1. Greedy

Nella vita di tutti i giorni, si adottano strategie che danno immediatamente buoni risultati, ma che a lungo andare si rivelano scadenti. È questa la strategia dell'ingordo (*greedy*):

"Meglio un uovo oggi che una gallina domani".

Esempio (Resto col numero minimo di monete) Si supponga di avere monete da 50, 10, 5 e 1 centesimo, e di dover dare un resto di 72 centesimi. Senza neanche rendercene conto, utilizziamo la filosofia dell'ingordo, scegliendo una moneta da 50 centesimi, poi due da 10, e infine due da 1. La soluzione prodotta è formata da cinque monete ed è ottima, nel senso che non ce n'è una con un numero minore di monete. L'ottimalità della soluzione, però, non dipende dalla furbizia dell'ingordo, ma da una proprietà delle monete! Si supponga di avere monete da 11, 5 e 1 centesimo, e di dover dare un resto di 15 centesimi. L'ingordo giudicherà la moneta da 11 più "appetibile" delle altre, e darà un resto con una moneta da 11 e quattro da 1, utilizzando cinque monete, mentre la soluzione ottima è data da tre monete da 5 centesimi! ■

Esempio (Algoritmi di Dijkstra e Johnson) Ad ogni passo, gli algoritmi di Dijkstra e Johnson effettuano la scelta ingorda di selezionare il nodo che può essere raggiunto con costo minore fra quelli non ancora visitati. ■

Il metodo *greedy* si può applicare a quei problemi di ottimizzazione in cui occorre selezionare un sottoinsieme S "ottimo" di oggetti, che verificano certe proprietà, da un insieme A di n oggetti. La procedura greedy() mostra uno "scheletro" di codice che adotta tale metodo.

```
SET greedy(SET A)
    SET S = ∅
    { ordina A per "appetibilità" decrescente }
    foreach a ∈ A do                    % Secondo l'ordinamento appena calcolato
        if a può essere aggiunto ad S then
            S = S ∪ {a}
    return S
```

Un algoritmo *greedy* ordina dapprima gli oggetti in base ad un criterio di "appetibilità". La soluzione del problema è poi costruita in modo incrementale considerando gli oggetti uno alla volta e aggiungendo ogni volta l'oggetto più appetibile, se possibile. In altri termini, l'algoritmo effettua una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato. Fatta la scelta, è successivamente risolto un sottoproblema dello stesso tipo ma di dimensione più piccola. Il problema più piccolo dipende dalla sequenza di scelte passate, ma non dalle scelte future.

Ovviamente, non sempre esiste un algoritmo *greedy* che trovi la soluzione ottima di un qualsiasi problema di ottimizzazione, come abbiamo già potuto osservare per il problema del resto col minimo numero di monete. Affinché un algoritmo *greedy* fornisca la soluzione ottima di un problema, occorre che siano verificate due proprietà, tra loro correlate:

- *Sottostruttura ottima*. Per mostrare che una scelta *greedy* riduce il problema ad un sottoproblema più piccolo dello stesso tipo, occorre dimostrare che una soluzione ottima del problema contiene al suo interno le soluzioni ottime dei sottoproblemi;
- *Scelta greedy*. Dopo aver fornito una caratterizzazione matematica della soluzione ottima, occorre dimostrare che tale soluzione può essere modificata in modo da utilizzare una prima scelta *greedy*, che riduce il problema ad un sottoproblema più piccolo dello stesso tipo, e poi mostrare per induzione che il procedimento può essere esteso ad una sequenza di scelte *greedy*.

Iniziamo studiando due varianti di problemi che abbiamo già risolto con la programmazione dinamica, per poi vedere un esempio notevole di problema per il quale la tecnica *greedy* si comporta bene: trovare il minimo albero di copertura di un grafo pesato.

1.1 Zaino reale

ZAINO REALE. *Dati un insieme di n oggetti, con profitti $p_1, \ldots, p_n$ e volumi $v_1, \ldots, v_n$, tutti interi positivi, e un intero positivo C, la capacità dello zaino, il problema consiste nello scegliere oggetti o porzioni di essi in modo da massimizzare il profitto degli oggetti scelti senza superare la capacità; ovvero determinare un vettore $x = [x_1, \ldots, x_n]$ di reali compresi tra 0 e 1 tali che il profitto totale $p(x) = \sum_{1 \le i \le n} p_i x_i$ sia massimo e il volume $v(x) = \sum_{1 \le i \le n} v_i x_i \le C$.*

Abbiamo visto che il problema originale (che viene anche chiamato ZAINO 0-1 o ZAINO INTERO, ovvero "prendere/1 o lasciare/0") gode di sottostruttura ottima. È facile adattare la dimostrazione nel caso di scelta frazionaria; una volta scelta una certa quantità di un oggetto, possiamo ridurci al sottoproblema in cui c'è un oggetto in meno e una capacità inferiore. Una domanda legittima è chiedersi se questo problema è anche dotato di scelta *greedy*. Facciamo qualche tentativo.

