Appendice A

Algoritmi e complessità

In questa appendice vogliamo brevemente richiamare alcuni concetti fondamentali della teoria della complessità computazionale, utili per meglio comprendere la diversa "difficoltà" della soluzione dei problemi di ottimizzazione. Presentiamo inoltre una breve introduzione ad alcuni dei principali paradigmi algoritmici che si utilizzano per la soluzione, eventualmente approssimata, di problemi di ottimizzazione.

A.1 Modelli computazionali

Una volta che un problema P sia stato formulato, deve essere risolto: siamo quindi interessati alla messa a punto di strumenti di calcolo che, data una qualsiasi istanza p, siano in grado di fornirne una soluzione in un tempo finito. Tali strumenti di calcolo si chiamano *algoritmi*.

Un algoritmo che risolve P può essere definito come una sequenza finita di istruzioni che, applicata ad una qualsiasi istanza p di P, si arresta dopo un numero finito di passi (ovvero di computazioni elementari), fornendo una soluzione di p oppure indicando che p non ha soluzioni ammissibili.

Per poter studiare gli algoritmi dal punto di vista della loro efficienza, o complessità computazionale, è necessario definire un modello computazionale: classici modelli computazionali sono la Macchina di Turing (storicamente il primo proposto), la R.A.M. (Random Access Machine), la Macchina a Registri (MR), etc.

La Macchina a Registri è un buon compromesso tra semplicità e versatilità: una MR consiste di un numero (finito ma non limitato) di registri, ciascuno dei quali può contenere un singolo numero intero, e di un programma, ossia di una sequenza finita di istruzioni del tipo

- incrementa il registro k e salta all'istruzione j;
- decrementa il registro k e salta all'istruzione j;

- se il registro k contiene 0 salta all'instruzione j, altrimenti salta all'istruzione h.

Si tratta di una macchina *sequenziale e deterministica*, poiché il comportamento futuro della macchina è univocamente determinato dalla sua configurazione presente. Una MR è un buon modello astratto di un calcolatore elettronico, ed è quindi in grado di compiere tutte le computazioni possibili in un qualunque sistema di calcolo attualmente noto (se si eccettuano i computer quantistici, la cui implementabilità è comunque ancora da dimostrare). D'altra parte, si può dimostrare che la classe delle funzioni computabili da una MR è equivalente alla classe delle funzioni computabili da una Macchina di Turing, il che, secondo la *Tesi di Church*, implica che le MR siano presumibilmente in grado di calcolare qualsiasi funzione effettivamente computabile con procedimenti algoritmici.

A.2 Misure di complessità

Dato un problema P, una sua istanza p, e un algoritmo A che risolve P, indichiamo con costo (o complessità) di A applicato a p una misura delle risorse utilizzate dalle computazioni che A esegue su una macchina MR per determinare la soluzione di p. Le risorse, in principio, sono di due tipi, *memoria occupata e tempo di calcolo*: nell'ipotesi che tutte le operazioni elementari abbiano la stessa durata, il tempo di calcolo può essere espresso come numero di operazioni elementari effettuate dall'algoritmo. Poiché molto spesso la risorsa più critica è il tempo di calcolo, nel seguito useremo soprattutto questa come misura della complessità degli algoritmi.

Dato un algoritmo, è opportuno disporre di una misura di complessità che consenta una valutazione sintetica della sua bontà ed eventualmente un suo agevole confronto con algoritmi alternativi. Conoscere la complessità di A per ognuna delle istanze di P non è possibile (l'insieme delle istanze di un problema è normalmente infinito), né sarebbe di utilità pratica: si cerca allora di esprimere la complessità come una funzione $g(n)$ della dimensione, n, dell'istanza cui viene applicato l'algoritmo. Poiché, per ogni dimensione, si hanno in generale molte istanze di quella dimensione, si sceglie $g(n)$ come il costo necessario per risolvere la più difficile tra le istanze di dimensione n: si parla allora di *complessità nel caso peggiore*.

Bisogna naturalmente definire in modo preciso il significato di dimensione di una istanza: chiameremo dimensione di p una misura del numero di bit necessari per rappresentare, con una codifica "ragionevolmente" compatta, i dati che definiscono p, cioè una misura della lunghezza del suo input. Per esempio, in un grafo con n nodi e m archi i nodi possono essere rappresentati dagli interi tra 1 ed n e gli archi per mezzo di una lista contenente m coppie di interi (l'arco che collega i nodi i e j è rappresentato dalla coppia (i, j)): trascurando le costanti moltiplicative, potremo allora assumere come misura

della dimensione della codifica del grafo, al variare del numero dei nodi e degli archi, la funzione $m \log n$, dato che interi positivi e non superiori ad n possono essere rappresentati con $\log n$ bit[1]. Nel seguito, per semplicità, oltre alle costanti moltiplicative trascureremo anche le funzioni sublineari, come la funzione logaritmo; diremo allora che m è la lunghezza dell'input per un grafo con m archi. Nelle ipotesi fatte, la misura della lunghezza dell'input non varia se usiamo una codifica in base $b > 2$: se invece si usasse una codifica unaria, la lunghezza dell'input nell'esempio in questione diventerebbe nm, aumentando considerevolmente.

A questo punto la funzione $g(n)$, introdotta precedentemente, risulta definita in modo sufficientemente rigoroso: in pratica essa continua però ad essere di difficile uso come misura della complessità, dato che risulta difficile, se non praticamente impossibile, la valutazione di $g(n)$ per ogni dato valore di n. Questo problema si risolve sostituendo alla $g(n)$ il suo ordine di grandezza: si parlerà allora di *complessità asintotica*.

Data una funzione $g(x)$, diremo che:

1. $g(x)$ è $O(f(x))$ se esistono due costanti c_1 e c_2 per cui, per ogni x, è $g(x) \leq c_1 f(x) + c_2$;

2. $g(x)$ è $\Omega(f(x))$ se $f(x)$ è $O(g(x))$;

3. $g(x)$ è $\Theta(f(x))$ se $g(x)$ è allo stesso tempo $O(f(x))$ e $\Omega(f(x))$.

Sia $g(x)$ il numero di operazioni elementari che vengono effettuate dall'algoritmo A applicato alla più difficile istanza, tra tutte quelle che hanno lunghezza di input x, di un dato problema P: diremo che la complessità di A è un $O(f(x))$ se $g(x)$ è un $O(f(x))$; analogamente, diremo che la complessità di A è un $\Omega(f(x))$ o un $\Theta(f(x))$ se $g(x)$ è un $\Omega(f(x))$ o un $\Theta(f(x))$.

