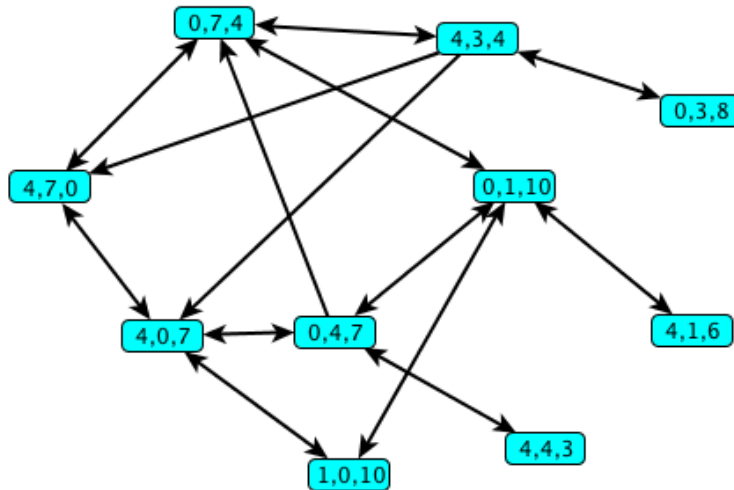


# Progettazione di algoritmi

## Discussione dell'esercizio [acqua]

Come nel caso del problema del 9-puzzle, anche per l'esercizio [acqua] si tratta di trovare un grafo che permetta di modellare il problema. Nel caso del 9-puzzle i nodi del grafo sono tutte le possibili configurazioni (anche quelle non raggiungibili tramite le mosse permesse) e c'è un arco tra un nodo  $u$  e un'altro nodo  $v$  solo se con una mossa si può passare da  $u$  a  $v$ . Così per risolvere il problema dobbiamo trovare un cammino dal nodo  $w$  che rappresenta la configurazione mescolata e il nodo della configurazione iniziale. Vedremo tra poco un algoritmo generale che trova, se esiste, un cammino tra una qualsiasi coppia di nodi.

