UNIVERSITA' DEGLI STUDI DI ROMA
"LA SAPIENZA"

SCUOLA DI SPECIALIZZAZIONE IN
RICERCA OPERATIVA E STRATEGIE DECISIONALI

CORSO DI LABORATORIO I
PROF. P. DELL'OLMO

CORSO DI APPLICAZIONI DEI GRAFI
PROF. G. STORCHI

DISCONNESSIONE DI UN GRAFO

Autore: Dr. Francesco Crisci

SOMMARIO

DESCRIZIONE DEL PROBLEMA

Il problema della connessione di un grafo è una misura importante dell'affidabilità e della stabilità di una rete.

L'**arc connectivity** di un grafo connesso è il minimo numero di archi la cui rimozione dal grafo lo disconnette in uno o più componenti.

Il problema che ci proponiamo di affrontare è quello di trovare un algoritmo per risolvere il problema dell'*arc connectivity* per un grafo non orientato.

Diamo, in primo luogo, alcune definizioni:

un **disconnecting set** è un insieme di archi la cui cancellazione dal grafo lo disconnette in due o più componenti. Per tale motivo, l'*arc connectivity* del grafo uguaglia la minima cardinalità di ogni *disconnecting set*: definiamo questo insieme di archi come il minimo *disconnecting set*.

L'*arc connectivity* di una coppia di nodi **i** e **j** è il minimo numero di archi la cui rimozione dal grafo disconnette questi due nodi. Rappresentiamo la coppia di nodi **i** e **j** come **[i , j]** e l'*arc connectivity* di questa coppia come **a[i , j]** . Poniamo anche **a(G)** denotante l'*arc connectivity* del grafo **G**.

Di conseguenza,

$$a(G) = \min \{ a[i,j] : [i,j] \in N \times N \}$$

Riportiamo alcune proprietà elementari riguardati l'*arc connectivity* di un grafo:

a) **a[i , j] = a[j , i]** per ogni coppia di nodi **[i , j];**

b) l'*arc connectivity* del grafo non può eccedere il grado minimo dei nodi del grafo, cioè

$$a(G) \leq \lfloor 2m/n \rfloor;$$

c) ogni *disconnecting set* minimo partiziona il grafo in esattamente due componenti;

d) l'*arc connectivity* di uno *spanning tree* è uguale ad 1;

e) l'*arc connectivity* di uno *ciclo* è uguale ad 2.

Denotiamo con d il grado minimo di un nodo nel grafo e sia **p** un nodo con grado uguale a d.

La proprietà (b) implica che $a(G) \leq d \leq \lfloor 2m/n \rfloor$.

Dal momento che un *disconnecting set* minimo partiziona l'insieme di nodi in esattamente due **componenti** $S^* \subset N$ **e** $\bar{S}^* \subset N - S^*$, possiamo rappresentare questo taglio con la notazione $[S^*, \bar{S}^*]$. Assumiamo, senza alcuna perdita di generalità, che il nodo $p \in S^*$.

Definiamo la capacità di ogni arco del grafo uguale ad 1; consideriamo ogni taglio s-t (sorgente-pozzo) $[S, \bar{S}]$ nel grafo. Dal momento che ogni arco ha capacità 1, la capacità di questo taglio è $|(S, \bar{S})|$ (è uguale al numero di archi nel taglio). Siccome ogni cammino dal nodo **s** al nodo **t** contiene almeno un arco in $(S, \bar{S})$, la rimozione degli archi in $(S, \bar{S})$ disconnette tutti i cammini dal nodo s al nodo t. Di conseguenza, il grafo contiene un *disconnecting set* di archi di cardinalità uguale alla capacità di ogni taglio s-t $[S, \bar{S}]$.

Il teorema *massimo flusso - minimo taglio* implica immediatamente che il massimo flusso da s a t sul grafo G uguaglia il numero minimo di archi la cui rimozione disconnette tutti i cammini dal nodo s al nodo t.

In base a queste considerazioni, per determinare l'*arc connectivity* di un grafo bisogna risolvere un problema di massimo flusso di capacità unitaria tra ogni coppia di nodi (variando i nodi sorgente e pozzo); il valore minimo tra tutti questi flussi è $\alpha(G)$.

Analizziamo gli algoritmi di massimo flusso che conosciamo:

Algoritmo	Complessità	Considerazioni
Ford-Fulkerson Labelling	O(nmU) U max capacità di un arco sul grafo	a) mantiene un flusso ammissibile ed aumenta i flussi nel grafo residuale dal nodo s al nodo t; b) facile da implementare e molto flessibile; c) la complessità è pseudopolinomiale (essendo U un dato di input del problema): l'algoritmo in pratica non è molto efficiente. Inoltre se i pesi degli archi sono numeri irrazionali, l'algoritmo opera una sequenza infinita di aumenti di flusso convergente a valori diversi dalla soluzione ottima;
Edmonds-Karp	$O(n^2 m)$	a) implementazione speciale dell'algoritmo di Ford-Fulkerson; b) aumenta i flussi lungo i cammini più brevi dal nodo s al nodo t sul grafo residuale; c) relativamente semplice da implementare e molto efficiente in pratica.
Dinic Distance label	$O(n^2 m)$	a) parte da una soluzione ammissibile e costruisce soluzioni migliori fino a giungere a quella ottima; b) costruisce una sottorete della rete detta rete stratificata e calcola un flusso massimale su tale sottorete.

Dal momento che il problema di massimo flusso riguarda un grafo di capacità unitaria degli archi, la complessità dell'algoritmo di Ford-Fulkerson, che sappiamo essere O(nmU), essendo U=1 diventa O(nm) che rappresenta la complessità migliore tra gli algoritmi esaminati.

Per determinare l'*arc connectivity* di un grafo dobbiamo applicare tale algoritmo tra ogni coppia di nodi, per cui l'algoritmo avrà una complessità $O(n^2) \cdot O(nm)$ e quindi $O(n^3 m)$.

Tuttavia questo approccio può essere facilmente migliorato di un fattore di n effettuando la seguente considerazione.

Consideriamo un nodo $\mathbf{k} \in \bar{S}^*$ ed il nodo $\mathbf{p} \in S^*$ con grafo minimo nel grafo. Dal momento che il taglio $[S^* , \bar{S}^*]$ disconnette i nodi p e k, la minima cardinalità dell'insieme di archi che disconnetterà questi due nodi è al più $|[S^* , \bar{S}^*]|$, cioè:

$$\mathbf{a[p, k]} \leq |[S^* , \bar{S}^*]|$$

Osserviamo inoltre che $[S^* , \bar{S}^*]$ è il *disconnecting set* minimo del grafo, per cui abbiamo che:

$$\mathbf{a[p, k]} \geq |[S^* , \bar{S}^*]|$$

Dalle due precedenti relazioni otteniamo che:

$$\mathbf{a[p, k]} = |[S^* , \bar{S}^*]|$$

Quindi se calcoliamo $\mathbf{a[p, j]}$ per ogni j, il minimo tra questi numeri è uguale ad $\mathbf{a[G]}$, cioè

$$\mathbf{a[G] = \min \{ a[p , j] : j \in N - \{p\} \}}$$

Le considerazioni precedenti ci permettono di determinare l'*arc connectivity* del grafo risolvendo (n - 1) problemi di massimo flusso di capacità unitaria richiedendo $O(n^2 m)$ tempo.