A.3 Problemi trattabili e problemi intrattabili

Chiameremo *trattabili* i problemi per cui esistono algoritmi la cui complessità sia un $O(p(x))$, con $p(x)$ un polinomio in x, e *intrattabili* i problemi per cui un tale algoritmo non esiste: le seguenti tabelle chiariscono il perché di questa distinzione. In questa tabella vengono forniti, per diverse funzioni di complessità f, i tempi di esecuzione (in secondi, ove non diversamente specificato) per alcuni valori di n su un calcolatore che richieda $1e^{-6}$ secondi per effettuare un'operazione elementare.

[1] In generale, tranne quando sarà detto il contrario, se i dati di un problema sono costituiti da n numeri considereremo come limitato da $\log n$ il numero di bits necessari per la codifica in binario dei numeri stessi.

f	10	20	40	60
n	$1e^{-5}$	$2e^{-5}$	$4e^{-5}$	$6e^{-5}$
n^3	$1e^{-3}$	$8e^{-3}$	$7e^{-2}$	$2e^{-1}$
n^5	$1e^{-1}$	3.2	1.7 min.	13 min.
2^n	$1e^{-3}$	1	13 giorni	36600 anni
3^n	$6e^{-2}$	1 ora	$4e^5$ anni	$1e^{13}$ anni

In questa tabella vengono invece indicati i miglioramenti ottenibili, in termini di dimensioni delle istanze risolvibili, per diverse funzioni di complessità, al migliorare della tecnologia dei calcolatori: con x_i abbiamo indicato la dimensione di un'istanza risolvibile oggi in un minuto per la i-esima funzione di complessità.

f	**Computer odierno**	**100 volte più veloce**	**10000 volte più veloce**
n	x_1	$100x_1$	$10000x_1$
n^3	x_2	$4.6x_2$	$21.5x_2$
n^5	x_3	$2.5x_3$	$6.3x_3$
2^n	x_4	$x_4 + 6.6$	$x_4 + 13.2$
3^n	x_5	$x_5 + 4.2$	$x_5 + 8.4$

Molti problemi di rilevante importanza pratica sono trattabili: sono problemi per i quali disponiamo di efficienti algoritmi di complessità polinomiale. Per potere effettuare una più rigorosa classificazione dei diversi problemi, facciamo riferimento a problemi in forma decisionale.

A.3.1 Le classi $\mathcal{P}$ e $\mathcal{NP}$

Una prima importante classe di problemi è la classe $\mathcal{NP}$, costituita da tutti i problemi decisionali il cui problema di certificato associato può essere risolto in tempo polinomiale. In altri termini, i problemi in $\mathcal{NP}$ sono quelli per cui è possibile verificare efficientemente una risposta "si", perché è possibile decidere in tempo polinomiale se una soluzione x è ammissibile per il problema.

Ad esempio, il problema della soddisfattibilità proposizionale (SAT), introdotto in 1.2.3.1, è un problema in $\mathcal{NP}$: dato un qualunque assegnaento di valori di verità, è possibile verificare se tale assegnamento rende vera la formula in tempo lineare nella dimensione della formula.

Equivalentemente, si può definire $\mathcal{NP}$ come la classe di tutti i problemi decisionali risolubili in tempo polinomiale da una MR nondeterministica ($\mathcal{NP}$ va infatti inteso come $\mathcal{P}$olinomiale nel calcolo $\mathcal{N}$ondeterministico). Una MR nondeterministica è il modello di calcolo (astratto) in cui una MR, qualora si trovi ad affrontare un'operazione di salto condizionale, può eseguire *contemporaneamente* entrambi rami dell'operazione, e questo ricorsivamente

per un qualsiasi numero di operazioni. In altre parole, i problemi in $\mathcal{NP}$ sono quelli per cui esiste una computazione di lunghezza polinomiale che può portare a costruire una soluzione ammissibile, se esiste, ma questa computazione può essere "nascosta" entro un insieme esponenziale di computazioni analoghe tra le quali, in generale, non si sa come discriminare.

Ad esempio, per SAT si può immaginare una MR nondeterministica che costruisca l'assegnamento di valori di verità ai letterali con una computazione in cui, sequenzialmente, viene assegnato il valore di verità a ciascun letterale, in un certo ordine prestabilito, in base ad una certa condizione logica. Questo identifica un *albero di computazione* che descrive tutte le possibili esecuzioni della MR, e le cui foglie sono tutti i 2^n possibili assegnamenti di valori di verità agli n letterali della formula. Se la formula è soddisfattibile, allora esiste un cammino nell'albero di computazione, di lunghezza lineare nel numero di letterali, che porta a costruire esattamente un certificato del problema, ossia un assegnamento di valori di verità che soddisfa la formula.

Un sottoinsieme della classe $\mathcal{NP}$ è la classe $\mathcal{P}$, costituita da tutti i problemi risolubili in *tempo polinomiale*, ossia contenente tutti quei problemi decisionali per i quali esistono algoritmi di complessità polinomiale che li risolvono *(problemi polinomiali)*.

Una domanda particolarmente importante è se esistano problemi in $\mathcal{NP}$ che non appartengano anche a $\mathcal{P}$ cioè se sia $\mathcal{P} \neq \mathcal{NP}$: a questa domanda non si è in grado di rispondere, anche si ritiene fortemente probabile che la risposta sia positiva, cioè che sia effettivamente $\mathcal{P} \neq \mathcal{NP}$. Una breve giustificazione di questo sarà data nel paragrafo successivo.

A.3.2 Problemi $\mathcal{NP}$-completi e problemi $\mathcal{NP}$-ardui

Molti problemi, anche se apparentemente notevolmente diversi, possono tuttavia essere ricondotti l'uno all'altro; dati due problemi decisionali, P e Q, diciamo che P *si riduce in tempo polinomiale a* Q, e scriveremo $P \propto Q$, se, supponendo l'esistenza di un algoritmo A_Q che risolva Q in tempo costante (indipendente dalla lunghezza dell'input), esiste un algoritmo che risolve P in tempo polinomiale utilizzando come sottoprogramma A_Q: parliamo in tal caso di *riduzione polinomiale di* P a Q.

Ad esempio, SAT si riduce polinomialmente alla Programmazione Lineare Intera, come mostrato in 1.2.3.1. Quindi, se esistesse un algoritmo polinomiale per la *PLI* allora esisterebbe un algoritmo polinomiale per SAT: data la formula in forma normale congiuntiva, basterebbe produrre (in tempo polinomiale) il problema di *PLI* corrispondente, applicare l'algoritmo a tale problema e rispondere "si" o "no" a seconda della risposta ottenuta. Possiamo quindi affermare che $SAT \propto PLI$.

E' facile verificare che la relazione $\propto$ ha le seguenti proprietà:

1. è riflessiva: $A \propto A$;

2. è transitiva: $A \propto B$ e $B \propto C \Rightarrow A \propto C$;

3. se $A \propto B$ e $A \notin P$; allora $B \notin P$;

4. se $A \propto B$ e $B \in P$, allora $A \in P$.

