Algoritmi

UniVR - Dipartimento di Informatica

Fabio Irimie

Corso di Roberto Segala
2° Semestre 2024/2025

Indice

1 Grafi **2**
1.1 Rappresentazione di un grafo 3
1.2 Esplorazione di un grafo 4
1.2.1 Visita in ampiezza (BFS: Breath First Search) 4
1.2.2 Visita in profondità (DFS: Depth First Search) 7
1.3 Componenti fortemente connesse 13
1.3.1 Cammini minimi 16
1.3.2 Cammini minimi con tutte le sorgenti 24

1 Grafi

I grafi permettono di risolvere problemi particolarmente complessi, ma la parte difficile è la conversione di un problema in un grafo. I grafi sono costituiti da nodi e archi:

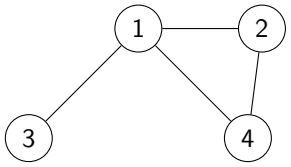


Figura 1: Esempio di grafo

- **Nodi**: rappresentano gli elementi del problema.
- **Archi**: rappresentano le relazioni tra i nodi.

I grafi in cui gli archi hanno un valore (o peso) vengono chiamati **grafi pesati**. Si possono anche aggiungere delle direzioni agli archi, ottenendo così un **grafo orientato**, in cui un arco si può attraversare in un solo verso.

Definizione 1.1 (Cammino)**.** Un **cammino** è una sequenza di nodi per cui esiste un arco tra ogni coppia di nodi adiacenti.

In un cammino, la ripetizione di un nodo rappresenta un **loop** e questo cammino viene detto **cammino ciclico**. (un cammino senza cicli si dice **cammino semplice**)

Il **grado** di un nodo è il numero di archi che incidono sul nodo. Ha senso parlare di grado di un nodo solo quando il grafo non è orientato perchè così ogni arco viene contato una sola volta.

- **Grado entrante**: numero di archi entranti in un nodo.
- **Grado uscente**: numero di archi uscenti da un nodo.

La definizione formale di un grafo è la seguente:

Definizione 1.2. Un grafo è definito come una coppia $G = (V, E)$ dove:

- V è un insieme di nodi.
- E è un insieme di archi:

$$E \subseteq V \times V$$

Dallla figura 1 si ha che:

- $V = \{1, 2, 3, 4\}$.
- $E = \{(1, 3), (3, 1), (1, 1), (1, 4), (4, 1), (1, 2), (2, 4), (4, 2)\}$.

La definizione formale dei concetti precedenti è:

Definizione 1.3. Il **grado uscente** di un nodo v in un grafo orientato $G = (V, E)$ è il numero di archi uscenti da v ($|\dots|$ è la cardinalità di un insieme):

$$\text{grado\_uscente}(v) = |\{u \mid (v, u) \in E\}|$$

Definizione 1.4. Il **grado entrante** di un nodo v in un grafo orientato $G = (V, E)$ è il numero di archi entranti in v:

$$\text{grado\_entrante}(v) = |\{u \mid (u, v) \in E\}|$$

Definizione 1.5. Un cammino è una sequenza di nodi in cui per ogni coppia di nodi consecutivi esiste un arco:

$$\forall i \in \{0 \dots n-1\} \quad (v_i, v_{i+1}) \in E$$

1.1 Rappresentazione di un grafo

Per rappresentare un grafo ci sono due modi:

- **Rappresentazione per liste di adiacenza**: Si crea una lista in cui si rappresentano i nodi e ad ogni nodo si associa la lista di tutti i nodi raggiungibili tramite un arco. Prendiamo in considerazione la figura 1:

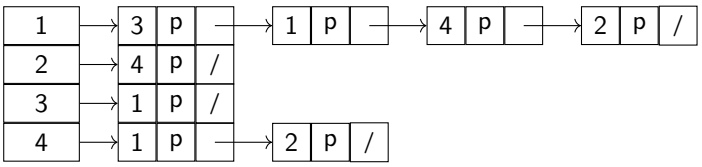


Figura 2: Rappresentazione per liste di adiacenza

  Lo spazio in memoria occupato è $\Theta(|V| + |E|)$.

- **Rappresentazione per matrice di adiacenza**: Si crea una matrice A di dimensione $|V| \times |V|$ in cui $A_{ij} = 1$ se esiste un arco tra i nodi i e j, altrimenti $A_{ij} = 0$. Prendiamo in considerazione la figura 1, dove p è il peso dell'arco:

/	1	2	3	4
1	1	1	1	1
2	0	0	0	1
3	1	0	0	0
4	1	1	0	0

Tabella 1: Rappresentazione per matrice di adiacenza

  Lo spazio in memoria occupato è $\Theta(|V|^2)$.

- Un **grafo trasposto** è un grafo in cui tutti gli archi sono invertiti.
- La **chiusura transitiva di un grafo** è un grafo in cui se esiste un cammino tra due nodi allora esiste un arco diretto tra i due nodi:

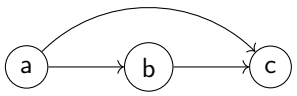


Figura 3: Grafo con chiusura transitiva

- Il **diametro** è il percorso più lungo fra i percorsi minimi

1.2 Esplorazione di un grafo

1.2.1 Visita in ampiezza (BFS: Breath First Search)

La visita in ampiezza (o a ventaglio) è un algoritmo che permette di visitare tutti i nodi di un grafo partendo da un nodo iniziale. L'algoritmo è il seguente:

```
// G e' un grafo composto da un insieme di nodi V e un insieme di
    archi E
// s e' un nodo dell'arco
bfs(G, s)
  for u in G.V
    u.color <- white // non esplorato
    u.distance <- +inf // distanza dal nodo s
    u.parent <- NIL // nodo da cui si arriva a u

  s.color <- gray // scoperto, ma non esplorato
  s.distance <- 0
  s.parent <- NIL
  Q <- {s} // coda FIFO che contiene i nodi scoperti non esplorati

  while Q != empty
    u <- q.head

    for v in G.adj(u) // lista di nodi adiacenti a u
      if v.color == white
        v.color <- gray
        v.distance <- u.distance + 1
        v.parent <- u
        Q.enqueue(v)

  Q.dequeue()
  u.color <- black // esplorato
```

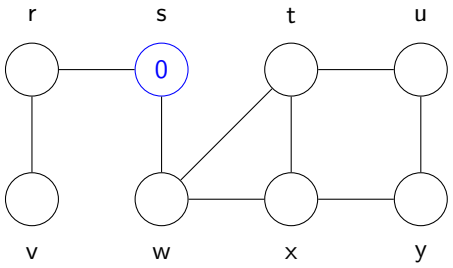
La complessità di questo algoritmo è $O(|V| + |E|)$.