> **Esempio (Profitto decrescente)** Supponiamo di scegliere oggetti in ordine di profitto decrescente. È facile trovare un controesempio per cui questo approccio non funziona: $p_1 = 3, p_2 =$

$2, p_3 = 2, v_1 = 2, v_2 = 1, v_3 = 1, C = 2$. La scelta *greedy* richiede di prendere il primo oggetto, con profitto 3 e volume 2, riempiendo lo zaino. Ma scegliere l'insieme dato dal secondo e terzo oggetto ha profitto ottimo 4 e volume 2, migliore della scelta *greedy*. ■

Esempio (Volume crescente) Neanche scegliere oggetti per volume crescente funziona. Il controesempio è dato da $p_1 = 1, p_2 = 1, p_3 = 3$, $v_1 = 1, v_2 = 1, v_3 = 2$ e capacità 2. La scelta *greedy* seleziona il primo e secondo oggetto, con profitto 2 e volume 2. Ma scegliere il terzo oggetto ha profitto ottimo 3 e volume 2, migliore della scelta *greedy*. ■

La scelta corretta è scegliere gli oggetti per "profitto specifico" decrescente, ovvero per rapporto profitto/volume. Si cerca poi di scegliere la porzione più grande possibile dell'oggetto che dà il maggior profitto per unità di volume, senza superare la capacità C; dopo di che, si risolve il sottoproblema dato dai rimanenti oggetti e dalla capacità residua.

Dimostriamo che la scelta *greedy* è sempre corretta. Supponiamo che esista una soluzione ottima x, che contenga una quantità $x[1] < 1$ tale che $x[1]v[1] < C$; ovvero, non contiene la porzione più grande possibile dell'oggetto 1 (quello con profitto specifico più alto dopo l'ordinamento). Costruiamo una nuova soluzione x' dove $x'[1] = \min\{v[1]/C, 1\}$, ovvero massimizziamo la porzione dell'oggetto 1 inserita nella soluzione, risolvendo poi il sottoproblema dato dai restanti $n-1$ oggetti con capacità $C - x'[1]v[1]$. La soluzione x' ha profitto totale maggiore o uguale al profitto di x, in quanto l'oggetto 1 ha il più alto profitto specifico.

Alla fine, saranno scelti per intero gli oggetti più appetibili, può essere scelta una frazione dell'oggetto immediatamente successivo, ed i rimanenti oggetti non saranno scelti. Il tempo di calcolo è $O(n \log n)$, dominato dal costo dell'ordinamento.

```
zaino(int[] p, int[] v, int C, int n, real[] x)
{ ordina p e v in modo che p[1]/v[1] ≥ p[2]/v[2] ≥ ··· ≥ p[n]/v[n] }
int i = 1
while i ≤ n and C > 0 do
    if v[i] ≥ C then
        x[i] = C/v[i]
        C = 0;
    else
        x[i] = 1
        C = C − v[i]
    i = i + 1
for j = i to n do x[j] = 0
```

1.2 Insieme indipendente di intervalli

Nel Capitolo 13, abbiamo descritto un algoritmo basato sulla programmazione dinamica per trovare l'insieme indipendente massimale per intervalli pesati. Consideriamo ora un caso particolare, in cui tutti gli intervalli hanno peso 1.

INSIEME INDIPENDENTE DI INTERVALLI. *Dati n intervalli distinti $I_1 = [a_1, b_1[, \ldots, I_n = [a_n, b_n[$ della retta reale, trovare un insieme indipendente massimo (ovvero un sottoinsieme di massima cardinalità formato da intervalli tutti disgiunti tra loro).*

La soluzione parte da premesse simili a quelle relative agli intervalli pesati. La dimostrazione di sottostruttura ottima resta valida. Ordiniamo gli intervalli per estremi finali non decrescenti, ovvero

in modo che $b_1 \leq b_2 \leq \cdots \leq b_n$. Inseriamo il primo intervallo I_1 e ci riduciamo al sottoproblema dato da tutti gli intervalli che non intersecano I_1.

Per provare la correttezza di questo algoritmo, dobbiamo dimostrare che la scelta *greedy* è sempre corretta; ovvero che esiste una soluzione ottima che include il primo intervallo (dopo l'ordinamento). Sia S una soluzione ottima; se $I_1 \in S$, la dimostrazione è banalmente completata. Se $I_1 \notin S$, sia I_s l'intervallo con minor estremo finale in S. Si consideri la soluzione $S - \{I_s\} \cup \{I_1\}$; poiché $b_1 < b_s$, sostituire I_s con I_1 non modifica l'indipendenza dell'insieme (l'intervallo I_s era indipendente da tutti gli altri intervalli in S, e l'intervallo I_1 che termina prima resta indipendente); inoltre, la cardinalità resta la stessa. Abbiamo quindi dimostrato che $S - \{I_s\} \cup \{I_1\}$ è un insieme indipendente massimo che contiene I_1.