Tuttavia l'algoritmo di Ford-Fulkerson, in questo caso, effettua al più $\lfloor m/n \rfloor$ aumenti di flusso per risolvere ogni problema di flusso massimo (perché il grado del nodo p è $d \leq \lfloor m/n \rfloor$) e richiederà $O(m^2/n)$ tempo per cui la complessità dell'algoritmo di *arc connectivity* è $O(m^2)$.

Il problema del flusso massimo

Flusso e valore del flusso, taglio e capacità del taglio

Sia $G = (N, A)$ un grafo orientato che abbia un nodo origine s e un nodo destinazione t, e su cui sia definito il vettore $u = [u_{ij}]$ delle capacità superiori degli archi, che ipotizziamo intere.

Il *problema di flusso massimo* consiste nel determinare la massima quantità di flusso inviabile in G da s a t, si vuole cioè determinare un flusso ammissibile x la cui variabile di bilancio v, detta *valore del flusso*, sia massima nel nodo t: $b_t = v$, e quindi $b_s = -v$.

Introduciamo alcune proprietà. Un taglio (N_s , N_t) è detto *taglio ammissibile*, o semplicemente *taglio*, se $s \in N_s$ e $t \in N_t$. Ricordiamo che l'insieme $A^+(N_s , N_t)$ degli *archi diretti* e quello $A^-(N_s , N_t)$ degli *archi inversi* del taglio (N_s , N_t) sono rispettivamente:

$A^+(N_s , N_t) = \{(i,j) \in A : i \in N_s , j \in N_t\}$, $A^-(N_s , N_t) = \{(i,j) \in A : i \in N_t , j \in N_s\}$.

Ad ogni taglio (N_s , N_t) associamo la *capacità del taglio* $u(N_s , N_t)$ e, dato un flusso x, il *flusso del taglio* $x(N_s , N_t)$ così definiti:

$$u(N_s , N_t) = \sum_{(i,j) \in A^+(N_s , N_t)} u_{ij} \qquad x(N_s , N_t) = \sum_{(i,j) \in A^+(N_s , N_t)} x_{ij} - \sum_{(i,j) \in A^-(N_s , N_t)} x_{ij}$$

Teorema 1

Dato un flusso ammissibile x^* di valore v^*, per ogni taglio (N_s , N_t) risulta:

$$v^* = x^*(N_s , N_t)$$

Dim.

La dimostrazione è immediata: basta sommare tra loro i vincoli di conservazione del flusso corrispondenti ai nodi di N_t.

Teorema 2

Sia x^* una soluzione ammissibile di valore v^*, e sia (N_s , N_t) un taglio. Si ha:

$$v^* = x^*(N_s , N_t) \leq u(N_s , N_t)$$

Inoltre, se $v^* = u(N_s , N_t)$, allora il flusso è massimo ed il taglio ha capacità minima.

Dim.

La proprietà $v^* = x^*(N_s , N_t)$ deriva dal teorema precedente. Inoltre:

$$x^*(N_s , N_t) = \sum_{(i,j) \in A^+(N_s , N_t)} x_{ij}^* - \sum_{(i,j) \in A^-(N_s , N_t)} x_{ij}^* \leq \sum_{(i,j) \in A^+(N_s , N_t)} u_{ij} - \sum_{(i,j) \in A^-(N_s , N_t)} x_{ij}^* \leq \sum_{(i,j) \in A^+(N_s , N_t)} u_{ij} \leq$$

$\leq \sum u(N_s, N_t)$

Se $v^* = x^*(N_s, N_t) = u(N_s, N_t)$ segue che, per ogni arco diretto $(i,j) \in A^+(N_s, N_t)$, il flusso è $x_{ij}^* = u_{ij}$ mentre, per ogni arco inverso $(i,j) \in A^-(N_s, N_t)$, il flusso è $x_{ij}^* = 0$. Se esistesse un flusso x' di valore $v' > v$, si avrebbe $v' > u(N_s, N_t)$, che è assurdo per il Teorema 1.

Un'altra formulazione del Teorema precedente è la seguente:

Teorema (MinCut-MaxFlow). *Dato un grafo orientato, sui cui archi siano definite delle capacità non negative, e dati un nodo origine s ed un nodo destinazione t, il massimo valore dei flussi ammissibili da s a t è uguale alla minima delle capacità dei tagli ammissibili.*

Nel seguito mostreremo che esiste un taglio (N_s^*, N_t^*) per cui è $v^* = u(N_s^*, N_t^*)$, dove con v^* indichiamo il valore del flusso massimo.

Cammini aumentanti

Dato un cammino (semplice) P_{st} di G tra s e t, si partizioni l'insieme dei suoi archi nei due insiemi P^+ e P^-, contenenti rispettivamente gli archi orientati in modo concorde con il verso del cammino da s a t, detti archi *concordi*, e gli archi orientati in modo discorde, detti archi *discordi*:

$$P^+ = \{(i,j) \in P: i \text{ precede } j \text{ nel verso da } s \text{ a } t\}, \quad P^- = \{(i,j) \in P: i \text{ segue } j \text{ nel verso da } s \text{ a } t\}.$$

Sia x un flusso ammissibile di valore v; dall'insieme degli archi A si possono selezionare due sottoinsiemi disgiunti $A^+(x)$ e $A^-(x)$, detti rispettivamente *insieme degli archi non saturi* e *insieme degli archi non vuoti*:

$$A^+(x) = \{(i,j) \in A: x_{ij} < u_{ij}, \}, A^-(x) = \{(i,j) \in A: x_{ji} > 0\}.$$

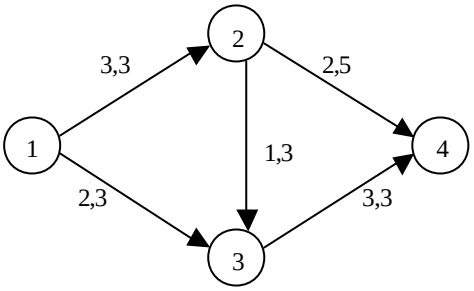
Un cammino P_{st} è detto *cammino di flusso aumentante* (o *cammino aumentante*) da s a t relativamente a x se $P^+ \subseteq A^+(x)$ e $P^- \subseteq A^-(x)$, cioè se tutti i suoi archi concordi sono non saturi e tutti i suoi archi discordi sono non vuoti. Il valore $\theta(P_{st}, x)$:

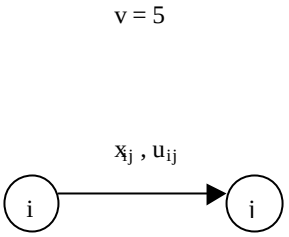
$$\theta(P_{st}, x) = \min\{\min\{u_{ij} - x_{ij} : (i,j) \in P^+\}, \min\{x_{ij} : (i,j) \in P^-\}\},$$

è detto la *capacità* del cammino aumentante P_{st}. È immediato verificare che $\theta(P_{st}, x) > 0$.

Avendo supposto che le capacità sono intere, se x è un vettore intero allora anche $\theta(P_{st}, x)$ è un intero per ogni cammino aumentante da s a t.

