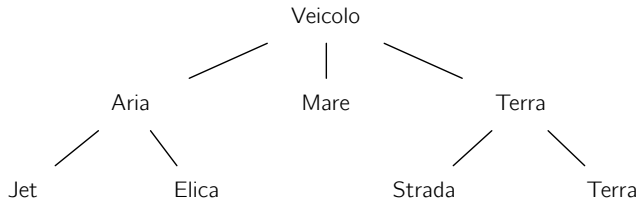


Figura 5.56: Tassonomia dei veicoli (con tre features).

La classificazione richiederà meno features ad alto livello rispetto a quante ne chieda a livelli più bassi.

Nelle strutture viste fin'ora, ogni nodo mantiene tutti i valori di tutte le dimensioni dello spazio: l'idea, qui, è quella di sottointendere alcuni valori data la posizione nell'albero e di esplicitare solo i necessari (e ora capiremo cosa s'intende per necessari).

A livello di foglia manterremo dunque i dati (le regioni) mentre nei nodi interni si avranno le solite MBR (minimum bounding regions), proprio come negli R-tree. I livelli più alti avranno un fan-out maggiore (poiché differenziano di più e quindi vi appartengono più regioni) rispetto a quelli inferiori (che differenziano di meno).
Ogni nodo avrà quindi una suddivisione di feature come segue

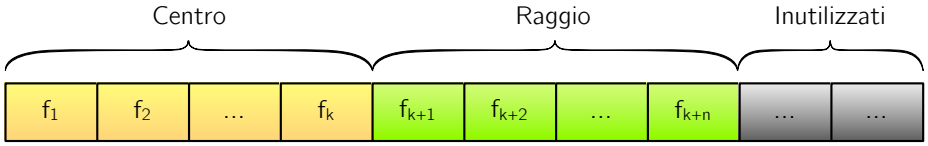


Figura 5.57: Le feature sono in ordine crescente di importanza.

Abbiamo cioè che ogni nodo è caratterizzato da due *insiemi di dimensioni*, una chiamata *raggio* e una chiamata *centro*. Il raggio è una quantità costante per ogni nodo della struttura e vale n. Le regioni mantenute all'interno di un nodo avranno valore comune sulle features facenti parte dell'insieme centro del nodo mentre avranno valori diversi per le features che fanno parte del raggio

(che sono quindi differenzianti). Quando un nodo è pieno, si procederà allo split considerando (nei figli del nodo) un centro più ampio ed un raggio che includerà nuove features precedentemente non considerate.
È quindi evidente che a seconda della quantità e della configurazione dei nodi all'interno dell'albero si potrà avere un centro più o meno ampio, quindi andremo a contrarre o espandere il centro a seconda che vi siano meno o più caratteristiche comuni nelle regioni che un nodo mantiene.

Ritroviamo quindi l'idea che avevamo nella tassonomia: nodi in alto nella struttura avranno alto potere differenziante (poche caratteristiche comuni ma comuni a tutti i nodi di tutti i sottoalberi) mentre nodi più in basso hanno minor potere differenziante (ed un centro più ampio).

Al fine di chiarificare quanto detto, vediamo un piccolo esempio in Figura 5.58.

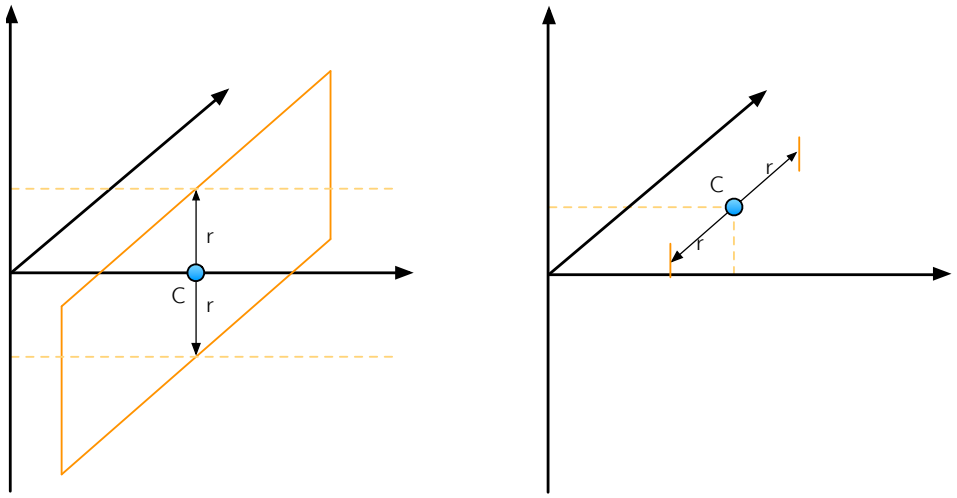


Figura 5.58: Esempio di TV-tree in evoluzione.

Come vediamo, a sinistra abbiamo un centro C che considera la sola dimensione x. Il raggio è poi sulla y, mentre non consideriamo la z.
Supponiamo poi che si saturi il nodo e si debba scindere: il nuovo nodo avrà centro più ampio (x ed y) ed il raggio avrà la sola dimensione z (la figura rappresenta un segmento).

Problematiche relative ai TV-trees

Il problema che affligge i TV-trees è che bisogna essere in grado di ordinare le features e quindi averne una tassonomia.

5.5.11 Pyramid trees

Si prova ancora una volta a risolvere il problema dell'overlapping usando un principio simile a quello visto quando abbiamo parlato delle Z-order curve (cio come scendere di dimensionalità mantenendo il più possibile le distanze coerenti fra loro) per indicizzare su B-tree dati non monodimensionali.
Oltre al problema dell'overlapping vogliamo anche contrastare la dimensionality curse e quindi gli alti costi di inserzione e cancellazione quando dobbiamo lavorare con tante dimensioni.

La prima cosa che si fa è normalizzare lo spazio da 0 a 1 sia sulle x che sulle y (è un passaggio necessario ai fini dei calcoli che faremo in seguito) e si suddivide in quattro piramidi (da cui il nome della struttura). Ad ogni piramide è associato un indice: Dopodiché si passa ad una fase in cui si determina l'appartenenza di

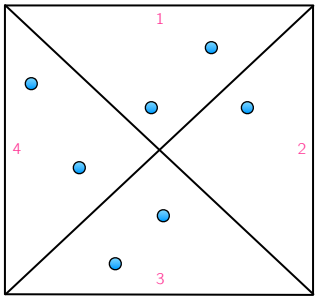


Figura 5.59: Lo spazio è normalizzato e suddiviso in quattro piramidi.

ogni punto ad una certa piramide. Per compiere questa operazione si ragiona sulle coordinate del punto. Giacché lo spazio è stato suddiviso dalle due rette:

- $r_1 : y = x$.
- $r_2 : y = -x + 1$.

Sarà sufficente usare le equazioni delle rette come disequazioni per determinare la piramide di appartenenza di un punto. Più precisamente:

- Se la $y > x$, ci troviamo nella piramide 1 o nella piramide 4.
- Se la $y < x$, ci troviamo nella piramide 2 o nella piramide 3.
- Se la $y > -x + 1$, ci troviamo nella piramide 1 o nella piramide 2.
- Se la $y < -x + 1$, ci troviamo nella piramide 3 o nella piramide 4.

Incrociando le due informazioni sarà quindi estremamente semplice determinare la piramide di appartenenza.

Una volta identificata la piramide di appartenenza si calcola la distanza del

punto rispetto al centro (che sarà sempre minore di 0,5): la coppia (indice_piramide, distanza_centro) identifica quindi quel punto: È evidente che

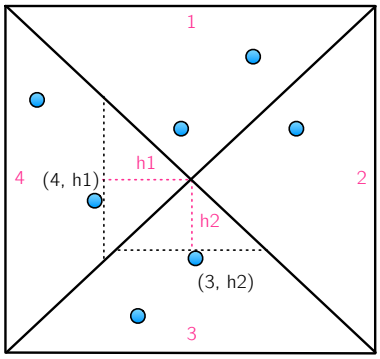


Figura 5.60: Due punti vengono identificati.

punti allineati orizzontalmente o verticalmente ed appartenenti alla stessa piramide abbiano lo stesso identificativo, cioè faranno parte di uno stesso insieme che eventualmente verrà sviscerato in fase di restituzione del risultato della query. Il punto focale è che grazie a questa identificazione è possibile memorizzare i punti in una struttura monodimensionale.

Piramidi a fette

La struttura in realtà viene utilizzata in modo leggermente diverso: si definiscono delle fasce all'interno delle piramidi e vengono indicizzate le distanze fra le fasce ed il centro (onde evitare di avere moltissimi punti all'interno dell'indice).

Si noti però che a causa della dimensionality curse, con l'allontanarsi dal centro (cioè dalle punte delle piramidi) lo spazio coinvolto aumenta (e con distribuzione uniforme dei dati, anche il numero di punti).
Per questa ragione fette che si trovano più distanti dal centro sono più piccole rispetto quelle al centro:

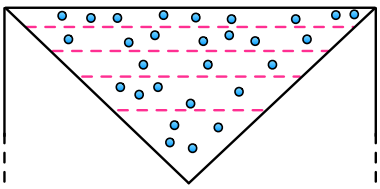


Figura 5.61: Fette più distanti dal centro sono più piccole.

5.5.12 La dimensionality curse

Come cappello finale, definiamo con più precisione il problema della dimensionalità (il cosiddetto dimensionality course). Tale problema indica la crescita esponenziale del numero di puntatori necessari con conseguente spreco di spazio e peggioramento delle prestazioni in fase di interrogazione (nei quadtree).

Per capire in termini numerici di cosa stiamo parlando, consideriamo uno spazio tridimensionale e tre diverse query per range (rappresentata dalle tre sfere concentriche in Figura 5.62). Le query per range hanno raggio r, $2r$ e $3r$.

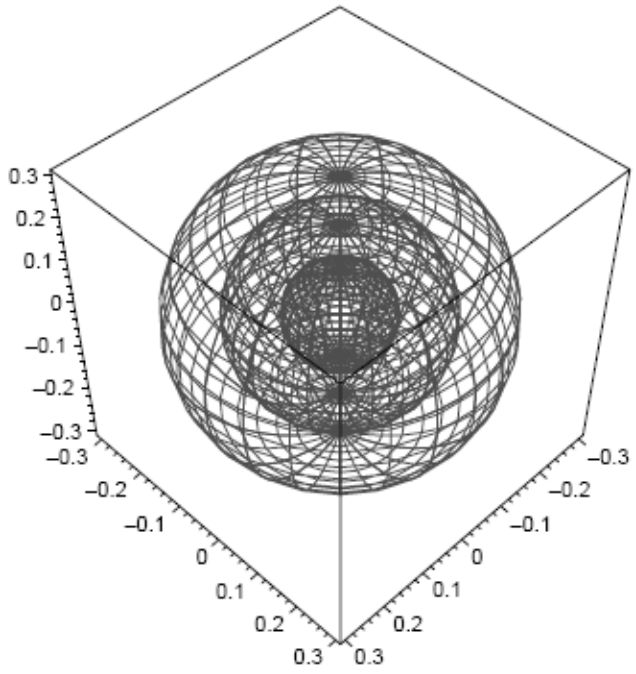


Figura 5.62: Tre query di range.

Di queste tre sfere possiamo calcolare le varie fette, cioè la differenza fra una e l'altra come segue:

$$\text{Prima fetta: } 4/3\pi r^3$$
$$\text{Seconda fetta: } 4/3\pi((2r)^3 - r^3)$$
$$\text{Terza fetta: } 4/3\pi((3r)^3 - (2r)^3)$$

Abbiamo calcolato il volume della seconda e della terza fetta semplicemente sottraendo i volumi delle sfere in esse incluse.

Cosa notiamo? In uno spazio n-dimensionale se il numero di punti nella sfera più interna è pari a k, allora:

- Il numero di punti nella seconda fetta è pari a $O(2^{n-1}k)$.
- Il numero di punti nella terza fetta è pari a $O(3^{n-1}k)$.
- Il numero di punti nella quarta fetta è pari a $O(4^{n-1}k)$.
- ...

Questo significa che all'aumentare delle dimensioni la maggiorparte dei punti si trovano nella slice più esterna, ovvero, sono molto distanti sia fra loro che dai punti interni!
Ecco spiegato matematicamente il fenomeno contro il quale ci siamo battuti per l'intero capitolo.

5.5.13 Intervalli temporali: segment tree

Introduciamo anche una struttura ad indice utile per rappresentare intervalli temporali. Nella fattispecie potremmo voler rappresentare vari eventi che accadono in un video in modo da poter successivamente interrogare il contenuto multimediale.

La prima cosa da fare è *identificare la presenza deli oggetti* all'interno del video. Più precisamente vogliamo sapere in quali frame un certo oggetto è presente. Consideriamo questo grafico:

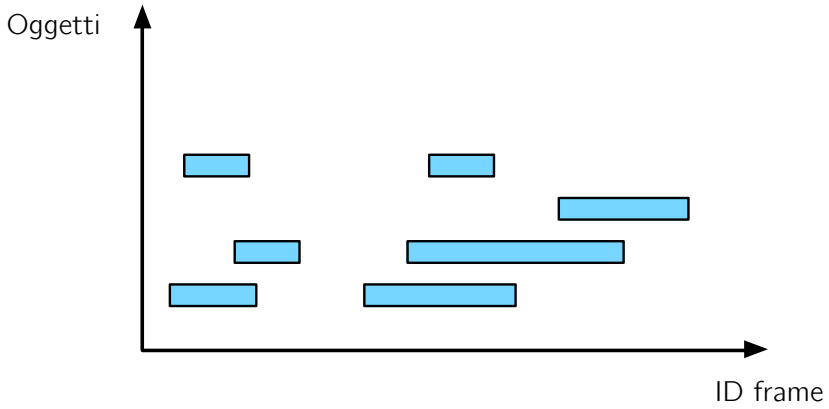


Figura 5.63: Grafico Oggetti/Frame.

Sulle y abbiamo la lista degli oggetti del video mentre sulle x abbiamo l'identificatore di ogni frame (possiamo pensarlo come un intero). Chiaramente per poter tracciare questo grafico si rende necessario elaborare il video (non è importante come vengono esaminati i frame).

Possiamo facilmente tradurre il grafico in forma tabellare:

Oggetto	Inizio	Fine
$obj1$	10	20
$obj1$	60	70
$obj2$	8	40
$obj2$	65	86
$obj2$	110	126
$obj3$	35	75
$obj3$	90	120

Il procedimento di indicizzazione

Adesso che siamo in grado di dire quali oggetti appaiono in quali frame, iniziamo a lavorare sui dati per poterli indicizzare. Supponendo di avere n frame totali, consideriamo come istanti tutti i frame di inizio e di fine e vi aggiungiamo 0 ed n, dopodiché li ordiniamo in ordine crescente.

Tornando all'esempio di prima e supponendo che il numero totale di frame fosse $n = 126$, otteniamo questa sequenza di istanti (si noti l'aggiunta dello 0 di 126 che indicano gli eventi di inizio e fine video):

$$0, 8, 10, 20, 35, 40, 60, 65, 70, 75, 86, 90, 110, 120, 126.$$

Tutti questi istanti rappresenteranno le *foglie dell'albero*. Da qui, possiamo iniziare a dare forma alla struttura costruendo nuovi intervalli a due a due a partire dalle foglie (si tratta quindi di una procedura bottom-up). Supponendo di considerare quindi tre foglie a, b e c, il nodo padre di a e b sarà un nodo che rappresenta l'intervallo temporale $[a, c)$, (attenzione: chiuso a sinistra e aperto a destra). Andiamo quindi a creare il primo livello (dal basso) del nostro segment tree:

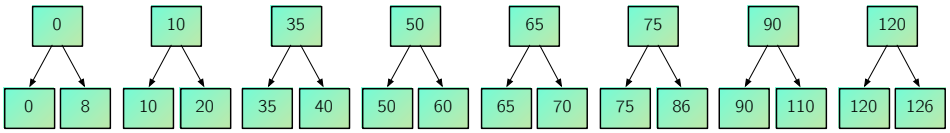


Figura 5.64: Nuovi intervalli vengono generati a partire dalle foglie.

Se prendiamo per esempio il nodo etichettato con 10 appena generato, stiamo rappresentando tutto l'intervallo temporale che va dal frame 10 (incluso) al frame 35 (escluso). In generale si ha quindi che un nodo x rappresenta l'intervallo $[x, y)$ dove y è il label del suo fratello destro.

Procedendo in questo modo arriviamo facilmente a definire l'intero albero che sarà in grado di indicizzare tutti gli oggetti presenti del video:

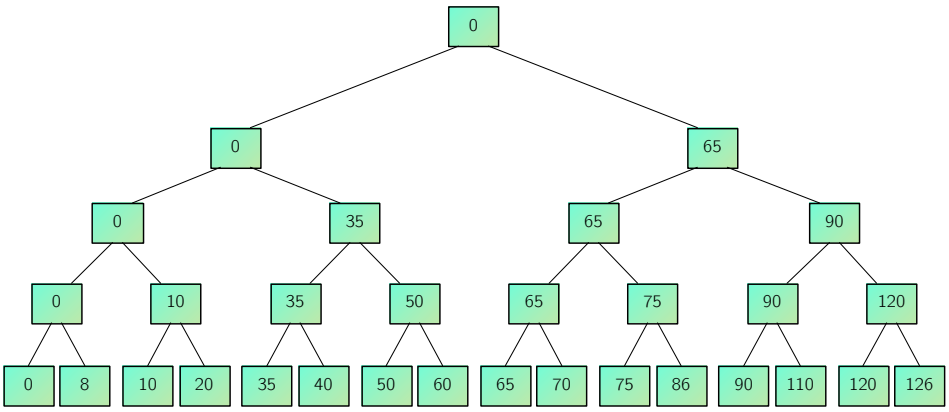


Figura 5.65: Il segment tree albero che conterrà gli oggetti dell'intero video.

Inserire gli elementi

Non resta che inserire i vari oggetti nella neonata struttura. Partiamo dall'oggetto $obj1$ che ha intervallo associato $[10, 20)$ e confrontiamolo con la regione rappresentata dalla radice (che di default è $[0, 126]$): c'è intersezione ma non è coperta tutta la zona della radice, dunque tutti i figli devono essere esaminati. Il nodo 0 (intervallo $[0, 65)$) ha intersezione con l'intervallo di $obj1$ ma quest'ultimo non ricopre tutta la regione: i figli del nodo 0 devono essere esaminati. Il nodo 65 (intervallo $[65, 126]$) invece non ha intersezione con l'intervallo di $obj1$, quindi non consideriamo né lui né i suoi figli.
Proseguiamo quindi con i nodi (chiamiamoli con l'intervallo che rappresentano onde evitare confusioni):

- $[0, 35)$: c'è intersezione con l'intervallo di $obj1$, ma quest'ultimo non copre tutto l'intervallo quindi scendiamo ancora.
- $[35, 50)$: non c'è intersezione con l'intervallo in cui appare $obj1$, quindi possiamo non considerarlo (e non considerare il suo sottoalbero).

Abbiamo quindi i due figli del nodo $[0, 35)$ da esaminare:

- $[0, 8)$: non c'è intersezione con l'intervallo in cui appare $obj1$, quindi possiamo non considerarlo (e non considerare il suo sottoalbero).
- $[10, 35)$: c'è intersezione con l'intervallo di $obj1$, ma quest'ultimo non copre tutto l'intervallo quindi scendiamo ancora.

A questo punto inseriamo l'oggetto nella foglia $[10, 20)$ (che rappresenta esattamente l'intervallo in cui l'oggetto $obj1$ si trova).

Andiamo quindi ad inserire un oggetto in un nodo solamente quando l'intervallo in cui appare l'oggetto è *esattamente* quello rappresentato dal nodo, senza

''avanzare'' spazio nell'intervallo del nodo. Se durante la fase di ordinamento degli intervalli questi erano abbastanza disgiunti, avremo molti oggetti nelle foglie e pochi nei livelli superiori, viceversa, nel caso in cui gli intervalli fossero molto sovrapposti fra loro gli oggetti si concentreranno nei nodi più in alto nella struttura.

Abbiamo un altro intervallo associato ad $obj1$, cioè $[60, 70)$. Seguendo il procedimento di prima, assegneremo alla foglia 60 e alla foglia 65 l'oggetto $obj1$ (infatti l'intervallo di $obj1$ ricopre totalmente i due intervalli rappresentati dalle foglie).

Ricerca degli elementi

La ricerca si sviluppa in modo del tutto simile a quanto visto per l'inserimento. Si noti che è importante capire con che semantica cerchiamo l'elemento: potremmo desiderare tutti gli oggetti *sempre presenti* in un certo intervallo oppure quelli *presenti in almeno un istante dell'intervallo*: nel primo caso si dovranno considerare come papabili più nodi durante la discesa e poi si dovranno unire i risultati alla ricerca di quegli oggetti che effettivamente sono sempre presenti nell'intervallo desiderato.

Altre operazioni interessanti

È possibile anche usare il segment tree per:

- Trovare tutti gli oggetti che appaiono prima o dopo un certo frame f.
- Trovare tutti gli oggetti che appaiono in maniera continua in un certo intervallo $[f, f']$ (l'abbiamo accennato prima).
- ecc.

Estensione degli R-tree

Volendo è possibile usare gli R-tree al posto dei segment tree: è sufficiente considerare il grafico che abbiamo visto poco fa come uno spazio dove ogni rettangolo rappresenta una regione: allora è possibile tracciare dei bounding box e poi indicizzare quelli all'interno di un R-tree.

5.6 Tecniche per implementare il clustering

Tutto quello che ci siamo detti fin'ora è bello e funzionante, ma abbiamo sempre dato per scontato un fatto: conoscevamo quali fossero le feature e che valore queste assumessero per ogni oggetto. Inoltre, conoscevamo le metriche adottate per confrontare gli oggetti fra loro.

Purtroppo, non si è sempre così fortunati. Andremo ora a fare una carrellata di situazioni nelle quali potremmo trovarci, mano a mano più imprecise ed indeterminate. Ricorreremo spesso ai clustering o comunque a tecniche alternative rispetto a quelle studiate fino a questo momento.

5.6.1 Conoscere le distanze ma non le coordinate (MDS)

La prima situazione che studiamo è tale per cui si conoscono le distanze reciproche fra gli oggetti, ma non la loro posizione precisa. Ad esempio potremmo sapere che tra un ospedale ad un altro vi sono 2km, ma non conoscere le coordinate cui i due si trovano. La prima idea che ci viene in mente, la più banale, potrebbe essere quella di indovinare le coordinate basandoci sulle distanze che conosciamo.

Multi Dimensional Scaling

Una tecnica più raffinata è invece chiamata *Multi Dimensional Scaling* (in breve MDS) che mira a trovare le coordinate dei punti (che potremo poi usare in un indice di quelli studiati fin'ora) ed eventualmente ad intuire la dimensionalità dello spazio in cui stiamo lavorando (se non la conosciamo, cosa che non è detta). In questo modo si delineano le feature specifiche e si rendono metriche le distanze (che eventualmente non lo erano).
Prima di vedere nel dettaglio il funzionamento del MDS, vediamo un piccolo esempio. Si hanno quattro punti e si conoscono le distanze (la distanza fra a e b è indicata come $d(a, b)$):

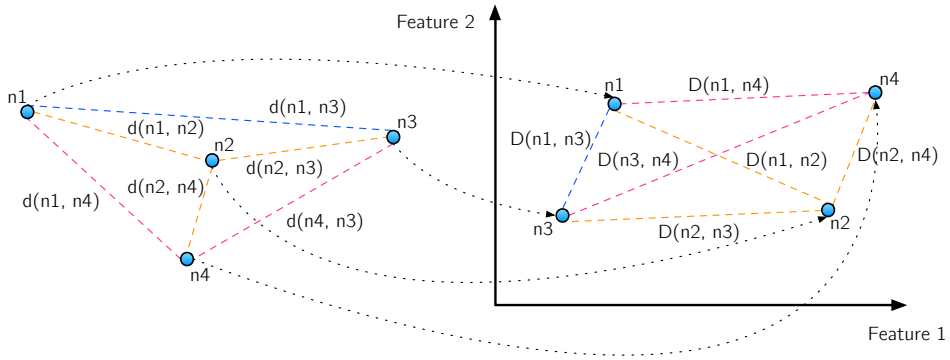


Figura 5.66: Applicazione dell'MDS (spazio bidimensionale).

Durante il processo si è cercato di mantenere coerenti le distanze (cioè ogni $(d(a, b) \approx D(a, b))$.

Si noti che l'MDS è usabile anche per scendere di dimensionalità, ma a differenza di quanto facevano le space filling curves (Sezione 5.4) che spazzavano

tutto lo spazio usando un algoritmo predefinito, quindi la trasformazione avviene basandosi sui punti stessi (quindi non esiste una tecnica a priori, ma è data driven). Quando si usa l'MDS in questa modalità, chiaramente si conosce la dimensione di partenza e quella a cui si vuole arrivare.

Si noti che comunque si potrebbero *conoscere alcune posizioni dei dati*, ma scegliere comunque di adottare questa metodologia (non è quindi corretto dire che è impossibile usare questa metodologia se si hanno le effettive posizioni degli oggetti). Se si dispone di queste informazioni, comunque, è possibile integrarle all'interno dell'MDS.

Il funzionamento. Ora che abbiamo capito che cosa faccia l'MDS, capiamo come funziona.
Si parte da una dimensione bassa (a piacere) entro la quale i punti vengono sistemati in maniera casuale. Dopodiché si calcola la funzione di *stress*:

$$stress = \sqrt{\frac{\sum_{i,j}(d'_{ij} - dij)^2}{\sum_{i,j} d_{ij}^2}}$$

La funzione di stress misura quando le distanze nello spazio che stiamo creando di discostano dalle distanze che conoscevamo. La funzione di stress è associata ad una configurazione particolare di tutto lo spazio (infatti coinvolge nel calcolo ogni coppia i, j di punti).

