Algoritmi non deterministici e classe NP

A.A. 2017-2018

Classi di complessità

Dopo aver parlato di sintesi e analisi degli algoritmi completando l'illustrazione delle principali tecniche di progettazione di un algoritmo, spostiamo l'attenzione sui problemi, studiando la possibilità di classificare questi ultimi in base alla quantità di risorse necessarie per ottenere la soluzione. Infatti per certi gruppi di problemi le difficoltà incontrate per trovare un algoritmo efficiente sono sostanzialmente le stesse.

Ciò ha portato a definire e studiare le cosiddette ***classi di complessità***, cioè classi di problemi risolubili con una certa quantità di risorse (tempo o spazio).
Lo scopo è anche quello di confrontare la difficoltà intrinseca dei vari problemi, verificando, per esempio, se un problema è più o meno facile di un altro o se è possibile trasformare un algoritmo per il primo in uno per il secondo che richieda all'incirca la stessa quantità di risorse.

Classi di complessità

Grossolanamente possiamo raggruppare i problemi in tre categorie:

- problemi che ammettono algoritmi di soluzione efficienti;
- problemi che per loro natura *non possono essere risolti mediante algoritmi efficienti* e che quindi sono *intrattabili*;
- problemi per i quali algoritmi efficienti *non sono mai stati* trovati, ma per i quali non è stato dimostrato che tali algoritmi non esistono
- problemi che *non possono* essere risolti in modo algoritmico

Molti problemi di notevole interesse appartengono al terzo gruppo e presentano caratteristiche così simili dal punto di vista algoritmico che risulta naturale introdurre metodi e tecniche che consentano di studiarne le proprietà complessivamente.

Tra le classi di complessità definite in base al tempo di calcolo sono di grande interesse le classi **P** e **NP** alle quali appartengono un gran numero di problemi di interesse pratico.

Algoritmi non deterministici

Prima di definire formalmente le classi **P** e **NP**, introduciamo il concetto di algoritmo non deterministico per la soluzione di problemi di decisione. Un problema si dice *di decisione* se ammette solo due possibili soluzioni (si, no). Esso può essere rappresentato da una coppia <I, q>, dove I è l'insieme delle istanze e q è un predicato su I, ovvero una funzione q: $I \rightarrow \{0, 1\}$

Per definire algoritmi *non deterministici* aggiungiamo al nostro linguaggio di disegno le tre istruzioni: *choice* (S), *failure* e *success*. L'istruzione *choice* (S), in prticolare, restituisce un elemento da un insieme S scelto *a caso*

Concettualmente, un algoritmo **non deterministico,** di fronte a una scelta:

- sceglie la strada giusta (se c'è)

o meglio

- segue tutte le alternative contemporaneamente generando più copie di se stesso

La risposta dell'algoritmo su un input *x* è "si" **se e solo se** esiste una sequenza di scelte che porta al successo.

Algoritmi non deterministici: commesso viaggiatore

Prima di tutto trasformiamo il problema del commesso viaggiatore in un problema di decisione

Dato un grafo G non orientato, completo e pesato, esiste un "ciclo semplice" (che tocca tutti i vertici) di costo non superiore a un costo prefissato ?

Siano 1, 2, …, n i vertici del grafo e il vertice 1 quello di partenza. Chiamiamo *W* la tabella delle distanze (cioè la matrice di adiacenza che memorizza il grafo), *k* il costo da non superare, entrambi forniti in input, e *C* il vettore che memorizza il percorso, se esiste; la variabile *costo* è usata per ricordare il costo del cammino.

Nella soluzione proposta l'insieme delle scelte varia, nel senso che ad ogni città visitata dal commesso viaggiatore tale città viene eliminata dall'insieme delle scelte che potranno essere fatte successivamente.

Algoritmi non deterministici: commesso viaggiatore

```
ND_commesso-viaggiatore (n, W, k)
    C[1] ← 1
    costo ← 0
    I_S ← {2, …, n}
    for i ← 2 to n
        C[i] ← choice (I_S)
        I_S ← I_S - {C[i]}
        costo ← costo + W[C[i-1],C[i]]
        if (costo > k) failure
    if (costo + W[C[n],C[1]] > k) failure
    success
```

Complessità: $\mathbf{O}(n)$ + costo aggiornamento I_S $\Longrightarrow$ $\mathbf{O}(n^2)$

Algoritmi non deterministici: cricca

Dato un grafo G non orientato, esiste in G una cricca di dimensione prefissata k, ossia esistono in G k vertici ogni coppia dei quali risulta adiacente ?

