Ricerca esaustiva e Backtracking

A.A. 2017-2018

Per applicare il backtracking, la soluzione deve essere esprimibile come una n-pla $S = (s_1, . . ., s_n)$, dove ogni s_i è scelto da un insieme finito I_i.
Spesso il problema da risolvere richiede di trovare un vettore che soddisfa una funzione criterio $P(s_1, . . ., s_n)$.

Se m_i è la cardinalità dell'insieme I_i, vi sono $m = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ n-ple candidate per soddisfare la funzione P.
L'approccio della ***ricerca esaustiva*** (forza bruta) forma tutte le n-ple e per ognuna si chiede se soddisfa il criterio.

Un algoritmo di *backtracking* ha invece la capacità di dare la stessa risposta con molti meno di m trials.

L'idea è quella di costruire il vettore componente per componente ed usare funzioni criterio modificate (***bounding functions***) per verificare se la soluzione parziale ha una chance di successo.

Molti problemi risolti con il backtracking richiedono che tutte le soluzioni soddisfino un insieme complesso di vincoli: ***espliciti*** e ***impliciti***.

Le tuple che soddisfano i vincoli espliciti definiscono un possibile ***spazio delle soluzioni*** per la particolare istanza: definiscono i valori che ogni s_i può assumere.
I vincoli impliciti determinano quali tuple nello spazio delle soluzioni soddisfano la funzione criterio: descrivono il modo in cui le s_i devono essere in relazione le une con le altre.

Gli algoritmi di backtracking ***determinano le soluzioni cercando sistematicamente nello spazio degli stati***, che contiene anche le soluzioni parziali, cioè le tuple $(s_1, . . ., s_i)$ per $i \leq n$.
La ricerca è facilitata usando una organizzazione ad albero per lo spazio delle soluzioni.

Ogni nodo nell'albero definisce uno ***stato del problema***.
Una volta che l'albero è stato concepito per un problema, il problema stesso può essere risolto generando in modo sistematico gli stati del problema e determinando quali sono stati soluzione.

L'efficienza di questi algoritmi dipende da quattro fattori:

- Il tempo per generare il prossimo valore per s_i
- Il numero di valori di s_i che soddisfano i vincoli espliciti
- *Il tempo per calcolare* le funzioni *bounding*
- Il *numero* di valori di s_i che soddisfano le funzioni bounding

Le funzioni bounding sono "buone" se riducono il numero di nodi generati. Di solito comunque funzioni buone richiedono più tempo per essere valutate.

Una volta determinata la struttura ad albero dello spazio degli stati, dei quattro fattori che determinano il tempo richiesto da un algoritmo di backtracking i primi tre sono relativamente indipendenti dall'istanza del problema. Solo *il numero di nodi generati varia da un'istanza ad un'altra*.

Un algoritmo di backtracking su un'istanza del problema può generare $O(n)$ nodi, mentre per una diversa istanza può generare tutti i nodi dell'albero dello spazio degli stati.

Se il numero di nodi dello spazio delle soluzioni è 2^n o $n!$, la complessità dell'algoritmo di backtracking sarà in generale $O(p(n) \cdot 2^n)$ o $O(q(n) \cdot n!)$ rispettivamente.

Riprendiamo l'algoritmo di visita e usiamolo per visitare tutti i cammini semplici di n vertici di un *grafo completo*, con vertici numerati 1, 2, …, n, a partire dal un vertice i_0 (*Visit* (G, i_0)).
(Ricordiamo che inizialmente h=0)

Visit (G, i)
$h \leftarrow h + 1$
X[i] $\leftarrow h$
<u>for</u> j $\leftarrow$ 1 <u>to</u> n
<u>if</u> ((i,j) $\in$ E)
<u>if</u> (X[j] = 0) *visit* (G, j)
$h \leftarrow h - 1$
X[i] $\leftarrow$ 0

Tenendo conto del fatto che il grafo è *completo* e che perciò è inutile verificare l'esistenza dell'arco, e sostituendo i valori di X[k] con *0,1* per indicare se il vertice è stato visitato sul cammino in esame o no, con piccole modifiche otteniamo la seguente versione dell'algoritmo:

Nota: asssumendo un nodo "dummy" 0 e partendo da *visit* (0) vengono visitati *tutti* i cammini semplici.

```
visit (i)
    for j ← 1 to n
        if (X[j] = 0)
            X[j] ← 1
            visit (j)
            X[j] ← 0
    return
```

Qual è il risultato della visita?

