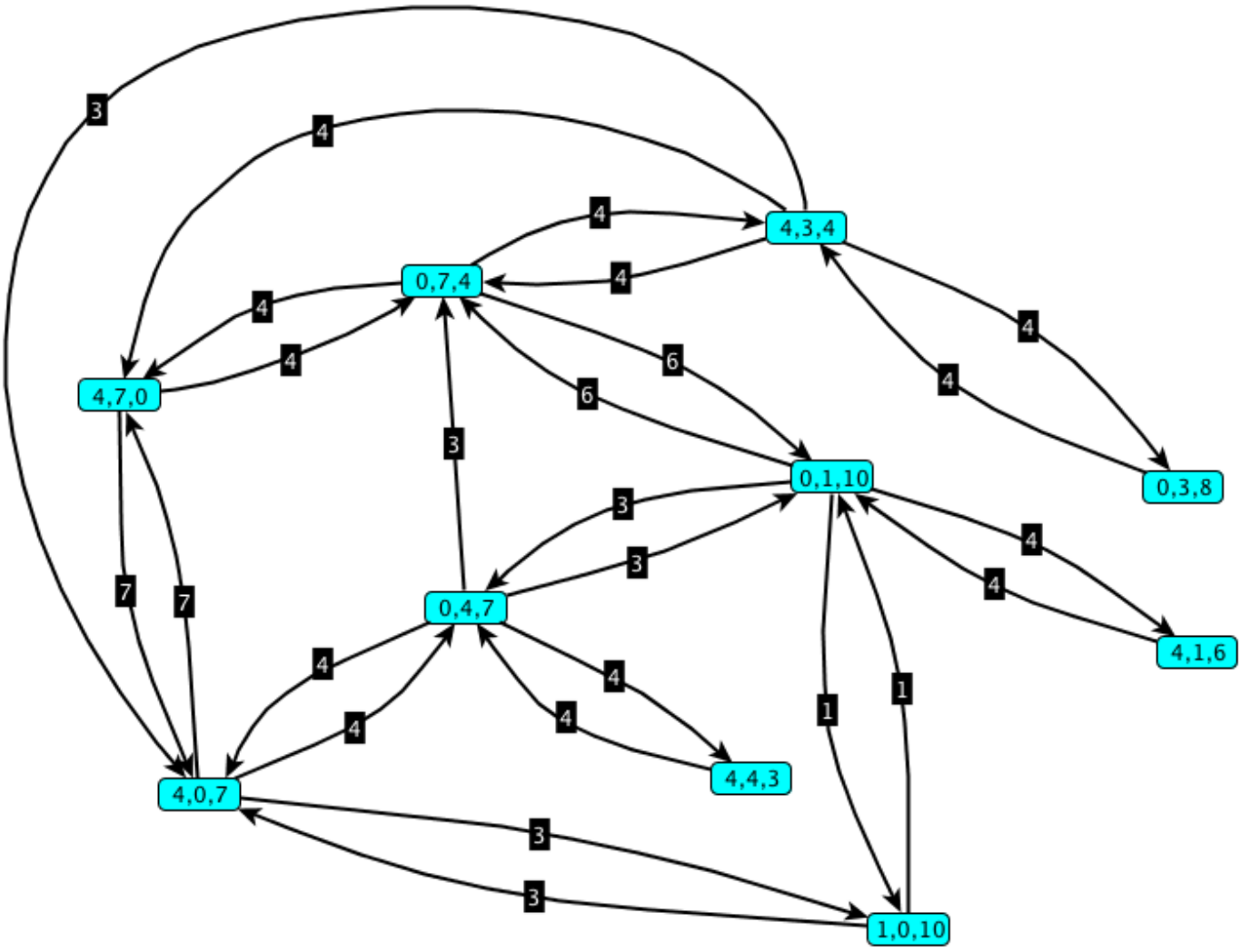
