Grafi

1 Introduzione e definizioni

Un grafo G è una coppia (V,E), dove V è un insieme finito di nodi, chiamati **vertici**, ed E è l'insieme degli **archi** di G che definiscono le connessioni tra i vertici. In un **grafo orientato** l'insieme degli archi E è una relazione binaria tra i vertici, mentre in un **grafo non orientato** E corrisponde ad un insieme di coppie non ordinate di vertici. In altre parole si dice che se i vertici u e v di un grafo *non orientato* sono **adiacenti** (connessi da un arco), la relazione di adiacenza è simmetrica. Nel caso di un grafo *orientato*, se u è adiacente a v, non è detto che v sia adiacente a u. Le due tipologie di arco vengono rappresentate come mostrato nella *Figura 1*.

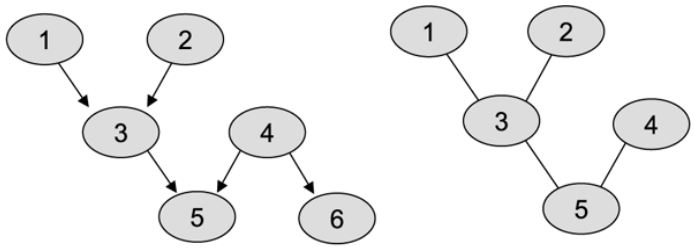

Figure 1: Grafi orientati (sinistra) e non orientati (destra)

Se (u,v) è un arco orientato di un grafo G, si dice che l'arco **esce** da u ed **entra** in v. Se invece (u,v) è un arco di un grafo non orientato, si dice che l'arco è **incidente** nei vertici u e v. In un grafo orientato sono ammessi archi che escono ed entrano nello stesso vertice, denominati **cappi** (*self-loops*); in un grafo non orientato i cappi non sono ammessi.

In un grafo non orientato il **grado** di un vertice corrisponde al numero di archi che incidono nel vertice. Per un vertice di un grafo orientato si distinguono un **grado uscente** ed un **grado entrante**, che sono rispettivamente il numero di archi uscenti ed entranti, mentre il **grado** del vertice è la somma dei due precedenti.

Un **cammino** è una sequenza di vertici di un grafo ottenuta partendo da un vertice iniziale e seguendo determinati archi fino al raggiungimento del nodo finale. La **lunghezza** del cammino è il numero di archi attraversati nel cammino. Se esiste un cammino p che

presenta come vertice iniziale u e come vertice finale v, si dice che v è **raggiungibile** da u attraverso p. Un cammino è definito **semplice** se tutti i vertici di un cammino sono distinti. In un grafo orientato, un cammino il cui nodo di partenza è lo stesso del nodo di arrivo è detto **ciclo**. Un grafo senza cicli è **aciclico**.

Un grafo *orientato* è **fortemente connesso** se due vertici qualsiasi sono raggiungibili l'uno dell'altro. Un sottoinsieme dei vertici di un grafo per il quale vale la regola sopracitata viene detto **componente fortemente connessa** del grafo. Un grafo orientato può avere molteplici componenti fortemente connesse, e un grafo è fortemente connesso se ha una sola componente fortemente connessa.

Un grafo *non orientato* è **connesso** se ogni coppia di vertici è collegata attraverso un cammino. Un generico grafo non orientato può essere caratterizzato da molteplici **componenti connesse**, per le quali vale la regola sopracitata. Un grafo non orientato è connesso di per sé se presenta una sola componente connessa.

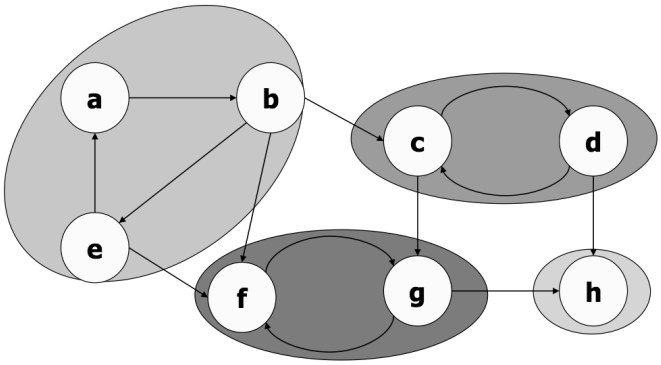

Figure 2: Grafo orientato. I vari insiemi di vertici corrispondono alle componenti fortemente connesse del grafo

Ad ogni arco di un generico grafo può essere associato un indice che ne indica il **peso** (o **costo**). In questo caso si parla di **grafo pesato**.

Infine, un grafo si dice **sparso** quando il numero di archi $|E|$ è molto più piccolo di $|V|^2$. Un grafo è invece **denso** se $|E|$ è prossimo al valore $|V|^2$.

Sulla base delle definizioni precedentemente riportate è possibile definire alcuni tipologie specifiche di grafo. Un **grafo completo** è un grafo non orientato in cui ogni coppia di vertici è adiacente. Un grafo aciclico e non orientato è una **foresta**. Un grafo aciclico, non orientato e connesso è un **albero** (senza radice). In quest'ultimo caso, ogni coppia di vertici è connessa da un singolo cammino semplice, e la rimozione di un qualsiasi arco comporta la disconnessione del grafo (generando una foresta); inoltre, l'aggiunta di un arco

tra due vertici qualsiasi introduce un ciclo facendo si che il grafo non sia più un albero.