⬇

I vertici {1, 2, …, n} sono stati visitati in tutti gli ordini possibili

⬇

tutte le permutazioni

Backtracking: permutazioni

Si ottiene immediatamente il seguente algoritmo che memorizza nel vettore S le permutazioni di volta in volta trovate e le stampa:

```
visit (i)
    for j ← 1 to n
        if (X[j] = 0) then
            X[j] ← 1
            visit (j)
            X[j] ← 0
    return
```

```
Perm (k, n, S)
        if (k > n) "stampa S[1..n]"  return
          j ← 1
          while  j  ≤ n  do
           if (X[j] = 0) then
               X[j] ← 1
               S[k] ←  j
               Perm (k+1, n, S)
               X[j] ← 0
            j ← j+1
         return
```

X[i] indica se la posizione i è già occupata,
S[i] ne determina il valore

Backtracking: permutazioni

Perm è un algoritmo di ricerca esaustiva, ma è il problema che la richiede. La complessità è $\Omega(n!)$

Se vogliamo stampare le permutazioni in cui *i numeri dispari precedono i numeri pari* possiamo invece scrivere un algoritmo di backtracking che effettua le chiamate ricorsive solo quando la soluzione parziale può essere estesa ad una soluzione totale. In tal modo la complessità risulterà proporzionale al numero di permutazioni che devono essere generate.

Possiamo ottenere l'algoritmo di backtracking osservando che i numeri dispari devono occupare le prime $\lceil n/2 \rceil$ posizioni, i numeri pari le posizioni dalla $\lceil n/2 \rceil + 1$-esima alla n-esima e che quindi basta considerare nelle due parti solo numeri dispari o solo numeri pari.

```
Disp_Pari (k, n, S)
        if (k > n)  “stampa S[1..n]”
                    return
        if (k <= ⌈n/2⌉)  j ← 1
        else  j ← 2
        while (j <= n)
                if (X[j] = 0)
                        S[i] ←  j
                        X[j] ← 1
                        Disp_Pari (k+1, n, S)
                        X[j] ← 0
                j  ← j + 2
        return
```

Backtracking: sottinsiemi

Un altro problema che richiede la *ricerca esaustiva* è la generazione dei sottoinsiemi di un insieme $A = \{a_1, a_2, \dots, a_n\}$. Le soluzioni sono ricordate nel vettore binario X (X[i] = 1 sse l'elemento i-esimo dell'insieme A, a_i, fa parte del sottoinsieme.

```
Genera_Sottinsiemi (i, n, A, X)
        if (i > n) "stampa il sottinsieme"
                return
        X[i] ← 0
        Genera_Sottinsiemi (i+1, n, A, X)
        X[i] ← 1
        Genera_Sottinsiemi (i+1, n, A, X)
        return
```

L'algoritmo ha complessità $\Omega(2^n)$ (sono 2^n i sottinsiemi di A).

Backtracking: sottinsiemi

Dalla struttura di questo algoritmo si possono ottenere algoritmi di backtracking che richiedono di prendere in esame sottinsiemi di un insieme che presentano particolari caratteristiche.

