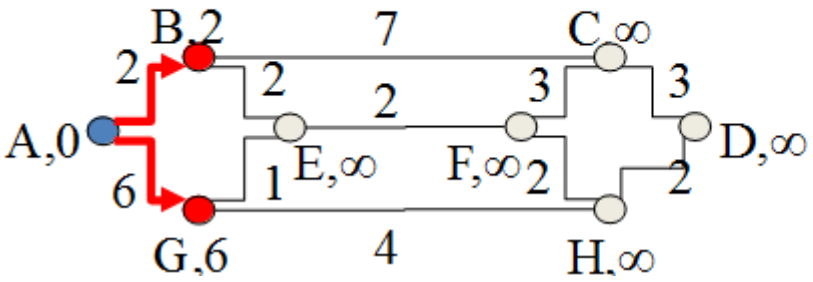
TFA 2014/15
SISTEMI E RETI DI CALCOLATORI PER L'INSEGNAMENTO
UNITA' DIDATTICA:
ALGORITMI DI ROUTING
DIJKSTRA

28/05/2015

Laura Ricci

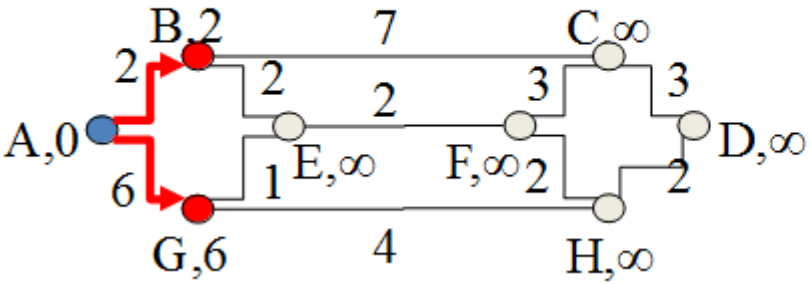
L'ALGORITMO DI BELLMAN-FORD

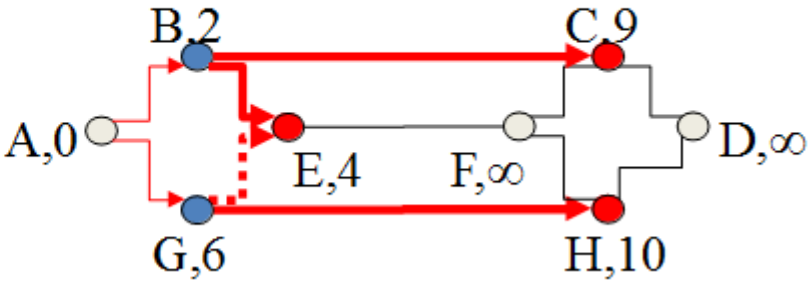
Bold = this iteration, Dashed = rejected



L'ALGORITMO DI BELLMAN-FORD

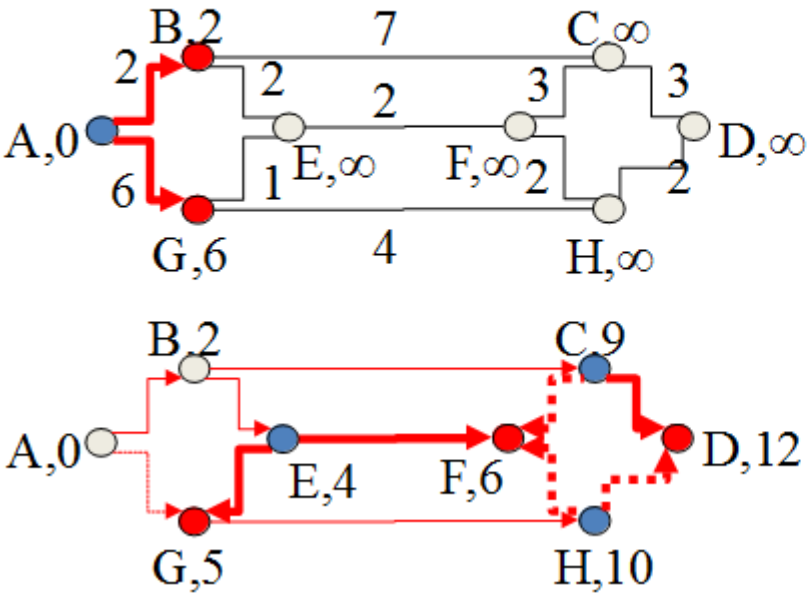
Bold = this iteration, Dashed = rejected

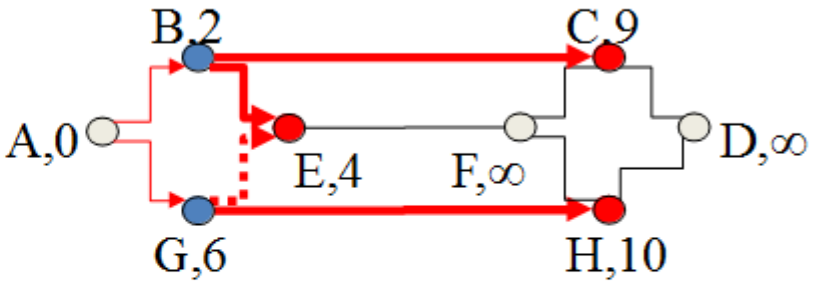




L'ALGORITMO DI BELLMAN-FORD

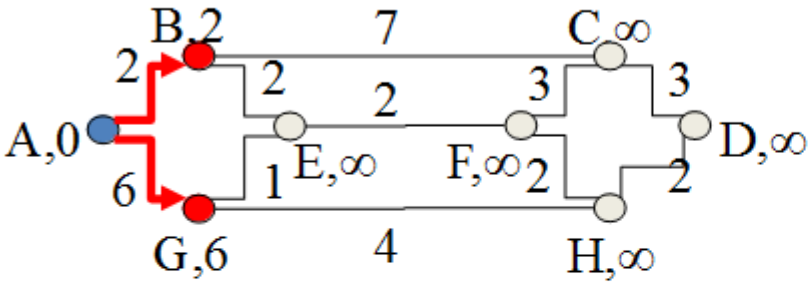
Bold = this iteration, Dashed = rejected