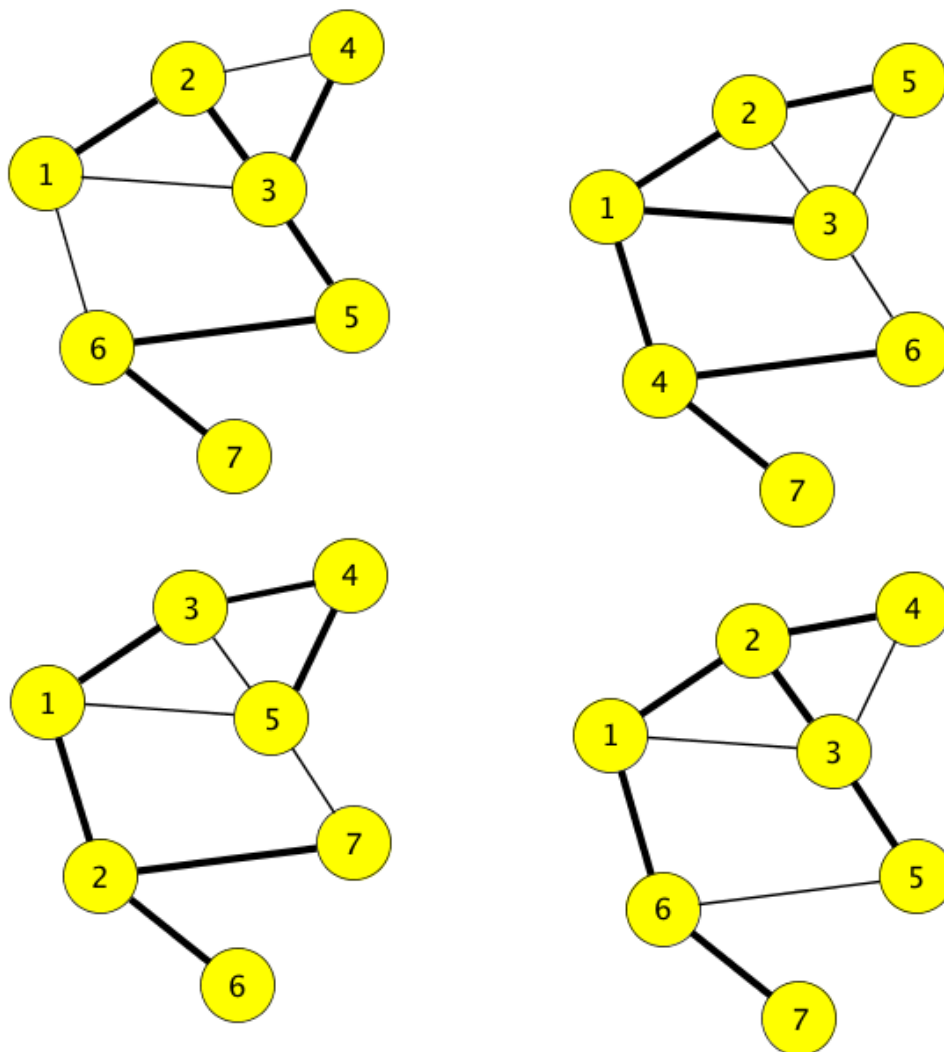
Per l'esercizio [acqua] le configurazioni sono i possibili stati di riempimento dei tre contenitori. Quindi ogni nodo rappresenta una configurazione data da una tripla  $(a, b, c)$  dove  $a$  è il numero di litri d'acqua nel contenitore da 4,  $b$  il numero di litri nel contenitore da 7 e  $c$  il numero di litri in quello da 10. Siccome un contenitore con capienza  $z$  può contenere  $0, 1, 2, \dots, z$  litri d'acqua, esso può assumere  $z + 1$  stati diversi. Quindi, il numero dei nodi è  $5 \times 8 \times 11 = 440$  (questi comprendono anche nodi che dalla configurazione iniziale non possono essere raggiunti). C'è un arco tra due nodi  $u$  e  $v$  se tramite un versamento permesso si passa da  $u$  a  $v$ . La figura qui sotto mostra un frammento del grafo.



La soluzione del problema si riduce così a trovare un cammino dal nodo  $(4, 7, 0)$  a un nodo del tipo  $(2, b, c)$  o  $(a, 2, c)$ .

## Visita di un grafo (DFS)

Per **visita di un grafo** si intende una procedura che partendo da un dato nodo  $u$  permette di visitare tutti i nodi raggiungibili da  $u$  toccando tutti gli archi raggiungibili e arrivando a nuovi nodi o archi solamente a partire da nodi o archi già visitati. Un nodo o arco è *raggiungibile* da  $u$  se esiste un cammino che da  $u$  arriva al nodo o arco. Ovviamente, ci sono tantissimi modi di visitare un grafo. Nella figura qui sotto sono mostrati quattro visite dello stesso grafo a partire dal nodo 1, i nodi sono numerati in ordine di visita e gli archi che hanno permesso di scoprire nuovi nodi sono marcati.



Tra tutte le possibili procedure per visitare un grafo ve ne sono due che sono particolarmente importanti per la loro semplicità e per le proprietà che hanno. La prima che vedremo si chiama **visita in profondità** (in inglese *Depth First Search*, abbreviato DFS). La procedura sulla quale si base la DFS è del tutto simile a quella che si potrebbe seguire nell'esplorazione di un labirinto. Partendo dall'entrata del labirinto prendiamo uno dei corridoi e quando arriviamo ad un bivio prendiamo uno dei corridoi disponibili e così via. Però dobbiamo stare attenti a non girare in circolo e quindi a non percorrere un corridoio che abbiamo già attraversato come se fosse nuovo. Quello che ci occorre è un gesso per marcare i bivi (che sarebbero i nodi del nostro grafo) che abbiamo già visitato. Ma questo non basta dobbiamo anche ricordarci i corridoi che dalla entrata ci hanno portato al punto in cui ci troviamo. Senza questa memoria potremmo non riuscire a esplorare l'intero labirinto perchè tutti i bivi visitati sarebbero indistinguibili e non sapremmo quali fra questi potrebbero avere corridoi che portano verso bivi inesplorati.