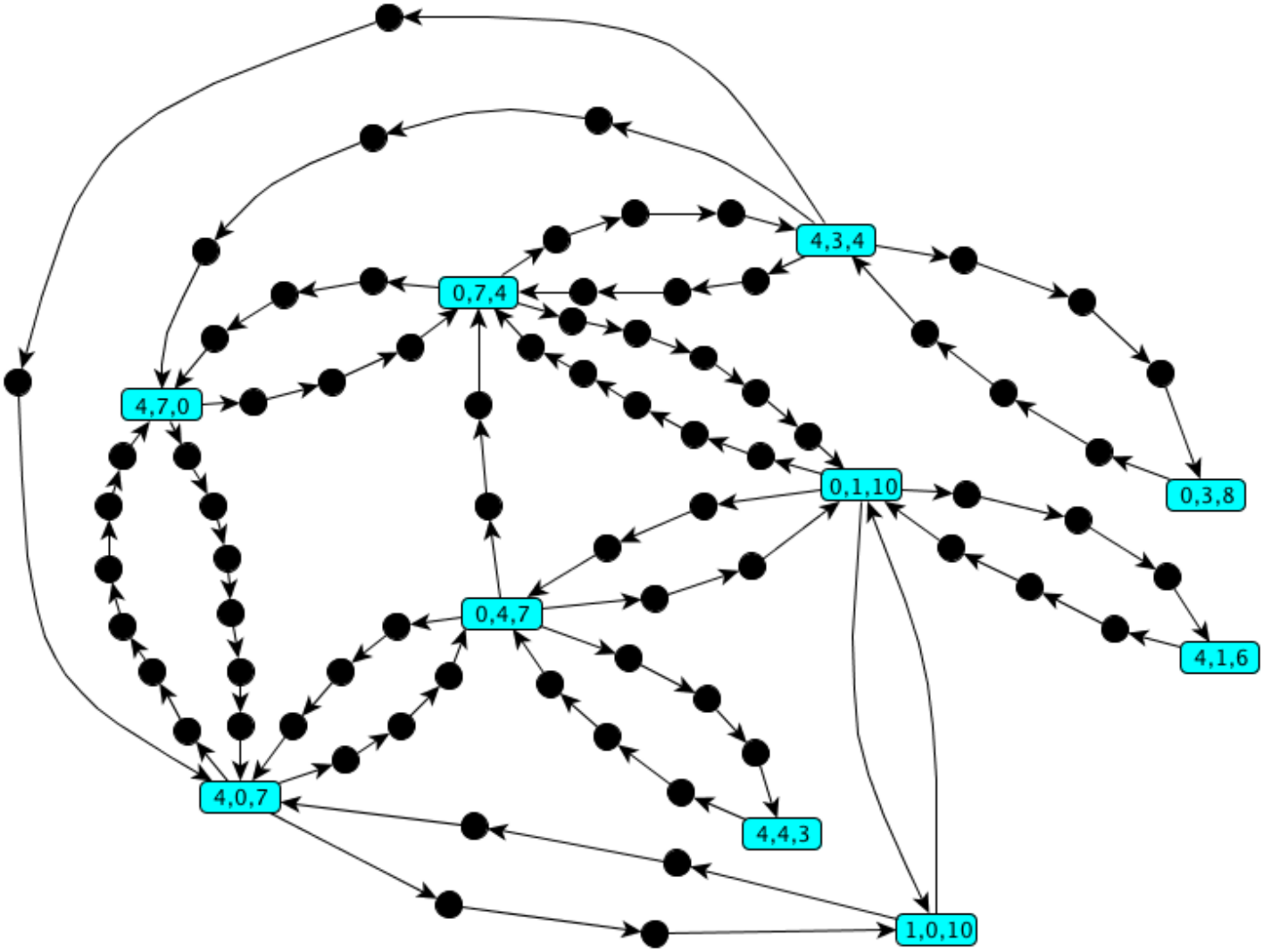
Progettazione di Algoritmi - lezione 9

