TECNICA *BACKTRACKING*

1. Introduzione

Il Backtracking è un modo sistematico di muoversi tra tutte le possibili configurazioni di uno spazio. Queste configurazioni ad esempio possono essere tutti i possibili arrangiamenti di un insieme di oggetti (permutazioni) o tutte le possibili sotto-collezioni di questi oggetti (sottoinsiemi). Altre applicazioni possono richiedere di enumerare tutti i cammini tra due nodi di un grafo o tutti gli alberi di copertura di un grafo etc. etc.
Ciò che questi problemi hanno in comune è che bisogna generare ciascuna delle possibili configurazioni *esattamente* una volta. Per evitare perdite o replicazioni di configurazioni bisogna definire un ordine sistematico di generazione tra le possibili configurazioni.
Una rappresentazione sufficientemente generale in grado di codificare gran parte degli oggetti combinatoriali di interesse è la seguente. Rappresentiamo le configurazioni con un vettore $SOL = (a_1, a_2, \ldots, a_n)$ dove ciascun elemento a_i va selezionato da un insieme ordinato S_i di possibili candidati per la posizione i. Stando così le cose, la procedura di ricerca lavora facendo crescere le soluzioni di un elemento alla volta. Ad ogni passo della ricerca avremo costruito una soluzione parziale con elementi fissati per i primi k elementi del vettore, dove $k \leq n$. Da questa soluzione parziale $(a_1, a_2, \ldots, a_k)$ costruiamo l'insieme S_{k+1} dei possibili candidati per la posizione $k+1$ e cerchiamo di estendere la soluzione parziale con un elemento di S_{k+1}. Finchè l'estensione genera una soluzione parziale continuiamo ad estenderla. Ad un certo punto S_{k+1} potrebbe essere vuoto, ciò significa che non c'è modo di estendere la corrente soluzione parziale. Quando ciò accade bisogna fare un *passo indietro* (vale a dire backtrack) e rimpiazzare l'ultimo elemento a_k nella soluzione con un altro candidato in S_k. E' proprio questo passo all'indietro che da il nome al metodo.
Il Backtracking costruisce un albero (*albero delle soluzioni*) dove ciascun vertice interno è una soluzione parziale e c'è un arco che va dal vertice x al vertice y se il vertice y è stato creato estendendo la soluzione parziale del vertice x. Le foglie dell'albero sono le soluzioni. L'albero delle soluzioni offre un modo alternativo di pensare al backtracking in quanto il processo di costruzione delle soluzioni corrisponde esattamente a quello di effettuare una visita in profondità dell'albero. Da questa visione del metodo come visita in profondità di un albero è del tutto naturale pensare, come schema del metodo, alla seguente procedura ricorsiva

```
BACKTRACKING(SOL, k)
    IF (a_1, ..., a_k) è una soluzione THEN stampala
    ELSE
    calcola S_{k+1}
    WHILE S_{k+1} ≠ ∅ DO
        a_{k+1} ← un elemento di S_{k+1}
        S_{k+1} ← S_{k+1} − {a_{k+1}}
        BACKTRACKING(SOL, k + 1)
    ENDWHILE
END
```

E' ovvio che per enumerare tutte le soluzioni si sarebbe potuta utilizzare anche una visita in ampiezza dell'albero delle soluzioni parziali, tuttavia la visita in profondità è di gran lunga da preferirsi per quanto riguarda il risparmio della risorsa spazio. Nella visita in profondità lo stato corrente della ricerca è rappresentato completamente dal cammino che va dalla radice allo stato corrente, il che richiede uno spazio proporzionale alla *profondità* dell'albero. Nella ricerca in ampiezza la coda memorizza tutti i nodi al livello corrente, il che richiede uno spazio proporzionale all'*ampiezza* dell'albero. Per la maggior parte dei problemi di interesse l'ampiezza dell'albero delle soluzioni parziali cresce esponenzialmente rispetto all'altezza.
Per comprendere come lavora il Backtrackig vedremo adesso come oggetti combinatori quali sottoinsiemi e permutazioni vengono costruiti dopo aver definito il loro corretto spazio di ricerca.