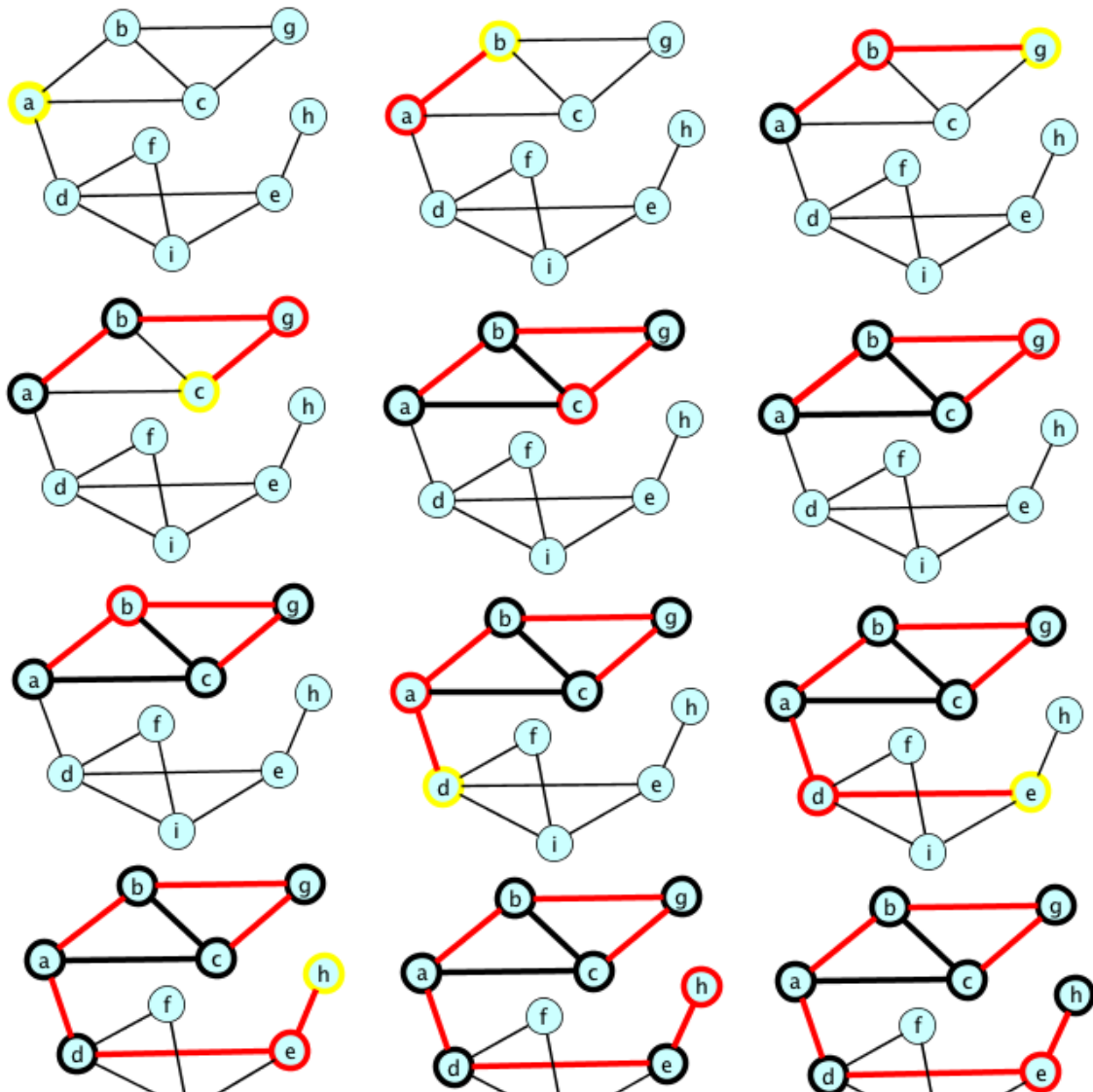
Traducendo questa procedura per un grafo dobbiamo mantenere durante la visita l'insieme dei nodi finora visitati e la sequenza dei nodi, ovvero il cammino, che va dal nodo di partenza al nodo in cui ci troviamo. Per mantenere il cammino possiamo usare una pila (o stack) perchè quando ci troviamo in un nodo in cui tutti gli archi incidenti sono stati attraversati (ovvero tutti gli adiacenti sono stati visitati) dobbiamo tornare al nodo precedente del cammino che equivale a fare una operazione di pop della pila. Quando invece attraversiamo un arco che porta ad un nuovo nodo aggiungiamo quest'ultimo al cammino, cioè effettuiamo un'operazione di push sulla pila.