Nella soluzione proposta ad ogni vertice i viene associato un attributo S[i], variabile booleana che ricorda se il vertice è scelto per l'inserimento nella cricca che si sta cercando di trovare.

L'insieme delle scelte non è mai modificato in quanto per ogni vertice si ha sempre la scelta binaria tra l'inserimento o meno nella cricca.

Algoritmi non deterministici: cricca

```
ND_cricca (G = <V, E>, k)
{S vettore booleano di n componenti memorizza la cricca}
        h ← 0
        for i ← 1 to |V|
                S[i] ← choice ({true, false})
                if (S[i])
                        h ← h+1
                        for j ← 1 to i – 1
                                if (S[j] and (i,j) ∉ E)  failure
                        if (h = k) success
        failure
```

Complessità: $\mathbf{O}(V^2)$, se k = $\mathbf{O}(V)$

Algoritmi non deterministici: schema generale

Gli algoritmi non deterministici operano una sequenza di scelte. Ogni scelta avviene su un sottinsieme I_i che è funzione delle scelte precedenti. Supponendo che il numero delle scelte sia limitato da n, si può fornire uno **schema generale di procedura non deterministica**

```
non_deterministica (…)
        ……
        ……
        for i ← 1 to n
                S[i] ← choice (Ii)
                if (S[1]…S[i] “non può diventare una soluzione”)
                        failure
                if (S[1]…S[i] “è una soluzione”)  success
        failure (success)
```

Algoritmi non deterministici: certficare la soluzione

Gli algoritmi non deterministici che ci interessano devono avere anche una caratteristica importante: una volta fatte le scelte la soluzione risultante **deve essere «certificata» in modo efficiente** (polinomiale).

Negli esempi precedenti questo è sempre verificato. La risposta ‘’*success*‘’ viene sempre data solo dopo la verifica che le scelte fatte determinano la soluzione del problema.

La classe dei problemi di decisione per la cui soluzione esistono algoritmi *non deterministici* di *complessità polinomiale* nella dimensione dei dati in input viene detta classe **NP**.

Non determinismo vs. determinismo

In altre parole, se potessimo prendere la decisione giusta in ogni punto di decisione si potrebbe determinare velocemente se esiste una soluzione!

- se la soluzione trovata è valida, allora True
- se la soluzione trovata non è valida, allora False

Questo processo “magico” che può sempre fare la mossa giusta é chiamato “Oracolo”.

Gli algoritmi non deterministici che ci interessano danno una risposta per mezzo di una serie di “mosse corrette”, che poi verificano in modo efficiente . Quelli deterministici prendono decisioni provando ad una ad una tutte le possibili scelte.

Non determinismo vs. determinismo

Simulare il comportamento del programma non deterministico con uno deterministico

generare ed esplorare deterministicamente
lo spazio delle soluzioni

cioè

visitare sistematicamente l'albero delle scelte corrispondente alla computazione non deterministica

numero di foglie superpolinomiale

per cricca $\mathbf{O}(2^n)$

Non determinismo vs. determinismo

```
ND_commesso-viaggiatore (n, D, k)

    C[1] ← 1
    costo ← 0
    I_S ← {2, …, n}
    for i ← 2 to n
        C[i] ← choice (I_S)
        I_S ← I_S - {C[i]}
        costo ← costo + W[C[i-1],C[i]]
        if (costo > k) failure
    if (costo + W[C[n],C[1]] > k) failure
    success
```

```
EN_C-V (i, I_S, costo)
    if (i ≤ n)
        for j ← 1 to |I_S|
            C[i] ← I_S[j]
            I_S ← I_S - {C[i]}
            costo ← costo + W[C[i-1],C[i]]
            if (costo ≤ k)
                if (i = n and costo + W[C[n],C[1]] ≤ k)
                    return true
                if (EN_C_V (i+1, I_S, costo))
                    return true
            costo ← costo - W[C[i-1],C[i]]
            I_S ← I_S ∪ {C[i]}
        return false
    return false
```

Se non può diventare una soluzione, fallimento

Se può diventare una soluzione, continua

Prova un'altra scelta, ma rimedia ai side-effect

Non determinismo vs. determinismo

```
enumerazione (i, …)
    if (i <= n)

        "determina I_i in funzione delle scelte precedenti"
        if I_i ≠ Φ
                for j ← 1 to | I_i |
                        S[i] ← "j-esimo elemento di I_i"
                        if (S[1]…S[i-1] "può diventare una soluzione")
                            if  ("è una soluzione")
                                                return  true
                            if enumerazione (i+1 …)
                                                return  true
                    return  false

        else  return  false

    return  false (true)
```

