Capitolo 5

Algoritmi euristici

Dato che molti problemi di *OC* sono "difficili", è spesso necessario sviluppare algoritmi *euristici*, ossia algoritmi che non garantiscono di ottenere la soluzione ottima, ma in generale sono in grado di fornire una "buona" soluzione ammissibile per il problema. Normalmente gli algoritmi euristici hanno una bassa complessità, ma in alcuni casi, per problemi di grandi dimensioni e struttura complessa, può essere necessario sviluppare algoritmi euristici sofisticati e di alta complessità. Inoltre, è possibile, in generale, che un algoritmo euristico "fallisca" e non sia in grado di determinare nessuna soluzione ammissibile del problema, pur senza essere in grado di dimostrare che non ne esistono.

La costruzione di algoritmi euristici efficaci richiede un'attenta analisi del problema da risolvere volta ad individuarne la "struttura", ossia le caratteristiche specifiche utili, ed una buona conoscenza delle principali tecniche algoritmiche disponibili. Infatti, anche se ogni problema ha le sue caratteristiche specifiche, esistono un certo numero di tecniche generali che possono essere applicate, in modi diversi, a moltissimi problemi, producendo *classi* di algoritmi di ottimizzazione ben definite. In questo Capitolo ci soffermeremo su due tra le principali tecniche algoritmiche utili per la realizzazione di algoritmi euristici per problemi di *OC*: gli algoritmi *greedy* e quelli di *ricerca locale*.

Queste tecniche algoritmiche non esauriscono certamente lo spettro delle euristiche possibili, per quanto forniscano una buona base di partenza per l'analisi e la caratterizzazione di moltissimi approcci. In particolare, vale la pena sottolineare qui che l'enfasi sulla "struttura" del problema di ottimizzazione è comune anche alle tecniche utilizzate per la costruzione di valutazioni superiori sul valore ottimo della funzione obiettivo, che saranno esaminate nel Capitolo 6. Questo fa sì che spesso una stessa struttura del problema venga utilizzata sia per realizzare euristiche che per determinare valutazioni superiori; si può così avere una "collaborazione" tra euristiche e rilassamenti, come nei casi delle *tecniche di arrotondamento* e delle *euristiche Lagrangiane*, che saranno discusse nel Capitolo 6.

Comunque, esempi di situazioni in cui la computazione di una valutazione superiore è parte integrante di—o comuque guida—un approccio euristico saranno presentate già in questo Capitolo. Per contro, le sopracitate tecniche di arrotondamento ed euristiche Lagrangiane sono spesso classificabili come euristiche greedy o di ricerca locale che sfruttano informazione generata dalla computazione di una valutazione superiore. Pertanto risulta ragionevole concentrarsi inizialmente su queste due grandi classi di approcci.

5.1 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, in generale non garantiscono l'ottimalità, ed a volte neppure l'ammissibilità, della soluzione trovata.

La definizione che abbiamo dato di algoritmo greedy è molto generale, e quindi possono essere ricondotti a questa categoria algoritmi anche all'apparenza molto diversi tra loro. È comunque possibile, a titolo di esemplificazione, costruire uno schema generale di algoritmo greedy, adatto a tutti quei

casi in cui l'insieme ammissibile può essere rappresentato come una famiglia $F \subset 2^E$ di sottoinsiemi di un dato insieme "base" E.

```
procedure Greedy( E , F , S )  {
  S := ∅; Q = E;
  do { e = Best( Q ); Q = Q \ { e };
       if( S ∪ { e } ∈ F ) then S = S ∪ { e }
       } while( Q ≠ ∅ and not Maximal( S ) )
  }
```

Procedura 5.1: Algoritmo *Greedy*

Nella procedura, S è l'insieme degli elementi di E che sono stati inseriti nella soluzione (parziale) corrente, e Q è l'insieme degli elementi di E ancora da esaminare: per tutti gli elementi in $E \setminus (S \cup Q)$ si è già deciso che *non* faranno parte della soluzione finale. La sottoprocedura $Maximal$ ritorna *vero* se non è più possibile aggiungere elementi alla soluzione S, per cui l'algoritmo può terminare senza esaminare gli elementi eventualmente ancora presenti in Q. In molti casi, le soluzioni ammissibili del problema sono solamente gli elementi "massimali" di F: quindi, se l'algoritmo greedy termina avendo esaurito gli elementi di Q senza che $Maximal$ ritorni *vero*, allora l'algoritmo "fallisce", ossia non è in grado di determinare una soluzione ammissibile del problema. 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.

5.1.1 Esempi di algoritmi greedy

È immediato verificare che l'algoritmo di Kruskal (si veda il pragrafo B.4.1) per (MST) ricade nello schema generale appena proposto: E è l'insieme degli archi del grafo, e S è la soluzione parziale corrente, ossia un sottografo privo di cicli. La procedura $Best$ determina semplicemente l'arco di costo minimo tra quelli non ancora considerati: ciò viene fatto semplicemente scorrendo la lista degli archi in ordine di costo non decrescente. Il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che l'arco $e = (i, j)$ selezionato non formi un ciclo nel sottografo individuato da S. La procedura $Maximal$ ritorna *vero* quando S contiene esattamente $n-1$ archi, ossia è un albero di copertura: se il grafo non è connesso, $Maximal$ non ritorna mai *vero* e l'algoritmo "fallisce", ossia non è in grado di determinare un albero di copertura (semplicemente perchè non ne esistono). Ricordiamo che l'algoritmo di Kruskal ha complessità $O(m \log n)$, essenzialmente dovuta all'ordinamento degli archi per costo non decrescente. Come abbiamo visto (e rivedremo, cf. §5.1.3), per il caso di (MST) si può dimostrare che la soluzione generata dall'algoritmo greedy è ottima (l'algoritmo è *esatto*). Vediamo adesso altri esempi di algoritmi greedy per problemi di OC che risultano invece essere algoritmi euristici.

5.1.1.1 Il problema dello zaino

Si consideri il problema dello zaino (KP) definito al paragrafo 1.2.2.1. Un semplice algoritmo greedy per questo problema consiste nell costruire una soluzione inserendo per primi nello zaino gli oggetti "più promettenti", ossia quelli che con maggiore probabilità appartengono ad una soluzione ottima, secondo un qualche criterio euristico. L'algoritmo inizializza l'insieme S degli oggetti selezionati come l'insieme vuoto, e poi scorre la lista degli oggetti ordinati secondo il criterio euristico: l'oggetto a_h di volta in volta selezionato viene accettato se la capacità residua dello zaino è sufficiente, cioè se $b - \sum_{i \in S} a_i \geq a_h$; in questo caso l'oggetto a_h viene aggiunto ad S, 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. È immediato verificare che anche questo algoritmo ricade nello schema generale. E è l'insieme degli oggetti tra i quali scegliere, la procedura $Best$ determina l'oggetto migliore secondo il criterio euristico, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che la capacità residua dello zaino sia sufficiente ad accogliere il nuovo oggetto, e la procedura $Maximal$ ritorna *vero* quando la capacità residua dello zaino è zero. In questo caso le soluzioni sono ammissibili anche se l'algoritmo termina senza che $Maximal$ ritorni *vero* (ciò non sarebbe più vero qualora il problema richiedesse di determinare un insieme di elementi di peso *esattamente uguale* a b).

Si noti che quello appena proposto non è *un* di algoritmo, ma piuttosto una *famiglia* di algoritmi greedy per (KP) che si differenziano per il criterio con cui vengono selezionati gli oggetti. Consideriamo ad esempio i seguenti tre criteri:

- *pesi non decrescenti*: $a_1 \leq a_2 \leq \ldots \leq a_n$;
- *costi non crescenti*: $c_1 \geq c_2 \geq \ldots \geq c_n$;
- *costi unitari non crescenti*: $c_1/a_1 \geq c_2/a_2 \geq \ldots \geq c_n/a_n$.

Ciascuno dei tre criteri è "ragionevole": col primo si cercano di inserire nello zaino "molti" oggetti, col secondo quelli di costo maggiore, col terzo quelli che hanno il maggiore costo per unità di spazio occupato. Nessuno dei tre criteri di ordinamento degli elementi domina gli altri; tuttavia, è facile rendersi conto del fatto che l'ultimo (costi unitari non crescenti) è il pi ragionevole, ed in generale quello che fornisce risultati migliori. Chiamiamo CUD l'algoritmo greedy per (KP) che utilizzi il criterio dei costi unitari non crescenti.

Esercizio 5.1 *Per ciascuno dei tre criteri, costruire un esempio in cui la soluzione fornita da esso domina le soluzioni fornite dagli altri.*

Esempio 5.1: Esecuzione di Greedy-CUD per (KP)
Consideriamo la seguente istanza del problema dello zaino:

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

In questo caso, l'algoritmo CUD esegue 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.

Esercizio 5.2 *Si esegua l'algoritmo greedy per (KP) sull'istanza dell'esempio precedente utilizzando gli altri due criteri euristici per la selezione dell'elemento.*

L'algoritmo greedy ha complessità $O(n \log n)$, con ciascuno dei tre criteri sopra descritti, essenzialmente dovuta all'ordinamento degli oggetti; se gli oggetti sono forniti in input già ordinati secondo il criterio selezionato, la complessità dell'algoritmo è lineare.

5.1.1.2 Il problema dell'assegnamento di costo minimo

Si consideri il problema accoppiamento di massima cardinalità e costo minimo discusso al paragrafo 3.5.2. Un algoritmo greedy per questo problema può essere facilmente costruito nel modo seguente. Si ordinano gli archi per costo non crescente, e si inizializza un vettore di booleani, con una posizione per ogni nodo, a $falso$, per indicare che nessun nodo è assegnato. Si esaminano poi gli archi nell'ordine dato; se entrambe gli estremi dell'arco non sono assegnati l'arco viene aggiunto all'accoppiamento e si pongono a *vero* le posizioni corrispondenti nell'array, per segnalare che i nodi sono adesso accoppiati.

L'algoritmo termina quando non ci sono più nodi da accoppiare, oppure quando sono stati esaminati tutti gli archi. Questo algoritmo ricade ovviamente nello schema generale. E è l'insieme A degli archi del grafo, la procedura *Best* determina l'arco di costo minimo tra quelli non ancora esaminati, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che entrambe gli estremi dell'arco non siano già assegnati, la procedura *Maximal* ritorna *vero* se tutti i nodi sono accoppiati. In particolare, se si desidera un assegnamento (accoppiamento perfetto) l'algoritmo "fallisce" se termina avendo esaminato tutti gli archi senza che *Maximal* abbia ritornato *vero*. È facile dimostrare che questo algoritmo non costruisce necessariamente una soluzione ottima del problema; in particolare può non essere neanche in grado di determinare un assegnamento nel grafo anche se ne esiste uno.

Esercizio 5.3 *Si fornisca un esempio che dimostri l'affermazione precedente (suggerimento: si veda il paragrafo 5.1.3).*

In compenso l'algoritmo ha complessità $O(m \log n)$, essenzialmente dovuta all'ordinamento degli archi, sostanzialmente inferiore a quella $O(mn^2)$ degli algoritmi esatti discussi nel paragrafo 3.5.2; quindi questo algoritmo potrebbe ad esempio risultare utile per ottenere rapidamente "buoni" assegnamenti per problemi di grandissima dimensione. Si noti inoltre che questo algoritmo, a differenza di quelli del paragrafo 3.5.2, è adatto anche al caso in cui il grafo G non sia bipartito. Sono comunque stati proposti algoritmi esatti per il caso bipartito con complessità inferiore a $O(mn^2)$ (uno dei quali basato sull'idea di "trasformare" la soluzione ottenuta dall'algoritmo greedy), ed anche algoritmi esatti per il caso non bipartito; per ulteriori dettagli si rimanda alla letteratura citata.

5.1.1.3 Il problema del commesso viaggiatore

Si consideri il problema del commesso viaggiatore (TSP) definito al paragrafo 1.2.2.3. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. L'algoritmo inizializza l'insieme S degli archi appartenenti al ciclo 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 ciclo parziale S: tra di essi seleziona l'arco (i, j) "più promettente" secondo un certo criterio euristico, lo aggiunge a S e definisce j come nuovo nodo corrente. L'algoritmo termina quando tutti i nodi sono toccati da S, inserendo l'arco di ritorno dall'ultimo nodo al nodo 1. Anche questo algoritmo ricade nello schema generale: E è l'insieme degli archi del grafo, *Best* determina l'arco più promettente, tra tutti quelli che escono dal nodo corrente, secondo il criterio euristico, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che il nodo terminale j dell'arco (i, j) non sia già stato visitato, e *Maximal* ritorna *vero* quando tutti i nodi sono stati visitati.

Anche in questo caso abbiamo una famiglia di algoritmi che si differenziano per il criterio utilizzato per determinare l'arco "più promettente". Un criterio molto intuitivo è semplicemente quello di selezionare l'arco di lunghezza minima: ciò corrisponde a scegliere ad ogni passo, come prossima tappa, la località più vicina a quella in cui il commesso si trova attualmente, il che è noto come algoritmo "Nearest Neighbour".

Esempio 5.2: Esempio di esecuzione di "Nearest Neighbour"
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 5.1(a). L'algoritmo "Nearest Neighbour", partendo dal nodo 1, produce il ciclo rappresentato in figura 5.1(b), con lunghezza 12, che è peggiore del ciclo ottimo rappresentato in figura 5.1(c), che ha costo 11.

Si noti che l'algoritmo costruisce sicuramente un ciclo hamiltoniano se il grafo G è completo, mentre può "fallire" nel caso in cui G non sia completo: ciò accade se tutti gli archi uscenti dal nodo corrente i portano a nodi già visitati dal ciclo parziale S, oppure se i è l'ultimo dei nodi da visitare ma non esiste l'arco fino al nodo iniziale. Quindi, a differenza del problema dello zaino, l'algoritmo greedy non solo non garantisce di determinare una soluzione ottima, ma può non essere in grado di produrre neanche una qualsiasi soluzione ammissibile. Ciò non deve stupire: mentre per il problema dello zaino è immediato costruire una soluzione ammissibile (lo zaino vuoto), il problema di decidere se esiste un

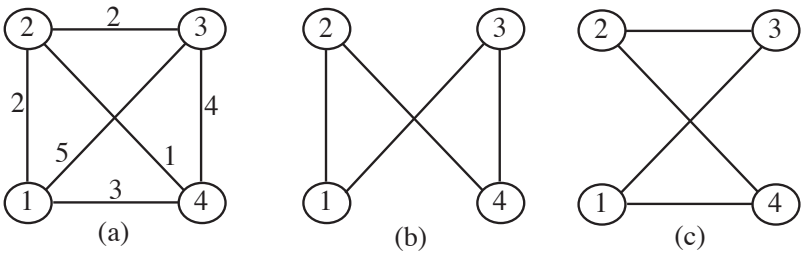

Figura 5.1: Un'istanza del problema del commesso viaggiatore

ciclo Hamiltoniano in un grafo non completo è $\mathcal{NP}$-arduo, e quindi non è pensabile che un algoritmo greedy sia in grado di risolverlo.

Sono comunque stati proposti molti altri criteri di selezione del nodo successivo che possono rivelarsi può efficienti in pratica. Ad esempio, quando il grafo G può essere rappresentato su un piano (ad esempio, quando i nodi corrispondono effettivamente a località geografiche) un criterio interessante è quello che seleziona j in modo tale che il segmento (arco) (i, j) formi il più piccolo angolo possibile con il segmento (arco) (h, i), dove h è il nodo visitato immediatamente prima di i nel ciclo parziale (ossia $(h, i) \in S$). Pur senza entrare nei dettagli, segnaliamo il fatto che questo criterio è motivato da alcune proprietà della frontiera dell'inviluppo convesso di un insieme di punti del piano e dalle relazioni che esistono tra l'inviluppo convesso dei punti che rappresentano i nodi ed il ciclo Hamiltoniano di costo minimo; quindi, per quanto il criterio sia semplice da capire e da implementare, la sua ideazione è stata resa possibile solamente da uno studio accurato delle proprietà (di alcuni casi rilevanti) del problema in oggetto. Si noti come, comunque, ancora una volta, il costo computazionale della procedura è molto basso, essendo lineare nel numero degli archi del grafo ($O(n^2)$).

5.1.1.4 Ordinamento di lavori su macchine con minimizzazione del tempo di completamento

Si consideri il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS) definito al paragrafo 1.2.9.1. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. All'inizio, nessun lavoro è assegnato e tutte le macchine sono scariche, ossia $N(j) = \emptyset$ per $j = 1, \ldots, m$. Ad ogni iterazione si seleziona uno dei lavori i ancora da assegnare, secondo un certo criterio euristico, e lo si assegna alla macchina "più scarica", ossia a quella con tempo di completamento $D(j) = \sum_{i \in N(j)} d_i$ (relativo alla soluzione parziale corrente) più basso; in caso di parità, si sceglie una qualunque delle macchine col tempo di completamento corrente minimo. L'algoritmo termina quando tutti i lavori sono stati assegnati. Anche questo algoritmo può essere fatto ricadere nello schema generale: E è l'insieme delle coppie (i, j) con $i = 1, \ldots, n$ e $j = 1, \ldots, m$, ossia dei possibili assegnamenti di lavori a macchine. *Best* seleziona prima un lavoro i non ancora assegnato, secondo il criterio euristico, e poi la macchina (più scarica) a cui assegnarlo. Il controllo "$S \cup \{ e \} \in F$" non esiste, in quanto dato un lavoro non assegnato è sempre possibile assegnarlo a qualsiasi macchina. Anche la procedura *Maximal* non fa nulla, in quanto le soluzioni sono ammissibili se e solo se tutti i lavori sono stati assegnati, ossia Q è vuoto.

Anche in questo caso quella appena proposta è una famiglia di algoritmi greedy, detti *list scheduling*, che si differenziano solamente per il criterio utilizzato per determinare il prossimo lavoro i da assegnare. Tra tutti i possibili criteri, due sono quelli più significativi:

- *SPT (Shortest Processing Time)*: i lavori vengono assegnati in ordine non decrescente dei tempi di esecuzione (quelli più "corti" per primi);

- *LPT (Longest Processing Time)*: i lavori vengono assegnati in ordine non crescente dei tempi di esecuzione (quelli più "lunghi" per primi).

Esempio 5.3: Esecuzione di algoritmi list scheduling
Nelle figure 5.2 e 5.3 sono riportati i risultati ottenuti con i due criteri su due esempi; osserviamo che nel primo caso LPT fornisce una soluzione ottima, mentre nel secondo caso nessuno dei due algoritmi riesce a raggiungere l'ottimo.

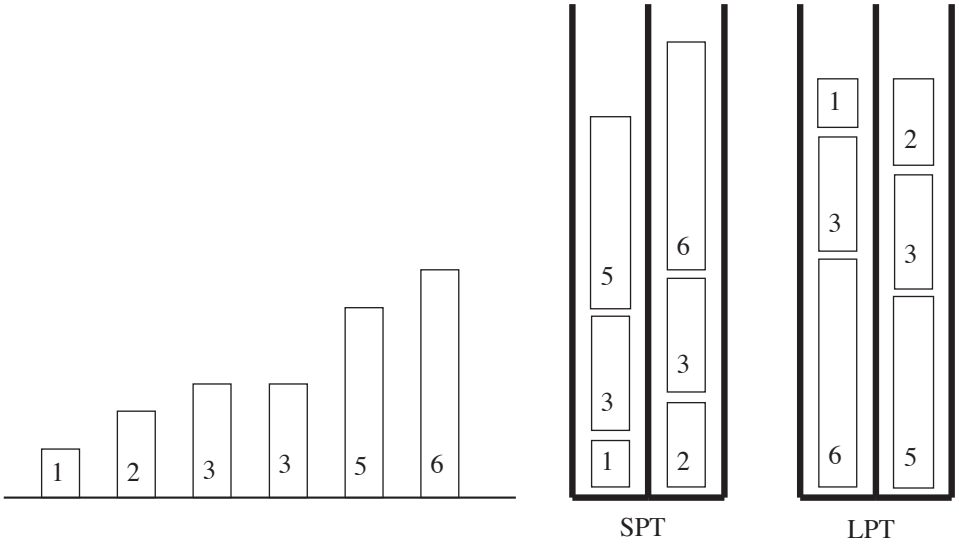

Figura 5.2: Un'istanza del problema (MMMS)

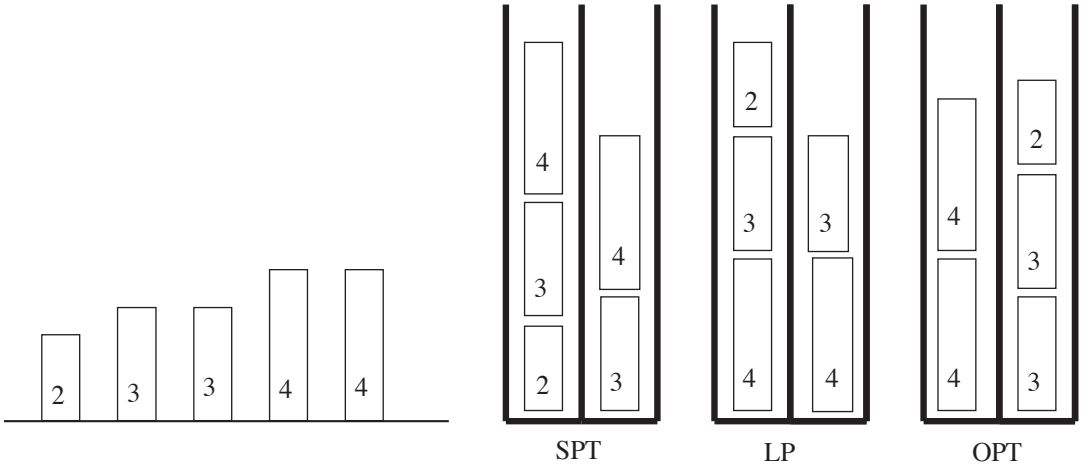

Figura 5.3: Un'istanza del problema (MMMS)

Come vedremo in seguito, LPT ha "migliori proprietà" di SPT, e di fatto in generale risulta più efficace, ossia produce soluzioni di migliore qualità. Per entrambe i criteri di ordinamento, comunque, l'algoritmo greedy è facile da implementare e risulta molto efficiente in pratica.