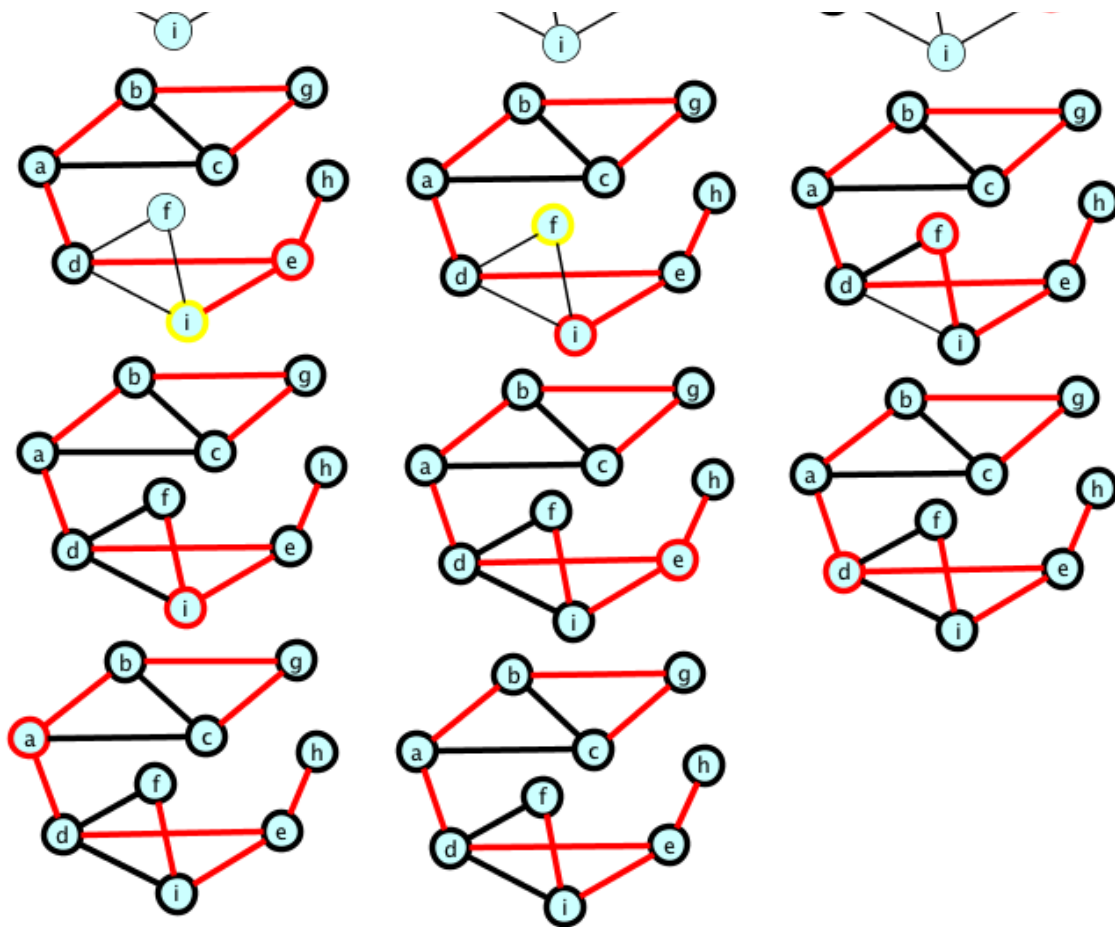
```

DFS(G(V, E): grafo, u: nodo di partenza)
  VIS <- insieme dei nodi visitati, inizialmente vuoto
  S <- stack, inizialmente vuoto
  S.push(u)
  VIS.add(u)
  WHILE S not empty DO
    v <- S.top()
    IF esiste un adiacente w di v con w non in VIS THEN
      VIS.add(w)
      S.push(w)
    ELSE
      S.pop()
  RETURN VIS