L'algoritmo independentSet() scandisce gli intervalli da sinistra verso destra e ad ogni passo include nella soluzione l'intervallo che finisce prima e che non interseca gli intervalli già inclusi. Il sottoinsieme di intervalli selezionati è un insieme indipendente massimo. La complessità è $O(n \log n)$, dovuta all'ordinamento.

```
SET independentSet(int[] a, int[] b)
{ ordina a e b in modo che b[1] ≤ b[2] ≤ ··· ≤ b[n] }
SET S = Set()
S.insert(1)
int last = 1                                  % Ultimo intervallo inserito
for i = 2 to n do
    if a[i] ≥ b[last] then                    % Controllo indipendenza
        S.insert(i)
        last = i
return S
```

1.3 Minimo albero di copertura

Si consideri il seguente problema di ottimizzazione su grafi pesati.

MINIMO ALBERO DI COPERTURA (MINIMUM SPANNING TREE). *Dato un grafo non orientato e connesso $G = (V, E)$, con pesi $w(u, v)$ non negativi sugli archi, trovare un albero di copertura $G' = (V, T)$ per G, cioè un albero avente tutti i nodi di V, ma solo $n - 1$ degli m archi in E, tale che la somma $\sum_{(u,v)\in T} w(u, v)$ sia la più piccola possibile.*

Esempio (Minimo albero di copertura) La Figura 1.1 illustra un grafo non orientato pesato G e due alberi di copertura per G, dei quali il secondo è il minimo albero di copertura. ■

Questo problema può essere risolto con molti algoritmi, dei quali i più noti sono quelli di Kruskal (1956) e Prim (1957). Entrambi questi algoritmi sono basati sul principio *greedy*; illustriamo prima gli algoritmi, e poi discutiamo i principi alla base della loro correttezza.

1.3.1 Algoritmo di Kruskal

L'algoritmo di Kruskal analizza gli archi in ordine crescente di peso; un generico arco è aggiunto all'albero T se non forma circuiti con gli archi inseriti in T.

Esempio (Algoritmo di Kruskal) La Figura 1.2 mostra l'esecuzione dell'algoritmo di Kruskal sul grafo di Figura 1.1. L'ordinamento degli archi è: [2,4], [3,4], [5,7], [1,3], [1,2], [2,3], [4,5],

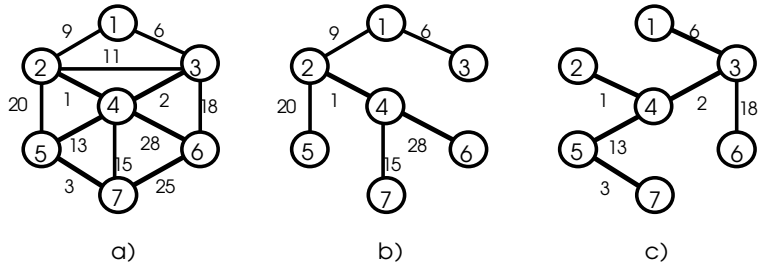

Figura 1.1: Minimo albero di copertura. a) Un grafo pesato; b) una soluzione ammissibile; c) una soluzione ottima.

[4,7], [3,6], [2,5], [6,7], [4,6]. ■

Il maggior sforzo computazionale dell'algoritmo di Kruskal è dato dall'ordinamento iniziale e dalla verifica che l'aggiunta di un nuovo arco non provochi la formazione di un circuito. L'ordinamento iniziale degli archi costa $O(m \log n)$ tempo, dato che $\log m \leq 2 \log n$. Inoltre, poiché la costruzione di T avviene per unione di insiemi disgiunti, cioè di due componenti connesse che si "fondono" in una sola con l'aggiunta del nuovo arco, è possibile utilizzare una struttura merge-find MFSET [cfr. Capitolo 10] per mantenere le componenti connesse di T, inizializzato con n componenti distinte, ciascuna formata da un singolo nodo del grafo. In questo modo, la verifica che un arco appartenga a due componenti distinte (cioè non provochi un circuito in T) ed anche il suo eventuale inserimento in T richiedono tempo ammortizzato costante $O(1)$ usando le operazioni find() e merge(). Poiché il ciclo **for** è ripetuto m volte, la complessità dell'algoritmo risulta essere $O(m \log n)$.

Per semplificare l'algoritmo di Kruskal, assumiamo che prenda in input un vettore di record ARCO, contenente i campi u, v e *peso*, corrispondenti agli estremi dell'arco e al suo peso.

Data l'usuale rappresentazione di un grafo con vettori di adiacenza, è facile vedere che il vettore degli archi può essere costruito in $O(n+m)$ tempo. L'albero T è restituito come un insieme di archi, che può essere realizzato con una lista non ordinata, in modo che l'inserimento di un arco in T richieda tempo $O(1)$.

Per risparmiare iterazioni inutili dopo che è stata raggiunta la soluzione ottima, la procedura kruskal() introduce una variabile intera c che conta il numero di inserimenti di archi nell'albero di copertura T, uscendo dal ciclo appena sono stati effettuati $n-1$ inserimenti. L'ordine di grandezza della complessità resta comunque $O(m \log n)$.