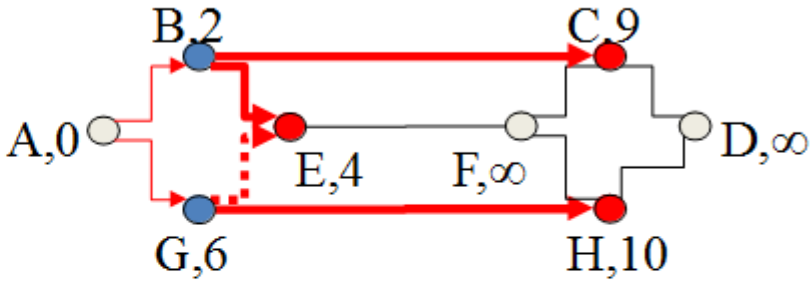


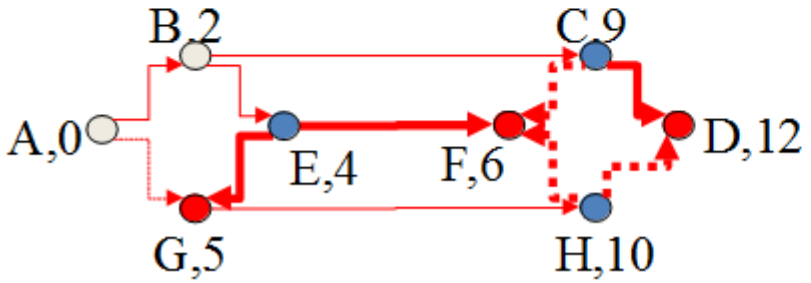


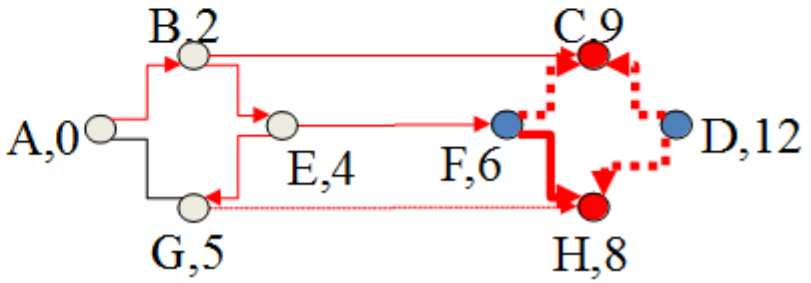
L'ALGORITMO DI BELLMAN-FORD

Bold = this iteration, Dashed = rejected



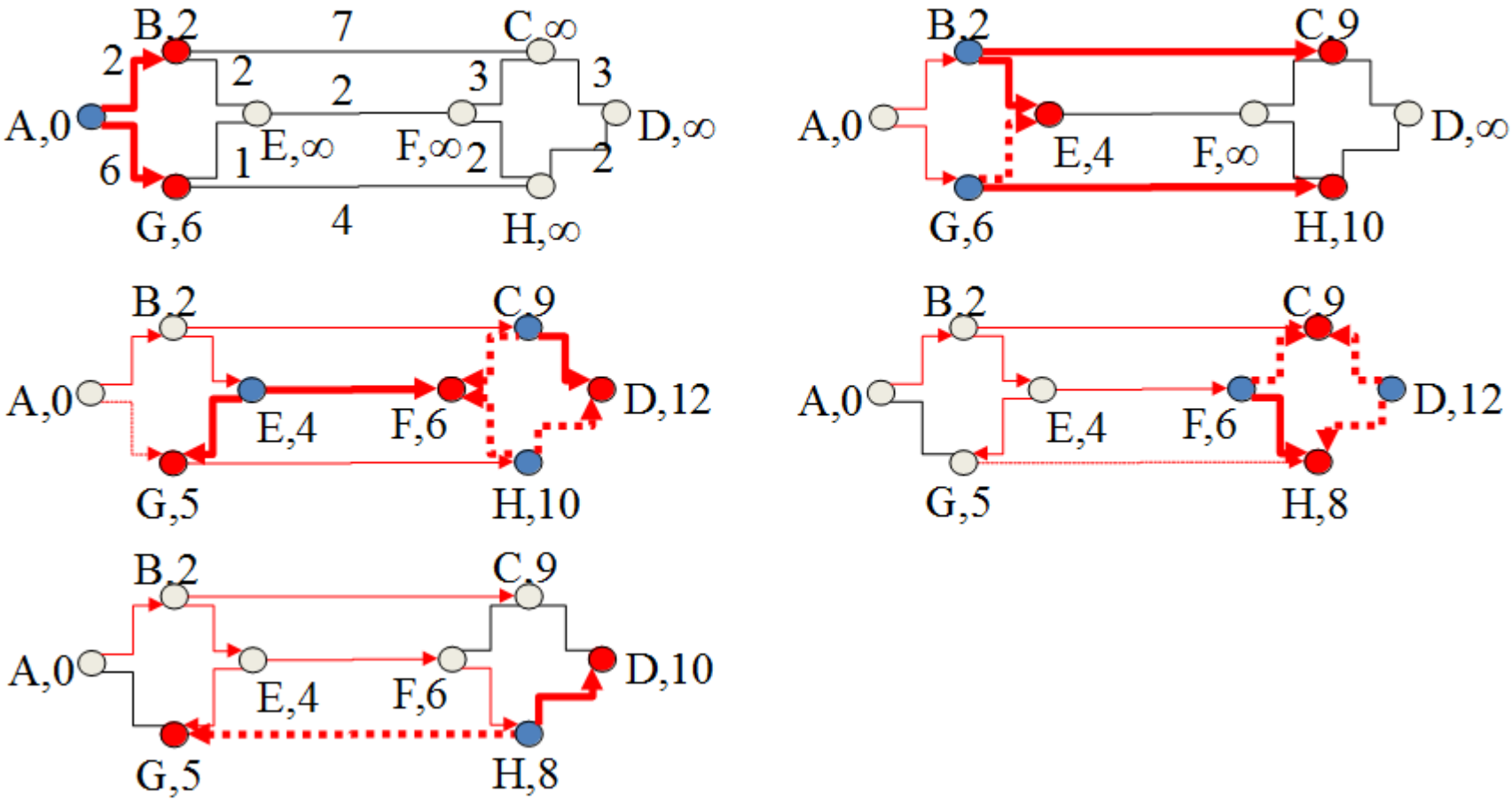






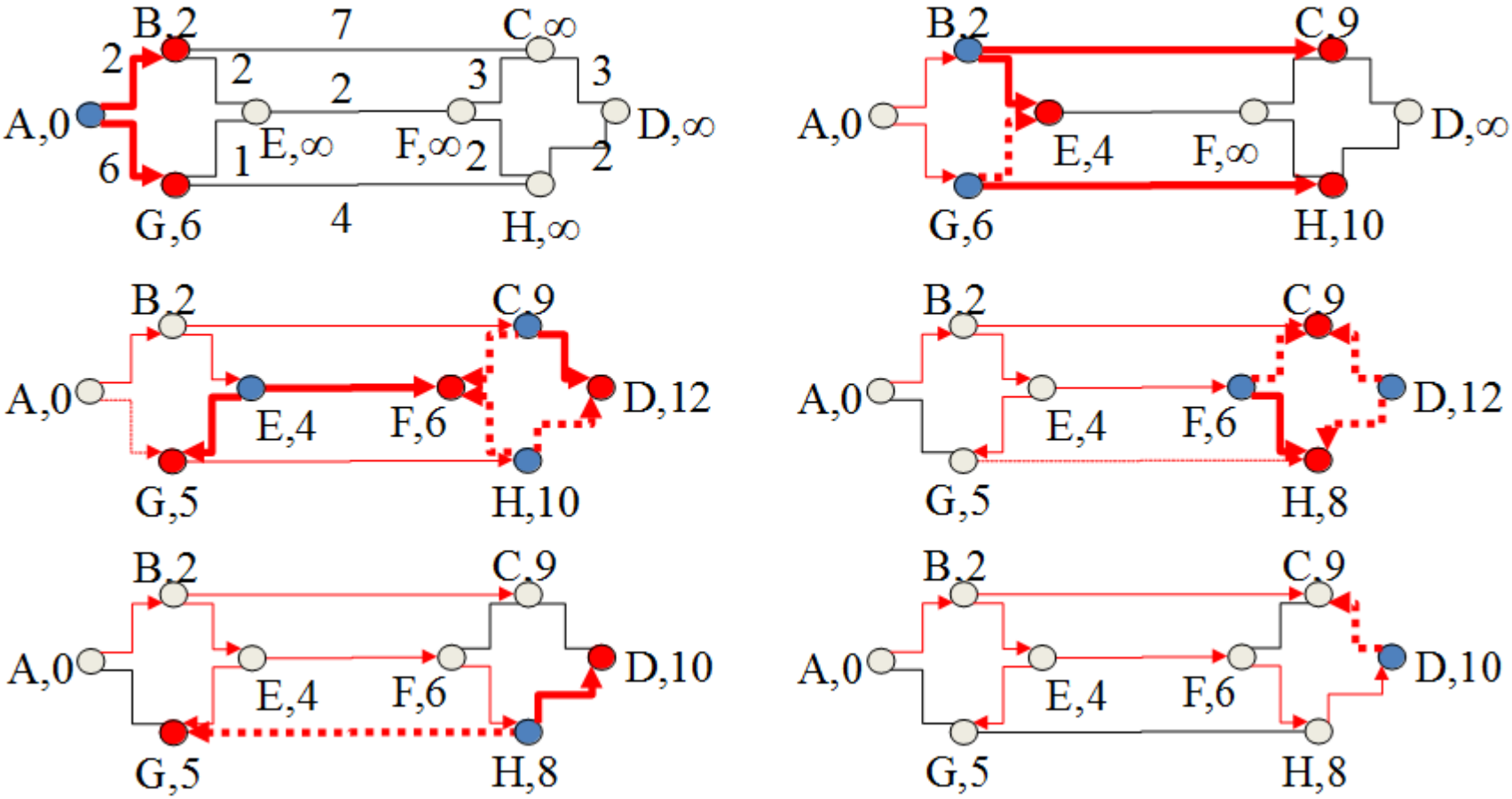
L'ALGORITMO DI BELLMAN-FORD

Bold = this iteration, Dashed = rejected



L'ALGORITMO DI BELLMAN-FORD

Bold = this iteration, Dashed = rejected



L'ALGORITMO DI DIJKSTRA

- Principi base: l'osservazione di partenza è che un percorso da un nodo sorgente s ad un nodo destinazione t che include un nodo intermedio i è ottimo solo se il sottopercorso da *s* a *i* è a sua volta percorso ottimo tra *s* e *i*.