Discussione dell'esercizio [acqua 2]

Vogliamo trovare una sequenza di versamenti che sia buona e parsimoniosa. Consideriamo quindi il grafo i cui nodi sono le possibili configurazioni (i livelli di riempimento dei tre contenitori) e vi è un arco da un nodo (*a*,*b*,*c*) al nodo (*a*',*b*',*c*') se c'è un versamento, o mossa, ammissibile che porta da (*a*,*b*,*c*) a (*a*',*b*',*c*'). Siccome dobbiamo misurare il numero di litri d'acqua versati nelle varie mosse, conviene etichettare ogni arco con il numero di litri che vengono versati nella corrispondente mossa. Così si ottiene un grafo come nella figura qui sotto (solo una piccola porzione dell'intero grafo è mostrata):



Il problema diventa allora quello di trovare un cammino da (4,7,0) a un nodo del tipo (2,*b*,*c*) o (*a*,2,*c*) che minimizza la somma dei valori sugli archi. Questo problema è una generalizzazione del problema dei cammini di lunghezza minima perché gli archi invece di valere tutti lo stesso valore, cioè 1, hanno valori differenti. Fra poco vedremo un algoritmo efficiente che risolve questo tipo di problemi qualunque siano i valori degli archi (anche numeri reali) purché non negativi. Ma in questo caso, in cui i valori sono interi relativamente piccoli, possiamo ricondurci a risolvere un semplice problema di cammini di lunghezza minima su un grafo opportuno. Infatti, possiamo sostituire ogni arco da *u* a *v* con valore *L* con un cammino con *L* archi da *u* a *v* aggiungendo *L* nuovi archi e *L* - 1 nuovi nodi. Ecco come diventa la prorzione di grafo precdente dopo le sostituzioni:



Nel nuovo grafo tutti gli archi valgono 1. È come se ogni arco corrisponda al versamento di un litro d'acqua. Così contando semplicemente gli archi, di un cammino tra due nodi configurazione, contiamo proprio il numero totale di litri versati nelle mosse relative al cammino. Quindi per risolvere il problema possiamo fare una BFS, nel nuovo grafo, a partire dal nodo (4,7,0) che si ferma non appena trova un nodo del tipo (2,*b*,*c*) o (*a*,2,*c*).

Grafi pesati

Come il problema precedente ha mostrato ci sono casi in cui un problema viene naturalmente rappresentato tramite un grafo i cui archi hanno un valore numerico. Ecco alcuni altri esempi di problemi di questo tipo.

- In relazione ad una rete stradale, si vuole determinare il percorso più breve tra due incroci. I nodi del grafo sono gli incroci e un arco da un incrocio ad un altro avrà un valore pari alla lunghezza della strada che collega i due incroci.
- La stessa rete stradale del problema precedente, ma ora si vuole trovare il percorso più veloce tra due incroci. In questo caso il valore di un arco è il tempo medio di percorrenza della corrispondente strada.

Di solito i valori degli archi vengono chiamati *pesi* e il corrispondente grafo è chiamato *grafo pesato*. Assumeremo che tutti i pesi sono non negativi. Più avanti vedremo come trattare anche grafi con pesi negativi. Per ogni arco non diretto $\{u, v\}$ o diretto (u, v), denotiamo con $p(u, v)$ il peso dell'arco. Dato un cammino C, denotiamo con $p(C)$ il peso del cammino, cioè la somma dei pesi degli archi di C. Se il grafo non è pesato si assume che i pesi degli archi siano tutti uguali a 1, così il peso di un cammino coincide con la lunghezza del cammino.

In un grafo pesato G possiamo definire la distanza tra due nodi u e v: $d_G(u, v)$ = *il peso minimo tra quello di tutti i cammini da u a v*. Se non ci sono cammini da u a v, si conviene che $d_G(u, v) = \infty$. Come per il caso dei grafi non pesati le distanze godono delle seguenti proprietà, per qualsiasi nodi u, v, w:

- $d_G(u, u) = 0$
- $d_G(u, v) \geq 0$
- $d_G(u, v) \leq d_G(u, w) + d_G(w, v)$

Inoltre, se il grafo G non è diretto, le distanze sono anche simmetriche: $d_G(u, v) = d_G(v, u)$.

Cammini minimi in grafi pesati (Dijkstra)

Vogliamo trovare i cammini minimi e quindi anche le distanze in un grafo pesato. L'idea è di generalizzare o estendere l'algoritmo che già conosciamo per grafi non pesati, cioè la BFS. Partendo da un nodo u, vogliamo determinare i cammini minimi e le distanze di tutti gli altri nodi da u. L'approccio della BFS è di trovare prima di tutto i nodi a distanza 1 da u, cioè i nodi a distanza minima da u. Possiamo fare lo stesso in un grafo pesato? Quale può essere un nodo a distanza minima da u? Sicuramente deve essere un adiacente di u. Perché se fosse un nodo w non adiacente possiamo prendere il nodo v che segue u nel cammino minimo che va da u a w e questo è ha distanza minore od uguale a quella di w (perché i pesi degli archi non sono negativi) ed inoltre è adiacente a u. Quindi tra tutti gli adiacenti di u troviamo quello a distanza minima. La BFS poi continua considerando il prossimo a distanza minima. In generale, ad un certo punto avremo determinato le distanze $d_G(u, v)$ per ogni v in un insieme di nodi R. Questi sono i nodi a distanza più piccola da u perché tutti gli altri saranno a distanza maggiore od uguale alla massima distanza di un nodo di R da u. Il prossimo nodo a distanza più piccola si troverà necessariamente tra gli adiacenti ai nodi in R. Quindi sarà un nodo w per cui c'è un arco (v, w) tale che v è in R, w non è in R e

$$d_G(u, v) + p(v, w) = \min(d_G(u, x) + p(x, y) \mid x \text{ è in } R, y \text{ non è in } R \text{ e } (x, y) \text{ è un arco })$$