```

Alla fine della visita in VIS c'è l'insieme dei nodi che sono stati visitati. Nella figura qui sotto, sono mostrati i passi della DFS (da sinistra verso destra e dall'alto verso il basso). Ogni passo corrisponde ad una iterazione del WHILE : marcato in rosso è il nodo appena estratto dallo stack, in giallo il nuovo nodo visitato (marcati in nero i nodi già visitati), gli archi che hanno portato a scoprire nuovi nodi sono marcati in rosso (marcati in nero gli archi attraversati).





Se il grafo è diretto la DFS attraversa gli archi seguendo la loro orientazione.

## Correttezza ed efficienza della DFS

Ora vogliamo dimostrare che la DFS partendo da un nodo  $u$  visita tutti i nodi raggiungibili da  $u$ . Infatti, supponiamo per assurdo che esista un nodo  $z$  raggiungibile da  $u$  ma che la DFS non visita. Siccome  $z$  è raggiungibile da  $u$ , esiste un cammino  $u_0, u_1, \dots, u_k$  (se il grafo è diretto, il cammino è orientato) con  $u_0 = u$  e  $u_k = z$ . Sia  $u_i$  il primo nodo del cammino che non è stato visitato, chiaramente  $0 < i \leq k$ . Allora,  $u_{i-1}$  è stato visitato e durante la visita prima che il nodo  $u_{i-1}$  sia estratto dallo stack tutti gli adiacenti di  $u_{i-1}$  devono essere stati visitati. Siccome  $u_i$  è adiacente a  $u_{i-1}$ , il nodo  $u_i$  deve essere stato visitato in contraddizione con la nostra ipotesi per assurdo.

Quindi, se il grafo non è diretto, la DFS visita esattamente tutti i nodi della componente connessa del nodo di partenza. Se il grafo è diretto, la DFS visita tutti i nodi raggiungibili da  $u$  tramite cammini orientati che, in generale, sono un soprainsieme della componente fortemente connessa di  $u$ .

Ma quanto è efficiente la DFS? Per valutare la complessità della DFS dobbiamo precisare alcuni dettagli implementativi. Per mantenere l'insieme dei nodi visitati possiamo usare un semplice array `VIS` booleano con un elemento per ogni nodo e inizializzato a `false` e ogni volta che un nuovo  $w$  viene visitato si pone `VIS[w] = true`. Così l'aggiornamento e il test relativo alla visita di un nodo costa tempo costante. Lo stack può essere facilmente implementato in modo che tutte le operazioni `push`, `top` e `pop` abbiano costo costante. Inoltre, assumiamo che il grafo sia rappresentato tramite liste di adiacenza. Così, la scansione degli adiacenti prende tempo costante per ogni adiacente considerato. Ora ad ogni iterazione del `WHILE` o viene visitato un nuovo nodo o è estratto un nodo dallo stack. Siccome ogni nodo è inserito nello stack una sola volta (quando viene visitato), il numero di iterazioni del `WHILE` è al più  $2n$ . Il numero di operazioni non costanti in una iterazione del `WHILE` sono solamente le scansioni degli adiacenti o in altri termini gli attraversamenti degli archi. Ogni arco è attraversato al