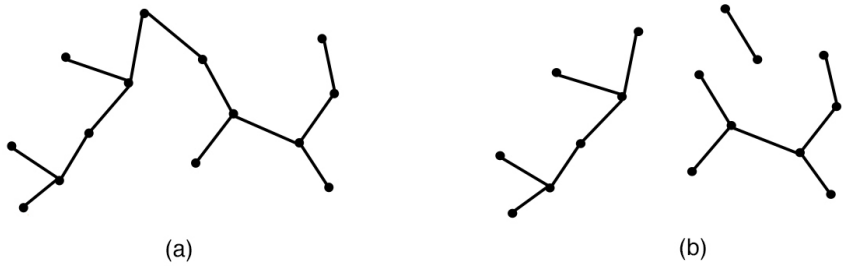

Figure 3: (a) Un albero libero; (b) Una foresta

Se in un albero come appena descritto viene identificato un **nodo radice**, si possono instaurare tutte le regole di parentela tra nodi. A tal proposito si rimanda al paragrafo introduttivo del capitolo inerente gli alberi per una più approfondita trattazione.

2 Rappresentazioni dei grafi

Un grafo è l'ideale per modellare numerose tipologie di problemi, per la soluzione dei quali è possibile adottare un determinato algoritmo. In generale, tutti gli algoritmi sui grafi sono basati su tecniche di ricerca in un grafo; effettuare ricerche in un grafo significa seguire gli archi del grafo per visitarne tutti i nodi ed estrapolare dall'operazione alcune informazioni.

In questo contesto, la rappresentazione dei grafi in memoria risulta essere un concetto chiave. In questo capitolo vengono proposti i due metodi standard per la memorizzazione di un grafo: la *rappresentazione con liste di adiecanza* e la *rappresentazione con matrici di adiacenza*.

2.1 Rappresentazione con liste di adiacenza

Questo metodo fa uso di un array A di $|V|$ liste, una per ogni vertice dell'insieme V. Per ogni nodo u del grafo, la lista $A[u]$ contiene tutti i vertici v tali per cui esiste un arco $(u, v) \in E$ (cioè tutti i vertici adiacenti a u).

Se il grafo G considerato è orientato, la somma delle lunghezze di tutte le liste di adiacenza è esattamente $|E|$, mentre se il grafo non è orientato la somma corrisponde a $2|E|$, in quanto per ogni arco non orientato (u, v), u appartiene alla lista di adiacenza di v e viceversa.

La quantità di memoria richiesta con questa tecnica di rappresentazione è $\Theta(V + E)$. La rappresentazione con liste di adiacenza è una soluzione molto compatta nel caso di grafi

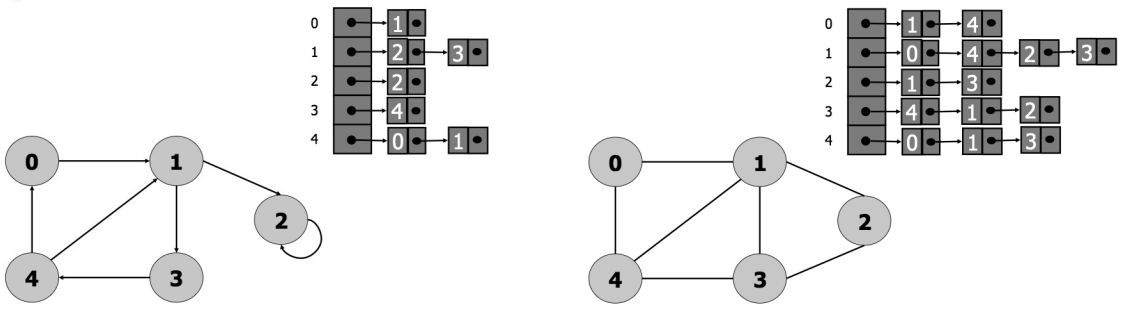

Figure 4: Rappresentazione con liste di adiacenza con grafi orientati (sinistra) e non orientati (destra)

sparsi ed è anche molto robusta, cioè supporta molte varianti di grafi con minime modifiche; ad esempio, per i grafi pesati è sufficiente memorizzare il peso $w(u, v)$ assieme al vertice v nella lista di adiacenza di u.

Un possibile svantaggio di questo metodo riguarda l'impossibilità di determinare in modo efficiente se un certo arco (u, v) è presente nel grafo: tale analisi necessita la scansione della lista di adiacenza $A[u]$. Da questo punto di vista assai migliore è la memorizzazione tramite matrice di adiacenza.

2.2 Rappresentazione con matrice di adiacenza

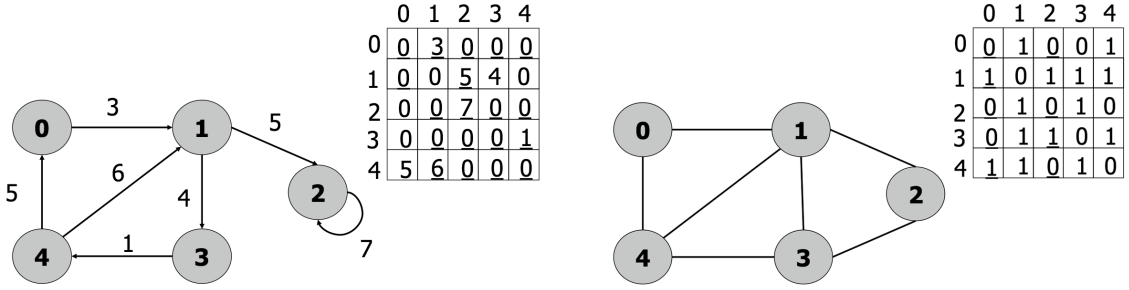

Figure 5: Rappresentazione con matrici in caso di orientati pesati (sinistra) e grafi non orientati (destra)