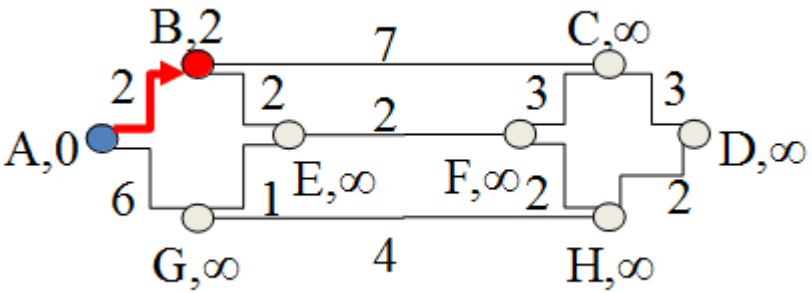
- Algoritmo greedy
- mantiene un insieme di nodi già analizzati ed un insieme di nodi da analizzare
- calcola, per ogni nodo già analizzato, tutti i cammini che passano per quel nodo e che raggiungono un nodo nell'insieme non ancora analizzato
- considerano il minimo tra quei cammini ed aggiunge il nodo raggiunto a quelli già analizzati

L'ALGORITMO DI DIJKSTRA

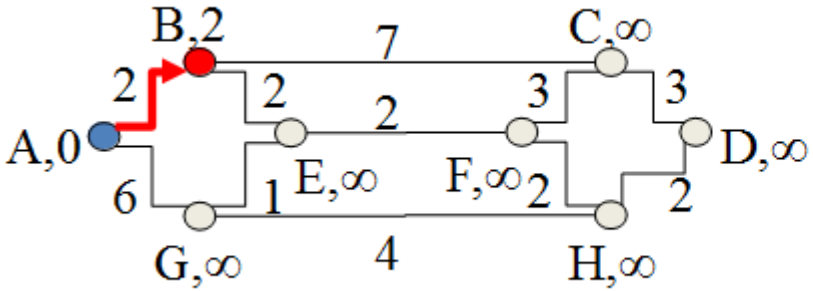
- siano i nodi numerati 0..N; l'algoritmo calcola i cammini migliori a partire dal nodo 0.
- `c(i,j)` costo dell'arco `(i,j)`
- `pred(i)` predecessore di i nell'albero che viene costruito
- `m(j)` distanza del nodo j dal nodo 0.
- `M`: insieme dei nodi già considerati, a cui è già stata assegnata una distanza dall'origine.

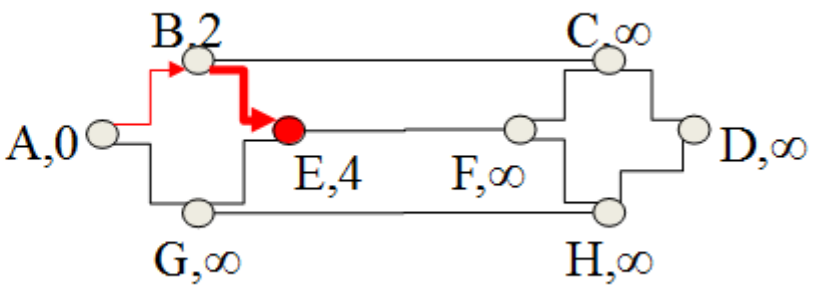
```
m(0) = 0; M = {0};
for k=1 to N {
   find (i0, j0) that minimizes m(i) + c(i,j),
                        with i in M, j not in M
   m(j0) = m(i0) + c(i0, j0)
   pred(j0) = i0
   M = M ∪ {j0}
}
```

