Inf Gen II

INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni
Università di Trento

Marco Roveri
roveri@irst.itc.it

Grafi

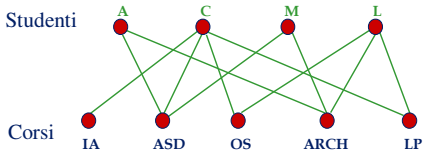
Parte del materiale qui presentato è presentata per gentile concessione di M. Benerecetti

Inf Gen II

Cosa è un *grafo*

- Esempio:

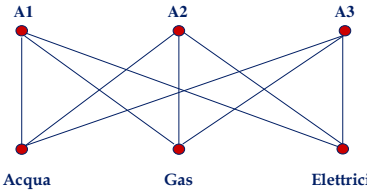
Studenti	Corsi
Marco	ASD, ARCH
Carla	IA, ASD, OS, LP
Andrea	ASD, ARCH
Laura	OS, ARCH, LP



Inf Gen II

Rappresentazioni a grafi di problemi

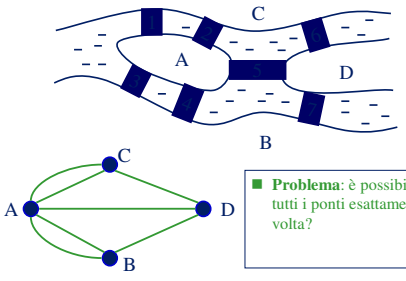
- **Problema**:
  - Dobbiamo connettere tre abitazioni A1, A2, A3 tramite tubature per fornire ad ognuna Acqua, Gas ed Elettricità.
  - Le tubature devono essere posizionate tutte alla stessa profondità.
  - È possibile offrire la fornitura alle tre abitazioni senza fare incrociare le tubature?



Inf Gen II

Rappresentazioni a grafi di problemi

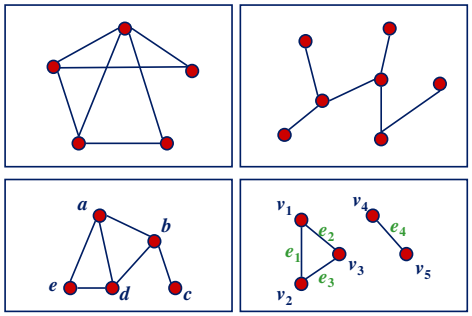
- Ponti di Königsburg



- **Problema**: è possibile attraversare tutti i ponti esattamente una sola volta?

Inf Gen II

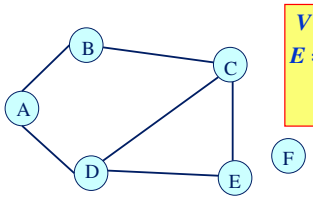
Esempi di grafi



Inf Gen II

Definizione di grafo

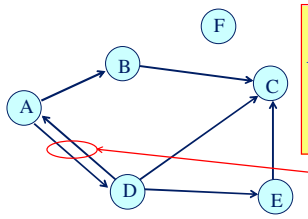
- Un ***grafo G*** è una coppia di elementi (***V***, ***E***) dove:
  - ***V*** è un insieme detto ***insieme dei vertici***
  - ***E*** è un insieme detto ***insieme degli archi*** (***edge***),
    - ***(v,w)*** ∈ ***E*** se e solo se ***v*** ∈ ***V*** e ***w*** ∈ ***V***.



$V = \{A, B, C, D, E, F\}$
$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$

Grafi orientati

- Un ***grafo orientato G*** è una coppia di elementi (***V***, ***E***) dove:
  - ***E*** è una ***relazione binaria*** tra vertici detta ***insieme degli archi.***

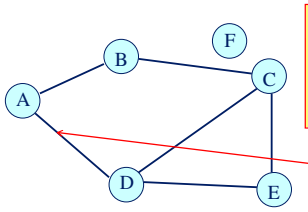


V = {A, B, C, D, E, F}
E ={(A,B), (A,D), (B,C), (D,C), (E,C),(D,E), (D,A)}

(A,D) *e* (D,A) denotano due archi diversi

Grafi non orientati

- Un ***grafo non orientato G*** è una coppia di elementi (***V***, ***E***) dove:
  - ***E*** è un ***insieme non ordinato*** di coppie di vertici detta ***insieme degli archi.***