Dato un grafo $G = (E, V)$, la rappresentazione con matrice di adiacenza fa uso di una matrice $M = (a_{ij})$ di dimensione $|V| \times |V|$. Tutti i vertici del grafo sono numerati in modo arbitrario, e nel caso in cui esista l'arco (i, j) viene contrassegnata la cella della matrice a_{ij}

mediante l'assegnazione del valore di flag 1. Se l'arco non è previsto nel grafo in questione, il valore 0 viene assegnato. Dunque, indipendentemente dal numero di archi, la memoria necessaria in questo caso è $\Theta(V^2)$. In prima approssimazione, la rappresentazione con matrice di adiacenza è una buona soluzione in caso di grafi densi. Un'ottimizzazione in termini di memoria occupata è possibile per i grafi non orientati: in questo caso la matrice risultante è sempre simmetrica rispetto alla diagonale principale (*Figura 5, destra*), ed è quindi possibile dimezzare i dati memorizzati senza perdita di informazione.

Questa tipologia di rappresentazione è anche ottima nel caso dei grafi pesati, in quanto per salvare il peso associato a un arco è sufficiente sostituire il flag 1 nella posizione corrispondente della matrice con il valore stesso del peso. Questa soluzione non aumenta la memoria richiesta come invece avviene nel caso dei grafi pesati rappresentati mediante liste di adiacenza, in cui un valore a parte deve essere aggiunto per ogni arco.

Infine è importante notare quanto la topologia del grafo sia molto più accessibile con il metodo delle matrici, in quanto per verificare l'esistenza di un arco è sufficiente controllare la posizione corrispondente nella matrice.

3 Algoritmi di ricerca

Eseguire un algoritmo di ricerca su un grafo è detto anche **visitare** il grafo. Il primo passo di una visita consiste nella scelta di un nodo di partenza, la **sorgente**, da cui si raggiungono gli altri nodi del grafo seguendo gli archi sotto determinate regole. Durante questo processo è possibile memorizzare certe informazioni sulla struttura e contenuto del grafo. Durante la visita i nodi vengono concettualmente divisi per colori, così da poter tenere traccia del lavoro svolto:

- *Bianco*: il nodo non è ancora stato visitato (non è stato scoperto);
- *Grigio*: il nodo è stato scoperto (visitato per la prima volta) ma non ancora completato (non sono stati ancora ispezionati tutti gli archi connessi al nodo);
- *Nero*: il nodo è stato scoperto e completato (tutti gli archi connessi al nodo sono stati ispezionati)

Esistono due principal algoritmi di ricerca per i grafi, sia orientati che non orientati: *visita in profondià* (depth-first search, DFS) e *visita in ampiezza* (breadth-first search, BFS).

3.1 Visita in profondità (DFS)

Partendo da un nodo sorgente s scelto in modo arbitrario tra quelli di un grafo G, la visita in profondià tocca tutti i nodi raggiungibili da s. Successivamente, se vi sono altri nodi non ancora visitati, tra questi viene scelto un nuovo nodo sorgente e la ricerca prosegue.

La visita termina quando tutti i nodi del grafo sono stati visitati (anche se inizialmente non tutti sono raggiungibili, come nel caso di un grafo non orientato con più componenti connesse).

L'algoritmo di visita in profondià si può suddividere in due componenti:

- **GRAPHsearch**: procedura in cui si visitano tutti i nodi di un grafo mediante un ciclo (non ricorsivo) finché tutti i nodi non siano diventati neri. Se un nodo visitato v è classificato come bianco viene richiamata la procedura ricorsiva *dfsR*.

- **dfsR**: procedura ricorsiva di visita in profondità a partire dal nodo sorgente v passato dalla *GRAPHsearch*. Con la *dfsR* gli archi vengono ispezionati a partire dall'ultimo vertice scoperto che ha ancora archi non ispezionati che escono da esso. Quando tutti gli archi di un certo nodo u sono stati ispezionati, si procede con l'ispezione degli archi del nodo dal quale u era stato scoperto. *dfsR* termina quando tutti i nodi raggiungibili dal nodo sorgente sono stati completati (neri).

Durante la visita, per ogni nodo viene registrato il **tempo di scoperta** (quando il nodo diventa grigio) e il **tempo di completamento** (quando il nodo diventa nero): il *tempo* consiste in un contatore che viene incrementato ad ogni fase della visita e i due parametri vengono salvati usando due array i cui indici sono associati ai vari nodi del grafo.

La visita di un grafo G costruisce anche una **foresta DF** (depth-first), tale per cui per ogni sorgente identificata durante la visita del grafo si genera un nuovo albero DF contente tutti i nodi raggiungibili dalla suddetta sorgente (oltre che la sorgente stessa). I nodi dei vari alberi DF vengono aggiunti e posizionati a seconda di come procede la visita del grafo. A partire dalla foresta DF risultante è possibile classificare gli archi di un grafo in diverse tipologie, utili per ulteriori analisi sul grafo (per esempio un grafo orientato è aciclico se e soltanto se una visita in profondità non genera "archi all'indietro"):

- *Archi d'albero*: l'arco (u, v) è un arco d'albero se v viene scoperto la prima volta durante l'ispezione del suddetto arco.

- *Archi all'indietro*: sono quegli archi che in un albero DF connettono un nodo con un suo antenato. Anche i cappi, che possono presentarsi nei grafi orientati, sono considerati archi all'indietro.

- *Archi in avanti*: sono gli archi (diversi da quelli d'albero) che connettono un nodo di un albero DF con un suo discendente.

- *Archi trasversali*: tutti gli altri archi.

Tali classificazioni valgono sia per i grafi orientati che non orientati; tuttavia, occorre specificare che in una visita di un grafo non orientato non si presentano mai archi in avanti e trasversali.