È quindi sufficente cercare di muovere i punti nello spazio al fine di minimizzare lo stress, fino ad arrivare ad una certa tolleranza. Dopo alcune iterazioni, se non si riesce a raggiungere un valore sufficentemente piccolo di stess, si aggiunge una dimensione e si ricomincia.

Drawbacks. L'MDS è utilizzabile solamente se conosciamo le query che faremo: ogni nuovo oggetto (espresso nella query) richiederebbe il ri-calcolo dell'intera struttura (per far sì di considerare la distanza del nuovo oggetto nello spazio che abbiamo generato artificialmente).

5.6.2 Introduzione ai metodi di clustering

Diciamo di non essere interessati ad individuare le coordinate e quindi non vogliamo usare un indice: ci limitiamo a raggruppare oggetti simili (anche qui a seconda delle informazioni di cui disponiamo applicheremo tecniche diverse). Il risultato che vogliamo ottenere sono alcuni insiemi di oggetti *simili fra loro*. Per ogni gruppo vi sarà poi un oggetto particolare, detto *rappresentante*, che verrà usato in fase di query per identificare quali cluster devono essere restituiti e quali no.

Vediamo in Figura 5.67 quattro cluster coi loro rappresentanti ed una query Q.

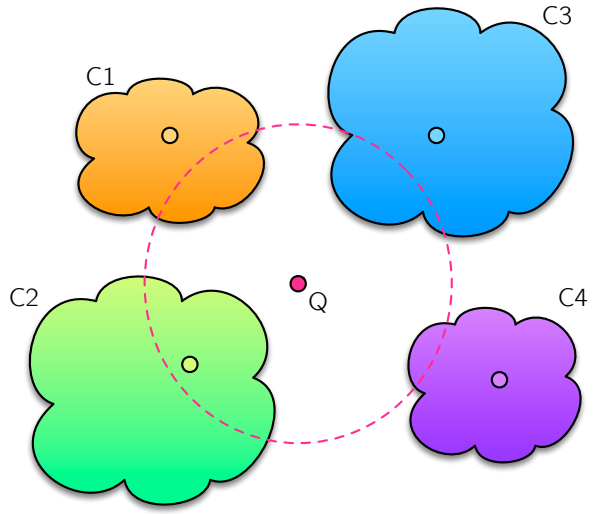


Figura 5.67: Solamente il cluster $C2$ è restituito.

Il cluster che verrà restituito sarà unicamente $C2$, poiché l'unico il cui rappresentante è incluso del range della query.

Esistono due grandi famiglie di metodi per ottenere i cluster a partire dai dati (e dalla loro similarità reciproca):

- Metodologie *sound*, che presuppongono la conoscenza della matrice di similarità A.
- Metodologie *iterative*, che costruiscono i cluster in maniera incrementale sfruttando i vettori di features.

Vediamoli nel dettaglio.

5.6.3 Sound clustering

Come detto sopra, è necessario conoscere la similarità fra ogni coppia di oggetti. Si vuole creare un grafo che descriva la similarità fra gli oggetti (ogni oggetto è un nodo) e per ottenerlo si definisce innanzi tutto una *soglia* T di similarità, dopodiché si considera ogni coppia di oggetti e si confronta con la soglia T:

- Se $sim(o1, o2) < T$, quindi la similarità è ampia, non si fa nulla.
- Se $sim(o1, o2) >= T$ quindi la similarità è oltre la soglia accettabile, si crea un arco fra $o1$ e $o2$ all'interno del grafo che stiamo costruendo.

Dopo aver effettuato questa computazione, potremmo quindi ottenere un risultato del genere:

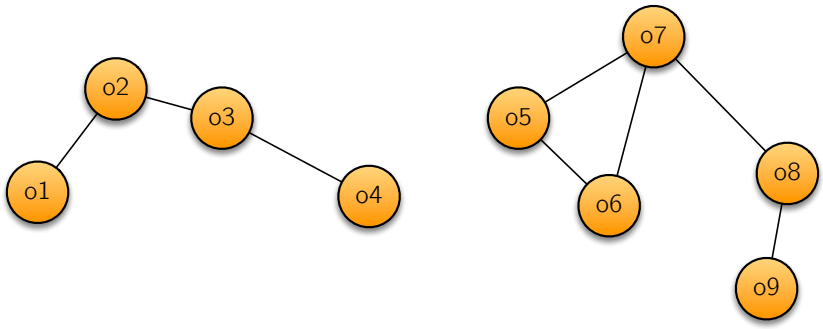


Figura 5.68: Grafo di similarità da cui trarre i cluster.

Dal grafo ai clusters

Una volta generato il grafo è necessario interpretarlo per delineare i diversi cluster. Esistono diversi modi di compiere questa operazione, consideriamone due.

Considerare componenti connesse. Considerare le componenti connesse è certamente il modo più semplice di procedere e ci darebbe questo risultato:

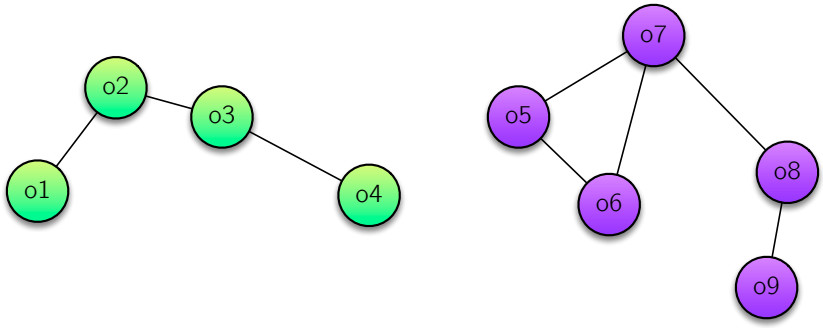


Figura 5.69: Si delineano due clusters diversi.

È evidente che una tecnica del genere sia troppo semplicistica: $o1$ ed $o4$ sono relativamente distanti fra loro, eppure appartengono allo stesso cluster.

Considerare cliques. Una tecnica alternativa e più fine è quella di considerare le clique. Le clique sono parti di grafo totalmente connesse fra loro. Purtroppo il calcolo delle clique è NP-hard inoltre si delina il problema degli overlap fra

diversi cluster. Lo vediamo bene nel nostro esempio: (ogni cluster è stato racchiuso da un cerchio tratteggiato):

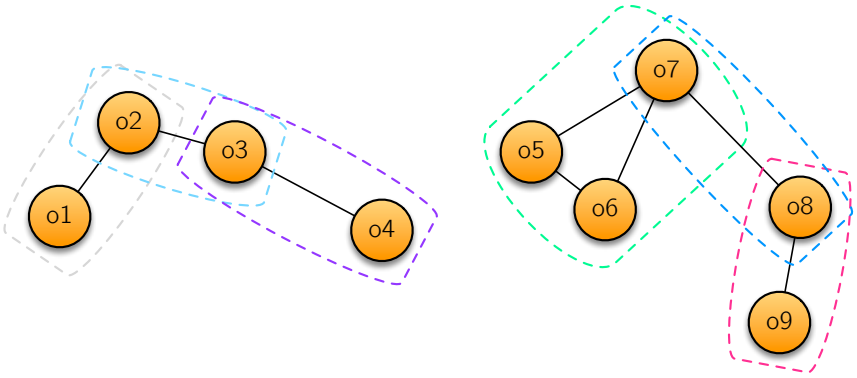


Figura 5.70: Si delineano sei clusters diversi.

5.6.4 Iterative clustering

L'iterative clustering è un processo incrementale (anche chiamato *leader algorithm*) che costruisce i cluster man mano annettendovi degli oggetti (o creandone dei nuovi per troppa dissimilarità). Anche in questo caso si usa una certa soglia T definita a priori. Più precisamente si parte scegliendo un oggetto a caso e rendendolo un cluster.
Dopodiché, si procede scegliendo iterativamente seguendo questi passi:

1. Si sceglie un oggetto o a caso.
2. Si trova il cluster più vicino ad o (chiamiamolo c).
3. Se la distanza fra o e c è minore della soglia T, allora o è incluso nel cluster c. Altrimenti, o genera un nuovo cluster a sé stante e ne diventa il rappresentante.

Questo ciclo viene ripetuto finché tutti gli oggetti non appartengono ad un cluster.

Come valutare la distanza fra cluster (nel leader algorithm)

Non resta che capire come valutare le distanze fra il cluster e l'oggetto. Abbiamo diverse alternative papabili.

- Usare il *rappresentante* del cluster (il primo che lo ha generato).
- Usare il *centroide*, cioè il centro del cluster. Tale valore deve però essere ri-calcolato ad ogni iterazione (se il cluster intanto è cambiato). Eventualmente si può aggiornare il centroide ogni tot di tempo.

5.6.5 Non conoscere la soglia

Continuiamo con la nostra opera di riduzione di conoscenza e rimuoviamo un ulteriore elemento: la soglia. Abbiamo fatto grande uso della soglia nelle tecniche esposte nei due paragrafi sopra. Che fare, dunque?

Si inzia dando una stima di T, chiamiamola T', e si costruisce così il grafo delle distanze reciproche. Si noti che la T' non deve essere necessariamente definita globalmente, ma può anche essere locale: ad esempio se c'è una zona di dati molto connessi fra loro, basterà una soglia T' piccolina per identificare un dato fuori dal cluster; invece se abbiamo dati molto separati, la T' dovrà essere ancora più grande per poter identificare nuovi cluster. Una scelta erronea della T' può portarci ad avere clusters enormi e poco rappresentativi piuttosto che una miriade di clusters piccolini e poco significativi.

Una volta ottenuto il grafo usando la T' stimata, si procede calcolando il minimum spanning tree (in breve MST) del grafo in input (che quindi riduce il numero di archi da un $O(N^2)$ ad un $O(N)$). Di tale grafo si va ad esaminare ogni nodo e gli archi ad esso associati: si rimuovono quegli archi che hanno una distanza maggiore della media degli altri archi del nodo preso in esame.
Una volta compiuta questa operazione possiamo procedere studiando le clique o le connected components come abbiamo imparato a fare poco fa.

5.6.6 Annettere un oggetto al cluster

Quando arriva un nuovo oggetto nella base di dati, questo verrà tendenzialmente incluso nel cluster a lui più vicino. Ma questo non è necessariamente vero quando si hanno diversi cluster nei pressi dell'oggetto (più precisamente vi sono più rappresentanti entro la tolleranza T). Vi sono tre princìpi (ognuno coi suoi pro e contro) che si possono addotare quando ci si trova nella situazione di dover scegliere in quale cluster (fra quelli a distanza al massimo T) inserire un oggetto:

- Un cluster a caso: è un approccio veloce, ma ovviamente non fornisce grandi garanzie.

- Il cluster effettivamente più vicino: è una buona idea poiché mantiene i cluster compatti, ma rischia di generare cluster enormi e lasciare cluster molto piccoli.

- Il cluster più piccolo: distribuisce in maniera più equa gli oggetti nei cluster e fornisce così alta entropia.

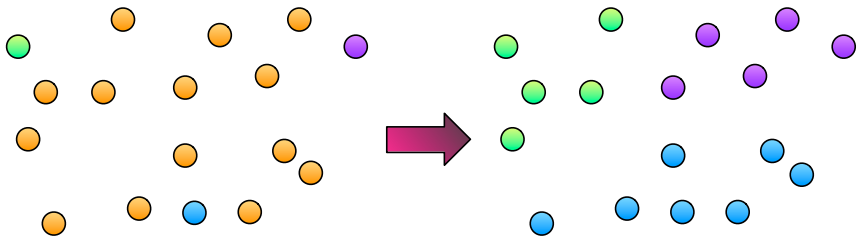
5.6.7 Ottenere un numero definito di clusters

Cambiamo ora filosofia: diciamo di non essere interessati ad ottenere dei cluster basandoci (solamente) sulle distanze reciproche, ma siamo prima di tutto limitati da una quantità r di cluster che vogliamo ottenere. Esistono diverse tecniche per ottenere questo risultato, vediamole insieme.

Max-a-min

Si inizia scegliendo un elemento a caso che sarà il leader di un cluster. Dopodiché, volendo generare r clusters, si sceglieranno gli $r-1$ nodi più *distanti* dal leader scelto casualmente. Abbiamo ottenuto in questo modo gli r rappresentanti del cluster.

Si procede dunque esaminando tutti i dati rimasti ed assegnandoli al cluster più consono (confrontandoli con il leader di ogni cluster). Vediamo un esempio con $r = 3$:



I leader sono scelti.

Ogni nodo è associato a cluster più vicino.

Figura 5.71: Esempio di esecuzione di max-a-min.

K-means

L'algoritmo K-means (dove K è il numero di cluster che si desidera) fornisce una miglioria iterativa all'algoritmo max-a-min. L'intento è quello di *minimizzare una funzione di costo globale*, e proviamo ad ottenere questo risultato spostando gli elementi da un cluster ad un altro. Se uno spostamento diminuisce la funzione di costo, allora viene mantenuto. Altrimenti se ne prova un'altro.

Ma quale funzione potremmo usare?

Primo criterio: compactness. La prima funzione globale che possiamo scegliere è la *compactness*, ovvero quanto i cluster sono compatti fra loro. Per

ogni cluster m viene calcolata la funzione:

$$RMSE_m = \sqrt{\frac{1}{N}\sum_{i=0}^{N}[\hat{F}_m(z_i) - F_m(z_i)]^2}$$

dove abbiamo che N è il numero totale di elementi nel cluster m, $\hat{F}_m(z_i)$ è il valore del rappresentante del cluster m e $F_m(z_i)$ è il valore dell'elemento considerato: se ne fa la differenza per calcolare la distanza fra il rappresentante e l'elemento preso in esame. Tale calcolo è fatto per ogni elemento del cluster e poi se ne fa la media.
Dopodiché viene ancora calcolata una ulteriore media fra tutti gli $RMSE_m$ usando questa formula:

$$RMSE = \frac{1}{M}\sum_{m=1}^{M} RMSE_m$$

dove M è il numero totale di clusters.
L'intenzione è quindi quella di *minimizzare* il valore di RMSE, che viene ricalcolato dopo ogni scambio.

Secondo criterio: entropia. Un altro criterio adottabile è l'entropia, che chiaramente voglia massimizzare. Come sappiamo l'entropia (Sezione 2.2.1) si calcola come:

$$H(X) = \sum_{x \in \mathbb{A}_X} P(x) \log\left(\frac{1}{P(x)}\right) \tag{5.1}$$

Trovare la massima entropia significa avere il massimo equilibrio fra i cluster.

5.6.8 Non conoscere le distanze avendo il numero di clusters

Continuiamo a togliere conoscenza: se non sappiamo neanche le distanze fra gli oggetti? Dobbiamo *impararle* dall'utente, sfruttando l'user feedback (che approfondiremo in seguito) oppure studando i pattern d'accesso degli utenti.
Ad esempio su un corso moodle potremmo non conoscere la distanza che c'è fra le slides del corso di DBM rispetto a quello di IALab, ma molti studenti scaricano entrambi i materiali nello stesso periodo di tempo: possiamo sfruttare questa informazione?

Confidence clustering

Attenzione: assumiamo che ogni cluster possa avere al massimo r oggetti al suo interno, e sappiamo anche il numero di cluster a priori.

Il confidence cluster è una tecnica abbastanza semplice: a partire da una distribuzione casuale degli oggetti nei clusters, sfruttiamo gli accessi per spostare gli oggetti nei cluster più promettenti. Si noti che l'algoritmo funziona *quando c'è concordanza latente* (cioè in effetti è possibile clusterizzare gli oggetti e non sono tutti scorrelati fra loro), altrimenti continuerà a spostare gli elementi senza mai situarli in cluster precisi.
Facciamo uso della funzione di confidenza $conf(i, j)$ dove i è l'indice di un oggetto della base di dati e j è l'indice di un cluster per sapere con quanta certezza un certo oggetto si trova dentro un certo cluster. Il valore della funzione varia da 0 a 10 (compresi).

Il primo passo, come già accennato, consiste nell'assegnare ogni oggetto all'interno di un cluster con un valore di confidenza casuale. Dopodich, quando l'utente fa accesso a due oggetti, chiamiamoli o_a ed o_b, si presentano alcune casistiche:

- Se o_a ed o_b si trovano nello stesso cluster C_j allora:
  - $conf(a, j) + +$
  - $conf(b, j) + +$
- Se o_a si trova nel cluster C_i ed o_b si trova nel cluster C_j allora:
  - Se $conf(a, i) > 1$ e $conf(b, j) > 1$ (cioè siamo *abbastanza* sicuri che i due oggetti si trovino comunque nei cluster che gli si confanno):
    - $conf(a, j) - -$
    - $conf(b, j) - -$
  - Se $conf(a, i) == 1$ e $conf(b, j) == 1$ (cioè non siamo poi così sicuri che *entrambi* gli oggetti si trovino nel corretto cluster):
    - $conf(b, i) = 1$, cioè si muove l'oggetto o_b dal cluster C_j al cluster C_i (potremmo anche muovere o_a, è indifferente)
    - $conf(c, j) = 1$, cioè un certo oggetto $o_c \in C_i$ è spostato in C_j (per equilibrare i cluster). Tendenzialmente si sceglierà un o_c con $conf(c_i) = 1$.
  - Se $conf(a, i) > 1$ e $conf(b, j) == 1$ (cioè uno dei due oggetti siamo *abbastanza* sicuri essere nel cluster corretto):
    - Verifichiamo l'esistenza di un certo o_c per cui $conf(c, i) == 1$ (un oggetto che quindi scambieremmo volentieri con o_b)
      - Se tale o_c *esiste*, allora si setta $conf(b, i) = 1$, cioè si muove l'oggetto o_b dal cluster C_i al cluster C_j e poi si setta $conf(c, j) = 1$ cioè o_c viene messo al posto di o_b.

· Se tale o_c *non esiste*, si fa $conf(a, i) - -$, cioè si riduce la confidenza di o_a senza intaccare nulla.

Nel caso in cui ci sia una correlazione fra l'uso degli oggetti, i valori di confidenza andranno a stabilizzare, altrimenti continueranno ad oscillare.

5.6.9 Non conoscere le distanze né il numero di cluster

Sempre più difficile: non conosciamo le distanze e non abbiamo neanche un numero di cluster a priori. Possiamo ancora una volta far uso dei pattern d'accesso dell'utente.

Adaptive clustering

Si inizia assegnando in maniera randomica una posizione su una retta ad ogni oggetto del database. Quando due oggetti hanno un accesso simultaneo, vengono avvicinati sulla retta.
Procedendo in questo modo, se c'è stabilità, alla lunga si formeranno dei gruppi di elementi che saranno i nostri clusters:

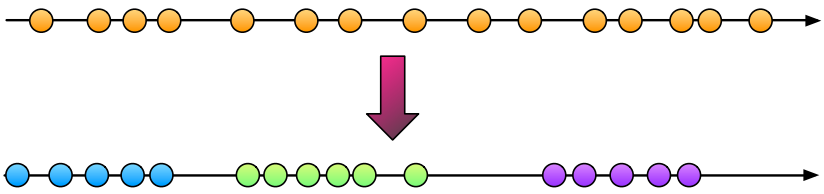


Figura 5.72: Risultato dell'esecuzione dell'adaptive clustering.

5.6.10 Stimare il numero di cluster del database

Alcuni dei metodi che abbiamo studiato richiedono in input il numero di cluster attesi (ad esempio il k-means). Vediamo dunque come stimare questa quantità per poterla dare in pasto a qualche algoritmo.
Sfruttiamo una grandezza che prende il nome di *covering* ed è così definita:

$$covering(o_i, o_j) = \sum_{k=1}^{n} p(k|o_i)p(o_j|k)$$

Dove abbiamo che:

- L'indice k scorre fra tutte le feature dei due oggetti o_i e o_j.
- $p(k|o_i)$ ci dice quanto, data una certa feature f_k, l'oggetto o_i ne goda. Quindi in sostanza quanto la feature f_k sia importante in o_i.

- $p(o_j|k)$ esprime il concetto inverso: cercando f_k qual è la probabilità che questa sia presente in o_j? Stiamo quindi esprimendo la probabilità che o_j sia un documento che gode della feature f_k.

Il covering, dunque, ci dice quanto due oggetti sono *comuni* fra loro.

Supponiamo ora di avere un database che è un cluster perfetto:

- Ogni feature è uniformemente distribuita.
- Ogni documento gode della stessa probabilità di essere selezionato.

Possiamo prendere il covering e andare a sostituire i valori (per quanto riguarda questo particolare database):

$$covering(o_i, o_j) = \sum_{k=1}^{n} \frac{1}{n}\frac{1}{D}$$

dove D è il numero totale di documenti e n è il numero totale di features.
La sostituzione è ovvia: se le feature sono uniformemente distribuite, allora ogni documento è restituibile allo stesso modo, dunque abbiamo probabilità $\frac{1}{D}$ che un certo oggetto goda della feature f_k e abbiamo probabilità $\frac{1}{n}$ che data una feature f_k, un certo oggetto ne goda.
Se andiamo ad esplicitare la sommatoria, otteniamo:

$$covering(o_i, o_j) = \sum_{k=1}^{n} \frac{1}{n}\frac{1}{D} = n\frac{1}{n}\frac{1}{D}$$

e quindi:

$$covering(o_i, o_j) = \frac{1}{D}$$

Abbiamo scoperto che nel caso di un database perfetto a unico cluster con feature uniformemente distribuite fra tutti gli oggetti, il covering fra due oggetti vale $\frac{1}{D}$. Ora, se sommiamo tutti i coverings di un oggetto con sé stesso (grandezza detta self-covering), otteniamo:

$$\sum_{o_i} covering(o_i, o_i) = \sum_{D} \frac{1}{D} = 1$$

Quindi, con *un cluster* la somma di tutti i *self-covering* è pari a 1. Si può ripetere il calcolo usando *due cluster* e scoprire che in tale situazione la somma dei self-covering da 2.

Se usiamo questa informazione al contrario, possiamo quindi dedurre che data la sommatoria dei *self-covering* di un database, otteniamo un valore p che è il numero di cluster (stimato) ottimale per quella base di dati.

5.6.11 Stima dell'errore di miss

Riprendiamo la Figura 5.67 e notiamo che, giacché solamente il cluster $C2$ viene restituito, tutte le zone di cluster comunque interne al raggio (ma il cui rappresentante non è incluso) vengono *perse* e non esiste post-processing che possa far recuperare tali dati.

Cerchiamo di quantificare la quantità di dati persi, e lo facciamo sul caso preciso di documenti testuali (quindi parliamo di ricerca di keywords).

Feature binarie indipendenti

Iniziamo a studiare il problema in un caso ridotto. Facciamo le seguenti assunzioni:

- Ogni documento è un vettore binario (non rappresentiamo quindi le frequenze ma solo se una keyword è presente o meno nel documento).
- I documenti sono organizzati in clusters.
- Ogni cluster ha un rappresentante.

Il nostro goal è trovare, per ogni cluster, il numero di documenti che hanno t o più keyword che matchano con le k fornite da una certa query. Stimando questa quantità saremo in grado di dire quanti documenti ''papabili'' ci sono in ogni cluster e di conseguenza valutare il miss.
Stando a quanto detto, ogni documento ha forma:

$$o_i = \langle f_{i,1}, f_{i,2}, ..., f_{i,n} \rangle$$

dove ogni $f_{i,j}$ è il valore (0 o 1) che indica la presenza della keyword j-esima nel documento i-esimo. Abbiamo quindi il solito vettore di features (binario in questo caso).

Secondo questo modello, un cluster O (O grande) è quindi un insieme di oggetti definiti come sopra. Sul cluster O possiamo definire la quantità R_O descritta come segue:

$$R_O = \langle r_1, r_2, ..., r_n \rangle = \frac{\sum_{o_i \in O} o_i}{\mid O \mid}$$

dove r_1 è la media fra tutti gli $f_{i,1}$, r_2 è la media fra tutti gli $f_{i,2}$ ecc. Quindi R_O è un vettore di dimensione pari al numero di features che all'elemento i contiene la media dei valori che assume la feature i-esima (media calcolata ovviamente considerando ogni oggetto contenuto nel cluster).

Abbiamo poi la nostra query Q nella forma:

$$Q = \langle 1, 1, 1, 1, ..., 1, 0, 0, ..., 0 \rangle$$

che ha un numero di 1 pari a k.

A questo punto la probabilità che un certo oggetto o^k (nella forma $o^k = \langle f_1, f_2, ..., f_n \rangle$) appartenga ad un cluster O è pari a:

$$p(o^k \in O) = \prod_{j=1}^{k} (r_j)^{f_j} (1 - r_j)^{1-f_j}$$

Cioè, scorriamo le prime k feature (che sono quelle a 1 nella query cioè quelle che ci interessano) usando l'indice j, ed andiamo a sfruttare la media degli oggetti nel cluster O per quella feature (r_j) per valorizzare la probabilità. Più precisamente se l'oggetto o^k ha valore 1 sulla feature j (cioè $f_j = 1$) allora conterà solamente la prima parentesi, quindi il valore assunto per quanto concerne la feature j sarà la media dei valori di quella feature in tutto il cluster. Se invece $f_j = 0$, allora sarà solo la seconda parentesi a dare contributo e quindi il valore assunto sarà 1 - la media dei valori di f_j su tutti gli oggetti del cluster. Tutti i valori assunti per ogni feature vengono moltiplicati insieme, fornendoci la probabilità totale che o^k appartenga al cluster O.