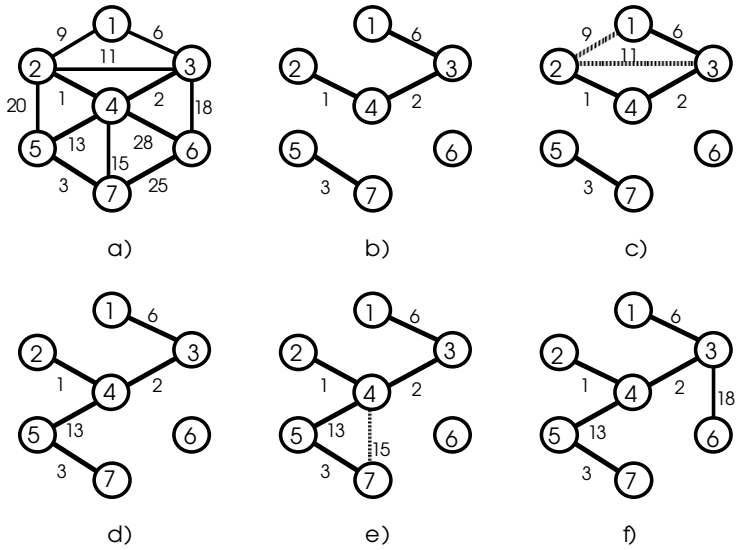

Figura 1.2: Algoritmo di Kruskal; a) Il grafo di ingresso. b) la foresta di copertura dopo l'inclusione di $[2,4]$, $[3,4]$, $[5,7]$ e $[1,3]$; c) $[1,2]$ e $[2,3]$ non sono inclusi; d) inclusione di $[4,5]$; e) $[4,7]$ non è incluso; f) inclusione di $[3,6]$ e soluzione ottima.

```
SET kruskal(ARCO[] A, int n, int m)
    SET T = Set()
    MFSET M = Mfset(n)
    { ordina A[1,...,m] in modo che A[1].peso ≤ ··· ≤ A[m].peso }
    int c = 0
    int i = 1
    while c < n − 1 and i ≤ m do                % Termina quando l'albero è costruito
        if M.find(A[i].u) ≠ M.find(A[i].v) then
            M.merge(A[i].u, A[i].v)
            T.insert(A[i])
            c = c + 1
        i = i + 1
    return T
```

1.3.2 Algoritmo di Prim

L'algoritmo di Prim parte da un albero T contenente uno specifico nodo radice r, e accresce T aggiungendo sempre l'arco con peso minore fra quelli che "escono" da T, ovvero fra quelli che uniscono un nodo di T con un nodo non incluso in T.

L'algoritmo prende in input un grafo G e un nodo radice r; come nel caso dei cammini minimi [cfr. Capitolo 11], la procedura ha accesso ad una funzione di costo w, lasciando alla realizzazione effettiva l'onere di definire dove debbano essere memorizzati tali costi (per es., nella matrice di adiacenza del grafo oppure nelle celle che formano la lista di adiacenza di un nodo). L'albero T viene rappresentato da un vettore di padri p.

La struttura di dati utilizzata in questo caso è una coda con priorità (min-heap). Tutti i nodi

vengono inseriti nella coda con priorità $+\infty$, tranne il nodo r che ha priorità 0. La priorità del nodo u corrisponde al peso minimo di un arco che raggiunge u a partire da un nodo compreso in T, mentre $pos[u]$ è la posizione che u occupa nello heap. Quando un nodo u viene estratto dalla coda, $[p[u],u]$ rappresenta l'arco con peso minore fra quelli che escono dall'albero; sia l'arco $[p[u],u]$ che il nodo u entrano a far parte dell'albero T, indicato con $pos[u] = \textbf{nil}$. Tutti gli archi incidenti ad u vengono analizzati, e se permettono di raggiungere un nodo v non ancora nell'albero ($pos[v] \neq \textbf{nil}$) con costo inferiore, la priorità nella coda viene aggiornata, così come il vettore dei padri.

Utilizzando una coda con priorità realizzata tramite heap binario, il ciclo di inserimento dei nodi nella coda con priorità viene eseguito n volte, ognuna con costo $O(\log n)$; il ciclo principale viene eseguito n volte, ognuna con costo $O(\log n)$ per l'estrazione; il ciclo esterno viene eseguito m volte, ognuna con costo $O(\log n)$. La complessità è quindi $O(m \log n)$, asintoticamente uguale all'algoritmo di Kruskal.

```
prim(GRAPH G, NODE r, int[] p)
    PRIORITYQUEUE Q = MinPriorityQueue()
    PRIORITYITEM[] pos = new PRIORITYITEM[1...G.n]
    foreach u ∈ G.V() − {r} do
        pos[u] = Q.insert(u, +∞)
    pos[r] = Q.insert(r, 0)
    p[r] = 0
    while not Q.isEmpty() do
        NODE u = Q.deleteMin()
        pos[u] = nil
        foreach v ∈ G.adj(u) do
            if pos[v] ≠ nil and w(u,v) < pos[v].priority then
                Q.decrease(pos[v], w(u,v))
                p[v] = u
```