Esercizio 5.4 *Si discuta la complessità dell'algoritmo greedy per (MMMS); come cambia il risultato se i lavori sono forniti in input già ordinati secondo il criterio selezionato?*

5.1.1.5 Ordinamento di lavori su macchine con minimizzazione del numero delle macchine

Si consideri la variante di (MMMS) in cui sono dati i tempi di inizio e di fine di ciascun lavoro e si vuole minimizzare il numero di macchine utilizzate, ossia il problema (MCMS) definito al paragrafo 1.2.4.2. Una famiglia di algoritmi greedy per questo problema può essere costruita in modo analogo a quello visto nel paragrafo precedente. All'inizio, nessun lavoro è assegnato e nessuna macchina è

utilizzata. Ad ogni iterazione si seleziona uno dei lavori i ancora da assegnare, secondo un certo criterio euristico, e si scorre la lista delle macchine già utilizzate, secondo un altro opportuno criterio euristico, assegnando il lavoro alla prima macchina sulla quale è possibile eseguirlo: se non è possibile eseguire i su nessuna delle macchine già utilizzate, lo si assegna ad una nuova macchina fino a quel momento scarica, che viene quindi aggiunta all'insieme di macchine utilizzate. Questo algoritmo può essere fatto ricadere nello schema generale analogamente a quanto visto nel paragrafo precedente; si noti che, ancora una volta, quella appena proposta è una famiglia di algoritmi greedy che si differenziano per i criteri utilizzati per determinare il prossimo lavoro i da assegnare e la macchina j già utilizzata a cui assegnarlo (se possibile).

Esercizio 5.5 *Si propongano almeno due criteri diversi per la selezione del prossimo lavoro i da assegnare e almeno due criteri diversi per la selezione della macchina j già utilizzata a cui assegnarlo, discutendo i possibili vantaggi e svantaggi di ciascuna combinazione.*

Tra tutti gli algoritmi greedy appartenenti allo schema appena introdotto ne esiste uno che costruisce certamente una soluzione ottima per il problema. Si consideri infatti l'algoritmo in cui il lavoro da assegnare viene selezionato in ordine di tempo di inizio t_i non crescente; in altre parole vengono assegnati per primi i lavori che iniziano prima. Supponiamo di ordinare le macchine in un qualsiasi ordine, e di esaminarle per l'inserzione del lavoro corrente sempre nell'ordinamento dato; se il lavoro non può essere inserito nelle macchine attualmente utilizzate, sarà attivata la successiva macchina nell'ordinamento. Questo algoritmo costruisce un assegnamento che utilizza sempre il minor numero possibile di macchine. Sia k l'indice dell'ultima macchina attivata nel corso dell'algoritmo, ossia il numero di macchine utilizzate nella soluzione costruita, e sia i il primo lavoro assegnato a quella macchina. Infatti, si consideri lo stato delle altre macchine "attive" al momento in cui viene esaminato i: a ciascuna di esse è stato assegnato un lavoro h incompatibile con i, ossia tale che $[t_i, t_i + d_i] \cap [t_h, t_h + d_h] \neq \emptyset$. Ma, per via della strategia di selezione dei lavori, ciascun lavoro h assegnato ad una macchina in quel momento ha $t_h \leq t_i$: non è quindi possibile che il lavoro i sia incompatibile con il lavoro h perchè $t_i < t_h \leq t_i + d_i \leq t_h + d_h$, ne consegue che deve risultare $t_i \in [t_h, t_h + d_h]$. Di consegeunza, nell'istante t_i devono necessariamente essere in esecuzione k lavori: i, che inizia in quel momento, e gli altri $k-1$ che occupano le altre macchine nell'assegnamento (parziale) costruito fino a quel momento. Di conseguenza sono necessarie almeno k macchine per eseguire tutti i lavori: poichè la soluzione costruita ne usa esattamente k, essa è ottima.

Esercizio 5.6 *Si discuta come implementare l'algoritmo greedy "ottimo" per (MCMS) in modo da ottenere una bassa complessità computazionale.*

Questo esempio mostra come la conoscenza di un algoritmo greedy per un certo problema di *OC* possa suggerire algoritmi greedy analoghi per problemi di *OC* "simili", ma anche come problemi apparentemente "simili" possano in effetti risultare molto diversi in termini di facilità di soluzione.

Esercizio 5.7 *Si proponga, fornendone una descrizione formale, un algoritmo greedy per il problema (GC) di colorare i nodi di un grafo $G = (N, A)$ con il minimo numero di colori con il vincolo che due nodi adiacenti non abbiano mai lo stesso colore (si veda il paragrafo 1.2.4.3). Si discuta sotto quali ipotesi sul grafo si può costruire un algoritmo greedy equivalente a quello "ottimo" per (MCMS) che riporti sicuramente una soluzione ottima per (GC). Si discuta poi come modificare l'algoritmo per il caso più generale in cui ad ogni nodo i devono essere assegnati esattamente n_i colori diversi e/o i colori assegnati a nodi adiacenti devono essere "distanti" di almeno una soglia δ fissata.*

5.1.1.6 Il problema di copertura

Si consideri il problema di copertura (PC) definito al paragrafo 1.2.5. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. L'algoritmo inizializza l'insieme S dei sottoinsiemi selezionati come l'insieme vuoto. Ad ogni iterazione, seleziona uno dei sottoinsiemi F_j ancora da esaminare, secondo un certo criterio euristico e lo aggiunge a S se il nuovo sottoinsieme

"copre" almeno un elemento di N che non era "coperto" dalla soluzione parziale precedente, ossia se $F_j \not\subset F_S = \cup_{F_i \in S} F_i$. L'algoritmo termina quando Q è vuoto oppure quando $F_S = N$, ossia tutti gli elementi di N sono "coperti" da S.

Esercizio 5.8 *Si mostri che l'algoritmo appena descritto ricade nello schema generale di algoritmo greedy.*

Anche in questo caso, quella appena proposta è una famiglia di algoritmi greedy che si differenziano per il criterio utilizzato per determinare il sottoinsieme "più promettente". Consideriamo ad esempio i seguenti tre criteri:

- *costi non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con costo c_j più basso;
- *costi unitari non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con "costo unitatio" $c_j/|F_j|$ più basso, ossia si tiene conto del numero di oggetti che un dato sottoinsieme può coprire;
- *costi unitari attualizzati non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con "costo unitatio attualizzato" $c_j/|F_j \setminus F_S|$ più basso, ossia si tiene conto del numero di oggetti che un dato sottoinsieme copre e che non sono già coperti dalla soluzione parziale S.

Non è sorprendente che il terzo criterio, quello dei costi unitari attualizzati, sia in pratica spesso migliore degli altri due, in quanto è l'unico dei tre che utilizza informazione sulla soluzione corrente S per decidere il prossimo sottoinsieme da esaminare. Si noti anche, però, che tale criterio è potenzialmente più costoso da implementare: infatti per i primi due criteri è possibile ordinare gli oggetti all'inizio e poi semplicemente scorrere la lista ordinata, il che ha complessità $(m \log m)$, mentre nel terzo l'ordinamento deve essere ricalcolato ogniqualvolta un oggetto viene inserito in S, e quindi F_S aumenta.

5.1.1.7 Il problema (CMST)

Si consideri il problema dell'albero di copertura di costo minimo capacitato (CMST) definito nell'esempio 4.1. Per il problema "più semplice" dell'albero di copertura di costo minimo (MST) conosciamo algoritmi greedy esatti, ossia in grado determinare una soluzione ottima. Chiaramente, tali algoritmi ottengono la soluzione ottima per (CMST) se la capacità Q degli archi uscenti dalla radice, ossia il massimo peso dei sottoalberi, è "grande"; per questo, è ragionevole cercare di costruire algoritmi greedy per (CMST) che si ispirino agli algoritmi per (MST). Nell'algoritmo di Kruskal, ad esempio, si pone $S = \emptyset$ e si esaminano gli archi in ordine di costo non decrescente: l'arco (i, j) esaminato viene aggiunto ad S se non crea cicli con gli archi già presenti in S, ossia se collega due diverse componenti connesse del grafo (N, S). Non è difficile modificare l'algoritmo in modo tale che tenga conto delle capacità: basta mantenere il peso (somma del peso dei nodi) di ogni componente connessa, e non accettare l'arco (i, j) se la sua inserzione in S causa l'unione di due componenti connesse la cui somma dei pesi è maggiore di Q. Questo ovviamente non si applica ad archi di tipo (r, i), ossia che collegano una componente connessa alla radice. Se esistono archi da r a tutti gli altri nodi allora l'algoritmo così modificato costruisce sicuramente una soluzione ammissibile per il problema, altrimenti può "fallire". È possibile implementare il controllo sul peso delle componenti connesse, mediante opportune strutture dati, in modo da non aumentare la complessità dell'algoritmo di Kruskal.

Questo esempio mostra come la conoscenza di algoritmi per un dato problema possa essere utilizzata per guidare la realizzazione di approcci per problemi simili ma più complessi. Naturalmente, non sempre è facile adattare gli algoritmi noti per risolvere problemi più complessi in modo naturale: ad esempio, adattare l'algoritmo di Prim al (CMST) è molto meno immediato.

5.1.2 Algoritmi greedy con garanzia sulle prestazioni

Nel paragrafo precedente abbiamo visto un certo numero di algoritmi greedy per alcuni problemi di ottimizzazione rilevanti. Una volta che un algoritmo sia stato ideato ed implementato, si pone il

problema di valutarne in qualche modo l'efficacia, ossia la capacità di fornire effettivamente "buone" soluzioni con errore relativo basso. Ci sono essenzialmente due modi per studiare l'efficacia di un algoritmo euristico:

- *sperimentale*: si seleziona un sottoinsieme "rilevante" di istanze del problema, ad esempio tali che siano rappresentative delle istanze da risolvere in una o più applicazioni pratiche, si esegue l'algoritmo su quelle istanze misurando l'errore relativo ottenuto e poi se ne esaminano le caratteristiche statistiche (media, massimo, minimo, varianza, ecc.); si noti che per fare questo è necessario essere in grado di calcolare il valore ottimo della funzione obiettivo, o una sua buona approssimazione, per le istanze test;

- *teorico*: si dimostrano matematicamente relazioni che forniscono valutazioni sul massimo errore compiuto dall'algoritmo quando applicato ad istanze con caratteristiche date; più raramente è possibile valutare anche altre caratteristiche statistiche dell'errore compiuto (media ecc.)

Le due metodologie di valutazione non sono alternative ma complementari: lo studio teorico risulta in genere meno accurato nel valutare l'errore compiuto nelle istanze reali, ma fornisce valutazioni generali valide per grandi classi di istanze e aiuta a comprendere meglio il comportamento dell'algoritmo, eventualmente suggerendo come modificarlo per renderlo più efficace; d'altro canto, lo studio sperimentale permette di valutare esattamente l'errore compiuto sulle istanze utilizzate e di estrapolare con ragionevole accuratezza l'errore che ci si può attendere su istanze simili, ma non fornisce alcuna garanzia, specialmente per istanze con caratteristiche diverse da quelle effettivamente testate. Ciò è del tutto analogo alla differenza tra lo studio della complessità al caso pessimo di un algoritmo e la verifica sperimentale dell'efficienza in pratica della sua implementazione.

In questo paragrafo mostreremo alcuni semplici risultati relativi alla valutazione dell'errore di alcuni algoritmi greedy, e come sia possibile costruire algoritmi in modo da garantire che abbiano determinate prestazioni. Si noti che lo studio teorico delle prestazioni degli algoritmi è in principio possibile anche per altre classi di algoritmi euristici, come quelli di ricerca locale che presenteremo nel paragrafo 5.2; essendo però in generale piuttosto complesso, risulta spesso troppo difficile per algoritmi che non abbiano una struttura molte semplice, ed è per questo che è principalmente effettuato su algoritmi di tipo greedy.

Dato un algoritmo euristico $\mathcal{A}$, si distinguono due diversi modi per valutare l'errore compiuto da $\mathcal{A}$: *a priori* ed *a posteriori*. La valutazione a posteriori dell'errore viene effettuata dopo che l'algoritmo ha determinato la soluzione corrispondente ad una singola istanza I, e permette di valutare l'errore compiuto per quella particolare istanza. La valutazione a priori invece fornisce una stime del massimo errore compiuto da $\mathcal{A}$ per *qualsiasi* istanza I, ed è quindi disponibile prima che l'istanza venga risolta. La valutazione a priori è quindi più generale, ma siccome è una valutazione al caso pessimo è anche usualmente meno precisa di quella a posteriori.

5.1.2.1 L'algoritmo CUD per (KP)

Per valutare l'errore compiuto dall'algoritmo CUD, è necessario per prima cosa ottenere una valutazione superiore sul valore ottimo della funzione obiettivo del problema. Per fare ciò consideriamo il rilassamento continuo del problema dello zaino, ossia il problema

$$(\underline{KP}) \qquad \max \left\{ \sum_{i=1}^{n} c_i x_i \ : \ \sum_{i=1}^{n} a_i x_i \leq b \ , \ x \in [0,1]^n \right\} .$$

È possibile verificare che una soluzione x^* ottima per $(\underline{KP})$ può essere costruita nel modo seguente: si ordinano gli oggetti per costo unitario non crescente, si inizializza l'insieme S degli oggetti selezionati (ossia degli indici delle variabili i con $x_i^* = 1$) all'insieme vuoto e si iniziano ad inserire oggetti in S (porre variabili a 1) finchè è possibile, esattamente come nell'algoritmo CUD. Quando però si raggiunge il primo oggetto h (nell'ordine dato) per cui la capacità residua dello zaino non è più sufficiente, cioè

$b - \sum_{i \in S} a_i < a_h$, si pone $x_h^* = (b - \sum_{i \in S} a_i)/a_h$ e $x_i^* = 0$ per $i \notin S \cup \{ h \}$ (si noti che h è ben definito in quanto per ipotesi risulta $b < \sum_i a_i$). Infatti, si consideri il duale di ($\underline{KP}$):

$$(\underline{DKP}) \qquad \min \left\{ yb + \sum_{i=1}^n w_i \ : \ ya_i + w_i \geq c_i \quad i = 1, \ldots, n \ , \ y \geq 0 \ , \ w_i \geq 0 \quad i = 1, \ldots, n \right\} .$$

Dalla teoria della *PL* sappiamo che una coppia di soluzioni $\bar{x}$ e $(\bar{y}, \bar{w})$ è ottima per ($\underline{KP}$) e ($\underline{DKP}$) se e solo è ammissibile e rispetta le condizioni degli scarti complementari, che in questo caso sono

$$\bar{y}(b - \sum_{i=1}^n a_i \bar{x}_i) = 0 ,$$

$$\bar{w}_i(1 - \bar{x}_i) = 0 \quad , \quad \bar{x}_i(\bar{y} a_i + \bar{w}_i - c_i) = 0 \qquad i = 1, \ldots, n \ .$$

È immediato verificare che la soluzione duale

$$y^* = c_h/a_h \qquad w_i^* = \begin{cases} c_i - y^* a_i & \text{se } i < h, \\ 0 & \text{altrimenti} \end{cases} :$$

è ammissibile, in quanto, essendo gli oggetti ordinati per costi unitari non crescenti, si ha $w_i^* = c_i/a_i - c_h/a_h \geq 0$. È anche facile controllare che (y^*, w^*) verifica le condizioni degli scarti complementari con x^*: per la prima condizione si ha $\sum_{i=1}^n a_i x_i^* = b$, per la seconda condizione si ha che $w_i^* > 0$ solo per per gli indici $i < h$ per cui $x_i^* = 1$, per la terza condizione si ha che $x_i^* > 0$ al più per gli indici $i \leq h$ (può essere $x_h^* = 0$) per cui $y^* a_i + w_i^* = c_i$.

Possiamo adesso procedere a valutare *a posteriori* l'errore relativo commesso da CUD. Infatti si ha $z(P) \leq \sum_{i=1}^n c_i x_i^* = \sum_{i<h} c_i + c_h(b - \sum_{i<h} a_i)/a_h$ e $\sum_{i<h} c_i \leq z_{CUD} \leq z(P)$, da cui

$$R_{CUD} = \frac{z(P) - z_{CUD}}{z(P)} \leq \frac{c_h \frac{b - \sum_{i<h} a_i}{a_h}}{\sum_{i<h} c_i} \leq \frac{c_h}{\sum_{i<h} c_i} \ .$$

Si noti che se $\sum_{i<h} a_i = b$, ossia l'ultimo oggetto (nell'ordine dei costi unitari non crescenti) che entra interamente nello zaino ne satura la capacità, allora $R_{CUD} = 0$, ossia CUD determina una soluzione ottima del problema. Infatti, in questo caso si ha che la soluzione prodotta da CUD è esattamente x^*, in quanto x_h^* (l'unica componente di x^* che può assumere valore frazionario) vale 0. Quindi, la soluzione ottima del rilassamento continuo di (KP) ha valori interi, ossia è una soluzione ammissibile per (KP); il Lemma 4.1 garantisce quindi che x^* sia una soluzione ottima per (KP).

Esempio 5.4: Stime dell'errore per Greedy-CUD
Consideriamo la seguente istanza del problema dello zaino:

$$\begin{array}{rrcrcrcrcrc} \max & 11x_1 & + & 8x_2 & + & 7x_3 & + & 6x_4 & & & \\ & 5x_1 & + & 4x_2 & + & 4x_3 & + & 4x_4 & \leq & 12 & . \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & \in & \{0,1\} & \end{array}$$

Gli oggetti sono già ordinati per costo unitario non crescente. L'algoritmo CUD riporta la soluzione $S = \{1, 2\}$ di costo 19, mentre la soluzione ottima è $\{2, 3, 4\}$ di costo 21: l'errore relativo commesso da CUD in questo caso è $(21-19)/21 \approx 0.095$, ossia del 9.5%. La soluzione ottima di $\underline{KP}$ è $x^* = [\, 1 \, , \, 1 \, , \, 3/4 \, , \, 0 \,]$ di costo $97/4 (= 24 + 1/4)$; ad essa corrisponde infatti la soluzione duale $y^* = 7/4$, $w^* = [\, 9/4 \, , \, 1 \, , \, 0 \, , \, 0 \,]$, anch'essa di costo 97/4. Utilizzando questa valutazione superiore su $z(P)$ per stimare l'errore si ottiene $R_{CUD} \leq (97/4 - 19)/19 \approx 0.276$, ossia il 27.6%. Utilizzando la formula che dipende solamente dai costi e da h si ottiene $R_{CUD} \leq 7/(11 + 8) \approx 0.368$.

Esercizio 5.9 *Per i valori*

$$U' = \sum_{i<h} c_i + c_{h+1} \frac{b - \sum_{i<h} a_i}{a_{h+1}} \qquad e \qquad U'' = \sum_{i \leq h} c_i + c_{h-1} \frac{b - \sum_{i \leq h} a_i}{a_{h-1}} \ ,$$

dimostrare che vale la relazione

$$z(KP) \leq max\{ U' \, , \, U'' \} \leq c_h \frac{b - \sum_{i<h} a_i}{a_h} \ ,$$

ossia che il massimo tra U' ed U'' fornisce una valutazione superiore su $z(P)$ non peggiore di quella utilizzata per valutare R_{CUD} (suggerimento: U' ed U'' sono i valori dele soluzioni ottime di due opportuni problemi di LP *ottenuti supponendo che x_h sia rispettivamente* 0 *e* 1 *in una soluzione ottima di KP).*

5.1.2.2 Gli algoritmi SPT e LPT per (MMMS)

Come per il caso del problema dello zaino, per poter valutare l'errore compiuto dagli algoritmi SPT e LPT per (MMMS) dobbiamo innanzitutto determinare una valutazione—in questo caso inferiore—-sul valore ottimo della funzione obiettivo. Per (MMMS) questo è facile: infatti, si ha

$$z(MMMS) \geq L = \sum_{i=1}^{n} d_i/m$$

(nel caso migliore, i lavori sono perfettamente ripartiti tra le macchine che terminano tutte allo stesso momento, ossia si ha "speedup lineare").