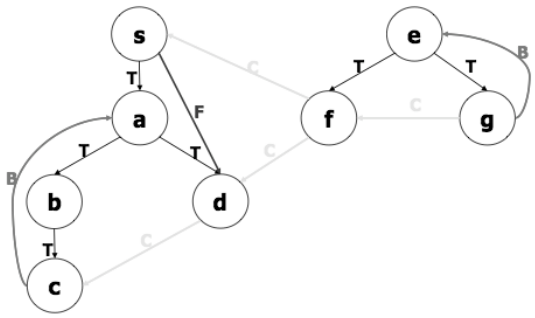

Figure 6: Esempio di classificazione degli archi per un grafo orientato, con i nodi s e e selezionati come sorgenti per la visita: [T]: archi d'albero; [B] archi all'indietro; [F] archi in avanti; [C] archi trasversali

Per ciò che riguarda l'analisi di complessità, questa può essere divisa per le due componenti dell'algoritmo: la parte del *GRAPHsearch* include un processo di inizializzazione che ha un costo per il quale si impiega un tempo $\Theta(|V|)$. La procedura *dfsR* viene invece invocata una volta per ogni arco del grafo: $\Theta(|E|)$. Se consideriamo l'adozione della rappresentazione del grafo con liste di adiacenza, il tempo di esecuzione della visita in profondità è dunque: $\Theta(|V| + |E|)$. Una memorizzazione con matrice di adiacenza comporta un costo $\Theta(|V|^2)$.

3.2 Visita in ampiezza (BFS)

La visita in ampiezza ispeziona sistematicamente gli archi di un grafo G per visitare tutti i nodi raggiungibili da un arbitrario nodo sergente s. Durante questo processo, l'algoritmo calcola il minimo numero di archi da s a ciascun vertice raggiungibile, e salva l'informazione in un array i cui indici sono associati ai vari nodi del grafo. Inoltre viene generato un **albero BF** (breadth-first) con radice s e contenente tutti i nodi raggiungibili da s, memorizzati in modo che il cammino unico nell'albero BF che va da s a v corrisponde a un cammino minimo (percorso che contiene il minor numero di archi) da s a v nel grafo G.

L'algoritmo di visita in ampiezza necessita di una coda Q con schema FIFO per gestire l'insieme dei nodi grigi mentre si procede con la visita. Inizialmente viene inserito nella coda l'elemento sorgente s (a cui è associata una distanza 0 da s, se stesso). Successivamente l'algoritmo si ripete sempre allo stesso modo:

- Estrarre un elemento v dalla coda. Il nodo diventa nero;
- Accodare tutti i nodi bianchi adiacenti al nodo v precedentemente estratto. Questi nodi diventano grigi e ad essi viene associato il valore di distanza minima da s, uguale

allo stesso valore assegnato a v e incrementato di una unità) ;

- Ripetere fino a quando non vi sono più elementi da poter estrarre dalla coda.

A seconda dell'ordine con cui vengono accodati i nodi bianchi adiacenti ad un certo nodo, l'albero BF risultante può variare leggermente, ma rimangono invariate le distanze calcolate dall'algoritmo per ognuno dei nodi del grafo.

Il tempo di esecuzione totale di BFS, come per il procedimento DFS, dipende da come il grafo è memorizzato nella memoria: nel caso di matrici di adiacenza, $T(n) = \Theta(|V|^2)$; nel caso di rappresentazione con liste di adiacenza: $T(n) = O(|V| + |E|)$.

4 Applicazioni degli algoritmi di ricerca

In questo paragrafo vengono brevemente presentati alcuni esempi di come gli algoritmi di ricerca trattati possano essere utilizzati per ricavare informazioni sulla struttura di un grafo.

4.1 Identificazione di cicli

Un grafo è aciclico se e solo se in seguito ad una visita in profondità DFS non viene marcato alcun arco come "arco all'indietro".

4.2 Componenti connessi

Per un grafo non orientato ogni albero della foresta DF corrisponde a una componente connessa del grafo.

4.3 Connettività

Dato un grafo non orientato e connesso è possibile identificare i nodi o gli archi la cui rimozione comporterebbe la disconnessione del grafo. In questo caso si parla di **ponte** per gli archi e di **punto di articolazione** per i nodi.

Un arco marcato come "arco all'indietro" non può essere un ponte. Un "arco d'albero" (u, v) è un ponte se e solo se non ci sono altri "archi all'indietro" che connettono un discendente di v con un antenato di u nell'albero DF.

Se la radice dell'albero DF ha almeno due figli allora è un punto di articolazione del grafo. Qualsiasi altro nodo v è un punto di articolazione se e solo se v ha un figlio u tale per cui non vi è alcun "arco all'indietro" da u o da uno dei suoi discendenti a un antenato proprio di v.

4.4 Ordinamento topologico

L'ordinamento topologico di un grafico orientato aciclico, o *dag* (dall'inglese *directed acyclic graph*), è un ordinamento lineare di tutti i suoi vertici tale che per ogni arco (u, v), u appare prima di v nell'ordinamento. Tale ordinamento si può intendere come una disposizione dei vertici di un *dag* lungo una linea orizzontale in modo che tutti gli archi del grafo (orientati) siano diretti da sinistra a destra. Questo procedimento può essere usato in molte applicazioni per modellare precedenze fra gli eventi.

L'ordinamento topologico di un *dag* può essere ottenuto mediante una visita in profondità del grafo: i tempi di completamento ottenuti dalla visita rappresentano l'ordine con cui disporre i nodi in modo da ottenere un orientamento topologico inverso, per il quale gli archi sono orientati da destra verso sinistra e per ogni arco (u, v) il nodo u appare alla destra del nodo v.