Perciò $d_G(u, w) = d_G(u, v) + p(v, w)$ e w sarà aggiunto R. E si continua così finché tutti i nodi raggiungibili da u saranno stati aggiunti ad R e le loro distanze determinate. L'algoritmo appena descritto si deve a Edsger Wybe Dijkstra. Ecco una descrizione ad alto livello dell'algoritmo di Dijkstra:

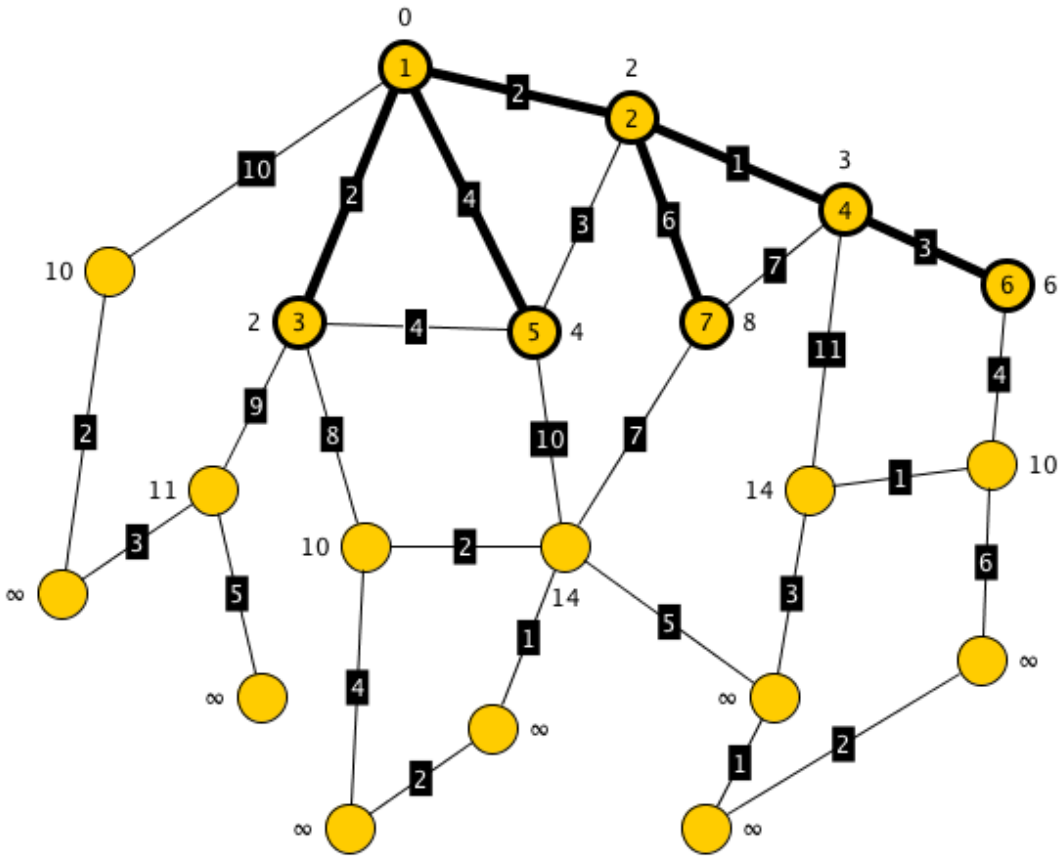
```
DIJKSTRA(G: grafo, u: nodo)
    Dist: array delle distanze, inizializzato a -1
    P: vettore dei padri, inizializzato a 0
    Dist[u] <- 0
    P[u] <- u
    R <- {u}      /* Insieme dei nodi la cui distanza è già stata determinata */
    WHILE esiste un arco da R a fuori di R DO
        Trova un arco (v, w) con v in R, w non in R per cui è minima Dist[v] + p(v, w)
        Dist[w] <- Dist[v] + p(v, w)
        P[w] <- v
        R <- R ∪ {w}
    RETURN Dist, P
```