Iniziamo l'analisi dimostrando una valutazione *a priori* che vale per qualsiasi algoritmo di tipo list scheduling, ossia indipendentemente dall'ordine con cui sono assegnati i lavori: in ogni caso, l'errore relativo compiuto non supera $(m-1)/m$. Si noti che l'errore peggiora all'aumetare del numero di macchine, tendendo a 1 (cioè al 100%) quando m cresce. Per dimostrare il risultato occorre definire alcune quantità critiche: in particolare, indichiamo con h il lavoro che termina per ultimo (l'ultimo lavoro eseguito sulla macchina che termina per ultima) e con $H = (\sum_{i=1}^{n} d_i - d_h)/m$: H è il minimo tempo possibile di completamento per le macchine se h fosse rimosso dalla lista dei lavori. Chiamiamo z_{LS} il valore della funzione obiettivo della soluuzione determinata da un generico algoritmo di list scheduling: l'osservazione fondamentale è che risulta $z_{LS} \leq H + d_h = L + d_h(m-1)/m$. Infatti, quando il lavoro h viene assegnato alla macchina "più scarica" rispetto alla soluzione corrente al momento in cui h è esaminato: è quindi ovvio che la situazione peggiore, quella alla quale corrisponde il maggior valore di z_{LS}, è quella in cui h è anche l'ultimo lavoro ad essere assegnato, e le macchine erano "perfettamente bilanciate" al momento in cui h è stato assegnato (si veda la figura 5.4). Da questo e da $z(MMMS) \geq L$ otteniamo

$$R_{LS} = \frac{z_{LS} - z(MMMS)}{z(MMMS)} \leq \frac{L + d_h(m-1)/m - L}{z(MMMS)}$$

Ma siccome chiaramente $z(MMMS) \geq d_h$ (almeno il tempo necessario a completare d_h deve passare) si ha $R_{LS} \leq (m-1)/m$.

Osserviamo che questa valutazione non è "ottimistica", ossia non è possibile migliorarla. Infatti, esiste almeno una classe di istanze ed un ordinamento, in particolare SPT, per cui si ottiene una soluzione che ha esattamente errore relativo $(m-1)/m$. Si consideri infatti l'istanza che contiene $(m-1)m$ lavori di lunghezza unitaria ed un solo lavoro di lunghezza m: l'algoritmo SPT assegna i lavori di lunghezza unitaria per primi, ottenendo $m-1$ macchine con tempo di completamento $m-1$ ed una singola macchina con tempo di completamento $2m-1$. In figura 5.4 è mostrato un esempio per il caso $m = 4$. Ma l'assegnamento ottimo consiste invece nell'assegnare ad una macchina l'unico lavoro di lunghezza m e distribuire poi uniformemente sulle altre i restanti $(m-1)m$: in questo caso tutte le macchine terminano in tempo m.

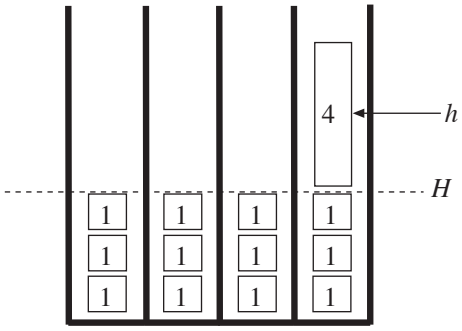

Figura 5.4: Valutazione dell'errore per (MMMS)

Si osservi che l'algoritmo LPT applicato all'istanza "critica" produce la soluzione ottima. In effetti, la valutazione precedente è valida per qualsiasi algoritmo di tipo list scheduling, ma se utilizziamo il particolare ordinamento LPT è possibile dimostrare una valutazione migliore, ossia che $R_{LPT} \leq (m-1)/3m$. Per semplicità di notazione ci limiteremo a considerare il caso $m = 2$ (per cui $R_{LPT} \leq 1/6$) e supporremo che i lavori siano forniti in input già ordinati per durata non crescente, ossia $d_1 \geq d_2 \geq \ldots \geq d_n$. La dimostrazione è per assurdo: supponiamo che esista un'istanza per cui l'errore relativo sia maggiore di 1/6. Consideriamo in particolare l'istanza I *con n minimo* per cui LPT ottiene un errore relativo maggiore di 1/6: in questa istanza si ha $h = n$, ossia l'ultimo lavoro completato è anche quello di durata minore. Infatti, se fosse $h < n$ potremmo eliminare tutti gli oggetti di indice maggiore di h, che chiaramente non contribuiscono alla determinazione del tempo di completamento

(essendo h quello che termina per ultimo): otterremmo così un'istanza I' con meno lavori e con errore relativo non minore di quello di I, contraddicendo l'ipotesi che I sia l'istanza con meno lavori che ha errore relativo maggiore di 1/6. Sia quindi z_{LPT} il tempo di completamento della soluzione ottenuta dall'algoritmo LPT sull'istanza I: per ipotesi abbiamo

$$R_{LPT} \geq \frac{z_{LPT} - z(MMMS)}{z(MMMS)} > \frac{1}{6}$$

da cui $z_{LPT} > 7z(MMMS)/6$. Siccome LPT è un particolare algoritmo di list scheduling, vale

$$z_{LPT} \leq H + d_h = L + d_h/2 \leq z(MMMS) + d_n/2$$

si ha quindi $7z(MMMS)/6 < z(MMMS) + d_n/2$, ossia $z(MMMS) < 3d_n$. Siccome d_n è il più corto dei lavori, dalla relazione precedente si ottiene che $n \leq 4$: qualsiasi istanza con almeno 5 lavori di lunghezza almeno d_n ha tempo di completamento non inferiore a $3d_n$. È però facile verificare che per istanze con non più di 4 lavori LPT fornisce la soluzione ottima, da cui l'assurdo.

Esercizio 5.10 *Si dimostri che qualsiasi istanza con al più 4 lavori viene sicuramente risolta all'ottimo da LPT (suggerimento: i casi con 1, 2 e 3 lavori sono banali; chiamando $a \geq b \geq c \geq d$ le lunghezze dei 4 lavori, i primi tre sono sempre assegnati da LPT nello stesso modo – a da solo sulla prima macchina, b e c insieme sulla seconda – mentre d viene assegnato in modo diverso a seconda che risulti $a \geq b + c$ oppure $a < b + c$).*

Esercizio 5.11 *Si estenda la dimostrazione della valutazione superiore dell'errore relativo di LPT al caso generale $m > 2$.*

5.1.2.3 Algoritmo "twice around MST" per il (TSP)

Consideriamo nuovamente il problema del commesso viaggiatore (TSP); è possibile costruire un'euristica greedy con errore relativo al caso pessimo pari ad 1 se il grafo G è completo ed i costi sugli archi sono non negativi e rispettano la *diseguaglianza triangolare*:

$$c_{ij} + c_{jh} \leq c_{ih} \qquad \forall\, i\,,\, j\,,\, h\ . \tag{5.1}$$

L'algoritmo si basa sulla seguente osservazione: il costo di un albero di copertura di costo minimo sul grafo G fornisce una valutazione inferiore del costo del ciclo hamiltoniano ottimo del grafo. Infatti, sia C^* il ciclo hamiltoniano di costo minimo del grafo, e consideriamo il cammino P^* ottenuto eliminando un qualsiasi arco da C^*: il costo di P^* è non superiore a quello di C^* e non inferiore a quello dell'albero di copertura di costo minimo T^*, essendo P^* un particolare albero di copertura per il grafo. L'albero di copertura di costo minimo T^* è efficientemente calcolabile con gli algoritmi visti nel paragrafo B.4: tale albero può essere trasformato in un cammino hamiltoniano che, se i costi soddisfano (5.1), non può avere costo maggiore di due volte il costo di T^*. La trasformazione è illustrata in figura 5.5: inizialmente si considera il grafo orientato ottenuto da T^* duplicando gli archi ed orientando ogni coppia degli archi così ottenuti nelle due direzioni possibili. Si ottiene così (figura 5.5(b)) un cammino hamiltiano orientato (non semplice) per il grafo, il cui costo (considerando i costi degli archi orientati pari a quelli degli archi non orientati originali) è pari a due volte il costo di T^*. Tale cammino può essere trasformato in un cammino hamiltoniano attraverso l'operazione di "scorciatoia" mostrata in figura 5.5(c). Partendo da un nodo qualsiasi, ad esempio il nodo 1, si inizia a percorrere il ciclo, marcando tutti i nodi visitati: se il successore j di un nodo i visitato per la prima volta è un nodo già visitato, si continua a percorrere il ciclo finché non si incontra un nodo h non ancora visitato. A quel punto, al posto dl cammino da i ad h sul ciclo hamiltoniano originale viene scelta la "scorciatoia" (i, h) ed il procedimento viene iterato. Quando viene visitato l'ultimo nodo, il ciclo si chiude aggiungendo l'arco fino al nodo 1 (ossia operando una scorciatoia dall'ultimo nodo visitato a 1). Da (5.1) segue immediatamente che ogni operazione di "scorciatoia" non incrementa la lunghezza del

ciclo hamiltoniano: di conseguenza, per il ciclo hamiltoniano H^* ottenuto al termine del procedimento si ha

$$C(T^*) \leq C(C^*) \leq C(H^*) \leq 2C(T^*)$$

e quindi l'errore relativo è non superiore a 1. Questo algoritmo prende, per ovvie ragioni, il nome di euristica "twice around MST".

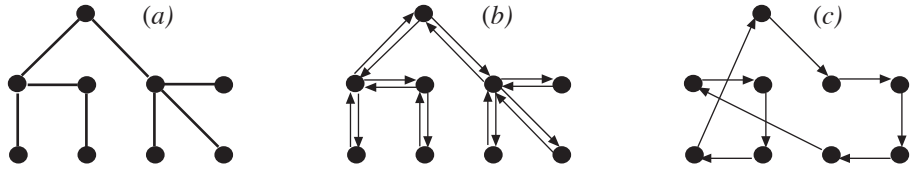

Figura 5.5: Funzionamento dell'euristica "twice around MST"

Esercizio 5.12 *Si proponga una descrizione formale, in pseudo-codice, dell'euristica "twice around MST".*

Esiste una versione di questa euristica, nota come *euristica di Christofides*, nella quale le operazioni di "scorciatoia" sono rimpiazzate dalla soluzione di un problema di assegnamento di costo minimo. Questo esempio mostra come la conoscenza di algoritmi per problemi "facili" possa essere utile per costruire approcci per problemi più "difficili".

5.1.2.4 Un algoritmo greedy per il Weighted Vertex Cover

Nei due esempi precedenti abbiamo prima introddoto gli algoritmi e poi, separatamente, valutato le loro prestazioni. È però possibile costruire algoritmi *facendosi guidare* dalle dimostrazioni di efficacia, ossia in modo tale che "naturalmente" se ne possano dimostrare buone proprietà di approssimazione. Esistono alcune tecniche generali per fare questo, una delle quali sarà illustrata per il problema del seguente esempio.

Esempio 5.5: Un problema di selezione di nodi
L'agenzia spionistica EMC, sempre un pò indietro rispetto alla concorrente CIA, ha deciso di dotarsi anch'essa di un sistema per monitorare tutte le comunicazioni di Internet. Per fare questo dispone di una mappa dei "backbones" di Internet attraverso un grafo (non orientato) $G = (V, E)$, in cui i nodi rappresentano i routers ed i lati rappresentano i collegamenti principali. L'agenzia dispone di apparecchiature che, se installate su un certo router, permettono di monitorare tutte le comunicazioni che transitano attraverso quel nodo. Installare l'apparecchiatura in un certo nodo i ha però un costo $c_i > 0$, dovuto in parte al costo dell'hardware ed in parte al costo di corrompere o intimidire i gestori del router per convincerli a permetterne l'installazione. L'agenzia dispone dei fondi necessari per installare le apparecchiature in tutti i nodi, ma ciò è evidentemente inutile: per poter monitorare tutte le comunicazioni, è sufficiente che per ogni lato $\{i, j\} \in E$ *almeno* uno dei nodi i e j sia monitorato. Per risparmiare soldi da poter inserire nei propri fondi neri, l'agenzia EMC deve quindi risolvere il seguente problema, detto di *Weighted Vertex Cover* (WVC): selezionare un sottoinsieme di nodi, $S \subseteq V$, di costo minimo (dove $C(S) = \sum_{i \in S} c_i$), che "copra" tutti i lati del grafo, ossia tale che per ogni $\{i, j\} \in E$ sia $i \in S$ oppure $j \in S$.

Un modello analitico per il problema è il seguente:

$$(WVC) \qquad \min \left\{ \sum_{i=1}^{n} c_i x_i \ : \ x_i + x_j \geq 1 \quad \{i, j\} \in E \ , \ x \in \mathbb{N}^n \right\} .$$

Chiaramente, qualsiasi soluzione ottima x^* del problema avrà solamente componenti 0 e 1, ossia $x^* \in \{0, 1\}^n$, anche se non sono presenti vincoli espliciti $x_i \leq 1$: infatti, per qualsiasi soluzione ammissibile $\bar{x}$ con $\bar{x}_i > 1$ è possibile costruire un'altra soluzione ammissibile x' identica a $\bar{x}$ tranne per il fatto che $x'_i = 1$, e siccome i costi sono positivi x' ha un costo minore di $\bar{x}$.

Vogliamo costruire un algoritmo greedy per (WVC) che abbia buone proprietà di approssimazione: come abbiamo visto nei casi precedenti, per stimare l'errore compiuto è necessario per prima cosa ottenere una valutazione (in questo caso inferiore) del valore ottimo della funzione obiettivo. Come

nel caso del problema dello zaino, otteniamo una tale valutazione considerando il *rilassamento continuo* di (WVC):

$$(\underline{WVC}) \qquad \min\left\{ \sum_{i=1}^{n} c_i x_i \ : \ x_i + x_j \geq 1 \quad \{i,j\} \in E \ , \ x \geq 0 \right\}$$

o, equivalentemente, il suo duale

$$(\underline{DWVC}) \qquad \max\left\{ \sum_{\{i,j\}\in E} y_{ij} \ : \ \sum_{\{i,j\}\in S(i)} y_{ij} \leq c_i \quad i \in V \ , \ y \geq 0 \right\} \ ,$$

dove $S(i)$ è l'insieme dei lati indicenti nel nodo i. Vogliamo costruire un algoritmo per (WVC) che costruisca contemporaneamente una soluzione ammissibile per (WVC) *intera*, e quindi ammissibile per (WVC), ed una soluzione ammissibile per (DWVC) che ci permetta di valutare la bontà della soluzione primale ottenuta. In effetti, ci serviremo della soluzione duale (parziale) per *guidare* la costruzione della soluzione primale: per questo, l'algoritmo viene detto *primale-duale*. Le condizioni degli scarti complementari per la coppia di problemi duali (WVC) e (DWVC) sono

$$x_i(c_i - \sum_{\{i,j\}\in S(i)} y_{ij}) = 0 \qquad i \in V \tag{5.2}$$

$$y_{ij}(x_i + x_j - 1) = 0 \qquad \{i,j\} \in E \ . \tag{5.3}$$

Se le soluzioni primale e duale ottenute dall'algoritmo rispettassero sia (5.2) che (5.3), allora avremmo determinato una coppia di soluzioni ottime per (WVC) e (DWVC); dato che la soluzione primale sarà costruita in modo da essere ammissibile anche per (WVC), si sarebbe quindi ottenuta una soluzione ottima per (WVC) (si veda il Lemma 4.1). Naturalmente, essendo (WVC) un problema $\mathcal{NP}$-arduo non è pensabile che ciò sia sempre possibile: per questo l'algoritmo si limiterà ad assicurare che sia verificata (5.2), mentre permetterà violazioni in (5.3).

```
Procedure Greedy-WVC( G , c , S ) {
  S = ∅; Q = E;
  foreach( {i,j} ∈ E ) do y_ij = 0;
  do { {i,j} = Next(Q); Q = Q \ { {i,j} };
       y_ij = min{ c_i − Σ_{i,h}∈S(i) y_ih , c_j − Σ_{j,h}∈S(j) y_jh };
       if( c_i = Σ_{i,h}∈S(i) y_ih ) then {
           foreach( {i,h} ∈ E ) do Q = Q \ { {i,h} };
           S = S ∪ {i};
           }
       if( c_j = Σ_{j,h}∈S(j) y_jh ) then {
           foreach( {j,h} ∈ E ) do Q = Q \ { {j,h} };
           S = S ∪ {j};
           }
       } while( Q ≠ ∅ );
  }
```

Procedura 5.2: Algoritmo *Greedy-WVC*

L'algoritmo mantiene in Q l'insieme dei lati non ancora "coperti" dalla soluzione corrente S, che inizialmente non contiene alcun nodo. Ad ogni passo seleziona un qualsiasi ;ato $\{i,j\}$ non ancora coperto ed aumenta il valore della corrispondente variabile y_{ij} al massimo valore possibile che non viola le condizioni (5.2) per i e j; questo valore è tale per cui, dopo l'aumento di y_{ij}, vale almeno una delle due condizioni

$$c_i = \sum_{\{i,h\}\in S(i)} y_{ih} \qquad \text{e} \qquad c_j = \sum_{\{j,h\}\in S(j)} y_{jh} \ .$$

Se vale la prima condizione i viene aggiunto ad S e tutti i lati incidenti in i (che sono a questo punto coperti da S) sono eliminati da Q; analogamente, se vale la seconda condizione j viene aggiunto ad S e tutti i lati incidenti in j sono eliminati da Q. È facile verificare che la soluzione y costruita dall'algoritmo è duale ammissibile ad ogni iterazione: lo è infatti sicuramente all'inizio (dato che i costi sono positivi), e non appena il vincolo duale corrispondente al nodo i diviene "attivo" tutti i lati incidenti in i vengono eliminati da Q, "congelando" i loro valori y_{ij} fino al termine dell'algoritmo e quindi assicurando che il vincolo resti verificato. A terminazione, la soluzione primale S "copre" tutti i nodi ($Q = \emptyset$), ed è quindi ammissibile.

Esempio 5.6: Algoritmo *Greedy-WVC*
Un esempio del funzionamento dell'algoritmo *Greedy-WVC* è mostrato in figura 5.6. L'istanza è mostrata in (a), con i costi indicati vicino a ciascun nodo.
La prima iterazione è mostrata in (b): viene selezionato il lato $\{1,2\}$, e si pone $y_{12} = 1$, in modo tale che $y_{12} + y_{16} + y_{14} = 1 + 0 + 0 = 1 = c_1$. Quindi, il nodo 1 viene inserito in S ed i lati $\{1,2\}$, $\{1,6\}$ e $\{1,4\}$ risultano quindi coperti.
La seconda iterazione è mostrata in (c): viene selezionato il lato $\{2,6\}$, e si pone $y_{26} = 2$; in questo modo risulta sia $y_{12} + y_{26} + y_{23} = 1 + 2 + 0 = 3 = c_2$ che $y_{16} + y_{26} + y_{36} + y_{46} + y_{56} = 0 + 2 + 0 + 0 + 0 = 2 = c_6$, e quindi sia il nodo 2 che il nodo 6 vengono aggiunti a S, coprendo i corrispondenti lati adiacenti.
La terza ed ultima iterazione è mostrata in (d): viene selezionato il lato $\{3,5\}$, e si pone $y_{35} = 4$, in modo che risulti $y_{35} + y_{45} + y_{56} = 4 + 0 + 0 = 4 = c_5$, e quindi che 5 sia aggiunto a S coprendo gli ultimi lati e determinando quindi la soluzione ammissibile $S = \{\,1\,,\,2\,,\,5\,,\,6\,\}$ d costo 10.

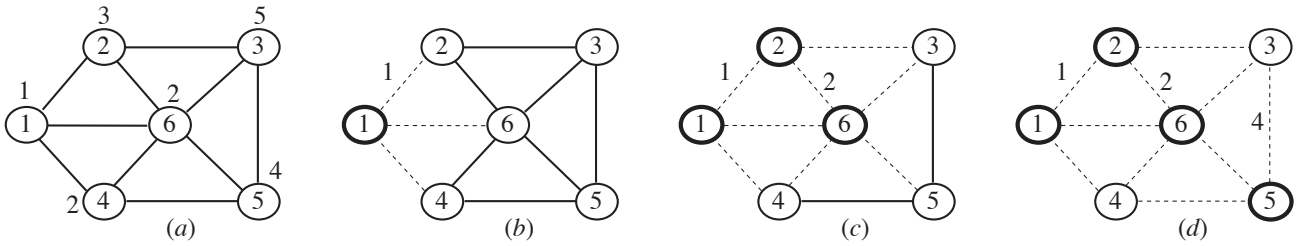

Figura 5.6: Esecuzione dell'algoritmo *Greedy-WVC*

Valutiamo adesso l'efficacia dell'algoritmo *Greedy-WVC*. L'osservazione fondamentale è che ad ogni passo dell'algoritmo si ha

$$c(S) = \sum_{i \in S} c_i \leq 2 \sum_{\{i,j\} \in E} y_{ij} \;;$$

infatti, quando il generico nodo i è stato inserito in S si aveva $c_i = \sum_{\{i,h\} \in S(i)} y_{ih}$ (e gli y_{ih} non sono più cambiato da quell'iterazione), ma ciascun y_{ij} può contribuire ad al più due sommatorie, quella corrispondente ad i e quella corrispondente a j. Di conseguenza si ha

$$\sum_{\{i,j\} \in E} y_{ij} \leq z(\underline{DWVC}) = z(\underline{WVC}) \leq z(WVC) \leq c(S) \leq 2 \sum_{\{i,j\} \in E} y_{ij}$$

da cui $R_{Greedy-WVC} \leq 1$; la soluzione ottenuta dall'algoritmo *Greedy-WVC* può costare al più il doppio della soluzione ottima. Quindi, *a priori* possiamo affermare che l'algoritmo *Greedy-WVC* compie al massimo un errore del 100%; questa valutazione può poi essere raffinata *a posteriori* esaminando i risultati per l'istanza specifica. Nel caso di figura 5.6(a) si ha ad esempio

$$\sum_{\{i,j\} \in E} y_{ij} = 7 \leq z(WVC) \leq c(S) = 10$$

da cui $R_{Greedy-WVC} \leq (10 - 7)/7 \approx 0.428$, ossia la soluzione ottenuta da *Greedy-WVC* è al più il 42.8% più costosa della soluzione ottima. Si noti che anche questa valutazione non è esatta: è possibile verificare (enumerando tutte le possibili soluzioni) che la soluzione ottenuta da *Greedy-WVC* è in effetti *ottima* per l'istanza in questione. La soluzione duale ottenuta non è però in grado di dimostrare l'ottimalità della soluzione primale.