4.5 Componenti fortemente connesse

È possibile identificare le componenti fortemente connesse di un grafo orientato mediante l'algoritmo di Kosaraju, che fa uso della procedura di visita in profondità. L'algoritmo prevede i seguenti passaggi:

- Ottenere il grafo trasposto di G, G^{T}. Dato il grafo $G = (V, E)$, il trasposto G^{T}=(V, E^{T}) è tale per cui per ogni arco $(u, v) \in E$ esiste l'arco (v, u) in E^{T}.
- Eseguire DFS su G^{T}, ottenendo i tempi di scoperta e tempi di completamento di ogni nodo.
- Eseguire una nuova DFS sul grafo G originale, facendo sì che nel ciclo principale in *GRAPHsearch* i vertici vengano considerati in ordine decrescente rispetto ai tempi di completamento calcolati dalla DFS del punto precedente.
- Gli alberi della foresta DF generata nel punto precedente corrispondono alle componenti fortemente connesse del grafo G.

5 Alberi di connessione minimi (MST)

Dato un grafo connesso e non orientato $G = (V, E)$, in cui a ogni arco $(u, v) \in E$ è associato un peso $w(u, v)$, un *albero di connessione minimo* (o "MST", dall'inglese *minmum spanning tree*) corrisponde ad un sottoinsieme aciclico $T \subseteq E$ che collega tutti i vertici in V e il cui peso totale (la somma dei pesi dei singoli archi) sia minimo. Per un certo grafo possono esistere molteplici alberi di connessione minimi.

Due algoritmi permettono di identificare alberi di connessione minimi: l'algoritmo di **Kruskal** e l'algoritmo di **Prim**. Entrambi gli algoritmi sono *algoritmi golosi*. In generale

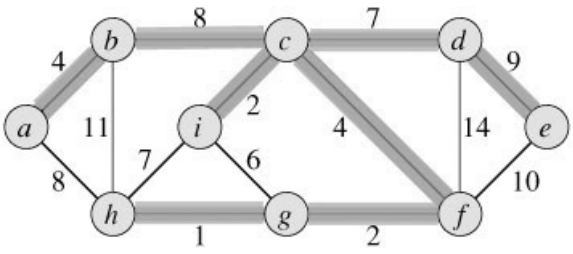

Figure 7: Gli archi evidenziati rappresentano un albero di connessione minimo per il grafo connesso d'esempio

un algoritmo goloso, basato sul fare le scelte che sembrano migliori in un particolare momento, non garantisce la soluzione globale ottimale. Tuttavia può essere dimostrato che i due algoritmi succitati, dato un grafo (connesso, non orientato e pesato) forniscono sempre l'albero di connessione minimo.

Sia Kruskal che Prim sono due algoritmi che si basano sul seguente **algoritmo generico**: dato un grafo $G = (V, E)$ come descritto in precedenza, viene definito un sottoinsieme A, inizialmente vuoto, degli archi in E; ad A viene aggiunto uno alla volta un arco, detto **arco sicuro**, tale per cui il sottoinsieme A rappresenti ancora un sottoinsieme di qualche albero di connessione minimo. Tale processo iterativo termina quando gli archi in A raggiungono tutti i vertici del grafo, e l'albero di connessione minimo è stato costruito.

L'intero problema per gli MST si riduce quindi all'identificazione, per ogni iterazione, di un arco sicuro. Per affrontare il problema è necessario introdurre alcune definizioni:

- Dato un grafo non orientato $G = (V, E)$, un **taglio** è una partizione di V in S e $V - S$. Dunque l'insieme dei vertici V è uguale a: $V \cup (V - S)$. Inoltre, $S \cap (V - S) = \oslash$.
- Si dice che un arco $(u, v) \in E$ **attraversa** un taglio $(S, V - S)$ se uno dei due vertici u e v si trova in S e l'altro in $V - S$.
- Un taglio **rispetta** un insieme A di archi se nessun arco dell'insieme A attraversa il taglio.
- Un arco che attraversa un taglio è definito **arco leggero** se il suo peso è minimo fra i pesi degli archi che attraversano il taglio stesso.

Partendo dalle precedenti definizioni è possibile introdurre il **teorema degli archi sicuri**: dato un grafo $G = (V, E)$ (connesso, non orientato e pesato), si definisce A come in sottoinsieme di E, nonché un sottoinsieme di un qualche albero di connessione minimo per G; se possiamo identificare un taglio qualsiasi di G che rispetta A e un *arco leggero* che lo attraversa, allora quell'arco è sicuro per A.

Un **corollario** del teorema degli archi sicuri stabilisce che, dato un grafo G come descritto in precedenza e lo stesso sottoinsieme A contenuto in qualche albero di connessione minimo per G, viene chiamata $C = (V_C, E_C)$ una componente connessa (un albero) della foresta $G_A = (V, A)$; un *arco leggero* che collega C con qualche altra componente in G_A è un arco sicuro per A.

5.1 Algoritmo di Kruskal

L'algoritmo di Kruskal, basato sull'algoritmo generico descritto precedentemente, adotta il corollario del teorema degli archi sicuri per selezionare ad ogni iterazione un arco sicuro e costruire così l'albero di connessione minimo.

In questo algoritmo l'insieme A forma una foresta $G_A = (V, A)$, i cui alberi inizialmente corrispondono ai singoli vertici del grafo G (in quanto inizialmente l'insieme A è vuoto). Ad ogni iterazione dell'algoritmo generico viene selezionato un arco sicuro da aggiungere ad A, cioè un arco di peso minimo che collega due alberi qualsiasi della foresta G_A. Da questo procedimento si evince il motivo per il quale l'algoritmo di Kruskal rientra nella categoria degli algoritmi golosi: ad ogni passaggio si aggiunge alla foresta un arco con il minor peso possibile in quel momento. Le successive connessioni terminano al raggiungimento di un unico albero che contiene tutti i vertici del grafo iniziale.