Chiaramente il `WHILE` esegue un numero di iterazioni pari a k - 1, dove k è il numero di nodi raggiungibili da u. Ad ogni iterazione del `WHILE` è scelto un nodo, tra quelli la cui distanza da u non è stata ancora determinata, che si trova a distanza minima da u. In modo equivalente si può anche dire che ad ogni iterazione è scelto un arco, tra quelli che possono estendere l'albero dei cammini minimi, il cui estremo nuovo ha la distanza minima. Quindi è anche chiaro che, se l'algoritmo è corretto, i cammini minimi da un nodo u trovati dall'algoritmo, al pari di quelli trovati da una BFS per grafi non pesati, formano un albero radicato in u.

Più avanti vedremo una dimostrazione di correttezza dell'algoritmo. Adesso vogliamo considerarne l'efficienza. Per farlo dobbiamo specificare in che modo, ad ogni iterazione del `WHILE`, troviamo un arco (v, w) con v in R, w non in R e che rende minima `Dist[v] + p(v, w)`. Se lo facciamo in modo diretto, per ogni nodo v in R dobbiamo esaminare tutti gli adiacenti di v. Questo comporta che gli adiacenti di molti nodi saranno esaminati tantissime volte. La complessità è dell'ordine di $O(nm)$, quindi molto elevata. Possiamo fare di meglio? Sì perché possiamo mantenere per ogni nodo nella frontiera, cioè adiacente ad un nodo di R, la distanza che avrebbe se fosse scelto. In questo modo ad ogni iterazione basterà scegliere un nodo non in R che ha la distanza minima. Per fare ciò in modo efficiente si può usare una coda con priorità, ovvero un min-heap. Per mantenere le distanze "provvisorie" possiamo usare lo stesso array `Dist`. Per semplicità assumiamo che tutti i nodi del grafo siano raggiungibili da u.

```
DIJKSTRA(G: grafo, u: nodo)
    Dist: array delle distanze, inizializzato a infinito
    P: vettore dei padri, inizializzato a 0
    Dist[u] <- 0
    P[u] <- u
    H <- min-heap inizializzato con tutti i nodi e le priorità sono i valori di Dist
    WHILE H non è vuoto DO
        v <- H.get_min()       /* Preleva il nodo con distanza minima */
        FOR ogni adiacente w di v DO
            IF Dist[w] > Dist[v] + p(v, w) THEN
                Dist[w] <- Dist[v] + p(v, w)
                P[w] <- v
                H.decrease(w)  /* Aggiorna l'heap a seguito del decremento */
    RETURN Dist, P
```

La figura seguente mostra, su un grafo d'esempio, la situazione durante l'esecuzione dell'algoritmo quando sono state determinate le distanze dei primi 7 nodi (il nodo 1 è il nodo di partenza). I 7 nodi sono numerati secondo l'ordine con cui la loro distanza è stata determinata e accanto ad ognuno c'è proprio la distanza. Accanto ad ogni altro nodo è indicata la sua distanza provvisoria.



L'inizializzazione del min-heap prende tempo O(*n*). Il `WHILE` esegue esattamente *n* iterazioni. L'estrazione del nodo con distanza minima dall'heap prende tempo O(log*n*). Poi c'è la scansione degli adiacenti del nodo estratto. L'operazione più onerosa per ognuno degli adiacenti è l'aggiornamento dell'heap `H.decrease(w, Dist[w])` a seguito del decremento della distanza dell'adiacente. Sappiamo che tale operazione può essere implementata in modo che abbia complessità O(log*n*) (sarà utile a tal scopo avere un array che mantiene per ogni nodo la sua posizione all'interno dell'heap). Chiaramente nelle varie scansioni degli adiacenti un arco sarà esaminato al più due volte (nel caso di grafi non diretti, nel caso di grafi diretti una sola volta). E quando è esaminato, il costo sarà al più O(log*n*). Quindi il costo totale di tutte le iterazioni del `WHILE` è O(*n*log*n* + *m*log*n*). Perciò la complessità di questa implementazione dell'algoritmo di Dijkstra è O((*n* + *m*)log*n*).