Esempio (Algoritmo di Prim) La Figura 1.3 mostra l'esecuzione dell'algoritmo di Prim sul grafo di Figura 1.1 a partire dal nodo 1. ■

È possibile dare una versione dell'algoritmo di Prim che richiede tempo $O(n^2)$, sostituendo la struttura min-heap con una lista, analogamente a quanto visto per l'algoritmo di Dijkstra per i cammini minimi. Questa versione risulta vantaggiosa per grafi densi (dove $m = \Theta(n^2)$) mentre quella con min-heap è migliore per grafi sparsi (dove $m = O(n)$).

1.3.3 Dimostrazione di correttezza

Per dimostrare la correttezza degli algoritmi di Kruskal e Prim abbiamo bisogno di alcune definizioni. Un *taglio* $(S, V-S)$ è una partizione di V in due sottoinsiemi non vuoti. Un arco $[u,v]$ *attraversa* il taglio se $u \in S$ e $v \in V-S$. Vale il seguente teorema:

Teorema 1.1 Sia $A \subseteq E$ un sottoinsieme di un minimo albero di copertura di G. Sia $(S, V-S)$ un taglio non attraversato da alcun arco di A. Sia $[u,v]$ l'arco di E con peso minimo fra quelli che attraversano il taglio. Allora l'insieme $A \cup \{[u,v]\}$ è un sottoinsieme di un minimo albero di copertura di G.

Dimostrazione. Sia T un minimo albero di copertura che contiene A. Se l'arco $[u,v]$ appartiene a T, la proprietà è banalmente dimostrata: $A \cup \{[u,v]\} \subseteq T$. Altrimenti, si noti che per definizione

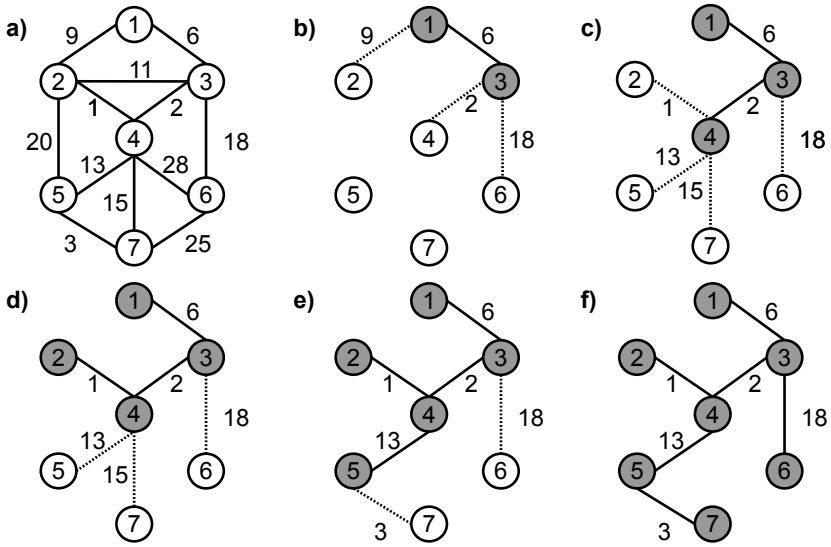

Figura 1.3: Algoritmo di Prim. I nodi in grigio sono quelli estratti dalla coda. Gli archi tratteggiati sono quelli che "escono" da T; gli altri archi rappresentano T. a) Il grafo di ingresso; b) l'albero dopo aver estratto 1 e 3; c),d),e) l'albero dopo aver estratto 4, 2, 5, rispettivamente; f) soluzione ottima.

di albero u e v sono connessi da un cammino c in T. Poiché per ipotesi $u \in S$ e $v \in V - S$, esiste almeno un arco $[x,y]$ in c che attraversa il taglio. Si noti che $[x,y] \notin A$, visto che nessun arco di A attraversa il taglio.

Se sostituiamo $[u,v]$ al posto di $[x,y]$, otteniamo un insieme di archi $T' = T - \{[x,y]\} \cup \{[u,v]\}$ che è ancora un albero (perché una volta sconnesso $[x,y]$, l'albero si spezza in due componenti che si riuniscono con $[u,v]$); inoltre, il peso di T' è inferiore o uguale al peso di T (in quanto per definizione $w(u,v) \leq w(x,y)$). Questo dimostra la proprietà, in quanto T' è un minimo albero di copertura, e sappiamo che $A \cup \{[u,v]\} \subseteq T'$. ■

Per concludere la dimostrazione di correttezza di Kruskal e Prim, dobbiamo provare che quando un arco $[u,v]$ viene aggiunto all'insieme A di archi, l'ipotesi della proprietà sia rispettata, ovvero esista un taglio $(S, V-S)$ tale che nessun arco di A attraversi il taglio e $[u,v]$ sia l'arco di peso minimo fra quelli che attraversano il taglio.