Data questa definizione, costruiamo tutti gli o^k possibili che abbiano almeno t features a 1 (fra quelle k espresse dalla query). È una operazione molto semplice, basta trovare tutte le combinazioni di t 1 nelle k features considerate dalla query. Poi tutte le configurazioni di $t + 1$ feature a 1 nelle k considerate dalla query, poi le configurazioni di $t + 2$... ecc. fino a giungere a k bit a 1. Una volta trovati tutti questi o^k, sommiamo la probabilità che ognuno di questi appartenga ad un dato O.

$$num(t, Q) = \sum_{o^k \text{con almeno } t1} (p(o^k \in O))$$

Siamo giunti all'obiettivo: abbiamo trovato il numero di oggetti in O che data una certa query Q la soddisfano (dove per soddisfare si intende che abbiano almeno t termini ad 1 sui k espressi dalla query).

Feature non-binarie indipendenti

Estendiamo ora il calcolo appena effettuato introducendo per ogni feature (keyword) invece che un valore binario (c'è o non c'è), una frequenza.
Ci poniamo quindi un nuovo goal: trovare la probabilità che un oggetto all'interno di un certo cluster sarà simile almeno S rispetto ad una data query.

Abbiamo sempre i nostri documenti

$$o_i = \langle f_{i,1}, f_{i,2}, ..., f_{i,n} \rangle$$

dove però $f_{i,j}$ questa volta assume un valore è che una frequenza e non più un valore binario. Muta quindi il contenuto di R_O:

$$R_O = \langle [r_1, w_1], [r_2, w_2], ..., [r_n, w_n] \rangle$$

dove r_j è la probabilità che una certa keyword j appartenga ad un documento del cluster e w_j è il peso con cui quella feature appare (la term frequency media in sostanza).
Abbiamo anche una nuova definizione dall query:

$$Q = \langle q_1, q_2, ..., q_k \rangle$$

dove ogni q_i rappresenta la frequenza di una certa keyword.

Definiamo ora la quantità $cont(i, Q)$:

$$cont(i, Q) = w_i q_i \text{ con probabilità } r_i$$

cioè si prende il peso medio di una certa keyword i e lo si moltiplica con la frequenza che quella keyword ha dentro la query. Quando si adopera il valore $cont(i, Q)$ bisogna comunque tenere da conto la probabilità r_i con cui la feature appartiene ad un documento del cluster.

Siamo ora in grado di definire $p(sim(O, Q) = S)$ cioè la probabilità la somiglianza fra un cluster ed una query sia esattamente S (sarà dunque immediatamente ottenibile la probabilità che la somiglianza sia maggiore di S, come volevamo fin dall'inizio).

$$p(sim(O, Q) = S) = coeff\left(x^S, \prod_{i=1}^{k}(r_i x^{w_i q_i} + (1 - r_i))\right)$$

Innanzi tutto si tenga presente che x^S non viene valutata, cioè non ci interessa il valore di x. Si tratta di una *funzione generatrice*, ovvero valutando la formula si otterrà un polinomio di grado S e noi siamo interessanti al coefficente di quel termine.
Osserviamo dunque come è generato questo polinomio. Si ha una produttoria che scorre le k keyword contenute nella query usando un indice i. Come si vede, si ha un doppio contributo. Il primo viene dalla prima parentesi dove prendiamo la probabilità che la keyword appartenga ad un documento del cluster (r_i) e la moltiplichiamo per il contributo di quella componente (come l'avevamo definito nella $cont(i, Q)$, messa ad esponente della x al solo fine della computazione del polinomio di cui parleremo dopo). Il secondo contributo è dato da (1-r_i), utile soltanto ai fini del calcolo (permette di ottenere tutte le combinazioni necessarie per creare i coefficenti necessari alla generazione del polinomio).

La computazione ci porterà così ad avere il famoso polinomio, generando fra gli altri degli elementi del tipo x^S i cui coefficenti saranno man mano sommati (ecco quindi cosa intendevamo con ''rappresentazione compatta'').

Il coefficente associato ad x^S è la probabilità che query ed il cluster si assomiglino esattamente S.

5.6.12 Se le feature non sono indipendenti (LSI)

Le misure metriche che abbiamo pesantemente sfruttato fino assumono che le features siano indipendenti fra loro. E se invece non lo fossero?

Facciamo uso di una tecnica chiamata *Latent Semantic Indexing* (in breve LSI) che cerca di scovare concetti latenti all'interno della base di dati (è una tecnica infatti molto adottata nell'ambito delle collezioni di testi che sfruttano la cosine similarity).

Diciamo di avere un database con $\mid O \mid$ oggetti, dove ogni oggetto o è rappresentato da un vettore di dimensione $\mid v \mid$ (che è quindi la quantità di features/keywords totale). Possiamo facilmente definire la matrice OF:

$$OF = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & \cdots \\ f_{2,1} & f_{2,2} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ 1 & 2 & \cdots & \mid v \mid \end{pmatrix} \begin{matrix} 1 \\ 2 \\ \vdots \\ \mid O \mid \end{matrix}$$

Come vediamo ogni riga costituisce il vettore di feature di un oggetto (quindi abbiamo una riga per ogni oggetto) e ogni colonna indica il valore di ogni oggetto su una determinata feature (e quindi abbiamo tante colonne quanto sono le features).

Si tratta dunque di una matrice che rappresenta l'intero database... possiamo farci qualcosa? Potremmo trovare delle similarità fra oggetti, o fra features? Vi sono concetti indipendenti all'interno del database? E soprattutto, possiamo sfruttare questa matrice per un indexing efficente?

Similarità fra gli oggetti: la matrice OO

Possiamo sintetizzare la similarità fra gli oggetti moltiplicando la matrice OF con la sua trasposta:

$$OF \times (OF)^T = \begin{array}{c} \left(\begin{array}{ccc} f_{1,1} & f_{1,2} & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid O \mid \end{array} \\ 1 \quad \cdots \quad \mid v \mid \end{array} \times \begin{array}{c} \left(\begin{array}{ccc} f_{1,1} & \cdots & \cdots \\ f_{1,2} & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid v \mid \end{array} \\ 1 \quad \cdots \quad \mid O \mid \end{array} =$$

$$OO = \begin{array}{c} \left(\begin{array}{ccc} \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid O \mid \end{array} \\ 1 \quad \cdots \quad \mid O \mid \end{array}$$

Come si vede, quello che si ottiene è una matrice di dimensione $\mid O \mid \times \mid O \mid$ dove l'elemento $oo_{i,j}$ è dato dalla moltiplicazione fra il vettore delle feature dell'oggetto i con il vettore delle feature dell'oggetto j. In sostanza si va a fare un prodotto scalare, cioè la cosine similarity fra due oggetti. Ecco quindi che ogni elemento della matrice rappresenta il prodotto cartesiano fra due oggetti del database.

Correlazione fra le features: la matrice FF

È possibile determinare dei coefficienti di correlazione fra feature usando il principio di prima ma ruotando la matrice di partenza. Otteniamo così la matrice FF di correlazione fra le features:

$$FO \times (FO)^T = \begin{array}{c} \left(\begin{array}{ccc} f_{1,1} & f_{2,1} & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid v \mid \end{array} \\ 1 \quad \cdots \quad \mid O \mid \end{array} \times \begin{array}{c} \left(\begin{array}{ccc} f_{1,1} & \cdots & \cdots \\ f_{2,1} & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid O \mid \end{array} \\ 1 \quad \cdots \quad \mid v \mid \end{array} =$$

$$FF = \begin{array}{c} \left(\begin{array}{ccc} \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots \end{array} \right) \begin{array}{c} 1 \\ \vdots \\ \mid v \mid \end{array} \\ 1 \quad \cdots \quad \mid v \mid \end{array}$$

Si ottiene questa volta una matrice $\mid v \mid \times \mid v \mid$ che in modo del tutto simmetrico a prima rappresenta la correlazione fra features (di fatto abbiamo fatto la trasposta di entrambe le matrici generanti la OO).

Singular valued decomposition (SVD)

Ulteriore applicazione della matrice OF (relativa finalmente all'indexing) è una tecnica simile all'eigendecomposition (Sezione 2.3.2) e prende il nome di singular valued decomposition:

$$A = OC \cdot CC \cdot CV =$$

$$\begin{pmatrix} oc_{1,1} & \cdots & oc_{1,|c|} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ oc_{|O|,1} & \cdots & oc_{|O|,|c|} \end{pmatrix} \cdot \begin{pmatrix} cc_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & cc_{|c|,|c|} \end{pmatrix} \cdot \begin{pmatrix} cf_{1,1} & \cdots & \cdots & cf_{1,|v|} \\ \vdots & \vdots & \vdots & \vdots \\ cf_{|c|,1} & \cdots & \cdots & cf_{|c|,|v|} \end{pmatrix}$$

La matrice centrale (CC) è detta *matrice dei concetti* e rappresenta il nuovo spazio in cui si lavorerà. È una matrice diagonale con gli elementi sulla matrice ordinati per significatività. La matrice a sinistra, OC, rappresenta la correlazione fra gli oggetti e i concetti (sintetizza quindi il mapping degli oggetti nel nuovo spazio). Come già nell'eigendecomposition, tale matrice ha colonne ortonormali (a somma uno e linearmente indipendenti fra loro). La matrice a destra, OV, *non è* la trasposta di OC come accadeva per l'eigendecomposition ma è una matrice particolare che mappa i concetti sulle vecchie feature prese in considerazione. Le righe sono ortonomali.

Grazie a questa decomposizione, abbiamo ottenuto un nuovo spazio (CC) grazie al quale sfruttare OC e OV per mappare gli oggetti e le feature in uno spazio ortogonale (sfruttando quindi le strutture ad indice che abbiamo già studiato).
Il numero di concetti che troveremo (ergo la cardinalità $\mid C \mid$, cioè il numero di righe/colonne di CC) sarà pari al rango di OF (la matrice object-feature da cui avevamo ottenuto la scomposizione). Si ricordi che il rango è il numero di righe linearmente indipendenti di una matrice, perciò avremo $\mid C \mid = \min\{\mid O \mid, \mid v \mid\}$. A seconda di quanto le feature fossero ridondanti avremo più o meno concetti.

Tralasciare i concetti meno importanti. Così come accadeva per l'eigendecompositon, nel caso in cui si voglia scendere di dimensionalità sarà sufficente ignorare i concetti più in basso nella matrice CC poiché meno significativi.

Risoluzione delle query. Quando si vanno a risolvere le query si sfrutterà quindi il nuovo spazio per mappare la query al suo interno. Supponendo di considerare solamente un array $| r |$ comprendente parte degli $| c |$ concetti totali (abbiamo quindi ridotto lo spazio), il costo di esecuzione della query sarà costituito dalla somma dei costi per le operazioni:

1. Mapping della query nello spazio dei concetti: $Q \cdot CV$, cioè $| v | * | v | * | r |$. Tale operazione produce un vettore colonna con $| r |$ elementi che rappresenta la query riscritta (chiamiamola Q_{riscr})

2. Mapping della query riscritta nello spazio dei concetti: $Q_{riscr} \cdot CC$, cioè $| r | * | r | * | r |$. Si ottiene così un vettore riga di $| r |$ componenti.

3. Non resta che prendere il vettore appena generato e moltiplicarlo con OC (costo $| O | * | r | * | r |$) al fine di ottenere un vettore colonna che per ogni elemento rappresenta la similarità di ogni oggetto rispetto alla query Q nello spazio CC. Sarà dunque sufficente ordinare questi elementi per ottenere il ranking degli oggetti simili alla query.

Il costo totale risulta quindi:

$$(| v | * | v | * | r |) + (| r | * | r | * | r |) * (| O | * | r | * | r |)$$

5.6.13 Valutare i metodi di clustering

Concludiamo il discorso in merito al clustering introducendo alcune tecniche per valutare i sistemi di clustering e che quindi possono permetterci di trarre delle conclusioni in merito al funzionamento più o meno buono di un certo sistema.

Precision e recall

Data una certa query ed un certo insieme di risultati restituiti, definiamo i concetti di *precision* e *recall*. Tali insiemi mirano a quantificare la differenza fra ciò che è stato restituito e ciò che l'utente avrebbe voluto vedere restituito, in termini di false hit e di misses.
Come vediamo nella Figura 5.73, dato che la query si trova nell'intorno del cluster rappresentato dai quadrati, saranno restituititi tutti gli oggetti che fanno parte di quel cluster. Nella zona gialla abbiamo indicato però ciò che l'utente si aspettava, quindi il ''miglior risultato'' possibile: abbiamo quindi un certo numero di risultati che sono false hit (quelli appartenenti al cluster restituito ma non desiderati dall'utente, indicati in rosso) e altri che sono miss (quelli non appartenenti al cluster restituito ma desiderati dall'utente, in viola).

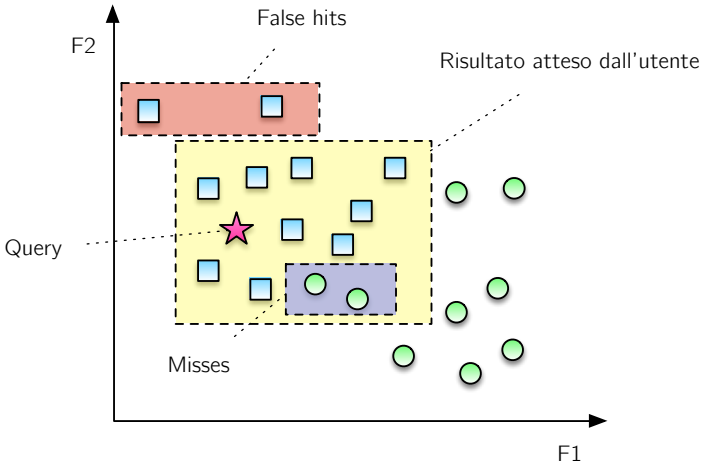


Figura 5.73: Query su un database con due cluster (tondi e quadrati).

Precision. La precision permette di calcolare l'effetto dei *false hit*, ed è determinata calcolando il sottoinsieme di risultati *rilevanti* e *restituiti* diviso l'insieme dei *restituiti*. In formule:

$$\text{Precision} = \frac{\text{Retrieved \& Relevant}}{\text{Retrieved}}$$

La definizione è intuitiva: saremo tanto più precisi tanto *meno* saranno i false hit, cioè tanto più piccola sarà la porzione di risultati restituiti non rilevanti.

Recall. La recall permette di calcolare l'effetto dei *miss*, ed è determinata calcolando il sottoinsieme di risultati *rilevanti* e *restituiti* diviso l'insieme dei *rilevanti*. In formule:

$$\text{Recall} = \frac{\text{Retrieved \& Relevant}}{\text{Relevant}}$$

Anche qui abbiamo una definizione intuitiva: la recall sarà tanto più alta quanto sarà alta la porzione di risultati rilevanti fra i restituiti.

È nostra intenzione avere entrambe le misure il più vicino possibile ad 1 (ma, come è chiaro, le due sono in conflitto fra loro: aumentare la recall implica andare a prendere anche dei risultati non rilevanti mentre aumentare la precision implica perdere dei risultati potenzialmente interessanti).

La curva precision/recall

Osserviamo la Figura 5.74: abbiamo considerato un database con due cluster, abbiamo definito una query e basandoci su questa abbiamo anche definito anche il cluster restituito e la porzione di risultati attesi dall'utente. Poi, è stato

associato ad ogni oggetto restituito un numero (così da poterli identificare più semplicemente durante le discussioni a venire).

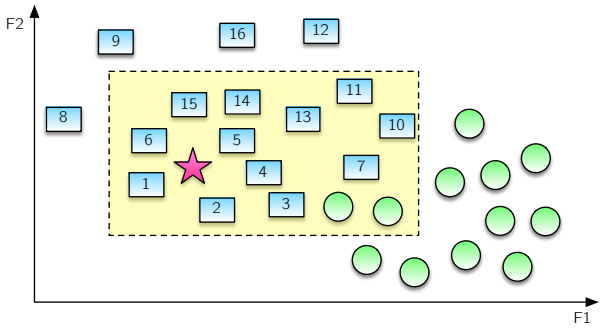


Figura 5.74: Query su un database con due cluster (tondi e quadrati).

Sono stati restituiti 12 dei 14 oggetti attesi, dunque la recall è pari a $12/14 = 0.86$.

Ciò che è possibile fare è andare a studiare la curva precision/recall, cioè l'andamento fra i due parametri a fronte dell'aggiunta (man mano) di un oggetto nell'insieme dei restituiti:

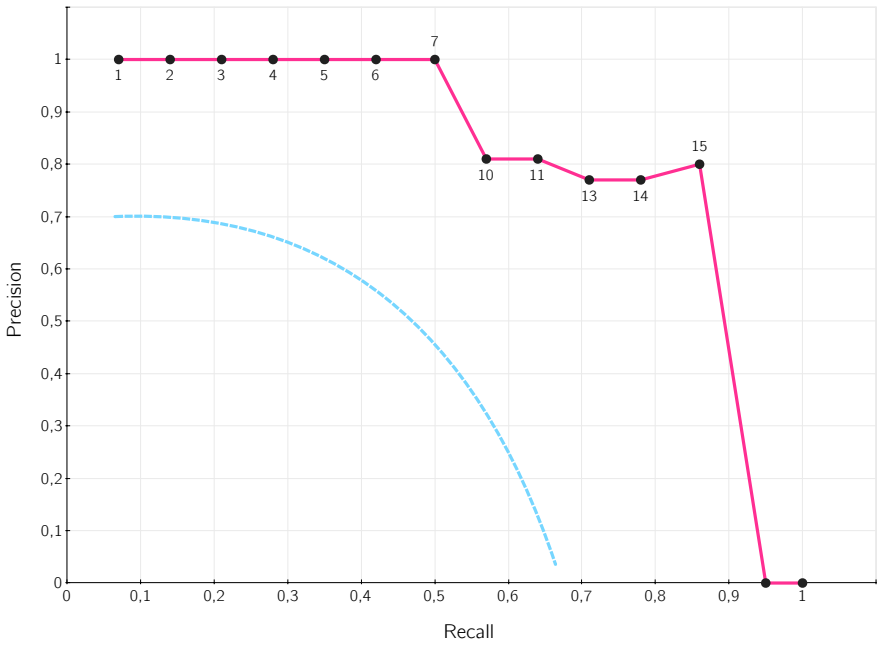


Figura 5.75: Andamento della precision e della recall.

Ogni punto rappresenta l'aggiunta di un oggetto nell'insieme di quelli restituiti

(precisamente l'oggetto indicato come indice vicino al punto). In blu è stato rappresentato il tipico andamento di una curva precision/recall: il risultato è quindi in questo caso molto buono rispetto al solito.

Abbiamo così ottenuto una sorta di serie temporale che ci fornisce l'andamento della precision e della recall.
Si noti comunque che per fare una analisi significativa è necessario studiare l'andamento dei due parametri *su diverse query*, dopodiché trarne una curva media e quindi un comportamento medio.

Indici unici

La precision è la recall sono ovviamente legati fra loro, ma non ''direttamente'' dal punto di vista matematico. Esistono per questo motivo altri indici unici (nel senso che sono esprimibili tramite un numero soltanto) che possono darci indicazione della qualità di un sistema di clustering. Il vantaggio di avere indici di valutazione unici sta nella possibilità di confrontare più facilmente fra loro diversi sistemi di clustering.

R-precision. La R-precision rappresenta il numero di documenti rilevanti restituiti entro i primi R, dove R è il numero totale di documenti rilevanti per l'utente. Nell'esempio di prima si avevano un totale di documenti rilevanti (per l'utente) di 14, dunque consideriamo i primi 14 risultati:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14$$

Fra questi, solamente 11 sono rilevanti (8, 9 e 12 non lo sono). La R-precision è quindi 11/14 = 0.876.

Media armonica. La media armonica di n numeri è:

$$\frac{1}{H} = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{x_i}$$

Possiamo dunque fare la media armonica fra precision (P) e recall (R) ottenendo così:

$$H(P, R) = \frac{2PR}{P + R}$$

La media armonica è ottima perché, per l'appunto, armonizza i valori. A fronte di entrambi i parametri alti, la media armonica è alta, mentre appena uno dei due parametri è basso, la media armonica è bassa.

Coverage e novelty

Quando si ha un processo iterativo può essere interessante capire quanto le iterazioni arricchiscano il risultato mostrato all'utente. Consideriamo quindi lo schema in Figura 5.76.

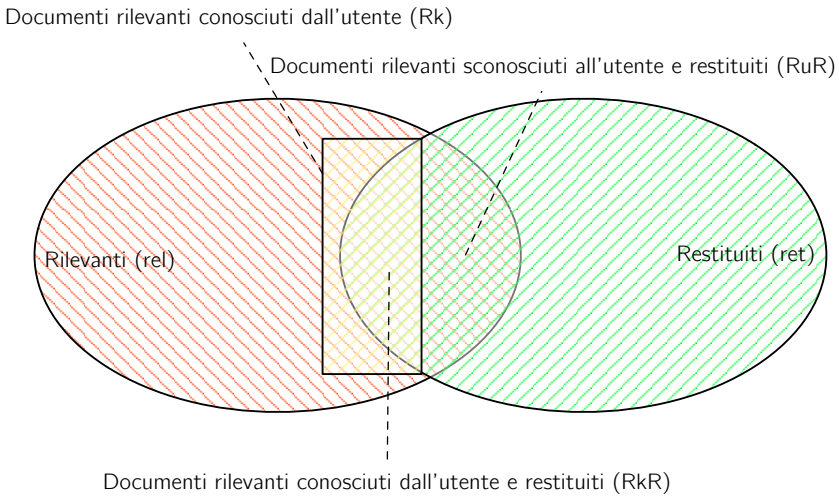


Figura 5.76: Vari insiemi di risultati in un sistema con processo iterativo.

Supponiamo che il sistema si trovi alla seconda iterazione (ma il discorso è lo stesso per una qualsiasi iterazione i). La zona gialla identifica tutti i risultati dell'iterazione precedente (la prima), dunque quelli in generale conosciuti dall'utente (Rk). Di questi, una parte è restituita anche dalla seconda iterazione (RkR). Abbiamo poi un insieme di documenti restituiti dalla seconda iterazione che sono rilevanti ma che l'utente non aveva mai visto (RuR).
A partire da questi insieme definiamo i concetti di *coverage* e *novelty*.

Novelty. La novelty rappresenta quanti nuovi oggetti rilevanti sono mostrati all'utente rispetto a quelli già conosciuti dall'utente. Si ha quindi:

$$\text{Novelty} = \frac{\text{RuR}}{\text{RuR} + \text{RkR}}$$

Coverage. La coverage rappresenta invece quanto il sistema sia in grado di identificare quegli oggetti che sono già indicati dall'utente come rilevanti. Si ha quindi:

$$\text{Coverage} = \frac{\text{RkR}}{\text{Rk}}$$

Capitolo 6

Relevance feedback

6.1 Introduzione

6.1.1 Perché relevance feedback

Parliamo finalmente di relevance feedback. Abbiamo accennato più e più volte a questo sistema: data una query Q si ottiene una serie di oggetti S. L'utente è in grado di selezionare (manualmente) all'interno degli S oggetti quelli che sono per lui rilevanti (e li chiamiamo Rel) e quelli che invece sono irrilevanti (e li chiamiamo I). Tendenzialmente l'utente sceglierà gli oggetti rilevanti tutti gli altri saranno dati per irrilevanti (quindi $Ir = S - Rel$) ma non è escluso che l'utente possa anche delineare un terzo insieme di oggetti sui quali non si vuole esprimere. Per semplicità, considereremo solamente gli insiemi Rel ed Ir.

Dati questi due insiemi vogliamo quindi migliorare la qualità del risultato della query, cercando di interpretare l'informazione latente che l'utente ci sta fornendo delineando l'insieme degli oggetti rilevanti e quello degli oggetti irrilevanti. La necessità di un sistema di questo tipo è intrinseca nella definizione dei database multimediali: si tratta di database *soggettivi*, poiché la rilevanza e la distanza sono concetti che dipendono dall'utente e non assoluti (ma ovviamente il database, essendo implementato all'interno di una macchina, ragiona in senso oggettivo).

6.1.2 Un retrieval efficace

Ciò che vogliamo ottenere (ovvero un retrieval efficace) è rappresentato da questa implicazione:

$$p(Rel|o_i) > p(Rel|o_j) \Leftrightarrow sim(Q, o_i) > sim(Q, o_j) \tag{6.1}$$

Si sta cioè dicendo che la probabilità che un certo oggetto o_i appartenga all'insieme degli oggetti rilevanti è maggiore della probabilità che un altro oggetto

o_j vi appartenga, se e solo se la similarità fra la query Q e l'oggetto o_i è maggiore di quella fra query Q e l'oggetto o_j.
È evidente che possiamo lavorare solamente sul lato destro della bi-implicazione: a sinistra abbiamo la verità dell'utente alla quale dobbiamo adattarci. Cosa possiamo fare dal punto di vista pratico?

6.1.3 Excursus sulle tecniche di relevance feedback

Vediamo ora una lista di tecniche adottabili per implementare il relevance feedback. Parleremo in maniera approfondita di alcune di queste in seguito.

1. *Spostare la query*: se notiamo una separazione sufficentemente netta fra gli oggetti rilevanti e quelli irrilevanti possiamo modificare la query spostandola verso la zona ove gli oggetti rilevanti sono più concentrati.
2. *Modificare il raggio della query*.
3. *Modificare i pesi delle features* all'interno della query.
4. *Rimuovere o aggiungere features*.
5. *Cambiare la metrica*.
6. *Modificare i cluster* (o considerare un altro rappresentante). Quest'ultima tecnica è evidentemente molto costosa e verrà usata in seguito a molte query che richiedono lo stesso tipo di aggiustamento (evidenziando quindi un problema strutturale relativo ai cluster).

6.2 Spostare la query

Studiamo nel dettaglio la prima tecnica delle tecniche cui abbiamo accennato poco fa. Come abbiamo detto, questa tecnica è adottabile se c'è sufficente *separazione* fra l'insieme degli oggetti rilevanti e quello degli oggetti irrilevanti.