Esempio 1.1. L'algoritmo passo per passo è il seguente, dove i colori rappresentano:

- Nero: non esplorato,
- Blu: scoperto, ma non esplorato,
- Rosso: esplorato,

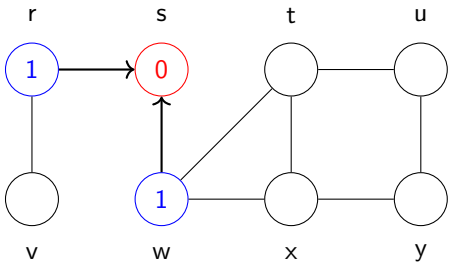
1. Primo passo:

Distanza	0
Coda	s



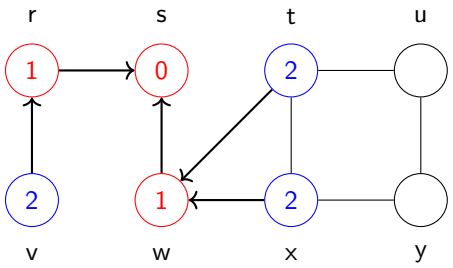
2. Secondo passo:

Distanza	0	1	1
Coda	s	w	r



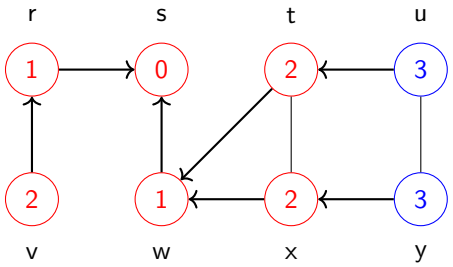
3. Terzo passo:

Distanza	0	1	1	2	2	2
Coda	~~s~~	~~w~~	~~r~~	t	x	v



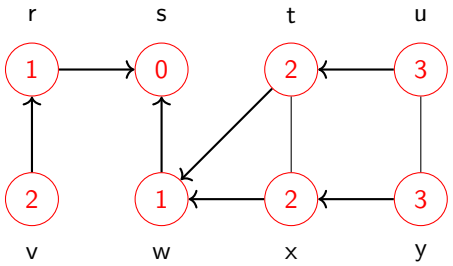
4. Quarto passo:

Distanza	0	1	1	2	2	2	3	3
Coda	~~s~~	~~w~~	~~r~~	~~t~~	~~x~~	~~v~~	u	y



5. Quinto passo:

Distanza	0	1	1	2	2	2	3	3
Coda	~~s~~	~~w~~	~~r~~	~~t~~	~~x~~	~~v~~	~~u~~	~~y~~



Se si vuole trovare il cammino minimo tra due nodi, si parte dal nodo di destinazione e si risale al nodo di partenza seguendo il campo `parent` di ogni nodo.

Questo algoritmo produce un **albero dei cammini di lunghezza minima** radicato in s che ha un cammino minimo per ogni nodo, se tale cammino esiste.

Dimostrazione: Dimostriamo che l'algoritmo BFS produce sempre un albero dei cammini di lunghezza minima:

Sia $\delta(v)$ la lunghezza del cammino minimo da s a v. Dimostrare che

$$\forall v \quad v.\mathtt{distance} = \delta(v)$$

Per dimostrare l'uguaglianza dimostriamo che sia comtemporaneamente maggiore e uguale e minore e uguale:

Lemma 1. $\forall (u, v) \in E \quad \delta(v) \leqslant \delta(u) + 1$

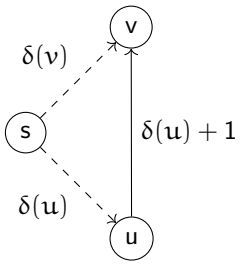


Figura 4: Lemma 1

Lemma 2. $\forall v \quad v.distance \geqslant \delta(v)$ perchè:

$$s.distance = 0 \geqslant 0$$
$$v.distance = u.distance + 1 \geqslant \delta(u) + 1 \geqslant \delta(v)$$

Lemma 3. Nella coda Q ci sono smpre al più 2 valori e la coda è ordinata per distanza crescente. Sia $\langle v_1, \ldots, v_r \rangle$ il contenuto di Q in un qualche istante, allora:

$$v_1.distance \leqslant v_2.distance \leqslant \ldots \leqslant v_r.distance \leqslant v_1.distance + 1$$

Questo è vero per ogni istruzione del programma, è un **invariante**. Ogni istruzione che non modifica Q e non modifica le distanze non modifica l'invariante. L'inizializzazione della coda e la modifica della distanza di un nodo da aggiungere alla coda non modificano l'invariante. L'aggiunta di un nodo alla coda mantiene l'invariante. Quindi tutte le istruzioni mantengono l'invariante.

Teorema 1.1. Sia V_k l'insieme di nodi $v \mid \delta(v) = k$, allora $\forall v \in V_k$ esiste un punto dell'algoritmo in cui:

- v è grigio (scoperto, ma non esplorato).
- k è assegnato a $v.distance$.
- se $v \neq s$ allora $v.parent = u$ per qualche $u \in V_{k-1}$.
- v è inserito in coda

1.2.2 Visita in profondità (DFS: Depth First Search)

L'algoritmo è il seguente:

```
// G e' un grafo composto da un insieme di nodi V e un insieme di
    archi E
dfs(G)
  for u in G.V
    u.color <- white // non esplorato
    u.parent <- NIL

  time <- 0

  for u in G.V
    if u.color == white
      dfs-visit(u)
```

```
// Le variabili della funzione dfs sono accessibili anche da dfs-visit
dfs_visit(u)
  u.color <- gray // scoperto, ma non esplorato
  u.start <- time <- time + 1

  for v in G.adj(u)
    if v.color == white // non esplorato
      v.parent <- u
      dfs-visit(v)

  u.color <- black // esplorato
  u.finish <- time <- time + 1
```

La complessità di questo algoritmo è $O(|V| + |E|)$.