Nel caso di Kruskal, l'arco $[u,v]$ che viene aggiunto connette due componenti connesse di $G' = (V,A)$; ponendo S uguale ad una delle due componenti, per definizione nessun arco di A attraversa il taglio (altrimenti sarebbero un'unica componente); essendo l'arco $[u,v]$ il primo nell'ordinamento ad attraversare il taglio ed unire tali componenti, esso ha il peso minore fra quelli che attraversano il taglio.

Nel caso di Prim, l'arco $[u,v]$ connette la componente connessa S formata dai nodi già visitati dell'albero e la componente $V - S$ che include tutti gli altri. Grazie all'utilizzo della coda con priorità, l'arco ha il peso minimo fra tutti quelli che escono dall'albero (ovvero attraversano il taglio).

1.4 Reality check

Un *algoritmo di compressione* prende in input una sequenza di bit e produce in output una sequenza di bit più corta, detta *sequenza compressa*. Un algoritmo di decompressione prende in input una

sequenza compressa e cerca di ricostruire la sequenza originale. Il problema della compressione è di fondamentale importanza nell'informatica, sia per quanto riguarda lo spazio di memorizzazione, ma anche e soprattutto per risparmiare tempo di trasmissione.

Esistono innumerevoli tecniche per la codifica; una tecnica molto potente è la compressione di Huffman (1952) per la codifica di testi, utilizzata ancora oggi nei programmi quali PkZip. È una tecnica di codifica a caratteri: ogni carattere c viene codificato da una sequenza di bit b. Nella codifica ASCII, tutti i caratteri sono rappresentati con 8 bit. Ma in un qualunque testo scritto, i caratteri non appaiano tutti con la stessa frequenza. Ad esempio, in italiano la lettera `j` è molto meno frequente della `a`. In una *codifica con lunghezza variabile*, i caratteri più frequenti vengono rappresentati con una codifica più breve rispetto ai caratteri meno frequenti.

Esempio (Codifica di Huffman) Si consideri un testo composto da n caratteri dall'alfabeto {`a`, `b`, `c`, `d`, `e`, `f`}, che appaiono con le frequenze 45%, 13%, 12%, 16%, 9%, 5%, nell'ordine. Se questo testo viene codificato in ASCII, richiederà $8n$ bit. Se viene codificato con 3 bit per carattere (sufficienti a rappresentare i 6 possibili caratteri), richiederà $3n$ bit. Se viene codificato nel modo seguente: `a=0`, `b=101`, `c=100`, `d=111`, `e=1101`, `f=1100`, date le frequenze di ognuno dei caratteri si ottiene una lunghezza pari a:

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4) \cdot n = 2.24n$$

Potremmo chiederci perché non si è utilizzata una codifica ancora più breve, quale: `a=0`, `b=1`, `c=00`, `d=01`, `e=10`, `f=11`. Il motivo è semplice: perché sarebbe impossibile decomprimere il testo "`000`", che potrebbe corrispondere ad "`aaa`", "`ca`" o "`ac`". La codifica proposta nell'esempio non ha questo problema, essendo una *codifica a prefissi*: nessun codice è prefisso di un altro codice. Questa è una condizione necessaria per permettere la decodifica.

Esempio (Prefissi telefonici) Il prefisso di Milano è 02; il prefisso di Bologna è 051; il prefisso di Trento è 0461. Non a caso le province hanno 3.2, 0.98, 0.52 milioni di abitanti: a provincia più grande (abitanti più "frequenti") corrisponde un prefisso più breve. Inoltre, non esiste il prefisso $02X$, con X una qualunque cifra: sarebbe indistinguibile da Milano. ■

L'obiettivo è costruire la minima codifica a caratteri con lunghezza variabile: data una sequenza s, una codifica c è minima per s se non esiste un'altra codifica con cui s possa essere compressa impiegando un numero inferiore di bit. L'algoritmo di Huffman risolve questo problema di ottimizzazione tramite la tecnica *greedy*.

Si inseriscono i caratteri in un min-heap, con priorità data dalla frequenza. Vengono estratti i due caratteri c_1 e c_2 con frequenze più basse f_1 e f_2; si crea un nuovo carattere "virtuale" c che ha come frequenza la somma delle frequenze $f_1 + f_2$, che viene inserito nel min-heap. I caratteri c, c_1 e c_2 vengono organizzati ad albero binario, con c_1 e c_2 figli destro e sinistro di c. L'operazione di doppia estrazione ed inserimento viene ripetuta per $n-1$ volte, fino a quando non si rimane con la radice dell'albero. A questo punto, si assegna il codice 0 agli archi che collegano un nodo al suo figlio sinistro, e 1 agli archi che collegano un nodo al suo figlio destro. Tutti i caratteri originali (non virtuali) sono foglie; la loro codifica è data dai codici sugli archi che si trovano nel percorso radice-foglia.