Esempio





L'insieme degli archi non saturi è $A^+(x) = \{(1,3), (2,3),(2,4)\}$, e quello degli archi non vuoti è $A^-(x) = A$. Il cammino $P_{14} = \{(1,3), (2,3), (2,4)\}$ è formato dall'insieme degli archi concordi $P^+ = \{(1,3), (2,4)\}$ e da quello degli archi discordi $P^- = \{(2,3)\}$; è un cammino aumentante di capacità $\theta(P\,14) = \min\{\min\{1,3\}, \min\{1\}\} = 1$.

Un cammino aumentante relativamente a x, P_{st} , permette la determinazione di un nuovo flusso ammissibile di valore maggiore semplicemente incrementando del valore $\theta(P_{st}, x)$ il flusso su ogni arco concorde $(i,j) \in P^+$, e decrementando dello stesso valore il flusso su ogni arco $(i,j) \in P^-$. Il nuovo flusso x', di valore $v' = v + \theta(P_{st}, x)$, è dato da:

$$x'_{ij} = \begin{cases} x_{ij} + \theta(P\,st, x), & \text{se } (i,j) \in P^+, \\ x_{ij} - \theta(P\,st, x), & \text{se } (i,j) \in P^-, \\ x_{ij}, & \text{se } (i,j) \notin P_{st}. \end{cases}$$

DESCRIZIONE DELL'ALGORITMO

Determinazione di un flusso massimo

Definiamo un algoritmo per la determinazione di un flusso massimo. Tale algoritmo parte da un flusso nullo e, ad ogni passo, determina un cammino aumentante, incrementando il valore del flusso; l'algoritmo si ferma quando non esiste più alcun cammino aumentante, restituendo quindi anche un taglio di capacità minima.

Algoritmo Cammino_aumentante

```
Procedure Cammino_aumentante (G, s, t, x, v, Ns , Nt):
 begin
        v := 0; ottimo := falso;
        ∀(i; j) ∈ A do x_ij := 0;
        repeat
                if Trova_Cammino(x, P, θ, N_s , N_t);
                then Aumenta_Flusso(x, P, θ)
                else ottimo = vero
        until ottimo
 end.
```

Analizziamo ora come determinare un cammino aumentante da *s* a *t*, relativamente a un flusso *x*. La ricerca può essere effettuata tramite una visita del grafo, partendo da *s*, in cui si permette di percorrere un arco lungo il suo verso solo se non è saturo, e nel verso opposto solo se non è vuoto.

Utilizziamo una procedura di visita del grafo modificata in modo che determini il cammino aumentante la sua capacità, memorizzando per ciascun nodo raggiunto nella visita la capacità dell'unico cammino da *s* al nodo sull'albero della visita $T_s = (N_s , A_s)$. Se, al termine della visita, si ha $t \in N_s$, si è determinato un cammino aumentante (e la sua capacità). Altrimenti si ottiene un taglio (N_s, N_t), con $N_t = N \setminus N_s$ che, come mostreremo in seguito, è un taglio di capacità minima.

La procedura *Trova_Cammino* deve determinare se esiste un cammino aumentante da s a t, dato il flusso x. Se il cammino esiste, *Trova_Cammino* restituisce vero e fornisce il cammino P e la sua capacità; altrimenti restituisce falso e fornisce un taglio, (N_s , N_t), di capacità minima. Tale procedura è sostanzialmente una procedura di visita. La procedura *Aumenta_Flusso* aggiorna il flusso x inviando sul cammino P un ulteriore quantità di flusso θ.

E' facile verificare che l'algoritmo *Cammino_aumentante*, nel caso in cui le capacità siano intere, ha complessità O(mnU), con U = max { u_{ij} : (i; j) ∈ A }. Infatti, se la soluzione x è intera (e la soluzione iniziale lo è certamente), allora anche θ(P; x) è intera, e quindi anche x' lo sarà: di conseguenza, ad ogni iterazione (a parte l'ultima) il valore del flusso viene aumentato di almeno una unità. nU è una maggiorazione della capacità del taglio ({s},N\{s}), e pertanto anche della capacità minima dei tagli, e di conseguenza anche del massimo valore del flusso; pertanto il numero di iterazioni sarà limitato superiormente da nU, ed ogni iterazione ha complessità O(m). Osserviamo che si tratta di una complessità pseudopolinomiale, essendo U un dato di input del problema.

Riportiamo la procedura di ricerca di un cammino aumentante, che chiameremo *Cammino_aumentante*.

```
Procedure Cammino_aumentante(s,t,x,P,θ,Ns,Nt,ottimo):
    begin
        ottimo = false; for i := 1 to n do P[i] :=0;                {inizializzazione}
        P[s] := nil; d[s] := M; Q := {s}; Ns := Ø;
        repeat
          select i from Q; Q := Q \ {i}; Ns := Ns∪{i};      {selezione e rimozione di un nodo da Q}
          for each (i,j) ∈ FS(i) do                          {esplorazione della stella uscente di i}
            if P[j] = 0 and x[i,j] < u[i,j] then             {l'arco (i,j) permette di raggiungere il
              begin                                           nodo j con un cammino aumentante}
                P[j] := i; Q := Q ∪ {j}
                 if d[i] < u[i,j] - x[i,j] then d[j] := d[i]   {determinazione della massima
                                      else d[j] := u[i,j]-x[i,j]   variazione di flusso da s a j}
              end;
          for each (j,i) ∈ BS(i) do                          {esplorazione della stella entrante in i}
            if P[j] = 0 and x[j,i] > 0 then                  {l'arco (j,i) permette di raggiungere il
              begin                                           nodo j con un cammino aumentante}
                P[j] := -i; Q := Q ∪ {j}
                 if d[i] < x[j,i] then d[j] := d[i]          {determinazione della massima
                                  else d[j] := x[j,i]          variazione di flusso da s a j}
              end
        until Q = Ø or P[t] ≠ 0;
        if P[t] ≠ 0 then θ := d[t]                           {se t è stato raggiunto, si determina θ}
                    else begin
                           ottimo := true; Nt := N \ Ns      {si costruisce il taglio di capacità
                         end                                  minima (Ns,Nt)}
    end.
```

La procedura *Cammino_aumentante*, che ha complessità in tempo $O(m)$, riceve in ingresso un flusso ammissibile x e restituisce un vettore $P[.]$ per descrivere il cammino aumentante individuato, nel caso ne esista uno. La procedura utilizza un vettore $d[.]$, in cui $d[i]$ rappresenta la capacità del cammino aumentante parziale da s ad i, se i viene raggiunto durante l'esplorazione. Nel caso venga raggiunto il nodo t, la procedura restituisce $\theta = d[t]$; altrimenti, restituisce il taglio memorizzato negli insiemi N_s ed N_t e segnala che il flusso è ottimo ponendo

la variabile *ottimo=true*. Infatti, per il modo in cui è stata effettuata la visita, tutti gli archi di A^+ (N_s , N_t) sono saturi, mentre tutti gli archi di A^- (N_s , N_t) hanno flusso nullo; quindi x è un flusso massimo. Vale pertanto:

Teorema 3

Dato un grafo orientato, sui cui archi siano definite delle capacità non negative, e su cui siano individuati un nodo origine s ed un nodo destinazione t, il massimo valore dei flussi ammissibili da s a t è uguale alla minima delle capacità dei tagli ammissibili.