Il tempo di esecuzione dell'algoritmo di Kruskal dipende dall'implementazione della struttura dati. Adottando un'implementazione efficiente è possibile raggiungere un complessità $T(n) = O(|E|lg|E|)$.

5.2 Algoritmo di Prim

Come per l'algoritmo di Kruskal, anche l'algoritmo di Prim si basa sull'algoritmo generico degli alberi di connessione minimi, ma in questo caso viene adottato il teorema degli archi sicuri (e non il suo corollario) per identificare, ad ogni iterazione, un arco sicuro.

Con l'algoritmo di Prim gli archi nell'insieme A formano sempre un singolo albero, che inizia da un arbitrario vertice del grafo per poi svilupparsi fino a coprire tutti i vertici in V. Un taglio divide i vertici raggiunti dagli archi in A da quelli non ancora raggiunti. Ad ogni passaggio viene aggiunto ad A un arco leggero che attraversa il taglio (arco sicuro); prima di ripetere l'operazione, il taglio viene modificato perché il nuovo vertice raggiunto dall'albero formato dagli archi in A venga inserito nel sottoinsieme opportuno di V. La procedura termina quando l'albero formato dagli archi in A raggiunge tutti i vertici V, diventando così un albero di connessione minimo del grafo.

Adottando strutture dati efficienti, l'analisi di complessità dell'algoritmo di Prim produce il seguente risultato: $T(n) = O(|E| + |V|lg|V|))$.

6 Cammini minimi da sorgente unica (SSSP)

Il problema dei *cammini minimi da sorgente unica* (o "SSSP", dall'inglese *single source shortest paths*), si propone di ispezionare un grafo di partenza $G = (V, E)$, che sia **orientato e pesato**, per trovare i cammini minimi che partendo da un arbitrario vertice sorgente $s \in V$ raggiungono ciascun vertice $v \in V$. In altre parole, per ogni vertice $v \in V$ si vuole trovare un cammino con peso minimo da s a v.

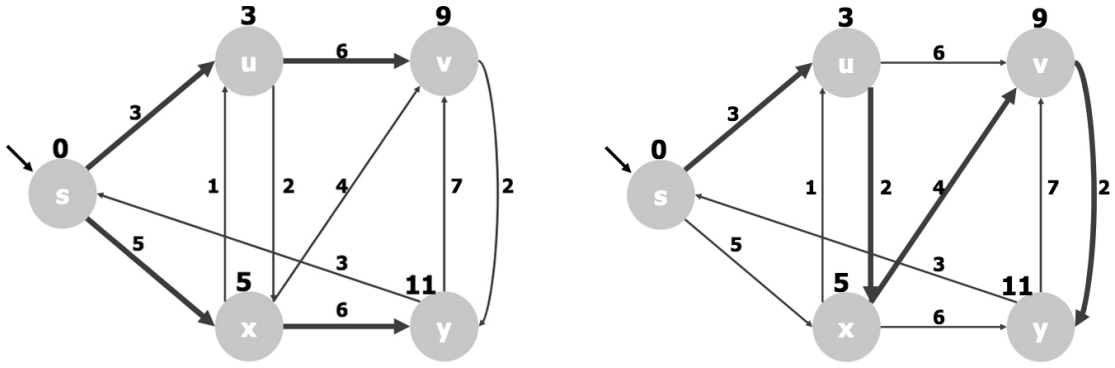

Figure 8: Per il grafo orientato e pesato d'esempio, due possibili cammini minimi da sorgente s sono indicati dagli archi più spessi. I valori evidenziati sopra i vari nodi indicano il peso del cammino minimo da s per ciascun nodo

Il peso di un cammino è definito come la somma dei pesi degli archi che lo compongono; il peso di un arco (u, v) viene indicato come $w(u, v)$. Il **peso di un cammino minimo** tra due nodi u e v è pari al peso del cammino che è il minore tra tutti i pesi dei possibili cammini che connettono u e v oppure è infinito se non esiste un cammino tra u e v. Un **cammino minimo** tra i nodi u e v è dunque un qualsiasi cammino tra u e v il cui peso è uguale al peso minimo (molteplici cammini possono corrispondere alla descrizione, infatti cammini minimi tra due nodi non sono sempre unici).

Il concetto di **peso** (o **costo**) associato agli archi di un grafo può essere utile per modellare in generale qualsiasi quantità penalizzante che si accumula linearmente lungo un cammino e che occorre minimizzare; ad esempio se il grafo è usato per modellare le connessioni stradali tra vari centri urbani, il costo degli archi può indicare la distanza tra le varie città.

L'algoritmo che si sceglie di adoperare per identificare i cammini minimi da sorgente unica, può risolvere numerose varianti del problema, incluse le seguenti:

- *Problema dei cammini minimi con destinazione unica*: trovare un cammino minimo da un ciascun vertice v a un dato vertice destinazione t. Il problema è simmetrico a

quello dei cammini minimi da sorgente unica;

- *Problema del cammino minimo per una coppia di vertici*: trovare un cammino minimo da u a v, noti i vertici u e v. La soluzione del problema dei cammini minimi da sorgente unica risolve anche questo problema;

- *Problema dei cammini minimi fra tutte le coppie di vertici*: trovare un cammino minimo da u a v per ogni coppia di vertici u e v. Questo problema può essere risolto applicando l'algoritmo per sorgente unica una volta per ogni nodo del grafo di partenza, anche se esistono soluzioni più efficienti.