Non determinismo vs. determinismo: cricca

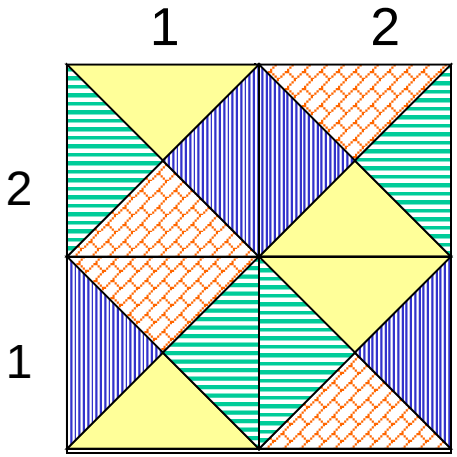
```
EN_cricca (i, h, k)
    if   (k-h <= n-i+1)   vertici disponibili
        S[i] ← false                                          {1^ possibilità}
        if (EN_cricca (i+1, h, k)) return true                {successo}
        S[i] ← true                                           {2^ possibilità}
        for j ← 1 to (i-1)
                if (S[j] and (i, j) ∉ E)                      {può diventare una
                                                               soluzione?}
                        return false
        if (h+1 = k) return true                              {se è una soluzione
        else return EN_cricca (i+1, h+1, k)                    successo}
    return false
```

Chiamata iniziale EN_cricca(1, 1, k)

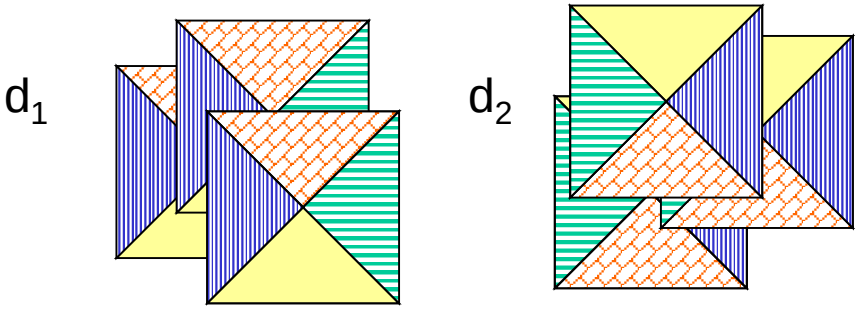
Non determinismo vs. determinismo: domino

Data una scacchiera n x n e tessere di m colori diversi,
è possibile ricoprire la scacchiera senza ruotare le tessere
in modo che
- i lati adiacenti delle tessere siano dello stesso colore
- una particolare tessera occupi la posizione più in basso a sinistra?

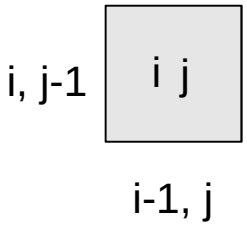
Esempio: Scacchiera 2 x 2



Tessere:



Non determinismo vs. determinismo: domino




```
ND_domino (n, m)
    S[1, 1] ← 1
    for i ← 1 to n do
        for j ← 1 to n do
            if i ≠ 1 or j ≠ 1
                S[i, j] ← choice({1,…, m})
                if (i = 1 and d[S[i, j-1]] ≠ s[S[i, j]])  failure
                if (i > 1 and j = 1 and a[S[i-1, j]] ≠ b[S[i, j]])
                        failure
                if (i > 1 and j > 1 and (d[S[i, j-1]] ≠ s[S[i, j]] or
                                or a[S[i-1, j]] ≠ b[S[i, j]]))
                        failure

    success
```

{soluzione parziale? No → fallimento}

Non determinismo vs. determinismo: domino

```
{ i ≤ n and j ≤ n and (i ≠1 or j ≠1)}
EN_domino (i, j)
    for k ← 1 to m do
        S[i, j] ← k
        if (i =1 and d[S[i, j-1]l =s[S[i, j]])
                if (j<n)        \\ riga aperta
                    if (EN_domino (i, j+1)) return true
                else if EN_domino (i+1, 1) return true
        if (i > 1 and j =1 and a[S[i-1, j]] =b[S[i, j]])
                if EN_domino (i, j+1) return true
        if (i > 1 and j > 1 and d[S[i, j-1]l = s[S[i, j]]
                                and a[S[i-1, j]] = b[S[i, j]]))
                if j< n) if (EN_domino (i, j+1)) return true
                if (i < n and j = n) if (EN_domino (i+1, j)) return true
                if (i =n and j =n) return true
    return false
```

→ {soluzione parziale?
Si chiamata ricorsiva}