GENERARE TUTTI I SOTTOINSIEMI. Dati n oggetti, ci sono 2^n differenti sottoinsiemi di questi oggetti. Per rappresentare un sottoinsieme degli n oggetti possiamo utilizzare il vettore caratteristico di n elementi dove nella cella i-esima troviamo il valore 0 se l'oggetto i-esimo non appartiene al sottoinsieme, il valore 1 altrimenti. Con questa rappresentazione

l'insieme S_k dei candidati per la k-ma posizione è dato dai due elementi $\{0, 1\}$. Lo pseudocodice della procedura è il seguente

```
SOTTOINSIEMI(SOL, k, n)
    IF k = n THEN stampa il vettore SOL
    ELSE
    FOR i ← 0 TO 1 DO
        SOL[k + 1] ← i
        SOTTOINSIEMI(SOL, k + 1, n)
    ENDFOR
END
```

La procedura viene invocata da SOTTOINSIEMI($SOL, 0, n$). L'albero delle soluzioni che ne deriva è un albero binario completo di altezza n in quanto ogni nodo a livello $k < n$ ha esattamente due figli. Il costo della visita di un nodo interno richiede $O(1)$ mentre quello di una foglia richiede $O(n)$. Il costo della visita sarà dato dalla somma dei costi dei $2^n - 1$ nodi interni e delle 2^n foglie ed è quindi $O(n \cdot 2^n)$.

GENERARE TUTTE LE PERMUTAZIONI. Ci sono n diverse scelte per il valore del primo elemento di una permutazione di $\{1, 2, \ldots, n\}$. Fissato il primo elemento a_1 della permutazione, ci sono $n - 1$ candidati per la seconda posizione in quanto possiamo avere un qualunque valore eccetto a_1. Ripetendo questo argomento si ottiene un totale di $n!$ distinte permutazioni. Per rappresentare una permutazione possiamo utilizzare un vettore di n elementi dove nella cella i-esima troviamo l'i-esimo elemento della permutazione. L'insieme S_k dei candidati per la k-esima posizione è dato dall'insieme di elementi che non compaiono tra i $k - 1$ elementi della soluzione parziale. Lo pseudocodice della procedura è il seguente

```
PERMUTAZIONI(SOL, k, ELEM, n)
    IF k = n THEN stampa il vettore SOL
    ELSE
    FOR i ← 1 TO n DO
        IF ELEM[i] = 0 THEN
            SOL[k + 1] ← i
            ELEM[i] ← 1
            PERMUTAZIONI(SOL, k + 1, ELEM, n)
            ELEM[i] ← 0
        ENDIF
    ENDFOR
END
```

Per verificare in modo efficiente se un dato elemento è presente o meno nella soluzione parziale si è utilizzato il vettore $ELEM$ i cui valori sono inizializzati a zero. La procedura viene invocata da PERMUTAZIONI($SOL, 0, ELEM, n$).

L'albero delle soluzioni che ne deriva è un albero di altezza n dove i nodi a livello $k < n$ hanno esattamente $n - k$ figli (quindi a livello k l'albero ha esattamente $\frac{n!}{(n-k)!}$ nodi. Il costo della visita di un nodo interno o di una foglia richiede richiede $O(n)$. Il costo della visita sarà dato dalla somma dei costi dei $\sum_{k=0}^{n} \frac{n!}{(n-k)!} = n! \cdot \sum_{j=0}^{n} \frac{1}{j!} < n! \cdot \sum_{j=0}^{n} \frac{1}{2^{j-1}} = O(n!)$ nodi dell'albero ed è quindi $O(n \cdot n!)$.

Fin'ora abbiamo visto come per problemi di enumerazione conviene strutturare l'insieme delle soluzioni come le foglie di un albero, albero che verrà poi visitato con una visita in profondità. Supponiamo ora di essere interessati a soluzioni con particolari proprità *soluzioni ammissibili*, un possibile approccio sarebbe quello di generare comunque tutte le soluzioni e di stampare poi solo quelle che soddisfano i vincoli. Un tale approccio tende ad essere piuttosto costoso. Durante la visita dell'albero delle soluzioni, se su di un nodo interno si ha abbastanza informazione per concludere che il suo sottoalbero non contiene soluzioni ammissibili, allora è del tutto inutile esplorare il sottoalbero. Stando così le cose, una tecnica generale per cercare di ridurre i costi consiste nel definire, quando possibile, delle *funzioni di taglio* in grado di *potare* l'albero su cui si effettua la visita. Esemplifichiamo quanto appena detto, considerando il seguente problema