Consideriamo ad esempio il problema Sottinsiemi vincolati: *dati due interi n e c, stampare tutti i sottinsiemi di* {*1, 2, …, n*} *di cardinalità uguale a c.*

La ricerca della soluzione può essere effettuata esaminando tutti i sottoinsiemi di {1, 2, …, n} in tempo $O(n \cdot 2^n)$ e verificando quali soddisfano la condizione *[ricerca esaustiva].*

numero di elementi

```
Sottinsiemi_vincolati (i, c, n, X)
    if (i > n)                    // calcola il numero di elementi richiesto
        num ← 0
        for k ← 1 to n
            if (X[k] =1) num ← num + 1
        if (num = c)  “stampa il sottinsieme”
        return
    X[i] ← 0
    Sottinsiemi_vincolati (i+1, c, n, X)
    X[i] ← 1
    Sottinsiemi_vincolati (i+1, c, n, X)
    return
```

tempo di esecuzione = $O(n \cdot 2^n)$

Gli insiemi da costruire sono $\binom{n}{c} = O(n^c)$ e, quanto più c è piccolo rispetto a n, tanto più l'algoritmo risulta inefficiente.

È possibile potare l'albero degli stati osservando che tutto il sottoalbero di un nodo relativo ad una soluzione parziale può essere eliminato nei due seguenti casi:

- nella soluzione parziale sono stati inseriti *più di "c" elementi*
- Il numero degli elementi della soluzione parziale più *il numero degli elementi non ancora considerati è minore di "c"* (*importante*).

Possiamo pertanto usare due funzioni *bounding*, una per visitare il sottoalbero destro di un nodo solo se escludere l'elemento i-esimo non preclude la possibilità di ottenere una soluzione, l'altra per visitare il sottoalbero sinistro solo se aggiungere l'elemento i-esimo non genera una soluzione parziale con più di "c" elementi.

Nu è il numero di ''1'' nella soluzione parziale X[1..i-1]

```
Sottinsiemi_vincolati (i, c, n, X, Nu)
    if (Nu = c) "stampa il sottinsieme"
                return
    if (Nu + (n – i) >= c)  // restano elementi sufficienti
    X[i] ← 0
        Sottinsiemi_vincolati (i+1, c, n, X, Nu)
    if (Nu < c)
        X[i] ← 1
        Sottinsiemi_vincolati (i+1, c, n, X, Nu + 1)
    return
```

Il costo in tempo di questo algoritmo, in cui un nodo dell'albero viene visitato se e solo se nel suo sottoalbero è presente almeno una soluzione, è $O(n \cdot n^c) = O(n^{c+1})$

Teorema

Sia h l'altezza dell'albero delle soluzioni, $\mathcal{O}(f(n))$ il tempo richiesto dalla visita di un suo nodo interno e $\mathcal{O}(g(n))$ il tempo richiesto dalla visita di una sua foglia.

Se nel corso della visita vengono visitati esclusivamente nodi nei cui sottoalberi sono presenti soluzioni, il tempo richiesto dalla visita e`:

$$\mathcal{O}(h \cdot f(n) \cdot D(n) + g(n) \cdot D(n))$$

Dove D(n) e` il numero di soluzioni.

Per l'esempio precedente si ha:

$h = n$, $D(n) = \mathcal{O}(n^c)$, $\mathcal{O}(f(n)) = \mathcal{O}(1)$, $\mathcal{O}(g(n)) = \mathcal{O}(n)$

E, applicando il teorema, ritroviamo la complessita`:

$$\mathcal{O}(n \cdot 1 \cdot n^c + n \cdot n^c) = \mathcal{O}(n \cdot n^c)$$

Backtracking: cicli Hamiltoniani

Un caso i cui non funziona….

```
Cicli-Hamiltoniani (S, k, X, n, G)
      if (k > n) if (S[n], S[1]) ∈ E)
                       "stampa S[1..n]"
             return
      for i ← 2 to n
         if (X[i] = 0 and (S[k-1], i) ∈ E)
             X[i] ← 1
             S[k] ← i
             Cicli-Hamiltoniani (S, k+1, X, n, G)
        X[i] ← 0
        return
```

La funzione bounding (il controllo che l'arco sia presente) non assicura che un nodo venga visitato solo se nel suo sottoalbero sono presenti soluzioni.