L'idea alla base dell'algoritmo *Greedy-WVC*, ossia quella di costruire contemporaneamente sia una soluzione primale intera che una duale ammissibile per il duale del rilassamento continuo, è generale e può essere applicata per produrre algoritmi con garanzia sul massimo errore compiuto per molti altri problemi combinatori. Questa è comunque soltanto una delle molte tecniche possibili, per ulteriori dettagli ed approfondimenti si rimanda alla letterature citata.

Esercizio 5.13 *Il problema (WVC) è un caso particolare del problema di copertura (PC) in cui tutti gli insiemi F_j hanno esattamente due elementi. Si estenda quindi l'algoritmo primale-duale per (WVC) a (PC): che valutazione può essere data sull'errore commesso di tale algoritmo?*

Esercizio 5.14 *Si mostri che l'algoritmo CUD per il problema dello zaino può essere interpretato come un algoritmo primale-duale (suggerimento: per qualsiasi valore di $\bar{y}$, la migliore soluzione $\bar{w}$ duale ammissibile compatibile con quel valore di $\bar{y}$ si ottiene ponendo $\bar{w}_i = c_i/a_i - \bar{y}$ se $c_i/a_i > \bar{y}$ e $\bar{w}_i = 0$ altrimenti).*

5.1.3 Matroidi

Nei paragrafi precedenti abbiamo visto che per alcuni algoritmi greedy è possibile ricavare valutazioni, a priori o a posteriori, sull'errore compiuto, e che un algoritmo greedy può anche essere esatto. Sorge quindi spontanea la domanda: è possibile caratterizzare i problemi per i quali gli algoritmi greedy hanno errore nullo, ossia sono esatti? In effetti questo è possibile, sia pure per una classe particolare di problemi e di algoritmi greedy, ossia facendo un alcune ulteriori assunzioni sul problema e sull'algoritmo. La prima assunzione è la seguente:

- (E, F) sono un *sistema di insiemi indipendenti*, ossia $A \in F$ e $B \subseteq A$ implica $B \in F$;

Da questo deriva immediatamente che $\emptyset \in F$; inoltre, possiamo assumere senza perdita di generalità che sia $\{e\} \in F\ \forall e \in E$, in quanto se fosse $\{e\} \notin F$ per un qualche $e \in E$ allora nessuno dei sottoinsiemi in F potrebbe contenere e, e quindi potremmo rimuovere e da E (ciò è analogo a quanto già osservato per gli oggetti nel problema dello zaino). Dato un sistema di insiemi indipendenti (E, F), diciamo che S è *massimale* per E se $S \in F$ e $S \cup \{e\} \notin F$ per ogni $e \in E \setminus S$, ossia se non esiste nessun elemento di F che contiene strettamente S. Più in generale, S è massimale per $E' \subseteq E$ se $S \in F$, $S \subseteq E'$ e $S \cup \{e\} \notin F$ per ogni $e \in E' \setminus S$. La seconda assunzione è:

- a ciascun elemento $e_i \in E$ è associato un costo $c_e \geq 0$, ed il problema da risolvere è

$$\text{(MSII)} \qquad \max \left\{ c(S) = \textstyle\sum_{e \in S} c_e \ :\ S \text{ massimale per } E \right\} .$$

Si noti che, anche per via di questa assunzione, la teoria sviluppata in questo paragrafo non copre tutti i casi di algoritmi greedy che determinano l'ottimo di un problema di OC; infatti, il problema (MCMS), che è risolto all'ottimo da un algoritmo greedy (cf. §5.1.1.5), non ha una funzione obiettivo di questo tipo. Si consideri un grafo non orientato e connesso $G = (V, E)$: (E, F), dove F è la famiglia dei sottoinsiemi di lati che non inducono cicli su G, è chiaramente un sistema di insiemi indipendenti, e gli alberi di copertura per G sono chiaramente i suoi insiemi massimali. Quindi, (MST) è un problema di tipo (MSII). Si noti che la funzione obiettivo *non* è di questo tipo in alcuni dei problemi per i quali abbiamo presentato algoritmi greedy, come ad esempio (MMMS) e (MCMS). Per un problema nella forma (MSII) esiste un'implementazione "naturale" della procedura *Best*; la terza assunzione è infatti

- la sottoprocedura *Best* dell'algoritmo ritorna l'elemento $e \in Q$ di costo c_e massimo.

Si noti come molte delle regole *Best* che abbiamo discusso *non* rientrano in questa categoria, quali ad esempio quelle per (TSP) e due di quelle per (PC). L'algoritmo di Kruskal è un esempio di algoritmo greedy che rispetta le assunzioni (se invertiemo il segno dei costi o, alternativamente, vogliamo risolvere il problema dall'albero di copertura di costo massimo) e che determina la soluzione ottima per il problema. Vogliamo caratterizzare i problemi di tipo (MSII) per cui un algoritmo greedy di questo tipo è esatto.

Un sistema di insiemi indipendenti (E, F) è detto *matroide* se tutti gli insiemi massimali hanno la stessa cardinalità; più precisamente, si richiede che

$$\forall E' \subseteq E,\ \text{se } I \text{ e } J \text{ sono massimali per } E',\ \text{allora } |I| = |J|. \tag{5.4}$$

La proprietà (5.4) è *necessaria* affinchè l'algoritmo greedy possa risolvere (MSII) per *qualsiasi* scelta dei costi c_i. Infatti, supponiamo che la proprietà non valga, ossia siano I e J due insiemi massimali rispetto ad un certo $V \subseteq E$ tali che $|I| < |J| \leq n$. Poniamo allora $c_e = 1 + \epsilon$ per $e \in I$, $c_e = 1$ per $e \in J \setminus I$, e $c_e = 0$ per tutti gli altri elementi di E. L'algoritmo greedy pone in S inizialmente tutti gli elementi di I, poi esamina e scarta tutti gli elementi di $J \setminus I$ ($V \supseteq I \cup J$ e I è massimale per V), infine eventualmente aggiunge ad S altri elementi di costo nullo, ottenendo una soluzione di costo $|I|(1 + \epsilon)$; la soluzione J di costo $|J| \geq |I| + 1$ è migliore di I se scegliamo $\epsilon < 1/|I|$. Si noti che dalla proprietà (5.4) segue che l'assunzione $c_e \geq 0$ può essere fatta senza perdita di generalità: come abbiamo visto per il caso di (MST), dato che ogni soluzione ammissibile del problema ha la stessa

cardinalità è possibile sommare al costo di ogni elemento un'opportuna costante C lasciando invariato l'insieme delle soluzioni ottime.
Dato un sistema di insiemi indipendenti (E, F), il *rango* di un qualsiasi insieme $E' \subseteq E$ è

$$rango(E') = \max \{ \, |S| \, : \, S \text{ massimale per } E' \, \} \ ;$$

se (E, F) è un matroide, allora la funzione *rango* può essere calcolata facilmente.

Esercizio 5.15 *Si dimostri che l'algoritmo greedy con costi $c_e = 1$ per $e \in E'$ e $c_e = 0$ altrimenti determina una soluzione S tale che $c(S) = rango(E')$ (suggerimento: se al termine dell'algoritmo $S \cup E'$ non fosse massimale per E', allora dovrebbe esistere un $S' \subseteq E'$ con $|S'| > |S|$; si consideri cosa accade al momento in cui l'algoritmo greedy esamina l'ultimo elemento di $S \cup S'$ e si usi (5.4) per ottenere una contradizione).*

Quindi, l'algoritmo greedy risolve all'ottimo almeno alcuni problemi di tipo (MSII). Vogliamo ora mostrare che l'algoritmo greedy risolve all'ottimo tutti i problemi di tipo (MSII) per qualsiasi scelta dei costi; un modo interessante per farlo è quello di considerare la seguente formulazione *PLI* di (MSII)

$$\text{(MSII-PLI)} \qquad \begin{array}{lll} \max & \sum_{e \in E} c_e x_e & \\ & \sum_{e \in S} x_e \leq rango(S) & \emptyset \subset S \subseteq E \\ & x_e \in \mathbb{N} & e \in E \end{array}$$

Esercizio 5.16 *Si verifichi che tutte le soluzioni ammissibili di (MSII-PLI) hanno $x_e \in \{0, 1\}$ per ogni $e \in E$ e che sono tutti e soli i vettori di incidenza degli elementi di F, ossia vettori nella forma*

$$x_e = \begin{cases} 1 & \text{se } e \in S \\ 0 & \text{altrimenti} \end{cases}$$

per un qualche $S \in F$ (suggerimento: per $S \in F$ si ha $rango(S) = |S|$ mentre per $S \notin F$ si ha $rango(S) < |S|$).

Come abbiamo visto nei paragrafi 5.1.2.1 e 5.1.2.4, consideriamo il rilassamento continuo di (MSII-PLI), (MSII-PLI), ottenuto sostituendo il vincolo $x_e \in \mathbb{N}$ con $x_e \geq 0$, ed il suo duale

$$\text{(DMSII)} \qquad \begin{array}{lll} \min & \sum_{S \subseteq E} rango(S) y_S & \\ & \sum_{S \, : \, e \in S} y_S \geq c_e & e \in E \\ & y_S \geq 0 & \emptyset \subset S \subseteq E \end{array} \ .$$

Possiamo mostrare che l'algoritmo greedy costrusce una soluzione primale ammissibile *intera* per (MSII-PLI) ed una soluzione duale ammissibile per (DMSII) che rispettano le condizioni degli scarti complementari

$$x_e (\sum_{S \, : \, e \in S} y_S - c_e) = 0 \qquad e \in E \ , \tag{5.5}$$

$$y_S (rango(S) - \sum_{e \in S} x_e) = 0 \qquad \emptyset \subset S \subseteq E \ . \tag{5.6}$$

Per semplificare la notazione, supponiamo che sia $E = \{1, 2, \ldots, n\}$ e che gli oggetti siano ordinati per costo non crescente, ossia $c_1 \geq c_2 \geq \ldots \geq c_n \geq 0$; introduciamo inoltre gli insiemi $S(1) = \{1\}$, $S(2) = \{1, 2\}$, $\ldots$, $S(e) = \{1, 2, \ldots, e\}$. Possiamo allora riscrivere l'algoritmo greedy sotto forma di un algoritmo primale-duale per la coppia (MSII-PLI), (DMSII):

```
Procedure Greedy-PD( E , F , S , y ) {
  y_{1} = c_1; S = S(1) = {1};                    /* x_1 = 1 */
  for( e = 2, ..., n ) do {
    y_{S(e)} = c_e; y_{S(e-1)} = y_{S(e-1)} - c_e;
    if( S ∪ {e} ∈ F ) then S = S ∪ { e };         /* x_e = 1 */
                    /* else                          x_e = 0 */
    }
  }
```

Procedura 5.3: Algoritmo *Greedy-PD*

È facile verificare che, data l'assunzione sulla procedura *Best* e l'ordinamento di E, la procedura *Greedy-PD* produce la stessa soluzione S della procedura *Greedy*. Inoltre, alla generica iterazione e la soluzione primale x (implicitamente) calcolata è ammissibile e rispetta le condizioni (5.5) e (5.6) con la soluzione duale y (prendendo ovviamente $y_S = 0$ per tutti gli S ai quali non è esplicitamente dato un valore diverso), in quanto si ha che:

- $S \subseteq S_e$;
- siccome (E, F) è un matroide, $rango(S) = rango(S(e))$;
- $\sum_{S\,:\,h\in S} y_S = c_h$ per $h = 1, \ldots, e$.

All'iterazione e viene soddisfatto il vincolo duale relativo all'elemento e, senza violare nessuno dei vincoli relativi agli elementi $h < e$; di conseguenza, a terminazione la soluzione duale y costruita dall'algoritmo è ammissibile, e quindi dimostra che la soluzione primale S ottenuta è ottima per (MSII-PLI).

Esempio 5.7: Algoritmo *Greedy-PD*
Esemplifichiamo i concetti della dimostrazione applicando l'algoritmo *Greedy-PD* al semplice problema di (MST) su un grafo completo con 3 nodi in cui $c_{12} = 6$, $c_{13} = 3$ e $c_{23} = 2$. In questo caso l'insieme E è l'insieme dei lati del grafo; per semplificare la notazione chiameremo $a = \{1, 2\}$, $b = \{1, 3\}$ e $c = \{2, 3\}$, per cui $E = \{a, b, c\}$. I problemi (MSII) e (DMSII) in questo caso sono

$$\begin{array}{lrcl}
\max & 6x_a + 3x_b + 2x_c & & \\
& x_a + x_b + x_c & \leq & 2 \\
& x_a + x_b & \leq & 2 \\
& x_a + x_c & \leq & 2 \\
& x_b + x_c & \leq & 2 \\
& x_a & \leq & 1 \\
& x_b & \leq & 1 \\
& x_c & \leq & 1 \\
& x_a\ ,\ x_b\ ,\ x_c & \geq & 0
\end{array}
\qquad
\begin{array}{lrcl}
\min & 2y_{abc} + 2y_{ab} + 2y_{ac} + 2y_{bc} + y_a + y_b + y_c & & \\
& y_{abc} + y_{ab} + y_{ac} + y_a & \geq & 6 \\
& y_{abc} + y_{ab} + y_{bc} + y_b & \geq & 3 \\
& y_{abc} + y_{ac} + y_{bc} + y_c & \geq & 2 \\
& y_{abc}\ ,\ y_{ab}\ ,\ y_{ac}\ ,\ y_{bc}\ ,\ y_a\ ,\ y_b\ ,\ y_c & \geq & 0
\end{array}$$

All'inizializzazione si pone $x_a = 1$ e $y_{\{a\}} = 6$. Alla prima iterazione si pone $y_{\{a,b\}} = 3$ e $y_{\{a\}} = 6 - 3 = 3$; siccome b non forma cicli, si pone $x_b = 1$. Alla terza iterazione si pone $y_{\{a,b,c\}} = 2$ e $y_{\{a,b\}} = 3 - 2 = 1$; siccome c forma un ciclo, si pone $x_c = 0$. È immediato verificare che y è duale ammissibile: tutti i vincoli del duale sono soddisfatti come uguaglianza. È anche facile verificare che le condizioni degli scarti complementari sono rispettate, ma più semplicemente si può notare che il costo della soluzione primale è $6 + 3 = 9$ ed il costo della soluzione duale è $2 \cdot 2 + 2 \cdot 1 + 1 \cdot 3 = 9$.

Un'interessante corollario dell'analisi appena svolta è il seguente:

Corollario 5.1 *Se (E, F) è un matroide allora (MSII) gode della proprietà di integralità.*

Quindi, i problemi di ottimizzazione su matroidi godono di una proprietà analoga a quella dei problemi di flusso (si veda il Teorema 3.11): esiste una formulazione di *PLI* "naturale" per il problema il cui rilassamento continuo ha sempre soluzioni intere, ossia tutti i vertici del poliedro sono interi. A differenza dei problemi di flusso, la formulazione è di dimensione esponenziale, ma, come abbiamo visto nel paragrafo 4.2.3, ammette un separatore polinomiale.
I matroidi sono dunque strutture combinatorie i cui corrispondenti problemi di ottimizzazione sono risolti esattamente dall'algoritmo greedy. Esempi di matroidi sono:

- E è l'insieme degli archi di un grafo non orientato ed F è la famiglia dei sottoinsiemi di archi che non inducono cicli: questo tipo viene detto un *matroide grafico*;
- $E = \{ A_1 , A_2 , \ldots , A_n \}$ con $A_i \in \mathbb{R}^m$, e F è la famiglia di tutti gli insiemi di vettori di E linearmente indipendenti: questo viene detto un *matroide matrico*;
- E è un insieme, $P = \{ E_1 , E_2 , \ldots , E_m \}$ è una sua partizione ed $F = \{ S \subseteq E : |I \cap E_j| \leq 1 \quad j = 1, \ldots, m \}$; questo viene detto un *matroide di partizione*.

Naturalmente molte strutture combinatorie *non* sono matroidi: si consideri ad esempio il caso in cui E è l'insieme degli archi di un grafo bipartito non orientato $G = (O \cup D, E)$ ed F è la famiglia dei suoi accoppiamenti (si veda il paragrafo 3.5). È facile verificare che questo non è un matroide; infatti, non tutti gli insiemi indipendenti massimali hanno la stessa cardinalità. Del resto, abbiamo visto che per risolvere i problemi di accoppiamento sono necessari algoritmi "più complessi" del semplice greedy. È interessante notare però che i nodi appartenenti a O ed a D inducono su E due diverse partizioni

$$P' = \{\ S(i)\ ,\ i \in O\ \} \qquad P'' = \{\ S(i)\ ,\ i \in D\ \}$$

corrispondenti alle stelle dei due insiemi di nodi. A partire da tali partizioni possiamo definire due matroidi di partizione (E, F') ed (E, F''): chiaramente $F = F' \cap F''$, ossia il sistema di insiemi indipendenti (E, F) che rappresenta gli accoppiamenti è l'intersezione dei due matroidi (E, F') e (E, F''), da cui si deduce che, in generale, l'intersezione di due matroidi *non* è un matroide. Per quanto gli algoritmi greedy non siano esatti per i problemi di accoppiamento, esistono algoritmi polinomiali per risolverli; in effetti si può dimostrare che qualsiasi problema di ottimizzazione che può essere espresso come l'*intersezione* di *due* matroidi ammette algoritmi polinomiali che ricalcano il funzionamento degli algoritmi per i problemi di accoppiamento visti al paragrafo 3.5.
Abbiamo anche già osservato al paragrafo 1.2.4.1 che il (TSP) può essere visto come un problema di ottimizzazione sull'intersezione di un problema di accoppiamento e di uno di albero minimo; in altri termini, (TSP) è problema di ottimizzazione sull'intersezione di *tre* matroidi. Ciò dimostra che i problema di ottimizzazione sull'intersezione di tre matroidi sono invece, in generale, $\mathcal{NP}$-ardui.

5.2 Algoritmi di ricerca locale

Gli algoritmi di ricerca locale sono basati su un'idea estremamente semplice ed intuitiva: data una soluzione ammissibile, si esaminano le soluzioni ad essa "vicine" in cerca di una soluzione "migliore" (tipicamente, con miglior valore della funzione obiettivo); se una tale soluzione viene trovata essa diventa la "soluzione corrente" ed il procedimento viene iterato, altrimenti—ossia quando nessuna delle soluzioni "vicine" è migliore di quella corrente—l'algoritmo termina avendo determinato un *ottimo locale* per il problema (si veda la figura 4.2). Elemento caratterizzante di un algoritmo di questo tipo è la definizione di "vicinanza" tra le soluzioni. In generale, se F è l'insieme ammissibile del problema in esame, possiamo definire una *funzione intorno* $I : F \to 2^F$: l'insieme $I(x)$ è detto *intorno* di x, e contiene le soluzioni considerate "vicine" ad x. Opportunamente definita la funzione intorno, un algoritmo di ricerca locale può essere schematizzato come segue:

```
procedure Ricerca_Locale ( F , c , x )  {
  x = Ammissibile(F);
  while( σ(x) ≠ x ) do
    x = σ(x)
  }
```

Procedura 5.4: Algoritmo di Ricerca Locale

L'algoritmo necessita di una soluzione ammissibile x^0 da cui partire: una tale soluzione può essere costruita ad esempio usando un algoritmo greedy. L'algoritmo genera una sequenza di soluzioni ammissibili $\{\ x^0\ ,\ x^1\ ,\ \ldots\ ,\ x^k\ ,\ \ldots\ \}$ tale che $x^{i+1} = \sigma(x^i)$; la tipica implementazione di σ, basata sulla funzione intorno I, è

$$\sigma(x) = \text{argmin}\{\ c(y)\ :\ y \in I(x)\ \}\ . \tag{5.7}$$

In questo modo, ad ogni passo dell'algoritmo viene risolto un problema di ottimizzazione ristretto all'intorno considerato. Più in generale, si può definire σ in modo tale 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; in questo modo, ad ogni passo dell'algoritmo viene risolto un problema di decisione. Si ha quindi $x = \sigma(x)$ quando nell'intorno di x non esiste nessuna soluzione migliore di x. In generale, la soluzione determinata dall'algoritmo di ricerca locale non è ottima per il problema, ma

solamente un *ottimo locale* relativamente alla funzione intorno scelta; I è detta una funzione intorno *esatta* per un problema P se l'algoritmo di ricerca locale basato sulla corrispondente trasformazione σ è in grado di fornire la soluzione ottima per ogni istanza di P e comunque scelto il punto di partenza x_0. Questo modo di operare è estremamente generale, ed infatti moltissimi algoritmi per problemi di ottimizzazione—tra cui quasi tutti quelli che abbiamo visto nei capitoli precedenti—possono essere classificati come algoritmi di ricerca locale.