più due volte (per grafi diretti una sola volta). Quindi gli attraversamenti degli archi costano complessivamente  $O(m)$ . In totale, la complessità della DFS su un grafo connesso è  $O(n + m)$ . In generale, la complessità è  $O(h + k)$  dove  $h$  è il numero di nodi e  $k$  il numero di archi della componente connessa del nodo di partenza. Perciò la DFS ha complessità ottimale perchè qualsiasi procedura di visita deve necessariamente visitare tutti i nodi e gli archi che sono raggiungibili.

## Versione ricorsiva

La DFS si presta ad essere implementata in modo ricorsivo. Infatti, ogni volta che visita un nuovo nodo  $w$  e come se facesse iniziare una DFS a partire da  $w$ .

```
VIS <- array booleano, inizialmente tutto a false

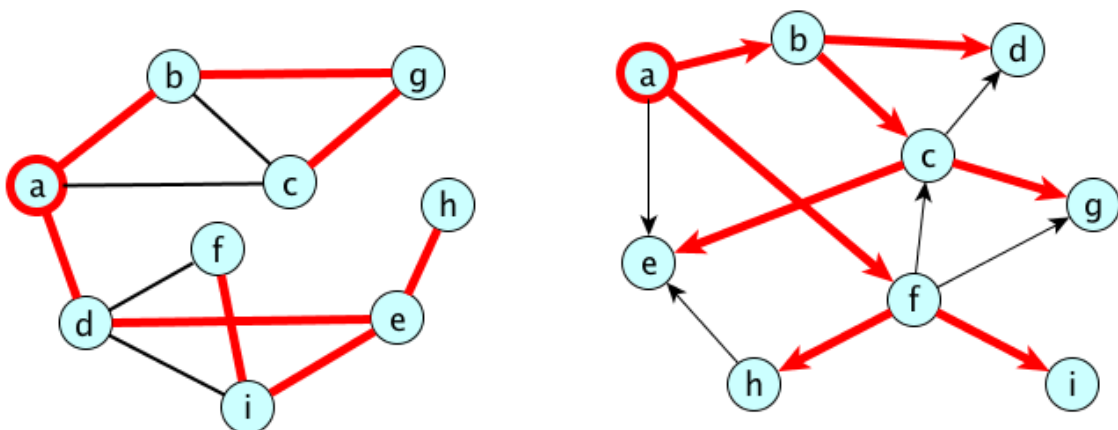
DFS(G: grafo, v: nodo, VIS: array booleano)
  VIS[v] <- true
  FOR ogni w adiacente di v DO
    IF NOT VIS[w] THEN
      DFS(G, w, VIS)
```

Nella versione ricorsiva lo stack è implicitamente gestito proprio dalla ricorsione. Il prezzo di questa semplicità è che lo stack delle chiamate ricorsive ha tipicamente una dimensione piuttosto ridotta e così l'implementazione ricorsiva non è adatta a grafi di grandi dimensioni.

## Albero di visita

Una procedura di visita di un grafo determina un **albero di visita** che è il sottografo formato da tutti i nodi visitati connessi dagli archi che hanno permesso di visitarli. In teoria dei grafi, per **albero** si intende un grafo connesso e aciclico (cioè, senza cicli). Si può dimostrare che un grafo di  $n$  nodi è un albero se è connesso e ha esattamente  $n - 1$  archi. Si può anche dire che un albero è un grafo minimamente connesso, nel senso che ha il numero minimo di archi per renderlo connesso o, equivalentemente, che nessun arco può essere eliminato senza sconnettere il grafo.