Nell'algoritmo *Cammino_aumentante* il modo in cui viene determinato il cammino aumentante non è specificato. A seconda di tale modalità si possono avere diverse versioni dell'algoritmo, con diverse complessità e proprietà.

L'algoritmo di Edmonds & Karp è una implementazione di *Cammino_aumentante*, in cui, ad ogni iterazione, viene determinato fra i cammini aumentanti uno con minimo numero di archi. Questo può essere ottenuto determinando il cammino aumentante per mezzo di una visita a ventaglio (FIFO) del grafo. E' possibile dimostrare che un qualsiasi arco (i, j) saturato mediante un cammino di lunghezza k potrà successivamente entrare a far parte solamente di cammini aumentanti di lunghezza almeno k +2; da questo è facile derivare che l'algoritmo di Edmonds & Karp ha una complessità pienamente polinomiale pari a $O(m^2n)$. Questo risultato è particolarmente interessante perché vale anche nel caso in cui le capacità non siano intere.

Indichiamo con *Aumenta_flusso* una procedura che incrementa della quantità $\theta(P_{st}, x)$ il flusso sul cammino aumentante P_{st} , restituendo il nuovo flusso. *Aumenta_flusso*, di complessità $O(n)$ viene chiamata se è stato determinato un cammino aumentante da *s* a *t*, descritto mediante il vettore *P*[.].

```
Procedure Aumenta_Flusso(s,t,x,P,θ):
    begin
      j := t;                          {il cammino è percorso da t verso s}
      while j ≠ s do
          if P[j] < 0 then             {se l'arco corrente è (j,i)}
             begin                     {si decrementa il flusso}
               i := -P[j];
               x[j,i] := x[j,i] - θ;
               j := i
             end
          else
             begin                     {altrimenti, si incrementa il flusso}
               i := P[j];
               x[i,j] := x[i,j] + θ;
               j := i
             end
    end
```

Algoritmo Flusso_Massimo

La procedura *Flusso_massimo* si basa sulla ricerca iterata di cammini aumentanti. Il flusso iniziale è nullo. La procedura ha termine quando viene determinato un taglio di capacità minima; vengono forniti il flusso massimo *x*, il suo valore *v* ed il taglio ottimo memorizzato nei due vettori N_s ed N_t .

```
Procedure Flusso_massimo(s,t,x,v,Ns,Nt):
          begin
              v := 0; ottimo := false;          {inizializzazione: si parte da una soluzione }
              for each (i,j)∈ A do x[i,j] := 0;  {ammissibile e si applica un algoritmo di  }
            repeat                               { ricerca locale}
              Cammino_aumentante(s,t,x,P,θ,Ns,Nt,ottimo);   {ricerca del cammino aumentante}
               if not ottimo then
                     begin                                  {aumento del flusso}
                       Aumenta_Flusso(s,t,x,P,θ);
                       v := v + θ
                       if v > minflusso then return         {branch & bound sull'albero delle decisioni}
                     end
             until ottimo
          end
```

Teorema 4

Se le capacità degli archi sono intere, la procedura Flusso_massimo è corretta e ha complessità pseudopolinomiale O(mU), dove $U = n \max\{u_{ij} : (i,j) \in A\}$.

Dim.

La procedura termina in un numero finito di iterazioni. Infatti, ad ogni iterazione, il valore v del flusso corrente cresce strettamente di almeno un'unità ($\theta \geq 1$); se si indica con U^* la capacità, finita, del taglio ottimo, segue che l'algoritmo esegue al più U^* iterazioni. Dal Teorema 3 discende allora che, al termine della procedura, viene individuato un flusso di valore massimo. Poiché ogni iterazione ha complessità $O(m)$, ed essendo $U^* \leq U$, si ha che la complessità in tempo della procedura è $O(mU)$.

Algoritmo Disconnecting_set

La procedura *Disconnecting_set* si basa sulla ricerca iterata di tagli minimi tra coppie di nodi di cui uno è il nodo p di grado minimo. La procedura ha termine quando viene determinato un disconnecting set minimo.

```
Procedure Disconnecting_set ( )
    begin
        minflusso := 0;
        p := carica_grafo( );                          {carica il grafo e restituisce il nodo}
                                                       {di grado minimo}
        for i:=1 to n do
            if i <> p then
            begin                                      {cerca il flusso massimo tra la}
                flusso := flusso_massimo(p,i,Nt ,Ns)   {coppia di nodi p ed i}
                if flusso < minflusso then
                    begin                              { memorizza il taglio minimo}
                        minflusso := flusso;
                        Ts := Ns;
                        Tt := Nt;
                    end
            end
    end
```

RAPPRESENTAZIONE DEL GRAFO IN MEMORIA

Sappiamo che la struttura per liste di adiacenza può essere anche semplificata sotto alcune ipotesi, in generale facilmente verificate, quali: i nodi hanno gli indici 1, 2, …, *n*, gli archi possono essere arbitrariamente ordinati, non si prevedono aggiunte di nuovi nodi ed archi, le eventuali rimozioni temporanee di nodi ed archi possono essere facilmente implementate mediante l'inserimento di puntatori dinamici. In tal caso le liste a puntatori dei nodi e degli archi possono essere agevolmente realizzate mediante vettori facendo corrispondere l'indice del nodo o dell'arco con l'indice della componente del vettore contenente le informazioni relative al nodo o all'arco.

Per realizzare la lista di adiacenza per stelle uscenti, con il minimo spazio-memoria, abbiamo utilizzato un vettore, *Nodi[.]*, ad *n*+1 componenti, una per ogni nodo più una ausiliaria, ed un vettore, *Star[.]*, ad *m* componenti, una per ogni arco. Gli archi vengono ordinati per stelle uscenti (cioè gli archi della stella uscente di *i*+1 seguono gli archi della stella uscente di *i*).

L'elemento i-esimo del primo vettore (i = 1,…,n), *Nodi[i]*, contiene il puntatore al primo arco della stella uscente del nodo i, cioè è l'indice della componente di *Star[.]* contenente il nodo testa del primo arco di *FS*(*i*); se *FS*(*i*) = Ø, allora si pone *Nodi[i] = Nodi[i+1].* Ovviamente *Nodi[1]=1*; il puntatore relativo al nodo fittizio *n*+1 punta all'*m*+1-esimo arco (arco fittizio).

Per conoscere la stella uscente del nodo i basta effettuare una scansione del vettore *Star[.]* tra la posizione *Nodi[i]* e la posizione *Nodi[i + 1] - 1*, ottenendo le teste degli archi aventi i come coda. La stella uscente è vuota se *Nodi[i] = Nodi[i + 1].*

L'occupazione di memoria di questa rappresentazione della lista di adiacenza è m+ n + 1.

RAPPRESENTAZIONE DELLE LISTE

Sappiamo che una *lista* $q = [x_1, x_2,\ldots, x_n]$ è una sequenza di elementi arbitrari. Gli elementi x_1 e x_n , le *estremità* della lista, sono anche detti rispettivamente la *testa* e la *coda* della lista stessa. Denoteremo con $|q|$ la cardinalità n della lista q e con $q[i]$ l'elemento i esimo della lista q, cioè x_i .