6.2.1 Massimizzare la separazione

Introduciamo una semplice misura:

$$sep = \left(\sum_{o_i \in Ir} dist(Q, o_i) \right) - \left(\sum_{o_i \in Rel} dist(Q, o_i) \right)$$

Tale quantità definisce la *separazione* fra l'insieme degli oggetti irrilevanti ($o_i \in Ir$) e quelli rilevanti ($o_i \in Rel$). La misurazione è effettuata sommando le distanze fra la query ed ogni oggetto dell'insieme degli irrilevanti e sottraendovi tutte le distanze fra la query e gli oggetti all'interno dell'insieme dei rilevanti.

La quantità *sep* così calcolata è evidentemente da *massimizzare*.

Supponendo di usare una misura di distanza lineare, possiamo effettuare questo passaggio algebrico:

$$sep = dist\left(Q, \sum_{o_i \in Ir} o_i - \sum_{o_i \in Rel} o_i\right) \tag{6.2}$$

Non si dimentichi che gli oggetti sono vettori, dunque si può effettuare la somma fra tutti gli oggetti irrilevanti e la sommatoria degli oggetti rilevanti, sottrarre i due vettori così ottenuti ottenendo un ulteriore vettore. Calcolando poi la distanza fra quel vettore e la query si ha la separazione.

6.2.2 La nuova query

La nuova query Q' (debitamente spostata) sarà quindi:

$$Q' = Q + \left(c_{rel} \times \sum_{o_i \in Rel} o_i\right) + \left(c_{ir} \times \sum_{o_i \in Ir} o_i\right)$$

Non abbiamo fatto altro che introdurre due moltiplicatori che possono essere regolati per amplificare l'effetto dello spostamento. Per semplicità consideriamo $c_{ir} = c_{rel}$.

Un esempio

Vediamo un piccolo esempio d'applicazione della tecnica di cui abbiamo appena parlato (seguendo l'Equazione 6.2). Consideriamo la query:

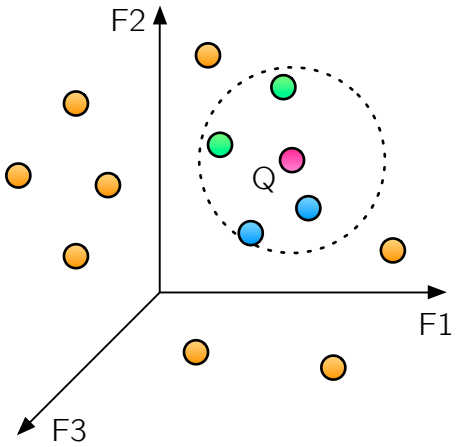


Figura 6.1: Una query con due oggetti rilevanti e due oggetti non rilevanti.

I due oggetti verdi (in alto) sono stati identificati dall'utente come rilevanti, mentre gli altri due blu (in basso) sono irrilevanti.
Ora, andiamo ad identificare le varie distanze dalla query.

Gli oggetti rilevanti sono identificati da una freccia che dalla query va verso di essi (quindi un valore positivo), mentre le distanze fra gli oggetti irrilevanti e la query sono rappresentati con una freccia dagli oggetti verso la query (quindi sarebbe la distanza cambiata di segno). Abbiamo quindi invertito i segni rispetto alla Equazione 6.2:

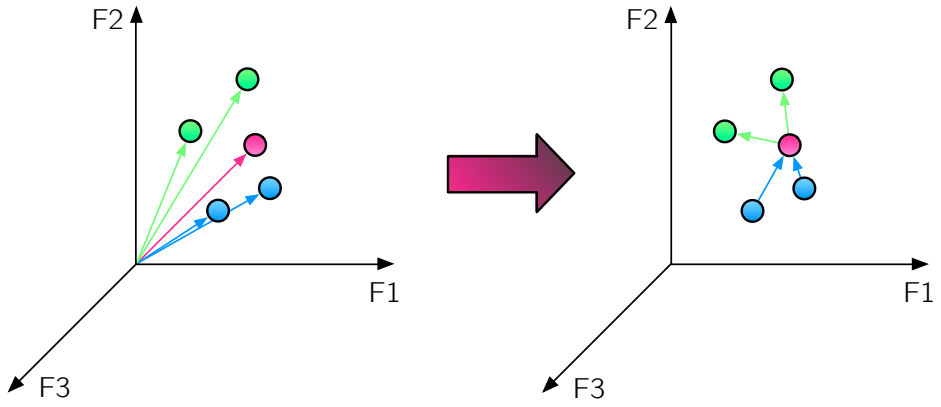


Figura 6.2: Vengono calcolate le distanze.

Una volta ottenute le distanze, le si somma al fine di ottenere il vettore la cui distanza dalla query ci darà la separazione:

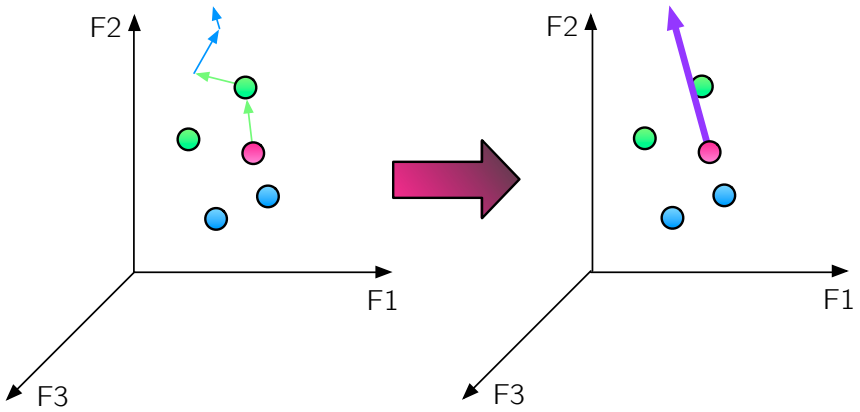


Figura 6.3: Le distanze sono sommate.

Non resta che scegliere (usando i coefficenti c_{rel} e c_{ir}) di quanto spostare la query (Figura 6.4).

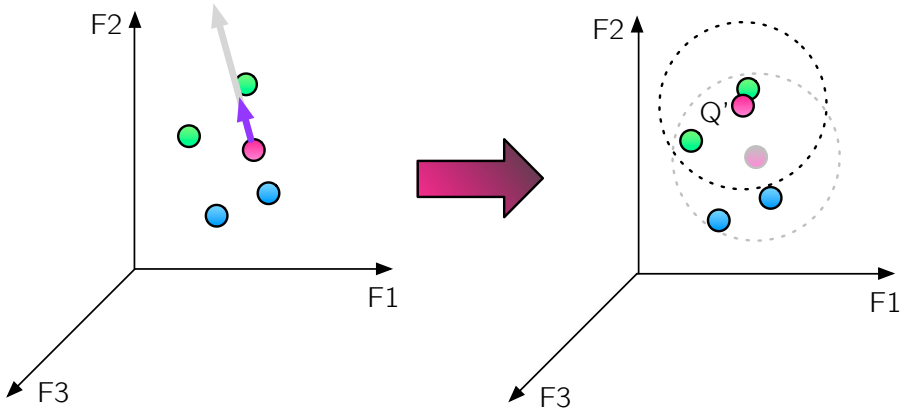


Figura 6.4: La query viene spostata.

6.3 Modificare i pesi delle features

La seconda ed ultima tecnica che andiamo ad introdurre è molto più sofisticata della precedente: si vanno a modificare i pesi delle feature a seconda di quanto espresso dall'utente. Si noti che l'eliminazione/aggiunta di una feature è semplicemente il caso particolare di un assegnamento di peso pari a 0/diverso da 0 di una certa feature.

6.3.1 Considerare la significatività di ogni feature

Quello che si vuole fare è considerare la significatività di *ogni* feature del database (non solo quelle contenute nella query). Il concetto di significatività è rappresentato dalla capacità di discernere fra gli oggetti rilevanti e quelli irrilevanti, quindi data una certa feature f_k abbiamo questa formula:

$$sig(f_k) = \frac{\log\left(\frac{p(f_k|R)}{1-p(f_k|R)}\right)}{\log\left(\frac{p(f_k|I)}{1-p(f_k|I)}\right)}$$

Attenzione: si noti che R ed I non sono gli insiemi $RelI$ ed Ir di cui abbiamo parlato fino ad ora, ma si tratta degli oggetti rilevanti e irrilevanti *in assoluto*, quindi anche quelli eventualmente non restituiti alla prima iterazione come risultato della query. Gli oggetti $RelI$ sono infatti intrinsecamente restituiti e rilevanti (è stato l'utente a sceglierli, ma fra gli S restituiti) e lo stesso vale per gli insiemi Ir ed I.

Con $p(f_k|R)$ si intende la probabilità che un certo oggetto o sia rilevante (in senso assoluto) e soddisfi la feature f_k. Dunque, al numeratore della frazione abbiamo il logaritmo della probabilità che una feature f_k sia soddisfatta da un

oggetto rilevante diviso la probabilità che non la soddisfi (pur essendo rilevante). Al denominatore abbiamo la stessa formula dove però viene sfruttata la probabilità che un oggetto sia irrilevante (anziché rilevante) e soddisfi la feature f_k. I logaritmi sono usati semplicemente per rendere più leggibile il risultato.

Quando il numeratore è alto ed il denominatore è basso, abbiamo una feature molto significativa (l'alto numeratore indica che oggetti rilevanti soddisfano f_k, il basso numeratore indica che gli oggetti irrilevanti non soddisfano f_k).

Il calcolo della formula si riduce quindi a trovare $p(f_k|R)$ ed $p(f_k|I)$. Ma come abbiamo già detto, R ed I sono insiemi assoluti di oggetti rilevanti ed irrilevanti, quindi il calcolo della $p(f_k|R)$ e della $p(f_k|I)$ deve essere differenziato a seconda che la feature f_k sia espressa nella query o meno.

Calcolare $p(f_k|R)$ con $f_k \notin Q$

Arriviamo alla soluzione partendo da questo esempio: se dovessimo studiare la probabilità che un oggetto che soddisfa la feature f_k sia rilevante per l'utente, potremmo semplicemente prendere 100 elementi *a caso* e studiare quanti fra questi sono rilevanti e soddisfano la feature f_k (semplicemente contandoli): se 60 elementi su 100 soddisfano la feature f_k e sono anche rilevanti per l'utente, allora abbiamo ottenuto la nostra probabilità.

Questo ragionamento ha senso fintanto che gli elementi sono presi *a caso* dal database: ma se la feature f_k non è stata espressa nella query, allora l'insieme Rel (dei risultati restituiti e rilevanti per la query) è sostanzialmente casuale rispetto ad f_k, dunque il nostro ragionamento regge e possiamo calcolare $p(f_k|R)$ così come abbiamo appena descritto: contiamo quanti elementi all'interno dell'insieme Rel (che sono per definizioni rilevanti) soddisfano f_k. In pratica abbiamo che:

$$p(f_k|R) = p(f_k|Retrieved\ \&\ Relevant)$$

e quindi calcoliamo direttamente su Rel (che è l'insieme degli elementi rilevanti e restituiti) la quantità di elementi che soddisfano f_k e la dividiamo per la cardinalità dell'insieme Rel (effettuiamo dunque un semplice calcolo casi favorevoli su casi totali).

Calcolare $p(f_k|R)$ con $f_k \in Q$

Quando f_k è espressa all'interno della query Q, non possiamo più dire che l'insieme Rel sia casuale rispetto ad f_k (è infatti stato costruito *basandosi* anche su f_k, dunque la maggiorparte degli elementi di Rel avranno f_k soddisfatta: è come se avessimo un bias nei dati). Abbiamo bisogno di trovare un modo per ricondurci al caso di prima.

Partiamo facendo alcune assunzioni:

- Abbiamo feature binarie, quindi abbiamo:
  - Oggetti del tipo $o =< f_1, f_2, ..., f_n >$ con $f_i = 0$ oppure 1.
  - Una query del tipo $q =< w_1, w_2, ..., w_n >$ con $w_i = 0$ oppure 1.
- Abbiamo la cosine similarity (prodotto scalare):
  - $sim(o, q) = \sum_{i=1}^{n} w_i f_i$

Le assunzioni che facciamo restringono l'applicabilità della tecnica che stiamo per introdurre ai casi dove le assunzioni risultano vere. Qualora questo non accada, esistono tecniche diverse che non studiamo (portiamo quindi un esempio soltanto a fini didattici).

Stando a quando detto, se un certo oggetto o è stato restituito abbiamo che:

$$sim(o, q) = \sum_{i=1}^{n} w_i f_i > T$$

cioè la similarità fra query e l'oggetto sarà stata maggiore di una certa soglia T. Tutti i documenti con similarità maggiore di T sono stati restitutiti. Concentriamoci ora su una feature f_j in particolare (estraendola dalla sommatoria):

$$sim(o, q) = \left(w_j f_j + \sum_{i \in \{1,...,n\} - \{j\}}^{n} w_i f_i \right) > T$$

La f_j scelta sarebbe la feature a cui siamo interessati (l'avevamo chiamata f_k prima). Introduciamo la notazione:

$$sim_{(-j)}(o, q)$$

per indicare la similarità fra l'oggetto o e la query q senza tener conto della feature f_j. Riscriviamo quindi la nostra formula sfruttando la nuova notazione:

$$sim(o, q) = (w_j f_j + sim_{(-j)}(o, q)) > T$$

Ricordiamo che tutti gli o per cui la disequazione appena scritta vale, sono quelli restituiti. Andiamo ad identificare due diverse componenti di questo insieme:

- L'insieme degli oggetti per cui $sim_{(-j)}(o, q) \leq T$: per questi oggetti la feature f_j è importante poiché senza di essa, l'oggetto non sarebbe stato restituito (ha similarità minore della soglia T).

- L'insieme degli oggetti per cui $sim_{(-j)}(o, q) > T$: per questi oggetti f_j non è così importante, poiché anche senza di essa l'oggetto ha similarità sopra la soglia (quindi sarebbe restituito anche senza f_j).

Diciamo che la prima componente abbia cardinalità a: vi saranno a oggetti all'interno dell'insieme dei restituti che avranno $f_j = 1$ (altrimenti non sarebbero stati restituiti). Si tratta quindi del bias di cui parlavamo prima, l'insieme di oggetti che si trovano nell'insieme dei restituiti *non indipendentemente* dal fatto di avere $f_j = 1$.
Della seconda componente possiamo poi distinguere una quantità b di oggetti che hanno $f_j = 1$ e una quantità c che invece hanno $f_j = 0$ (sono comunque tutti restituiti).

Abbiamo ottenuto quanto desiderato, possiamo calcolare:

$$p(f_k|R) = \frac{p\Big((f_k = 1) \wedge (sim_{(-k)}(o, q) > T)\Big)}{p(sim_{(-k)}(o, q) > T)}$$

e quindi al momento del calcolo useremo:

$$p(f_k|R) = \frac{b}{b + c}$$

Abbiamo semplicemente sostituito le probabilità con le cardinalità degli insiemi che abbiamo prima definito.

6.4 Garantire il ranking

Riprendiamo in mano l'Equazione 6.1:

$$p(Rel|o_i) > p(Rel|o_j) \Leftrightarrow sim(Q, o_i) > sim(Q, o_j)$$

Tramite le tecniche di relevance feedback che abbiamo introdotto fin ora abbiamo garantito di restituire man mano gli oggetti più rilevanti per l'utente ma non abbiamo garantito l'*ordinamento* fra gli elementi, in modo che si rispetti anche l'ordinamento fra gli elementi restituiti.

6.4.1 Dal teorema di Bayes alla rilevanza fra oggetti

Riprendiamo in mano un teorema classico delle probabilità, il teorma di Bayes:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

Tale teorema ci può essere utile, poiché noi siamo in grado di calcolare $p(o_i|Rel)$ e non $p(Rel|o_i)$ (che però appare nell'Equazione 6.1). Trasformiamolo ulteriormente, riscrivendo $p(B)$ ed ottenendo:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B|A)p(A) + p(B|\neg A)p(\neg A)}$$

Ecco quindi che siamo in grado di scrivere:

$$p(Rel|o_i) = \frac{p(o_i|Rel)p(Rel)}{p(o_i|Rel)p(Rel) + p(o_i|Ir)p(Ir)}$$

$$>$$

$$p(Rel|o_j) = \frac{p(o_j|Rel)p(Rel)}{p(o_j|Rel)p(Rel) + p(o_j|Ir)p(Ir)}$$

Abbiamo sfruttato il fatto che essere $\neg Rel$ significhi appartenere all'insieme Ir.
Da qui, grazie al alcuni passaggi algebrici, otteniamo:

$$\frac{p(o_i|Rel)}{p(o_i|Ir)} > \frac{p(o_j|Rel)}{p(o_j|Ir)}$$

e quindi la nostra equazione di partenza diventa:

$$\frac{p(o_i|Rel)}{p(o_i|Ir)} > \frac{p(o_j|Rel)}{p(o_j|Ir)} \Leftrightarrow sim(Q, o_i) > sim(Q, o_j) \tag{6.3}$$

Come facciamo a computare quelle probabilità?

6.4.2 Se le feature sono indipendenti

Come già accaduto in precedenza, assumiamo di avere feature indipendenti. Qualora questo accada anche nella realtà si adotterà la tecnica che stiamo per esporre. Tratteremo in seguito il caso in cui le feature non sono indipendenti fra loro.

Grazie al fatto che le feature sono indipendenti, possiamo scrivere:

$$p(o_i|Rel) = \prod_{k=1}^{n} p(f_{i,k}|Rel)$$

ovvero, la probabilità che un oggetto appartenga all'insieme di quelli rilevanti è pari alla moltiplicazione fra le probabilità che ogni feature di quell'oggetto sia soddisfatta e appartenga all'insieme dei rilevanti. Possiamo quindi scrivere:

$$\frac{p(o_i|Rel)}{p(o_i|Ir)} > \frac{p(o_j|Rel)}{p(o_j|Ir)} \Leftrightarrow \frac{\prod_{k=1}^{n} p(f_{i,k}|Rel)}{\prod_{k=1}^{n} p(f_{i,k}|Ir)} > \frac{\prod_{k=1}^{n} p(f_{j,k}|Rel)}{\prod_{k=1}^{n} p(f_{j,k}|Ir)}$$

Possiamo poi applicare il logaritmo ad entrambi i membri ed ottenere:

$$\log\left(\frac{\prod_{k=1}^{n} p(f_{i,k}|Rel)}{\prod_{k=1}^{n} p(f_{i,k}|Ir)}\right) > \log\left(\frac{\prod_{k=1}^{n} p(f_{j,k}|Rel)}{\prod_{k=1}^{n} p(f_{j,k}|Ir)}\right)$$

Possiamo poi raccogliere le produttorie (di numeratori e denominatori) e giacché il logaritmo del prodotto è la somma dei logaritmi, possiamo ottenere:

$$\sum_{k=1}^{n} \log\left(\frac{p(f_{i,k}|Rel)}{p(f_{i,k}|Ir)}\right) > \sum_{k=1}^{n} \log\left(\frac{p(f_{j,k}|Rel)}{p(f_{j,k}|Ir)}\right)$$

Sfrutteremo quindi questa disequazione (dove $\log\left(\frac{p(f_{i,k}|Rel)}{p(f_{i,k}|Ir)}\right)$ sono i pesi attribuiti alla k-esima feature) per ordinare i risultati ed ottenere così un ranking efficente.

6.4.3 Se le feature non sono indipendenti

Qualora le feature del database non siano indipendenti fra loro, non abbiamo più l'uguaglianza dalla quale siamo partiti prima, ovvero:

$$p(o_i|Rel) \neq \prod_{k=1}^{n} p(f_{i,k}|Rel)$$

Introduciamo per questo I, che misura l'indipendenza fra probabilità. Tale funzione è così definita:

$$I(p1, p2) = \sum_{x} p1(x) \log \frac{p1(x)}{p2(x)}$$

Ovvero, data una serie di eventi x due probabilità $p1$ e $p2$ sono tanto più indipendenti quanto $I(p1, p2)$ è alto. Infatti, abbiamo che:

- Con $p1 = p2$, $I = 0$.
- Con $p1 \neq p2$, $I > 0$.

Applicando questa formula su due feature (cercandone quindi l'indipendenza) otteniamo:

$$D_{ij} = I(p(f_i \wedge f_j), p(f_i)p(f_j))$$

dove D_{ij} è detto *grado di indipendenza* fra le feature f_i ed f_j. Infatti, misuriamo l'indipendenza fra la probabilità che siano soddisfatti contemporaneamente f_i e f_j e la probabilità che siano soddisfatte entrambe in maniera indipendente (quindi con la moltiplicazione fra probabilità).

Per quanto riguarda la computazione, le probabilità $p(f_i)$ (e analogamente $p(f_j)$) si calcolano contando il numero di oggetti che soddisfano la feature f_i all'interno dell'insieme Rel, mentre $p(f_i \wedge f_j)$ si calcola contando il numero di compresenze delle due feature soddisfatte (sempre nell'insieme Rel).

Il grafo delle dipendenze. Una volta calcolate le varie D_{ij} per ogni coppia di feature, si costruisce il *grafo di dipendenze*, dove ogni feature è un nodo e si costruisce un arco di peso D_{ij} fra i nodi f_i ed f_j. Di questo grafo calcoliamo il maximum spanning tree (consideriamo quindi gli archi di massima dipendenza). Potremmo quindi ottenere un grafo del genere:

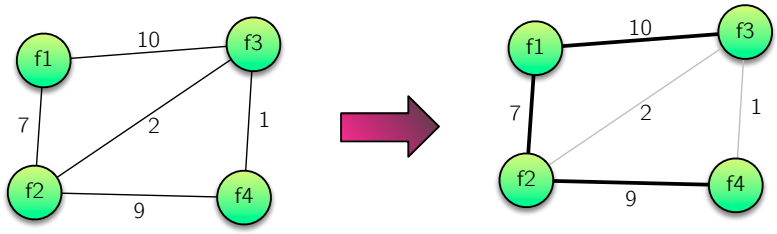


Figura 6.5: Grafo delle dipendenze fra features.

A questo punto, siamo in grado di calcolare la probabilità:

$$p(f_1 \wedge f_2 \wedge f_3 \wedge f_4) = p(f_1)p(f_2|f_1)p(f_3|f_1)p(f_4|f_2)$$

Ovvero, la probabilità di compresenza di tutte le feature è calcolabile come probabilità della radice del maximum spanning tree, moltiplicata per le varie probabilità congiunte ottenute percorrendo semplicemente l'albero (una per ogni arco).

Grazie a questa probabilità, siamo in grado di:

- Calcolare $p(o_i|Rel)$ sfruttando la distribuzione delle features in Rel.
- Calcolare $p(o_i|Ir)$ sfruttando la distribuzione delle features in Ir.

E grazie a queste probabilità (calcolate anche su o_j) possiamo computare l'Equazione 6.3 anche quando le features sono indipendenti.

6.5 Feedback senza l'aiuto dell'utente

Nel caso in cui l'utente non sia in grado di esprimere preferenze in merito ai risultati rilevanti e quelli non rilevanti, possiamo solamente basarci sulla query di partenza: si presuppone quindi che il ranking della prima query sia già sufficentemente raffinato. Vi sono diversi approcci adottabili, fra cui:

- I primi N risultati vengono inclusi nell'insieme Rel e gli altri nell'insieme Ir.
- Si prendono tutti i risultati più simili, fino a che non c'è un grosso salto di similarità fra un risultato e l'altro. In altre parole, se abbiamo quattro

risultati con similarità 0.98, 0.97, 0.80 e 0.85 essendoci un grosso salto fra 0.97 e 0.80 si consideranno i primi due oggetti come appartenenti all'insieme Rel e gli altri come appartenenti all'insieme Ir.

Capitolo 7

Multimedia query processing

7.1 Introduzione

Trattiamo ora i problemi relativi al processing delle query, più precisamente, ci occuperemo di:

- Trattare l'imperfezione (database fuzzy).
- Trattare il ranking di risultati imperfetti.
- Trattare predicati costosi (più precisamente cerceremo di evitare di usare determinate feature qualora non siano necessarie).

Riprendiamo l'esempio fatto a inizio volume (Sezione 1.6.3): avevamo accennato al problema di *combinare* i vari ranking ottenuti sulle feature per poter ottenere un ranking unico da presentare all'utente. Ora, vedremo come farlo. Sappiamo infatti che ogni feature (il colore, le texture, le forme) è una dimensione da considerare (ci si intenda, ogni feature è ovviamente multidimensionale all'interno del suo indice, ma delinea un singolo ranking) e da combinare con le altre. Combinare i risultati può essere complesso, infatti, già da questa query:

```
1  SELECT image P, object obj1, object, obj2
2  where P contains obj1
3    and P contains obj2
4    and obj1.semantical_property is_like "mountain"
5    and obj1.image_property image_match "FujiMountain.png"
6    and obj2.semantical_property is_like "lake"
7    and obj2.image_property image_match "lakeImageSample.png"
8    and obj1.position is_above obj2.position
```

Listing 7.1: Traduzione in pseudo query.

delineiamo sia predicati *crisp* (binari), come il `contains` ma anche predicati fuzzy come il `is_like`. Si tratta di predicati di natura diversa che coesistono all'interno di una stessa query.

7.1.1 Le ragioni dell'imperfezione

L'abbiamo già citate diverse volte, le ragioni per cui abbiamo imperfezioni sono:

- Similarità fra features (arancio e giallo sono difficilmente distinguibili).
- Imprecisione degli algoritmi di estrazione delle feature.
- Imprecisione del linguaggio con cui si esprimono le query.
- Ci può essere match parziale (abbiamo il sistema di ranking ad aiutarci in questo caso).
- Gli indici o i cluster sono imprecisi.