I due algoritmi proposti in questo capitolo per la soluzione del problema dei cammini minimi da sorgente unica sono l'algoritmo di **Dijkstra** e l'algoritmo di **Bellman-Ford**. I due algoritmi si comportano in modo differente in presenza di archi con peso negativo e cicli con peso negativo (cicli per i quali la somma dei pesi degli archi che li compongono sia negativa).

Se esistono archi del grafo con peso negativo ma non esistono cicli negativi, l'algoritmo di Dijkstra non garantisce la soluzione ottimale al problema, mentre l'algoritmo di Bellman-Ford garantisce la soluzione ottimale.

Se esistono cicli con peso negativo **non esiste una soluzione** al problema dei cammini minimi da sorgente unica: l'algoritmo di Dijkstra mostra un risultato senza valore, mentre l'algoritmo di Bellman-Ford è in grado di identificare i cicli negativi e non mostra alcun risultato.

Nel contesto degli SSSP i cicli negativi non permettono di raggiungere un risultato valido in quanto i cammini verso i nodi facenti parte del ciclo (ed eventualmente altri nodi) risulterebbero avere un peso di cammino minimo negativo infinito; infatti è sempre possibile trovare un cammino di peso minore seguendo il ciclo di peso negativo.

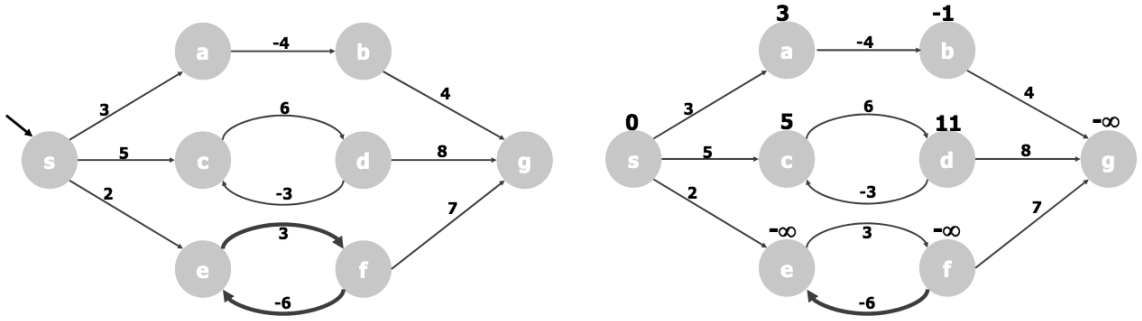

Figure 9: Soluzione (senza valore) del problema dei cammini minimi da sorgente unica in caso di cicli negativi

Se si sceglie di rappresentare la soluzione del problema SSSP tramite un albero dei cammini minimi si può notare il forte legame con la procedura di visita in ampiezza di un grafo e l'albero BF risultante. Il problema dei cammini minimi per grafi non pesati è infatti risolubile adottando la procedura BFS.

In relazione ai cammini minimi viene ora introdotto un importante lemma: i **sotto-cammini di cammini minimi sono cammini minimi**. Per sottocammino si intende una porzione del cammino originario, tale da connettere due nodi qualsiasi i e j attraversati dal cammino originario. Il sottocammino sarà dunque un cammino minimo da i a j.

È possibile suddividere un cammino minimo p tra i vertici s e v in due parti: un sottocammino da s a u e un arco (u, v); seguendo il lemma precedente, il peso del cammino minimo tra s e v corrisponde al peso del cammino minimo tra s e u più il peso dell'arco (u, v) ottenuto dalla suddivisione del cammino minimo originario. Dunque, per $\forall(u, v) \in E$, il cammino minimo tra s e v non può avere un peso maggiore del peso del cammino minimo da s a u sommato al peso di un arco (u, v).

Gli algoritmi di Dijkstra e Bellman-Ford adottano la tecnica del **rilassamento**. Dato un grafo $G = (V, E)$ come definito precedentemente (orientato e pesato), per ogni nodo $v \in V$ manteniamo un attributo $wt[v]$ che chiamiamo *stima del cammino minimo* per quel dato nodo. Il processo di rilassamento di un arco (u, v) consiste nel verificare se, passando per u, è possibile migliorare la stima del cammino minimo di v; in altre parole viene verificato se la somma $wt[u] + w(u, v)$ è minore del valore $wt[v]$. Se ciò è verificato viene aggiornata la stima del cammino minimo di v con il valore $wt[u] + w(u, v)$, e gli algoritmi di Dijkstra o Bellman-Ford registrano l'operazione per il raggiungimento del risultato finale (ottimale per tutti i nodi). Dopo il rilassamento di un arco (u, v) o viene aggiornata (diminuita) la stima $wt[v]$ oppure non viene modificato nulla.

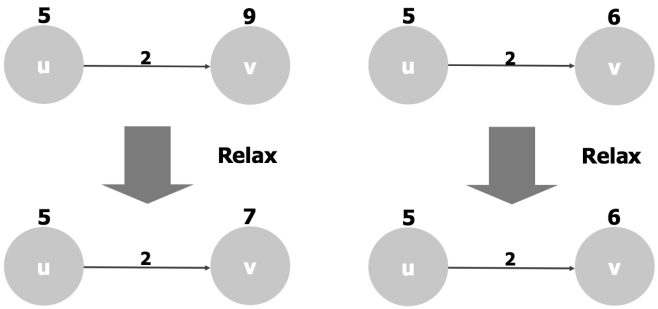

Figure 10: Due esempi di rilassamento di un arco (u, v)

Vengono ora proposti due ulteriori lemmi sul rilassamento, su cui sono basati gli algoritmi di Dijkstra e Bellman-Ford:

- Quando $wt[v]$ corrisponde al peso del cammino minimo dalla sorgente s al nodo v, non è più possibile ridurre $wt[v]$ tramite rilassamento.
- Quando un cammino minimo da s a v è scomponibile in un cammino minimo tra s e u e un arco $e = (u, v)$, se prima del rilassamento dell'arco e la stima $wt[u]$ è uguale al peso del cammino minimo tra s e u, dopo il rilassamento la stima $wt[v]$ è uguale al peso del cammino minimo tra s e v.