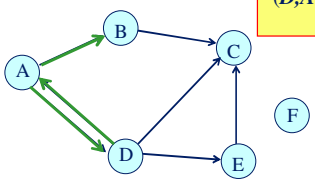


V = {A, B, C, D, E, F}
E = {(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)}

(A,D) *e* (D,A) denotano lo stesso arco

Definizioni sui grafi

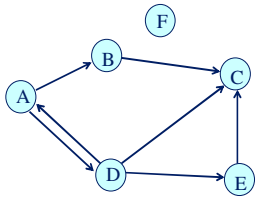
- In un grafo orientato, un arco (*w*,*v*) si dice *incidente* da *w* in *v*



- (***A***,***B***) è ***incidente*** da ***A*** a ***B***
- (***A***,***D***) è ***incidente*** da ***A*** a ***D***
- (***D***,***A***) è ***incidente*** da ***D*** a ***A***

Definizioni sui grafi

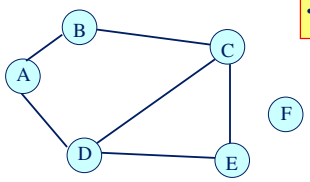
- Un vertice *w* si dice adiacente a *v* se e solo se *(v, w)* ∈ *E*.
- In un grafo non orientato la relazione di adiacenza tra vertici è simmetrica.



- ***B*** è ***adiacente*** ad ***A***
- ***C*** è ***adiacente*** a ***B*** e a ***D***
- ***A*** è ***adiacente*** a ***D*** e vice versa
- ***B NON*** è ***adiacente*** a ***D NÉ*** a ***C***
- ***F NON*** è ***adiacente*** ad alcun vertice

Definizioni sui grafi

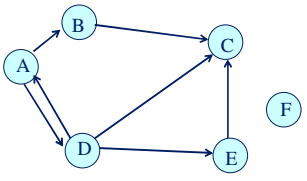
- In un grafo non orientato il *grado di un vertice* è il numero di archi che da esso si dipartono



- ***A***, ***B*** ed ***E*** hanno ***grado 2***
- ***C*** e ***D*** hanno ***grado 3***
- ***F*** ha ***grado 0***

Definizioni sui grafi

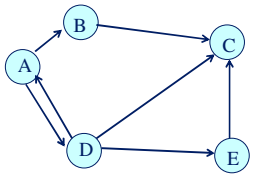
- In un grafo orientato il *grado entrante* (*uscente*) di un vertice è il numero di archi incidenti in (da) esso



- ***A*** ha ***grado uscente 2*** e ***grado entrante 1***
- ***B*** ha ***grado uscente 1*** e ***grado entrante 1***
- ***C*** ha ***grado uscente 0*** e ***grado entrante 3***
- ***D*** ha ***grado uscente 3*** e ***grado entrante 1***

Definizioni sui grafi

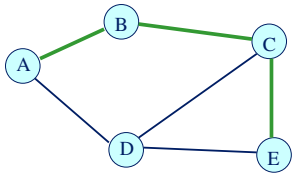
- In un grafo orientato il *grado* di un vertice è la somma del suo grado entrante e del suo grado uscente



- ***A*** e ***C*** hanno ***grado* 3**
- ***B*** ha ***grado* 2**
- ***D*** ha ***grado* 4**

Definizioni sui grafi

- Un percorso si dice *semplice* se tutti i suoi vertici sono distinti (compaiono una sola volta nella sequenza), eccetto al più il primo e l'ultimo che possono essere lo stesso.

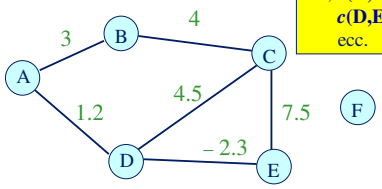


- il percorso <A, B, C, E> è semplice.
- così come anche <A, B, C, E, D, A>.
- il percorso <A, B, C, E, D, C > non è semplice, poiché C è ripetuto.

Definizioni sui grafi

In alcuni casi ogni arco ha un ***peso*** (o ***costo***) associato.

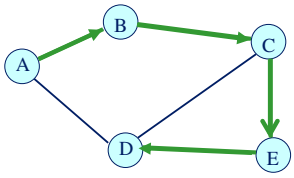
Il costo può essere determinato tramite una funzione di costo, $c: E \rightarrow R$, dove R è l'insieme dei numeri reali (o interi).



Es., ***c*(A, B) = 3,**
***c*(D,E) = – 2.3,**
ecc.