Una lista su cui siano definite le operazioni di accesso ed inserzione in testa e di estrazione dalla testa è detta una *pila*; essa funziona in modo LIFO (*last in first out*).

Una lista su cui siano definite le operazioni di accesso in testa, inserzione in coda e di estrazione dalla testa è detta una *fila* (o anche spesso *coda*); una fila funziona in modo FIFO (*first in first out*).

Accesso

in testa data la lista q restituisce l'elemento $q[1]$;

in coda data la lista q restituisce l'elemento $q[|q|]$;

Inserzione

in testa data la lista q e l'elemento x restituisce la nuova lista $[x]\&q$;

in coda data la lista q e l'elemento x restituisce la nuova lista $q\&[x]$;

Estrazione

dalla testa data la lista q restituisce l'elemento $q[1]$ e sostituisce q con $q[2,\ldots, |q|]$;

dalla coda data la lista q restituisce l'elemento $q[|q|]$ e sostituisce q con $q[1,\ldots, |q|-1]$.

Una lista può essere rappresentata sia per mezzo di un *vettore* che di una *struttura a puntatori*.

Ad esempio, nel programma cammino_aumentante abbiamo rappresentato l'insieme Q come una fila rappresentata per mezzo del vettore lista, mantenendo negli interi *testa* e *coda* l'indice rispettivamente del primo e dell'ultimo elemento della fila; se la fila è vuota si pone *testa* = coda. Chiaramente, con questa rappresentazione, le operazioni proprie della fila richiedono tempo $O(1)$.

L'uso di vettori è una scelta ragionevole per rappresentare liste, purché si disponga di una valutazione (per eccesso) sufficientemente accurata del massimo numero di elementi che esse possono contenere, e purché non si debbano effettuare troppe operazioni quali la concatenazione di liste o estrazioni di elementi in posizioni diverse dalle estremità.

APPENDICE: CLASSI DI ALGORITMI UTILIZZATE

ALGORITMI ENUMERATIVI

Un *algoritmo enumerativo* è un algoritmo che effettua una visita sistematica ed, in principio, esaustiva dello spazio delle soluzioni.

L'insieme delle soluzioni può essere enumerato utilizzando un struttura ad albero denominata albero delle decisioni.

Gli algoritmi enumerativi effettuano una esplorazione dell'albero delle decisioni fino alla determinazione di un nodo foglia che corrisponde ad una soluzione ottima.

Le strategie di visita dell'albero di ricerca possono essere suddivise in due classi:

1) *Strategie senza informazione* (cieche o topologiche): si basano solo sulla topologia della soluzione.

2) *Strategie con informazione* (guidate): si basano, oltre che sulla topologia, anche su informazioni riguardanti la soluzione cercata, come ad esempio il valore della funzione obiettivo.

Questi algoritmi vengono chiamati enumerativi per il fatto che, almeno nel caso peggiore, portano all'enumerazione di tutte, o comunque di un gran numero di soluzioni ammissibili.

Questi metodi sono anche noti con i seguenti nomi: *enumerazione implicita*, *branch and bound*, *ricerca euristica.*

Dato che il numero di nodi dell'albero delle decisioni cresce in modo esponenziale con la dimensione del problema, si cerca di effettuare la visita dell'albero in modo da esaminare un numero il più possibile piccolo di nodi. Per questo è importante la capacità di riconoscere precocemente quando, proseguendo l'esame di un cammino oltre un certo nodo, non sia più possibile trovare soluzioni interessanti: interi sottoalberi possono così essere scartati e considerati visitati implicitamente. Ad esempio, nel problema del taglio minimo è inutile generare il sottoalbero di un certo nodo pozzo t se la soluzione parziale raggiunta assume un valore di taglio superiore a quello già trovato per un altro nodo terminale: si parla allora di algoritmi di enumerazione implicita.

Tipicamente, il modo per determinare se un certo sottoalbero può essere scartato è quello di calcolare valutazioni inferiori (nel caso di problemi di minimo) sul valore della soluzione ottima del sottoproblema corrispondente a quel nodo. Questo viene fatto risolvendo rilassamenti del

problema originale, cioè problemi con "meno vincoli" che risultino per questo molto più facili da risolvere. Se la valutazione inferiore corrispondente ad un certo nodo (il valore della soluzione ottima del rilassamento) è maggiore od uguale al valore della migliore soluzione del problema originario correntemente disponibile, allora il sottoalbero può non essere esaminato: è facile vedere, infatti, che nessuna soluzione ottima del problema può corrispondere ad una foglia nel sottoalbero.

L'efficienza degli algoritmi di enumerazione implicita dipende da diversi fattori, tra cui: il metodo utilizzato per ottenere approssimazioni inferiori, l'algoritmo euristico utilizzato per ottenere soluzioni ammissibili, le regole per la selezione della variabile da usare e le regole per la visita dell'albero delle decisioni (quale dei diversi nodi ancora disponibili esaminare per primo). Inoltre, è di grande importanza il modo stesso con cui viene definito l'albero delle decisioni: ad uno stesso problema possono essere associati diversi alberi delle decisioni.

Comunque, anche usando la migliore tra le strategie di ricerca, non si può mai escludere l'eventualità di dovere esaminare tutti i nodi, o comunque una frazione consistente dell'albero delle decisioni.

ALGORITMI DI RICERCA LOCALE

Gli algoritmi di ricerca locale sono caratterizzati dal fatto che, a partire da una soluzione ammissibile per il problema in esame, costruiscono una successione di soluzioni "vicine" ognuna alla precedente, fino a che non ne viene determinata una che viene riconosciuta come "ottimo locale" rispetto ad un intorno opportunamente definito.

Il ruolo degli algoritmi di ricerca locale nell'ottimizzazione nel discreto è particolarmente rilevante: a questa classe appartiene ad esempio l'algoritmo del simplesso.

Un algoritmo di ricerca locale è in generale caratterizzato dalle seguenti operazioni:

- si determina una soluzione ammissibile, x_0 , di partenza; spesso, tale soluzione è costruita usando un algoritmo greedy.

- si determina quindi una sequenza di soluzioni ammissibili, x_0 , x_1 ,..., x_k ,..., fermandosi non appena una delle soluzioni trovate viene riconosciuta come una soluzione ottima (oppure come un "ottimo locale").

Elemento caratterizzante di un algoritmo di questo tipo è una trasformazione σ, che data una soluzione, x_k , della sequenza generata, fornisce la soluzione successiva $x_{k+1} = \sigma(x_k)$.

Uno schema generale di algoritmo di ricerca locale è pertanto il seguente.

```
Procedure RICERCA_LOCALE(F,c,soluzione):
    begin
            x := AMMISSIBILE(F);
            while σ(x) ≠ x do x := σ(x);
            soluzione := x
    end.
```

Dove *F è* l'insieme delle possibili risposte o soluzioni, cioè l'*Insieme Ammissibile,* e $c : F \to \mathbf{R}$ rappresenta la *Funzione Obiettivo.*

Si ha $x = \sigma(x)$ quando la soluzione corrente x viene riconosciuta come ottima dall'algoritmo.