Per l'ottenimento dei cammini minimi da sorgente unica, l'algoritmo di Dijkstra applica il rilassamento una volta per ogni arco del grafo, mentre lo stesso arco viene rilassato più volte con l'algoritmo di Bellman-Ford.

6.1 L'algoritmo di Dijkstra

Per garantire la correttezza del risultato finale, l'algoritmo necessita che non siano presenti archi di peso negativo. I vertici del grafo di partenza vengono suddivisi in due sottoinsiemi: l'insieme S dei nodi per i quali è stato compiuto il cammino minimo, e l'insieme $V - S$, i cui nodi (non ancora completati) sono inseriti in una coda min-prioritaria Q. Ad ogni iterazione dell'algoritmo viene estratto un nodo dalla coda per essere elaborato e aggiunto ad S. L'algoritmo termina quando la coda è vuota.

L'algoritmo prevede una prima fase di inizializzazione durante la quale la stima del cammino minimo del nodo scelto come sorgente s è impostata a 0 mentre la stima per tutti gli altri nodi è impostata ad un valore sufficientemente grande $maxWT$. Successivamente la coda viene riempita con tutti i nodi del grafo, seguendo una priorità tanto maggiore quanto minore è la stima del cammino minimo.

Dopo la fase di inizializzazione inizia un processo iterativo per il quale ad ogni ciclo viene estratto dalla coda l'elemento u avente il minimo valore di stima del cammino minimo $wt[u]$; tale elemento (che per la prima iterazione sarà il nodo sorgente), viene aggiunto all'insieme S e per *ogni* vertice v adiacente a u, viene rilassato l'arco (u, v) (in altre parole, vengono rilassati tutti gli archi che escono da u).

La necessità di scegliere, ad ogni iterazione, il vertice "più leggero" della coda Q è il motivo per cui l'algoritmo di Dijkstra è definito come un algoritmo goloso, anche se è caratterizzato dalla proprietà non ordinaria di fornire sempre la soluzione globale corretta (se vengono rispettate certe condizioni sulle proprietà del grafo di partenza).

La complessità dell'algoritmo di Dijkstra dipende dal metodo d'implementazione della coda di priorità. Per l'analisi si possono studiare le operazioni critiche di cui l'algoritmo è composto. Se viene adottato un heap di Fibonacci per implementare la coda di priorità, l'estrazione di un elemento ha un costo $O(lg|V|)$ e tale operazione è eseguita $|V|$ volte in quanto ogni nodo è estratto una sola volta. L'operazione di rilassamento richiede implicitamente delle operazioni di diminuzione di valore chiave per gli elementi della coda; tali

operazioni vengono eseguite con un costo unitario nel caso dell'heap di Fibonacci. Il rilassamento è eseguito esattamente una volta per ogni arco in E. È dunque possibile ottenere un tempo di esecuzione pari a $O((|V|lg|V|) + |E|)$ se viene usato l'heap di Fibonacci per implementare la coda di priorità.

6.2 L'algoritmo di Bellman-Ford

L'algoritmo consente la presenza di archi con peso negativo, ed è in grado di identificare i possibili cicli negativi presenti nel grafo; se almeno un ciclo negativo è presente, l'algoritmo indica che il problema non ha soluzione.

La procedura Bellman-Ford necessita di una prima fase di inizializzazione delle stime dei cammini minimi simile a quella vista per l'algoritmo di Dijkstra: la stima di cammino minimo del nodo sorgente è impostata a 0, mentre la stima degli altri nodi è impostata a *maxWT*. Successivamente è previsto un ciclo per cui ad ogni iterazione *tutti* gli archi in E vengono rilassati senza un preciso ordine; tale ciclo viene eseguito un numero di volte pari al numero di vertici del grafo diminuito di uno, $|V| - 1$. Al termine di questo processo viene eseguito il test per verificare la presenza di cicli negativi, che consiste in un ulteriore rilassamento per ogni arco: infatti, se al rilassamento $|V|$-esimo di uno degli archi viene migliorata una stima di cammino minimo, si può dimostrare che il grafo presenta almeno un ciclo negativo, e l'algoritmo non fornisce alcuna soluzione.

L'algoritmo di Bellman-Ford necessità di un numero di operazioni di rilassamento relativamente molto maggiore rispetto alla soluzione di Dijkstra. Tuttavia è possibile ottimizzare l'algoritmo in questione seguendo la regola per la quale se ad un qualsiasi ciclo *precedente* al ciclo $|V|$-esimo (di verifica) nessuna stima viene migliorata, la procedura può terminare garantendo che la soluzione ottimale è stata raggiunta.

L'algoritmo di Bellman-Ford prevede un'inizializzazione che richiede un tempo $\Theta(|V|)$, ciascuno dei $|V| - 1$ cicli di rilassamento richiede un tempo $\Theta(|E|)$ e l'ultimo ciclo di verifica richiede un tempo $O(|E|)$. L'algoritmo viene dunque eseguito nel tempo $O(|V||E|)$.