7.2 La logica fuzzy

Per risolvere tutta questa imprecisione si è reso necessario adottare un modello differente rispetto alla logica booleana; un modello che se desiderato si comporti come il modello booleano ma che in generale sia più potente. Si tratta della logica *fuzzy*. Vi sono una serie di insiemi, ed ogni oggetto ha un grado di plausibilità di appartenervi. La particolarità degli insiemi fuzzy sta nel fatto che le plausibilità di due insiemi opposti non danno 1, ovvero, una persona può essere alta con plausibilità 0.9 e bassa con plausibilità 0.3.

Più formalmente, scriviamo che un fuzzy set F con dominio D definisce una funzione di membership:

$$f_F : D \to [0, 1]$$

Un set crips C (booleano) con dominio D definisce invece una funzione di membership:

$$f_C : D \to \{0, 1\}$$

Si noti la differenza fra graffe e quadre: nel primo caso abbiamo un intervallo (da 0 ad 1), nel secondo un set di valori possibili (0 oppure 1).

7.2.1 Definizioni in merito ai fuzzy set

Introduciamo ora un po' di definizioni e notazioni in merito ai fuzzy set. Alcune di queste verranno usate seguito.

Empty fuzzy set. Il fuzzy set vuoto ϕ è definito come:

$$\forall x \in X : f_\phi(x) = 0$$

Cioè ogni elemento x del dominio X ha plausibilità 0 di essere incluso nel set ϕ.

Universal fuzzy set. Il fuzzy set universale u è definito come:

$$\forall x \in X : f_u(x) = 1$$

Cioè ogni elemento x del dominio X ha plausibilità 1 di essere incluso nel set u.

α-universal fuzzy set. Il fuzzy set α-universale $X^{[\alpha]}$ è definito come:

$$\forall x \in X : f_{X^{[\alpha]}}(x) = \alpha$$

Cioè ogni elemento x del dominio X ha plausibilità $alpha$ di essere incluso nel set $X^{[\alpha]}$.

Supporto di un fuzzy set. Il supporto $supp(F)$ di un fuzzy set F è definito come:

$$supp(F) = \{x \in X | f_F(x) > 0\}$$

Cioè un elemento x del dominio X fa parte del supporto di F se ha plausibilità maggiore di zero di appartenere al fuzzy set F.

Altezza di un fuzzy set. L'altezza $height(F)$ di un fuzzy set F è definita come:

$$height(F) = max_{x \in X}\{f_F(x)\}$$

Cioè l'altezza di un fuzzy set F è determinata dalla massima plausibilità che un oggetto ha di appartenervi. Quando un fuzzy set ha altezza 1, si dice *normale*, altrimenti è detto *subnormale*.

Fuzzy set normalizzato. È possibile normalizzare un fuzzy set F ottenendo quindi il fuzzy set F^* così definito:

$$F^* = \frac{F}{height(F)}$$

Il calcolo è effettuato per le plausibilità di tutti gli elementi del fuzzy set (anche se avrà effetto solo sugli elementi del supporto, gli altri hanno valore di plausibilità 0). Si tratta di una normalizzazione classica (abbiamo diviso per il massimo). Chiaramente, vale l'equivalenza:

$$supp(F^*) = supp(F)$$

Cardinalità di un fuzzy set. La cardinalità $card(F)$ di un fuzzy set F è definita come:

$$card(F) = \sum_{x \ inX} f_F(x)$$

Nel caso in cui si trattino plausibilità continue, si può usare un integrale al posto della sommatoria. Nei database probabilistici, la cardinalità è 1 (mentre nel mondo fuzzy possiamo avere cardinalità maggiore di 1).

α-cut di un fuzzy set. L'α-cut di un fuzzy set F è definito come:

$$F^{>\alpha} = \{x \in X | f_F(x) > \alpha\}$$

Cioè un elemento x del dominio X fa parte dell'α-cut di F se ha plausibilità maggiore ad α di appartenere al fuzzy set F. Nel caso in cui $\alpha = 0$ si ottiene l'insieme $supp(F)$. Esiste anche la versione debole (con il maggiore uguale al posto del maggiore).

Kernel di un fuzzy set. Il kernel di un fuzzy set F è il suo α-cut debole con $\alpha = 1$, ovvero:

$$kernel(F) = F^{\geq 1}$$

L'operatore di sottostima

Il fuzzy set A è una sottostima del fuzzy set B se per ogni elemento del dominio la plausibilità di appartenere ad A è minore della plausibilità di appartenere a B. In formule:

$$A \subseteq B \leftrightarrow \forall x \in X f_A(x) \leq f_B(x)$$

Valgono poi le seguenti proprietà:

$$A \subseteq B \leftrightarrow A^{>\alpha} \subseteq B^{>\alpha}$$
$$A \subseteq B \leftrightarrow A^{\geq\alpha} \subseteq B^{\geq\alpha}$$
$$A \subseteq B \leftrightarrow supp(A) \subseteq supp(B)$$
$$A \subseteq B \leftrightarrow height(A) \leq height(B)$$

7.2.2 Funzioni di scoring

Ora che sappiamo come funziona la logica fuzzy ed i suoi predicati, come possiamo usarla? Dobbiamo ricorrere a funzioni di aggregazione (anche dette di scoring) che traducano i nostri operatori logici. Vediamo un esempio introduttivo:

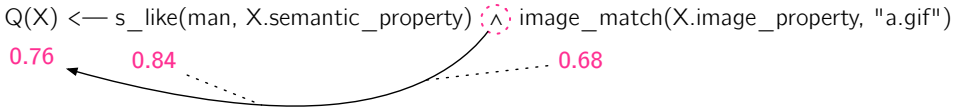


Figura 7.1: Processing della query tramite fuzzy set.

Come vediamo, abbiamo valutato per un certo oggetto $x \in X$ i due predicati `s_like` e `image_match` ed abbiamo ottenuto la plausibilità relativa. Dopodiché, le abbiamo combinate facendo uso dell'AND e abbiamo ottenuto il valore 0.76. Compiendo questo calcolo su ogni oggetto del dominio, possiamo ottenere una serie di valori e quindi stilare un ranking definitivo.

Non resta che capire da dove esce questo valore 0.76: qual è il significato dell'AND? Esistono semantiche diverse?

Un primo approccio potrebbe essere quello di trasformare i risultati dei vari predicati in valori crisp (cioè se maggiori di una certa soglia pari a 1 e 0 altrimenti): ovviamente non ha molto senso come approccio (avremmo sprecato tutto il lavoro fatto per ottenere quei valori!). Definiamo quindi diverse semantiche fuzzy (nessuna è più giusta delle altre: dipende da quale significato vogliamo attribuire agli operatori logici).

Esistono sia semantiche binarie che N-arie, facciamo ora una lista e poi esamineremo pro e contro di ogni semantica.

Semantiche binarie

Le semantiche binarie sono principalmente due.

Semantica del minimo. Ecco la descrizione degli operatori per la semantica del *minimo*:

- $\mu_{P_i \wedge P_j}(x) = \min\{\mu_i(x), \mu_j(x)\}$
- $\mu_{P_i \vee P_j}(x) = \max\{\mu_i(x), \mu_j(x)\}$
- $\mu_{\neg P_i}(x) = 1 - \mu_i(x)$

Nell'esempio fatto prima, con la semantica del minimo, avremmo avuto un risultato di 0.68 (abbiamo usato la prima regola).

Semantica del prodotto. Ecco la descrizione degli operatori per la semantica del *prodotto*:

- $\mu_{P_i \wedge P_j}(x) = \dfrac{\mu_i(x) \times \mu_j(x)}{\max\{\mu_i(x), \mu_j(x), \alpha\}}$ con $\alpha \in [0, 1]$
- $\mu_{P_i \vee P_j}(x) = \dfrac{\mu_i(x) + \mu_j(x) - (\mu_i(x) \times \mu_j(x)) - \min\{\mu_i(x), \mu_j(x), 1 - \alpha\}}{\max\{1 - \mu_i(x), 1 - \mu_j(x), \alpha\}}$
  con $\alpha \in [0, 1]$
- $\mu_{\neg P_i}(x) = 1 - \mu_i(x)$

Si noti che α è utile soltanto per evitare che accadano divisioni per 0.
Nell'esempio fatto prima, con la semantica del prodotto, avremmo avuto un risultato di 0.68 (abbiamo usato la prima regola).

Semantiche N-arie

Se si ha una query che esprime più di due predicati è anche possibile usare semantiche N-arie.

Semantica della media aritmetica. Ecco la descrizione degli operatori per la semantica della *media aritmetica*:

- $\mu_{P_i \wedge ... \wedge P_j}(x) = \frac{\mu_i(x) + ... + \mu_j(x)}{|\{P_i, ..., P_j\}|}$
- $\mu_{P_i \vee ... \vee P_j}(x) = \frac{|\{P_i, ..., P_j\}| - ()\mu_i(x) + ... + \mu_j(x))}{|\{P_i, ..., P_j\}}$
- $\mu_{\neg P_i}(x) = 1 - \mu_i(x)$

Semantica della media geometrica. Ecco la descrizione degli operatori per la semantica della *media geometrica*:

- $\mu_{P_i \wedge ... \wedge P_j}(x) = (\mu_i(x) \times ... \times \mu_j(x))^{\frac{1}{n}}$ con n la cardinalità dei predicati fuzzy usati.
- $\mu_{P_i \vee ... \vee P_j}(x) = 1 - ((1 - \mu_i(x)) \times ... \times (1 - \mu_j(x))^{\frac{1}{n}}$ con n la cardinalità dei predicati fuzzy usati.
- $\mu_{\neg P_i}(x) = 1 - \mu_i(x)$

7.2.3 Comparazione fra le funzioni di scoring

Ora che le abbiamo introdotte, proviamo a rappresentarle e a capire quali vantaggi ci portino.

Norme triangolari e co-norme triangolari

Abbiamo già detto che ci piacerebbero delle semantiche che si comportino come predicati crisp laddove necessario (quando ad esempio abbiamo tutti valori 0 o 1 provenienti dai vari predicati fuzzy). Una certa semantica emula le proprietà di un predicato crisp se rispetta quattro condizioni:

- Condizioni ai limiti.
- Commutatività.
- Monotonicità.
- Associatività.

Se la semantica rispetta queste proprietà per quanto riguarda l'operatore di AND, allora è una norma triangolare; se le rispetta per l'operatore di OR è una co-norma triangolare.

Bellman e Giretz hanno dimostrato che l'unica funzione di aggregazione che preservi tutte queste proprietà sono le semantiche del *minimo* e del *massimo*.

D'altronde una politica del minimo potrebbe essere troppo restrittiva per i nostri scopi: la condizione di monotonicità è troppo debole per permetterci di esprimere alcune delle sfaccettature che vorremmo invece poter rappresentare. Infatti, nella semantica del minimo, accade questo:

- $mu_{\wedge}(0.71, 0.70) = 0.70$
- $mu_{\wedge}(0.99, 0.70) = 0.70$

Non c'è alcuna differenza dal punto di vista del rank finale mentre i rank parziali sono molto diversi (sul primo predicato).

Visualizzazione del minimo

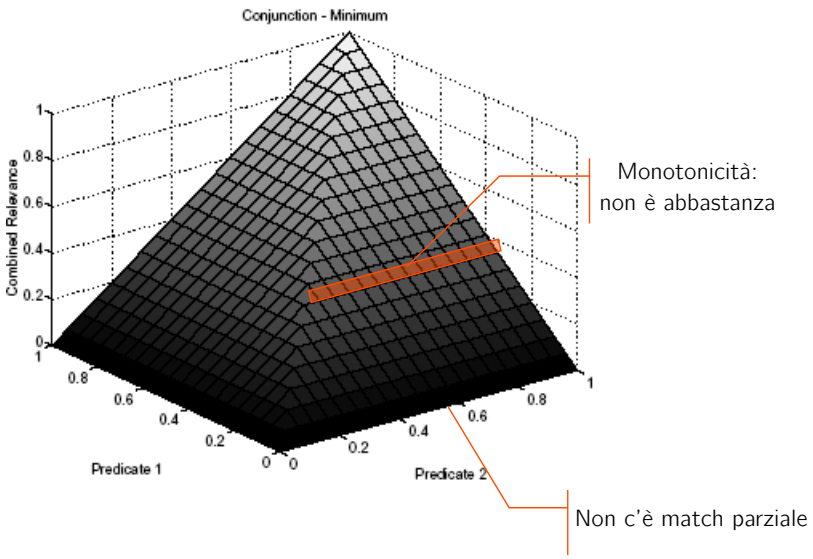


Figura 7.2: Rappresentazione del minimo.

Come vediamo, la condizione di monotonicità fa sì che al variare delle x la y rimanga sempre la stessa (se la y è minore). Inoltre i match parziali sono impossibilitati (appena un elemento è zero, il rank è zero).

Visualizzazione della media aritmetica

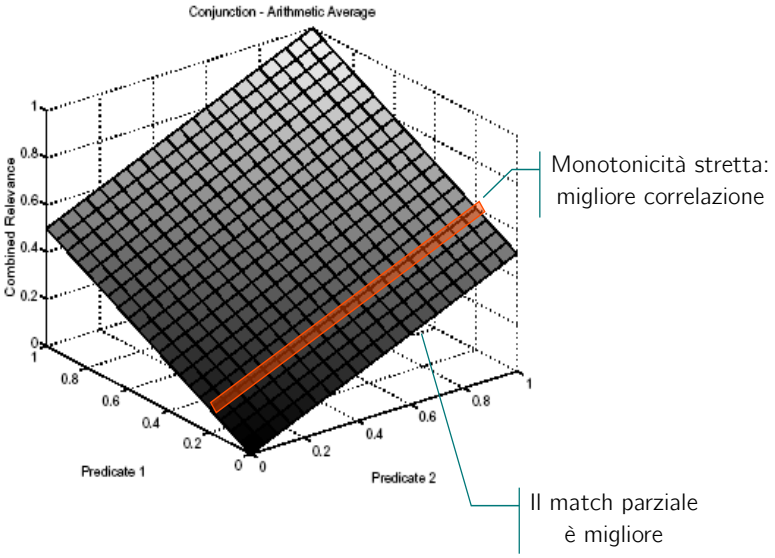


Figura 7.3: Rappresentazione della media aritmetica.

Avendo una monotonicità stretta, la correlazione fra x ed y è migliore dunque i match parziali risultano a loro volta migliorati. Poniamoci però in questa situazione:

Q(X) <— s_like(man, X.semantic_property) ∧ image_match(X.image_property, "a.gif")

0.40	0.00	0.80
0.45	0.10	0.80
0.45	0.00	0.90

Figura 7.4: Problematiche della media aritmetica.

Come vediamo, il rank è lo stesso per i due oggetti, però il secondo dispone anche di uno 0.10 per il primo risultato: è in un certo senso più ''equilibrato''. La media aritmetica non è in grado di cogliere questo aspetto.

Visualizzazione della media geometrica

Come vediamo da Figura 7.5 la media geometrica è in grado di risolvere il problema grazie alla sua inclinazione adattiva, ma putroppo il match parziale è nuovamente perso.

Per ovviare a questa problematica, è stata introdotta una misurazione che prende il nome di *media geometrica parametrica*, che non fa altro che inserire

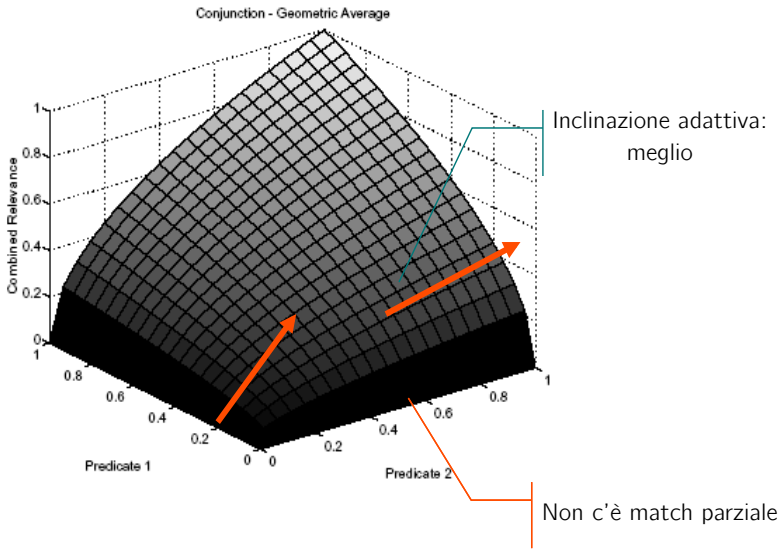


Figura 7.5: Rappresentazione della media geometrica.

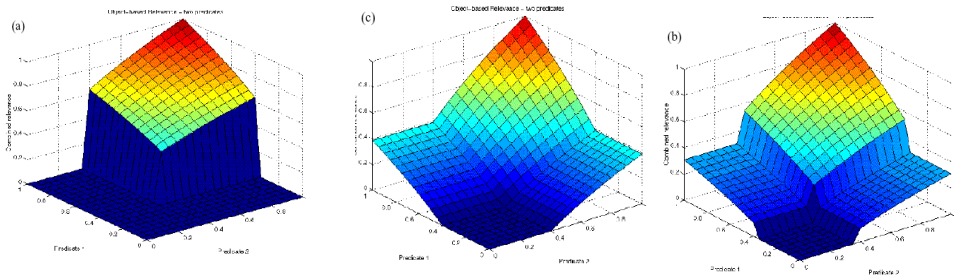
un cutoff (chiamato r_{true}) sotto il quale il valore assunto dal rank è sempre β (e non 0).

La formula è la seguente:

$$\mu_{P_i \wedge \ldots \wedge P_j}(x, r_{true}, \beta) = \frac{((\prod_{\mu_k(x) \geq r_{true}} \mu_k(x))) \times ((\prod_{\mu_k(x) < r_{true}} \beta))^1 n - \beta}{1 - \beta}$$

con n la cardinalità dei predicati fuzzy usati.

Vediamo qualche rappresentazione con parametri diversi della media gometrica parametrica:



r_{true} = 0.4, β = 0.4 r_{true} = 0.4, β = 0.2 r_{true} = 0.4, β = 0.0

Figura 7.6: Rappresentazione della media geometrica parametrica.

7.2.4 Pesare i predicati

Fino ad ora tutti i predicati coinvolti all'interno della query hanno avuto peso identico. Ma se volessimo invece attribuire maggiore importanza ad un predicato rispetto che ad un altro?

Fagin's proposal

Fagin (che abbiamo già incontrato in precedenza) ha dato un grosso contributo teorico al problema, stilando un desiderata per il sistema di distribuzione dei pesi: se le condizioni espresse nel desiderata sono rispettate allora è garantito che gli stessi algoritmi che funzionano per sistemi dove i predicati non sono pesati (e si ha la stessa complessità). Si ha dunque un grande vantaggio, poiché si ha omogeneità nelle tecniche adoperabili (che studieremo in seguito), che si scelga di adottare i pesi oppure no.

Il desiderata. Vediamo dunque quali sono le condizioni espresse da Fagin:

- Se tutti i pesi sono uguali, il risultato deve essere equivalente alla applicazione della funzione di scoring in assenza di pesi.
- Se uno dei predicati ha peso zero, quel predicato può essere rimosso dalla computazione senza aver effetto sul resto dei predicati.
- La funzione di distribuzione dei pesi deve essere continua (questo per garantire la correttezza degli algoritmi).

La soluzione. Consideriamo ora un polinomio particolare, che come dimostreremo soddisfa le tre condizioni espresse dal buon Fagin. Supponiamo di avere m predicati e che l'i-esimo predicato abbia peso θ_i. I pesi sono definiti in modo che la loro somma sia uno, che siano tutti maggiori o uguali a zero e che siano ordinati. In simboli:

- $\theta_1 + \theta_2 + ... + \theta_m = 1$
- $\theta_1, \theta_2, ..., \theta_m \geq 0$
- $\theta_1 \geq \theta_2 \geq ... \geq \theta_m$

Allora possiamo assegnare i pesi usando questi coefficenti (abbiamo la solita funzione di scoring f):

$$\begin{aligned} f_{(\theta_1,\theta_2,\ldots,\theta_m)}(x_1, x_2, \ldots, x_m) = \\ & (\theta_1 - \theta_2)f(x_1)+ \\ & 2(\theta_2 - \theta_3)f(x_1, x_2)+ \\ & \ldots \\ & (m-1)(\theta_{m-1} - \theta_m)f(x_1, x_2, \ldots, x_{m-1})+ \\ & m(\theta_m)f(x_1, x_2, \ldots, x_m) \end{aligned}$$

Sebbene la formula sembri parecchio astrusa, è facile individuare il pattern: l'i-esimo addendo è costruito da un coefficente pari ad i moltiplicato per la sottrazione $(\theta_i - \theta_{i+1})$ e moltiplicato per la funzione di scoring applicata fino al predicato i-esimo.

Il desiderata di Fagin è rispettato, infatti:

- Se tutti i pesi sono uguali (quindi $\frac{1}{m}$), ogni sottrazione fra i pesi (all'interno di ogni addendo) ha risultato zero, lasciando solamente l'ultimo elemento:

  $$m(\theta_m)f(x_1, x_2, \ldots, x_m) = m\frac{1}{m}f(x_1, x_2, \ldots, x_m) = f(x_1, x_2, \ldots, x_m)$$

  Si ha dunque la funzione di scoring applicata su tutti i predicati e senza l'uso dei pesi.

- Se ci sono pesi nulli, questi si troveranno al fondo (poiché ω_m è il peso minimo), quindi sarà sufficente effettuare la somma fino al punto in cui i pesi sono valorizzati.

- Se f era continua (essendo la nuova funziona una somma di diverse applicazioni di f), la continuità è preservata.

Esempio d'uso (con media aritmetica)

Vediamo ora una applicazione di quanto detto usando come funzione di scoring la media aritmetica. Come sappiamo:

$$score(a \wedge b) = \frac{score(a) + score(b)}{2}$$

Volendo pesare i predicati a e b usando i due pesi θ_a e θ_b otteniamo quindi:

$$\begin{aligned} score_{(\theta_a,\theta_b)}(a \wedge b) =& (\theta_a - \theta_b)score(a) + 2\theta_b score(a \wedge b) = \\ & (\theta_a - \theta_b)score(a) + 2\theta_b \frac{score(a) + score(b)}{2} \end{aligned}$$

e quindi in definitiva si ottiene:

$$score_{(\theta_a,\theta_b)}(a \wedge b) = \theta_a score(a) + \theta_b score(b)$$

La forma che si ottiene è assai leggibile: ogni peso è il coefficente dell'applicazione della funzione di score al predicato. Nel caso in cui la funzione di score sia quindi la media aritmetica, potremmo implementare molto più comodamente il sistema dei pesi.

Esempio d'uso (con prodotto)

Se usiamo come funzione di score il prodotto:

$$score(a \wedge b) = score(a) \times score(b)$$

Otteniamo questo risultato:

$$score_{(\theta_a,\theta_b)}(a \wedge b) = (\theta_a - \theta_b) score(a) + 2\theta_b scorea(a) \times score(b)$$

Non siamo fortunati come nel caso della media aritmetica e non c'è una rappresentazione diretta: si sarà costretti a implementare la funzione di scoring pesata seguendo la formula originale stilata da Fagin.

Oltre il desiderata di Fagin

Possiamo ritenerci soddisfatti di quanto ci garantiscono i desiderata di Fagin? Ovviamente no, altrimenti di cosa avremmo parlato in questo paragrafo?.

Non siamo però riusciti a cogliere un aspetto (di cui abbiamo già parlato): la variazione di valore su un predicato che ha peso alto non conta più della variazione su un predicato che invece ha peso basso. Riprendiamo questa figura: Se il predicato `image_match` avesse molta più importanza rispetto al

Figura 7.7: Problematiche della media aritmetica.

predicato `s_like`, vorremmo poter cogliere la variazione del primo e tralasciare quella del secondo (che invece la media geometrica, come abbiamo visto, accentuerebbe). Dobbiamo sfruttare *le derivate parziali*.

Una variazione adattiva. Data una funzione $f(x, y)$, x da maggiore contributo rispetto ad y se:

$$\forall a, b \quad \left.\frac{\partial f}{\partial x}\right|_{\langle a,b\rangle} > \left.\frac{\partial f}{\partial y}\right|_{\langle a,b\rangle}$$

ovvero, se per ogni coppia di punti (cioè valori sui due predicati) la derivata parziale su x è maggiore della derivata parziale su y si ha che x è più rilevante di y.

Valutiamo l'importanza relativa di un predicato x rispetto al predicato y grazie alla funzione:

$$relimp(x, y)|_{\langle a,b\rangle} = \frac{\left.\frac{\partial f}{\partial x}\right|_{\langle a,b\rangle}}{\left.\frac{\partial f}{\partial y}\right|_{\langle a,b\rangle}}$$

Se $relimp > 1$, allora x è relativamente più importante di y. Vediamo come si comportano le due funzioni di score che abbiamo usato come esempi in merito alla variazione adattiva.

Valutazione della variazione adattiva per la media aritmetica. Ricordiamo la funzione di score pesata per la media aritmetica:

$$score_{(\theta_x,\theta_y)}(x \wedge y) = \theta_x score(x) + \theta_y score(y)$$

Dopodiché, calcoliamo le due derivate parziali:

$$\left.\frac{\partial score(x \wedge y)}{\partial score(x)}\right|_{\langle a,b\rangle} = \theta_x|_{\langle a,b\rangle} = \theta_x$$

$$\left.\frac{\partial score(x \wedge y)}{\partial score(y)}\right|_{\langle a,b\rangle} = \theta_y|_{\langle a,b\rangle} = \theta_y$$

Come vediamo, nessuna delle due derivate dipende dal valore dei punti, ma ogni derivata parziale è semplicemente il peso del predicato relativo (sul quale è calcolata la derivata). Questo significa che la media aritmetica *non è adattiva*, come ci conferma anche la $relimp$:

$$relimp(x, y)|_{\langle a,b\rangle} = \frac{\theta_x}{\theta_y}$$

L'importanza relativa è data dal rapporto dei pesi, dunque l'importanza è *esattamente proporzionale* ai pesi, quale che sia il valore della funzione di score sui predicati (come abbiamo detto a e b non sono neanche coinvolti).
Quindi, riassumendo:

- La media aritmetica *non è adattiva*.
- L'importanza relativa è data dal rapporto *esatto* fra i pesi.