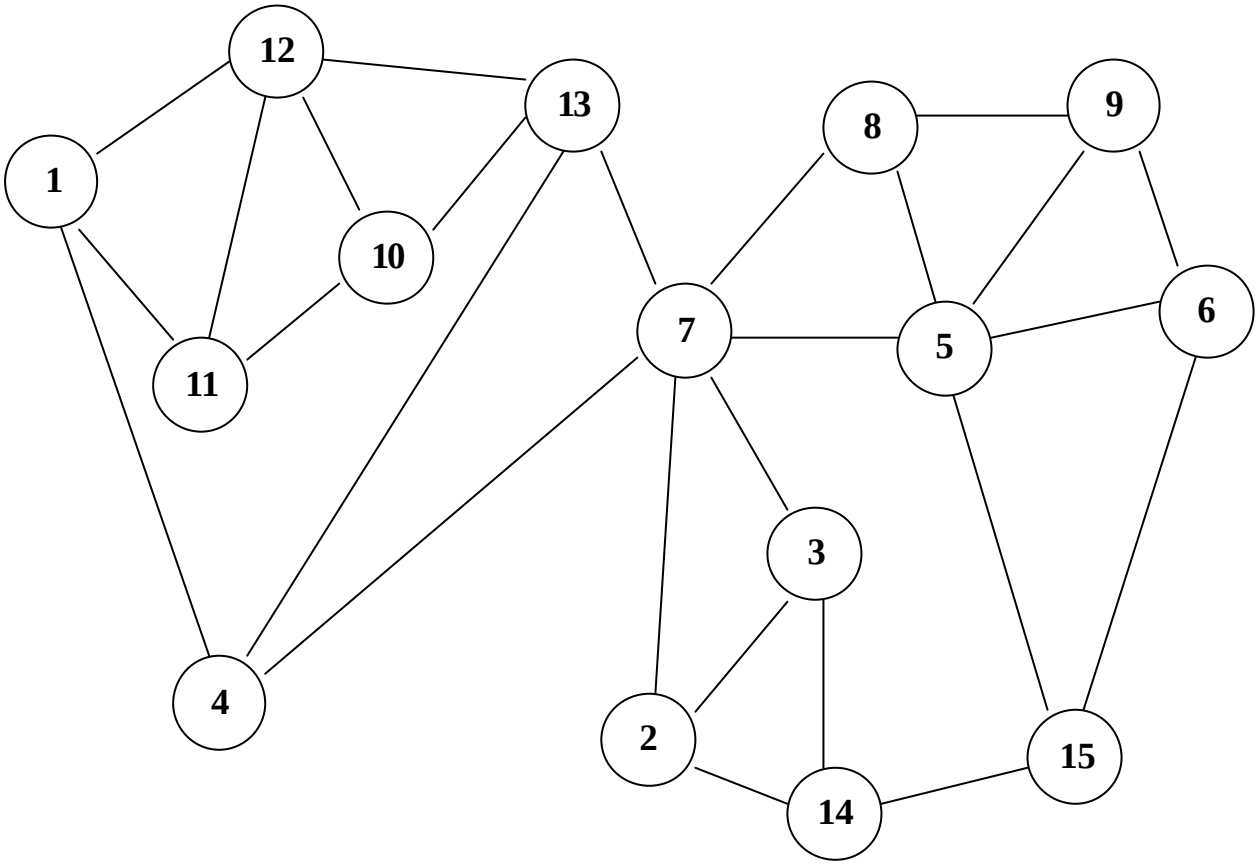
Ciò non vuole dire necessariamente che la soluzione sia realmente quella ottima, ma semplicemente che soddisfa un prefissato criterio di ottimalità.

BIBLIOGRAFIA

- R.K. Ahuja, l. Magnanti, J.B. Orlin - "Network flows: theory, algorithms and applications"
- Maffioli - "Elementi di programmazione matematica"
- G. Ausiello, A. Marchetti-Spaccamela, M. Protasi - "Teoria e progetto degli algoritmi fondamentali"
- A. Gibbons - "Algorithmic graph theory"
- Gallo, Pallottino, Storchi - "IAC29 - Algoritmi sui cammini minimi"
- Gallo, Pallottino - "Ottimizzazione ed esistenza"
- Gallo, Pallottino - "Appunti di Programmazione Matematica e Ricerca Operativa"
- Gallo, Pallottino - "Appunti di Programmazione Matematica" - IV edizione
- Gallo, Pallottini - "Reti di comunicazione"

ALLEGATO 2 : ESEMPI DI ESECUZIONE DEL PROGRAMMA

Applichiamo il programma realizzato al seguente grafo:



INDIVIDUAZIONE DEL MINIMO NUMERO DI ARCHI CHE DISCONNETTONO UN GRAFO

CARICAMENTO DEL GRAFO

Immettere il numero di nodi del grafo : 15
immetti i nodi adiacenti al nodo 1 (0 = fine)
-> 4
-> 11
-> 12
-> 0
immetti i nodi adiacenti al nodo 2 (0 = fine)
-> 3
-> 7
-> 14
-> 0
immetti i nodi adiacenti al nodo 3 (0 = fine)
-> 2
-> 7
-> 14
-> 0
immetti i nodi adiacenti al nodo 4 (0 = fine)
-> 1
-> 7
-> 13
-> 0
immetti i nodi adiacenti al nodo 5 (0 = fine)
-> 6
-> 7
-> 8
-> 9
-> 15
-> 0
immetti i nodi adiacenti al nodo 6 (0 = fine)
-> 5
-> 9
-> 15
-> 0
immetti i nodi adiacenti al nodo 7 (0 = fine)
-> 2
-> 3
-> 4
-> 5
-> 8
-> 13
-> 0
immetti i nodi adiacenti al nodo 8 (0 = fine)
-> 5
-> 7
-> 9
-> 0

immetti i nodi adiacenti al nodo 9 (0 = fine)
-> 5
-> 6
-> 8
-> 0
immetti i nodi adiacenti al nodo 10 (0 = fine)
-> 11
-> 12
-> 13
-> 0
immetti i nodi adiacenti al nodo 11 (0 = fine)
-> 1
-> 10
-> 12
-> 0
immetti i nodi adiacenti al nodo 12 (0 = fine)
-> 1
-> 10
-> 11
-> 13
-> 0
immetti i nodi adiacenti al nodo 13 (0 = fine)
-> 4
-> 7
-> 10
-> 12
-> 0
immetti i nodi adiacenti al nodo 14 (0 = fine)
-> 2
-> 3
-> 15
-> 0
immetti i nodi adiacenti al nodo 15 (0 = fine)
-> 5
-> 6
-> 14
-> 0