Esercizio 5.17 *Per tutti gli algoritmi visti ai capitoli precedenti si discuta se essi possono o no essere considerati algoritmi di ricerca locale, fornendo la definizione delle relative funzioni intorno.*

La definizione della funzione intorno è quindi la parte fondamentale della definizione di un algoritmo di ricerca locale. Usualmente, si richiede che I possieda le due seguenti proprietà:

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

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

La proprietà 1) richiede che ogni soluzione appartenga all'intorno di se stessa, mentre la proprietà 2) richiede che sia teoricamente possibile per l'algoritmo di ricerca locale raggiungere in un numero finito di passi qualsiasi soluzione $y \in F$ (ad esempio quella ottima) a partire da qualsiasi altra soluzione $x \in F$ (ad esempio quella iniziale x^0). Si noti che la proprietà 2) non garantisce affatto che un algoritmo di ricerca locale, partendo da x, potrebbe effettivamente arrivare a y: per questo sarebbe necessario anche che $c(z_i) < c(z_{i-1})$ per $i = 1, 2, \ldots, p$, il che in generale non è vero. Inoltre, in pratica si utilizzano anche funzioni intorno che non soddisfano questa proprietà.

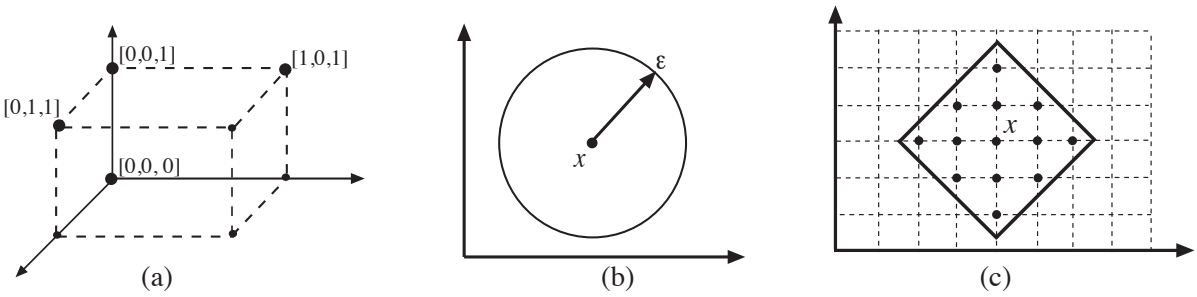

Figura 5.7: Alcuni esempi di funzioni intorno

Alcuni esempi di funzioni intorno sono mostrati in figura 5.7, in particolare

- figura 5.7(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 5.7(b): $F \subseteq \mathbb{R}^n$, $I_\varepsilon(x) = \{\ y \in F\ :\ \| x - y \| \leq \varepsilon\ \}$ (intorno Euclideo);
- figura 5.7(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 ("mosse") che trasformano una soluzione ammissibile del problema in un'altra soluzione ammissibile.

5.2.1 Esempi di algoritmi di ricerca locale

Discutiamo adesso funzioni intorno per alcuni problemi di *OC*, in modo da fornire una panoramica di alcune delle principali metodologie utilizzate per costruire approcci di ricerca locale.

5.2.1.1 Il problema dello zaino

Si consideri il problema dello zaino (KP) definito al paragrafo 1.2.2.1. Un primo esempio di funzione intorno per questo problema potrebbe essere identificato dalle seguenti "mosse di inserzione e cancellazione": data una soluzione ammissibile dello zaino, si costruisce una diversa soluzione ammissibile inserendo nello zaino uno degli oggetti attualmente non selezionati – e per il quale esiste ancora sufficiente capacità residua – oppure togliendo dallo zaino uno degli oggetti attualmente selezionati (il che non può che determinare una nuova soluzione ammissibile). Data una soluzione ammissibile x per il problema dello zaino, che possiamo considerare un vettore di n variabili binarie x_i, $i = 1, \ldots, n$, questa funzione intorno, che chiameremo I_1, associa ad x tutte le soluzioni ammissibili che possono essere ottenute trasformando un singolo x_i da 0 a 1 o viceversa; si noti che questa è esattamente la funzione intorno rappresentata in figura 5.7(a). Tali soluzioni sono quindi al più n. È quindi facile verificare che la funzione σ data da (5.7) può essere calcolata in tempo lineare $O(n)$ nel numero di oggetti dello zaino.

Esercizio 5.18 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ in $O(n)$ per la funzione intorno I_1.*

È però facile vedere che questo intorno non permette sicuramente di migliorare le soluzioni ottenute dall'algoritmo *greedy* descritto nel paragrafo 5.1.1.1. Infatti, al termine di quell'algoritmo lo zaino non ha capacità residua sufficiente per nessuno degli oggetti non selezionati: se l'avesse allora l'avrebbe avuta anche al momento in cui l'oggetto è stato esaminato (la capacità residua è non crescente nel corso dell'algoritmo), e quindi l'oggetto sarebbe stato inserito. Inoltre, per l'ipotesi che i costi siano non negativi togliere un oggetto dallo zaino non può migliorare (aumentare) il valore della fuunzione obiettivo. In altri termini, l'algoritmo *greedy* produce un *ottimo locale* rispetto all'intorno I_1. Se si volesse utilizzare un algoritmo di ricerca locale per tentare di migliorare la soluzione ottenuta dall'algoritmo *greedy* occorrerebbe quindi sviluppare un intorno diverso. Ad esempio, si potrebbe utilizzare la funzione intorno I_2 che, oltre alle mosse di inserzione e cancellazione, utilizza anche "mosse di scambio" tra un oggetto selezionato ed uno non selezionato. In altri termini, la funzione associa ad x tutte le soluzioni (ammissibili) x' che differiscono da x in *al più* due posizioni; se x' differisce da x in i e j, con $i \neq j$, deve essere $x_i = 0$ e $x_j = 1$, e quindi $x'_i = 1$ e $x'_j = 0$. È facile verificare che la funzione σ data da (5.7) per I_2 può essere calcolata in tempo $O(n^2)$, in quanto il numero di soluzioni $x' \in I(x)$ è certamente minore o uguale al numero di coppie (i, j) con i e j in $1, \ldots, n$, e la verifica dell'ammissibilità di una mossa può essere fatta in $O(1)$.

Esercizio 5.19 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ in $O(n^2)$ per la funzione intorno I_2.*

Le soluzioni prodotte dall'algoritmo *greedy* possono *non* essere ottimi locali rispetto a questo nuovo intorno.

Esempio 5.8: Mosse di scambio per il problema dello zaino
Si consideri la seguente istanza del problema dello zaino:

$$\begin{array}{lrcrcrcrcrcl} \max & 2x_1 & + & 8x_2 & + & 5x_3 & + & 6x_4 & + & x_5 & & \\ & 3x_1 & + & 2x_2 & + & 4x_3 & + & 6x_4 & + & 3x_5 & \leq & 8 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & \in & \{0,1\} \end{array}$$

L'ordine CUD è: $2, 3, 4, 1, 5$. Pertanto, l'euristica Greedy CUD determina la soluzione $x = [\,0\,,\,1\,,\,1\,,\,0\,,\,0\,]$ di valore $cx = 13$. Ma scambiando l'oggetto 3, nello zaino, con l'oggetto 4, fuori dallo zaino, si ottiene la soluzione $x' = [\,0\,,\,1\,,\,0\,,\,1\,,\,0\,]$, pure ammissibile, di valore $cx' = 14$. Pertanto, x non è un ottimo locale rispetto all'intorno I_2.

Esercizio 5.20 *Per ciascuno degli altri due ordinamenti (costi non crescenti e pesi non decrescenti), si costruisca (se esiste) un'istanza del problema dello zaino per cui la soluzione prodotta dall'algoritmo greedy con l'ordinamento dato non sia un ottimo locale rispetto all'intorno I_2.*

Analizzando ulteriormente le mosse si possono poi scoprire proprietà che risultano utili nell'implementazione dell'algoritmo. Ad esempio, si consideri il caso di due oggetti i e j che hanno lo stesso costo; se hanno anche lo stesso peso allora qualsiasi scambio è inutile. Se invece si ha $c_i = c_j$ e $a_i < a_j$, allora è facile vedere che se esiste una soluzione ottima che contiene j allora esiste anche una soluzione ottima che contiene i al posto di j (la si ottiene semplicemente scambiando i con j). Si può quindi operare in modo tale che in ciascuna soluzione generata siano presenti, tra tutti gli oggetti con un dato costo, solo quelli di peso minore, e quindi in modo tale che non vengano mai scambiati oggetti con lo stesso costo, evitando di valutare mosse "inutili".

La funzione intorno I_2 *domina* la funzione intorno I_1, ossia $I_2(x) \supseteq I_1(x)$ per ogni x. In questo caso si può affermare che I_2 è "migliore" di I_1 nel senso che tutti gli ottimi locali di I_2 sono anche ottimi locali di I_1, ma il viceversa può non essere vero. Infatti, x è un ottimo locale per I_2 se $c(x') \leq c(x)\ \forall x' \in I_2(x)$, e questo sicuramente implica che $c(x') \leq c(x)\ \forall x' \in I_1(x)$. Ciò non significa che un algoritmo di ricerca locale che usa I_1 determinerà necessariamente una soluzione peggiore di quella determinata da un algoritmo di ricerca locale che usa I_2, perchè le "traiettorie" seguite dai due algoritmi nello spazio delle soluzioni ammissibili saranno diverse. È però vero che, avendo I_1 "più" minimi locali, un algoritmo di ricerca locale che usa I_1 può arrestarsi una volta giunto ad una certa soluzione dalla quale un algoritmo di ricerca locale che usa I_2 proseguirebbe invece la ricerca, come l'esempio precedente mostra.
È interessante notare come ciascuna mossa di scambio sulla coppia (i, j) possa essere considerata la *concatenazione* di due mosse, rispettivamente una di cancellazione su j ed una di inserzione su i. Il motivo per cui l'intorno basato sullo scambio è "più potente" è che il risultato della concatenazione delle due mosse viene "visto" immediatamente. Infatti, nessuna mossa di cancellazione presa a sè potrebbe mai essere accettata da un algoritmo di ricerca locale, in quanto comporta (se i costi sono tutti positivi) un peggioramento della funzione obiettivo. Cancellare un oggetto può però permettere di inserirne uno di costo maggiore, ottenendo un miglioramento complessivo del valore della funzione obiettivo; quindi la concatenazione delle due mosse è conveniente, anche se la prima mossa presa singolarmente non lo è. In generale, intorni basati su "mosse complesse", ottenute concatenando un certo numero di "mosse semplici", permettono di "vedere le conseguenze" delle singole mosse semplici e quindi di effettuare mosse che non sarebbero accettabili da intorni basati direttamente sulle "mosse semplici". Per contro, spesso, come nell'esempio precedente, valutare la funzione σ per intorni basati su "mosse complesse" è più costoso. Inoltre, si noti che questo comportamento è dovuto al fatto che, nell'algoritmo di ricerca locale, si insiste sull'ottenere un miglioramento ad ogni passo. L'osservazione precedente suggerisce due strategie, in qualche modo alternative, per migliorare la qualità delle soluzioni determinate da un algoritmo di ricerca locale:

- aumentare la "complessità" delle mosse, ossia la "dimensione" dell'intorno, per permettere di "vedere" una frazione maggiore dello spazio delle soluzioni ad ogni passo,
- permettere peggioramenti della funzione obiettivo purchè ci sia un "miglioramento a lungo termine".

Nei prossimi paragrafi discuteremo entrambe queste strategie, non prima però di aver introdotto altri esempi di intorno.

5.2.1.2 Ordinamento di lavori su macchine con minimizzazione del tempo di completamento

Si consideri il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS) definito al paragrafo 1.2.9.1. È facile definire per questo problema una funzione intorno basata su "mosse" analoghe a quelle viste per il problema dello zaino:

- spostamento: selezionare un lavoro i attualmente assegnato ad una certa macchina h ed assegnarlo ad una macchina $k \neq h$;

- scambio: selezionare un lavoro i attualmente assegnato ad una certa macchina h ed un lavoro j attualmente assegnato ad una certa macchina $k \neq h$, assegnare i alla macchina k ed assegnare j alla macchina h.

Come nel caso del problema dello zaino, una mossa di scambio può essere vista come la concatenazione di due mosse di spostamento, e l'intorno che usa entrambe i tipi di mosse domina l'intorno che usa solamente quelle di spostamento. Si consideri ad esempio il caso di un problema con due macchine e cinque lavori di lunghezza 5, 4, 4, 3 e 2, e la soluzione (determinata da LPT) $\{5, 3, 2\}$, $\{4, 4\}$ con makespan 10. Qualsiasi mossa di spostamento peggiora il makespan della soluzione e quindi non viene accettata; per contro, scambiare il lavoro di lunghezza 5 con uno di quelli di lunghezza 4 produce una soluzione migliore (in particolare ottima).

Esercizio 5.21 *Si determini la complessità di valutare la funzione σ data da (5.7) per l'intorno proposto; si indichi poi come cambia il risultato qualora si effettuino solamente mosse di spostamento.*

Questo esempio mostra come per costruiree funzioni intorno per un certo problema di *OC* si possa trarre ispirazione da funzioni intorno già viste per altri problemi di *OC*. Ogni problema ha però le sue caratteristiche, che devono essere prese in debita considerazione. Il caso di (MMMS) e quello di (KP) sono diversi in molti aspetti: ad esempio, mentre in (KP) le mosse devono tenere in conto del vincolo di capacità dello zaino, in (MMMS) non ci sono di fatto vincoli, e quindi tutte le mosse producono soluzioni ammissibili. Inoltre, in (KP) tutte le mosse producono una variazione della funzione obiettivo (a meno che non siano scambiati due oggetti dello stesso costo, il che come abbiamo visto può essere evitato), mentre in (MMMS) tutte le mosse che non coinvolgono almeno una delle macchine che determinano il makespan della soluzione non possono migliorare il valore della funzione obiettivo. In effetti, è facile vedere che sono ottimi locali per l'algoritmo di ricerca locale basato sull'intorno così definito tutte le soluzioni in cui ci siano almeno due macchine che determinano il makespan: nessuna mossa di scambio può ridurre il tempo di completamento di due macchine (tipicamente ridurrà il tempo di completamento di una delle macchine coinvolte ed aumenterà quello dell'altra), e quindi migliorare il makespan della soluzione. Per questo è conveniente, nell'algoritmo di ricerca locale, accettare di spostarsi anche su soluzioni con lo stesso makespan di quella corrente, perché siano migliori per qualche altra caratteristica. Ad esempio, è possibile accettare mosse che diminuiscano il *numero* di macchine che determinano il makespan; ciò corrisponde a minimizzare una funzione obiettivo del tipo $v + \varepsilon ns$, dove v è il valore del makespan, ns è il numero di macchine che hanno tempo di completamento esattamente pari a v e ε è un valore opportunamente piccolo ($\varepsilon < 1/m$). Questa funzione obiettivo discrimina tra soluzioni con lo stesso makespan, e quindi le soluzioni in cui ci siano almeno due macchine che determinano il makespan non sono più (necessariamente) ottimi locali per l'intorno.

Esercizio 5.22 *In generale, non è necessariamente vero che funzioni intorno per due problemi di* OC *"simili" possano essere simili; si proponga una funzione intorno per il problema dell'ordinamento di lavori su macchine con minimizzazione del numero delle macchine.*

5.2.1.3 Il problema del commesso viaggiatore

Si consideri il problema del commesso viaggiatore (TSP) definito al paragrafo 1.2.2.3. In questo caso, i costituenti elementari di una soluzione sono lati, e quindi si può pensare, in analogia con gli esempi precedenti, ad operazioni di tipo "scambio" che coinvolgano gli archi del ciclo. In questo caso non è chiaramente possibile operare su un solo lato del grafo, in quanto cancellando un qualsiasi lato dal ciclo si ottiene un cammino (Hamiltoniano), e l'unico modo possibile

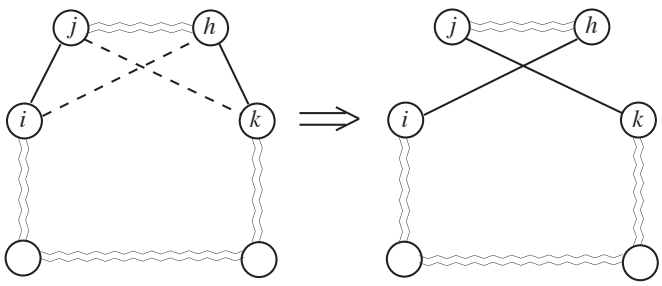

Figura 5.8: Un "2-scambio" per il (TSP)

per trasformare il cammino in un ciclo Hamiltoniano è quello di aggiungere nuovamente il lato appena cancellato. È quindi necessario operare su almeno due lati contemporaneamente: ad esempio, data una soluzione ammissibile x, l'intorno $I(x)$ basato sui "2-scambi" contiene tutti i cicli Hamiltoniani che si possono ottenere da x selezionando due lati $\{i, j\}$ ed $\{h, k\}$ non consecutivi del ciclo e sostituendoli con gli archi $\{i, h\}$ e $\{j, k\}$, se esistono. Questa operazione è esemplificata in figura 5.8.

Esempio 5.9: Ricerca locale per il (TSP)
Si consideri l'istanza in figura 5.9(a) ed il suo ciclo Hamiltoniano $C = \{1, 2, 3, 4, 5\}$ (lati in neretto), di costo $c = 14$. Per generare l'intorno basato sul 2-scambio di C è sufficiente enumerare tutte le coppie di lati $\{i, j\}$ ed $\{h, k\}$ non consecutive su C; in questo caso si ottengono i tre cicli Hamiltoniani mostrati in figura 5.9(b), (c) e (d), ciascuno col corrispondente costo e con indicati i lati di C sostituiti. Quindi, C non è un ottimo locale rispetto all'intorno basato sui 2-scambi (si noti che, per definizione, C appartiene all'intorno): il ciclo $C' = \{1, 2, 3, 5, 4\}$ ha costo minore.

Il costo di valutare la funzione σ corrispondente a questo intorno è $O(n^2)$, poichè tante sono le coppie di lati non consecutivi del ciclo, e la valutazione del costo del ciclo ottenuto da un 2-scambio può essere fatta in $O(1)$ (conoscendo il costo del ciclo originario). Gli algoritmi di ricerca locale per il TSP attualmente ritenuti piè efficienti fanno uso di operazioni di 2-scambio o simili.

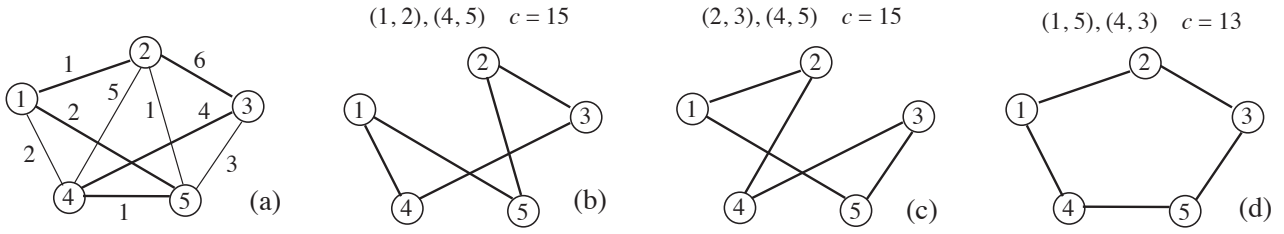

Figura 5.9: Ricerca locale basata sui "2-scambi" per il (TSP)

5.2.1.4 Il problema del Constrained MST

Si consideri il problema del Constrained MST definito nell'Esempio 4.1. In questo caso le soluzioni ammissibili del problema sono alberi, e si può pensare ad operazioni analoghe a quelle viste in precedenza che coinvolgano lati dell'albero. È però necessario porre attenzione al vincolo sul massimo peso dei sottoalberi della radice. Dato che ogni soluzione ammissibile x del problema è un albero di copertura radicato del grafo, possiamo rappresentarla mediante un vettore $p[\cdot]$ di predecessori. Indicheremo inoltre con $T(i)$ il sottoalbero di radice i della soluzione (se i è una foglia, $T(i)$ contiene il solo nodo i) e con $Q(i) = \sum_{h \in T(i)} q_h$ il suo peso: siccome la soluzione è ammissibile si avrà sicuramente $Q(i) \leq Q$ per ogni $i \neq r$, dove r è la radice dell'albero,
Un primo esempio di mossa per il (CMST) è la cosiddetta "Cut & Paste", che consiste nel selezionare un nodo $i \neq r$ ed un nodo $j \notin T(i)$ tale che $j \neq p[i]$ e porre $p[i] = j$. Ciò corrisponde ad eliminare dall'albero il lato $\{p[i], i\}$ e sostituirlo con il lato $\{j, i\}$, ossia a "potare" il sottoalbero $T(i)$ dalla sua posizione ed "innestarlo sotto il nodo j", come mostrato in figura 5.10(a). Naturalmente, la mossa può essere compiuta solamente se $Q(i) + Q(h) \leq Q$, dove h è il figlio della radice tale che $j \in T(h)$.
È possibile implementare il calcolo della funzione σ in modo tale che il controllo di tutte le possibili mosse abbia costo $O(n)$; è necessario mantenere per ogni nodo i il valore di $Q(i)$ ed esaminare i nodi $j \neq i$ in ordine opportuno (ad esempio visitando l'albero a partire dalla radice) per disporre in $O(1)$ di $Q(h)$ e dell'informazione che j non appartiene a $T(i)$. Poichè la variazione del valore della funzione obiettivo corrispondente ad una mossa di questo tipo è $c_{ji} - c_{p[i]i}$, e quindi può essere calcolato in $O(1)$, il costo di valutare la funzione σ data da (5.7) per l'intorno che usa mosse di "Cut & Paste" è $O(n^2)$.