L'ALGORITMO DI DIJKSTRA

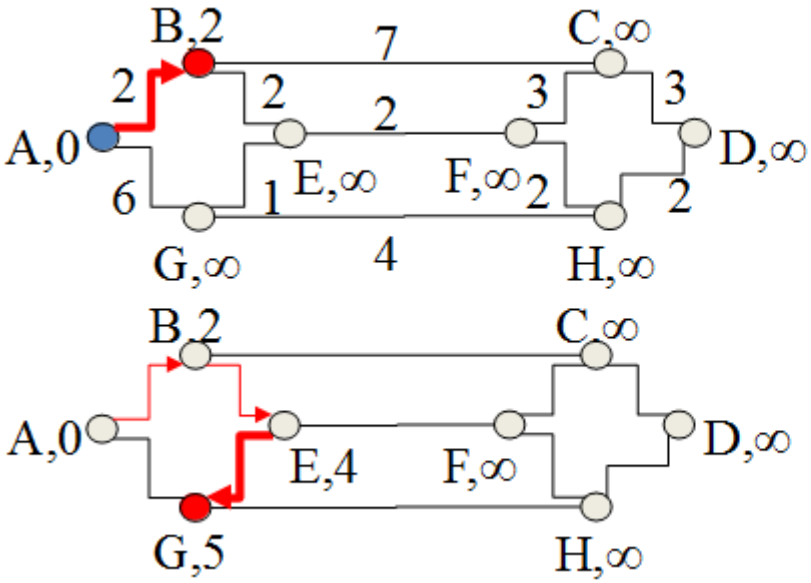


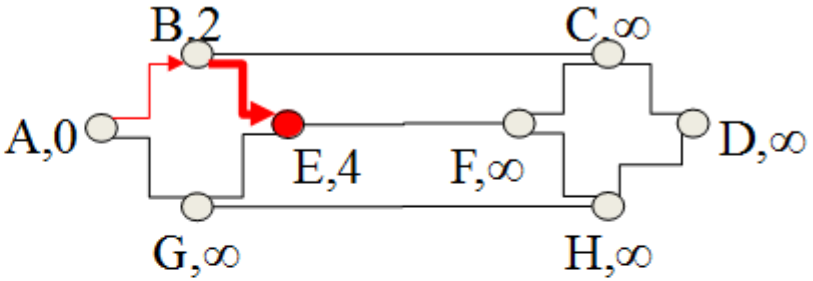
L'ALGORITMO DI DIJKSTRA



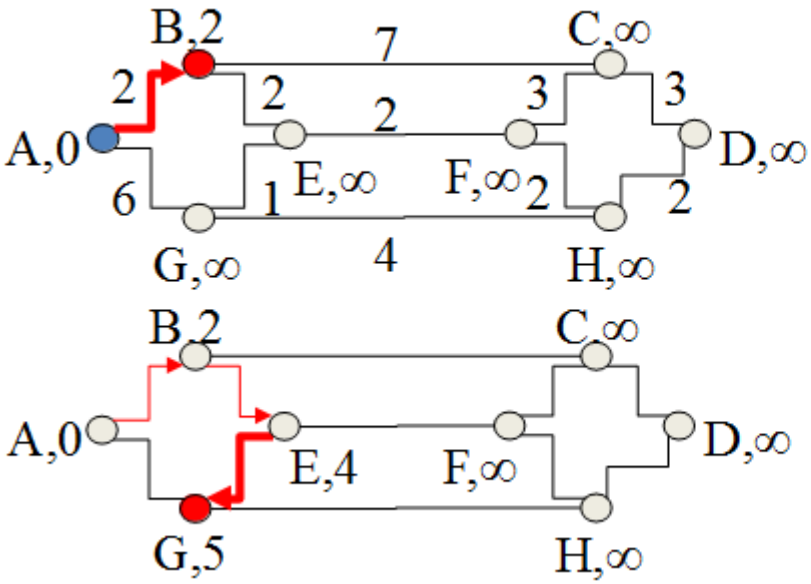


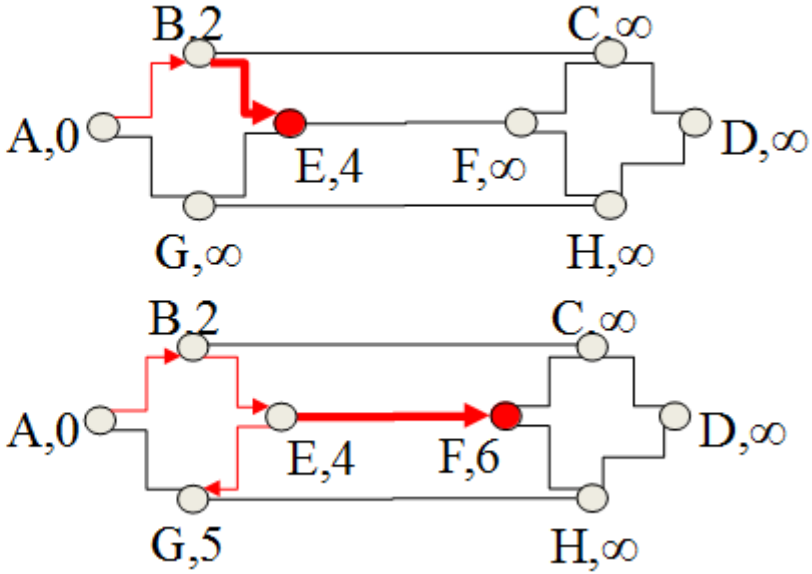
L'ALGORITMO DI DIJKSTRA



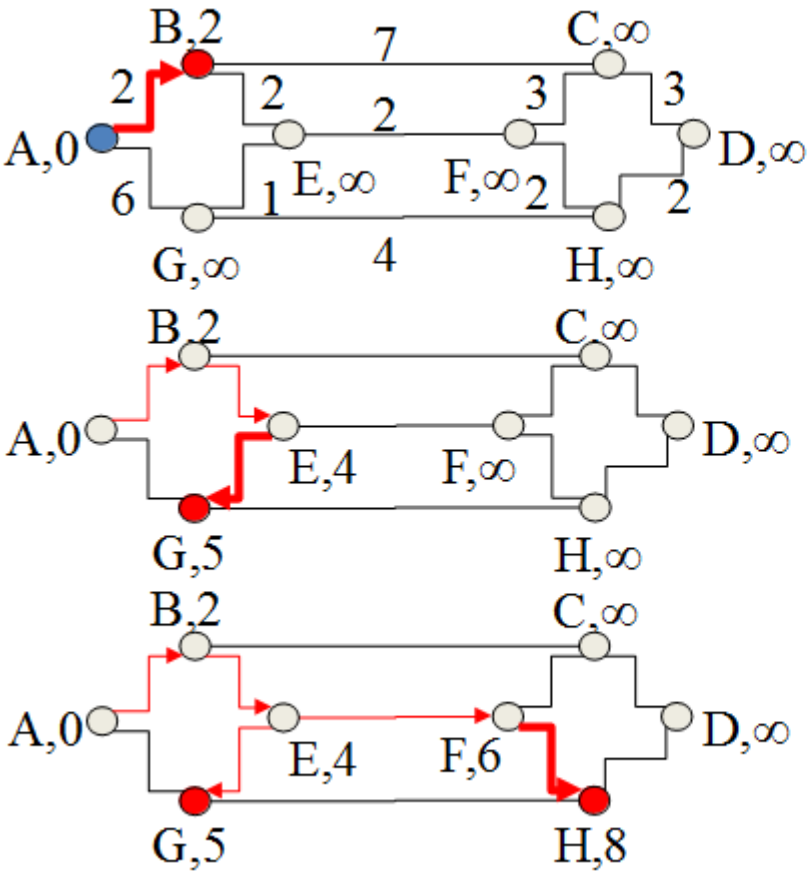


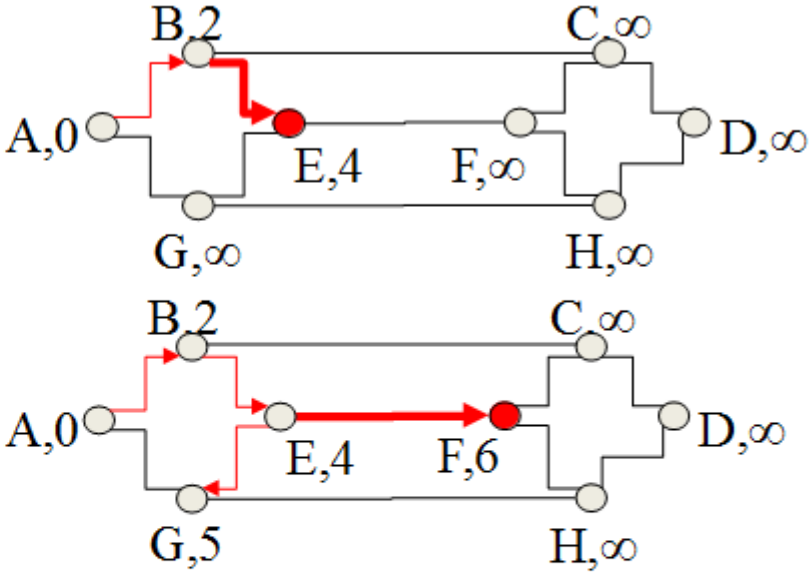
L'ALGORITMO DI DIJKSTRA



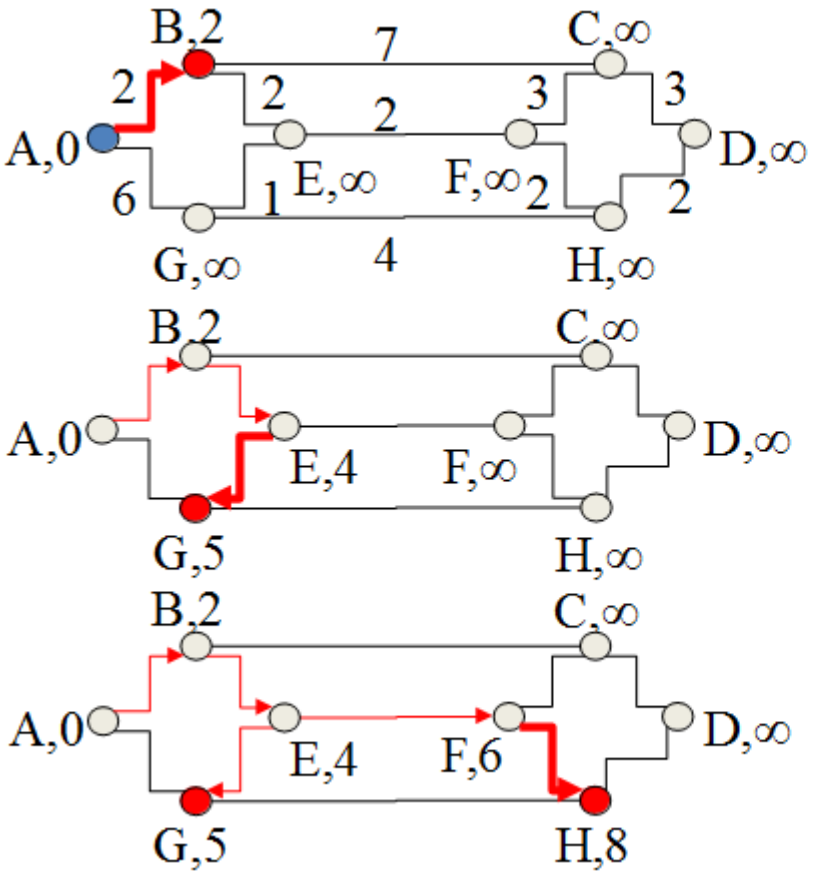


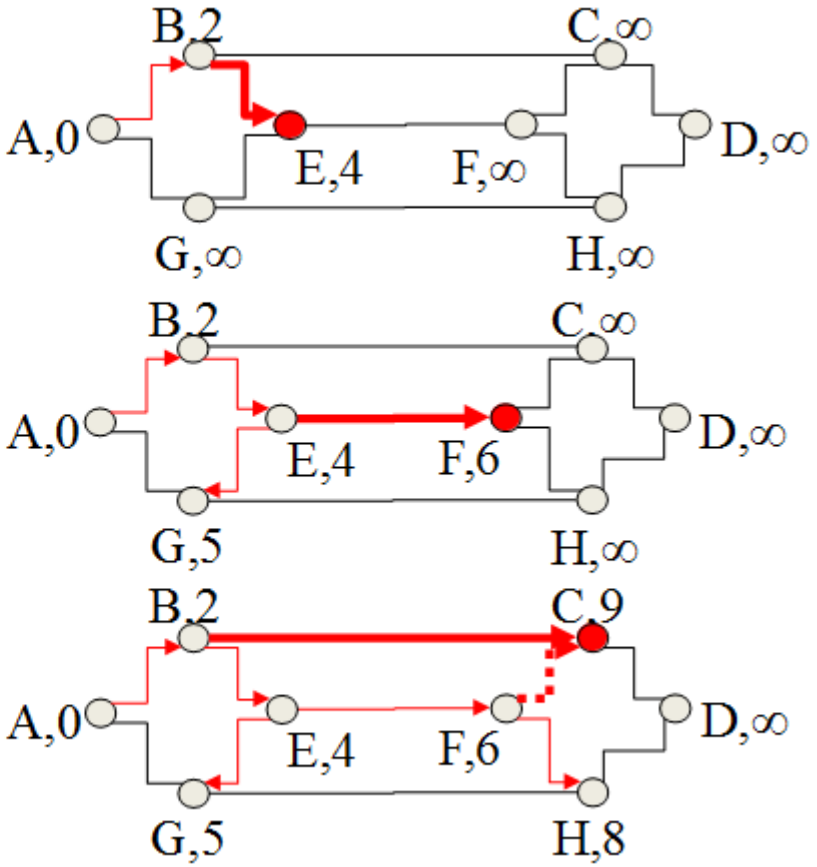
L'ALGORITMO DI DIJKSTRA



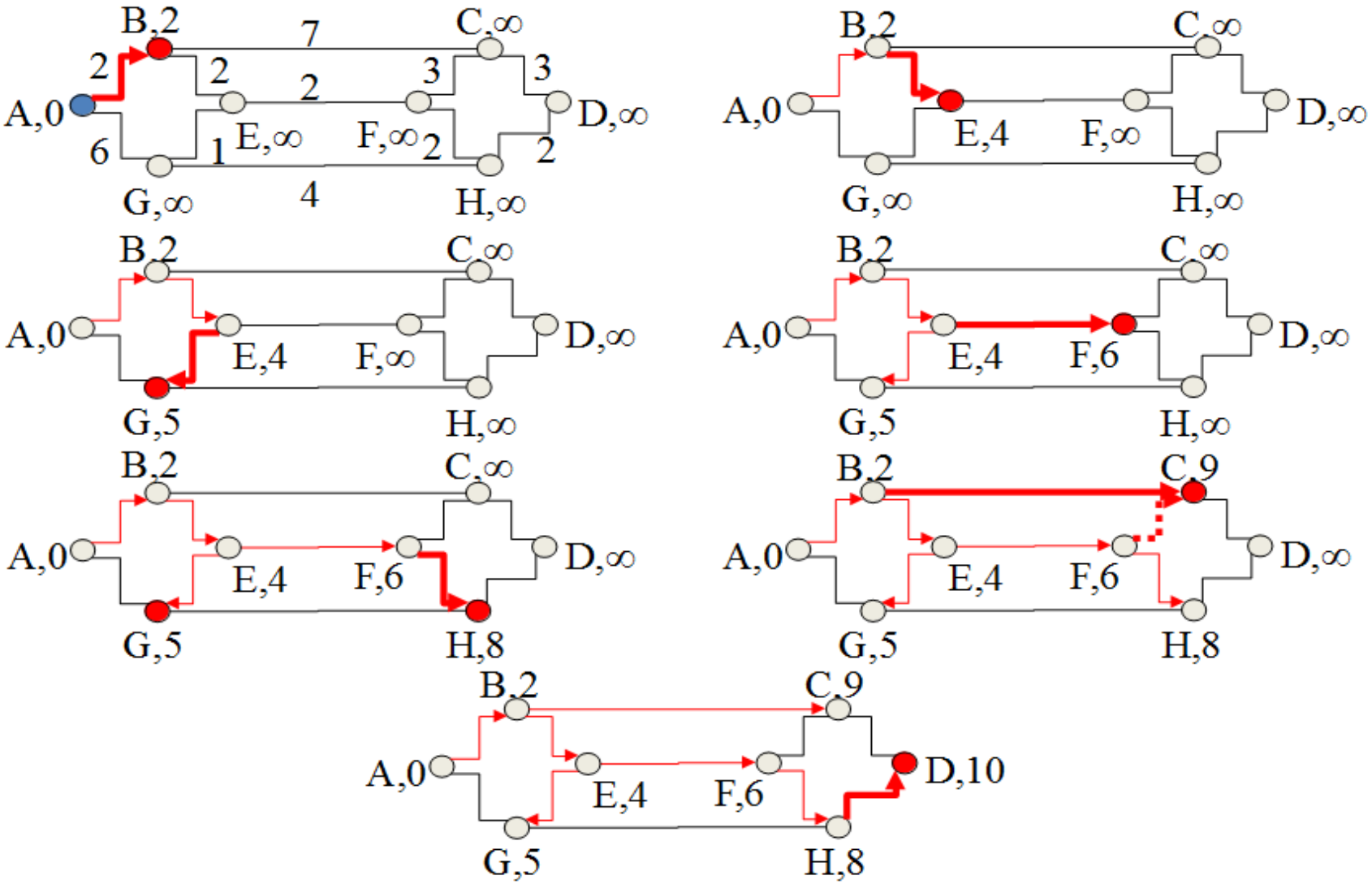


L'ALGORITMO DI DIJKSTRA





L'ALGORITMO DI DIJKSTRA



OSPF: OPEN SHORTEST PATH FIRST

- il protocollo di routing maggiormente utilizzato

- storia

  1989: RFC 1131 OSPF versione 1

  1991: RFC 1247 OSPF versione 2

  1994: RFC 1583 OSPF versione 2 (rivista)

  1997: RFC 2178 OSPF versione 2 (rivista)

  1998: RFC 2328 OSPF versione 2 (versione corrente)

OSPF: OPEN SHORTEST PATH FIRST

- ogni router conosce lo stato delle linee (costi) che lo collegano ai nodi vicini
- ogni router invia a tutti gli altri router le informazioni relative ai links ad esso connessi (link state advertisement)
- quando un router ha ricevuto lo stato dell'intera rete, crea una mappa completa della topologia della rete
- ogni router mantiene le informazioni ricevute in una struttura dati, il link-state database (perchè mantiene informazioni sullo stato dei links dell'intera rete)
- quando un nuovo router viene inserito nella rete, l'informazione viene aggiornata.
- i diversi routers possono avere temporaneamente diverse visioni della rete, ma le diverse visioni convergono mediante l'invio degli advertisements

OSPF: OPEN SHORTEST PATH FIRST

- ogni router R
  - utilizza il suo link state database per calcolare il cammino di minor costo che lo collega ad un qualsiasi altro router del sistema autonomo
  - crea uno shortest path tree (SPT) indica il cammino minimo da R ad un qualsiasi altro router del sistema autonomo
  - l'albero contiene la stessa informazione contenuta nel database, ma vista 'dal punto di vista' di R
- Lo SPT viene modificato dinamicamente se si ricevono nuovi advertisements che notificano la modifica della topologia della rete

•Ogni router applica l'algoritmo di Dijkstra per il calcolo dei cammini minimi per il calcolo dello SPT