Valutazione della variazione adattiva per il prodotto. Ricordiamo la funzione di score pesata per il prodotto:

$$score_{(\theta_x,\theta_y)}(x \wedge y) = (\theta_x - \theta_y)score(x) + 2\theta_b scorea(x) \times score(y)$$

Dopodiché, calcoliamo le due derivate parziali:

$$\left.\frac{\partial score(x \wedge y)}{\partial score(x)}\right|_{\langle a,b\rangle} = \theta_x|_{\langle a,b\rangle} = (\theta_x - \theta_y) + 2\theta_y b$$
$$\left.\frac{\partial score(x \wedge y)}{\partial score(y)}\right|_{\langle a,b\rangle} = \theta_y|_{\langle a,b\rangle} = 2\theta_y a$$

Questa volta i punti sono presenti all'interno della derivata parziale, dunque il prodotto *è adattivo*. Inoltre, calcolando la $relimp$:

$$relimp(x,y)|_{\langle a,b\rangle} = \frac{(\theta_x - \theta_y) + 2\theta_y b}{2\theta_y a}$$

possiamo anche studiare la correlazione fra il peso dei predicati e il valore da loro assunto. In pratica se noi assegniamo pesi 2/3 ed 1/3, l'importanza relativa di ogni predicato non sarà esattamente 2/3 ed 1/3 ma sarà un valore che tiene conto anche di a e b, cioè il valore assunto dalla funzione di score dei singoli predicati (è quindi lampante che il prodotto sia adattivo).
Quindi, riassumendo:

- Il prodotto *è adattivo*.
- L'importanza relativa non è data dal semplice rapporto fra pesi (si può osservare la correlazione studiando la funzione $relimp$).

Può risultare interessante studiare il comportamento del prodotto qualora si abbiano score uguali per predicati con peso diverso (supponiamo $\theta_x > \theta_y$). Se gli score sono uguali ($a = b$), la $relimp$ diventa:

$$relimp(x,y)|_{\langle a,a\rangle} = 1 + \frac{(\theta_x - \theta_y)}{2\theta_y a}$$

che è maggiore di 1. Ciò significa che x è relativamente più importante di y (quando $relimp > 1$ significa che la derivata parziale del primo parametro è maggiore di quella del secondo parametro), il che è gradito (poiché il peso assegnato ad x era maggiore di quello assegnato a y).
Quindi, in caso di score uguali il ranking è comunque coerente rispetto a quanto assegnato tramite i pesi.

Qualora si vogliano assegnare i pesi come esponenti, la funzione di score sarebbe la seguente:

$$score_{(\theta_x,\theta_y)}(x \wedge y) = score(x)^{\theta_x} \times score(y)^{\theta_y}$$

E si avrebbe *relimp*:

$$relimp(x,y)|_{\langle a,b\rangle} = \frac{\theta_x b}{\theta_y a}$$

Valgono quindi tutte le riflessioni fatte quando si è studiato il prodotto.

7.3 Algoritmi di ranking

Studiamo ora come impiegare tutte le tecniche che abbiamo studiato al fine di automatizzare il processo di ranking. Gli algoritmi che vedremo sono quelli che abbiamo già citato parlando del Fagin proposal e di come le funzioni di score pesate siano impiegate all'interno degli algoritmi nello stesso modo di quelle non pesate senza inficiare sulla complessità (qualora il desiderata sia rispettato).

Per contestualizzare il problema, riprendiamo la nostra query d'esempio e consideriamo questo risultato:

Q(X) <— s_like(man, X.semantic_property) ∧ image_match(X.image_property, "a.gif")

	0.90	X_2	0.85	X_3
	0.80	X_5	0.80	X_5
?	0.70	X_6	0.75	X_2
	0.60	X_4	0.74	X_6
	0.50	X_1	0.74	X_1
	0.40	X_3	0.70	X_4

Figura 7.8: Risultato della computazione di due predicati.

Grazie a quanto studiato in precedenza, siamo in grado di combinare i risultati relativi ai due predicati giungendo ad una sola lista di punteggi che, ordinati, ci forniscono il ranking finale.

L'idea più intuitiva sarebbe quindi quella di eseguire due query di top-K separate, poi combinare i risultati e ottenere i top-K totali. Purtroppo però non abbiamo garanzia che i top-K relativi ad un predicato siano anche i top-K relativi all'altro, quindi l'unione dei due insiemi di oggetti potrebbe non darci K risultati, bensì un numero minore. D'altronde non siamo disposti a processare l'intero database così da ottenere tutti gli oggetti ordinati relativamente al primo ed al secondo predicato per poi combinare i risultati. Si rende quindi necessario sviluppare degli algoritmi che minimizzino l'accesso e ci forniscano il risultato voluto.

Negli algoritmi che studiamo, si assume che:

- Q sia monotona.

- Vi sia una funzione sui predicati ''`sorted_access`'' che restituisca gli elementi in ordine decrescente di punteggio.

- Vi sia una funzione sui predicati ''`random_access`'' che restituisca il punteggio di un certo oggetto X_i desiderato.

7.3.1 Ranked join per query top-K (per predicati in AND)

L'algoritmo si sviluppa in tre fasi. Nell'esempio trattato useremo $K = 3$ e useremo come funzione di scoring la media aritmetica.

Fase di `sorted_access`. Si elencano i risultati di tutti i predicati coinvolti (due nell'esempio) finché non si ottengono K oggetti in comune. Per fare questo si sfrutta il `sorted_access`, quindi i risultati sono restituiti in ordine. Al termine di questa fase si ha garanzia che fra tutti gli oggetti incontrati vi si trovino i migliori K (e ciò è vero grazie al fatto che Q sia continua). Non è però detto che tali K siano proprio quelli in comune fra le liste dei due predicati; è necessaria una fase di scrematura.

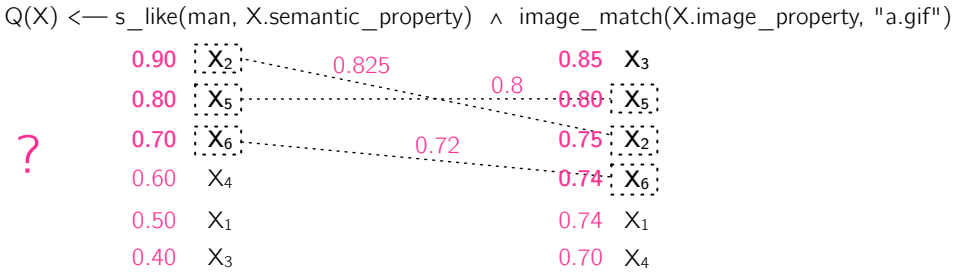


Figura 7.9: La prima fase individua 4 oggetti totali.

Nell'esempio sopra ci si è fermati nonappena si sono trovati tre oggetti comuni (X_2, X_5, X_6) e anche X_3. Fra questi quattro risultati si trovano i migliori tre (cioè i top-K).

Fase di `random_access`. Per poter identificare quali siano i K migliori risultati fra tutti quelli trovati (in comune e non) è necessario calcolare la funzione di score combinata anche per gli oggetti rimasti spaiati. È possibile compiere questa operazione grazie al `random_access` (se ne compirà uno per ogni oggetto spaiato).

In Figura 7.10 vediamo che è stato effettuato un `random_access` per X_3 e si è calcolata la funzione di score fra i due predicati relativi ad esso.

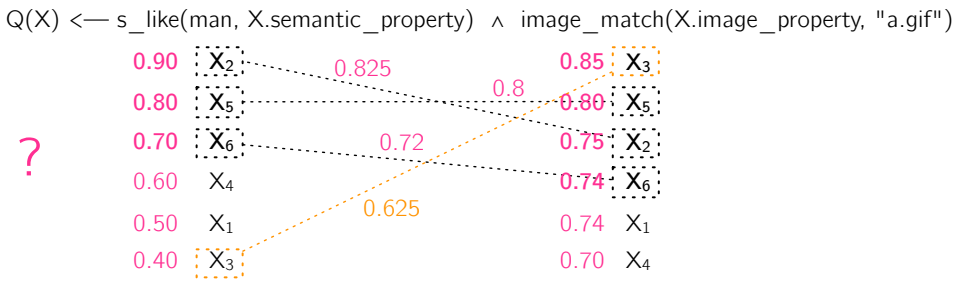


Figura 7.10: L'unico oggetto spaiato per cui fare `random_access` è X_3.

Fase di ordinamento dei risultati. A questo punto è sufficente prendere i K oggetti che hanno funzione di scoring combinata maggiore e restituirli all'utente:

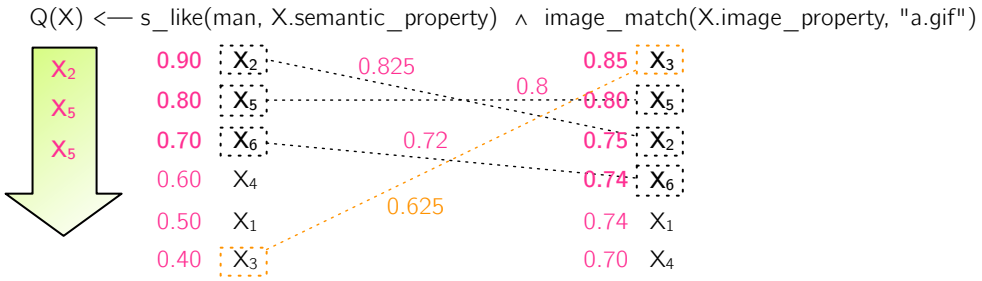


Figura 7.11: Il risultato è restituito all'utente.

I risultati del nostro esempio sono quindi X_2, X_5, X_6. Grazie all'impiego di questo algoritmo, non abbiamo *mai fatto accesso* né ad X_1 né ad X_4. Se immaginiamo un database molto popolato, il risparmio in termini computazionali è evidentemente enorme.

Miglioria in caso di semantica del minimo

Qualora la funzione di score sia il minimo è possibile attuare una miglioria all'algoritmo appena visto. Si sceglie un predicato a caso e se ne esamina la lista di oggetti (sempre in maniera ordinata, sfruttando il `sorted_access`).
Per ogni oggetto trovato nella lista di quel predicato si effettua un `random_access` nella lista degli altri predicati, fino a che il nuovo elemento da esaminare ha valore associato minore dell'attuale peggiore candidato (il K-esimo quindi). Al termine di questa fase, si deve come sempre ricorrere alla scelta dei top-K effettivi.
Come osserviamo in Figura 7.12, nel nostro esempio ci siamo fermati poiché il valore di X_4 (0.60) è minore di 0.70, che è l'attuale valore del terzo (ovvero l'ultimo) candidato. Grazie alla semantica del minimo, sappiamo che includere

Figura 7.12: Il valore di X_4 è minore di 0.7.

X_4 non avrebbe senso poiché il suo valore dato dalla funzione di score sarebbe al massimo 0.60 che è già peggiore di quanto sia il peggior candidato attuale (0.70).

Come si nota, abbiamo risparmiato anche l'accesso ad X_3 (al quale invece avevamo fatto accesso prima).

Riflessioni in merito all'algoritmo

Questo algoritmo (e le sue variazioni) sono molto vantaggiosi poiché sono *incrementali*: possiamo partire da quanto già fatto durante l'esecuzione di una query top-5 per ottenere una query top-10.

7.3.2 Ranked join per query top-K (per predicati in OR)

Vediamo anche una versione particolare per i predicati in OR con semantica del massimo. Tale versione è addirittura più semplice dell'ultima che abbiamo visto: è infatti sufficiente considerare i primi K risultati di tutte le liste dei predicati coinvolti (senza cercare oggetti in comune!) e poi unirle, al fine di trovare i migliori K (valutando chiaramente la funzione di score).

L'algoritmo è estremamente semplice poiché grazie all'operatore di OR non ci interessa che entrambi i predicati siano rispettati (e quindi che gli oggetti siano in comune) e grazie alla politica del massimo abbiamo garanzia che fra i primi K risultati relativi ad ogni predicato si trovino effettivamente i K migliori.

7.4 Processare query top-K in database regolari

Le tecniche che abbiamo studiato fin'ora prevedono particolari engine che effettuino il ranking. Ma, nel caso in cui non si disponga di questi engine, è comunque possibile processare query top-K?

7.4.1 Il formalismo di Chauduri e Gravano

Chauduri e Gravano hanno formalizzato come esprimere query tipiche di database relazionali aggiungendo però le funzionalità relative a query top-K. Si tratta quindi di database che *contengono comunque multimedia*, ma non dispongono delle engine per effettuare il ranking così come l'abbiamo studiato fin'ora.

```
select oid
from Repository
where Filter_condition
order[k] by Ranking_expression
```

Listing 7.2: Schema di query top-K su database regolari.

Abbiamo introdotto:

- `Filter_condition` che esprime delle condizioni secondo le quali selezionare i risultati. Si tratta ovviamente di condizioni di tipo multimediale (altrimenti saremmo nel caso classico di un database relazionale contenente dati esatti).
- `Ranking_expression` che esprime secondo quali principi i risultati devono essere ordinati.
- `order[k]` che esprime la quantità di risultati che vogliamo (il K in pratica).

Ad esempio, potremmo avere una query del tipo:

```
select oid
from Repository
where (v => 0.5 and p <= 0.9) or f >= 0.9
order[10] by max(f, v)
```

Listing 7.3: Esempio di query top-K su database regolari.

Ciò che vogliamo fare è sfruttare le condizioni indicate nel filtro ed i principi di ranking specificati al fine di non esplorare tutto il database e quindi (come sempre) ridurre gli accessi.

7.4.2 Approccio algoritmico

Ecco quindi l'approccio algoritmico adoperabile:

1. Sfruttare le statistiche del database per trovare una soglia di score S_q che fungerà da filtro.
2. Si costruisce una query C_q che restituisca tutte le tuple che hanno uno score maggiore a S_q.

3. Si esegue C_q.

4. Se vi sono almeno k (dove k è il valore specificato in `order[k]`) tuple con punteggio maggiore di S_q, allora si ritornano i primi k oggetti. Se invece non vi sono abbastanza risultati (cioè ve ne sono meno di k) allora si ripete il procedimento con una S_q meno restrittiva.

L'idea è quindi abbastanza banale: si sfrutta ciò che si sa in merito al database per imporre una soglia, dopodiché su di questa si genera una query (in classico SQL) e si cercano i k migliori risultati fra quelli così ottenuti.

7.4.3 Adoperare le condizioni di filtro

Dei quattro punti esposti prima è certamente il primo ad essere il più nebuloso. Vediamo sul nostro esempio che cosa significhi ''sfruttare le statistiche'' al fine di ottenere una soglia.

Innanzi tutto si assume la presenza delle seguenti tipologie d'accesso:

- `GradeSearch(attribute, value, min_grade)`, che restituisce tutti gli oggetti che sull'attributo `attribute` hanno un valore almeno pari a `min_grade`.

- `TopSearch(attribute, value, count)`, che restituisce i migliori `count` oggetti (relativamente all'attributo `attribute`).

- `Probe(attribute, value, {oid})`, che restituisce il valore sull'attributo `attribute` per l'oggetto con identificatore `oid`.

Date le condizioni espresse nel filtro, possiamo procedere in due modi diversi:

- Valutando ogni condizione singolarmente (sfruttando la `GradeSearch`) e poi sfruttare gli operatori logici fra le condizioni per rimuovere i risultati non coerenti (usando `GradeSearch` e `Probe`).

- Sfruttare le statistiche per scegliere quale delle condizioni sia la più selettiva, valutare quella condizione (sempre con `GradeSearch`) e poi sui risultati ottenuti applicare gli operatori logici (ottenendo quindi un prune iniziale molto più corposo) (usando `Probe`).

È chiaramente la seconda scelta la più interessante, quella adottata in pratica.

Nel nostro esempio avremmo potuto quindi compiere:

- Ottenere tutti gli oggetti per cui `v >= 0.5 or f => 9` e poi verificare la condizione in `and` in merito a p (quindi usando la `Probe` su tutti gli oggetti restituiti e verificando che il valore sia almeno 0.09).

- Ottenere tutti gli oggetti per cui `p >= 0.9 or f=> 9` e poi verificare la condizione in `and` in merito a v (sempre usando la `Probe`).
- Compiere direttamente l'`and` fra le condizioni per `v` ed `p` e poi valutare la condizione di `or` (che però reincluderebbe la ricerca su tutto il database).

Saranno quindi le statistiche del database a suggerisci quale condizione esaminare per prima (compiendo il ciclo di cui abbiamo parlato nella Sezione 7.4.2) e quindi risparmiare accessi (poiché la condizione scelta sarà la più selettiva).

7.4.4 Sulla complessità

Usando l'approccio di Fagin per trovare il numero stimato di oggetti necessari per generare almeno k risultati si ha complessità:

$$L = k^{\frac{1}{n}} \cdot O^{1-\frac{1}{n}}$$

supponendo l'indipendenza delle n sottoquery.

Capitolo 8

Approfondimenti

In questo capitolo tratteremo alcune ulteriori tecniche che meritano particolare attenzione.

8.1 Skyline queries

Le skyline queries (e le tecniche che le risolvono) sono una particolare tipologia di query che si distingue da quelle studiate fin'ora poiché è possibile sfruttarle anche in database che non siano prettamente multimediali. Si tratta infatti di query la cui risoluzione non richiede di avere dati particolarmente trattati, ma molto semplicemente dati che siano rappresentabili come punti su un piano (quindi ad esempio il prezzo di un hotel e la sua distanza dal mare, dato facilmente serbabile all'interno di una base di dati tradizionale). Chiaramente, le tecniche per risolvere query di skyline funzionano ugualmente anche con database multimediali.

8.1.1 La definizione di skyline

Innanzi tutto parliamo del concetto di skyline: dato un set di punti $p_1, ..., p_N$, una query *skyline* restituisce un insieme di punti P (detti punti della skyline) tale che ogni punto $p_i \in P$ non sia dominato da nessun altro punto del database.

Cosa si intende con il verbo ''*dominare*'' ? Un punto p_i domina un altro punto p_j se e solo se *tutte* le coordinate del punto p_i sono ''migliori'' rispetto alle coordinate del punto p_j. Il concetto di ''migliore'' dipende dal nostro interesse: potremmo voler massimizzare o minimizzare i valori delle coordinate dei punti. Se prendiamo l'esempio di prima in cui abbiamo distanza dal mare e il prezzo dell'hotel, saremo chiaramente interessati a minimizzare le coordinate dei pun-

ti, ovvero ad avere distanza dal mare e prezzo molto bassi (il più possibile). Dunque un hotel che ha distanza dal mare di 2 metri e prezzo di 100 euro dominerà un hotel che ha distanza dal mare di 3 metri e prezzo 150 euro.

Non dimentichiamo però che noi siamo interessati a quei punti che *non sono dominati*, quindi è sufficiente che il valore di una delle coordinate del punto non sia ''battuta'' dal valore sulla stessa coordinata di un qualche altro punto per far sì che quel punto non sia dominato (poiché dovrebbe essere battuto su ogni coordinate per essere dominato).

Un esempio

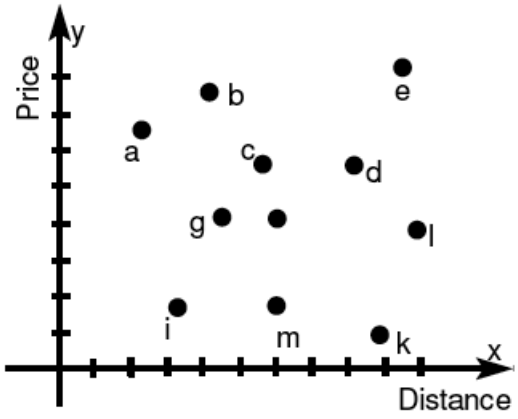


Figura 8.1: Un esempio di database.

Consideriamo questo database sul quale vogliamo compiere una query di skyline. Stiamo considerando l'esempio dell'hotel a cui abbiamo fatto riferimento fin ora: sulle ordinate abbiamo il prezzo e sulle ascisse abbiamo la distanza. Useremo quindi una relazione di minore per stabilire quali punti siano i migliori, ma tutti i discorsi che faremo valgono anche quando cerchiamo le coordinate massime.
L'insieme di skyline per questo esempio è:

$$\{a, i, k\}$$

questo perché:

- i non è dominato da nessun punto, giacché qualunque punto che abbia miglior distanza (cioè solo a) non ha miglior prezzo. Inoltre, qualunque punto che abbia minor prezzo (quindi k) non ha minor distanza. Abbiamo quindi determinato che non esiste alcun punto che batta i sia intermini di distanza che in termini di prezzo (contemporanamente), ergo i non è dominato (e quindi fa parte dell'insieme skyline).

- a è il punto che ha minore distanza del database: non esiste quindi un punto che, seppur migliore in termini di prezzo (c, d, g e molti altri lo sono) riesca a dominarlo (poiché a avrà sempre la miglior distanza). a entra perciò nell'insieme skyline.

- k è il punto che ha minore prezzo del database: non esiste quindi un punto che, seppur migliore in termini di distanza (m, i, g e molti altri lo sono) riesca a dominarlo (poiché k avrà sempre il miglior prezzo). k entra perciò nell'insieme skyline.

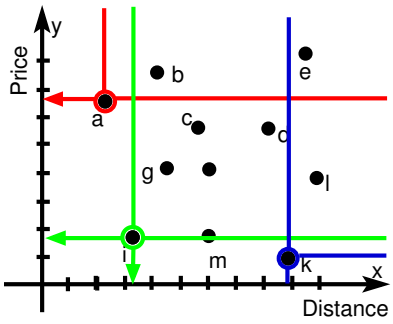


Figura 8.2: Un esempio di database.

Curiosità: la query di skyline è così chiamata perché l'insieme dei punti di skyline assomiglia al profilo di grattaceli.

Skyline query per fare pruning

Si noti che non esiste una funzione di scoring che combini i due valori sulle coordinate, ovvero non si hanno degli score parziali poi combinati per ottenere un ordine totale: ciò che viene restituito è semplicemente un insieme di oggetti ''plausibilmente interessanti''. Esistono però alcune garanzie in merito all'insieme di skyline: se si usa una funzione di scoring monotona (media artimetica, media geometrica, minimo, ecc.), allora se un certo oggetto ω è il top-1, siamo sicuri che tale oggetto si troverà nell'insieme di skyline (questo grazie alla monotonia).

Quindi, se usiamo un algoritmo molto veloce per ottenere un insieme di skyline, possiamo fare *prune* dei risultati e applicare in seguito la funzione di scoring per ottenere un ranking più preciso (ma effettuato su un insieme ristretto di elementi).

Si noti che la funzione di scoring deve essere monotona e non strettamente monotona.

8.1.2 Soluzioni algoritmiche

Andiamo ora a vedere differenti approcci algoritmici al fine di ottenere l'insieme di skyline. Ecco la lista dei metodi che vedremo:

- Block Nested Loop.
- Divide and Conquer.
- Plane sweep.
- Nearest Neighbor Search.
- Branch and Bound Skyline.

Block Nested Loop

Una query di skyline su un database D può essere vista come un self-join:

$$skyline(D) = D - \prod_{o_j}(D_1 \bowtie_{o_i \in D_1, o_j \in D_2, dominating(o_i, o_j)} D_2)$$

Chiaramente questa operazione ha costo $O(\mid D \mid^2)$. Come tutte le query, è possibile un Block Nested Loop per ridurre la quantità accessi in memoria e quindi ridurre la complessità totale dell'algoritmo.

Si usa in realtà una versione modificata del Nested Block, che permette una ottimizzazione ancora superiore. Si utilizzano tre strutture diverse oltre al database:

- La `window`, una area di memoria mantenuta in RAM e di dimensione contenuta (più grande è, meglio è). Gli oggetti mantenuti nella `window` non sono dominati (tranne in alcuni casi, che spiegheremo in seguito).
- Il `trash`, zona di memoria dove vengono messi gli oggetti che sono dominati (dagli oggetti mantenuti nella finestra).
- Il `tempfile`, una zona di memoria che ha lo stesso significato della `window`, ma che funge da overflow nel caso in cui la `window` sia piena.

Sia `trash` che `tempfile` sono mantenuti in memoria secondaria.

Si procede dunque ad uno scan del database. Ogni iterazione si considera quindi un oggetto o_i: tale oggetto viene confrontato con ogni oggetto o_j nella `window`, e possono verificarsi tre diversi scenari:

- Se l'oggetto o_i domina o_j, allora quest'ultimo viene rimosso dalla `window` e viene inserito nel `trash`. Si prosegue con l'oggetto successivo ad o_j mantenuto nella `window`.

- Se l'oggetto o_i è dominato da o_j, allora o_i viene inserito nel `trash` e si considera l'oggetto successivo ad o_i del database.
- Se nessuna delle due condizioni è vera, si procede ad esaminare l'oggetto successivo ad o_j.

Se, una volta confrontati tutti gli oggetti nella `window` (ed aver rimosso gli eventuali o_j dominati da o_i), o_i non è stato dominato, possiamo inserire a pieno diritto o_i nella `window`. Potremmo però avere una `window` già piena: in tal caso o_i viene inserito nella zona `tempfile`.