C'è una implementazione più semplice che non usa un heap ma solamente l'array delle distanze `Dist`. Ad ogni iterazione del `WHILE` si trova un nodo con distanza minima tra quelli la cui distanza non è stata ancora determinata (quelli la cui distanza è già determinata sono marcati tramite, ad esempio, un apposito array). Questo ha costo O(*n*). Poi si esaminano gli adiacenti e per ognuno si controlla se viene decrementata la distanza provvisoria. Ovviamente tutti questi controlli relativi agli adiacenti

hanno costo complessivo O(m). Quindi la complessità di questa implementazione più semplice è dominata dal costo ad ogni iterazione di trovare il nodo con distanza minima che è O(n^2), dato che m è al più O(n^2). Questa implementazione sicuramente è migliore di quella che usa l'heap quando il grafo è molto denso, cioè quando ha ordine di n^2 archi.

Algoritmi greedy

Come abbiamo visto l'algoritmo di Dijkstra ad ogni iterazione estende l'albero dei cammini minimi con l'arco che, potremmo dire, è il più "conveniente", nel senso che è quello che aggiunge all'albero il nodo a distanza minima tra tutti quelli che si possono aggiungere con un solo arco. In termini più generici possiamo dire che l'algoritmo di Dijskstra costruisce una soluzione (cioè, l'albero dei cammini minimi da un nodo u) partendo da una *soluzione parziale iniziale* (l'albero costituito dal solo nodo u) che viene estesa ad ogni passo ad una *soluzione parziale* più grande scegliendo tra tutte le possibili *estensioni locali* (gli archi che possono estendere direttamente l'albero) quella *più conveniente* (quella che aggiunge un nuovo nodo con distanza minima). Questo è uno schema generale comune, come vedremo, a moltissimi algoritmi e che per molti problemi porta ad algoritmi ottimi. Lo schema generale può essere descritto come segue:

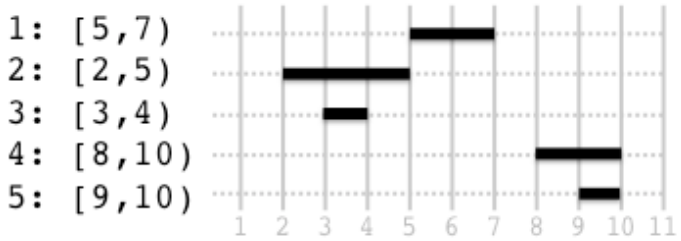
```
INPUT I         /* Istanza del problema */
    S <- S_0    /* Soluzione parziale iniziale per I */
    WHILE S può essere estesa DO
        Trova una estensione locale S' di S più conveniente
        S <- S'
OUTPUT S
```

Gli algoritmi che seguono questo schema sono detti **algoritmi greedy**. Questo nome deriva dal fatto che nell'estendere una soluzione parziale un tale algoritmo cerca solamente tra le estensioni più "vicine" alla soluzione parziale e sceglie l'estensione che è più conveniente rispetto solamente a queste relativamente poche estensioni locali. Quindi si comporta un po' come una persona golosa o avida (appunto *greedy*) che cerca di ottenere qualcosa di buono subito piuttosto che aspettare e ottenere magari qualcosa di meglio dopo.

Ovviamente, lo schema generale per gli algoritmi greedy non è una ricetta precisa per costruire tali algoritmi, piuttosto è semplicemente l'indicazione di un possibile approccio che si può adottare nel trovare un algoritmo. Inoltre, come vedremo con vari esempi, per un certo problema è possibile dare diversi algoritmi che possono dirsi greedy.

Selezione Attività

Per illustrare il progetto e l'analisi di algoritmi greedy consideriamo un problema piuttosto semplice chiamato *Selezione Attività*: date n attività (lezioni, seminari ecc.) e per ogni attività i, $1 \le i \le n$, l'intervallo temporale $[s_i, f_i)$ in cui l'attività dovrebbe svolgersi, selezionare il maggior numero di attività che possono essere svolte senza sovrapposizioni in un'aula. Ad esempio, se si hanno le seguenti 5 attività:



se ne possono selezionare 3 (ad esempio, le attività 1,2,4 o 1,3,5). Si inizi a pensare a un algoritmo (greedy) che risolve il problema Selezione Attività.

Esercizio [pesi]

Dato un grafo pesato G, sia T l'albero dei cammini minimi da un nodo s verso tutti gli altri nodi. Sia G' il grafo che si ottiene da G incrementando di uno il peso di ogni arco. Dimostrare oppure fornire un controesempio, che l'albero dei cammini minimi a partire da s in G' è ancora T.

Se G'' è ottenuto raddoppiando il peso di ogni arco di G, l'albero T dei cammini minimi da s è anche un albero dei cammini minimi da s in G''?