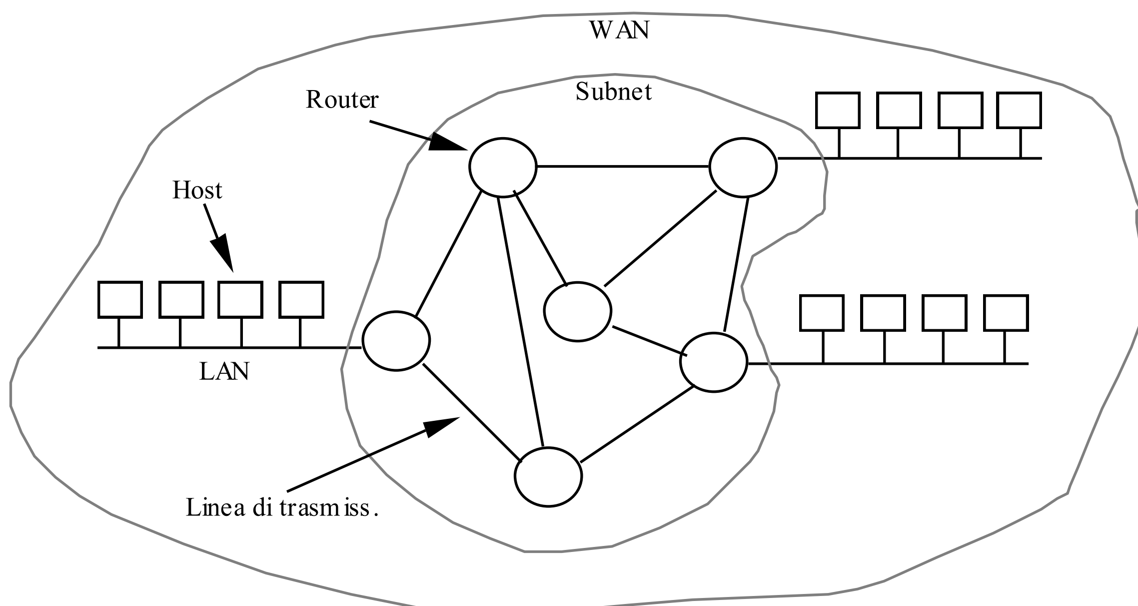
9.3.5 Reti e grafi

Numerose reti di importanza vitale per la nostra società sono modellate per mezzo di grafi. Si pensi ad esempio alle:

- reti autostradali;
- reti ferroviarie;
- reti per la distribuzione dell'energia elettrica;
- reti telefoniche;
- la rete planetaria Internet.

Nell'ambito di tali strutture un problema di grande rilevanza è determinare il **cammino di costo minimo** fra un punto ed un altro nella rete (ossia fra un vertice ed un altro nel grafo che modella la rete). In tali scenari ogni arco del grafo ha associato un **costo di attraversamento**, di norma strettamente positivo, la cui determinazione dipende da una combinazione delle grandezze in gioco nello specifico contesto (as es. distanza chilometrica e costi di trasporto e pedaggio nelle reti stradali e ferroviarie, ampiezza di banda trasmissiva e costi di affitto delle linee nelle reti telefoniche e di elaboratori, ecc.).

In particolare, la rete Internet è organizzata secondo questo schema:



Ciò che consente il dialogo fra i vari host sparsi per tutto il pianeta è una infrastruttura di rete, chiamata **subnet di comunicazione**, costituita da centinaia di migliaia di apparati (detti **router**) che sono collegati fra loro mediante una rete di collegamenti punto-a-punto e che si incaricano di inoltrare tutti i dati trasmessi da ogni host sorgente ad ogni host di destinazione. E' evidente come sia di cruciale importanza che tali dati vengano instradati, nella subnet, lungo la via più conveniente fra quelle che consentono di raggiungere la destinazione.

Esistono vari algoritmi per la ricerca dei cammini migliori, noi vedremo l'algoritmo di Dijkstra (1956).

9.3.6 Algoritmo di Dijkstra per la ricerca dei cammini minimi

Dato un grafo orientato e pesato *G* con pesi degli archi non negativi ed uno specifico ***vertice di partenza s***, l'algoritmo di Dijkstra si muove sistematicamente lungo gli archi di G per costruire un ***albero dei cammini minimi*** (che inizialmente consiste del solo vertice s, la sua radice) che, alla fine, contiene tutti i vertici raggiungibili da *s*. Per ogni vertice *v* raggiungibile da *s* il cammino da *s* a *v* nell'albero dei cammini minimi è un **cammino di peso minimo da *s* a *v***, ossia un cammino che ha la seguente proprietà:

- la somma dei pesi degli archi che costituiscono il cammino è minore o uguale alla somma dei pesi degli archi di qualunque altro cammino da s a v.

