Algoritmi

UniVR - Dipartimento di Informatica

Fabio Irimie

2° Semestre 2024/2025

Indice

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:

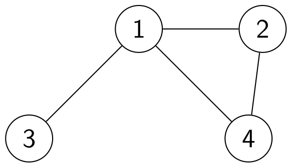


Figura 1: Esempio di grafo

- **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 ($|\ldots|$ 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:

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

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:

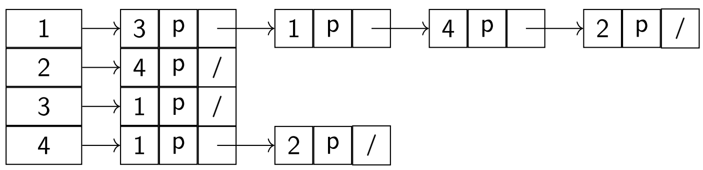


Figura 2: Rappresentazione per liste di adiacenza

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:

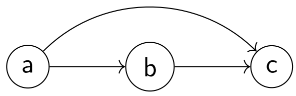


Figura 3: Grafo con chiusura transitiva

- 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
```

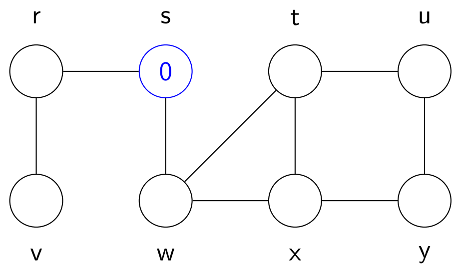
La complessità di questo algoritmo è $O(|V| + |E|)$.

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,

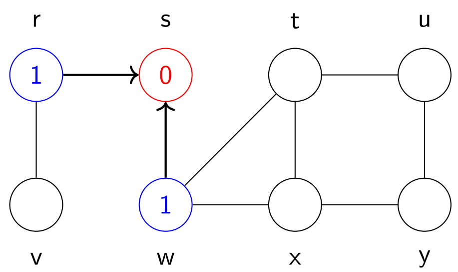
1. Primo passo:

Distanza	0
Coda	s



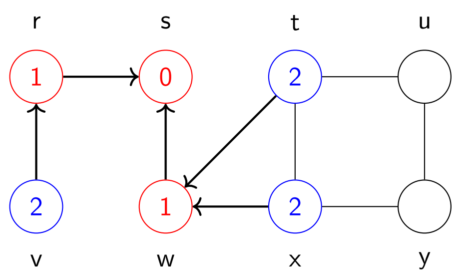
2. Secondo passo:

Distanza	0	1	1
Coda	s	w	r



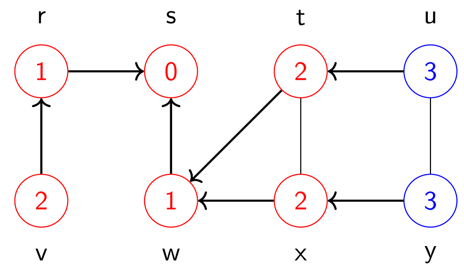
3. Terzo passo:

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



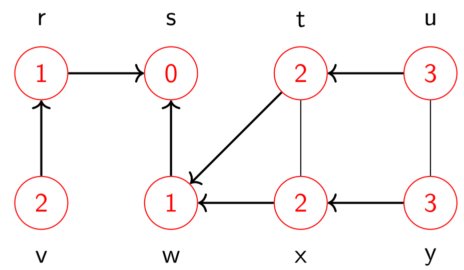
4. Quarto passo:

Distanza	0	1	1	2	2	2	3	3
Coda	$\cancel{s}$	$\cancel{w}$	$\cancel{r}$	$\cancel{t}$	$\cancel{x}$	$\cancel{v}$	u	y



5. Quinto passo:

Distanza	0	1	1	2	2	2	3	3
Coda	$\cancel{s}$	$\cancel{w}$	$\cancel{r}$	$\cancel{t}$	$\cancel{x}$	$\cancel{v}$	$\cancel{u}$	$\cancel{y}$



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

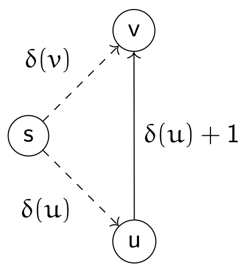


Figura 4: Lemma 1

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

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

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:

$$v_1.\mathrm{distance} \leqslant v_2.\mathrm{distance} \leqslant \ldots \leqslant v_r.\mathrm{distance} \leqslant v_1.\mathrm{distance} + 1$$

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 $v.\mathrm{distance}$.