Esempio 1.2. I numeri a sinistra indicano il tempo di inizio della visita e i numeri a destra la fine della visita

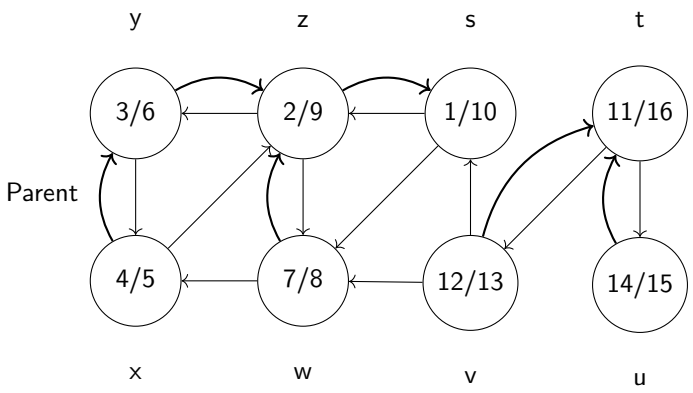


Figura 5: Visita in profondità

Riprendendo l'esempio precedente scriviamo i passaggi nel seguente modo: Il tipo di nodo è denotato dal tipo di parentesi:

- Parentesi aperta: inizio a visitare il nodo
- Parentesi chiusa: fine della visita del nodo

Tempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	(s	(z	(y	(x	x)	y)	(w	w)	z)	s)	(t	(v	v)	(u	u)	t)

Questa espressione è ben parentesizzata:

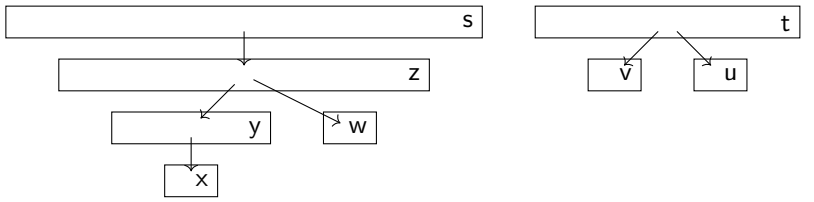


Figura 6: Visualizzazione dell'albero

Gli archi si dividono in:

- **Arco dell'albero** (T): è un arco che collega un nodo a un suo discendente
- **Arco all'indietro** (B): è un arco che collega un nodo a un suo antenato
- **Arco in avanti** (F): è un arco che collega un nodo a un discendente non diretto
- **Arco trasversale** (C): è un arco che collega due nodi non correlati

Quindi nell'esempio precedente abbiamo:

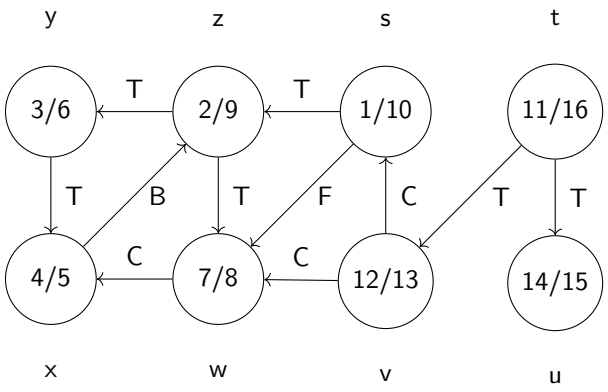


Figura 7: Tipi di archi

nel nostro algoritmo abbiamo che il clolore degli archi distingue i vari tipi:

- **Bianco** (non esplorato): arco trasversale (T)
- **Grigio** (scoperto, ma non esplorato): arco all'indietro (B)
- **Nero** (esplorato): arco dell'albero o arco in avanti (F,C)

Da questo consegue che se ci sono archi all'indietro è presente un ciclo, quindi esiste un algoritmo di complessità $O(|V| + |E|)$ per trovare se un grafo è ciclico e questo algoritmo è il DFS.

Teorema 1.2. Dopo una DFS $\forall u, v$ gli intervalli $[u.start, u.finish]$ sono disgiunti, oppure uno sottointervallo dell'altro

Dimostrazione:

1. Supponiamo che $u.start < v.start$
   (a) Se $u.finish < v.start$ allora i due intervalli sono disgiunti
   (b) Se $u.start < v.finish$ allora v è un sottointervallo di u

   Corollario: In DFS v discende da u se e solo se:

$$u.start < v.start < v.finish < u.finish$$

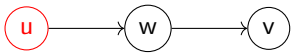
Teorema 1.3. Nella foresta di alberi generata da una DFS, un nodo v è un discendente di un nodo u se e solo se al tempo $u.start$ esiste un cammino da u a v fatto di soli nodi bianchi (non esplorati).

Dimostrazione: Supponiamo che v discende da u, sia w un nodo del cammino da $u \to v$ della foresta:

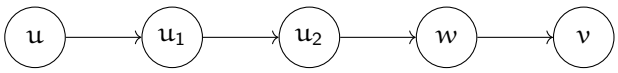
$$u.start < w.start$$

Quindi nel momento in cui u viene scoperto, w è ancora bianco.

- Nero: non esplorato,
- Blu: scoperto, ma non esplorato,
- Rosso: esplorato,



v raggiungibile da u al tempo $u.start$ con cammino di nodi bianchi (non esplorati). Supponiamo per assurdo che v non discende da u



Supponiamo, senza perdita di generalità, che il predecessore di v discende da u. Sia w il predecessore di v, allora w discende da u, quindi:

$$w.finish < u.finish$$

di conseguenza:

$$u.start < w.start < \underbrace{v.start < v.finish}_{\text{Sottointervallo di } u} < w.finish < u.finish$$

Quindi l'intervallo v è un sottointervallo di u contraddicendo l'ipotesi e dimostrando che v discende da u.

Teorema 1.4. Un grafo è aciclico se e solo se DFS **non** trova archi indietro, ed è ciclico se e solo se trova almeno un arco indietro.

Dimostrazione: consideriamo un qualsiasi grafo ciclico. DFS scoprirà un nodo per primo e quel nodo lo chiamiamo u e ci sarà un nodo scoperto per ultimo chiamato v. Se v discende da u allora esiste un cammino da u a v fatto di nodi bianchi (non esplorati) e quindi esiste un ciclo.