Definizioni sui grafi

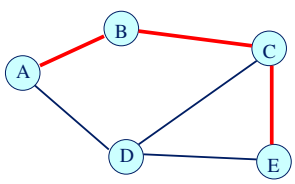
- Se esiste un percorso *p* tra i vertici *v* e *w*, si dice che *w* è raggiungibile da *v* tramite *p*.



- A è raggiungibile da D e vice versa
- A è raggiungibile da D

Definizioni sui grafi

- Sia $G = (V, E)$ un grafo.
- Un *percorso* nel grafo è una sequenza di vertici $\langle w_1, w_2, \ldots, w_n \rangle$ tale che $(w_i, w_{i+1}) \in E$ per $1 \leq i \leq n-1$.



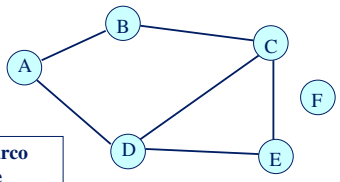
Es.,
<A, B, C, E>
è un percorso nel grafo

Definizioni sui grafi

- Se *G* è un grafo non orientato, diciamo che *G* è *connesso* se esiste un *percorso* da *ogni vertice* ad *ogni altro vertice*.

Un grafo non orientato *non connesso* si dice *sconnesso*.

Questo grafo non orientato *non è connesso*.

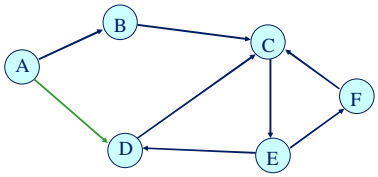


Se aggiungiamo un arco (E,F) allora il grafo è *connesso*.

Definizioni sui grafi

- Se G è un grafo orientato, diciamo che G è *fortemente connesso* se esiste un percorso da ogni vertice ad ogni altro vertice.

Questo grafo orientato **non è *fortemente connesso*; ad es., *non esiste percorso da* D *a* A**.



Definizioni sui grafi

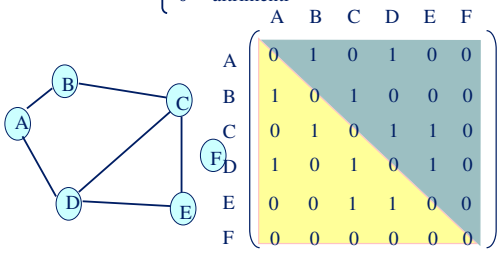
- Un ciclo in un grafo è un percorso $<w_1, w_2, \ldots, w_n>$ di lunghezza almeno 1, tale che $w_1 = w_n$.
- Un grafo senza cicli è detto *aciclico*.
- Un grafo *completo* è un grafo che ha un arco tra ogni coppia di vertici.

Rappresentazione di grafi

Rappresentazione con ***matrice di adiacenza***:

$$M(v, w) = \begin{cases} 1 & \text{se } (v, w) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Spazio: $|V|^2$

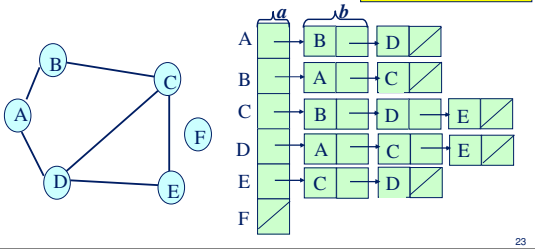


Rappresentazione di grafi

Rappresentazione con ***liste di adiacenza***:

L*(*v*)** = lista di ***w, tale che **(*v*, *w*) ∈ *E*,** per ***v* ∈ *V***

Spazio: $a\,|V| + 2\,b\,|E|$



Rappresentazione di grafi

- Per la rappresentazione di grafi vengono comunemente utilizzate due tipologie distinte:
  - Rappresentazione mediante *matrice delle adiacenze*.
  - Rappresentazione mediante *liste di adiacenze*.

Rappresentazione di grafi

- Matrice di adiacenza
  - Spazio richiesto $O(|V|^2)$.
  - Verificare se i vertici *u* e *v* sono adiacenti richiede tempo O(1).
  - Molti 0 nel caso di *grafi sparsi*.
- Liste di adiacenza
  - Spazio richiesto $O(|E|+|V|)$.
  - Verificare se i vertici *u* e *v* sono adiacenti richiede tempo $O(|V|)$.