Possiamo definire adesso la classe dei problemi $\mathcal{NP}$-completi: un problema A è detto $\mathcal{NP}$-*completo* se $A \in \mathcal{NP}$ e se per ogni $B \in$ si ha che $B \propto A$.

La classe dei problemi $\mathcal{NP}$-completi costituisce un sottoinsieme di $\mathcal{NP}$ di particolare importanza; un fondamentale teorema, dovuto a Cook (1971), garantisce che ogni problema $P \in \mathcal{NP}$ si riduce polinomialmente a SAT, ossia che tale classe non è vuota.

I problemi $\mathcal{NP}$-completi hanno la proprietà che se esiste per uno di essi un algoritmo polinomiale, allora necessariamente tutti i problemi in $\mathcal{NP}$ sono risolubili polinomialmente, e quindi è $\mathcal{P} = \mathcal{NP}$; in un certo senso, tali problemi sono i più difficili tra i problemi in $\mathcal{NP}$. Un problema che abbia come caso particolare un problema $\mathcal{NP}$-completo ha la proprietà di essere almeno tanto difficile quanto i problemi $\mathcal{NP}$-completi (a meno di una funzione moltiplicativa polinomiale): un problema di questo tipo si dice $\mathcal{NP}$-arduo. Si noti che un problema $\mathcal{NP}$-arduo può anche non appartenere a $\mathcal{NP}$. Ad esempio, sono $\mathcal{NP}$-ardui i problemi di ottimo la cui versione decisionale è un problema $\mathcal{NP}$-completo: infatti, come si è già visto, è sempre possibile ricondurre un problema decisionale ad un problema di ottimo con una opportuna scelta della funzione obiettivo.

Ad esempio, il problema della *PLI* è $\mathcal{NP}$-arduo, poichè ad esso si riduce SAT, che è $\mathcal{NP}$-completo.

Fino ad oggi, sono stati trovati moltissimi problemi $\mathcal{NP}$-ardui; si può affermare che la grande maggioranza dei problemi combinatori siano $\mathcal{NP}$-ardui. Nonostante tutti gli sforzi dei ricercatori, non è stato possibile determinare per nessuno di essi un algoritmo polinomiale (il che avrebbe fornito algoritmi polinomiali per tutti i problemi della classe); questo fa oggi ritenere che non esistano algoritmi polinomiali per i problemi $\mathcal{NP}$-completi, ossia che sia $\mathcal{P} \neq \mathcal{NP}$. Ad oggi non si conosce nessuna dimostrazione formale di questa ipotesi, ma essa è suffragata da molti indizi. Ad esempio, $\mathcal{P}$ ed $\mathcal{NP}$ sono solo i primi elementi di una *gerarchia polinomiale* di classi di problemi, sempre "più difficili", che collasserebbero tutte sulla sola classe $\mathcal{P}$ qualora fosse $\mathcal{P} = \mathcal{NP}$, il che fa ritenere questa eventualità altamente improbabile.

Per riassumere, possiamo dire che, allo stato delle conoscenze attuali, tutti i problemi $\mathcal{NP}$-ardui sono presumibilmente intrattabili. Naturalmente, ciò non significa che non sia in molti casi possibile costruire algoritmi in grado di risolvere efficientemente istanze di problemi $\mathcal{NP}$-ardui di dimensione significativa (quella richiesta dalle applicazioni reali); questo non rientra comunque nel campo della teoria della complessità computazionale.

A.3.3 Complessità ed approssimazione

Esistono molti altri risultati interessanti della teoria della complessità computazionale che non rientrano nello scopo di queste note. Risultati molto importanti permettono, ad esempio, di caratterizzare classi di problemi per cui non solo il problema originario, ma anche ottenere una *soluzione approssimata* sia un problema intrattabile.

Dato un problema

$$(P) \qquad \min\{c(x) : x \in F\} .$$

ed una sua soluzione ammissibile x, definiamo

$$E_x = c(x) - z(P) (\geq 0)$$

l'*errore assoluto* di x (si ricordi che $z(P)$ indica il valore ottimo della funzione obiettivo per P). In pratica, "risolvere" un problema di ottimizzazione può significare determinare una soluzione x con un "basso" errore assoluto. Siccome però tale misura non è invariante rispetto ai cambiamenti di scala, si preferisce utilizzare l'*errore relativo*

$$R_x = \frac{c(x) - z(P)}{|z(P)|}$$

(per evitare problemi nel caso in cui $z(P) = 0$, in pratica si pone il denominatore uguale a $\max\{|z(P)|, 1\}$). Dato $\varepsilon > 0$, la soluzione x si dice *ε-ottima* se $R_x \leq \varepsilon$.

Si noti che, in generale, $z(P)$ non è noto, per cui calcolare l'errore (assoluto o relativo) di una soluzione x è non banale. In effetti, esiste una consistente differenza tra *produrre* una soluzione ε-ottima e *certificare* la ε-ottimalità di una soluzione data. Ad esempio, in molti algoritmi enumerativi per problemi "difficili" (si veda A.4.3) capita sovente che l'algoritmo determini la soluzione ottima in tempo relativamente breve, ma sia poi ancora necessario un grandissimo sforzo computazionale per dimostrare che tale soluzione è effettivamente ottima.

Un algoritmo per P che costruisca una soluzione non necessariamente ottima viene detto *algoritmo euristico*, mentre un algoritmo in grado di determinare la soluzione ottima viene detto *algoritmo esatto*. Nella costruzione di algoritmi euristici, è utile (ma non sempre facile, né possibile) effettuare un'analisi dell'errore che si commette accettando la soluzione ottenuta invece di quella ottima. Esistono classi di algoritmi euristici con proprietà specifiche nei confronti dell'approssimazione. Un algoritmo A si dice *ε-approssimato* se la soluzione x_A prodotta dall'algoritmo è ε-ottima per ogni istanza. Algoritmi ε-approssimati con ε piccolo costituiscono spesso alternative interessanti ad algoritmi esatti. Quando – come spesso accade – non sia

possibile costruire algoritmi di questo tipo, si può ricorrere a *stime dell'errore a posteriori*, che consentano almeno di avere una valutazione sull'errore commesso, e quindi sulla "qualità" della soluzione fornita.

Uno *schema di approssimazione* è un algoritmo che, oltre all'istanza p di P, prende in input anche un parametro $\varepsilon > 0$ e produce una soluzione x che sia ε-ottima; tipicamente, la complessità di un tale algoritmo sarà anche una funzione dell'approssimazione ε voluta (crescente con $1/\varepsilon$). Uno schema di approssimazione è detto:

- *polinomiale*, se la sua complessità è polinomiale nella dimensione dell'istanza p;
- *pienamente polinomiale*, se la sua complessità è polinomiale nella dimensione di p ed in $1/\varepsilon$.