Esempio 1.3. Un robot si vuole vestire e deve indossare degli indumenti in un certo ordine. Bisogna trovare un algoritmo che trovi una soluzione tenendo in considerazione tutti i vincoli.

Una rappresentazione più astratta del problema potrebbe essere un grafo orientato in cui i nodi sono gli indumenti e tra due nodi c'è un arco se un indumento deve essere indossato prima dell'altro. Ad esempio:

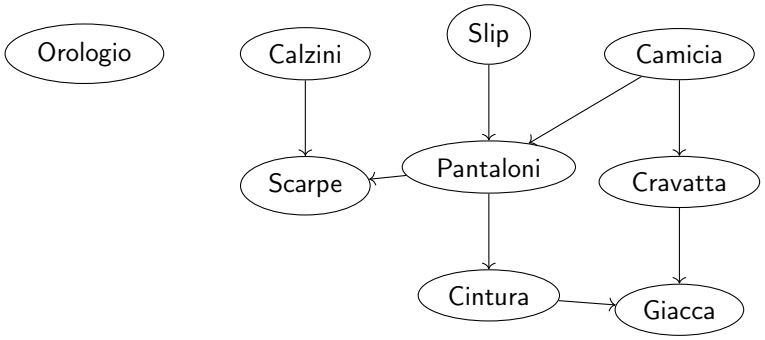


Figura 8: Esempio di rappresentazione del problema

Questo grafo **rappresenta una relazione** di ordinamento **parziale** tra gli indumenti (se non sono presenti cicli). L'obiettivo è di prendere la relazione parziale e renderla totale, ma mantenendola compatibile coi vincoli già imposti. In questo caso ad esempio bisogna imporre altri vincoli e dire che l'orologio va messo prima di qualcos'altro mantenendo l'ordinamento parziale dato all'inizio.

Questo problema si chiama **Ordinamento topologico** e la risoluzione è la seguente:

```
1 // G e' un grafo
2 topological_sorting(G)
3   stack = dfs(G) // DFS ritorna una pila con i nodi in ordine
      decrescente di finish
```

Questo algoritmo ha complessità $O(|V| + |E|)$, quindi si può risolvere un ordinamento topologico in tempo lineare.

L'applicazione dell'algoritmo sul grafo preso in esempio è la seguente considerando che si inizi visitando la camicia (i numeri a sinistra indicano il tempo di inizio della visita e i numeri a destra la fine della visita):

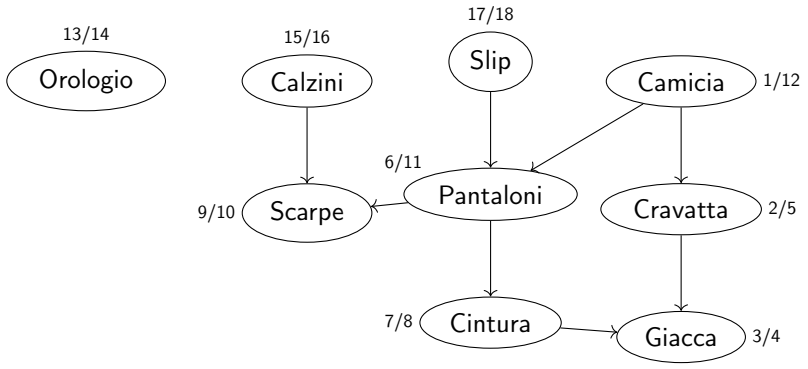


Figura 9: Esempio di rappresentazione del problema

Slip
Calzini
Orologio
Camicia
Pantaloni
Scarpe
Cintura
Cravatta
Giacca

Tabella 2: Stack

Per dire che l'algoritmo funziona bisogna dimostrare che:

$$\forall (u, v) \in E \quad \texttt{v.finish} < \texttt{u.finish}$$

Dimostrazione: Quando (u, v) viene esplorato, allora se:

- v è bianco (non esplorato) allora v è un discendente di u, quindi:

$$\texttt{v.finish} < \texttt{u.finish}$$

- v è grigio (scoperto, ma non esplorato) non è possibile, perchè se v vosse grigio ci sarebbe un grafo ciclico e la soluzione non esiste.
- v è nero (esplorato) allora u non è ancora stato esplorato, quindi:

$$\texttt{v.finish} < \texttt{u.finish}$$

Quindi l'algoritmo funziona.

Teorema 1.5. In un grafo aciclico c'è per forza almeno un nodo che non ha archi entranti.

Dimostrazione: Supponiamo per assurdo che tutti i nodi abbiano almeno un arco entrante. Si può creare una catena di nodi di grandezza $|V|+1$ all'infinito, che prima o poi si ripeterà, creando un ciclo.

Esempio 1.4. Consideriamo le strade d'Italia come grafo, dove i nodi sono le città e gli archi sono le strade a doppio senso. Ci sono parti non raggiungibili (come la Sardegna) e quindi ci sono più grafi separati chiamati **componenti connesse**.

```
cc(G)
  for v in G.V
    make_set(v)
  for (u,v) in G.E
    union(u,v)
```

Questo algoritmo costruisce le componenti connesse di un grafo in tempo $O(|V| + |E|)$. Quindi dati 2 elementi si può trovare in tempo costante se appartengono alla stessa componente connessa e di conseguenza se sono raggiungibili.

Un'alternativa è modificare il DFS in modo che quando si visita un nodo si metta un'etichetta con il numero della componente connessa.

```
dfs_cc(G)
  for v in G.V
    v.color <- white
    v.parent <- NIL

  cc <- 0
  time <- 0

  for u in G.V
    if u.color == white
      cc <- cc + 1
      dfs_visit_cc(u, cc)
```

```
dfs_visit_cc(u, cc)
  u.color <- gray
  u.start <- time <- time + 1

  u.cc <- cc

  for v in G.adj(u)
    if v.color == white
      v.parent <- u
      dfs_visit_cc(v, cc)

  u.color <- black
  u.finish <- time <- time + 1
```

1.3 Componenti fortemente connesse

Definizione 1.6. Dato un grafo orientato $G = (V, E)$ una componente fortemente connessa (SCC) è un sottoinsieme $V' \subseteq V$ tale che:

$$\forall u, v \in V' \quad \exists \text{ cammino da } u \to v \quad \text{e} \quad \exists \text{ cammino da } v \to u$$

Esempio 1.5. Nel seguente esempio ci sono 2 componenti fortemente connesse, una composta da a, b, c, d e l'altra da e, f, g.