OSPF: OPEN SHORTEST PATH FIRST

- Each router knows directly connected networks

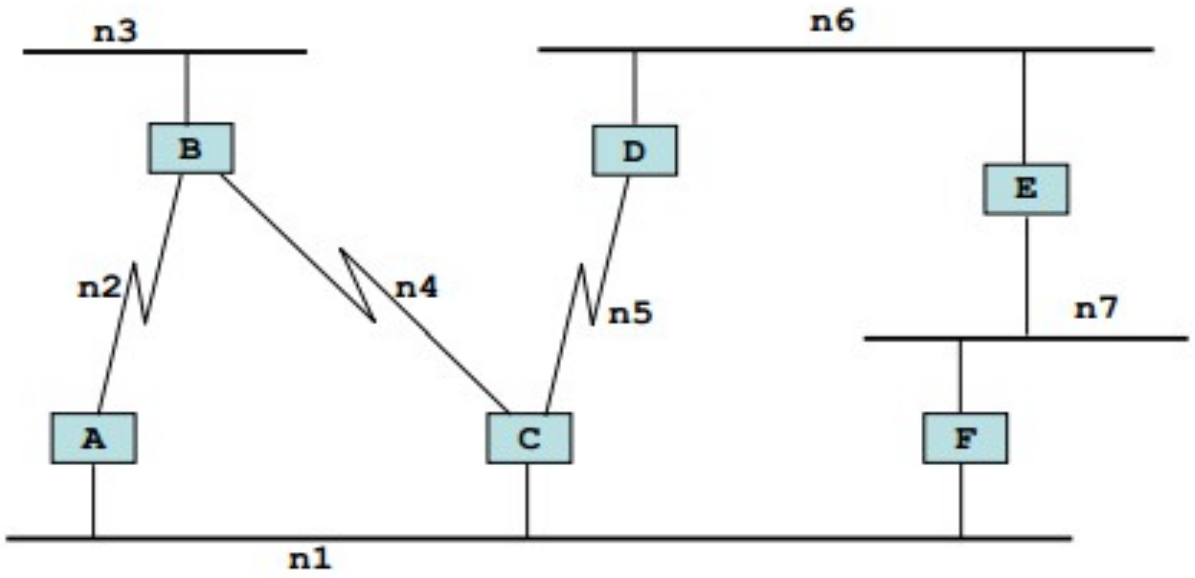


TABELLE DI ROUTING INIZIALI

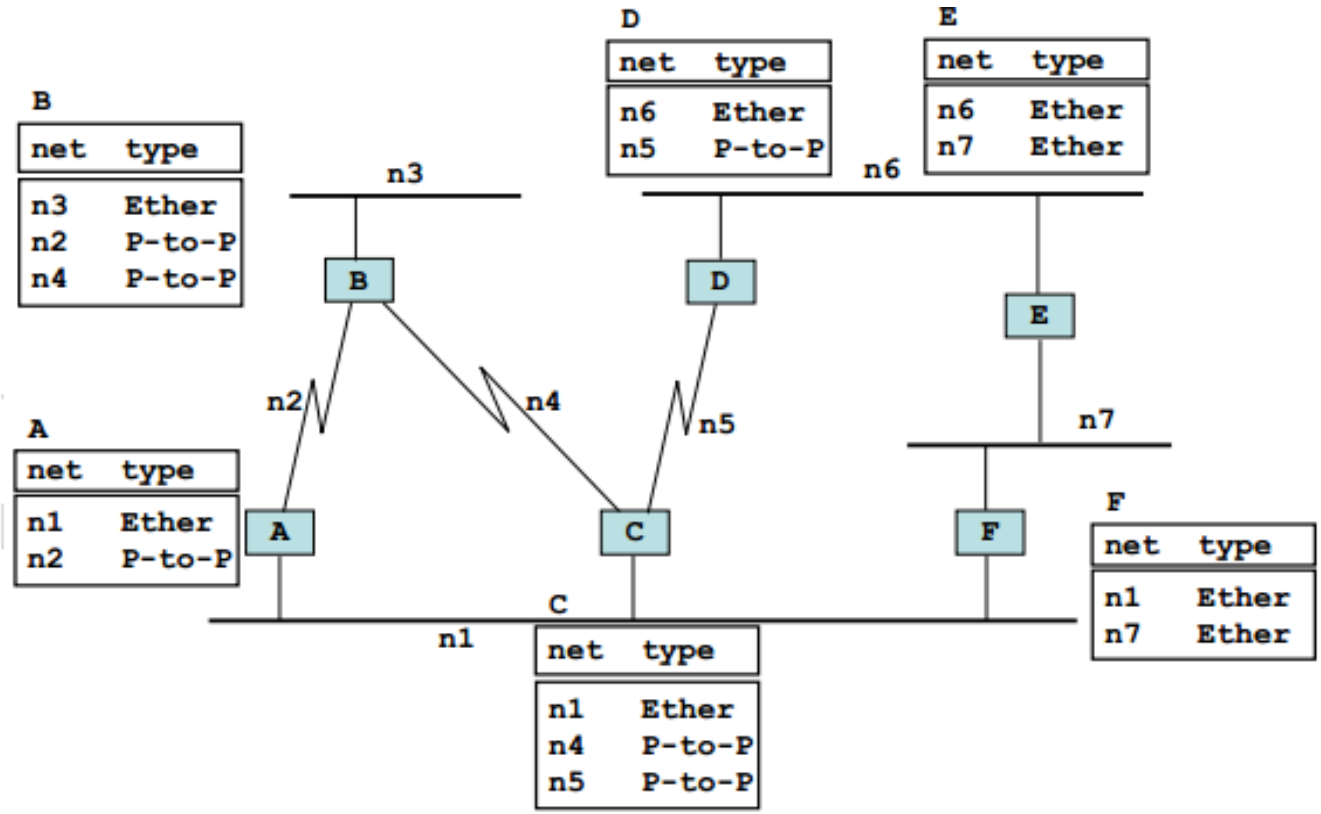
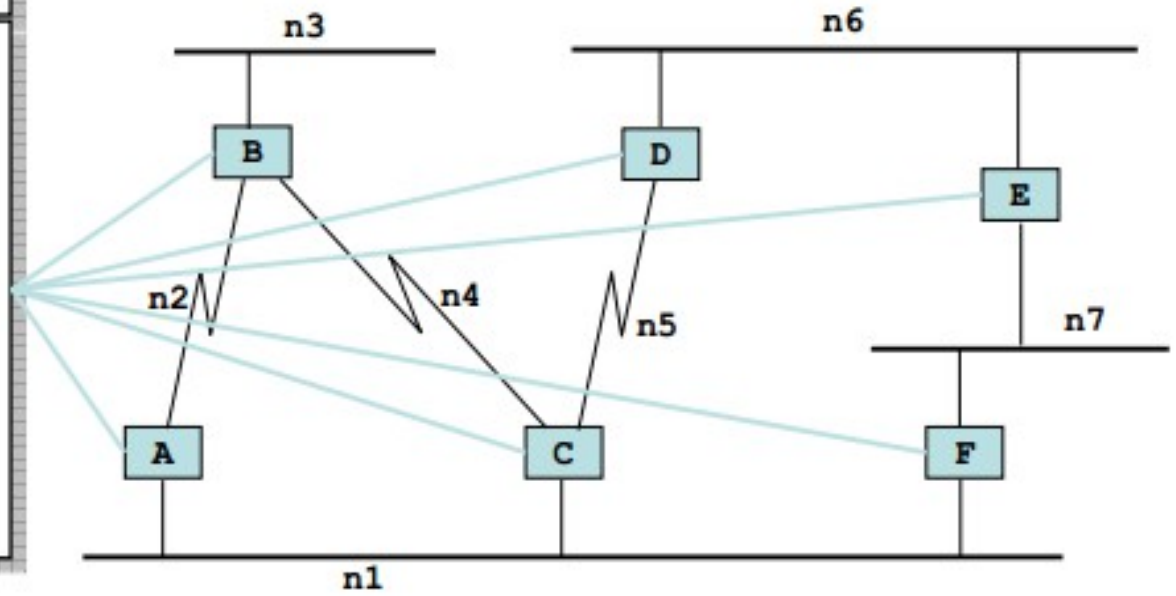


TABELLE DI ROUTING INIZIALI

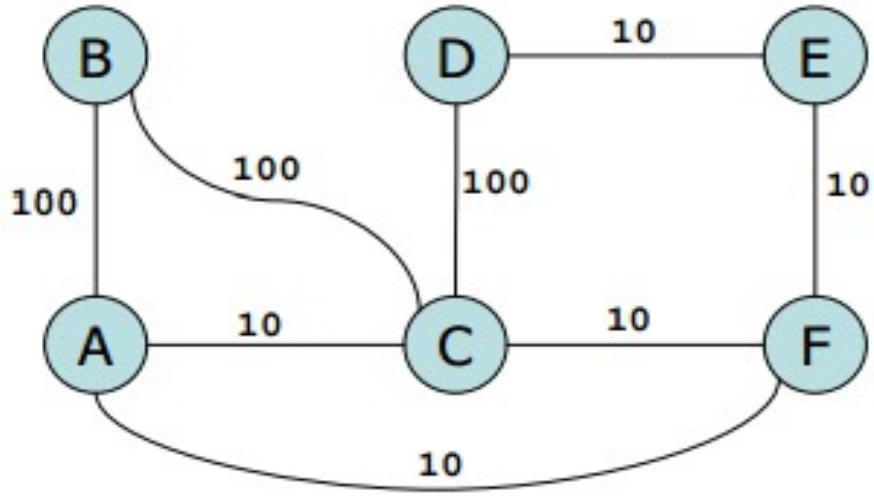
- The local metric information is flooded to all routers
- After convergence, all routers have the same information

rtr	net	cost
A	n1	10
A	n2	100
B	n3	10
B	n2	100
B	n4	100
C	n1	10
C	n4	100
C	n5	100
D	n6	10
D	n5	100
E	n6	10
E	n7	10
F	n1	10
F	n7	10