Accenniamo brevemente qui ad un noto risultato che riguarda la difficoltà di costruire schemi di approssimazione pienamente polinomiali per un problema di ottimizzazione.

Esistono problemi $\mathcal{NP}$-completi che sono risolvibili polinomialmente nella lunghezza dell'input se codificati in unario, mentre non lo sono con codifiche più compatte: un esempio è il problema dello zaino (1.2.2.1). Tali problemi sono in un certo senso più facili di quelli che richiedono algoritmi esponenziali indipendentemente dalla codifica dell'input. Viene detto *pseudopolinomiale* un algoritmo che risolva uno di tali problemi in tempo polinomiale nella lunghezza dell'input codificato in unario; un problema è quindi detto *pseudopolinomiale* se può essere risolto per mezzo di un algoritmo pseudopolinomiale. Viene chiamato $\mathcal{NP}$-completo *in senso forte* un problema $\mathcal{NP}$-completo per cui non esistono algoritmi pseudopolinomiali; analogamente, viene chiamato $\mathcal{NP}$-arduo *in senso forte* un problema di ottimizzazione che abbia come versione decisionale un problema $\mathcal{NP}$-completo in senso forte.

I problemi $\mathcal{NP}$-completi in senso forte sono tipicamente quelli che "non contengono numeri": ad esempio, SAT è un problema $\mathcal{NP}$-completo in senso forte. Un diverso esempio è il problema del commesso viaggiatore (TSP) (si veda 1.2.2.3), che resta $\mathcal{NP}$-arduo anche se i costi degli archi sono limitati a due soli possibili valori, ad esempio "0" e "1". Infatti, dato un algoritmo per TSP con costi 0/1 è possibile risolvere (in modo ovvio) il problema del ciclo Hamiltoniano su un grafo, che è notoriamente $\mathcal{NP}$-completo.

È possibile dimostrare che l'esistenza di algoritmi pseudopolinomiali per un problema è equivalente all'esistenza di *schemi di approssimazione pienamente polinomiali* per il problema; in altri termini, nessun problema $\mathcal{NP}$-arduo in senso forte (la grande maggioranza) ammette schemi di approssimazione pienamente polinomiali (a meno che $\mathcal{P} = \mathcal{NP}$). Di conseguenza, per la maggior parte dei problemi $\mathcal{NP}$-ardui è difficile non solo risolvere il

problema originario, ma anche una sua approssimazione arbitraria. Risultati di questo tipo sono stati dimostrati anche per approssimazioni con errore relativo fissato.

A.4 Algoritmi

Al termine di questa appendice, presentiamo una brevissima panoramica di alcune classi generali di algoritmi per la soluzione, eventualmente approssimata, di problemi di ottimizzazione. Questa panoramica non ha certamente la pretesa di essere esaustiva, ma solo di offrire un "primo assaggio" di alcune fondamentali tecniche algoritmiche che sono usate in queste note.

A.4.1 Algoritmi greedy

Gli algoritmi *greedy* (voraci) determinano la soluzione attraverso una sequenza di decisioni "localmente ottime", senza mai tornare, modificandole, sulle decisioni prese. Questi algoritmi sono di facile implementazione e notevole efficienza computazionale, ma, sia pure con alcune eccezioni di rilievo, non garantiscono l'ottimalità, ed a volte neppure l'ammissibilità, della soluzione trovata.

Vediamo adesso tre esempi di algoritmi greedy.

A.4.1.1 Algoritmo "Costi Unitari Decrescenti"

Questo algoritmo per il problema dello zaino costruisce una soluzione inserendo per primi nello zaino gli oggetti "più promettenti", cioè quelli che hanno maggior valore per unità di peso. L'algoritmo inizializza l'insieme I degli oggetti selezionati come l'insieme vuoto, e poi scorre la lista degli oggetti in ordine di costo unitario non crescente, cioè iniziando da quelli che hanno il più alto valore di c_i/a_i. L'oggetto a_h di volta in volta selezionato viene accettato se la capacità residua dello zaino è sufficiente, cioè se $b - \sum_{i \in I} a_i \geq a_h$; in questo caso l'oggetto a_h viene aggiunto ad I, altrimenti viene scartato e si passa al successivo nell'ordinamento. L'algoritmo termina quando tutti gli oggetti sono stati esaminati oppure la capacità residua dello zaino diviene 0.

Consideriamo ad esempio la seguente istanza del problema dello zaino:

$$\begin{array}{lrrrrrrcl} \text{Max} & 7x_1 & +2x_2 & +4x_3 & +5x_4 & +4x_5 & +x_6 & & \\ & 5x_1 & +3x_2 & +2x_3 & +3x_4 & +x_5 & +x_6 & \leq & 8 \\ & x_1, & x_2, & x_3, & x_4, & x_5, & x_6 & \in & \{0,1\} \end{array}$$

In questo caso, l'algoritmo greedy farebbe i seguenti passi:

1. la variabile con costo unitario maggiore è x_5, per cui risulta $c_5/a_5 = 4$: si pone allora $x_5 = 1$, e lo zaino rimane con una capacità residua di 7 unità;

2. la seconda variabile, nell'ordine scelto, è x_3, il cui costo unitario è 2: essa ha un peso minore della capacità residua, e si pone quindi $x_3 = 1$; la capacità residua dello zaino scende di conseguenza a 5;

3. la terza variabile esaminata è x_4, il cui costo unitario è 5/3: anche essa ha un peso minore della capacità residua, e si pone quindi $x_4 = 1$ cosicché lo zaino rimane con una capacità residua di 2 unità;

4. la quarta variabile considerata è x_1, il cui costo unitario è 7/5: essa ha però peso 5, superiore alla capacità residua 2 dello zaino, e pertanto si pone $X_1 = 0$.

5. la quinta variabile, nell'ordine, è x_6, che ha costo unitario 1: la variabile ha peso 1, inferiore alla capacità residua, pertanto si pone $x_6 = 1$ e lo zaino rimane con una capacità residua di 1 unità;

6. l'ultima variabile considerata è x_2: tale variabile ha un peso (5) superiore alla capacità residua (1), e quindi si pone $x_2 = 0$.

La soluzione ottenuta è allora [0, 0, 1, 1, 1, 1], con costo 14 e peso totale 7: è facile vedere che questa soluzione non è ottima, dato che la soluzione [1, 0, 1, 0, 1, 0], con peso totale 8, ha un costo di 15. Pertanto, l'algoritmo descritto non garantisce l'ottimalità; tuttavia ha il vantaggio di essere di semplice implementazione e computazionalmente molto efficiente, essendo la sua complessità lineare se gli oggetti sono forniti in input già ordinati per costo unitario decrescente.