Gli alberi di visita dipendono dall'ordine con cui i nodi e gli archi vengono visitati. La DFS produce alberi di visita che hanno particolari proprietà che come avremo modo di vedere risultano molto utili per risolvere efficientemente alcuni problemi fondamentali. Ecco qui sotto gli alberi della DFS (a partire dal nodo marcato in rosso) su due grafi, uno non diretto e l'altro diretto (gli archi degli alberi sono marcati in rosso):



L'albero della DFS è anche determinato dall'ordine con cui sono scanditi gli adiacenti dei nodi visitati. Il nodo di partenza della visita fa sì che gli alberi di visita hanno una radice e quindi restano determinate anche le foglie che sono i nodi  $z$  dell'albero per cui il cammino dalla radice non può essere esteso nell'albero in direzione di  $z$ . Nel caso di grafi diretti l'albero di visita è più propriamente chiamato **arborescenza** per indicare che è un grafo

diretto che se trascuriamo le orientazioni degli archi è un albero e gli archi hanno una orientazione determinata dal nodo radice, cioè sono tutti orientati dalla radice verso le foglie. E le foglie sono esattamente quei nodi dell'arborescenza che non hanno archi uscenti (nell'arborescenza).

## Determinazione delle componenti connesse

Qualsiasi procedura di visita permette di determinare le componenti connesse di un grafo. Infatti, basterà fare una visita a partire da un nodo, questa determinerà la componente connessa del nodo, e poi scegliere, se esiste, un nodo non visitato e fare una visita da quest'ultimo determinando un'altra componente connessa e così via finché tutti i nodi del grafo sono stati visitati. Per tenere traccia delle componenti connesse si può usare un array che ad ogni nodo assegna l'indice della sua componente connessa (gli indici sono determinati dall'ordine con cui sono trovate le componenti):

```
CC(G: grafo)
  cc <- array di n elementi, inizializzato a 0
  nc <- 0
  FOR ogni nodo u di G DO
    IF cc[u] = 0 THEN
      nc <- nc + 1
      DFSCC(G, u, cc, nc)
  RETURN cc

DFSCC(G: grafo, u: nodo, cc: array, nc: intero)
  cc[u] <- nc
  FOR ogni w adiacente di v DO
    IF cc[w] = 0 THEN
      DFSCC(G, w, cc, nc)
```

Quindi  $CC(G)$  ritorna un array che per ogni nodo di  $G$  dà l'indice della sua componente connessa. Per grafi non diretti ciò è corretto ed è anche efficiente in quanto la complessità è ancora una volta  $O(n + m)$ . Per grafi diretti l'algoritmo non determina in generale le componenti fortemente connesse. Come vedremo la DFS può essere usata anche per determinare in modo efficiente le componenti fortemente connesse ma l'algoritmo è più complicato.

## Un problema

Consideriamo ora un problema. Dobbiamo preparare il calendario degli esami per un corso di laurea. Conosciamo per ognuno dei  $k$  studenti quali tra gli  $n$  esami vuole sostenere. Ci sono solamente due settimane a disposizione e ognuno degli  $n$  esami deve essere messo in una sola delle due settimane. Vogliamo, se possibile, trovare un calendario che eviti che qualche studente debba sostenere due o più esami nella stessa settimana. Come possiamo risolvere il problema?