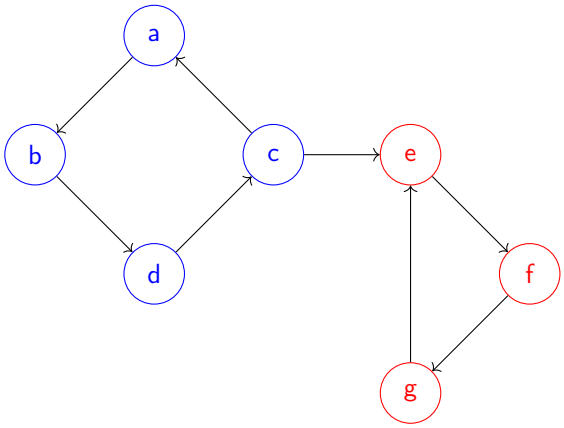


Figura 10: Esempio di grafo con 2 componenti fortemente connesse

L'algoritmo per individuare le componenti fortemente connesse è il seguente:

```
1 scc(G)
2   finish[] = dfs_finish(G) // Usa DFS per trovare i tempi di fine
3   Gt = G.transpose() // Grafo trasposto
4   dfs_from_highest_finish(Gt) // Visita i nodi in ordine
      decrescente di finish
```

Questo algoritmo ha complessità $O(|V| + |E|)$.

Esempio 1.6. L'algoritmo eseguito passo passo su un grafo è il seguente:

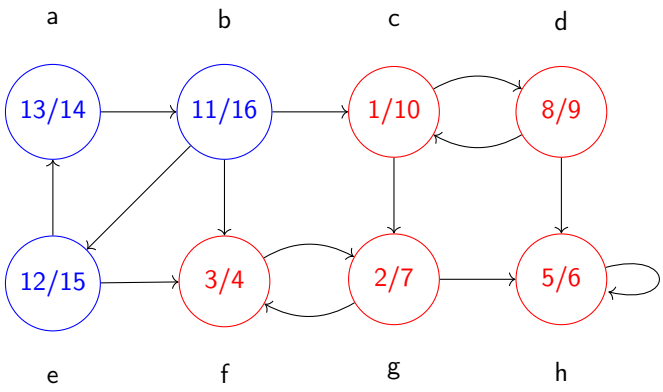


Figura 11: Tipi di archi

Mentre sul grafo trasposto:

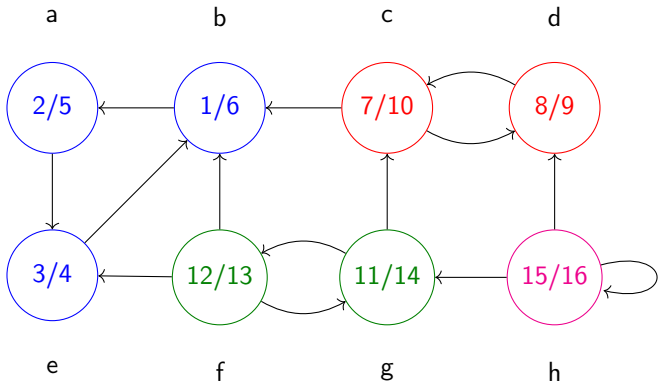


Figura 12: Tipi di archi

Lemma: Dopo la prima DFS, ogni componente fortemente connessa è interamente inclusa in un albero della foresta DFS.

Il **grafo delle componenti fortemente connesse** è un grafo in cui ogni nodo corrisponde a una componente fortemente connessa e gli archi tra due nodi indicano che esiste un arco tra due nodi delle due componenti connesse.

Se c'è un ciclo tra due componenti fortemente connesse, tutti i nodi che fanno parte del ciclo sono in una stessa componente fortemente connessa.

Un grafo delle componenti fortemente connesse contiene un cammino infinito che raggiunge infinite volte uno stato finale quando il cammino arriva in una componente fortemente connessa che contiene almeno 2 nodi o un solo nodo con un autoanello.

1.3.1 Cammini minimi

I ponti di Verona sono tutti rotti e non si possono più riparare, ma bisogna garantire un servizio minimo, cioè permettere ad ogni persona di spostarsi in qualunque zona a piacimento. Quindi tra 2 zone c'è bisogno di un collegamento. Ogni ponte ha un costo di riparazione.

L'obiettivo è quello di trovare il costo minimo per riparare i ponti in modo che ogni zona sia raggiungibile da ogni altra zona.

I nodi saranno le zone e gli archi saranno i ponti. Sappiamo che se il numero di nodi è n, allora il numero di ponti per garantire la connettività è $n-1$. E per induzione se si aggiunge un nodo bisogna aggiungere un arco.

1. Se si vuole minimizzare il costo, bisogna minimizzare il numero di ponti riparati.

2. Presi due nodi esiste un unico modo per andare da un nodo all'altro. Se non fosse così vuol dire che ci sono più ponti che collegano due nodi e quindi il numero di ponti non è minimo. (Quindi il grafo è un albero)

3. Bisogna trovare un albero, tra tutti gli alberi che garantisce la connettività del grafo (chiamati **albero di copertura** o **spanning tree**), con costo minimo. Quindi bisogna trovare il **Minimum Spanning Tree** (MST).

4. Se i pesi negativi, allora la soluzione potrebbe non essere più un albero, però da questa soluzione si può trovare un albero che collega tutti i nodi con costo minimo. Quindi si può risolvere il problema in modo più generale senza imporre i pesi positivi.

5. Si può assegnare ad ogni ponte un indice di gradimento e trovare il MST che massimizza il gradimento. Si può usare un algoritmo che calcola il costo minimo e invertire i segni per trovare il massimo.

L'idea dell'algoritmo è quella di **tagliare** in due un grafo, cioè una bipartizione dell'insieme dei nodi. Se gli estremi di un arco fanno parte dello stesso insieme, allora l'arco è un **arco interno**, altrimenti è un **arco di taglio**. Tra tutti gli archi del taglio, quello con costo minimo è chiamato **arco sicuro**.

Lemma: Dato un albero qualsiasi con un taglio, fra tutti gli MST c'è almeno uno che contiene l'arco sicuro.