A.4.1.2 L'Algoritmo di Kruskal

Questo algoritmo per il problema dell'albero di copertura di costo minimo di un grafo connesso costruisce una soluzione inserendo gli archi uno per volta in ordine di costo non decrescente (i meno costosi prima), e verificando ogni volta che il sottografo corrispondente sia privo di cicli. L'algoritmo inizializza l'insieme T degli archi appartenenti all'albero come l'insieme vuoto, e poi scorre la lista degli archi in ordine di costo non decrescente. L'arco (i, j) di volta in volta selezionato viene accettato se il sottografo avente tutti gli archi in $T \cup \{(i, j)\}$ è privo di cicli; in questo caso (i, j) viene aggiunto a T, altrimenti viene scartato e si passa al successivo nell'ordinamento. L'algoritmo termina quando T contiene esattamente $n - 1$ archi.

A differenza del caso precedente, si può dimostrare che la soluzione generata dall'algoritmo è ottima: l'algoritmo è esatto, ed può essere implementato in modo da avere complessità polinomiale $O(m \log n)$.

A.4.1.3 Algoritmo "Nearest Neighbour"

Questo algoritmo per il problema del commesso viaggiatore costruisce una soluzione scegliendo ad ogni passo come prossima tappa la località più vicina

a quella in cui il commesso si trova attualmente. L'algoritmo inizializza l'insieme P degli archi appartenenti al cammino come l'insieme vuoto, e definisce come nodo "corrente" il nodo iniziale (1). Ad ogni iterazione, poi, esamina il nodo corrente i e tutti gli archi che lo uniscono a nodi che non sono ancora toccati dal cammino parziale P: tra di essi seleziona l'arco (i, j) di lunghezza minima, lo aggiunge a P e definisce j come nuovo nodo corrente. L'algoritmo termina quando tutti i nodi sono toccati da P, inserendo l'arco di ritorno dall'ultimo nodo al nodo 1.

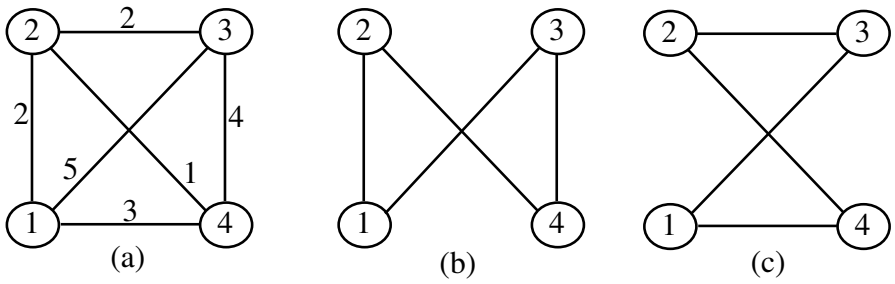

Figura A.1: Un'istanza del problema del commesso viaggiatore

Come nel caso del problema dello zaino, l'algoritmo greedy non è esatto (del resto, entrambi i problemi sono NP-ardui): questo può essere facilmente verificato mediante l'istanza rappresentata in figura A.1(a). L'algoritmo "Nearest Neighbour", partendo dal nodo 1, produce il ciclo rappresentato in figura A.1(b), con lunghezza 12, che è peggiore del ciclo ottimo rappresentato in figura A.1(c), che ha costo 11. Del resto, ancora una volta il costo computazionale della procedura è molto basso, essendo lineare nel numero degli archi del grafo ($O(n^2)$).

A.4.1.4 Uno schema generale

L'analisi dei tre esempi precedenti suggerisce uno schema generale di algoritmo greedy, adatto a tutti quei casi in cui:

- le soluzioni ammissibili del problema possono essere rappresentate come una famiglia $F \subset 2^E$ di sottoinsiemi di un dato insieme "base" $E = \{e_1, e_2, \ldots, e_n\}$;
- dati due sottoinsiemi ammissibili $A \in F$ e $B \in F$, se $A \subset B$ allora B è una soluzione preferibile ad A.

Lo schema generale è descritto nella seguente procedura:

```
Procedure Greedy:
    begin
        S := ∅; R := ∅; Q := E;
        repeat
            e := Best(Q); Q := Q \ {e};
            if S ∪ {e} ∈ F
                then S := S ∪ {e}
                else R := R ∪ {e}
        until Q = ∅
    end.
```

Procedura 1.1: Algoritmo *Greedy*

Nella procedura, S è l'insieme degli elementi di E che sono stati inseriti nella soluzione (parziale) corrente, R è l'insieme degli elementi per i quali si è già deciso che non faranno parte della soluzione finale, e Q è l'insieme degli elementi ancora da esaminare. La sottoprocedura *Best* fornisce il miglior elemento di E tra quelli ancora in Q sulla base di un prefissato criterio, ad esempio l'elemento di costo minimo nel caso di problemi di minimo.

Analizziamo ora la complessità della procedura *Greedy* dipende fondamentalmente dal costo di controllare se $S \cup \{e\} \in F$, che denoteremo con $k(n)$, e dalla complessità della procedura *Best*. Se supponiamo che gli elementi di E vengano ordinati all'inizio, e che quindi ad ogni passo *Best* fornisca semplicemente il primo fra i rimanenti, la complessità della procedura *Greedy* è $O(n(\log n + k(n)))$.

Esercizio A.1 *Sia dato un grafo $G = (N, A)$; si proponga, fornendone una descrizione formale, un algoritmo greedy per la colorazione dei suoi nodi con il minimo numero di colori, con il vincolo che due nodi adiacenti non abbiano mai lo stesso colore.*

A.4.2 Algoritmi di ricerca locale

Gli algoritmi di ricerca locale determinano una soluzione ammissibile di partenza x^0 come "soluzione corrente"; ad ogni passo, poi, esaminano le soluzioni "vicine" a quella corrente, e se ne trovano una "migliore" (tipicamente, con miglior valore della funzione obiettivo) la selezionano come nuova soluzione corrente. Viene così generata una sequenza di soluzioni ammissibili $\{x^0, x^1, \ldots, x^k, \ldots\}$; l'algoritmo si ferma quando la soluzione corrente viene riconosciuta come un *ottimo locale*, ossia quando nessuna delle soluzioni "vicine" è migliore di quella corrente.

Elemento caratterizzante di un algoritmo di questo tipo è la definizione di "vicinanza" tra le soluzioni; se F è l'insieme ammissibile del problema in

esame, viene definita una *funzione intorno* $I : F \to 2^F$ che possiede le due seguenti proprietà:

1) $x \in F \Rightarrow x \in I(x)$;

2) $x, y \in F \Rightarrow$ esiste un insieme finito $\{z^0, z^1, \ldots, z^p\} \subseteq F$ tale che $z^0 = x$, $z^k \in I(z^{k-1})$ per $k = 1, 2, \ldots, p$, $z^p = y$.