L'idea è di modellare il problema con un opportuno grafo. Quello che dobbiamo rappresentare tramite il grafo sono i possibili conflitti tra coppie di esami dovuti a studenti che vogliono sostenere entrambi gli esami. Consideriamo quindi un grafo (non diretto) i cui nodi sono gli esami e tra due esami  $u$  e  $v$  mettiamo un arco se c'è almeno uno studente che vuole sostenere sia  $u$  che  $v$ . Così una soluzione del problema può essere formulata come l'assegnazione ad ogni nodo-esame di una delle due settimane in modo che due qualsiasi nodi connessi da un arco devono avere assegnate settimane diverse. Questo è un caso particolare di un problema di *colorazione di un grafo*. In generale, dato un grafo si vogliono colorare i nodi usando il numero minimo di colori che garantiscono che due qualsiasi nodi adiacenti hanno colori diversi. Per il nostro problema i colori sono le settimane e quindi sono solamente due. Quindi vogliono sapere se esiste e in tal caso trovare una cosiddetta *2-colorazione* del grafo che abbiamo definito. Un altro modo di vedere il problema è di vedere l'assegnamento dei due colori ai nodi come la partizione del grafo in due insiemi (disgiunti) tali che non ci sono archi tra nodi appartenenti alla stessa parte. Un grafo che può essere così partizionato si dice *bipartito*.

Per determinare se un grafo è 2-colorabile (o equivalentemente se è bipartito) ed in tal caso trovare la 2-colorazione, si può usare una qualsiasi procedura di visita. È sufficiente mantenere un array con elementi i nodi del grafo e inizializzato a 0 eccetto il nodo di partenza della visita a cui è assegnato il colore 1, ogni volta che un

nuovo nodo  $w$  viene visitato gli viene assegnato il colore differente a quello assegnato al nodo  $v$  che ha portato a visitare  $w$ , e infine ogni volta che si attraversa un arco che arriva in un nodo già visitato si controlla che abbia colore diverso da quello del nodo di partenza dell'arco. Se tutti i controlli sugli archi attraversati dalla visita sono soddisfatti la colorazione ottenuta è una 2-colorazione. Questo perchè tutti gli archi sono stati attraversati e sia che essi abbiano portato a visitare nuovi nodi che a toccare nodi già visitati i colori dei nodi incidenti è sempre diverso. Se invece uno dei controlli fallisce, il grafo non è 2-colorabile. La ragione sta nel fatto che l'assegnazione dei colori ai nodi effettuata durante la visita è sempre obbligata (eccetto quella del primo nodo).

Il seguente algoritmo usa la DFS per cercare di 2-colorare un grafo non diretto. Per semplicità assumiamo che il grafo sia connesso:

```
TWOCOL(G:grafo)
  col <- array, inizializzato a 0
  col[u] <- 1
  IF NOT DFS2C(G, u, col) THEN
    RETURN NULL
  RETURN col

DFS2C(G: grafo, u:nodo, col:array)
  FOR ogni adiacente w di u DO
    IF col[w] = col[u] THEN
      RETURN FALSE
    ELSE IF col[w] = 0 THEN
      col[w] <- (2 IF col[u] = 1 ELSE 1)
      IF NOT DFS2C(G, w, col) THEN
        RETURN FALSE
  RETURN TRUE
```

Usando la DFS si può risolvere il problema della 2-colorazione (o bipartizione) in modo molto efficiente, cioè, in tempo  $O(n + m)$ .

In generale, il problema della colorazione è molto più difficile. Determinare se un grafo è 3-colorabile è già un problema per cui non si conoscono algoritmi efficienti. Gli algoritmi migliori hanno complessità esponenziale nelle dimensioni del grafo.

## Esercizio [snodi critici]

Consideriamo una rete ferroviaria formata da lunghe tratte di binari senza interruzioni connesse tra di loro da snodi. Ogni tratta è percorribile in entrambe le direzioni. Uno snodo può connettere due o più tratte in modo tale che un treno può passare da una tratta a una qualsiasi altra dello snodo. Due snodi  $A$  e  $B$  sono collegati se un treno partendo da  $A$  può arrivare allo snodo  $B$  percorrendo gli snodi e le tratte delle rete. Relativamente a due snodi  $A$  e  $B$  collegati, uno snodo  $C$  è detto *critico* se un treno per andare da  $A$  a  $B$  deve necessariamente passare per  $C$ . Data la rete ferroviaria, come possiamo trovare gli eventuali snodi critici per due dati snodi collegati  $A$  e  $B$ ?