Esercizio 5.23 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ data da (5.7) in $O(n^2)$ per la funzione intorno che usa mosse di "Cut & Paste".*

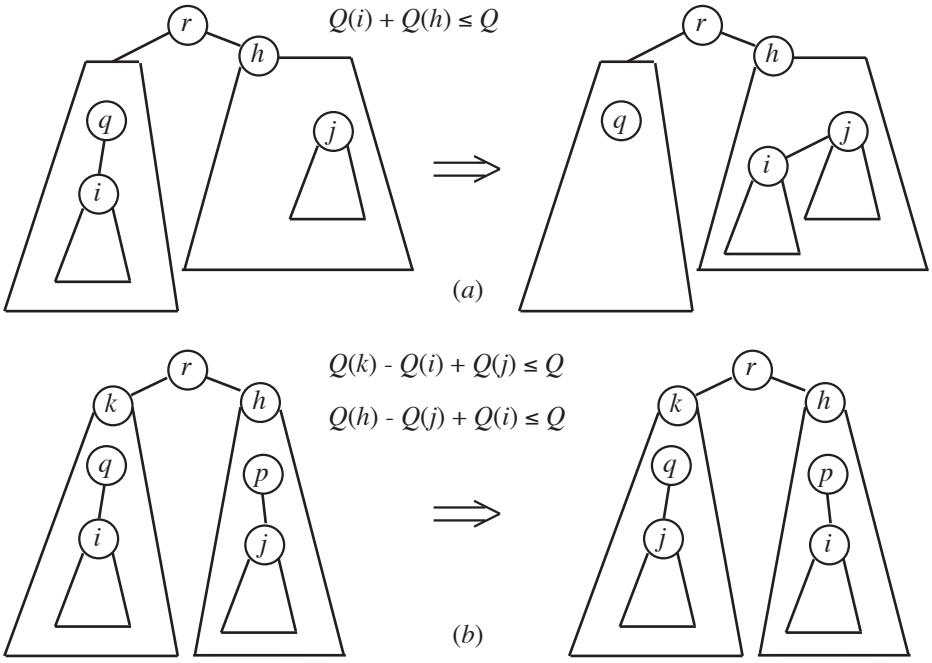

Figura 5.10: Un'operazione di "Cut & Paste" per il (CMST)

Analogamente ai casi visti in precedenza, si possono realizzare mosse "più complesse" combinando opportunamente mosse "semplici", in questo caso quelle di "Cut & Paste". Ad esempio, è possibile pensare a mosse di scambio in cui si selezionano due nodi i e j tali che $i \notin T(j)$ e $j \notin T(i)$ e si scambiano i predecessori di i e j, purché ovviamente ciò non violi il vincolo sul massimo peso dei sottoalberi della radice: l'operazione è esemplificata in figura 5.10(b). Siccome durante la mossa di scambio si tiene conto, nel valutare l'ammissibilità della soluzione ottenuta, del fatto che un insieme di nodi viene rimosso da ciascun sottoalbero della radice coinvolto mentre un altro sottoinsieme viene aggiunto, è chiaro che possono esistere ottimi locali per l'intorno basato sulle sole mosse di "Cut & Paste" che non sono ottimi locali per l'intorno che usa anche mosse di scambio, in quanto le due mosse di "Cut & Paste" corrispondenti ad una mossa di scambio potrebbero non essere eseguibili sequenzialmente per via del vincolo sul massimo peso dei sottoalberi della radice.

Esercizio 5.24 *Si discuta la complessità di calcolare la funzione σ data da (5.7) per la funzione intorno che usa sia mosse di "Cut & Paste" che mosse di scambio.*

È interessante notare che dopo ogni mossa di ricerca locale è possibile effettuare un'operazione di "ottimizzazione globale" sui sottoalberi coinvolti. Infatti, il problema (CMST) sarebbe risolubile efficientemente se si conoscessero i sottoinsiemi di nodi che formano ciascun sottoalbero della radice in una soluzione ottima: basterebbe applicare una procedura per l'(MST) a ciascun grafo parziale individuato da un sottoinsieme di nodi per determinare gli archi del sottoalbero ottimo. In generale, i sottoalberi ottenuti dopo una mossa di scambio possono non essere alberi di copertura di costo minimo per l'insieme di nodi che coprono, anche nel caso in cui lo fossero i sottoalberi di partenza, per cui applicando una procedura per determinare l'albero di copertura di costo minimo ai sottoalberi coinvolti dalla mossa si potrebbe ulteriormente migliorare la soluzione ottenuta. Questo è vero anche per il sottoalbero "destinazione" di una mossa di "Cut & Paste", ma chiaramente non per quello di "partenza".

Esercizio 5.25 *Si discuta come modificare le procedure note per l'(MST) in modo tale che risolvano in modo efficiente i problemi di (MST) generati da una mossa di "Cut & Paste" o di scambio.*

5.2.1.5 Dislocazione ottima di impianti

Si consideri il problema della dislocazione ottima di impianti definito al paragrafo 1.2.13.1: sono date n possibili località dove aprire impianti, ciascuno con un dato costo di installazione d_i ed un numero massimo u_i di clienti che può servire, ed m clienti, ciascuno dei quali può essere servito dall'impianto i al costo c_{ij}, $j = 1, \ldots, m$. Si vuole decidere in quali delle n località aprire gli impianti e, per ciascuno di essi, l'insieme dei clienti assegnati, in modo tale che ogni cliente sia assegnato ad uno ed un solo impianto e che il costo complessivo (di installazione e gestione) sia minimo. Il problema ha due tipi diversi di variabili binarie: le y_i, $i = 1, \ldots, n$, per rappresentare la scelta relativa agli impianti da aprire, e le x_{ij}, $i = 1, \ldots, n, j = 1, \ldots, m$, per assegnare i clienti agli impianti. Per analogia con gli esempi precedenti si possono definire mosse che riguardano poche variabili: ad esempio, sono mosse possibili l'apertura o chiusura di un singolo impianto (fissare ad 1 o a 0 una data variabile y_i), lo scambio tra un impianto chiuso ed uno aperto, l'assegnazione di un cliente ad un diverso impianto e così via. In questo caso è bene però tener presente che esiste una chiara "gerarchia" tra le variabili: le y rappresentano le decisioni "principali", mentre le x rappresentano decisioni "secondarie". Ciò deriva dal fatto che, una volta fissato il valore delle y, il valore ottimo delle x può essere facilmente ottenuto risolvendo un problema di flusso di costo minimo.

Esercizio 5.26 *Si descriva come istanza di un problema di (MCF) il problema di determinare, data una soluzione y, l'assegnamento ottimo dei clienti agli impianti aperti; in particolare, si discuta cosa accade nel caso in cui non siano aperti abbastanza impianti per servire tutti i clienti.*

Quindi, in linea di principio le mosse potrebbero essere effettuate solamente sulle variabiil y: una volta effettuata una mossa, applicando un algoritmo per (MCF) è possibile determinare il miglior valore possibile per le variabili x data la nuova scelta di y.
Un possibile svantaggio di operare in questo modo consiste nel dover risolvere un problema di (MCF) per valutare qualsiasi mossa: anche limitandosi a mosse semplici, quali l'apertura e chiusura di un singolo impianto, ciò richiede la soluzione di n problemi di (MCF) per valutare la funzione σ ad ogni passo dell'algoritmo di ricerca locale. Sono possibili diverse strategie per cercare di ridurre il costo computazionale del calcolo di σ. Ad esempio, si potrebbero specializzare gli algoritmi noti per (MCF) alla particolare forma delle istanze da risolvere in questo caso, analogamente a quanto fatto nel paragrafo 3.5 per i problemi di accoppiamento. Alternativamente, si potrebbero determinare inizialmente soluzioni approssimate del (MCF), ossia costruire (velocemente) un assegnamento non necessariamente ottimo dei clienti agli impianti aperti. Ad esempio, nel caso di chiusura di un impianto i si potrebbero distribuire gli utenti j precedentemente assegnati a quell'impianto con un semplice criterio greedy basato sui costi, esaminandoli in ordine arbitrario ed assegnando ciascuno all'impianto aperto h con costo c_{hj} minimo tra quelli che hanno ancora capacità residua; analogamente, nel caso di apertura di un impianto i si potrebbero assegnare ad i utenti j, attualmente assegnati ad un diverso impianto h, per cui risulti $c_{ij} < c_{hj}$, partendo da quelli in cui la $c_{hj} - c_{ij}$ è maggiore. Se la nuova soluzione (x, y) così ottenuta è migliore della soluzione corrente allora sicuramente lo sarà anche quella ottenuta risolvendo il (MCF); si può quindi selezionare la mossa da compiere sulla base di questa valutazione approssimata, risolvendo il (MCF) una sola volta per iterazione. Se invece così facendo non si determina nessuna souzione migliore di quella corrente si possono riesaminare le mosse risolvendo esattamente il (MCF), in quanto la soluzione approssimata può aver portato a non effettuare mosse che invece risultano convenienti. Infine, invece che valutazioni superiori del costo del(MCF) si potrebbero utilizzare valutazioni inferiori, sfruttando informazione duale analogamente a quanto mostrato nel paragrafo 2.3.3. È in generale difficile valutare a priori quale di queste strategie possa risultare più conveniente; sono necessari a tal proposito esperimenti su un adeguato insieme di istanze "campione" che confrontino la qualità delle soluzioni ottenute e lo sforzo computazionale richiesto dall'algoritmo di ricerca locale con le diverse varianti di funzione intorno.
I due esempi precedenti mostrano come lo studio della struttura dei problemi sia fondamentale per costruire funzioni intorno opportune. In particolare, in molti casi emergono "gerarchie" tra gruppi di variabili del problema, in quanto è possibile utilizzare algoritmi efficienti per determinare il valore ottimo di un gruppo di variabili una volta fissato il valore delle altre. Questo è un caso in cui si rivela

l'importanza della conoscenza degli algoritmi per i problemi "facili" per costruire algoritmi efficienti per problemi "difficili".

5.2.2 Intorni di grande dimensione

Il costo computazionale 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. È necessario quindi operare un attento bilanciamento tra la complessità della trasformazione σ e la qualità delle soluzioni determinate; come esempio estremo, la trasformazione σ data da (5.7) basata sull'intorno $I(x) = X$ fornisce sicuramente la soluzione ottima, ma richiede di risolvere all'ottimo il problema originario, e quindi è di fatto inutile. In molti casi risulta comunque conveniente utilizzare intorni "di grande dimensione", corrispondenti a "mosse complesse". Ciò è spesso legato alla possibilità di implementare la funzione σ in modo efficiente, ad esempio utilizzando algoritmi per problemi di ottimizzazione per calcolare (5.7) senza dover esaminare esplicitamente tutte le soluzioni nell'intorno $I(x)$. Discutiamo nel seguito alcuni esempi di intorni di questo tipo.

5.2.2.1 Il problema del commesso viaggiatore

Una famiglia di funzioni intorno di grande dimensione che generalizza quella discussa nel §5.2.1.3 è quella basata sui "k-scambi": in un k-scambio si selezionano e rimuovono k archi del ciclo e si costruiscono tutti i possibili cicli Hamiltoniani che è possibile ottenere combinando i sottocammini rimanenti. Non è difficile vedere che la complessità di calcolare σ quando I è definita mediante operazioni di "k-scambio" cresce grosso modo come n^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.
Non essendo noti modi per calcolare σ che non richiedano di esaminare sostanzialmente tutte le soluzioni nell'intorno, si possono utilizzare schemi di ricerca locale in cui la dimensione dell'intorno (il numero k di scambi) varia dinamicamente. Si può ad esempio utilizzare preferibilmente 2-scambi, passando ai 3-scambi solamente se il punto corrente si rivela essere un ottimo locale per l'intorno basato sui 2-scambi; analogamente, se il punto corrente si rivela essere un ottimo locale anche per l'intorno basato sui 3-scambi si può passare ai 4-scambi e così via, fino ad un qualche valore limite di k fissato a priori. In questo caso risulta spesso conveniente ritornare immediatamente ai 2-scambi non appena si sia effettuata una mossa di ricerca locale con un k-scambio per $k > 2$, in quanto il k-scambio potrebbe aver generato una soluzione che non è un ottimo locale per il 2-scambio. Inoltre, quando k è "grande" può essere conveniente evitare di calcolare esattamente (5.7), terminando la computazione non appena si determina una qualunque soluzione nell'intorno che migliori "abbastanza" il valore della funzione obiettivo.

5.2.2.2 Il problema del Constrained MST

Come abbiamo già rilevato, il problema del (CMST) consiste di fatto nel determinare la partizione ottima dell'insieme dei nodi in un numero opportuno (non noto a priori) di sottoinsiemi disgiunti, in ciascuno dei quali la somma dei pesi dei nodi non sia superiore alla soglia Q; nota la partizione ottima, il problema è facilmente risolubile. Quindi, il (CMST) è un esempio di problema in cui le soluzioni ammissibili sono identificabili con una partizione di un insieme base in un certo numero di sottoinsiemi disgiunti con opportune proprietà: altri problemi con questa struttura sono ad esempio i problemi di ordinamento di lavori su macchine (ogni macchina corrisponde ad un insieme).

Un modo abbastanza generale per costruire intorni di grande dimensione per problemi con questo tipo di struttura è quello dello "scambio ciclico". Per il caso del (CMST), ciò corrisponde a selezionare un insieme di nodi i_1, i_2, ..., i_k, appartenenti ciascuno a un diverso sottoalbero della radice, ed

effettuare una mossa di scambio simultanea che coinvolge tutti i nodi: porre $p[i_1] = p[i_2]$, $p[i_2] = p[i_3]$, ..., $p[i_{k-1}] = p[i_k]$ e $p[i_k] = p[i_1]$. Naturalmente, la mossa è possibile solo se la soluzione così ottenuta non viola il vincolo di capacità sui sottoalberi. È chiaro come queste mosse generalizzino quella di scambio vista al paragrafo 5.2.1.4, e quindi che permettano potenzialmente di non rimanere "intrappolati" in soluzioni che siano ottimi locali per queste ultime.
Implementare un'algoritmo di ricerca locale basato su mosse di scambio ciclico richiede però di determinare in modo efficiente, ad ogni iterazione, un insieme di nodi i_1, i_2, ..., i_k a cui corrisponda una mossa di scambio ciclico ammissibile e che migliori il valore della funzione obiettivo. Può risultare utile esprimere tale problema come un problema di decisione su grafi (su un grafo diverso da quello del problema originale). Sia data infatti una soluzione ammissibile di (CMST), e si consideri il grafo orientato che ha per nodi quelli del grafo originale, tranne la radice, e nel quale esiste l'arco (i, j), di costo $c_{p[j]i} - c_{p[i]i}$, se e solo se i e j appartengono a sottoalberi della radice diversi ed è possibile "innestare" i sotto il predecessore di j senza violare il vincolo di capacità, purchè nel contempo j sia "potato" (ed innestato sotto un diverso sottoalbero, non importa quale). È facile verificare che un ciclo orientato di costo negativo su questo grafo, in cui tutti i nodi appartengano a sottoalberi diversi, rappresenta una mossa di scambio ciclico che migliora il valore della funzione obiettivo, e, viceversa, qualsiasi mossa di scambio ciclico corrisponde ad un ciclo di questo tipo. Il problema di determinare una mossa di scambio ciclico può quindi essere quindi formulato nel seguente modo: dato un grafo orientato $G = (N, A)$ con costi associati agli archi, in cui l'insieme dei nodi N sia partizionato in un certo numero di sottoinsiemi disgiunti $N_1, \ldots, N_k$, determinare se esiste nel grafo un ciclo orientato di costo negativo che contenga al più un nodo per ciascuno dei sottoinsiemi N_i. Risolvere questo problema di decisione corrisponde a determinare se esiste oppure no una mossa di scambio ciclico che determina una soluzione migliore di quella corrente; la funzione σ data da (5.7) richiederebbe invece di determinare il ciclo di costo minimo tra tutti quelli aventi le caratteristiche desiderate. Come abbiamo visto nel paragrafo 3.2.5, il problema di determinare se esiste un ciclo orientato di costo negativo in un grafo ha complessità polinomiale; invece, determinare il ciclo orientato di costo minimo è $\mathcal{NP}$-arduo. Il problema in esame richiede però non di determinare un qualsiasi ciclo di costo negativo, ma un ciclo di costo negativo che contenga al più un nodo per ciascuno dei sottoinsiemi N_i; come spesso accade, pur essendo una variante apparentemente minore di un problema polinomiale, questo problema è $\mathcal{NP}$-completo. È però possibile costruire euristiche per questo problema che cerchino di determinare un ciclo di costo negativo con le caratteristiche richieste, pur non garantendo di determinarlo anche nel caso in cui esista; se il ciclo viene trovato è possibile effettuare la mossa di scambio ciclico corrispondente, altrimenti l'algoritmo di ricerca locale si ferma dichiarando—forse erroneamente—che la soluzione corrente è un ottimo locale per l'intorno basato sullo scambio ciclico. Utilizzando euristiche opportune per la determinazione del ciclo di costo negativo si possono ottenere algoritmi di ricerca locale in grado di determinare efficientemente, in pratica, soluzioni di buona qualità per il problema.

Esercizio 5.27 *Si discuta come modificare l'algoritmo basato su* SPT.L *per la determinazione dei cicli orientati di costo negativo in un grafo in modo da ottenere un algoritmo euristico per determinare mosse di scambio ciclico.*

Questo esempio mostra come tecniche di ottimizzazione, esatte o euristiche, possano essere utilizzate per implementare in modo efficiente la funzione σ anche per intorni di grandi dimensioni. Si noti come la conoscenza di un algoritmo esatto per risolvere un problema di decisione "facile", in questo caso il problema di determinare l'esistenza di un ciclo orientato di costo negativo in un grafo, suggerisca algoritmi euristici per un problema simile, che possono a loro volta essere utilizzati per implementare algoritmi euristici per un problema molto diverso. Questo mostra il livello di complessità e sofisticazione che è a volte necessario affrontare per realizzare algoritmi efficienti, esatti o euristici, per problemi di *OC*, e quindi l'importanza della conoscienza degli algoritmi per problemi di ottimizzazione "facili".

Esercizio 5.28 *Si proponga una funzione intorno basata su mosse di scambio ciclico per il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS), discutendo l'implementazione della procedura σ.*

5.2.3 Metaeuristiche

Anche utilizzando intorni di grande dimensione, quali quelli visti nel paragrafo precedente, tutti gli algoritmi di ricerca locale per problemi di *OC* si arrestano per aver determinato un ottimo locale rispetto alla funzione intorno utilizzata, o, più in generale, perchè non sono in grado di trovare "mosse" che generino soluzioni migliori della soluzione corrente. Qualora ci sia motivo per credere che la soluzione corrente non sia un ottimo globale, si pone quindi il problema di cercare di determinare un diverso, e possibilmente migliore, ottimo locale. In questo paragrafo discuteremo brevemente alcune possibili strategie che permettono di fare questo, cercando di illustrare alcune delle loro principali caratteristiche e potenziali limitazioni. Queste strategie sono chiamate *metaeuristiche* perchè non sono algoritmi specifici per un dato problema, ma metodi generali che possono essere applicati per tentare di migliorarne le prestazioni di molti diversi algoritmi di ricerca locale.