L'algoritmo di Dijkstra visita i nodi nel grafo, in maniera simile a una ricerca in ampiezza o in profondità.

In ogni istante, l'insieme dei vertici del grafo è diviso in tre parti:

- l'insieme ***VIS*** dei nodi visitati, per i quali è ormai fissato definitivamente il cammino minimo;
- l'insieme ***F*** dei nodi di frontiera, che sono successori dei nodi visitati, per i quali ancora non è definitivo il cammino minimo;
- i nodi sconosciuti, che sono ancora da esaminare.

Per ogni nodo v, l'algoritmo tiene traccia:

- di un valore *distanza(v)*, che rappresenta il costo noto finora del cammino minimo da s al vertice v;
- dell'indice *predecessore(v)*, il cui ruolo è identico a quanto discusso nelle visite in ampiezza e profondità: esso identifica il predecessore di v nell'attuale cammino minimo da s a v.

L'algoritmo consiste nel ripetere la seguente iterazione finché l'insieme F non si svuota:

- si prende dall'insieme F un qualunque nodo v avente *distanza(v)* minima;
- si sposta v da F in VIS;
- si inseriscono tutti i successori di v ancora sconosciuti in F;
- per ogni successore w di v si aggiornano i valori *distanza(w)* e *predecessore(w)*. L'aggiornamento viene effettuato con la regola:

  distanza(w) = minimo fra *distanza(w)* e (*distanza(v)* + peso dell'arco (u,w))

- se il valore di *distanza(w)* è stato effettivamente modificato, allora *predecessore(w)* viene posto uguale a v, il che ci permette di ricordare che, al momento, il cammino di peso minimo che conosciamo per arrivare da s a w ha come penultimo nodo v.

L'algoritmo segue un'idea piuttosto naturale: se sappiamo che con peso pari a *distanza(v)* possiamo arrivare fino a v, allora arrivare a w non può costare più di arrivare a v e spostarsi da v lungo l'arco (v,w) fino a w.

Si noti che una fondamentale differenza con la visita in ampiezza è legata al fatto che dobbiamo mantenere i nodi in F ordinati rispetto alla loro distanza nota da s, il che implica che la struttura d'appoggio usata nella visita non può più essere una coda ma diviene una coda con priorità, nella quale l'elemento da estrarre è quello avente distanza minima dal vertice s di partenza.

Si noti che l'algoritmo funziona anche su grafi non orientati: è sufficiente sostituire, ad ogni arco non orientato del grafo, una coppia di archi orientati in senso opposto l'uno all'altro ed aventi ciascuno lo stesso peso dell'arco non orientato che sostituiscono.

Pseudocodice

Lo pseudocodice riportato nel seguito assume che il grafo *G = (V, E)* in input sia rappresentato mediante liste di adiacenza.

Lo pseudocodice utilizza alcune informazioni aggiuntive, necessarie al suo corretto funzionamento:

- un campo *dist[v]* in ciascun vertice per gestire il costo del cammino minimo da s a v (inizialmente posto a ∞);
- un campo *pred[v]* in ciascun vertice per memorizzare il predecessore (inizialmente posto a *NULL*);
- un campo *vis[v]* in ciascun vertice per indicare se il vertice è o no in VIS (inizialmente posto a *false*);
- una coda con priorità q per gestire l'ordine di visita dei vertici: i vertici presenti via via nella coda q sono quelli dell'insieme *F*.

```
Funzione Dijkstra (G: grafo pesato con pesi non negativi; Q: coda con priorità)
       Assegna a tutti i nodi v
              dist [v] ← ∞
              pred[v] ← NULL
              vis[v] ← false
       scegli il nodo di partenza s
       dist[s] ← 0
       Enqueue(Q, s)
       finché la coda Q non è vuota
              v ← Dequeue(Q) // v è il vertice su cui lavorare
              vis[v] ← true
              //per il vertice v il cammino minimo è ora quello definitivo
              per ogni vertice w nella lista di adiacenza di v tale che vis[w]=false
                     if dist[w] > dist[v] + peso_arco(v,w) then
                            dist[w] = dist[v] + peso_arco(v,w)
                            pred[w] ← v
                     if (w non è nella coda q)
                            Enqueue(Q, w)
                     else   "aggiusta la posizione di w in Q"
       return
```