Implementazione di grafi

- Mediante matrice di adiacenze.

```
struct Grafo {
   int V; // numero vertici
   int  **adj;
   Grafo(int n) {
     V = n;
     adj = new int * [V];
     for(int i = 0; i < V; i++) {
       adj[i] = new int [V];
       // Inizializzazione grafo con tutti vertici isolati
       for(int j = 0; j < V; j++) adj[i][j] = 0;
     }
   }
};
```


Implementazione di grafi

```
// ritorna numero vertici del grafo
int getnumvertex(Grafo * g) {
  return g->V;
}

void addedge(Grafo * g, int i, int j) {
  g->adj[i][j] = 1;
  // se il grafo non e' orientato
  // g->adj[j][i] = 1;
}

void deledge(Grafo * g, int i, int j) {
  g->adj[i][j] = 0;
  // se il grafo non e' orientato
  // g->adj[j][i] = 0;
}
```

```
bool connected(Grafo * g, int i, int j) {
  return (g->adj[i][j] == 1);
}

void printadj(Grafo * g) {
  int V = getnumvertex(g);
  for(int i = 0; i  < V; i++) {
    for(int j = 0; j  < V; j++)
      if (connected(g, i, j) == true)
            cout << " 1 " ;
      else
            cout << " 0 " ;
    cout << endl;
  }
}
```

Nota: mancano controlli che *i* e *j* siano minori stretti di *V*

Implementazione di grafi

```
struct Node {
  int name;
  node * next;
  Node(int nm, Node * ne) {
    name = nm;
    next = ne;
  };
};

Node * node_empty() {
  return NULL;
}

bool node_isempty(Node * n) {
  return n == NULL;
}

int node_getname(Node * n) {
  return n->name;
}
```

```
Node * node_getnext(Node * n) {
  return n->next;
}

void node_setnext(Node * n, Node * n1) {
  n->next = n1;
}

Node * node_remove(Node * & n, int j) {
  if (node_isempty(n)) return n;
  if (node_getname(n) == j) {
    Node * d = n;
    n = node_getnext(n);
    delete d;
  }
  else {
    node_setnext(n, node_remove(node_getnext(n), j));
  }
  return n;
}
```


Implementazione di grafi

```
struct Grafo {
  int V; // numero vertici
  node  * * adj;
  Grafo(int n) {
    V = n;
    adj = new node * [V];
    for(int i = 0; i < V; i++) {
      adj[i] = node_empty();
    }
  };
};

int getnumvertex(Grafo * g) {
  return g->V;
}
```

```
void addedge(Grafo * g, int i, int j) {
  Node * n0 = new Node(j, g->adj[i]);
  g->adj[i] = n0;
  // Se il grafo non e' orientato
  // Node * n1 = new Node(i, g->adj[j]);
  // g->adj[j] = n1;
}

void deledge(Grafo * g, int i, int j) {
  g->adj[i] = node_remove(g->adj[i], j);
  // Se il grafo non e' orientato
  // g->adj[j] = node_remove(g->adj[j], i);
}
```

Nota: mancano controlli che *i* e *j* siano minori stretti di *V*

Implementazione di grafi

```
bool connected(Grafo * g, int i, int j) {
  bool found = false;
  for(Node * adj = g->adj[i];
      !node_isempty(adj) && !found;
      adj = node_getnext(adj)) {
    if (node_getname(adj) == j) found = true;
  }
  return found;
}

void printadj(Grafo * g) {
  int V = getnumvertex(g);
  for(int i = 0; i  < V; i++) {
    for(int j = 0; j  < V; j++)
      if (connected(g, i, j) == true) cout << " 1 " ;
      else  cout << " 0 " ;
    cout << endl;
  }
}
```

Nota: mancano controlli che *i* e *j* siano minori stretti di *V*

Visita/Attraversamento di grafi

- Gli algoritmi di visita o attraversamento di grafi sono una generalizzazione degli algoritmi di attraversamento degli alberi.
- Esistono diverse tecniche di attraversamento.
- L'algoritmo forse più importante è l'algoritmo di attraversamento in *profondità*, meglio noto con il nome di *depth-first search algorithm (DFS)*.
- L'algoritmo DFS costruisce la base di partenza per molti algoritmi fondamentali di elaborazione di grafi.