> **Enumerazione di sottoinsiemi di cardinalità limitata.** *Dato un insieme di n oggetti ed un intero c, con $1 \leq c \leq n$, enumerare tutti i sottoinsiemi con esattamente c oggetti.*

L'idea di riutilizzare la procedura che genera tutti i sottoinsiemi modificata in modo che prima di stampare una soluzione verifica se l'insieme generato ha esattamente k elementi porta al seguente algoritmo

```
SOTTOINSIEMI-VINCOLATI(SOL, k, c, n)
    IF k = n THEN
        tot ← 0
        FOR i ← 1 TO n DO
            IF SOL[i] = 1 THEN tot ← tot + 1
        ENDFOR
        IF tot = c THEN stampa il vettore SOL
    ELSE
    FOR i ← 0 TO 1 DO
        SOL[k + 1] ← i
        SOTTOINSIEMI-VINCOLATI(SOL, k + 1, c, n)
    ENDFOR
END
```

e la complessità dell'algoritmo è $O(n \cdot 2^n)$ in quanto l'albero delle soluzioni viene visitato comunque per intero indipendentemente dal numero di soluzioni ammissibili. Ora gli elementi da enumerare sono $\left(\begin{array}{c} n \\ c \end{array} \right) = O(n^c)$ e, quanto più c è piccolo rispetto ad n, tanto più la procedura risulta inefficiente.

La possibilità di potare l'albero delle soluzioni deriva dalle seguenti due osservazioni

(1) Un nodo corrispondente ad una soluzione parziale in cui sono gia stati inseriti più di c elementi ha come foglie del suo sottoalbero solo soluzioni con più di c elementi e quindi nel sottoalbero di questo nodo non sono presenti soluzioni ammissibili.
(2) un nodo corrispondente ad una soluzione parziale la somma dei cui elementi più tutti gli elementi non ancora considerati è inferiore a c ha come foglie del suo sottoalbero solo soluzioni con meno di c elementi e quindi nel sottoalbero di questo nodo non sono presenti soluzioni ammissibili.

nella visita dell'albero posso quindi utilizzare due funzioni di taglio, una che interviene per la visita del sottoalbero sinistro del nodo e l'altra che interviene per la visita del sottoalbero destro del nodo. La prima fa si che il figlio sinistro venga visitato solo se escludere l'elemento $k+1$-esimo dalla soluzione che si sta costruendo non preclude la possibilità di poter ottenere una soluzione con almeno c elementi. La seconda fa si che il nodo destro venga visitato solo se aggiungere l'elemento $k+1$-esimo alla soluzione parziale non genera una nuova soluzione parziale con più di c. elementi. L'algoritmo è il seguente

```
SOTTOINSIEMI-VINCOLATI1(SOL, k, c, n, tot)
    IF k = n THEN stampa il vettore SOL
    ELSE
        IF n − (k + 1) + tot ≥ c THEN
            SOL[k + 1] ← 0
            SOTTOINSIEMI(SOL, k + 1, c, n, tot)
        ENDIF
        IF tot < c THEN
            SOL[k + 1] ← 1
            SOTTOINSIEMI-VINCOLATI1(SOL, k + 1, c, n, tot + 1)
        ENDIF
    ENDIF
END
```

grazie ad una variabile *tot* con il numero di elementi presenti nella soluzione parziale il calolo delle due funzioni di taglio richiederà tempo costante. La funzione di taglio per il sottoalbero sinistro è un semplice test per verificare che $n-(k+1)+tot$ sia almeno c (ciò assicura che restano ancora da considerare un numero di oggetti sufficienti a completare la soluzione parziale in modo da ottenere un sottoinsieme di almeno c oggetti). La funzione di taglio per il sottoalbero destro è un semplice test che verifica che $tot + 1$ non superi il valore c.

Grazie alle due funzioni di taglio si è sicuri che durante la visita dell'albero delle soluzioni un nodo dell'albero viene visitato se e solo se nel suo sottoalbero è presente almeno una soluzione ammissibile. Per il calcolo della complessità di un algoritmo di backtracking con questa proprietà è utile il seguente teorema.