L'insieme $I(x)$ è detto *intorno* di x; la proprietà 1) richiede che ogni soluzione appartenga all'intorno di se stessa, mentre la proprietà 2) richiede che sia possibile raggiungere in un numero finito di passi qualsiasi soluzione $y \in F$ a partire da qualsiasi altra soluzione $x \in F$ muovendosi sempre nell'intorno della soluzione corrente. Questo è necessario se si vuole che l'algoritmo abbia la possibilità di determinare la soluzione ottima per qualunque scelta del punto iniziale x^0.

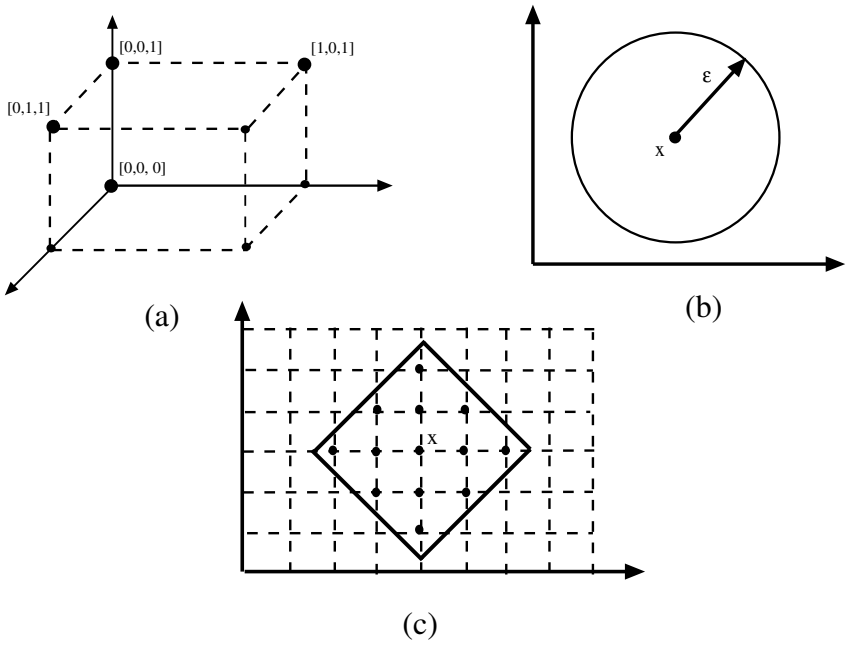

Figura A.2: Alcuni esempi di funzioni intorno

Alcuni esempi di funzioni intorno sono mostrati in figura A.2, in particolare

- figura A.2(a): $F \subseteq \{0,1\}^n$, $I(x) = \{y \in F : \sum_i |x_i - y_i| \leq 1\}$ (è mostrato l'intorno di [0,0,1]);
- figura A.2(b): $F \subseteq \mathbb{R}^n$, $I_\varepsilon(x) = \{y \in F : \|x - y\| \leq \varepsilon\}$ (Intorno Euclideo);

- figura A.2(c): $F \subseteq \mathbb{Z}^n$, $I(x) = \{y \in F : \sum_i |x_i - y_i| \leq 2\}$ (i punti evidenziati costituiscono l'intorno di x).

I tre intorni precedenti sono molto generali; usualmente, gli intorni utilizzati negli algoritmi di ricerca locale sono più specifici per il problema trattato. Inoltre, spesso la definizione di intorno è fornita in modo implicito, ovvero definendo una serie di operazioni che trasformano una soluzione ammissibile del problema in un'altra soluzione ammissibile. Un esempio di intorno con queste caratteristiche è quello definito dalle operazioni di "2-scambio" per il problema del commesso viaggiatore.

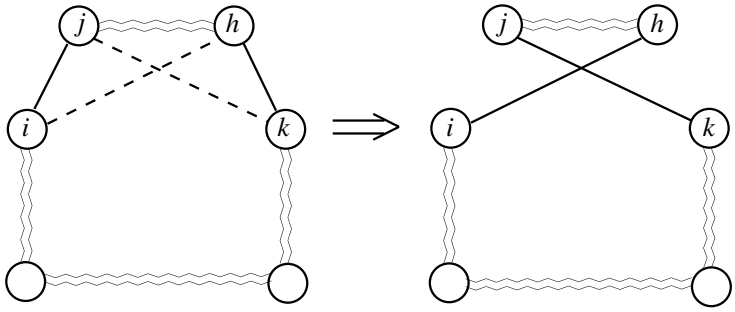

Figura A.3: Un "2-scambio" per il TSP

Nel TSP, una soluzione ammissibile x è un ciclo Hamiltoniano: l'intorno $I(x)$ basato sui "2-scambi" contiene tutti i cicli Hamiltoniani che si possono ottenere da x selezionando due archi (i, j) ed (h, k) non consecutivi del ciclo e sostituendoli con gli archi (i, h) e (j, k). Questa operazione, detta di "2-scambio", è esemplificata in figura A.3. Analogamente si possono definire intorni basati sul "k-scambio" che coinvolgono k archi del ciclo.

Opportunamente definita la funzione intorno, un algoritmo di ricerca locale può essere schematizzato come segue:

Procedure *Ricerca_Locale*(F, c, x):
  begin
    $x :=$ *Ammissibile*(F);
    while $\sigma(x) \neq x$ **do** $x := \sigma(x)$
  end.

Procedura 1.2: Algoritmo di Ricerca Locale

L'algoritmo necessita di una soluzione ammissibile da cui partire: spesso, tale soluzione è costruita usando un algoritmo greedy. σ è una trasformazione che, data una soluzione x^k della sequenza generata, fornisce la soluzione

successiva $x^{k+1} = \sigma(x^k)$. La tipica implementazione di σ, basata su una qualche funzione intorno I, è

$$\sigma_I(x) = \text{argmin}\{c(y) : y \in I(x)\},$$

dove "argmin" fornisce uno degli elementi di $I(x)$ aventi costo minimo. In questo modo, ad ogni passo dell'algoritmo viene risolto un problema di ottimo ristretto all'intorno considerato; alternativamente si può definire una trasformazione che per ogni x fornisca un qualsiasi elemento $y \in I(x)$, con $c(y) < c(x)$, se un tale elemento esiste, oppure x stesso se un tale elemento non esiste. Si ha $x = \sigma(x)$ quando nell'intorno non esiste nessuna soluzione migliore di x, ossia x è un *ottimo locale* per la funzione intorno I.