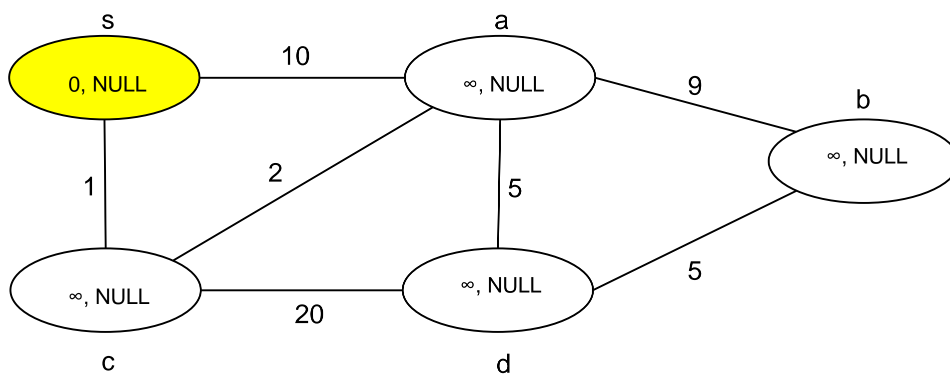
Esempio

Nell'esempio si adotta la seguente simbologia:

- il vertice s è il vertice di partenza;
- i vertici verdi sono quelli in VIS;
- i vertici gialli sono quelli in F;
- i vertici arancio sono vertici già presenti in F i cui valori vengono aggiornati durante un'iterazione;
- all'interno di ogni vertice v si indicano i valori di *distanza(v)* e *predecessore(v);*
- gli archi spessi sono quelli dell'albero dei cammini minimi.

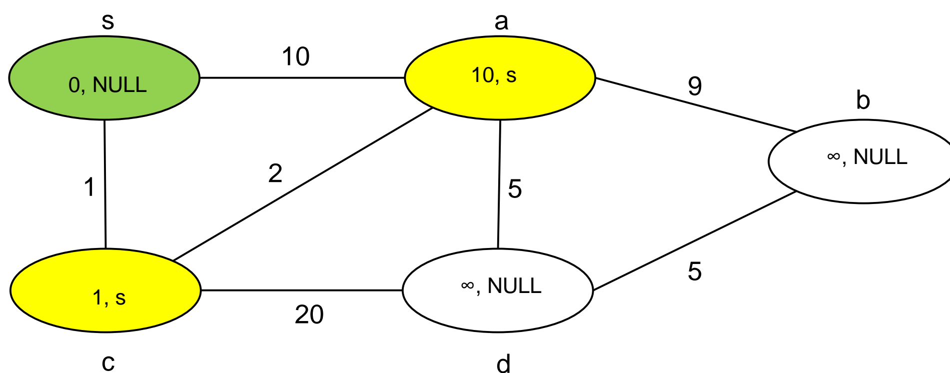
Inizializzazione

Dopo l'inizializzazione la situazione è questa (s è l'unico vertice nella coda):



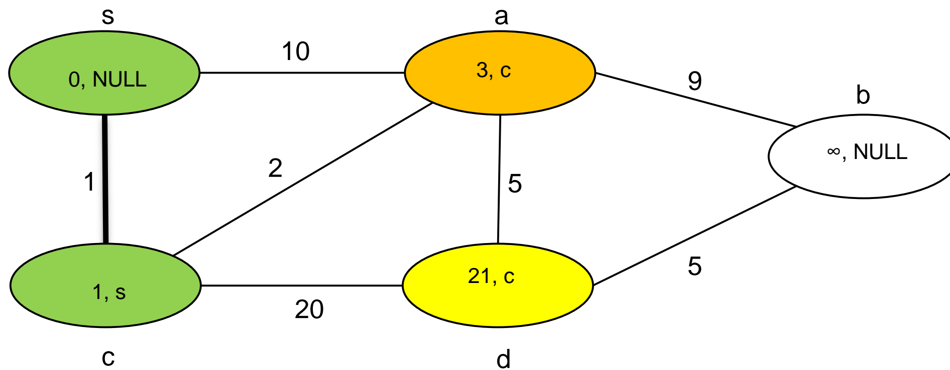
Prima iterazione

Nella prima iterazione del while si estrae dalla coda la sorgente (che quindi viene spostata in VIS) e si mettono in coda (e quindi in F) i suoi adiacenti, aggiornandone i valori:



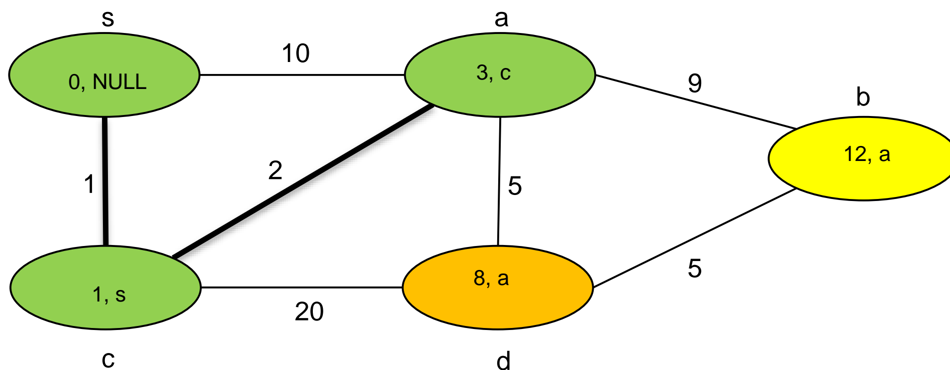
Seconda iterazione

Nella seconda iterazione fra i vertici in F si sceglie quello a distanza minima (c), che viene spostato in VIS, si aggiungono i nuovi vertici (d) scoperti tramite c e si aggiornano i valori di quelli (a) già in F:



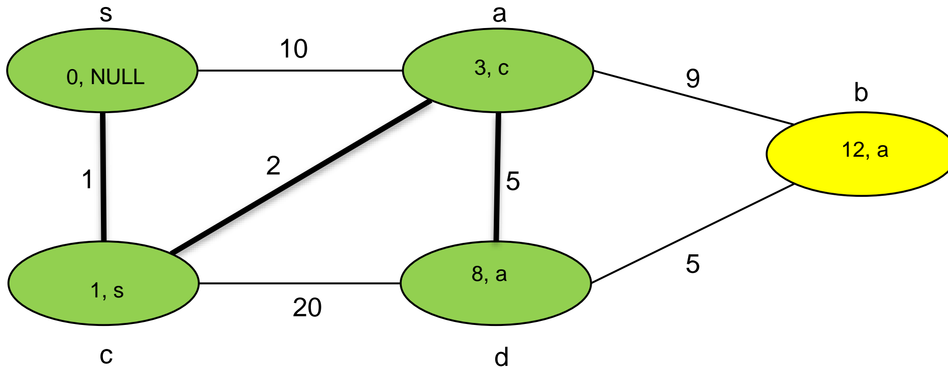
Terza iterazione

Nella terza iterazione fra i vertici in F si sceglie quello a distanza minima (a), che viene spostato in VIS, si aggiungono i nuovi vertici (b) scoperti tramite a e si aggiornano i valori di quelli (d) già in F:



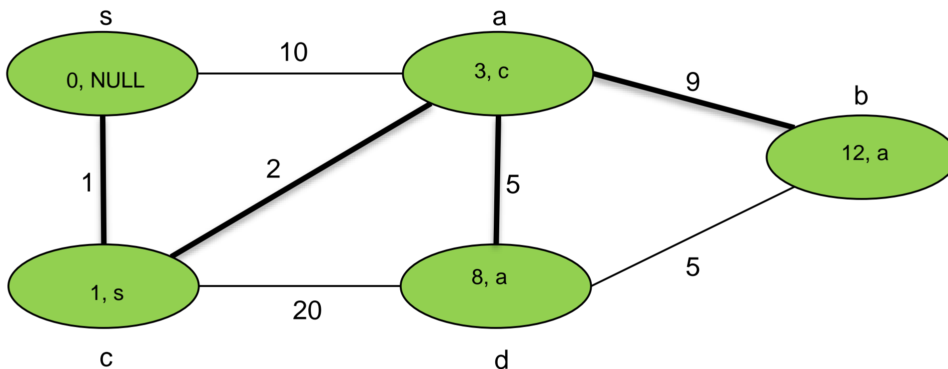
Quarta iterazione

Nella successiva iterazione fra i vertici in F si sceglie quello a distanza minima (d), che viene spostato in VIS, e si aggiornano i valori di quelli già in F (non c'è più alcun nuovo vertice da scoprire). In questo caso però l'aggiornamento non modifica i valori del vertice b dato che *distanza(d)* + *peso arco (d,b)* > *distanza(b)*:



Quinta ed ultima iterazione

Fra i vertici in F si sceglie quello a distanza minima (b), che è anche l'ultimo presente nella coda. Esso viene spostato in VIS e si termina:



Complessità

L'inizializzazione ha complessità $O(|V|)$.

L'algoritmo compie successivamente $O(|V|)$ iterazioni, dato che ad ogni iterazione si fissa il cammino minimo per un nuovo vertice (quello estratto dalla coda).