Con questo si può ridurre il problema ad uno dello stesso tipo, ma con un nodo in meno, e questo si può fare rappresentando due nodi collegati da un arco sicuro come un nodo singolo e rifare un altro taglio (senza dividere ciò che è stato unito prima).

L'algoritmo è il seguente (algoritmo di Kruskal):

```
// G e' un grafo
// w e' una matrice che associa ad ogni arco il suo peso
kruskal(G, w)
  A <- {}

  for v in G.V
    make_set(v)

  sort(G.E, w.not_descending) // Ordina per peso NON decrescente
```

```
10  for (u,v) in G.E
11    if find_set(u) != find_set(v)
12      A <- union(A, {(u,v)})
13      union(u,v)
14
15  return A
```

Esempio 1.7. L'applicazione di questo algoritmo è la seguente:

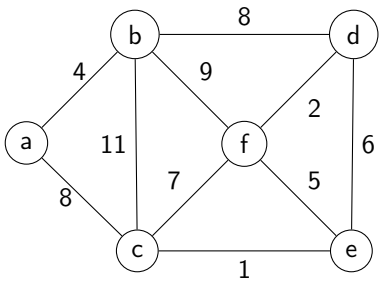
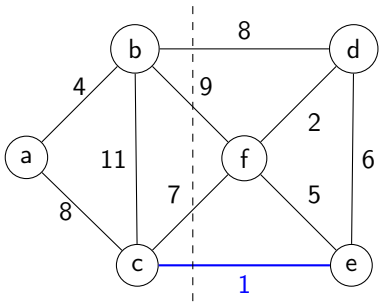
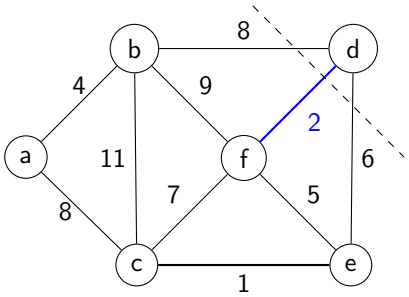


Figura 13: Esempio di grafo

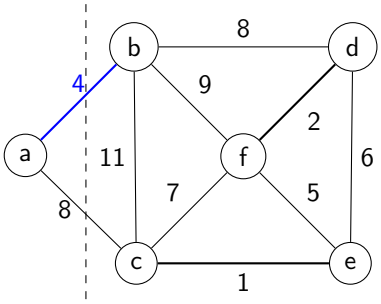
1. Eseguiamo il primo taglio e prendiamo il nodo sicuro:



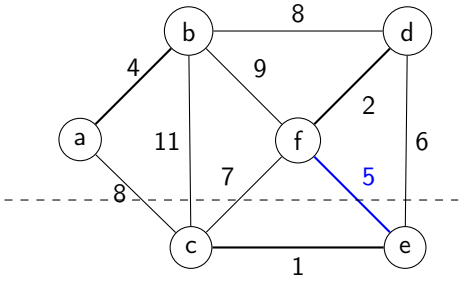
2. Secondo taglio



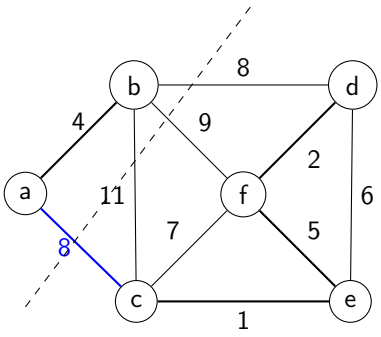
3. Terzo taglio



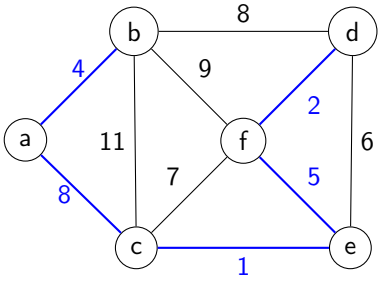
4. Quarto taglio



5. Quinto taglio



6. Alla fine la soluzione è:



Un altro algoritmo per risolvere questo problema è l'algoritmo di Prim:

```
// G e' un grafo
// w e' una matrice che associa ad ogni arco il suo peso
// s e' il nodo di partenza
prim(G,w,s)
  Q <- G.V
  for u in G.V
    u.key <- +inf
    s.key <- 0
    s.parent <- NIL

    while Q != {}
      u <- extract_min(Q)
      for v in G.adj(u)
        if Q.contains(v) and w(u,v) < v.key
          v.parent <- u
          v.key <- w(u,v)
```

La complessità di questo algoritmo è:

- V per la `make_set`
- $E \log E$ per l'ordinamento
- E per la `find_set`
- V union

Il risultato finale è:

$$(V + E)\alpha(\dots) + E \log E$$

Quindi la complessità è:

$$O(E \log E)$$

Esempio 1.8. L'applicazione dell'algoritmo è la seguente

1. Il grafo è il seguente

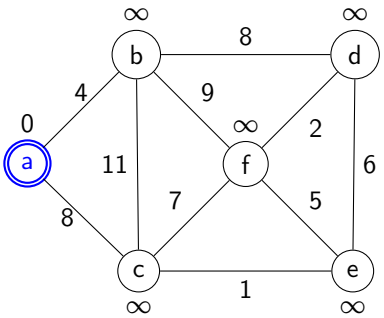


Figura 14: Esempio di grafo

2. Secondo passo

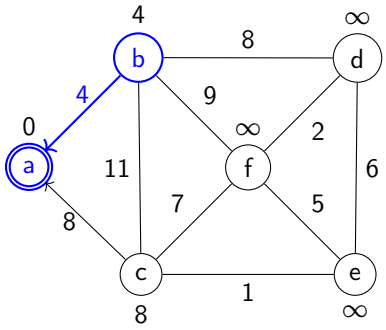


Figura 15: Esempio di grafo

3. Terzo passo

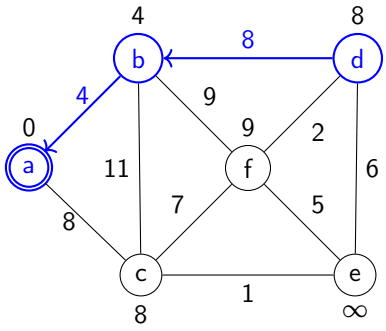


Figura 16: Esempio di grafo

4. Quarto passo

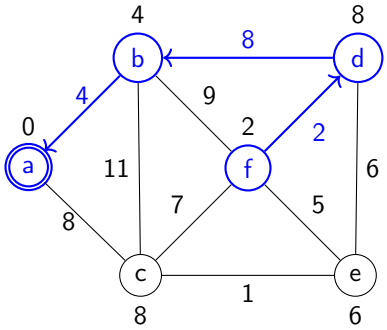


Figura 17: Esempio di grafo

5. Quinto passo

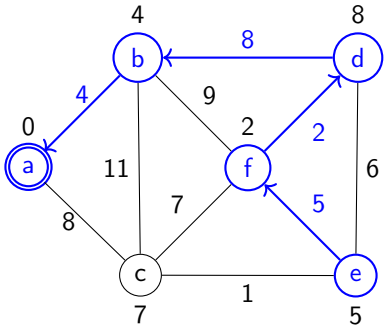


Figura 18: Esempio di grafo

6. Ultimo passo

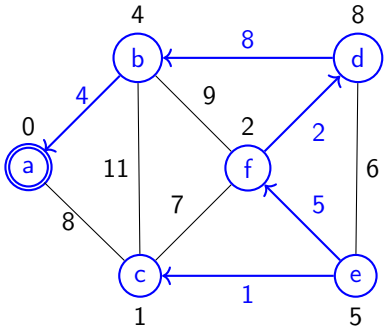


Figura 19: Esempio di grafo

Anche con questo algoritmo abbiamo trovato un MST.

Esiste un algoritmo per risolvere il problema dei cammini minimi che parte dal presupposto che tutti gli archi non devono essere negativi:

```
// G e' un grafo
// s e' il nodo di partenza
init(G, s)
  for v in G.V
    v.distance <- +inf
    v.parent <- NIL
  s.distance <- 0

// u e v sono due nodi
// w rappresenta il tempo di percorrenza tra i 2 nodi
// Prende l'arco u-v che costa w e controlla se
// esiste un cammino minimo migliore di quello da s a v
// passando per u
relax(u, v, w)
  if v.distance > u.distance + w(u,v)
    v.distance <- u.distance + w(u,v)
    v.parent <- u

// G e' un grafo
```

```
// s e' il nodo di partenza
// w indica la distanza tra i nodi
dijkstra(G,s,w)
  init(G,s)
  Q <- G.V
  while Q != {}
    u <- extract_min(Q)
    for v in G.adj(u)
      relax(u,v,w)
```

In questo algoritmo la relax viene effettuata una sola volta per arco perchè ogni nodo viene estratto dalla coda una sola volta e la lista di adiacenza viene esaminata soltanto una volta per nodo.

La complessità di questo algoritmo è

- V volte la init
- V volte la extract min
- E volte la reduce key, quindi a seconda di come viene implementata la coda cambia la complessità dell'algoritmo
  - Liste non ordinate: Complessità V per trovare il nodo minimo
    - V^2 per la extract min, quindi la complessità diventa:

$$V + V^2 + E = O(V^2)$$

  - Liste ordinate: La extract min è costante, la reduce key è lineare
    - V volte un lavoro costante ed E volte un lavoro linere:

$$EV$$

  - Heap: La extract min è logaritmica, la reduce key è costante
    -

$$(V + E) \log V$$

Un altro algoritmo per trovare i cammini minimi è quello di Bellman-Ford:

```
bellman_ford(G, w, s)
  init(G,s)
  for i <- 1 to |G.V| - 1 // --
    for (u,v) in G.E      //  | VE
      relax(u,v,w)        // --

  for (u,v) in G.E        // E
    if v.distance > u.distance + w(u,v)
      return false // ciclo negativo
  return true
```

Se esistono archi rilassabili la soluzione non è stata trovata, se invece non esistono archi rilassabili è stata trovata la soluzione. La complessità di questo algoritmo è:

$$O(VE)$$

Lemma: Dopo i iterazioni di Bellman-Ford, tutti i cammini minimi di lunghezza

al più i sono stati trovati.

Quindi l'algoritmo di Bellman-Ford trova sempre la soluzione se essa esiste.

Consideriamo un grafo che contiene un ciclo negativo.

$$u_1 \to u_2 \to u_3 \to \ldots \to u_n \to u_1$$

Supponiamo per assurdo che Bellman-Ford non dia false, sappiamo che per ognuno dei seguenti archi vale la seguente disequazione:

$$\begin{aligned} u_2.\text{distance} &\leqslant u_1.\text{distance} + w(u_1, u_2) \\ u_3.\text{distance} &\leqslant u_2.\text{distance} + w(u_2, u_3) \\ &\vdots \\ u_n.\text{distance} &\leqslant u_{n-1}.\text{distance} + w(u_{n-1}, u_n) \\ u_1.\text{distance} &\leqslant u_n.\text{distance} + w(u_n, u_1) \end{aligned}$$

Se sommiamo queste cose si ottiene:

$$\sum_{i=1}^{n} u_i.\text{distance} \leqslant \sum_{i=1}^{m} u_i + \text{costo\_ciclico}$$

$$\cancel{\sum_{i=1}^{n} u_i.\text{distance}} \leqslant \cancel{\sum_{i=1}^{m} u_i} + \text{costo\_ciclico}$$

Quindi otteniamo una contraddizione.

Per risolvere il problema dei cammini minimi sui grafi aciclici esiste l'algoritmo Directed Acyclic Graph Shortest Path:

```
dag_sp(G, s, w)
  init(G,s) // V
  topological_sort(G) // V + E
  for u in G.V        // -------+
    for v in G.adj(u) // -| < E | V = V + E
      relax(u,v,w)    // -------+
```

Ogni arco viene rilassato soltanto una volta. La complessità di questo algoritmo è:

$$V + (V + E) + (V + E) = 3V + 2E = O(V + E)$$

Esercizio 1.1. In una gara vengono consegnati dei dischi con un diametro e una morbidezz. L'obiettivo è quello di formare la pila più alta, ma un piatto grande non può essere messo sopra un piatto piccolo e un piatto duro non deve essere messo sopra un piatto morbido.

Il problema si può rappresentare come un grafo in cui i nodi sono i dischi e due nodi a e b sono connessi se a può stare sotto a b. Gli archi sono orientati e il peso è -1.