5.2.3.1 Multistart

In generale, la qualità dell'ottimo locale determinato da un algoritmo di ricerca locale dipende da due fattori: l'intorno utilizzato e la soluzione ammissibile iniziale da cui la ricerca parte. Nei paragrafi precedenti ci siamo soffermati principalmente sul primo fattore, assumendo che la soluzione iniziale venisse determinata con qualche euristica, ad esempio di tipo greedy. Come abbiamo visto in molti degli esempi discussi nel paragrafo 5.1.1, spesso è possibile definire più di un algoritmo greedy per lo stesso problema, ed in molti casi gli algoritmi sono anche molto simili, differendo solamente per l'ordine in cui sono compiute alcune scelte, per cui non è irragionevole pensare di avere a disposizione più di un algoritmo in grado di produrre soluzioni iniziali. In questo caso, le soluzioni prodotte saranno normalmente diverse; inoltre, non è detto che l'ottimo locale determinato eseguendo l'algoritmo di ricerca locale a partire dalla migliore delle soluzioni così ottenute sia necessariamente il migliore degli ottimi locali ottenibili eseguendo l'algoritmo di ricerca locale a partire da ciascuna delle soluzioni separatamente. Tutto ciò suggerisce un'ovvia estensione dell'algoritmo di ricerca locale: generare più soluzioni iniziali, eseguire l'algoritmo di ricerca locale a partire da ciascuna di esse, quindi selezionare la migliore delle soluzioni così ottenute.
Questo procedimento è particolarmente attraente quando sia possibile, tipicamente attraverso l'uso di tecniche randomizzate, generare facilmente un insieme arbitrariamente numeroso di soluzioni iniziali potenzialmente diverse. Si pensi ad esempio agli algoritmi greedy di tipo *list scheduling* per (MMMS) presentati al paragrafo 5.1.1.4: è facile costruire un'*euristica randomizzata* per il problema semplicemente costruendo un ordinamento pseudo-casuale dei lavori e poi assegnando i lavori alle macchine (selezionando ogni volta la macchina "più scarica") in quell'ordine. In questo modo è chiaramente possibile produrre un gran numero di soluzioni iniziali diverse, a partire da ciascuna delle quali si può poi eseguire un algoritmo di ricerca locale. Mediante l'uso di numeri pseudo-casuali è di solito facile trasformare euristiche deterministiche per un problema di ottimizzazione in euristiche randomizzate.

Esercizio 5.29 *Si discuta se e come sia possibile trasformare ciascuno degli algoritmi euristici visti nel paragrafo 5.1.1 in euristiche randomizzate.*

La combinazione di un algoritmo di ricerca locale e di un'euristica randomizzata viene denominato *metodo multistart*; quando, come spesso accade, l'euristica randomizzata è di tipo greedy, si parla di *GRASP* (greedy randomized adaptive search procedure). Una volta sviluppata una qualsiasi euristica randomizzata per determinare la soluzione iniziale ed un qualsiasi algoritmo di ricerca locale per un dato problema, è praticamente immediato combinare le due per costruire un metodo multistart.
Sotto opportune ipotesi tecniche è possibile dimostrare che ripetendo un numero sufficientemente alto di volte la procedura, ossia generando un numero sufficientemente alto di volte soluzioni iniziali, si è praticamente certi di determinare una soluzione ottima del problema. Si noti che, al limite, non è neppure necessaria la fase di ricerca locale: un'euristica randomizzata ripetuta più volte è comunque un modo per generare un insieme di soluzioni ammissibili, e se l'insieme è abbastanza ampio è molto probabile che contenga anche la soluzione ottima. Questo risultato non deve illudere sulla reale possibilità di giungere ad una (quasi) certezza di ottimalità utilizzando un metodo di questo tipo: in

generale, il numero di ripetizioni necessarie può essere enorme, in modo tale che risulterebbe comunque più conveniente enumerare tutte le soluzioni del problema.
Le euristiche multistart forniscono quindi un modo semplice, ma non particolarmente efficiente, per cercare di migliorare la qualità delle soluzioni determinate da un algoritmo di ricerca locale. Anche se non esistono regole che valgano per qualsiasi problema, è possibile enunciare alcune linee guida che si sono dimostrate valide per molti problemi diversi. In generale, il meccanismo della ripartenza da un punto casuale non fornisce un sostituto efficiente di una ricerca locale ben congegnata: se si confrontano le prestazioni di un metodo multistart che esegue molte ricerche locali con intorni "piccoli" e di uno che esegue poche ricerche locali con intorni "grandi", è di solito il secondo a fornire soluzioni migliori a parità di tempo totale utilizzato. Questo comportamento è spiegato dal fatto che una ripartenza da un punto pseudo-casuale "cancella la storia" dell'algoritmo: l'evoluzione successiva è completamente indipendente da tutto quello che è accaduto prima della ripartenza. In altri termini, il metodo multistart non è in grado di sfruttare in alcun modo l'informazione generata durante la ricerche locali precedenti per "guidare" la ricerca locale corrente. Ad esempio, è noto che per molti problemi le soluzioni di buona qualità sono normalmente abbastanza "vicine" le une alle altre (in termini di numero di mosse degli intorni utilizzati); quindi, l'aver determinato una "buona" soluzione fornisce una qualche forma di informazione che il metodo multistart non tiene in alcun modo in considerazione. Al contrario, la ricerca locale ha un qualche tipo di informazione sulla "storia" dell'algoritmo, in particolare data dalla soluzione corrente. La capacità di sfruttare l'informazione contenuta nelle soluzioni precedentemente generate è in effetti uno degli elementi che contraddistinguono le metaeuristiche più efficienti, quali quelle discusse nel seguito.

5.2.3.2 Simulated annealling

L'idea di base del *simulated annealling* è quella di modificare l'algoritmo di ricerca locale sostituendo i criteri deterministici di selezione del nuovo punto nell'intorno corrente e di accettazione della mossa con un criteri randomizzati. Uno schema generale di algoritmo di tipo simulated annealling (per un problema di minimo) è il seguente:

```
Procedure Simulated-Annealling( F , x )  {
  x = Ammissibile(F); x* = x; c = InitTemp();
  while( c ≥ c̄ ) do {
    for i = 1 to k(c) do
      ⟨ seleziona x' ∈ I(x) in modo pseudocasuale ⟩;
      if( c(x') < c(x*) ) then x* = x';
      if( c(x') < c(x) )  then x = x';                          /* downhill */
                          else { r = random(0,1);
                                 if( r < e^{-(c(x')-c(x))/c} ) then x = x';   /* uphill */
                                 }
    ⟨ decrementa c ⟩;
    }
  x = x*;
  }
```

Procedura 5.5: Algoritmo *Simulated annealling*

L'algoritmo inizia determinando una soluzione iniziale ed un opportuno valore del parametro "temperatura" c, che controlla l'accettazione di mosse che peggiorano il valore della funzione obiettivo. L'algoritmo esegue quindi un certo numero di "fasi", all'interno della quali la "temperatura" è costante. Ad ogni iterazione si seleziona in modo pseudo-casuale un punto nell'intorno del punto corrente: se questo punto ha un miglior valore della funzione obiettivo (si tratta cioè di una normale mossa di ricerca locale, o "downhill") viene certamente accettato come il punto corrente. In caso contrario il punto può essere ugualmente accettato come punto corrente (mossa "uphill"), basandosi sull'estrazione di un numero pseudo-casuale: la probabilità che il punto sia accettato dipende dal peggioramento della funzione obiettivo e dal valore di c. Una fase in cui la temperatura è costante termina quando il siste-

ma raggiunge uno "stato stazionario"[1], ossia quando si suppone che il valore della funzione obiettivo della soluzione corrente x sia sufficentemente "vicino"—usando c come "unità di misura"—a quello della soluzione ottima; in pratica si fissa il numero di iterazioni di ogni fase secondo regole opportune. Alla fine di ogni fase la temperatura viene diminuita, rendendo meno probabile l'accettazione di mosse che peggiorino sensibilmente il valore della funzione obiettivo, e quindi concentrando la ricerca su soluzioni più "vicine" a quella corrente. L'algoritmo termina quando la temperatura è sufficientemente bassa (il sistema è "congelato"), riportando la migliore delle soluzioni trovate: nell'ultima fase di fatto l'algoritmo si comporta in modo simile ad un normale algoritmo di ricerca locale.

L'idea alla base dell'algoritmo è quella di permettere consistenti peggioramenti del valore della funzione obiettivo nelle fasi iniziali dell'esecuzione, in modo da evitare di rimanere intrappolati in ottimi locali molto "lontani" dall'ottimo globale. Dopo un numero sufficiente di iterazioni l'algoritmo dovrebbe avre raggiunto una parte dello spazio delle soluzioni "vicina" all'ottimo globale: a quel punto la temperatura viene diminuita per raffinare la ricerca. L'algoritmo prende spunto da un metodo usato in pratica per produrre cristalli con elevato grado di regolarità: si scalda il materiale in questione per rompere i legami chimici non desiderati (di più alta energia), si lascia ad una certa temperatura per un tempo sufficientemente lungo affinchè si creino con alta probabilità i legami chimici desiderati (di più bassa energia), quindi si raffredda il materiale per rendere più stabili i legami chimci di bassa energia, ripetendo il tutto finchè, sperabilmente, il materiale contiene solo legami di più bassa energia.

Come per il caso del multistart, anche il simulated annealling possiede, sotto opportune ipotesi tecniche, interessanti proprietà teoriche. In particolare, si può dimostrare che esiste una costante C tale che se la temperatura decresce non più rapidamente di $C/\log(k)$, dove k è il numero di iterazioni compiute, allora l'algoritmo determina una soluzione ottima con probabilità pari a uno. Come nel caso del multistart, però, queste proprietà teoriche non hanno grande utilità in pratica: la *regola di raffreddamento* sopra indicata corrisponde ad una diminuzione molto lenta della temperatura, e quindi ad un numero di iterazioni talmente grande da rendere più conveniente l'enumerazione di tutte le soluzioni ammissibili. In pratica si usano quindi *regole di raffreddamento esponenziali* in cui la temperatura viene moltiplicata per un fattore $\theta < 1$ dopo un numero fissato di iterazioni, e la temperatura iniziale c viene scelta come la maggior differenza di costo possibile tra due soluzioni x ed x' tali che $x' \in I(x)$ (in modo tale da rendere possibile, almeno inizialmente, qualsiasi mossa). In questo modo non si ha alcuna certezza, neanche in senso stocastico, di determinare una soluzione ottima, ma si possono ottenere euristiche di buona efficienza in pratica.
Gli algoritmi di tipo simulated annealling hanno avuto un buon successo in alcuni campi di applicazioni dell'Ottimizzazione Combinatoria, ad esempio nella progettazione di circuiti VLSI. La ragione di questo successo risiede nel fatto che sono relativamente semplici da implementare, poco dipendenti dalla struttura del problema e quindi facilmente adattabili a problemi diversi ed abbastanza "robusti", nel senso che, se lasciati in esecuzione sufficientemente a lungo, solitamente producono soluzioni di buona qualità in quasi tutti i problemi in cui sono applicati. Un vantaggio in questo senso è il fatto che contengano relativamente pochi parametri, ossia la temperatura iniziale, il valore di θ e la durata di ciascuna fase. Ogniqualvolta il comportamento di un algoritmo dipende da parametri che possono essere scelti in modo arbitrario (sia pur seguendo certe regole) si pone infatti il problema di determinare un valore dei parametri che produce soluzioni di buona qualità quanto più rapidamente possibile per le istanze che si intende risolvere. Per fare questo è normalmente necessario eseguire l'algoritmo molte volte su un nutrito insieme di istanze di test con combinazioni diverse dei parametri, in modo da ottenere informazione sull'impatto dei diversi parametri sull'efficacia ed efficienza dell'algoritmo. Questo processo può richiedere molto tempo, specialmente se l'algoritmo è poco "robusto", ossia il suo comportamento varia considerevolmente per piccole variazioni dei parametri o, con gli stessi parametri, per istanze simili. Il simulated annealling risulta piuttosto "robusto" da questo punto di vista, anche per la disponibilità di linee guida generali per l'impostazione dei parametri che si sono rivelate valide in molti diversi contesti. In linea generale, si può affermare che normalmente la tecnica del

[1] La successione dei passi compiuti dall'algoritmo quando c è costante è un processo stocastico noto come *Catena di Markov*, del quale sono note molte importanti proprietà statistiche; in particolare, dopo un numero opportuno di passi il sistema tende a raggiungere uno stato stazionario indipendentemente dal punto di partenza.

simulated annealling permette di migliorare la qualità delle soluzioni fornite da un algoritmo di ricerca locale con una ragionevole efficienza complessiva. Il metodo risulta molto spesso più efficiente dei metodi multistart basati sullo stesso intorno, poiché mantiene una maggiore quantità di informazione sulla storia dell'algoritmo – non solo la soluzione corrente, ma anche la temperatura – che in qualche modo "guida" la ricerca di una soluzione ottima. Questo tipo di tecnica risulta particolarmente adatta a problemi molto complessi, di cui sia difficile sfruttare la struttura combinatoria, e nel caso in cui l'efficacia del metodo (la qualità della soluzione determinata) sia più importante della sua efficienza (la rapidità con cui la soluzione è determinata). In casi diversi altri tipi di tecniche, come quelle discusse nel seguito, possono risultare preferibili; questo è giustificato dal fatto che l'informazione sulla storia dell'algoritmo mantenuta da un algoritmo di tipo simulated annealling è molto "aggregata", e quindi non particolarmente efficiente nel guidare la ricerca. Inoltre, il simulated annealling si basa fondamentalmente su decisioni di tipo pseudo-casuale, il che tipicamente produce un comportamento "medio" affidabile ma difficilmente è più efficiente di decisioni deterministiche guidate da criteri opportuni. Infine, la generalità del metodo, ossia il fatto che l'algoritmo sia largamente indipendente dal problema, implica che il metodo non sfrutta appieno la struttura del problema; la capacità di sfruttare quanto più possibile tale struttura è uno dei fattori fondamentali che caratterizzano gli algoritmi più efficienti.

5.2.3.3 Ricerca Taboo

Un problema fondamentale che deve essere superato quando si intende modificare un algoritmo di ricerca locale consiste nel fatto che accettare mosse che peggiorano il valore della funzione obiettivo ("uphill") pone ad immediato rischio di ritornare sulla precedente soluzione corrente, e quindi di entrare in un ciclo in cui si ripetono sempre le stesse soluzioni. Infatti, normalmente accade che $x' \in I(x)$ implica che $x \in I(x')$; di conseguenza, qualora si abbia $c(x') > c(x)$ (in un problema di minimo) e si accetti ugualmente di porre x' come punto corrente, all'iterazione successiva si avrà la soluzione x nell'intorno del punto corrente con un valore minore della funzione obiettivo, e quindi sarà possibile (e probabile) effettuare una mossa "downhill" ritornando su x. Le tecniche di tipo simulated annealling non sono immuni da questo problema, ma sono basate su decisioni stocastiche, per cui la ripetizione del punto corrente non causa necessariamente l'entrata in un ciclo. Infatti, nella situazione precedente il nuovo punto corrente viene scelto in modo casuale nell'intorno di x', per cui anche se $c(x)$ fosse l'unico punto in $I(x')$ con valore migliore della funzione obiettivo non sarebbe necessariamente scelto; anche qualora ciò accadesse, comunque, il nuovo punto x'' scelto nell'intorno di x sarebbe con alta probabilità diverso da x', e quindi la probabilità di ritornare un numero molto alto di volte sullo stesso punto corrente x è estremamente bassa.

Qualora si vogliano implementare algoritmi deterministici questo tipo di considerazione non è più valida, ed è quindi necessario porre in effetto tecniche che impediscano, o comunque rendano altamente improbabile, l'entrata in un ciclo. La più diffusa da queste tecniche è quella delle *mosse Taboo*, che caratterizza un'ampia famiglia di algoritmi detti di *ricerca Taboo* (TS, da Taboo Search). Questi algoritmi sono, in prima approssimazione, normali algoritmi di ricerca locale, in cui cioè si compiono mosse "downhill" fin quando ciò sia possibile; quando però il punto corrente è un ottimo locale per l'intorno utilizzato, e quindi un normale algoritmo di ricerca locale terminerebbe, un algoritmo TS seleziona comunque una soluzione $x' \in I(x)$ secondo criteri opportuni e compie una mossa "uphill". Per evitare i cicli, l'algoritmo mantiene una *lista Taboo* che contiene una descrizione delle mosse "uphill"; ad ogni iterazione, nel cercare una soluzione $x' \in I(x)$ con $c(x') < c(x)$, l'algoritmo controlla la lista Taboo, e scarta tutte le soluzioni che sono generate da una mossa Taboo. Siccome l'algoritmo permette mosse "uphill", non è garantito che la soluzione corrente al momento in cui l'algoritmo termina sia la migliore determinata; come nel caso del simulated annealling, si mantiene quindi, oltre alla soluzione corrente, la miglior soluzione x^* tra tutte quelle determinate. In prima approssimazione si potrebbe pensare che la lista Taboo contenga tutte le soluzioni precedentemente trovate con valore della funzione obiettivo migliore di quello della soluzione corrente; questo chiaramente eviterebbe i cicli, ma normalmente risulta eccessivamente costoso, sia in termini di memoria richiesta che in termini

del costo di verificare che una data soluzione appartenga alla lista. In generale si preferisce quindi mantenere nella lista Taboo una descrizione, anche parziale delle mosse (che definiscono la funzione intorno) che hanno generato la soluzione corrispondente. Questa descrizione dipende quindi dalla particolare funzione intorno utilizzata, e deve essere "compatibile" con la funzione σ, ossia non deve rendere "eccessivamente più costosa" la determinazione della nuova soluzione.

Si consideri ad esempio il problema del (TSP) e la semplice funzione intorno basata su mosse di 2-scambio descritta nel §5.2.1.3, ossia sullo scambio della coppia di lati $\{\{i,j\},\{h,k\}\}$ con $\{\{i,k\},\{h,j\}\}$. Una possibile implementazione della lista Taboo per questo caso potrebbe contenere semplicemente la coppia $\{\{i,k\},\{h,j\}\}$ che caratterizza la mossa: una mossa di 2-scambio sarebbe dichiarata Taboo—e quindi scartata—se coinvolgesse esattamente quei due lati. Si noti che questa scelta può impedire di generare molte soluzioni oltre a quella che originariamente ha causato l'inserzione di una data coppia di lati nella lista Taboo. Infatti, se x ed x' sono rispettivamente il punto corrente ed il nuovo punto corrispondenti ad una mossa "uphill", ed $x'' \neq x$, $x'' \in I(x')$ è il punto accettato all'iterazione successiva, allora effettuare la mossa Taboo (qualora possibile) non genererebbe x, e quindi non causerebbe necessariamente l'entrata in un ciclo. Quindi, questa implementazione della lista Taboo è in qualche modo "eccessiva" rispetto al compito di evitare i cicli; ciononostante può risultare opportuna perchè può essere facilmente ed efficientemente integrata nel calcolo della funzione σ.

In effetti, per semplificare il controllo della lista Taboo non è infrequente che si implementi una lista Taboo che contiene una *descrizione parziale* delle mosse utilizzate. Si consideri ad esempio il problema (MMMS) e la funzione intorno basata su mosse di "spostamento" e "scambio" descritta nel §5.2.1.2. Una possibile implementazione della lista Taboo per questo caso potrebbe contenere semplicemente coppie (d,i) dove d è una durata di lavoro ed i è una macchina: è considerata una mossa Taboo assegnare alla macchina i un lavoro di durata d. Questa è ovviamente una descrizione parziale sia di una mossa di spostamento che di scambio: se si pone (d,i) nella lista Taboo ogniqualvolta si elimina un lavoro di durata d dalla lista di quelli assegnati alla macchina i, si evitano sicuramente i cicli. Analogamente a quello che abbiamo visto per il (TSP), questa implementazione impedisce di generare molte altre soluzioni oltre a quelle già visistate dall'algoritmo.

Inserire mosse nella lista Taboo diminuisce il numero di soluzioni considerate ad ogni passo come possibili nuove soluzioni correnti: questo può seriamente limitare la capacità dell'algoritmo di scoprire soluzioni migliori. Per questo sono necessari dei meccanismi che limitino l'effetto della lista Taboo. Il primo meccanismo, usato in tutti gli algoritmi di TS, consiste semplicemente nel limitare la massima lunghezza della lista Taboo ad una costante fissata: quando la lista ha raggiunto tale lunghezza ed una nuova mossa deve essere resa Taboo, la più "vecchia" delle mosse nella lista viene nuovamente resa ammissibile. In generale non è facile determinare valori della massima lunghezza della lista delle mosse Taboo che garantiscano che l'algoritmo non entri in un ciclo; in pratica, però, gli algoritmi di tipo TS non entrano in ciclo se la lista Taboo contiene anche solo poche decine di mosse. Ciò dipende dal fatto che il numero di possibili soluzioni che possono essere visitate con k mosse di ricerca locale aumenta esponenzialmente in k, e quindi è molto poco probabile che l'algoritmo visiti esattamente, tra tutte quelle possibili, una delle soluzioni già generate. Limitare la lunghezza della lista Taboo ha anche, chiaramente, un effetto positivo sulle prestazioni dell'algoritmo: normalmente, il costo di verificare che una mossa non sia Taboo cresce con la lunghezza della lista, e quindi ciò potrebbe risultare molto costoso qualora la liste divenisse molto lunga.