Algoritmo DFS

- È una variazione della visita in preordine di un albero binario.
- Si parte da un vertice *s:*
  - Si processa il vertice *s;*
  - Ricorsivamente si processano tutti i nodi adiacenti ad *s.*
- Bisogna evitare di visitare vertici già visitati:
  - Ovvero bisogna evitare i cicli.
  - Questo lo si ottiene *marcando* (*colorando*) il grafo.

Algoritmo DFS: intuizioni

I passi dell'algoritmo DFS
1. Si sceglie un vertice non visitato *s*
2. Si sceglie un nodo non visitato adiacente ad *s*
3. Da *s* si attraversa quindi un percorso di vertici adiacenti (visitandoli) finché possibile:
   - Cioè fintanto che non si incontra un nodo già visitato
4. Appena si resta "bloccati" (tutti gli archi da un vertice sono stati visitati), si torna indietro (*backtracking*) di un passo (vertice) nel percorso visitato (aggiornando il vertice *s* al vertice corrente).
5. Goto 2.

Algoritmo DFS: implementazione

```
void DFS(Grafo * g, int s, void visit(Grafo *, int)) {
  // alloco un array di boolean per tenere traccia se
  // ho gia' visitato il vertice
  bool * visited = new bool [getnumvertex(g)];

  // inizializzazione dell'array visited
  for(int i = 0; i  < getnumvertex(g); i++) visited[i] = false;

  // chiamata alla routine ricorsiva
  DFS_Recur(g, s, visit, visited);

  // deallocazione array visited
  delete [] visited;
}
```


Algoritmo DFS: implementazione

```
void DFS_Recur(Grafo * g, int s, void visit(Grafo *, int),
                          bool visited[]) {
  // invoco della procedura che processa il vertice s
  visit(g, s);
  // tengo traccia del fatto che ho visitato il nodo s;
  visited[s] = true;

  // ricorsione sui nodi adiacenti ad s
  for(int i = 0; i < getnumvertex(g); i++) {
    if ((connected(g, s, i)  && ! visited[i])
        DFS_Recur(g, i, visit, visited);
}
```

Implementazione generica **non** sfrutta implementazione del dato astratto grafo (i.e. che usa o meno liste di adiacenza o matrice di adiacenza).

Algoritmo DFS: implementazione

```
void DFS(Grafo *g, int s, void visit(Grafo *, int)) {
 bool * visited = new bool [getnumvertex(g)];
 for(int i = 0; i  < getnumvertex(g); i++) visited[i] = false;

// utilizziamo uno stack inizializzato vuoto
Stack st = new Stack();
 push(st, s);
 while(!stack_isempty(st)) { // iteriamo fintanto che non svuotiamo lo stack
   s = pop(st);
   if (!visited[s]) {
     visit(g, s);  visited[s] = true;
     // aggiungiamo nodi successori allo stack
     for(int j = 0; j < getnumvertex(g); j++)
       if (connected(g, s, j) && !visited[j])
         push(st, j);
 }
 delete [] visited;
 delete st;                                  // Implementare questo algoritmo
}
```


Caratteristiche DFS

- L'algoritmo DFS richiede un tempo proporzionale a |V| + |E| in un grafo G=(V,E), quindi ha limite superiore asintotico O(|V|+|E|).

Algoritmo BFS

- È un algoritmo alternativo all'algoritmo di ricerca in profondità.
- È anche detto algoritmo di vista in ampiezza.
- Dato un vertice *s*
  - Visito *s*
  - Visito i vertici a distanza 1 da *s*
  - Proseguo visitando i vertici a distanza 2 da *s* (ovvero a distanza 1 da quelli a distanza 2 da *s*).
- Si basa sull'algoritmo di attraversamento di alberi binari *level-order*.
- Similmente al caso dell'attraversamento level-order per alberi binari, l'algoritmo BFS si ottiene dall'algoritmo DFS iterativo usando una "Coda" invece che uno "Stack".

Algoritmo BFS: implementazione

```
void BFS(Grafo *g, int s, void visit(Grafo *, int)) {
 bool * visited = new bool [getnumvertex(g)];
 for(int i = 0; i  < getnumvertex(g); i++) visited[i] = false;

// utilizziamo uno stack inizializzato vuoto
 Queue q = new Queue();
 put(q, s);
 while(!queue_isempty(q)) { // iteriamo fintanto che non svuotiamo la coda
   s = get(q);
   if (!visited[s]) {
     visit(g, s);  visited[s] = true;
     // aggiungiamo nodi successori alla coda
     for(int j = 0; j < getnumvertex(g); j++)
       if (connected(g, s, j) && !visited[j])
         put(q, j);
 }
 delete [] visited;
 delete q;
}
```


Algoritmo BFS: caratteristiche

- L'algoritmo di visita in breadth-first impiega tempo proporzionale alla somma del numero di vertici e del numero di archi (dimensione delle liste di adiacenza).
- Il limite superiore asintotico è O(|E| + |V|).