Non è definito però il nodo di partenza, quindi bisogna ridurre il problema ad uno in cui il nodo di partenza è definito, e di conseguenza bisogna trasformare il grafo. Si possono aggiungere due nodi che sono collegati a tutti gli altri nodi

con archi a peso 0 che funzionano rispettivamente da punto di partenza e da punto di arrivo. Questi nodi sono chiamati **supernodi**.

Se esistessero più piatti uguali, allora essi dovrebbero essere messi tutti insieme, quidi si aggiunge al peso il numero di piatti uguali messi nella pila. In questo modo il cammino di costo minimo risolve il problema.

1.3.2 Cammini minimi con tutte le sorgenti

Il problema dei cammini minimi con tutte le sorgenti è la risoluzione del problema dei cammini minimi per ogni nodo del grafo. Si avrà un vettore di parent e un vettore di distanze che rappresentano la soluzione per ogni sorgente:

$$\pi = \begin{pmatrix} \pi_1 \\ \pi_2 \\ \vdots \\ \pi_n \end{pmatrix} \quad D = \begin{pmatrix} D_1 \\ D_2 \\ \vdots \\ D_n \end{pmatrix}$$

dove π è la matrice dei parent e D è la matrice delle distanze.

La matrice D ha 0 sulla diagonale e infinito altrove:

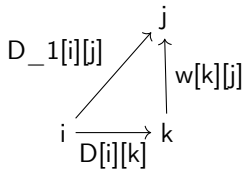
$$\begin{pmatrix} 0 & \infty & \dots & \infty \\ \infty & 0 & \dots & \infty \\ \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \dots & 0 \end{pmatrix}$$

Per calcolare D si usa il seguente algoritmo:

```
// G e' un grafo
// w e' la matrice dei pesi
distance_matrix(G,w)
  D <- init(G)
  for l <- 1 to n-1
    D <- extend_SP(D, w)
```

```
// D e' la matrice delle distanze
// w e' la matrice dei pesi
extend_sp(D, w)
  n <- rows(D)
  for i <- 1 to n // per ogni istanza
    for j <- 1 to n // per ogni sorgente
      D_1[i][j] <- inf
      for k <- 1 to n // per ogni destinazione
        D_1[i][j] <- min(D_1[i][j], D[i][k] + w[k][j])
```

Per ogni arco si tenta di "rilassare" l'arco $i \to j$ passando per k:



L'algoritmo che moltiplica due matrici è:

```
mult(D,w)
  n <- rows(D)
  for i <- 1 to n
    for j <- 1 to n
      D_1[i][j] <- 0
      for k <- 1 to n
        D_1[i][j] <- D_1[i][j] + D[i][k] * w[k][j]
```

si nota che la moltiplicazione è molto simile alla extend_sp, solo che al posto dell'operazione di sommma si fa un minimo e al posto del prodotto si fa una somma. Quindi Bellman Ford è una moltiplicazione tra matrici, ma in un algebra diversa:

$$D \cdot w$$

Nello specifico si sta facendo un prodotto tante volte tenendo in considerazione che dopo aver trovato la soluzione il risultato non varia più:

$$I \cdot w \cdot w \cdots w = w^{n-1} = w^n = w^{n+l} \forall l > 0$$

Questo algoritmo ha una complessità di: $\Theta(n^4)$, però si possono sfruttare tutti quegli algoritmi fatti per migliorare la complessità della moltiplicazione tra matrici.

Per eseguire moltiplicazioni di seguito si può sfruttare il **iterative squaring**:

$$\begin{aligned} w^2 &\leftarrow w \cdot w \\ w^4 &\leftarrow w^2 \cdot w^2 \\ w^8 &\leftarrow w^4 \cdot w^4 \\ &\vdots \end{aligned}$$

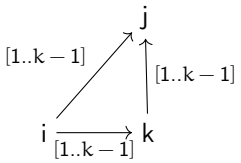
e questo permette di calcolare potenze pari in $\log(n)$ operazioni.

L'algoritmo di Floyd Warshal calcola gli elementi D_{ij}^k dove k è il numero di nodi intermedi che si possono usare per calcolare il cammino minimo tra i e j:

$$D^0 = w$$

La soluzione sarà quindi D^n

```
floyd_warshal(G,w)
  n <- rows(G)
  D^0 <- w
  for k <- 1 to n
    for i <- 1 to n
      for j <- 1 to n
        D^k[i][j] <- min(D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][
    j])
```



Se i pesi degli archi fossero tutti positivi si potrebbe applicare l'algoritmo di Dijkstra ad ogni singola sorgente, questo algoritmo si chiama algoritmo di Johnson:

```
johnson(G,w)
  for i <- 1 to n
    D[i] = dijkstra(G, i, w)
```

e la complessità è:

$$V(V + E)\log(V)$$

se il grafo è connesso si ottiene:

$$VE\log(V)$$

Il problema è che questo algoritmo non si può usare su grafi con archi negativi. Per farlo si deve trasformare il grafo in modo da non avere archi negativi e mantenere il fatto che i cammini minimi non cambiano tra il grafo originale e quello trasformato. Se la trasformazione ha costo maggiore dell'algoritmo da eseguire, non ha senso farla.

I cammini minimi dipendono solo dal punto di partenza e punto di arrivo, quindi Johnson ha definito la seguente funzione:

$$h : V \to R \quad \hat{w}(u,v) = w(u,v) + h(v) - h(u)$$

quindi:

$$\hat{w}(v_0, v_1, \ldots, v_k) = \sum_{i=0}^{k-1} \hat{w}(v_i, v_{i+1}) =$$

$$\begin{aligned}
&w(v_0,v_1) + \cancel{h(v_1)} - h(v_0)\\
&w(v_1,v_2) + \cancel{h(v_2)} - \cancel{h(v_1)}\\
&\vdots\\
&w(v_{k-1},v_k) + \cancel{h(v_k)} - \cancel{h(v_{k-1})}\\
&= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + h(v_k) - h(v_0)
\end{aligned}$$

Questa trasformazione produce nuovi pesi in cui i cammini minimi non cambiano. Bisogna ora trovare la funzione h che produca pesi che non sono negativi. La disequazione di Bellman Ford era:

$$v.distance \leqslant u.distance + w(u,v)$$

e questo diventa:

$$w(u,v) + u.distance - v.distance \geqslant 0$$