Un secondo metodo per limitare gli effetti negativi della lista Taboo è quello di inserire un cosiddetto *criterio di aspirazione*, ossia un insieme di condizioni logiche tale per cui se la soluzione $x' \in I(x)$ generata da una mossa soddisfa almeno una di queste condizioni, allora x' può essere considerata tra i possibili candidati a divenire la prossima soluzione corrente anche se la mossa appartiene alla lista Taboo. Ovviamente, il criterio di aspirazione deve essere scelto in modo tale da garantire—o rendere molto probabile—che l'algoritmo non entri in un ciclo: un possibile criterio di aspirazione è ad esempio $c(x') < c(x^*)$, dove x^* è la migliore delle soluzioni generate, dato che in questo caso x' non può sicuramente essere stata visitata in precedenza. Sono comunque possibili anche altri criteri di aspirazione: come il nome suggerisce, e l'esempio precedente mostra, i criteri di aspirazione corrispondono spesso a condizioni che richiedono che x' sia sotto qualche aspetto una "buona" soluzione. Ad esempio, nel

problema (MMMS) si potrebbe permettere di assegnare un lavoro di durata d ad una data macchina i anche se (d, i) fa parte della lista Taboo purchè il tempo di completamento della macchina i dopo l'assegnamento sia minore del tempo di completamento che la macchina aveva al momento in cui (d, i) è stata inserita nella lista delle mosse Taboo (questo richiederebbe chiaramente di memorizzare anche tale durata nella lista). Chiaramente questo evita i cicli in quanto la soluzione ottenuta dalla mossa è "localmente", ossia limitatamente alla macchina i, migliore di quella che c'era al momento in cui la mossa Taboo è stata effettuata.
Oltre a queste componenti base, gli algoritmi TS spesso contengono altre strategie che risultano utili per rendere il processo di ricerca più efficiente. Ad esempio, per quei problemi in cui le "buone" soluzioni sono spesso "vicine" sono spesso utilizzate tecniche di *intensificazione*, che concentrano temporaneamente la ricerca "vicino" alla soluzione corrente. Questo può essere ottenuto ad esempio restringendo la dimensione dell'intorno, oppure modificando la funzione obiettivo con un termine che penalizza leggermente le soluzoni dell'intorno più "lontane" da quella corrente. L'intensificazione viene solitamente effettuata quando si determina una nuova miglior soluzione x^*, e mantenuta per un numero limitato di iterazioni, nella speranza che vicino alla migliore soluzione determinata fino a quel momento ci siano altre soluzioni ancora migliori. Una strategia in un certo senso opposta è quella della *diversificazione*, che cerca di indirizzare la ricerca verso aree dello spazio delle soluzioni diverse da quelle esplorate fino a quel momento. Modi possibili per implementare strategie di diversificazione sono ad esempio penalizzare leggermente le soluzioni dell'intorno "vicine" alla soluzione corrente x, allargare la dimensione dell'intorno, effettuare in una sola iterazione combinazioni di molte mosse, ad esempio selezionandole in modo randomizzate, per "saltare" in una zona dello spazio delle soluzioni "lontana" da x, o persino effettuare una ripartenza pseudo-casuale come nel metodo multistart. Normalmente, vengono effettuate mosse di diversificazione quando per "molte" iterazioni successive la migliore soluzione trovata x^* non cambia, suggerendo che l'area dello spazio delle soluzioni in cui si trova la soluzione corrente sia già stata sostanzialmente esplorata, e che sia quindi necessario visitarne una diversa. Nell'implementare tecniche di intensificazione e diversificazione risulta spesso utile sfruttare l'informazione contenuta nella sequenza di soluzioni generate, ad esempio associando a ciascun costituente elementare delle soluzioni un qualche valore numerico che ne rappresenti in qualche modo "l'importanza". Si pensi ad esempio di applicare un qualche algoritmo di ricerca locale per risolvere il problema del commesso viaggiatore, e di mantenere per ogni arco un contatore che indichi di quante soluzioni correnti l'arco ha fatto parte fino a quel momento: archi con un valore alto del contatore sono in qualche senso "più rilevanti" di archi con un valore basso del contatore. In una procedura di diversificazione potrebbe essere ragionevole concentrarsi su archi con valore alto del contatore, in modo da esplorare soluzioni "diverse" da quelle visitate fino a quel momento. Alternativamente si potrebbe contare per ogni arco di quante delle migliori k soluzioni generate ha fatto parte, dove k è un valore fissato; più in generale, si potrebbe mantenere per ogni arco un valore numerico calcolato a partire dal costo delle soluzioni di cui ha fatto parte, in modo tale che il valore sia "alto" per archi cha fanno "spesso" parte di soluzioni "buone". Con un'opportuna scelta della funzione e dei parametri, questo tipo di informazione sulle soluzioni generate dall'algoritmo può rivelarsi utile per guidare la ricerca.
Infine, una strategia che talvolta si rivela utile è quella di permettere che la soluzione corrente x diventi inammissibile per un numero limitato di iterazioni, se questo serve a migliorare sensibilmente il valore della soluzione obiettivo. Questo viene fatto per i problemi in cui sia particolarmente difficile costruire soluzioni ammissibili: le "poche" soluzioni ammissibili del problema possono quindi essere "lontane", ed è necessario permettere mosse in cui si perde l'ammissibilità, seguite da una successione di mosse in cui si cerca di riguadagnarla.

È chiaro come, rispetto ad un algoritmo di tipo simulated annealling, un algoritmo TS sia più complesso da realizzare. L'implementazione della lista Taboo, e spesso anche dei criteri di aspirazione, è fortemente dipendente dalla scelta della funzione intorno, e quindi l'algoritmo deve essere progettato tenendo in conto di tutti questi aspetti contemporaneamente. Normalmente, anche le strategie di intensificazione e diversificazione dipendono in qualche modo dall'intorno utilizzato, per cui anche questi aspetti devono essere considerati insieme agli altri. Per tutte le strategie sopra elencate è

normalmente necessario determinare sperimentalmente diversi parametri, quali la lunghezza della lista Taboo, il numero di iterazioni per l'intensificazione, il numero di iterazioni per la diversificazione, e così via: il numero complessivo di parametri può essere alto, il che può rendere difficoltosa la determinazione di "buoni" valori per tutti i parametri. La grande flessibilità data dai molti parametri, ed il fatto che ogni aspetto di un algoritmo TS debba essere adattato allo specifico problema di OC risolto, sono però anche le caratteristiche che possono rendere questi algoritmi molto efficienti in pratica: un algoritmo TS ben congegnato ed in cui i parametri siano scelti in modo opportuno si rivela spesso più efficiente, ad esempio, di un algoritmo di tipo simulated annealling per lo stesso problema che usi la stessa funzione intorno, nel senso che produce soluzioni di migliore qualità a parità di tempo. Qualora l'efficienza sia almeno altrettanto importante della qualità delle soluzioni fornite, quindi, può essere ragionevole investire le risorse necessarie a realizzare un algoritmo TS.

5.2.3.4 Algoritmi genetici

Uno dei principali fattori che determina l'efficienza di un algoritmo di ricerca locale è la capacità di sfruttare l'informazione contenuta nella sequenza di soluzioni generate per guidare la ricerca di soluzioni migliori. Come abbiamo visto, le diverse strategie algoritmiche ottengono questo risultato in gradi diversi, e con diversi accorgimenti. Esiste una classe di algoritmi euristici, che in linea di principio *non* sono algoritmi di ricerca locale, che portano questo concetto alle estreme conseguenze: si tratta dei cosiddetti *algoritmi genetici*, che cercano di riprodurre alcuni dei meccanismi che si ritiene stiano alla base dell'evoluzione delle forme di vita.

La caratteristica peculiare di questo tipo di algoritmi è di mantenere non una singola soluzione corrente ma una *popolazione* di soluzioni. L'algoritmo procede per fasi, corrispondenti a "generazioni" nella popolazione. In ogni fase vengono ripetute un numero opportuno di volte le seguenti operazioni:

- all'interno della popolazione vengono selezionate in modo pseudo-casuale due (o più) soluzioni "genitrici";
- a partire da tali soluzioni vengono generate in modo pseudo-casuale un certo numero di soluzioni "discendenti", ottenute "mescolando" le caratteristiche di tutte le soluzioni genitrici;
- a ciascuna soluzione così generata vengono applicate alcune "mutazioni casuali" che cercano di introdurre nella popolazione caratteristiche altrimenti non presenti.

Alla fine di ogni fase si procede alla *selezione*: a ciascuna delle soluzioni, sia quelle presenti nella popolazione all'inizio della fase che quelle generate durante la fase, viene associato un *valore di adattamento* (fitness), ad esempio il valore della funzione obiettivo, e vengono mantenute nella popolazione (sopravvivono alla generazione successiva) solo le soluzioni con miglior fitness, mantenendo costante la dimensione della popolazione; alternativamente, le soluzioni sopravvissute sono selezionate in modo pseudo-casuale con probabilità dipendente dal fitness. Sono state poi proposte molte varianti a questo schema base, ad esempio suddividendo la popolazione in sotto-popolazioni più piccole con scambi limitati (in modo da simulare le barriere geografiche presenti in natura) o imponendo la "morte" delle soluzioni dopo un numero massimo di generazioni indipendentemente dal loro valore di fitness.
Tutte le operazioni di un algoritmo genetico devono essere specializzate per il problema da risolvere, anche se per alcune esistono implementazioni abbastanza standard. Ad esempio, la selezione delle soluzioni genitrici viene normalmente effettuata semplicemente estraendo in modo casuale dalla popolazione; la probabililità di essere estratta può essere uniforme oppure dipendere dal valore di fitness. Analogamente, la selezione alla fine di ogni fase è normalmente effettuata, in modo deterministico o pseudo-casuale, semplicemente privilegiando le soluzioni con miglior fitness. Per quanto riguarda la definizione del valore di fitness, spesso si usa semplicemente il valore della funzione obiettivo. In alcuni casi può risultare opportuno modificare la funzione obiettivo per premiare soluzioni che, a parità di funzione obiettivo, appaiano più desiderabili: ad esempio, per il problema (MMMS) si può assegnare un valore di fitness leggermente più alto, a parità di makespan, a quelle soluzioni in cui le macchine

che non contribuiscono a determinare il makespan sono "bilanciate", ossia hanno tutte tempo di completamento simile. Per i problemi in cui sia particolarmente difficile costruire soluzioni ammissibili è possibile ammettere nella popolazione soluzioni non ammissibili, penalizzando opportunamente la non ammissibilità nel corrispondente valore di fitness. La generazione di soluzioni e le mutazioni casuali, invece, sono strettamente dipendenti dal problema. Questo non è stato inizialmente ben compreso: dato che qualsiasi soluzione ammissibile rappresentata in un computer può essere vista come una stringa di bits, si è sostenuto che le operazioni di generazione e mutazione potessero essere implementate in modo generico, ad esempio rimpiazzando nel "patrimonio genetico" di un genitore una o più sottostringhe di bits con quelle prelevate dalle corrispondenti locazioni nel "patrimonio genetico" dell'altro genitore, e cambiando il valore di un piccolo numero dei bits scelti in modo pseudo-casuale. Questo risulta in effetti possibile per problemi con "pochi vincoli", in cui sia molto facile costruire una soluzione ammissibile. Si pensi ad esempio a (MMMS): una soluzione ammissibile è data da un qualisasi assegnamento dei lavori alle macchine, ossia da qualsiasi vettore $s[\cdot]$ di n componenti in cui ogni elemento $s[i]$ appartenga all'insieme $\{1, \ldots, m\}$, col significato che $s[i] = h$ se e solo se il lavoro i è assegnato alla macchina h. Dati due vettori $s_1[\cdot]$ ed $s_2[\cdot]$ di questo tipo, è molto facile produrre un nuovo vettore $s_3[\cdot]$ possibilmente diverso da entrambe: ad esempio si possono selezionare in modo pseudo-casuale due numeri $1 \leq i \leq j \leq n$ e porre $s_3[h] = s_1[h]$ per $i \leq h \leq j$ e $s_3[h] = s_2[h]$ per tutti gli altri indici h. Le mutazioni possono poi essere ottenute semplicemente cambiando in modo pseudo-casuale il valore di un piccolo numero di elementi di $s_3[\cdot]$, scelti in modo pseudo-casuale. Nella maggioranza dei casi questo però non risulta effettivamente possibile: per la maggior parte dei problemi, operando in modo analogo a quanto appena visto si ottengono quasi sempre soluzioni non ammissibili. È quindi necessario sviluppare euristiche specifiche che costruiscano soluzioni ammissibili cercando di replicare, per quanto possibile, le caratteristiche di entrambe i genitori.

Si consideri ad esempio il problema del (TSP): dati due cicli Hamiltoniani diversi, si devono produrre uno o più cicli che "combinino" le caratteristiche dei due genitori. Un possibile modo di procedere è quello di considerare fissati, nei cicli "figli", tutti gli archi che sono comuni ad entrambe i genitori, e poi cercare di completare questa soluzione parziale fino a formare un ciclo Hamiltoniano mediante un'euristica costruttiva randomizzata analoga alla "Nearest Neighbour" descritta nel paragrafo 5.1.1.3. Definito un nodo "corrente" iniziale i, l'euristica prosegue sicuramente sul nodo j se il lato $\{i, j\}$ appartiene ad entrambe i genitori. Se invece il "successore" di i nei due cicli Hamiltoniani è diverso, l'euristica seleziona uno dei due successori che non sia ancora stato visitato secondo un qualche criterio opportuno; ad esempio, quello a cui corrisponde il lato di costo minore, oppure in modo pseudo casuale con probabilità uniforme, oppure con probabilità dipendente dal costo del lato, oppure ancora con probabilità dipendente dal valore della funzione obiettivo del "genitore" a cui il lato corrispondente appartiene (favorendo le scelte del "genitore migliore"). Qualora entrambe i successori siano stati già visitati l'euristica selezionerà un diverso nodo secondo altri criteri (ad esempio quello "più vicino); se tutti i nodi raggiungibili da i sono già visitati (e non si è formato un ciclo) l'euristica fallirà, "abortendo" la generazione della nuova soluzione.

Una procedura in qualche modo analoga si può utilizzare nel caso di (CMST). Le due soluzioni "genitrici" sono alberi di copertura diversi dello stesso grafo: esse avranno quindi un certo numero di sottoalberi comuni, che risulterebbero quindi sicuramente ammissibili se fossero utilizzate come sottoalberi della radice (al limite i sottoalberi potranno essere formati da un solo nodo). In ciascun sottoalbero, il predecessore di tutti i nodi tranne la radice è identico nelle due soluzioni, mentre la radice ha due predecessori diversi, uno per ciascuna soluzione. Si può quindi realizzare una procedura euristica che tenti di combinare questi sottoalberi per formare una soluzione ammissibile: selezionata la radice i di un sottoalbero, l'euristica seleziona uno dei due predecessori secondo un qualche criterio opportuno, ad esempio quello a cui corrisponde l'arco di costo minore, oppure in modo pseudo-casuale con probabilità uniforme, oppure con probabilità dipendente dal costo dell'arco, oppure ancora con probabilità dipendente dal valore della funzione obiettivo del "genitore" a cui l'arco corrispondente appartiene. Naturalmente un predecessore può essere selezionato solamente se ciò non crea un sottoalbero di peso superiore alla soglia massima fissata: se ciò accade per entrambe i predecessori corrispondenti alle soluzioni genitrici l'euristica tenterà di selezionare un diverso predecessore – al li-

mite, la radice dell'albero – mediante criteri opportuni. Se nessun predecessore può essere selezionato senza violare i vincoli l'euristica fallirà, "abortendo" la generazione della nuova soluzione.
Si noti che euristiche simili possono essere utilizzate anche per problemi in cui sia "facile" produrre soluzioni ammissibili. Nel caso di (MMMS), ad esempio, si potrebbe utilizzare un approccio simile alle euristiche list scheduling viste al paragrafo 5.1.1.4: ordinati i lavori secondo un qualche criterio, si potrebbe assegnare ciascun lavoro h ad una delle due macchine $s_1[h]$ ed $s_2[h]$, scegliendo quella che ha il tempo di completamento minimo oppure con un qualche criterio pseudo-casuale analogo a quelli visti precedentemente (i lavori che siano assegnati alla stessa macchina in entrambe le soluzioni genitrici sarebbero quindi sicuramente assegnati a quella macchina in tutte le soluzioni discendenti).

Le mutazioni casuali sono normalmente più facili da realizzare, soprattutto qualora si disponga già di un algoritmo di ricerca locale per il problema. Infatti, un modo tipico per realizzare mutazioni è quello di effettuare un piccolo numero di mosse scelte in modo pseudo-casuale, a partire dalla soluzione generata. Ad esempio, per il (TSP) si potrebbero effettuare un piccolo numero di 2-scambi tra coppie $\{i, j\}$ e $\{h, k\}$ di lati del ciclo selezionate in modo pseudo-casuale; analogamente, per (CMST) si potrebbero effettuare un piccolo numero di mosse di "Cut & Paste" tra coppie di nodi i e j scelte in modo pseudo-casuale. Si noti che nel caso del (CMST) le mosse di "Cut & Paste" potrebbero produrre una soluzione non ammissibile: in questo caso si ha la scelta tra non effettuare la mossa oppore "abortire" la soluzione così generata e passare a generarne un'altra.

Infine, qualsiasi algoritmo genetico richiede una prima fase in cui viene generata la popolazione iniziale. Ciò può essere ottenuto in vari modi: ad esempio, si può utilizzare un'euristica randomizzata analogamente a quanto visto per il metodo multistart. Alternativamente si può utilizzare un algoritmo di ricerca locale, eventualmente con multistart, tenendo traccia di un opportuno sottoinsieme delle soluzioni visitate, che costituiranno a terminazione la popolazione iniziale per l'algoritmo genetico.

Anche gli algoritmi genetici, come quelli di ricerca Taboo, dipendono da un numero di parametri il cui valore può non essere facile da fissare. Un parametro molto importante è ad esempio la cardinalità della popolazione: popolazioni troppo piccole non permettono di diversificare a sufficienza la ricerca, che può rimanere "intrappolata" in una zona dello spazio delle soluzioni, mentre una popolazione troppo numerosa può rendere poco efficiente l'algoritmo. Altri parametri importanti riguardano il numero di nuove soluzioni costruite ad ogni generazione, i parametri per la scelta dei genitori e la selezione e così via.

Gli algoritmi genetici "puri", ossia che seguono lo schema precedente, possono rivelarsi efficienti ed efficaci; spesso, però algoritmi basati sulla ricerca locale risultano maggiormente efficienti. In effetti, gli algoritmi genetici di maggior successo sono normalmente quelli che integrano entrambe le tecniche: ad ogni soluzione generata, dopo la fase di mutazione, viene applicata una procedura di ricerca locale per tentare di migliorarne il valore di fitness. Un algoritmo di questo tipo può anche essere considerato un metodo "multistart con memoria", in cui cioè le soluzioni di partenza del metodo multistart non sono selezionate in modo completamente pseudo-casuale, ma si cerca di utilizzare l'informazione precedentemente generata. La combinazione degli algoritmi genetici e degli algoritmi di ricerca locale può assumere molte forme diverse. I casi estremi corrispondono all'algoritmo genetico "puro", in cui cioè non si effettua nessuna ricerca locale, ed all'algoritmo di ricerca locale "puro", in cui cioè la popolazione è formata da un solo individuo. I metodi ibridi si differenziano fondamentalmente per la quantità di risorse che vengono spese nell'uno e nell'altro tipo di operazione: un algoritmo in cui la ricerca locale sia molto semplice, ed eventualmente eseguita comunque per al più un numero fissato di mosse, avrà un comportamento più simile a quello di un algoritmo genetico "puro", mentre un algoritmo in cui la ricerca locale sia effettuata in modo estensivo – ad esempio con tecniche Taboo e con intorni di grande dimensione – avrà un comportamento più simile a quello di un algoritmo di ricerca locale "puro" con multistart. Esiste inoltre la possibilità di eseguire sequenzialmente un algoritmo genetico ed uno di ricerca locale, eventualmente più volte: l'algoritmo di ricerca locale fornisce la popolazione iniziale per quello genetico e questi fornisce una nuova soluzione iniziale per la ricerca locale una volta che questa si sia fermata in un ottimo locale. Dato che esistono moltissimi modi per combinare i due approcci, è molto difficile fornire linee guida generali; ciò è particolarmente vero in quando in un

approccio ibrido è necessario determinare il valore di molti parametri (per entrambe gli algoritmi e per la loro combinazione), rendendo la fase di messa a punto dell'algoritmo potenzialmente lunga e complessa. Solo l'esperienza può quindi fornire indicazioni sulla migliore strategia da utilizzare per un determinato problema di ottimizzazione. Si può però affermare che un'opportuna combinazione delle tecniche descritte in questo capitolo può permettere di determinare efficientemente soluzioni ammissibili di buona qualità per moltissimi problemi di ottimizzazione, per quanto al costo di un consistente investimento nello sviluppo e nella messa a punto di approcci potenzialmente assai complessi.

Riferimenti Bibliografici

E. Aarts and J.K. Lenstra (eds.) "**Local Search in Combinatorial Optimization**", *J. Wiley & Sons*, 1997.

F. Maffioli "**Elementi di Programmazione Matematica**", *Casa Editrice Ambrosiana*, 2000.

V. Varzirani "**Approximation Algorithms**", *Springer-Verlag*, 2001.