Raggiungibilità di un vertice

- Due nodi *s* e *t* sono connessi se e solo se esiste un percorso tra *s* e *t*.
- Si vuole sviluppare un algoritmo che ritorni true se esiste un percorso tra *s* e *t*, false altrimenti.
- Osservazione:
  - il nodo finale è raggiungibile dal nodo iniziale se e solo se un algoritmo di visita che parte dal nodo iniziale riesce a visitare il nodo finale.
- Il modo più semplice per realizzarlo consiste nel modificare uno degli algoritmi di visita analizzati.
  - Alla fine della visita basta chiedersi se il nodo finale è marcato visited.

Raggiungibilità: implementazione

```
bool are_connected(Grafo *g, int s, int d) {
 bool * visited = new bool [getnumvertex(g)];
 for(int i = 0; i  < getnumvertex(g); i++) visited[i] = false;

// utilizziamo uno stack inizializzato vuoto
 Queue q = new Queue();
 put(q, s);
 while(!queue_isempty(q)) { // iteriamo fintanto che non svuotiamo la coda
   s = get(q);
   if (!visited[s]) {
     visited[s] = true;
     for(int j = 0; j < getnumvertex(g); j++)
       if (connected(g, s, j) && !visited[j]) put(q, j);
 }
 bool result = visited[d];
 delete [] visited;  delete q;
 return result;
}
```


Percorso a distanza minima

- Problema: calcolare percorso a distanza minima tra due vertici *s* e *d*.
- Intuizione: sfruttiamo algoritmo BFS per costruire tale percorso.
  - Usiamo un array u[] tale che u[i] mi dice chi è il vertice predecessore del vertice i.
  - Per ogni vertice v adiacente al vertice s in esame, memorizziamo in u[v] il predecessore di v, ovvero s.
  - Alla fine della visita, un attraversamento dei nodi di u partendo dal nodo d ci da il percorso minimo tra s e d.

Percorso a distanza minima

```
void print_minimumpath(Grafo *g, int s, int d, void visit(Grafo *, int)) {
 bool * visited = new bool [getnumvertex(g)];
 int * U = new int [getnumvertex(g)];
 for(int i = 0; i  < getnumvertex(g); i++) {
   visited[i] = false; U[i] = -1;
 }
 Queue q = new Queue();  put(q, s);
 while(!queue_isempty(q)) {
   int s = get(q);
   if (!visited[s]) {
     visited[s] = true;
     for(int j = 0; j < getnumvertex(g); j++)
       if (connected(g, s, j) && !visited[j]) {
          put(q, j);  U[j] = s;
       }
 }
 delete [] visited;  delete q;
 print_path(g, s, d, U, visit);
 delete [] U;
}
```


Percorso a distanza minima

```
void print_path(Grafo g, int s, int d, int [] U, void visit(Grafo g, int)) {
  if (s == d) visit(g, s);
  else if (U[d] == -1) {
       cout << "There is no path among " << s << " and " << d << endl;
  else {
       print_path(g, s, U[d], visit);
       visit(g, d);
  }
}
```


Esercizi

- Scrivere una funzione bool check_path(Grafo g, Node * n) che ritorni true se il percorso rappresentato dalla lista concatenata n appartiene al grafo, false altrimenti.
- La lunghezza del percorso tra due vertici *s* e *t* rappresenta la distanza secondo quel percorso tra *s* e *t*. Scrivere una funzione
  int get_distance(Grafo g, int s, int t);
  che ritorni la distanza di un percorso tra *s* e *t* se tale percorso esiste, -1 altrimenti.
- Scrivere una funzione che ritorni una lista corrispondente al percorso minimo tra due vertici *s* e *d*.
  - Hint: sfruttare la struttura della funzione print_path.
- Scrivere una funzione che ritorni true se il grafo non orientato su cui è invocata è connesso.
  - Hint: se il grafo è connesso esiste un percorso da ogni nodo ad ogni altro nodo. Quindi basta verificare che ogni vertice sia stato visitato applicando una visita ad un qualsiasi vertice del grafo.