La dimostrazione di correttezza non è complessa, ma è lunga e tediosa. Basti notare che i caratteri meno frequenti hanno codifica più lunga, in quanto si trovano a maggior profondità nell'albero.

1.5 Esercizi

Esercizio 1.1 (Zaino intero)**.** Si dimostri con un controesempio che, al contrario dello ZAINO reale, il problema dello ZAINO INTERO non si può risolvere con la tecnica *greedy* suggerita in questo capitolo.

Esercizio 1.2 (Algoritmo di Kruskal)**.** Si assuma che i pesi degli archi di un grafo non orientato G siano interi e compresi nell'intervallo $1,\ldots,n$, dove n è il numero di nodi di G. È possibile abbassare la complessità dell'algoritmo di Kruskal?

Esercizio 1.3 (Grafi blu)**.** Un grafo G ha archi colorati di rosso e blu. Scrivere un algoritmo che restituisca un albero di copertura per G con il numero minimo di archi rossi.

Esercizio 1.4 (Algoritmo di Prim)**.** Qual è la complessità dell'algoritmo di Prim se invece di utilizzare una realizzazione della coda con priorità basata su heap binario, si utilizza un heap di Fibonacci [cfr. Capitolo 11.5]?

Esercizio 1.5 (Shortest Job First)**.** Dati n programmi tali che l'i-esimo programma richiede t_i unità di tempo di esecuzione, trovare una sequenza di esecuzione (una permutazione) $a_1,\ldots,a_n$ di $\{1,\ldots,n\}$ che minimizzi il tempo medio di completamento. Il *tempo di completamento del programma* i è dato dal tempo che occorre dall'inizio dell'esecuzione del primo programma s_1 al completamento del programma i-esimo.

Esercizio 1.6 (Minima colorazione di intervalli)**.** Siano dati n intervalli distinti $I_1 = [a_1, b_1[, \ldots, I_n = [a_n, b_n[$ della retta reale. Si progetti un algoritmo *greedy* di complessità $O(n \log n)$ per colorare ciascun intervallo in modo che intervalli che si intersecano siano colorati con colori diversi e che il numero di colori usati sia minimo.

Esercizio 1.7 (Torri cellulari lineari)**.** Si consideri una strada diritta di lunghezza k punteggiata da n abitazioni collocate ai chilometri $k_1, k_s, \ldots, k_n$, con $k_i \in [0,k]$ (come esempio, considerate una di quelle strade americane che sembrano infinite). Una compagnia telefonica vuole collocare un insieme di torri cellulari, in modo tale che ogni abitazione sia ad una distanza massima d da una torre. Progettare un algoritmo per risolvere questo problema utilizzando il numero minimo di torri e discuterne la complessità.

Esercizio 1.8 (Sciatori e sci)**.** Siano dati n sciatori di altezza $p_1,\ldots,p_n$, e n paia di sci di lunghezza $s_1,\ldots,s_n$. Il problema è assegnare ad ogni sciatore un paio di sci, in modo da minimizzare la differenza totale fra le altezza degli sciatori e la lunghezza degli sci; ovvero, se allo sciatore i-esimo è assegnato il paio di sci $h(i)$, minimizzare la seguente quantità:

$$\sum_{i=1}^{n} |p_i - s_{h(i)}|$$

Si consideri il seguente algoritmo greedy. Si individui la coppia (sciatore, sci) con la minima differenza. Si assegni allo sciatore questo paio di sci. Si ripete con gli sciatori restanti fino a quando non si è terminato.

Provare la correttezza di questo algoritmo o trovare un controesempio.

Esercizio 1.9 (Archi minimi)**.** Sia dato un grafo non orientato $G = (V,E)$ e una funzione di pesi $w : E \to \mathbb{R}$, con pesi sugli archi tutti distinti. Siano e_1, e_2 ed e_3 l'arco con peso minimo, l'arco con il secondo peso minimo e l'arco con il terzo peso minimo, rispettivamente. Confutare o provare le seguenti affermazioni:

- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_1.

- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_2.
- Tutti gli alberi di copertura di peso minimo del grafo G contengono l'arco e_3.

Esercizio 1.10 (Massimizzazione e minimizzazione)**.** Si dimostri se i problemi MASSIMO ALBERO DI COPERTURA e CAMMINI MASSIMI sono equivalenti o meno ai problemi MINIMO ALBERO DI COPERTURA e CAMMINI MINIMI.

1.6 Soluzioni

Soluzione Esercizio 1.1
Si consideri uno zaino di capacità 6 e tre oggetti di volume $3,3,4$ e profitto $3,3,5$, rispettivamente; l'algoritmo basato sul valore specifico sceglierebbe il terzo oggetto con profitto totale 5, in quanto il profitto specifico $5/4$ è più alto del profitto 1 degli altri due. Ma data la capacità dello zaino, è possibile scegliere i primi due oggetti con profitto totale 6.