Il nodo di grado minimo e' il numero 1

Individuazione del minimo disconnecting set

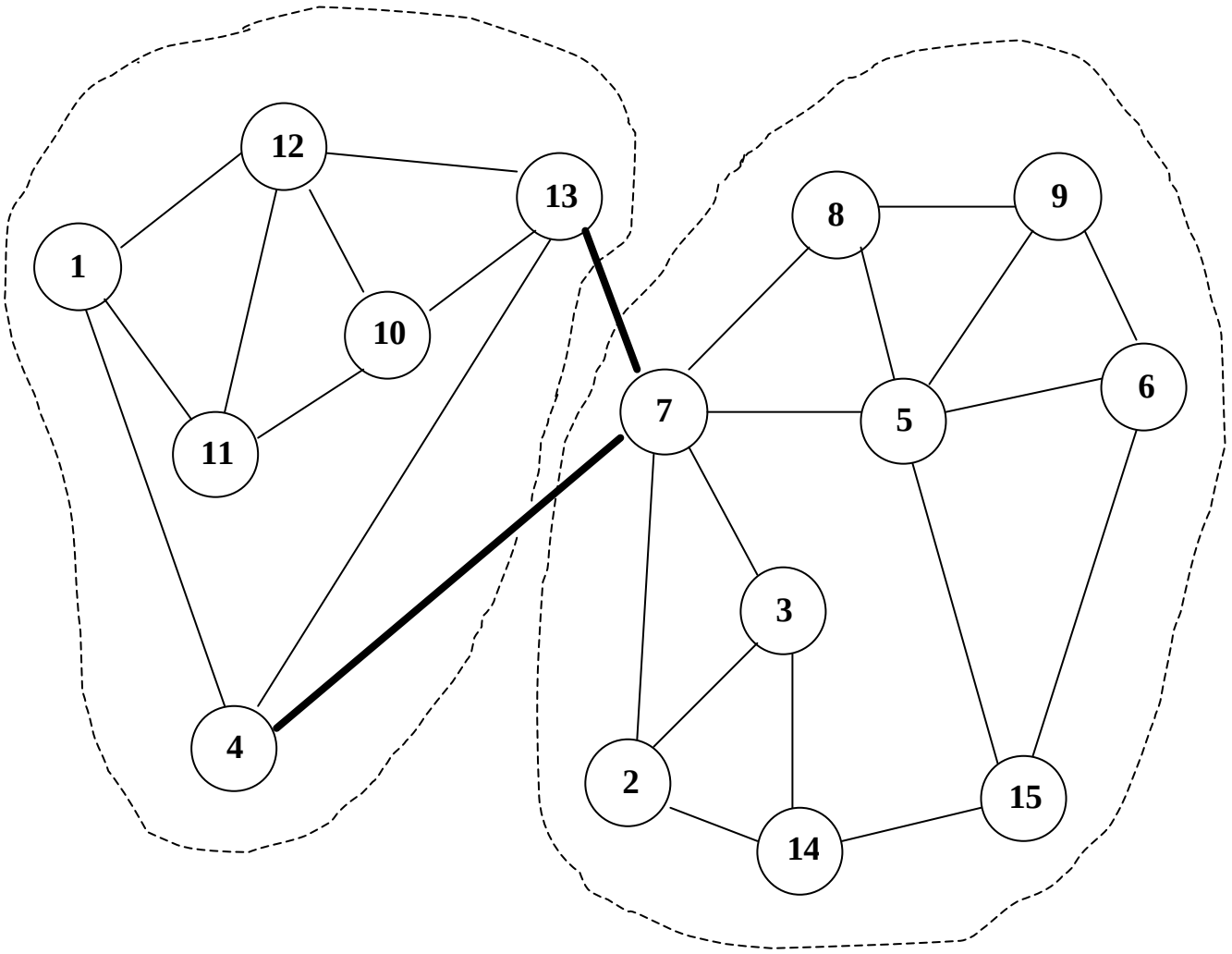
Il minimo numero di archi che disconnette il grafo e' 2

Il taglio minimo formato dall'arc connectivity e' costituito dai seguenti insiemi di nodi :

Ns = { 1 4 10 11 12 13 }
Nt = { 2 3 5 6 7 8 9 14 15 }

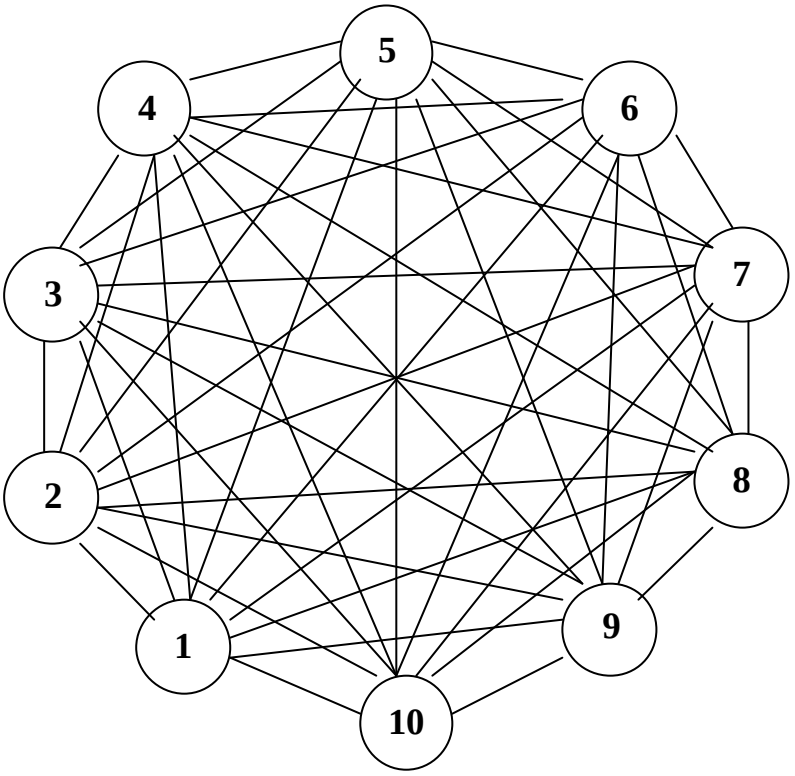
premere un tasto per finire programma

Otteniamo, in tal modo, l'arc connectivity - costituito dagli archi più marcati - che dà origine al taglio minimo formato dagli insiemi di nodi racchiusi tra le linee tratteggiate.



ESEMPIO DI GRAFO COMPLETO

Verifichiamo il comportamento dell'algoritmo nel caso di trattamento di un grafo completo.



INDIVIDUAZIONE DEL MINIMO NUMERO DI ARCHI CHE DISCONNETTONO UN GRAFO

CARICAMENTO DEL GRAFO

Immettere il numero di nodi del grafo : 10
immetti i nodi adiacenti al nodo 1 (0 = fine)
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 2 (0 = fine)
-> 1
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 3 (0 = fine)
-> 1
-> 2
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 0
immetti i nodi adiacenti al nodo 4 (0 = fine)
-> 1
-> 2
-> 3
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
->0
immetti i nodi adiacenti al nodo 5 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 6
-> 7
-> 8
-> 9

-> 10
-> 0
immetti i nodi adiacenti al nodo 6 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 7 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 8 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 9 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 10
-> 0
immetti i nodi adiacenti al nodo 10 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 0