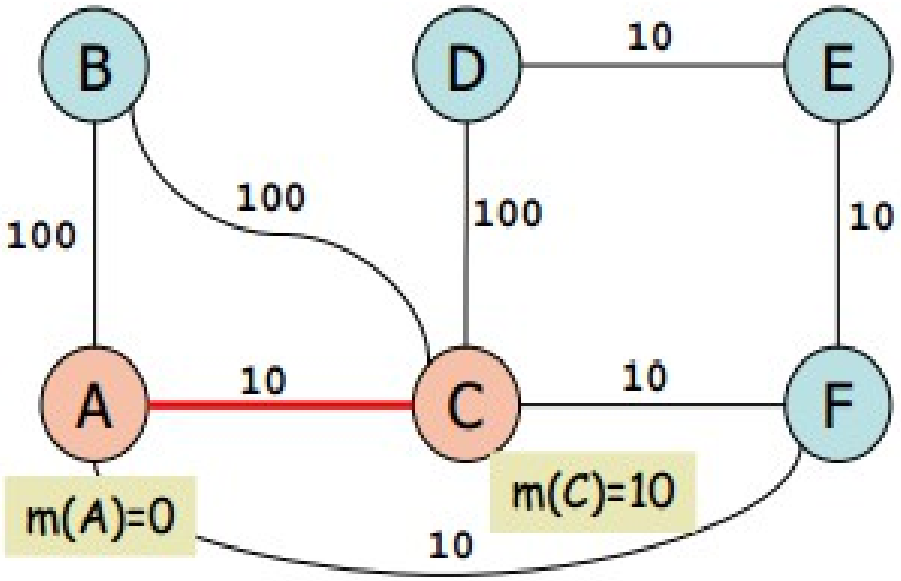


IL GRAFO SEMPLIFICATO

- Only arrows with metrics between routers
- Every node executes the shortest path computation on the graph – same graph, but different sources