L'algoritmo proseguirà in questo modo, tenendo conto *solamente* degli elementi nella `window` e non quelli nel `tempfile`! Questo significa che al termine dell'algoritmo gli oggetti mantenuti nella `window` potrebbero ancora essere dominati da quelli nel `tempfile`: si fa per questa ragione una seconda passata, ovvero, il `tempfile` diventa il nuovo database da cui partire. Si procederà in questo modo, con iterazioni man mano meno corpose (il `tempfile` diminuirà sempre più), finché non si farà più uso del `tempfile` e nella `window` rinverremo i risultati desiderati (gli oggetti facenti parte della skyline).

Quando inizia una nuova iterazione non si partirà dalla finestra così com'è, ma potremo già considerare come risultato parziale tutti quegli oggetti che hanno timestamp di inserimento nella finestra minore del timestamp del primo oggetto mantenuto nel `tempfile`. Infatti, tutti gli oggetti che corrispondono a tale caratteristica sono stati inseriti e sono rimasti nella `windows` prima dell'inserimento del primo oggetto nel `tempfile` e quindi sono certamente non dominati da tutti gli oggetti che abbiamo incontrato nel futuro (poiché ogni o_i viene *sempre* confrontato con tutti gli oggetti nella `window` e poi eventualmente messo nella zona `tempfile`). Grazie a questa accortezza potremo svuotare preventivamente parte della finestra e permettere all'iterazione successiva di confrontare un numero più ampio di oggetti (e quindi ridurre globalmente il numero di iterazioni).

Divide and conquer

Vediamo direttamente l'applicazione di questo metodo sull'esempio di cui abbiamo parlato prima. L'idea è quella di suddividere il problema principale in tanti sottoproblemi, di dimensione tale da essere mantenuti in memoria centrale nella loro interezza come vediamo in Figura 8.3.
Come vediamo, sono computate quattro differenti query di skyline sulle quattro diversi sottospazi. La suddivisione può essere fatta sfruttando il punto medio o il punto più bilanciato. Si noti che laddove la suddivisione sia regolare (come quella mostrata in figura), è possibile ottimizzare la computazione: il quadrante in alto a destra, ad esempio, non è neanche da prendere in considerazione (poiché ogni elemento incluso in quel quadrante è dominato da un

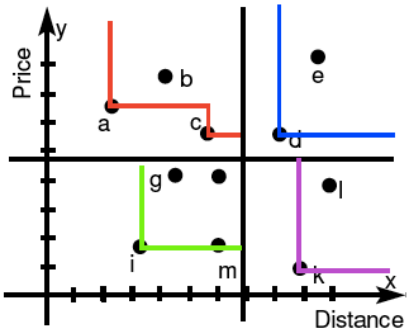


Figura 8.3: Vengono computate quattro query di skyline.

qualunque elemento nel quadrante in basso a sinistra). Inoltre i risultati ottenuti dalla computazione sui due quadranti in alto a sinistra ed in basso a destra non veranno neanche confrontati fra loro, poiché per costruzione nessuno degli oggetti nell'insieme skyline di uno di quei sottospazi potrà mai dominare un oggetto dell'altro quadrante.
Il punto fondamentale è che per come abbiamo definito il concetto di ''dominare'' sappiamo che se un oggetto non fa parte della skyline su un sottospazio, certamente non farà parte della skyline sullo spazio intero. Grazie a questa relazione siamo certi che unire i risultati della computazione su più sottospazi ci permetta comunque di trovare l'insieme di skyline globale senza escludere alcun risultato.

Plane sweep

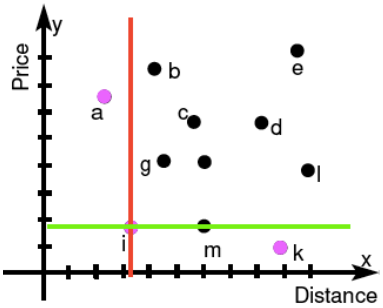


Figura 8.4: Viene eseguito un plane sweep.

Abbiamo già parlato della tecnica di plane sweep (Sezione 3.5.2), quindi la diamo per scontata: per ogni dimensione si fa scorrere un piano e si include il primo punto incontrato all'interno dell'insieme di skyline (e sono i punti *a* e *k*). Inoltre, se c'è un punto (un dato) dove i piani si incrociano, anche quello viene aggiunto all'insieme di skyline (è il caso di *i*, in figura). L'algoritmo può

terminare a questo punto poiché qualsiasi altro punto sarà dominato da quello di incrocio dei piani (nel nostro caso qualsiasi punto è dominato da i).

Nearest Neighbor Search

Questa tecnica è leggermente più complessa, quindi la trattiamo con più calma. Faremo uso della misura di distanza $L1$ (distanza di Manhattan) che come sappiamo è monotona.

Il primo passo consiste nell'andare ad individuare il punto più vicino all'origine, nel nostro caso i:

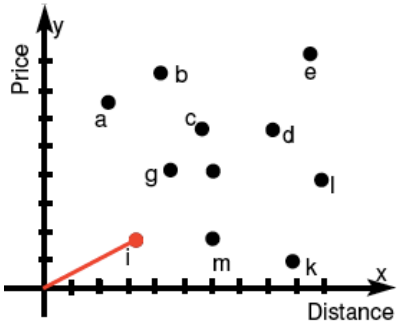


Figura 8.5: Primo passo della nearest neighbor search.

Il punto viene inserito nell'insieme di skyline e suddivide il nostro spazio in quattro parti:

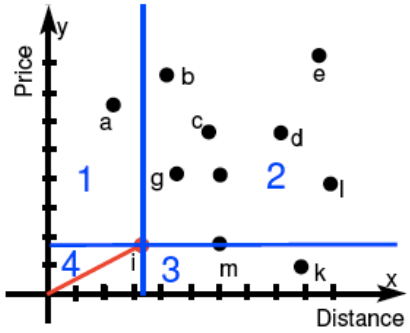


Figura 8.6: Secondo passo della nearest neighbor search.

A questo punto si procede ricorsivamente su ognuno di questi spazi, facendo però attenzione al fatto che:

- La zona 4 sarà sempre vuota (per costruzione).

- La zona 2 non è da elaborare poiché il punto scelto domina tutti i punti mantenuti in quel sottospazio (è un ragionamento simile a quello che abbiamo fatto parlando della tecnica divide and conquer).

Rimangono così solamente le aree 1 e 3 da esaminare.

Partendo dalla zona identificata come 1, identifichiamo il punto più vicino all'origine (attualmente *i*) e quindi il punto *a*: Anche qui, come prima, abbiamo

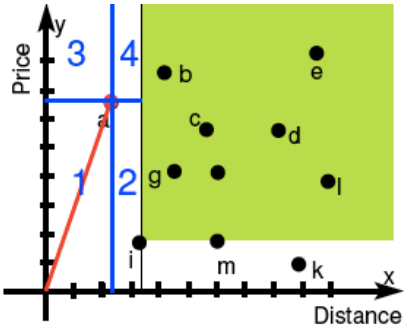


Figura 8.7: Terzo passo della nearest neighbor search.

identificato quattro aree (i label sono in senso inverso rispetto a prima, comunque due aree non sono da esaminare). Le altre due aree rimaste sono vuote. Torniamo quindi indietro alla chiamata ricorsiva che aveva identificato *i*, e passiamo all'analisi della zona 3. Lì, identifichiamo *k* (che darà origine a due zone vuote e a due zone da non esaminare) ad abbiamo così concluso.

Branch and Bound Skyline

La prima cosa da fare è costruire un R-tree sui dati. Dopodiché si sfrutta una coda di priorità ove andiamo ad inserire tutte le regioni del primo livello (appese quindi alla radice), ordinate in base al punto in basso a sinistra (maggiore priorità per i punti più vicini all'origine). Vediamo l'applicazione di questo primo passo in Figura 8.8.
Si procede pescando un elemento dalla queue ed esplodendolo (inserendo chiaramente gli elementi in maniera ordinata nella coda) come vediamo in Figura 8.9.
Giacché il primo elemento è ancora una regione, proseguiamo esplodendola ulteriormente (Figura 8.10). Adesso che il primo elemento della coda è un punto, e dato che l'insieme di skyline è vuoto, il punto viene direttamente inserito nell'insieme di skyline.
Si procede in questo modo, intervallando però delle fasi di pruning dalla coda (onde evitare che cresca troppo) sfruttando i punti già presenti. Ad esempio, in Figura 8.11 usiamo *i* per eliminare alcuni elementi della coda. Quando

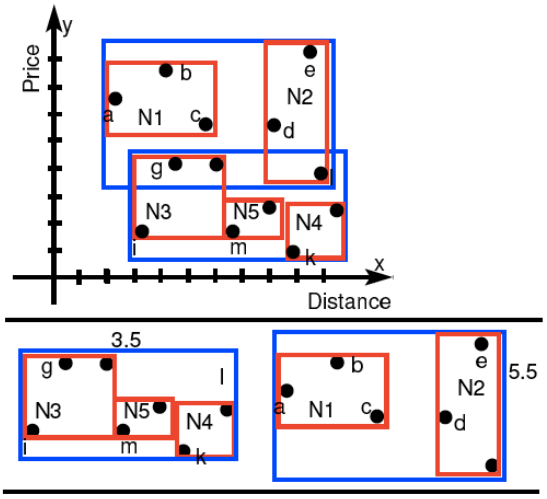


Figura 8.8: Creazione della coda di priorità basata sull'R-tree.

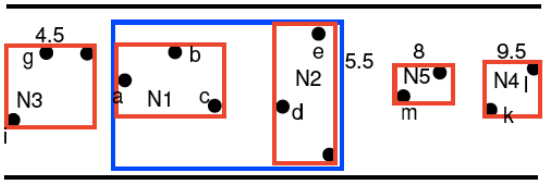


Figura 8.9: È esplosa la prima regione della coda.

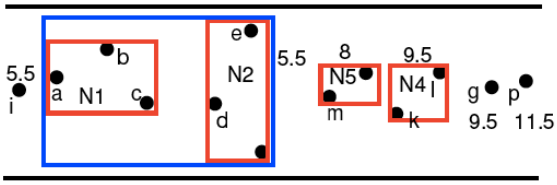


Figura 8.10: È esplosa la prima regione della coda.

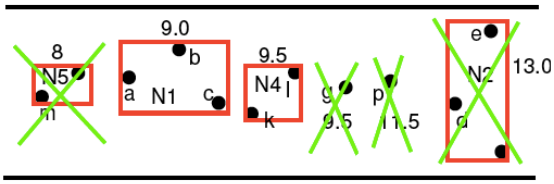


Figura 8.11: Viene fatto pruning nella coda.

troviamo un punto alla testa della coda, verifichiamo che questo non sia dominato dai punti già presenti nell'insieme di skyline: se così non è, possiamo

aggiungerlo all'insieme.

8.2 Locality Sensitive Hashing (LSH)

Il secondo argomento sul quale vogliamo porre l'attenzione è il *Locality Sensitive Hashing* (in breve LSH). Come sappiamo le funzioni di hash funzionano molto bene quando si cerca l'uguaglianza fra elementi (si deve ovviamente considerare il problema delle collisioni), ma nella nostra applicazione vediamo pochi scopi: siamo infatti interessati ad un principio di località, ovvero, desideriamo funzioni di hash che mappino oggetti simili in ''zone'' simili. Disponendo di funzioni di hash siffatte, possiamo poi definire una griglia entro la quale gli oggetti vengono mappati dalla funzione di hash e quindi in definitiva risolvere query di range (prendendo una o più celle). La tecnica appena descritta (l'uso di una griglia) è uno dei vari approcci adottabili: faremo chiarezza sull'argomento in seguito.

8.2.1 Obiettivo: trovare una funzione di hash

La domanda che ci poniamo è: possiamo sviluppare una funzione di hash *random* che rispetti il principio di località? Definiamo il problema in maniera più precisa.

Consideriamo una funzione di similarità $sim()$ (ad esempio Hamming o $L1$). Una funzione di hash $h()$ che rispetti la località relativa a $sim()$ rispetta questa proprietà:

$$prob(h(o_1) = h(o_2)) = sim(o_1, o_2)$$

cioè la probabilità di collisione della funzione di hash è uguale alla similarità fra gli oggetti: oggetti simili avranno probabilità alta di collidere sullo stesso valore della funzione di hash, mentre oggetti diversi avranno scarsa probabilità di collisione. La sfida è quindi quella di trovare una funzione $h()$ appropriata data una certa $sim()$.

8.2.2 Definizione: famiglie LSH

Introduciamo una definizione formale che ci sarà utile in seguito. Una famiglia LSH (cioè un insieme di funzioni di hash), chiamata H, è detta $(r, c \cdot r, P_1, P_2)$-*sensitive* se per qualsiasi coppia di oggetti o_i e o_j e una qualsiasi funzione $h() \in H$ valgono le seguenti proprietà:

- $P_1 > P_2$.
- Se la $dist(o_i, o_j) \leq r$ allora $prob(h(o_i) = h(o_j)) \geq P_1$, ovvero, qualsiasi coppia di oggetti che disti al più r (quindi sufficientemente simile) deve avere probabilità di collisione almeno P_1.

- Se la $dist(o_i, o_j) \geq c \cdot r$ allora $prob(h(o_i) = h(o_j)) \leq P_2$, ovvero, qualsiasi coppia di oggetti che disti almeno c volte r (quindi sufficientemente dissimile) deve avere probabilità di collisione al massimo P_2.

Una famiglia LSH $(r, c \cdot r, P_1, P_2)$-sensitive definisce quindi delle probabilità minime e massime di collisione relativamente ad un certo valore r che determina la sensibilità della famiglia stessa. Si noti che queste proprietà devono valere per ogni h della famiglia.

8.2.3 Dalla famiglia alla funzione

Ora che disponiamo della nozione di famiglia LSH, supponiamo di averne una e capiamo come usarla. Capiremo in seguito come generare una famiglia LSH. Consideriamo infatti una famiglia $(r, c \cdot r, P_1, P_2)$-sensitive. Dopodiché generiamo L funzioni di hash composte (chiamate $g()$), ognuna nella forma:

$$g_j(o) = (h_{1,j}(o), ..., h_{k,j}(o))$$

Abbiamo quindi L funzioni di hash composite, ognuna delle quale coinvolge k funzioni di hash, tutte facenti parte della stessa famiglia. Tali funzioni sono scelte in maniera indipendente e randomica da H. Ovviamente le funzioni sono tutte applicate sullo stesso oggetto o.
Detto così il procedimento può risultare poco chiaro, vediamo quindi un piccolo esempio di ciò che stiamo dicendo. Prendiamo una funzione di hash h_1 e andiamo a mappare il nostro spazio:

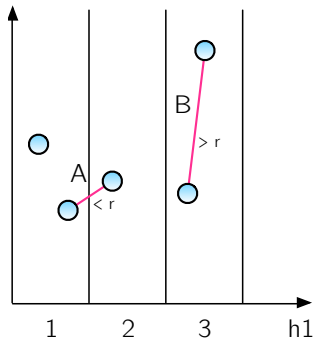


Figura 8.12: I quattro elementi sono mappati in tre bucket.

Grazie alla funzione h_1, abbiamo due oggetti mappati sul valore 1, uno sul valore 2 e due sul valore 3. Notiamo però due problematiche:

- Nel caso A la distanza fra i due punti è minore di r, ma appartengono a due bucket diversi.
- Nel caso B la distanza fra i due punti è maggiore di r, ma appartengono allo stesso bucket.

Ecco quindi perché decidiamo di applicare k funzioni di hash differenti in maniera composita (in questo caso $k = 2$):

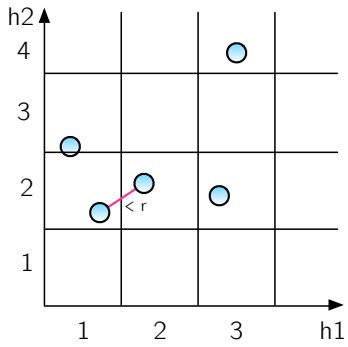


Figura 8.13: Sono applicate due funzioni di hash in maniera composita.

Come vediamo il problema riscontrato nel caso B è stato risolto (nel senso che oggetti distanti si trovano in bucket diversi) ma permane la problematica riscontrata nel caso A. Ecco quindi che capiamo l'uso di L differenti funzioni di hash composite, che danno questo effetto (anche in questo caso $L = 2$):

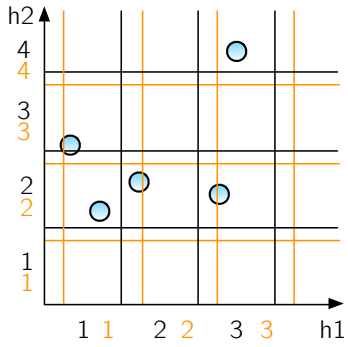


Figura 8.14: Sono applicate due funzioni di hash composite.

Ora, usando due funzioni composite otteniamo una sorta di disallineamento tale per cui riusciamo ad includere nello stesso bucket i due oggetti distanti meno di r. Sarà infatti molto bassa la probabilità che se due oggetti distano fra loro un quantitativo minore di r, saranno in due bucket diversi in tutte le L funzioni di hash.

Abbiamo quindi capito che:

- Al crescere di k ridurremo il numero di false hit.
- Al crescere di L ridurremo il numero di miss.

8.2.4 Usare la funzione in una query

Perciò, all'arrivo di una certa query q, si procederà con il mapping su tutte le L funzioni di hash composte (da k funzioni di hash della famiglia), trovando così un certo valore che identificherà un bucket: si riterrà il contenuto di quel bucket (ed eventualmente il suo intorno) il risultato della query.

8.2.5 Scegliere L e k

È evidente che il dimensionamento di L e di k sia in grado di cambiare i costi computazionali e la precisione del risultato. Ci può essere utile questo risultato (che dimostriamo a breve); se:

$$L = log_{1-P_1^k}\delta$$

allora qualsiasi oggetto nel range r sarà restituito con una probabilità di almeno $1-\delta$. Capiamo da dove arriva questo risultato e poi ne capiremo l'utilità.

La probabilità di collisione su uno stesso valore per una certa funzione $g_i()$ (presa fra le L) è pari alla moltiplicazione fra le proprietà di collisione di ogni funzione che compone $g_i()$ stessa, dunque P_1 moltiplicato k volte (ovvero P_1^k). Un oggetto non è restituito se nessuna delle L funzioni composite collide sullo stesso valore, e affinché una $g_i()$ non collida è sufficente che una delle sue funzioni interne collida, ovvero $(1-P_1^k)$. Di queste funzioni ne abbiamo L dunque si ha $(1-P_1^k)^L$. Se chiamiamo tale probabilità δ (che è quindi la probabilità di non avere alcuna collisione e quindi non essere un oggetto restituito) allora possiamo facilmente ottenere:

$$L = log_{1-P_1^k}\delta$$

Quindi, $1-\delta$ è la probabilità di essere restituiti. Capiamo quindi che una volta scelto un δ, sarà sufficente giocare con k ed L per ottenere il risultato voluto.

8.2.6 Costruire una famiglia LSH

Tutto il nostro discorso è molto interessante ma ci siamo dimenticati di un passaggio fondamentale: come otteniamo questa tanto anelata famiglia LSH? Quanto detto fin ora si basava sul fatto di disporre di una di queste famiglie così da poter ottenere le L funzioni composte.
Chiaramente la definizione della famiglia è strettamente correlata alla funzione di similarità $sim()$. Come anticipato, vedremo due metodologie secondo cui creare una famiglia LSH: nel primo caso useremo la distanza di Hamming come funzione di similarità, nel secondo caso useremo invece $L1$.

Similarità come distanza di Hamming

Assumiamo di avere oggetti descritti da vettori binari d-dimensionali, quindi ad esempio $(0, 1, 1, 0, ..., 1)$. Giacché la nostra funzione di somiglianza $sim()$ è la distanza di Hamming, allora la nostra famiglia H conterrà tutte le proiezioni dei punti di input x per ognuna delle coordinate del vettore. In altre parole abbiamo che $h_i(x) = x_i$ (e quindi la cardinalità di H sarà pari a d). In questo modo le funzioni di hash contenute in H non faranno altro che proiettare i vari valori binari del vettore, quindi la:

$$prob(h(q) = h(p))$$

è uguale alla frazione di bit concordi fra q e p. Per questa ragione definiremo:

- $P_1 = 1 - (r/d)$
- $P_2 = 1 - c(r/d)$ (con $c > 1$)

così da garantire $P_1 > P_2$. Volendo fare un esempio, se abbiamo $r/d = 20\%$ (ovvero la soglia di tolleranza è il 20% di valori differenti), allora avremo probabilità di collisione P_1 pari all'80%.

Similarità come distanza di Manhattan ($L1$)

La tecnica che sfrutta $L1$ l'abbiamo già descritta nei nostri esempi che facevano uso della griglia: si sceglie una quantita w superiore ad r, che sarà la nostra finestra (la larghezza di una cella). Dopodiché si prende una serie di valori di shift $s_1, s_2, s_3, ..., s_d$ tutti strettamente maggiori di 0 e minori di w. A questo punto possiamo definire una h per ogni s scelto, ognuna del tipo:

$$h_{s_i}(x) = (x - s_i)/w$$

Abbiamo quindi fornito le basi numeriche per ottenere quanto già visto nella Sezione 8.2.3. Si noti che gli assi (le funzioni di hash) non devono perforza essere ortogonali fra loro (tipicamente non lo sono): nel nostro esempio li abbiamo considerati ortogonali per meri fini grafici.

8.3 Il Web

Il web è uno dei più importanti fenomeni che si siano sviluppati negli ultimi anni. Il documenti del web potrebbero essere usati in maniera naïve, tramite l'impiego dell'information retrieval (quindi con $tf - idf$): con questo approccio andremmo però a perdere quelle che sono le informazioni intrinseche alla rete stessa, ovvero come sono collegati fra loro i documenti (in sostanza i link). Studiare i link può fornire notevoli informazioni latenti all'interno della rete. È compito della scienza delle Reti Complesse andare a scovare questi legami

(ad esempio identificando comunità, pattern ricorrenti, ecc.). In questo corso ci limiteremo ad accennare ad alcuni algoritmi che sfruttano i link per fornirci risultati pratici in merito a singole problematiche.

8.3.1 Link analysis

A seconda della tipologia dei link, possiamo distinguere diverse relazioni fra documenti:

- Due documenti sono connessi se si linkano reciprocamente.
- Due documenti sono co-citati se sono linkati da uno stesso documento terzo.
- Abbiamo un fenomeno di social filtering quando due documenti linkano ad un documento terzo.
- Abbiamo un fenomeno di transizione quando un documento linka ad un documento che ne linka a sua volta un terzo.

8.3.2 L'algoritmo HITS

L'algoritmo HITS è molto usato per stabilire l''importanza'' di un certo sito web. Potremmo quindi usare l'IR per trovare un primo insieme di documenti correlati alla ricerca dell'utente, dopodiché usare HITS per trovare i siti più rilevanti collegati a quelli trovati tramite l'IR.

Cosa intendiamo per ''importante''? Intuitivamente un buon sito è un sito che è suggerito da *esperti* di cui ci fidiamo (concetto di *trust*). Pensiamo dunque ad un nodo esperto: un suo link ad un sito significherebbe un voto di positività a quest'ultimo.
Ogni nodo è quindi caratterizzato da due valori:

- *Hub*: valuta la qualità dell'esperto, cioè il numero totale di voti che ha assegnato. Ottimi esempi di hub sono liste di giornali, guide per lo studente, slashdot, ecc.
- *Authority*: valuta la qualità del contenuto di un certo sito (la quantità di voti ricevuti dagli esperti). Ottime autorità potrebbero essere home page di quotidiani, home pagine di corsi, ecc.

La misura è ottenibile sia in maniera *iterativa* che in maniera *matriciale*.

Definizione matematica

Definiamo con a_i il valore di authority di un nodo i e con h_i il suo valore di hub. Inizializziamo $a_j = 1, h_i = 1$.
Dobbiamo dunque iterare (con j i nodi che puntano ad i):

$$\forall i : a_i = \sum_j A_{ji} h_j$$

$$\forall j : h_i = \sum_j A_{ij} a_j$$

$$\forall i : \sum_i a_i = 1, \sum_j h_j = 1$$

Abbiamo fatto uso della matrice A, detta matrice di adiacenza (l'elemento $a_{i,j}$ è pari ad 1 se esiste un arco che va dal nodo i verso il nodo j, zero altrimenti).

Definizione matriciale

In termini di matrici possiamo definire due vettori $a = h = 1^n$ (elevato a n poiché è il valore di n per quanti nodi ci sono con archi entranti). Quindi possiamo ripetere:

$$\vec{h} = A\vec{a}$$

$$\vec{a} = A^T \vec{h}$$

e dunque normalizzare.
Pertanto $\vec{a} = A^T A\vec{a}$ e $\vec{h} = AA^T \vec{h}$. Dopo k passi del ciclo avremo: $\vec{a} = (A^T A)^k \vec{a}$ e $\vec{h} = (AA^T)^k \vec{h}$.

Autovalori ed autovettori

Dato un vettore x e una matrice M, sia $Mx = \lambda x$ per qualche scalare λ. Abbiamo che x è un *autovettore* e λ è il suo *autovalore*.
Pertanto osserviamo che se la matrice M è simmetrica, (cioè $A^T A$ e AA^T sono simmetriche) allora M ha n autovettori ortogonali $w_1, w_2, ..., w_n$ che formano una base (un sistema di coordinate) con autovalori $\lambda_1, \lambda_2, ..., \lambda_n$ dove vale sempre che $\mid \lambda_i \mid \geq \mid \lambda_{i+1} \mid$.

Riprendendo quindi le due equivalenze:

$$\vec{a} = A^T A\vec{a}$$

$$\vec{h} = AA^T \vec{h}$$

Abbiamo che il vettore a delle autorità corrisponde all'autovettore di $A^T A$, associato all'autovalore maggiore λ_1. Analogamente accade con h e AA^T.