La complessità di un algoritmo di ricerca locale dipende fondamentalmente dalla complessità della trasformazione σ, che a sua volta dipende tipicamente dalla dimensione e dalla struttura della funzione intorno I. La dimensione dell'intorno $I(x)$ è anche collegata alla qualità degli ottimi locali che si determinano: intorni "più grandi" tipicamente forniscono ottimi locali di migliore qualità. Esiste quindi un "trade-off" tra complessità della trasformazione σ e qualità delle soluzioni determinate che deve essere accuratamente valutato.

Nel caso del TSP, ad esempio, la cardinalità di $I(x)$ per l'intorno basato sui "2-scambi" è $O(n^2)$: è quindi possibile enumerare in un tempo relativamente breve tutte le soluzioni dell'intorno e scegliere la migliore. Non è difficile vedere che la complessità di calcolare σ_I quando I è definita mediante operazioni di "k-scambio" cresce grosso modo come $k!$: quindi, per valori di k superiori a 2 o 3 determinare la migliore delle soluzioni in $I(x)$ può diventare molto oneroso dal punto di vista computazionale. D'altra parte, gli ottimi locali che si trovano usando operazioni di "3-scambio" sono usualmente migliori di quelli che si trovano usando operazioni di "2-scambio", e così via. Per questo si usano schemi di ricerca locale in cui la dimensione dell'intorno (il numero k di scambi) varia dinamicamente, e nei quali, quando k è "grande", ci si accontenta di determinare una qualunque soluzione nell'intorno che migliori il valore della funzione obiettivo, senza necessariamente richiedere che sia la migliore.

Esercizio A.2 *Si consideri il seguente problema di "clustering": ripartire gli m vettori reali ad n componenti $y_1, y_2, \ldots, y_m$ in tre sottoinsiemi in modo che sia minima la somma delle distanze fra vettori appartenenti allo stesso sottoinsieme. Descrivere un algoritmo "greedy" per trovare una soluzione ammissibile, ed un algoritmo di ricerca locale per la determinazione di un ottimo locale.*

A.4.3 Algoritmi enumerativi

Come abbiamo visto, né gli algoritmi greedy né quelli basati sulla ricerca locale sono in grado, in molti casi, di garantire l'ottimalità della soluzione

trovata. Nel caso in cui non esistano algoritmi greedy o di ricerca locale che garantiscano di determinare una soluzione ottima, e qualora si ritenga realmente importante determinare una soluzione ottima del problema, è necessario ricorrere ad algoritmi enumerativi.

Per descrivere i concetti base degli algoritmi enumerativi ci aiuteremo con un esempio, basato sul seguente problema dello zaino:

$$\begin{array}{lrrrrl} \text{Max} & 4x_1 & +x_2 & +3x_3 & +x_4 & \\ & 5x_1 & +4x_2 & +3x_3 & +3x_4 & \leq 8 \\ & x_1, & x_2, & x_3, & x_4 & \in \{0,1\} \end{array}$$

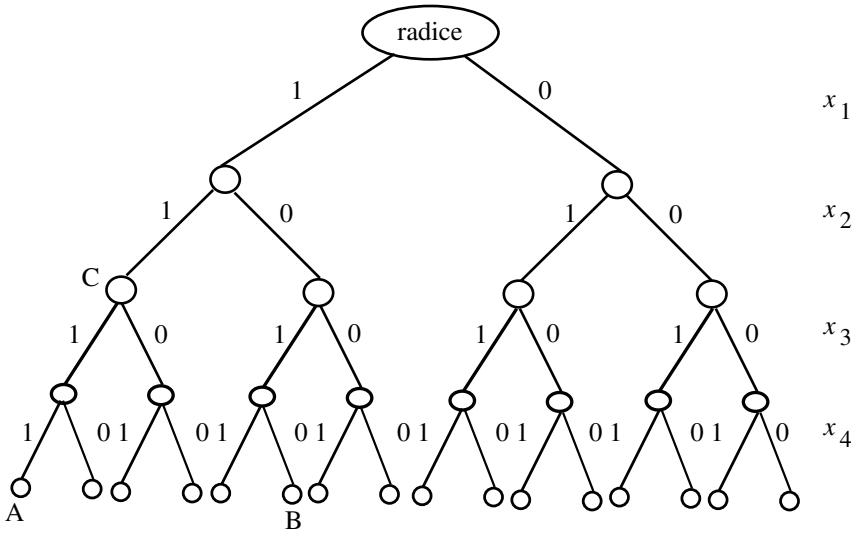

Figura A.4: L'albero delle decisioni per un'istanza del KP

L'insieme delle soluzioni di questo problema può essere *enumerato* utilizzando *l'albero delle decisioni* rappresentato in figura A.4. Al livello i dell'albero delle decisioni, viene fatta una scelta sul valore da assegnare alla variabile x_i (o alla i-esima variabile secondo un determinato ordine): i nodi del livello i sono 2^i. I cammini dalla radice alle foglie individuano le attribuzioni di valori $\{0,1\}$ alle variabili e quindi le soluzioni, non necessariamente ammissibili, per il problema dato: ad esempio, il cammino dalla radice alla foglia A individua una soluzione non ammissibile del problema, mentre il cammino dalla radice alla foglia B individua una soluzione ammissibile con valore 7.

Gli algoritmi enumerativi effettuano un'esplorazione dell'albero delle decisioni fino alla determinazione di un nodo foglia che corrisponde ad una soluzione ottima, o meglio, fino a che non sia garantito che la migliore soluzione trovata tra i nodi foglia raggiunti sia proprio una soluzione ottima. Dato che il numero di nodi dell'albero delle decisioni cresce in modo esponen-

ziale con la dimensione del problema, si cerca di effettuare la visita dell'albero in modo da esaminare "esplicitamente" il numero più piccolo possibile di nodi. Per questo è importante la capacità di riconoscere precocemente quando, proseguendo l'esame di un cammino oltre un certo nodo, non sia più possibile trovare soluzioni interessanti: interi sottoalberi possono così essere scartati e considerati *visitati implicitamente*. Ad esempio, nel problema in esame è inutile generare ed esplorare il sottoalbero di radice C in quanto la soluzione parziale corrispondente a C ($x_1 = x_2 = 1$) risulta non ammissibile qualunque sia il valore attribuito a x_3 e x_4: si parla allora di *algoritmi di enumerazione implicita*.

Tipicamente, il modo per determinare se un certo sottoalbero può essere scartato è quello di calcolare *valutazioni inferiori* (nel caso di problemi di minimo) sul valore della soluzione ottima del sottoproblema corrispondente a quel nodo; questo viene fatto risolvendo rilassamenti del problema originale. Se la valutazione inferiore corrispondente ad un certo nodo (il valore della soluzione ottima del rilassamento) è maggiore od uguale al valore della migliore soluzione del problema originario correntemente disponibile, allora il sottoalbero può non essere esaminato: è facile vedere, infatti, che nessuna soluzione ottima del problema può corrispondere ad una foglia nel sottoalbero.