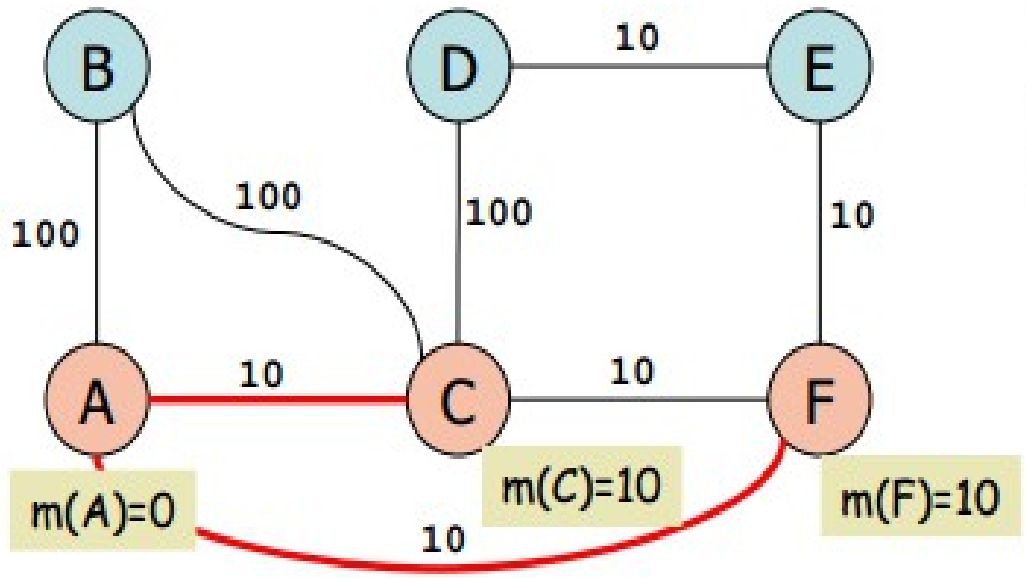
ESEMPIO: DIJKSTRA IN *A*



init: M = { A }

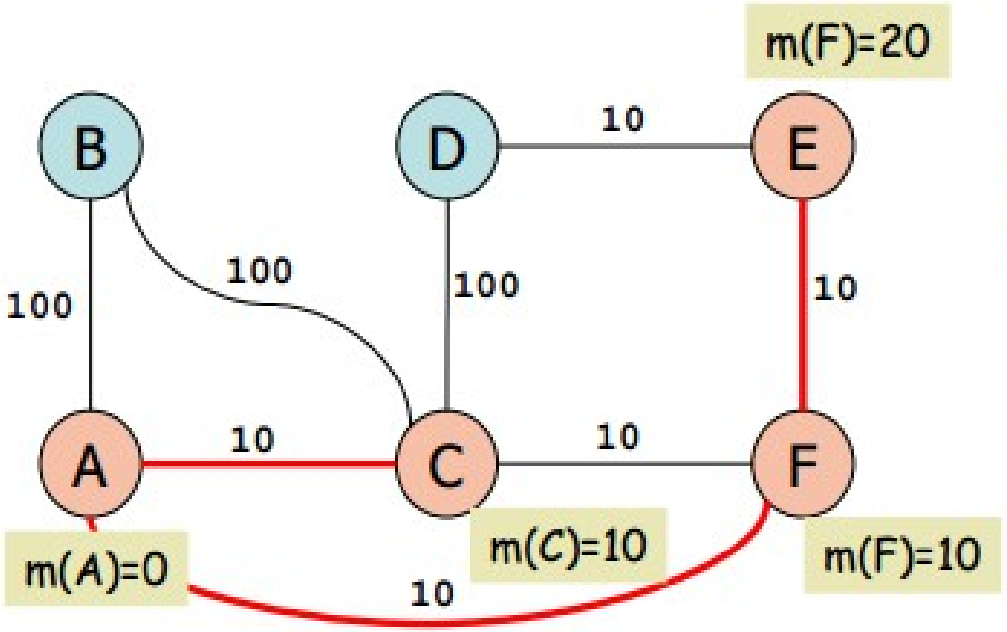
step 1:
i0=A
j0=C
m(C)=10
M = {A, C}

ESEMPIO: DIJKSTRA IN *A*

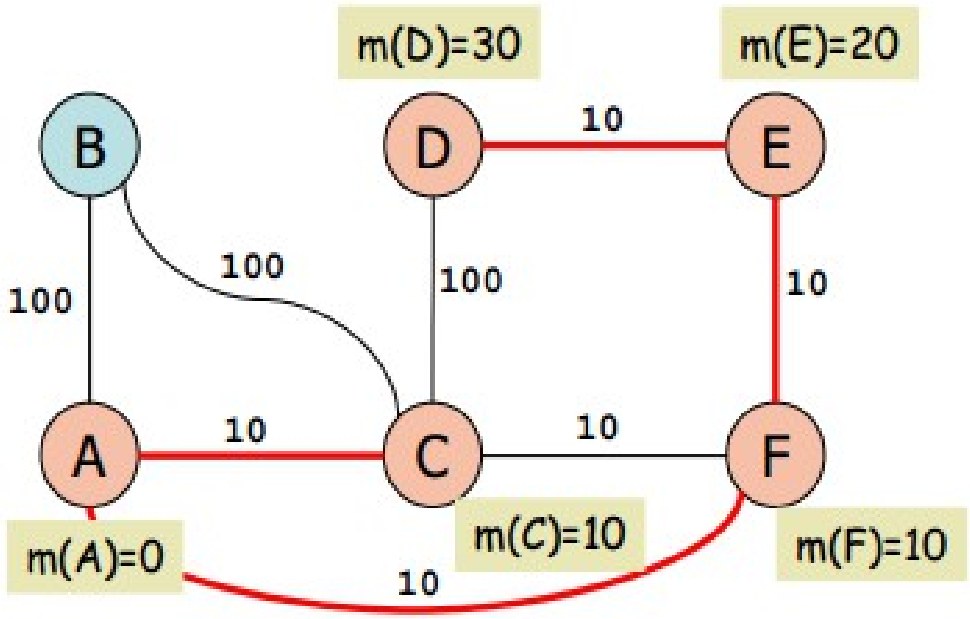


i0=*A*
j0=F
m(F)=10
M = {*A*,*C*,F}

ESEMPIO: DIJKSTRA IN *A*

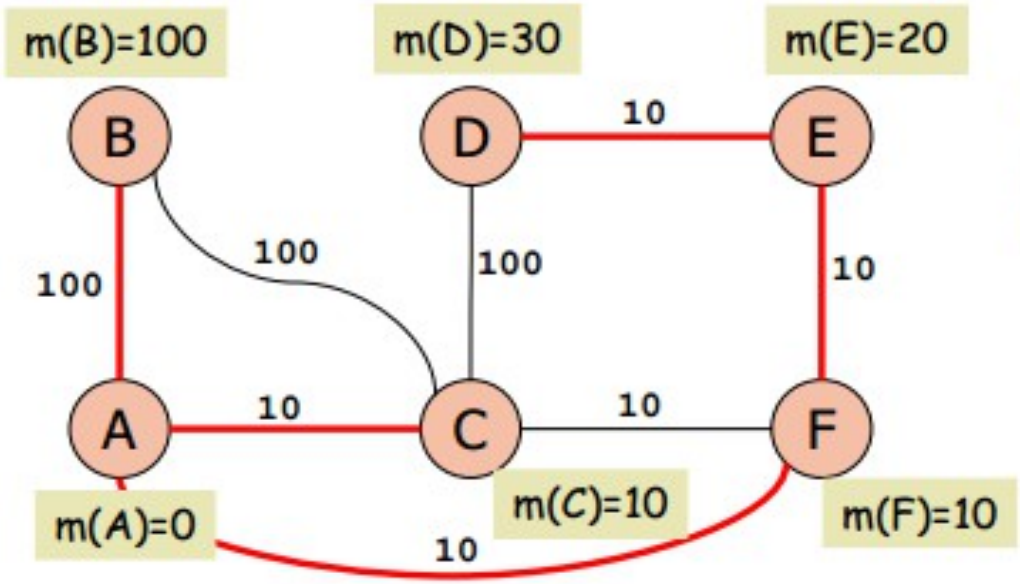


ESEMPIO: DIJKSTRA IN *A*



i0=E
j0=D
m(D)=40
M = {A,C,F,E,D}

ESEMPIO: DIJKSTRA IN A

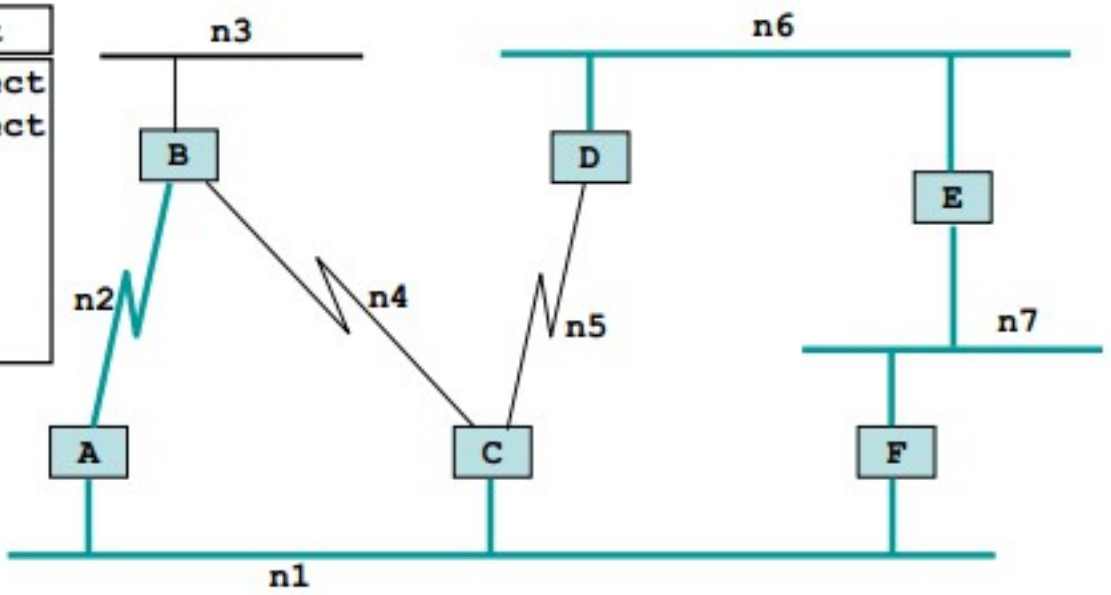


i0=A
j0=B
m(B)=100
M = {A,C,F,E,D,B}

ROUTING TABLE DI A

A

net	next
n1	direct
n2	direct
n3	B
n4	C
n5	C
n6	F
n7	F



LINK STATE: SOMMARIO

- algoritmi di routing basati su Dijkstra: Link state algorithms
- tutti i nodi calcolano il loro database con la topologia della rete
- il database rappresenta l'intera rete
- fortemente sincronizzato
- tutti i nodi calcolano il loro albero con i migliori path verso tutte le destinazioni
- la tabelle di routing sono costruite a partire dall'albero ed utilizzate per individuare il next hop di routin
- Link state (LS) a confronto con distance vector (DV)
  - LS evita i problemi di convergenza di DV
  - LS più complesso di DV
  - LS maggior overhead di DV
  - LS centralizzato DV distribuito

LABORATORIO: ILTIPO DI DATO ASTRATTO GRAFO

```
public class WeightedGraph {
   private int [][]  edges;  // adjacency matrix
   private String [] labels; // etichette associate ai nodi
   public WeightedGraph (int n) {
          edges  = new int [n][n];
          labels = new String[n]; }

  public void setLabel (int vertex, String label)
                       {labels[vertex]=label; }

  public String getLabel (int vertex)
                       {return labels[vertex]; }

  public void addEdge(int source, int target, int w)
                       {edges[source][target] = w; }
```

LABORATORIO: ILTIPO DI DATO ASTRATTO GRAFO

```
public boolean isEdge (int source, int target)
                { return edges[source][target]>0; }
public void removeEdge (int source, int target)
                { edges[source][target] = 0; }
public int getWeight  (int source, int target)
                { return edges[source][target]; }
public int [] neighbors (int vertex) {
     int count = 0;
     for (int i=0; i<edges[vertex].length; i++) {
          if (edges[vertex][i]>0) count++;       }
     final int[]answer= new int[count];
     count = 0;
     for (int i=0; i<edges[vertex].length; i++) {
          if (edges[vertex][i]>0) answer[count++]=i;}
     return answer;}
```

LABORATORIO: ILTIPO DI DATO ASTRATTO GRAFO

```
public void print () {
        for (int j=0; j<labels.length; j++) {
           System.out.print (labels[j]+": ");
           for (int i=0; i<edges[j].length; i++) {
              if (edges[j][i]>0)
                  System.out.print (labels[i]+":"+edges[j][i]+" ");
           }
           System.out.println ();
        }
    }
```

LABORATORIO: ILTIPO DI DATO ASTRATTO GRAFO

```
public static void main (String args[])
     {final WeightedGraph t = new WeightedGraph (6);
      t.setLabel (0, "v0");
      t.setLabel (1, "v1");
      t.setLabel (2, "v2");
      t.setLabel (3, "v3");
      t.setLabel (4, "v4");
      t.setLabel (5, "v5");
      t.addEdge (0,1,2);
      t.addEdge (0,5,9);
      t.addEdge (1,2,8);
      t.addEdge (1,3,15);
      t.addEdge (1,5,6);
      t.addEdge (2,3,1);
      t.addEdge (4,3,3);
      t.addEdge (4,2,7); t.addEdge (5,4,3); t.print();}}
```