Teorema 1. *Sia h l'altezza dell'albero delle soluzioni, $O(f(n))$ il tempo richiesto dalla visita di un suo nodo interno ed $O(g(n))$ il tempo richiesto dalla visita di una sua foglia. Se nel corso della visita vengono visitati*

esclusivamente nodi nei cui sottoalberi sono presenti soluzioni ammissibili, il tempo richiesto dalla visita è $O(h \cdot f(n) \cdot D(n) + g(n) \cdot D(n))$ *dove* $D(n)$ *è il numero di soluzioni ammissibili.*

Dimostrazione. Poichè un nodo dell'albero viene visitato solo se nel suo sottoalbero sono presenti soluzioni ammissibili, il sottoalbero effettivamente visitato avrà come foglie le $D(n)$ soluzioni ammissibili. Ogni nodo interno visitato appartiene ad almeno uno dei cammini radice-foglia ciascuno dei quali ha lunghezza al più h. Quindi i nodi interni visitati sono al più $h \cdot D(n)$. Il costo della visita è dato dalla somma dei costi di visita dei nodi interni e delle foglie effettivamente visitate. Da quanto detto questo costo è $O(h \cdot f(n) \cdot D(n) + g(n) \cdot D(n))$.

Ritornando al problema del calcolo della complessità della procedura SOTTOINSIEMI-VINCOLATI1 notiamo che: l'albero delle soluzioni ha altezza n, il costo per la visita di un nodo interno è $O(1)$, quello di una foglia è $O(n)$ e il numero $D(n)$ di soluzioni ammissibili (vale a dire sottoinsiemi con esattamente c degli n oggetti) è $\left(\begin{array}{c} n \\ c \end{array} \right) = O(n^c)$. Dal Teorema 1 derivo quindi che la complessità di tempo della procedura è $O(n^{c+1})$.

Cicli hamiltoniani. Come altro esempio di utilizzo di funzioni di taglio nell'enumerazione di soluzioni ammissibili consideriamo ora una problema su grafi. Sia G in grafo di n nodi. Un *ciclo hamiltoniano* per G è un cammino chiuso nel grafo che attraversa esattamente n archi e tocca tutti i nodi del grafo.
Consideriamo il problema di enumerare tutti i cicli hamiltoniani di un dato grafo $G = (V, E)$.
Possiamo rappresentare i cicli hamiltoniani come permutazioni degli n nodi del grafo G. Ovviamente non tutte le permutazioni degli n nodi sono necessariamente un ciclo. Un grafo completo avrà $(n-1)!$ cicli hamiltoniani mentre un grafo sparso potrebbe non averne neanche uno (si pensi ad esempio ad un albero).

Perchè una permutazione sia un ciclo hamiltoniano per G tra un nodo e quello che segue nella permutazione deve esserci un arco in E e lo stesso deve valere anche per l'ultimo nodo ed il primo nodo della permutazione. Una semplice funzione di taglio è quella che evita la generazione di permutazioni parziali in cui fra un nodo e il successivo non sia presente un arco in E (ovviamente permutazioni di questo genere non potranno completarsi in soluzioni ammissibili). Infine uno stesso ciclo hamiltoniano può essere rappresentato da n diverse permutazioni in funzione del nodo da cui si parte fa partire il ciclo. Onde evitare di stampare più volte uno stesso ciclo assumiamo quindi che il primo elemento della permutazione sia sempre il vertice 1 del grafo (vale a dire $SOL[1] = 1$). L'algoritmo è il seguente:

```
CICLI-HAMILTONIANI(SOL, k, ELEM, n, G)
    IF k = n AND {SOL[n], SOL[1]} ∈ E THEN stampa il vettore SOL
    ELSE
    FOR i ← 2 TO n DO
        IF ELEM[i] = 0 AND {SOL[k], i} ∈ E THEN
            SOL[k + 1] ← i
            ELEM[i] ← 1
            CICLI-HAMILTONIANI(SOL, k + 1, ELEM, n, G)
            ELEM[i] ← 0
        ENDIF
    ENDFOR
END
```

La procedura viene invocata da CICLI-HAMILTONIANI$(SOL, 1, ELEM, n, G)$. Per quanto visto sull'enumerazione di permutazioni la visita totale dell'albero richiede $O(n!)$, tuttavia grazie alla funzione di taglio l'albero viene parzialmente potato. E' ovvio che la funzione di taglio proposta non assicura che un nodo venga visitato solo se nel suo sottoalbero sono presenti soluzioni ammissibili, quindi il teorema 1 non può essere utilizzato e la complessità è al caso pessimo ancora $O(n!)$. Quel che si può dire è che, quanto più il grafo è sparso tanto più l'albero delle soluzioni verrà potato e i tempi richiesti della visita ridotti.