Ritornando all'esempio del problema del commesso viaggiatore, abbiamo visto che il problema dell'albero di copertura di costo minimo ne costituisce un rilassamento. Con riferimento all'istanza rappresentata in figura A.1(a), si supponga di aver determinato la soluzione ammissibile di figura A.1(b) mediante l'algoritmo "Nearest Neighbour", e di voler determinare se tale soluzione sia o meno ottima mediante un algoritmo enumerativo che usi MST come rilassamento. L'algoritmo potrebbe fare i seguenti passi:

1. Il MST del grafo originario (al nodo radice) è {(1,2), (2,3), (2,4)} con costo 5: questa valutazione inferiore non è superiore al valore della soluzione ammissibile che abbiamo a disposizione (7), e quindi dobbiamo esaminare il nodo. Supponiamo di selezionare la variabile x_{24} come quella da fissare.

2. Fissando $x_{24} = 0$, l'MST diventa {(1,2), (2,3), (1,4)} con costo 7, uguale a quello della soluzione ammissibile disponibile, e quindi il nodo può essere eliminato: nel sottoalbero dell'albero delle decisioni che ha quel nodo come radice non esistono certamente soluzioni ammissibili per il TSP con valore minore di 7.

3. Fissando $x_{24} = 1$, l'MST resta quello del nodo radice, e quindi dobbiamo selezionare un'altra variabile da fissare: supponiamo di selezionare x_{23}.

4. Fissando $x_{23} = 0$, l'MST contenente l'arco $(2, 4)$ diventa {(1,2), (2,4),

(3,4)} con costo 7: come nel caso precedente, possiamo evitare di esaminare ulteriormente il sottoalbero con radice in questo nodo.

5. Fissando $x_{23} = 1$, l'MST resta quello del nodo radice, e quindi dobbiamo selezionare un'altra variabile da fissare: supponiamo di selezionare x_{12}.

6. Fissando $x_{12} = 0$, l'MST contenente gli arco $(2, 3)$ e $(2, 4)$ è {(1,4), (2,4), (2,3)} con costo 6, ed è anche un cammino hamiltoniano: abbiamo ottenuto una soluzione migliore di quella iniziale, che quindi diviene l'ottimo candidato del problema. Siccome la soluzione del rilassamento (MST) in questo nodo dell'albero delle decisioni è ammissibile per il problema originario (TSP), il nodo può essere chiuso in quanto abbiamo già determinato la soluzione ammissibile migliore per tutto il sottoalbero.

7. Fissando $x_{12} = 1$ troviamo tre archi della soluzione parziale che incidono nel nodo 2: la soluzione parziale è quindi non ammissibile per il TSP, ed il nodo puo' essere chiuso.

8. Visto che non sono rimasti altri nodi da esaminare, l'algoritmo termina.

La parte dell'albero delle decisioni esplorata dall'algoritmo è mostrata in figura A.5: si osservi che solamente 7 dei $2^6 = 64$ nodi dell'intero albero sono stati effettivamente visitati. In particolare, il solo fatto di "chiudere" il nodo corrispondente a $x_{24} = 0$ ha consentito di visitare implicitamente la metà dei nodi dell'albero delle decisioni.

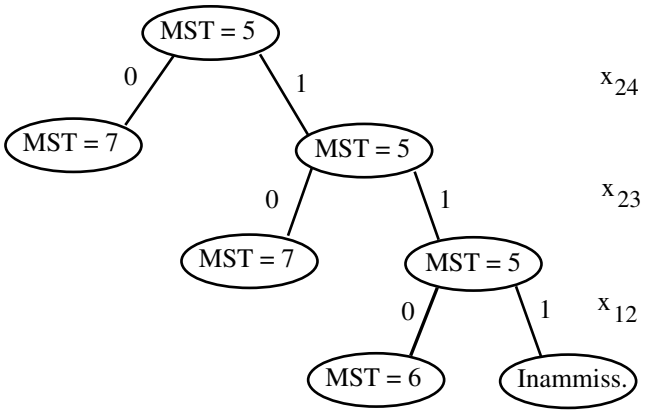

Figura A.5: L'algoritmo di enumerazione implicita applicato al TSP

L'efficienza degli algoritmi di enumerazione implicita dipende da diversi fattori, tra cui: il modo con cui è definito l'albero delle decisioni (ad uno

stesso problema possono essere associati diversi alberi delle decisioni), il metodo utilizzato per ottenere approssimazioni inferiori, l'algoritmo euristico utilizzato per ottenere soluzioni ammissibili, le regole per la selezione della variabile da fissare e le regole per la visita dell'albero delle decisioni (quale dei diversi nodi ancora disponibili esaminare per primo). Questo giustifica in parte l'interesse di sviluppare algoritmi efficienti per problemi "facili", sui quali ci soffermeremo a lungo in queste note: tali algoritmi forniscono, tra l'altro, modi efficienti per calcolare approssimazioni inferiori e superiori all'interno di algoritmi di enumerazione implicita per problemi combinatori.

Comunque, anche usando la migliore tra le strategie di ricerca, non si può mai escludere l'eventualità di dovere esaminare tutti i nodi dell'albero delle decisioni, o comunque una frazione consistente di essi.

Esercizio A.3 *È possibile dimostrare che il rilassamento continuo del problema dello zaino, ossia il problema*

$$(\underline{KP}) \qquad \begin{array}{lll} Max & \sum_{i=1}^{n} c_i x_i & \\ & \sum_{i=1}^{n} a_i x_i & \leq b \\ & x_i \in \{0,1\} & i = 1, 2, \ldots, n. \end{array}$$

si può risolvere facilmente mediante il seguente algoritmo: si ordinano gli oggetti per costo unitario decrescente, si inizializza l'insieme I degli oggetti selezionati (ossia degli indici delle variabili i con $x_i = 1$) all'insieme vuoto e si iniziano ad inserire oggetti (indici di variabili) in I finchè c'è spazio, come nell'euristica greedy per costi unitari non decrescenti. Quando però si raggiunge il primo oggetto h (nell'ordine dato) per cui la capacità residua dello zaino non è più sufficiente, cioè $b - \sum_{i \in I} a_i < a_h$, si pone

$$x_h = \frac{b - \sum_{i \in I} a_i}{a_h}$$

e $x_i = 0$ per $i \notin I \cup \{h\}$. Si risolva nuovamente l'istanza di KP data con un algoritmo enumerativo in cui $\underline{KP}$, risolto mediante questo algoritmo, fornisce la valutazione superiore per KP e l'euristica per costi unitari decrescenti fornisce la valutazione inferiore in ciascun nodo dell'albero delle decisioni.