Soluzione Esercizio 1.2
Se tutti i pesi sono compresi nell'intervallo $1,\ldots,n$, è possibile utilizzare Counting Sort [cfr. Capitolo 2] per ordinare gli archi in tempo $O(m+n)$. L'ordinamento in tempo $O(m\log n)$ viene a mancare e l'algoritmo di Kruskal può essere eseguito in tempo $O(m+n)$.

Soluzione Esercizio 1.3
È necessario costruire una funzione di peso che associa il peso 1 agli archi rossi, il peso 0 agli archi blu. Si applica poi un algoritmo per ottenere il minimo albero di copertura, che conterrà il numero minimo di archi rossi.

Soluzione Esercizio 1.5
Il titolo dell'esercizio suggerisce una soluzione. Un algoritmo *greedy* che minimizza il tempo medio di completamento ordina i programmi per tempo di esecuzione crescente. Dimostriamo che esiste sempre una soluzione ottima che contiene la scelta ingorda.

Supponiamo di avere una soluzione ottima A, rappresentata da una permutazione $a_1, a_2, \ldots, a_n$ degli indici $1,2,\ldots,n$. Supponiamo che il programma di durata inferiore sia in posizione a_k, con $k > 1$ (se $k = 1$, abbiamo terminato). Costruiamo una soluzione A', scambiando gli elementi 1 e k della permutazione.

- Tutti i programmi eseguiti dopo il programma k hanno lo stesso tempo di completamento in A e A', perché lo scambio non influisce su di loro.
- Poiché la durata del k-esimo job è più breve di quella del primo, se ne deduce che, quando viene spostato al primo posto, ha un tempo di completamento inferiore; d'altra parte, il tempo di completamento del job in k-esima posizione in entrambe le sequenze è uguale, in quanto comunque devono essere completati tutti i primi k job.
- Tutti i programmi compresi fra 1 e k, estremi esclusi, hanno un tempo di completamento inferiore o uguale in A', perché lo scambio può avere ridotto il tempo di completamento per il primo programma.

Quindi il tempo di completamento totale di A' è inferiore o uguale al tempo di completamento di A; ma poiché A è ottima, questo significa che A e A' hanno lo stesso tempo di completamento e quindi anche A' è ottima.

Soluzione Esercizio 1.6
È immediato verificare che il numero minimo di colori non può essere inferiore al numero massimo c di sovrapposizioni tra intervalli, calcolato nell'Esercizio 9.8. Supponiamo senza perdere in generalità che tutti gli estremi degli intervalli siano distinti. Ordiniamo gli intervalli per estremi iniziali crescenti, ovvero in modo che $a_1 < a_2 < \cdots < a_n$. Per $j = 1,\ldots,n$, consideriamo l'intervallo

I_j. Per ogni intervallo I_i con $a_i < a_j$ e che interseca I_j, escludiamo il colore di I_i ed assegnamo ad I_j un colore non escluso. Vediamo che in questo modo sono colorati tutti gli intervalli e che intervalli che si intersecano hanno colori distinti. Se per assurdo esistesse un intervallo I_j non colorato, preceduto nell'ordinamento da m intervalli che lo intersecano, allora esisterebbe una massima sovrapposizione di $m+1 \leq c$ intervalli. Risulterebbe quindi $m \leq c-1$ e si giungerebbe alla contraddizione che esiste un colore libero. Quindi I_j è stato colorato. Inoltre, non esistono intervalli colorati con lo stesso colore di I_j perché sono stati esclusi per costruzione. Infine, la colorazione è minima, dato che si sono usati c colori e c è una limitazione inferiore al numero di colori. In pratica, l'algoritmo dapprima calcola c come mostrato nell'Esercizio 9.8 e inserisce i colori in una pila. Indi scandisce i $2n$ estremi $\{a_1,\ldots,a_n,b_1,\ldots,b_n\}$ (ordinati in modo crescente) da sinistra verso destra. Se l'elemento scandito è un estremo iniziale, cioè un a_i, allora si estrae un colore dalla pila e lo si assegna ad I_i. Se invece è un estremo finale, cioè un b_i, allora si libera il colore di I_i reinserendolo nella pila. La complessità dell'algoritmo è $O(n\log n)$ a causa dell'ordinamento.

Soluzione Esercizio 1.7

Un algoritmo *greedy* potrebbe essere il seguente. Innanzitutto, si assuma che i valori k_i siano ordinati, oppure si ordinino in $O(n\log n)$ tempo. Si costruisca una torre al chilometro k_1+d; questa serve l'abitazione 1 e potenzialmente tutte quelle comprese in $[k_1,k_1+2d]$, che possono essere eliminate. Si ripete il procedimento con le torri restanti. La soluzione è corretta, in quanto qualunque soluzione ottima può essere trasformata in una soluzione che comprende la torre in k_1+d.

```
torri(int[] k, int n, int d)
    SET S = Set()
    int t = k[1] + d                          % Ultima torre piazzata
    S.insert(t)
    for i = 2 to n do
        if k[i] > t + d then
            t = k[i] + d                      % Nuova torre, che copre la prima abitazione esclusa
            S.insert(t)
```