1.1. **Backtracking e problemi di ottimizzazione.** Per la soluzione di problemi di ottimizzazione un possibile approccio è quello di enumerare tutte le soluzioni ammissibili e, nel corso dell'enumerazione, tener traccia della miglior soluzione ammissibile fino al momento trovata (*ottimo attuale*). Ovviamente, al termine dell'enumerazione, l'ottimo attuale coinciderà con la soluzione al problema. Nel corso della visita dell'albero delle soluzioni, al fine di ridurre i tempi di calcolo, conviene ricorrere a funzioni di taglio in grado di potare sottoalberi che non contengono soluzioni ammissibili. Nel contesto dei problemi di ottimizzazione si rivela utile anche un'altro tipo di funzioni di taglio: funzioni in grado di potare sottoalberi

che non contengono soluzioni ammissibili in grado di migliorare l'ottimo attuale. In quest'ultimo caso si parla di *funzioni di taglio dinamiche* mentre nel primo caso si parla di *funzioni di taglio statiche*.

Il problema dello Zaino. Per esemplificare quanto detto consideriamo il problema di ottimizzazione dello Zaino. Per questo problema si hanno n oggetticiascuno con un proprio valore ed un proprio peso ed uno zaino con una fissata capacità. Quel che si vuole è il sottoinsieme di oggetti il cui peso complessivo non eccede la capacità dello zaino e il cui valore complessivo sia massimo. Le soluzioni sono quindi i diversi sottoinsiemi di oggetti, le soluzioni ammissibili sono i sottoinsiemi di oggetti il cui valore complessivo non eccede la capiacità dello zaino e la soluzione ottima è la soluzione ammissibile per cui risulta massimo il valore complessivo dei suoi oggetti. Nel visitare l'albero delle soluzioni possiamo introdurre una funzione di taglio sui sottoalberi destri dei nodi che eviti di generare soluzioni parziali di peso superiore alla capacità dello zaino. Si tratta in questo caso di una funzione di taglio statica in quanto evita la visita di sottoalberi privi di soluzioni ammissibili. Possiamo pure introdurre una funzione di taglio dinamica sui sottoalberi sinistri dei nodi. Per far ciò possiamo pensare di calcolare una limitazione superiore alla bontà delle soluzioni ammissibili presenti nel sottoalbero, se questo valore non è superiore al valore dell' ottimo attuale allora il sottoalbero può essere potato. Ovviamente quanto più precisa sarà la limitazione, tanto più la funzione di taglio sarà effettiva. Una limitazione banale del tipo $+\infty$ non genererebbe alcun taglio, d'altra parte limitazioni superiori molto precise potrebbero richiedere tempi di calcolo tali da rendere preferibile comunque l'esplorazione del sottoalbero. Nel nostro caso un buon compromesso potrebbe essere sommare al valore degli oggetti della soluzione parziale il valore di tutti gli oggetti rimanenti. Per una limitazione più attendibile si può procedere come segue: se c' è il peso complessivo della soluzione attuale e C la capacità dello zaino, si risolve con l'efficiente algoritmo greedy il problema dello zaino frazionario sull' istanza che comprende tutti gli oggetti non ancora considerati ed uno zaino di capacità $C-c'$. Ovviamente il valore lim della soluzione ottenuta è un limite superiore al valore aggiuntivo che che ci si può attendere estendendo la soluzione parziale. Il limite superiore alla bontà delle soluzioni ammissibili prodotte dal sottoalbero è quindi $val + lim$ dove val è il valore della soluzione parziale. Lo pseudocodice dell'algoritmo è il seguente

```
ZAINO(P, V, c, n, SOL, SOL*, val*, valore, peso, k)
    IF k = n THEN
        FOR i ← 1 TO n DO SOL*[i] ← SOL[i]
        val* ← valore
    ELSE
        IF peso + P[k + 1] ≤ c THEN
            SOL[k + 1] ← 1
            ZAINO(P, V, n, c, SOL, SOL*, val*, valore + V[k + 1], peso + P[k + 1], k + 1)
        ENDIF
        y ← limite superiore al valore delle soluzioni ottenibili estendendo le scelte fatte per i primi k
   oggetti e
        non prendendo l'oggetto k + 1-esimo.
        IF y > val* THEN
            SOL[k + 1] ← 0
            ZAINO(P, V, n, c, SOL, SOL*, val*, valore, peso, k + 1)
        ENDIF
    ENDIF
END
```