HITS ed LSI

Notiamo una interessante somiglianza fra quanto appena detto in merito all'algoritmo HITS. Se infatti consideriamo la singular valued decomposition (SVD, già studiata nella Sezione 5.6.12) della matrice A, otteniamo:

$$A = UMV^T$$

dove, come sappiamo:

- M è una matrice diagonale.
- U e V sono ortonormali.
- $U^T U$ e $V^T V$ sono la matrice identità.

A questo punto, la moltiplicazione vista prima (in merito all'array h), è scrivibile come:

$$AA^T = UMV^T VMU^T$$

La trasposta di A è infatti V^T a sua volta trasposta (quindi V), moltiplicata per M (che essendo diagonale rimane M), ed infine moltiplicata per U^T. Ma per le proprietà elencate sopra ($VV^T = I$), abbiamo:

$$AA^T = UM^2U^T$$

Se poi moltiplichiamo entambri i lati per U otteniamo:

$$AA^T U = UM^2U^T U$$

che, sempre per le condizioni espresse prima equivale a:

$$AA^T U = UM^2$$

Abbiamo ottenuto nuovamente la definizione di autovettore per una certa riga U_j che avrà m_j come autovalore, in altre parole: $\vec{h} = U_j m_j^2$. Infatti, applicando la sostituzione otteniamo la definizione già vista:

$$AA^T \vec{h} = \vec{h}$$

Quindi LSI e HITS si assomigliano, solo che il primo è basato sull'associazione termine-documento, il secondo sull'associazione sorgente-destinazione.

8.3.3 Difetti di HITS e correttivo

Il grande vantaggio di HITS è il fatto che calcolare i due autovettori di autority e hub è molto semplice (si può fare iterativamente oppure con metodi numerici). I due vettori verranno ri-calcolati ogni tanto, al fine di avere informazioni sempre aggiornate.

HITS però, nella sua semplicità, pecca di superficialità: solamente perché un vicino è un buon hub/autority, non è detto che sia necessariamente pertinente rispetto a quanto cercato dall'utente. Un sistema più intelligente è quello esposto di seguito.

Come già fatto in precedenza, si sfrutta l'IR per trovare le pagine simili a quanto espresso dall'utente. Da questo insieme primordiale, andiamo a trovare tutte le pagine linkate e per ognuna di queste computiamo i vettori $\vec{a}$ e $\vec{h}$: questa volta però la computazione coinvolge un correttivo. Tale correttivo consta nell'andare ad esaminare il *contesto in cui il link è situato*: si fa quindi una analisi testuale intorno al link al fine di capire l'effettiva pertinenza della pagina linkata (potrebbe ad esempio esssere soltanto una pubblicità). La computazione dei due vettori di hub ed autority dovrà quindi essere compiuta in maniera iterativa (e query dependent):

$$a_i = \sum_{j \in in(i)} w(j \to i) h_j$$

$$h_i = \sum_{j \in out(i)} w(i \to j) a_j$$

Abbiamo introdotto il peso w che è calcolato, come detto sopra, esaminando la porzione di testo in cui è immerso il link.

Ricapitolando, abbiamo risolto il problema del *topic drift* (recuperare vicini di pagine interessanti che però non sono rilevanti in merito alla query dell'utente), ma a grande costo computazionale: l'iterazione non ci piace, inoltre dobbiamo computare i due autovettori per ogni query (la metodologia è query dependent).

8.3.4 L'algoritmo PageRank

L'algoritmo *PageRank* è il celeberrimo algoritmo adottato da Google per attribuire importanza ai vari documenti del web (al fine di avere un ranking quanto più soddisfacente possibile). L'algoritmo di PageRank si basa sul concetto di *random surfer*: una persona naviga il web muovendosi di pagina in pagina (all'interno di una pagina ogni link è selezionabile con probabilità uniforme). Tante più volte un utente si troverà su una pagina, tanto più quella pagina sarà popolare (e quindi importante). Si tratta quindi di un algoritmo probabilistico, che cerca di individuare la popolarità delle pagine sfruttando i random walk di un utente casuale.
È anche contemplata una probabilità detta di *teleportation*, pari a $1 - \beta$, che rappresenta la probabilità di muoversi in una pagina casuale del web (non necessariamente linkata dalla pagina in cui si trova il random surfer). Nel caso in cui il random surfer finisca in una pagina priva di link, si muoverà su una

pagina a caso del web (ogni pagina ha uguale probabilità di essere scelta).

La formula che deriva da quanto appena descritto otteniamo una matrice di transizione (una matrice dove ogni elemento rappresenta la probabilità di transizione dallo stato rappresentato dalla riga i verso lo stato rappresentato dalla riga j):

$$\mathbf{Z} = (1-\beta)\left[\frac{1}{N}\right]_{N\times N} + \beta\mathbf{M}$$

dove con $\left[\frac{1}{N}\right]_{N\times N}$ intendiamo una matrice $N \times N$ dove ogni elemento è pari a $\frac{1}{N}$ e dove N è il numero totale di documenti mantenuti nel web.

La matrice $\mathbf{M}$ è invece definita come:

$$M_{ji} = \begin{cases} \frac{1}{\mid out(i) \mid} & \text{se c'è un arco fra } i \text{ e } j \\ 0 & \text{altrimenti} \end{cases}$$

ovvero ogni elemento $_{ji}$ rappresenta la probabilità di finire sulla pagina j considerando di trovarsi sulla pagina i (e quindi la probabilità è 1 sul totale dei link della pagina i-esima qualora j sia fra le pagine linkate da i, zero altrimenti). A questi due elementi sono state assegnate le probabilità β (di percorrere uno dei link della pagina sulla quale ci si trova) e la probabilità $1-\beta$ di saltare su una pagina random (con probabilità $1/N$). Google ha impostato un valore di β pari a 0.85. Abbiamo quindi che la singola probabilità di trovarsi su una pagina j sia:

$$P(j) = \frac{1-\beta}{N} + \beta \sum_{i \in in(j)} \frac{P(i)}{out(i)}$$

dove i è ogni pagina che linka la pagina j-esima.

Ciò che si ottiene è che l'autovettore principale (associato all'autovalore $\lambda = 1$) della matrice di transizione $\mathbf{Z}$ contiene nell'elemento j-esimo la probabilità $P(j)$. Tale autovettore è quindi l'autovettore che contiene tutti i valori di probabilità che ci servono, ovvero, il ranking delle pagine del web.

8.3.5 PageRank e dipendenza dalla query

L'algoritmo di PageRank fornisce un ordinamento delle pagine ottenuto esaminando la struttura del web. Ma ancora una volta, vorremo cercare di sposare l'aspetto strutturale con quello semantico (in sostanza lo score dei documenti ottenuto grazie alla query dell'utente).
Il problema del *topic drift* torna quindi a farsi sentire: forse dovremmo eseguire

l'algoritmo di PageRank solamente *dopo* aver identificato un insieme di pagine rilevanti per l'utente? Cerchiamo di dare risposta a questa domanda e soprattutto di trovare una soluzione pratica al problema (cerchiamo un correttivo proprio come già fatto per HITS).

Quali informazioni abbiamo?

Una volta definito un insieme di documenti rilevanti rispetto alla query (che chiameremo *seed*), possiamo considerare le altre pagine nelle vicinanze. Dobbiamo interrogarci in merito a tre diverse questioni:

- Quanto è correlata una certa pagina rispetto ai seed?
- Qual è la distanza fra una pagina ed i seed?
- Quanti path ci sono fra una pagina ed i seed?

Si tratta quindi di coinvolgere tre aspetti: *contenuto*, *distanza* e *connettività*.

Misura esplicita di rappresentatività. Potremmo considerare una formula di questo tipo:

$$rep(v) = \sum_{p \in paths(A,B,v)} \frac{score(p)}{length(p)}$$

Si ha che la rappresentatività di una certa pagina v è data dalla sommatoria (per ogni path p passante per la pagina v e che collega due seed A e B) del punteggio attribuito a quel path, diviso per la lunghezza del path stesso (al fine di normalizzare la misurazione). Lo score di un certo path è computabile come la somma degli score di tutte le pagine v' incontrate sul path.

In pratica si ha che una pagina è tanto più rilevante quanto sono rilevanti le pagine che si trovano su un path che unisce due seed e anche la pagina considerata attualmente. Le problematiche riscontrabili in un approccio così esplicito sono sia computazionali (il caso peggiore è esponenziale) che strutturali: la lunghezza dei path aumenta linearmente, mentre il numero di path aumenta esponenzialmente. Questo significa che la normalizzazione non può funzionare più di tanto e a cammini lunghi si attribuirà sempre un punteggio maggiore.

Una misura implicita.

Come unire quindi le tre informazioni di cui abbiamo parlato prima in maniera implicita? Il nostro problema è formalizzabile come segue.
Dati:

- Un insieme di pagine seed $S = \{s_1, ..., s_n\}$.
- Il web rappresentato da un grafo diretto $G(V, E)$.

- Un grafo connesso non diretto che mantiene i seed ed i suoi vicini N.

vogliamo trovare R, un insieme di pagine che meglio rifletta l'associazione fra tutte le pagine in S.

Primo passo: rilevanza e distanza. Iniziamo con il coinvolgere la distanza e la rilevanza delle pagine, ci occuperemo in seguito della connettività. Ad ogni vertice del grafo viene assegnata una *penalità*, direttamente dipendente dai due parametri appena citati. Per quanto concerne la distanza possiamo calcolarla in un paio di modi diversi:

1. Il valore di distanza di un certo nodo è la sommatoria di tutti i cammini minimi da quel nodo verso ogni seed.

2. Il valore di distanza di un certo nodo è dato dallo steiner tree minimo (il cammino minimo passante per tutti i seed più ulteriori nodi qualora necessari).

Si predilige la seconda tecnica, onde evitare di overstimare le distanze (cosa che accade usando la prima tecnica, più semplice).

Per quanto concerne la rilevanza possiamo usare una delle tecniche che abbiamo studiato durante questo corso, e possiamo basarci su un due approcci:

1. Content-focused (la rilevanza è calcolata di nodo in nodo basandosi soltanto sulla query).

2. Content-sensitive (la rilevanza è calcolata basandosi sui seeds e non più sulla query).

Una volta ottenute le due misure di rilevanza e di distanza, possiamo unirle usando una di queste formule (anche qui, a discrezione del progettista del database):

$$penality_1(v) = \frac{distance(v, S)}{relevance(v, S) + 1}$$

$$penality_2(v) = \frac{distance(v, S)}{relevance(v, S)}$$

$$penality_3(v) = distance(v, S) \times (2 - relevance(v, S))$$

Secondo passo: la connettività. Aggiungiamo ora l'elemento relativo alla connettività. Introduciamo prima la nozione di *random walk* su un grafo e di *catena di Markov*.

Una catena di Markov è fondamentalmente un automa probabilistico, dove la probabilità di transizione fra uno stato ed un altro è indipendente rispetto alla

storia (ovvero non importa quali altri stati si siano visitati in precedenza ma la probabilità di transizione dipende solo dallo stato attuale). Notiamo che questa nozione è rispettata nell'algoritmo di PageRank (la probabilità dipende solo dal nodo *i* che linka *j* e non dai precedenti). Il random walk su un grafo è quindi una catena di Markov il cui stato in un qualsiasi momento è descritto da un vertice del grafo stesso e la probabilità di transizione verso un altro stato è equalmente distribuita su tutti i nodi collegati allo stato attuale.

Il nostro intento è quello di usare le penalità appena affibbiate ad ogni nodo per modificare le probabilità di transizione (che, come appena detto, in un random walk sono uniformi) e quindi in un certo senso ''guidare il random walk''. Vedremo a breve come ottenere i pesi relativi.

Ora l'algoritmo di PageRank non si muoverà più in maniera del tutto casuale, bensì sarà mediato dalla presenza dei pesi sugli archi (che rappresentano la probabilità di transizione, calcolata a partire dalle penalità ovvero dai concetti di rilevanza e distanza): abbiamo unito tutti gli aspetti che ci interessavano!

Osserviamo questo esempio:

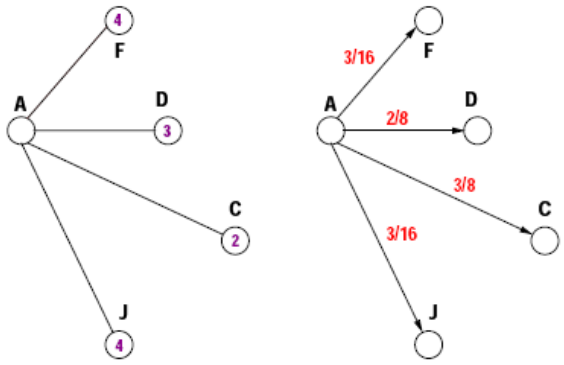


Figura 8.15: Dalle penalità si è passati alle probabilità di transizione sugli archi.

La distribuzione dei pesi avviene in maniera immediata, sfruttando un sistema lineare:

$$\begin{aligned} \mathcal{T}[F,A] + \mathcal{T}[D,A] + \mathcal{T}[C,A] + \mathcal{T}[J,A] &= 1.0; \\ 4 \times \mathcal{T}[F,A] = 3 * \mathcal{T}[D,A] \;\; ; &\quad 3 \times \mathcal{T}[D,A] = 2 * \mathcal{T}[C,A]; \\ 2 \times \mathcal{T}[C,A] = 4 * \mathcal{T}[J,A] \;\; ; &\quad 4 \times \mathcal{T}[J,A] = 4 * \mathcal{T}[F,A]; \\ 4 \times \mathcal{T}[F,A] = 2 * \mathcal{T}[C,A] \;\; ; &\quad 4 \times \mathcal{T}[J,A] = 3 * \mathcal{T}[D,A]; \end{aligned}$$

Figura 8.16: Sistema lineare che ci fornisce i pesi da attribuire agli archi.

La somma delle probabilità attribuite agli archi uscenti da un nodo deve essere uno (e questo spiega la prima equazione). Ciascuna coppia di nodi, poi, impone

dei vincoli sugli archi. La seconda equazione ad esempio rappresenta la coppia F e D.

8.3.6 Cenni: PPR (Personal PageRank)

L'algoritmo di PageRank è stato migliorato nel tempo considerando ulteriori aspetti. Nel caso del PPR, ci si è concentrati sul fatto che quando si effettua un salto random (e ciò accade con probabilità $1-\beta$), raramente si salta effettivamente su una pagina casuale. Se infatti stiamo leggendo la Stampa, difficilmente andremo a finire (con un salto casuale) su una rivista medica (molto specializzata). Si va quindi a modificare la componente di teleportation dell'algoritmo di PageRank: ci sarà probabilità $1-\beta$ di finire in una pagina appartenente all'insieme seed e non più una pagina casuale del web.

Si noti che questo cambiamento è profondamente diverso rispetto a quanto abbiamo fatto con il correttivo per il random walk: prima modificavamo le probabilità di transire da un nodo ad un altro, mentre qui modifichiamo effettivamente il cammin

8.4 Serie Temporali

L'ultimo degli argomenti che tratteremo sono le serie temporali. Abbiamo già detto che una serie temporale è un tipo di dato particolare: abbiamo un rilevamento di un certo valore ogni t secondi, così da ottenere un grafico che rappresenta l'andamento del fenomeno che stiamo rilevando nel tempo.
Le serie temporali sono molto utilizzate (più di quanto si creda): due esempi su tutti sono il campionamento di file musicali e le analisi di tipo finanziario (andamento della borsa).

Il nostro intento è quello di confrontare due serie temporali al fine di interpretarne la somiglianza (ed eventualmente capire se rappresentano lo stesso fenomeno o fenomeni simili).

8.4.1 Confrontare due serie: Dynamic Time Warping (DTW)

Il modo più intuitivo di confrontare due serie temporali è quello di prendere ogni punto delle due serie e di confrontarlo: sommando le distanze fra ogni coppia di punti, otteniamo la distanza totale fra le due serie temporali.
È evidente che questo approccio sia troppo semplicistico: non permettiamo alcun errore e le due serie devono essere della stessa lunghezza.

Ci viene in aiuto una soluzione molto più astuta: il *Dynamic Time Warping* (in breve DTW). Come il nome stesso suggerisce, l'idea è quella di modificare il tempo (restringendolo e allargandolo) per far sì che serie uguali a meno di

una certa imprecisione vengano comunque considerate simili. Prendiamo ad esempio queste due serie temporali (una rossa e una blu): Come vediamo, il

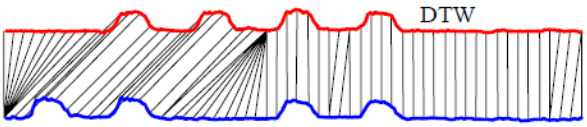


Figura 8.17: Due serie temporali vengono confrontate tramite DTW.

primo punto della serie blu è mappato su molti punti della serie rossa. Abbiamo anche casi di più punti rossi mappati verso uno stesso punto della serie blu. Non possiamo però avere punti mappati su punti precedenti (cioè tracciare linee in senso inverso: il warping non può violare l'ordine temporale degli eventi). Questo approccio ci permette di ignorare tutti i problemi derivanti da rilevazioni ritardate o prese in tempi differenti.
Ciò che si andrà a fare sarà quindi un passaggio di DTW al fine di rendere le serie più simili, dopodiché potremo calcolare le distanze fra i punti (correttamente mappati grazie al DTW).

8.4.2 DTW: la definizione formale

Andiamo ora a dare una definizione più formale di quanto abbiamo appena visto (e soprattutto capiamo come farlo in maniera automatica). Supponiamo di avere due serie temporali:

- $X = (x_1, x_2, ...x_N)$.
- $Y = (y_1, y_2, ..., y_M)$.

dove x_i e y_i appartengono allo stesso dominio D. Chiamiamo poi $\Delta()$ una funzione che misura la distanza fra due punti all'interno del dominio D. Tipicamente i valori di x_i e y_i saranno numerici (nel dominio dei numeri reali o naturali), ma potremmo anche avere delle parole come elementi del dominio.

Un *allineamento* da X verso Y è definito come un particolare cammino (una serie di mosse) detto *warp path* $W = (w_1, w_2, ..., w_K)$. Ogni elemento w_i è una coppia di punti (uno proveniente da una serie e l'altro proviente dall'altra). Inoltre, valgono le seguenti condizioni:

- $\max(N, M) \leq K \leq N+M$, ovvero il warp path è lungo almeno $\max(N, M)$ passi ed al massimo $N + M$.
- $w_1 = (1, 1)$ cioè il passo iniziale è il primo punto delle due serie.
- $w_K = (N, M)$ cioè il passo finale coincide con il finale delle due serie.

- $w_i - w_{i-1} \in \{(1,0),(0,1),(1,1)\}$, cioè la differenza fra un punto ed il suo seguente può rappresentare o l'incremento di posizione sulla una delle due serie temporali e non sull'altra (i primi due casi) oppure un incremento di punti per entrambe le serie temporali.

Nell'esempio di prima, quindi, avremo qualcosa del genere:

$$W = (1,1),(2,1),(3,1),(4,1)...$$

Possiamo quindi definire la distanza totale fra due serie dato il loro warp path. Più precisamente, dato $W = (w_1, w_2, ..., w_K)$ fra le serie X ed Y avremo una distanza totale pari a:

$$\Delta(W) = \sum_{i=1}^{K} \Delta(x_{w_i[1]}, y_{w_i[2]})$$

ovvero la somma fra le distanze di ogni punto.

Allineamento ottimo

È definito *allineamento ottimo* il warp path fra le serie X ed Y che minimizza la distanza totale fra le serie stesse, ovvero il minimo $\Delta(W)$ fra la serie X e la serie Y. Ecco quindi il nostro nuovo intento: calcolare il DTW in maniera efficente. Introduciamo la notazione $\Delta DTW(X,Y)$ per indicare il minimo DTW fra X e Y. Si noti che la distanza DTW è simmetrica ma non è una metrica (non soddisfa la disuaglianza triangolare).

8.4.3 Calcolare l'allineamento ottimo

Un approccio esaustivo (calcolare tutti i DTW e poi scegliere il minimo) è ovviamente infattibile. Ricorriamo quindi ad un impianto algoritmico simile a quello già studiato per calcolare la distanza fra stringhe per fare approximate string matching (Sezione 5.3.5).

Introduciamo un paio di notazioni che ci saranno utili per scrivere la definizione ricorsiva del nostro algoritmo:

- $X(1:i)$ rappresenta i primi i valori della serie temporale X.
- $Y(1:j)$ rappresenta i primi j valore della serie temporale Y.
- $D(i,j)$ equivale a $\Delta DTW(X(1:i), Y(1:j))$, cioè il path minimo che coinvolge i primi i e j valori rispettivamente delle serie X e Y.

Ecco quindi che tramite la ricorsione possiamo scrivere:

$$D(i,j) = \min\{D(i-1,j), D(i,j-1), D(i-1,j-1)\} + \Delta(x_i, y_j) \qquad (8.1)$$

Cioè, un certo cammino minimo che coinvolga i passi della serie X e j passi della serie Y, è definito come il cammino minimo fino a giungere al passo precedente (che può essere raggiunto in tre modi, ma noi consideriamo quello di costo minimo) più il costo del nuovo passo (che viene aggiunto al cammino, ed è rappresentato da $\Delta(x_i, y_j)$).
È possibile computare questo calcolo tramite una matrice $N \times M$:

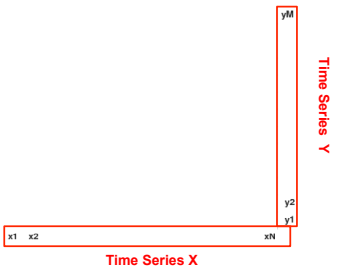


Figura 8.18: Matrice di dimensione $N \times M$.

Dopodiché, andremo a popolare ogni elemento (i, j) con il corrispondente valore $D(i, j)$. Sarà facile partire dall'angolo in basso a destra (che ha valore (0, 0)) e procedere fino a popolare tutto la matrice. Ogni singola cella sarà valorizzata tenendo conto della definizione ricorsiva espressa dall'Equazione 8.1.
Una volta popolata tutta la matrice, sarà sufficente ripercorrere il percorso al contrario scegliendo fra le caselle adiacenti quella con valore minimo, fino a giungere a (0,0) e a completare in questo modo il path. Vediamo in questa immagine uno sketch del processo:

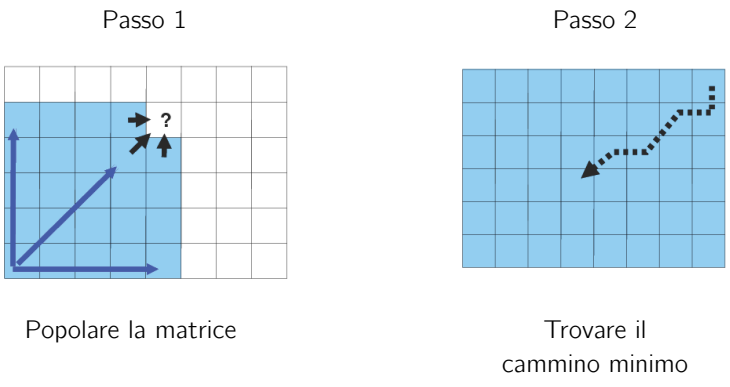


Figura 8.19: Calcolo dell'allineamento ottimo.

Il primo passo ha costo $O(NM)$, mentre il secondo ha costo $O(N + M)$. Abbiamo un netto miglioramento rispetto al caso esaustivo, ma quando trat-

tiamo serie temporali parliamo di dati molto numerosi e quindi tale complessità potrebbe ancora risultare pericolosa.

8.4.4 Migliorare la complessità

Esistono alcune migliorie per quanto concerne la complessità. Nella fattispecie, cerchiamo di evitare di calcolare elementi della matrice che sono poco interessanti: gli elementi molto distanti dalla diagonale principale indicano grandi differenze fra le due serie temporali (quindi un grande accorpamento o allungamento del tempo) e quindi possono essere tralasciati dal calcolo. Due delle migliorie proposte sono le seguenti:

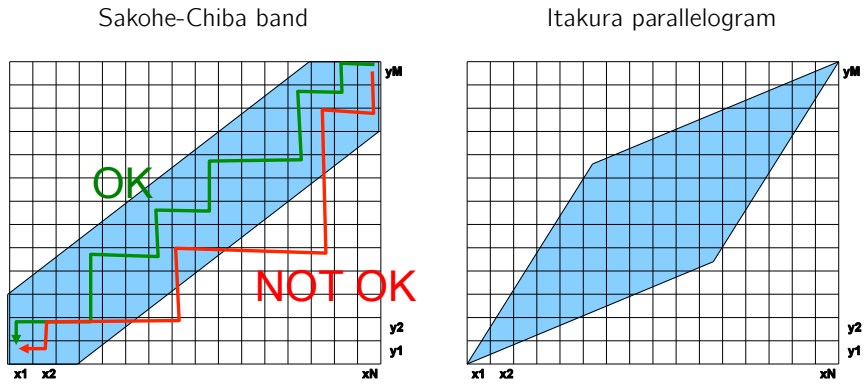


Figura 8.20: Le aree blu sono quelle popolate: c'è risparmio computazionale.

Il primo esempio (la Sakoe-Chiba band) illustra che un cammino minimo come quello verde verrebbe rilevato nonostante la miglioria computazionale, mentre un cammino minimo come quello rosso verrebbe perso. Itakura propone invece una forma ''a parallelogramma'' che quindi permette una maggiore distensione temporale nella zona centrale delle serie ma invece è più restrittiva sui limiti delle serie.

Bibliografia

[1] K. Seluk Candan and Maria Luisa Sapino. *Data Management for Multimedia Retrieval*. Cambridge University Press, New York, NY, USA, 2010.