Nel complesso di tali iterazioni si effettuano:

- $O(|V|)$ inserzioni nella coda (ogni vertice è inserito una sola volta);
- $O(|V|)$ estrazioni dalla coda (ogni vertice è estratto una volta sola);
- $O(|E|)$ aggiornamenti di un vertice nella coda (un aggiornamento possibile per ogni arco del grafo).

I costi delle varie operazioni sulla coda dipendono dalla sua implementazione.

Se la coda è implementata ad esempio tramite un vettore non ordinato:

- l'inserimento ha costo $\Theta(1)$;
- l'estrazione ha costo $O(|V|)$;
- l'aggiornamento di un vertice ha costo $\Theta(1)$.

Dunque in tal caso la complessità risulta $O(|V| + |V| + |V|^2 + |E|) = O(|V|^2 + |E|)= O(|V|^2)$.

Viceversa, se la coda è implementata tramite uno heap:

- l'inserimento, con riaggiustamento dello heap, ha costo $O(log|V|)$;
- l'estrazione, con riaggiustamento dello heap, ha costo $O(log|V|)$;
- l'aggiornamento di un vertice, con riaggiustamento dello heap, ha costo $O(log|V|)$.

e la complessità diviene $O(|V| + |V|\ log|V| + |V|\ log|V| + |E|\ log|V|) = O((|V| + |E|)\ log|V|)$.

Si noti che questa complessità è migliore della precedente se il grafo è sparso, ossia ha pochi archi: in tal caso, se ad esempio $|E| = O(|V|)$ la complessità totale è $O((|V|log|V|)$.

Viceversa, essa diviene peggiore della precedente se il grafo è molto denso, ossia molto ricco di archi, poiché in tal caso $|E| = O(|V|^2)$ e il costo totale diviene $O(|V|^2\ log|V|)$.

Correttezza dell'algoritmo

La correttezza dell'algoritmo di Dijkstra si dimostra per induzione:

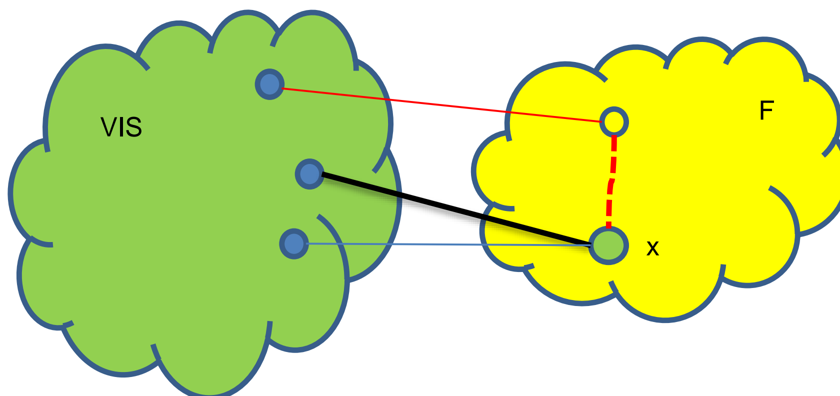
L'algoritmo è banalmente corretto quando:

- |VIS| = 1: VIS contiene solo la sorgente, che è a distanza zero da se stessa;
- |VIS| = 2: VIS contiene solo la sorgente s ed il vertice collegato ad s dall'arco di peso minimo fra quelli uscenti da s;

Ora, sia |VIS| = k; il prossimo vertice che verrà inserito in VIS è un vertice x (in verde nella figura), fra quelli in F, tale che:

- x è raggiungibile direttamente da un vertice in VIS;
- *distanza(x)* ha valore minimo fra i vertici in F.

Se esistesse un cammino C che permette di raggiungere x con un peso complessivo strettamente minore di *distanza(x)*, esso dovrebbe necessariamente passare anche per altri vertici in F dato che passando per i soli vertici in VIS la distanza minima è appunto *distanza(x)*. Ma tale cammino sarebbe costituito da una porzione in VIS, da un arco che porta da VIS a F (in rosso nella figura), più una porzione in F (in rosso tratteggiato nella figura). Dato che la somma del costo della porzione di cammino in VIS più il costo dell'arco da VIS ad F non può essere minore di *distanza(x)* (altrimenti x non sarebbe il nodo scelto) ne segue che la porzione in F dell'ipotetico cammino C dovrebbe avere peso negativo, il che contraddice l'ipotesi di partenza che tutti gli archi del grafo abbiano peso ≥ 0.



Dunque anche il nuovo insieme VIS, avente cardinalità k + 1, contiene vertici per i quali è determinato il cammino minimo da s.

CVD