V è il vettore dei pesi degli n oggetti, P è il vettore dei valori degli n oggetti e c è la capacità dello zaino. SOL è il vettore caratteristico della soluzione parziale ed ha valore $valore$ e peso $peso$ mentre SOL^* è la soluzione ottima attuale e val^* il suo valore . La procedura viene chiamata con ZAINO$(P, V, c, SOL, SOL^*, -\infty, 0, 0, 0)$ ed al termine in SOL^* troveremo il vettore caratteristico della soluzione ottima il cui valore sarà in val^*.
Resta da dettagliare il calcolo del limite superiore.

- Se si decide di sommare al valore degli oggetti della soluzione il valore di tutti gli oggetti rimanenti, allora il calcolo può essere fatto in tempo $O(1)$ precalcolando un vettore $RESTI$ dove nella i-ma cella troviamo il valore $\sum_{i=k}^{n} V[i]$.Assumendo che alla procedura ZAINO tra i vari parametri viene passato anche il vettore $RESTI$ si ha:

  IF $k < n-1$ THEN $y \leftarrow valore + RESTI[k+2]$
  ELSE $y \leftarrow valore$

- Se si decide di utilizzare l'algoritmo greedy allora può essere conveniente preordinare gli oggetti per rapporto valore/peso non decrescente, stando così le cose il calcolo di y richiede $O(n)$ con la seguente implementazione

$y \leftarrow valore$
$p \leftarrow peso$
$i \leftarrow k+2$
WHILE $i \leq n$ AND $p + P[i] \leq C$ DO
    $y \leftarrow y + V[i]$
    $p \leftarrow p + P[i]$
    $i \leftarrow i + 1$
ENDWHILE
IF $i \leq n$ THEN $y \leftarrow y + V[i] \cdot \frac{(C-p)}{P[i]}$

Il limite superiore ottenuto grazie alla procedura greedy è ovviamente di qualità superiore a quello ottenuto tenendo conto di tutti gli oggetti non ancora considerati. Quindi se si verifica un taglio con l'utilizzo del secondo limite allora lo stesso taglio si verifica anche col primo limite mentre non è vero il viceversa. In altre parole il numero di nodi dell'albero delle soluzioni effettivamente visitati quando si usa il primo limite deve essere non superiore a quello che si verifica utilizzando il secondo limite e appare quindi da preferirsi. Tuttavia il costo della visita di un nodo quando si utilizza il limite superiore ottenuto col greedy è $O(n)$ a fronte di un costo di visita $O(1)$ dell'altro caso. Il ridotto numero di nodi effettivamente visitati grazie alla limitazione greedy potrebbe non compensare il costo superiore richiesto dai nodi effettivamente visitati. In questo caso non v'è dubbio che l'utilizzo del limite greedy è da preferisti. In generale decidere se vale la pena di ricorrere a limiti inferiori di migliore qualità ma con costo di calolo superiore è un problema cui solo la sperimentazione può dare risposta.
Infine è ovvio che le funzioni di taglio dinamiche si mostrano tanto più utili quanto migliore è il valore dell'ottimo attuale. Nel nostro caso ad esempio all'inizio val^* varrà meno infinito e, qualunque sia il limite inferiore scelto, fino alla scoperta della prima soluzione ammissibile la funzione di taglio non produrrà alcun effetto. Gli effetti cominceranno ad essere sensibili quando finalmente si disporrà di un ottimo attuale di buona qualità. Tutto questo consiglia di precalcolare con una qualche euristica una soluzione ammissibile di buona qualità e invocare poi la procedura di Backtracking settando val^* al valore della soluzione prodotta dall'euristica.