- se $v \neq s$ allora $v.\mathrm{parent} = u$ 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

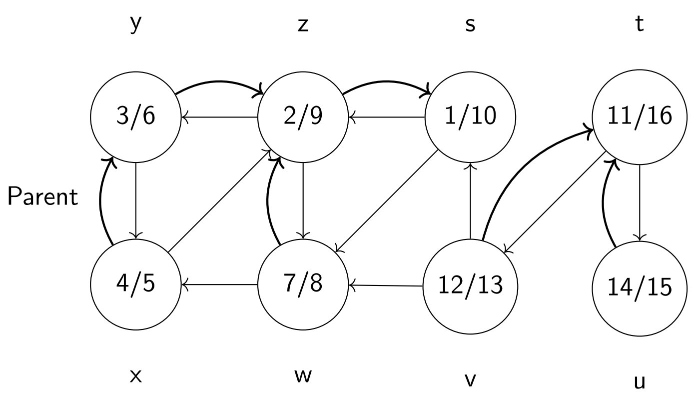


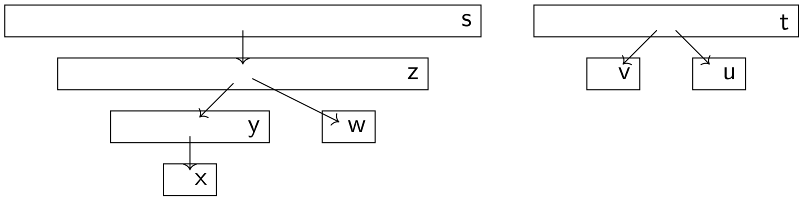
Figura 5: Visita in profondità

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:



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)

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:

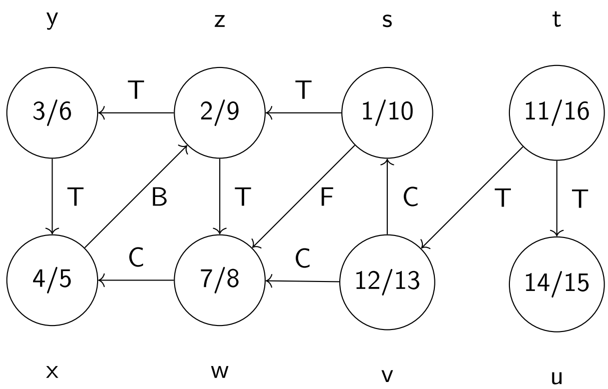


Figura 7: Tipi di archi

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 $[\texttt{u.start}, \texttt{u.finish}]$ sono disgiunti, oppure uno sottointervallo dell'altro

Dimostrazione:

1. Supponiamo che $\texttt{u.start} < \texttt{v.start}$

    (a) Se $\texttt{u.finish} < \texttt{v.start}$ allora i due intervalli sono disgiunti

    (b) Se $\texttt{u.start} < \texttt{v.finish}$ allora v è un sottointervallo di u

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

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

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 $\texttt{u.start}$ 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:

$$\texttt{u.start} < \texttt{w.start}$$

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

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

$$\text{u} \to \text{w} \to \text{v}$$

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

$$u \to u_1 \to u_2 \to w \to v$$

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:

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

di conseguenza:

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

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:

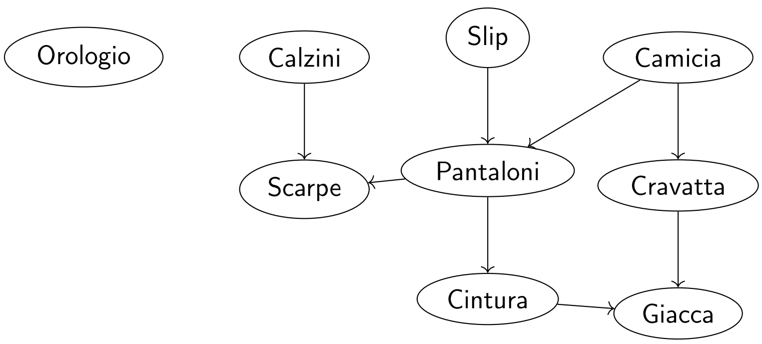


Figura 8: Esempio di rappresentazione del problema

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:

```
// G e' un grafo
topological_sorting(G)
  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 consideramdo 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):

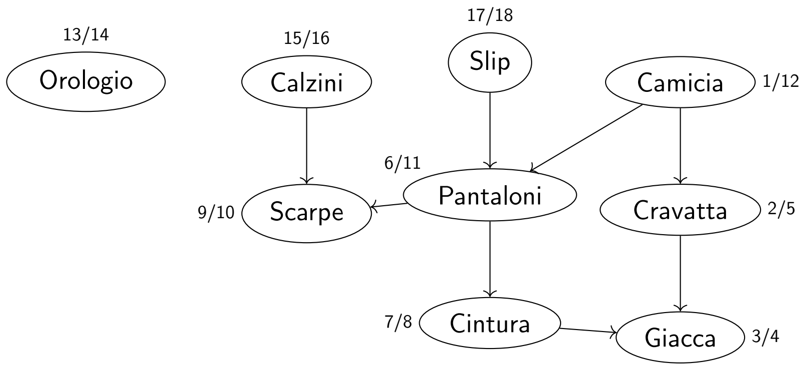


Figura 9: Esempio di rappresentazione del problema

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
```