Il nodo di grado minimo e' il numero 1

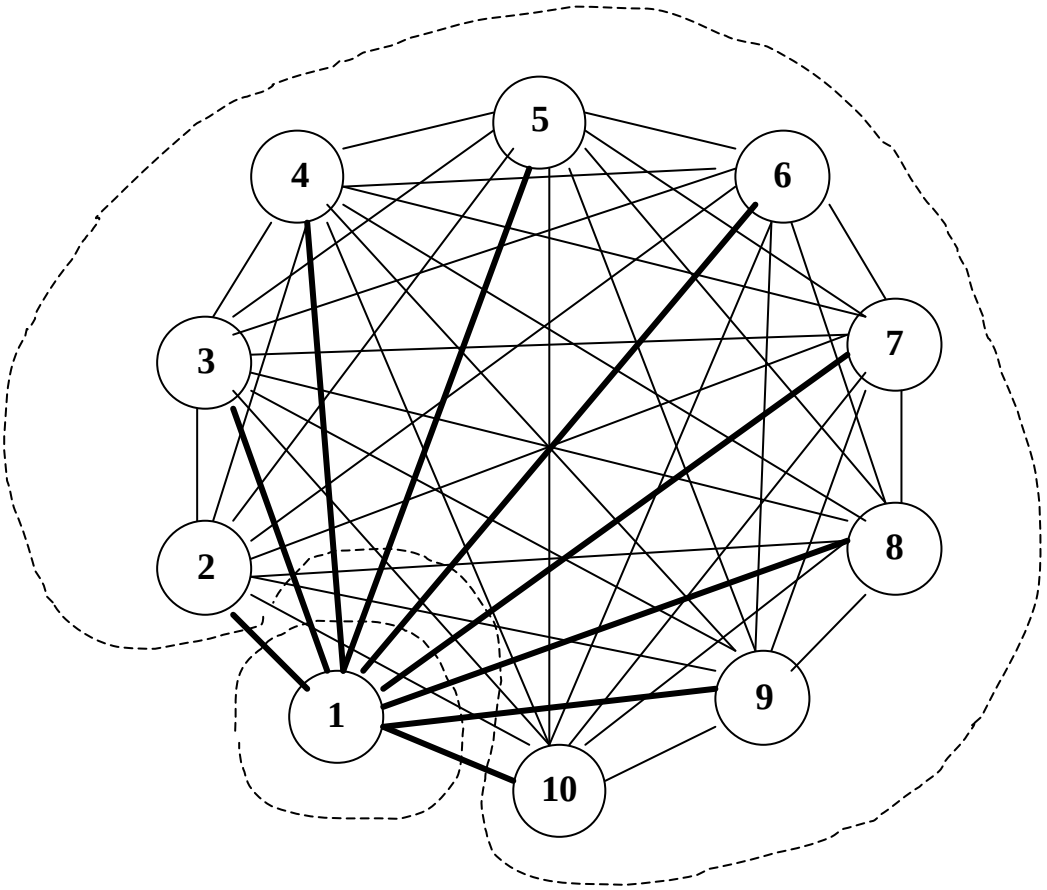
Individuazione del minimo disconnecting set

Il minimo numero di archi che disconnette il grafo e' 9

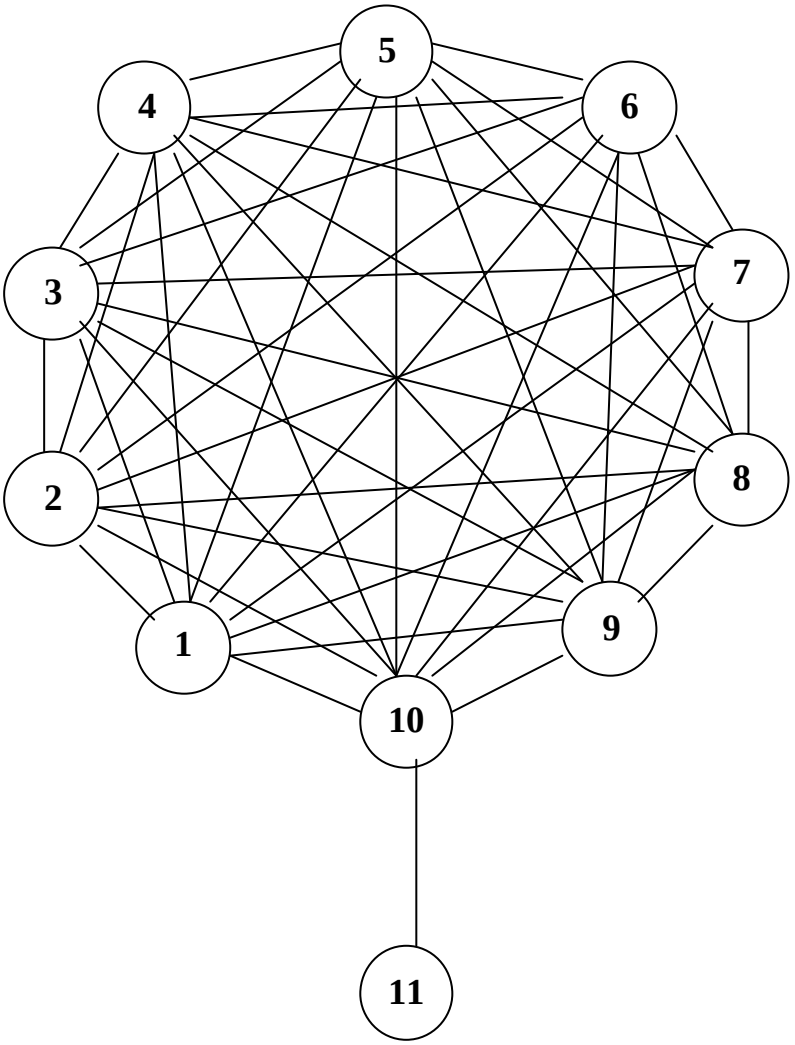
Il taglio minimo formato dall'arc connectivity e' costituito dai seguenti insiemi di nodi:

Ns = { 1 }
Nt = { 2 3 4 5 6 7 8 9 10 }

premere un tasto per finire programma



ESEMPIO DI GRAFO COMPLETO CON UN'APPENDICE



INDIVIDUAZIONE DEL MINIMO NUMERO DI ARCHI CHE DISCONNETTONO UN GRAFO

CARICAMENTO DEL GRAFO

Immettere il numero di nodi del grafo : 11
immetti i nodi adiacenti al nodo 1 (0 = fine)
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 2 (0 = fine)
-> 1
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 3 (0 = fine)
-> 1
-> 2
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 0
immetti i nodi adiacenti al nodo 4 (0 = fine)
-> 1
-> 2
-> 3
-> 5
-> 6
-> 7
-> 8
-> 9
-> 10
->0
immetti i nodi adiacenti al nodo 5 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 6
-> 7
-> 8
-> 9

-> 10
-> 0
immetti i nodi adiacenti al nodo 6 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 7
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 7 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 8
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 8 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 9
-> 10
-> 0
immetti i nodi adiacenti al nodo 9 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 10
-> 0
immetti i nodi adiacenti al nodo 10 (0 = fine)
-> 1
-> 2
-> 3
-> 4
-> 5
-> 6
-> 7
-> 8
-> 9
-> 11

-> 0
immetti i nodi adiacenti al nodo 11 (0 = fine)
-> 10
-> 0

Il nodo di grado minimo e' il numero 11

Individuazione del minimo disconnecting set

Il minimo numero di archi che disconnette il grafo e' 1

Il taglio minimo formato dall'arc connectivity e' costituito dai seguenti insiemi di nodi:

Ns = { 11 }
Nt = { 1 2 3 4 5 6 7 8 9 10 }

premere un tasto per finire programma