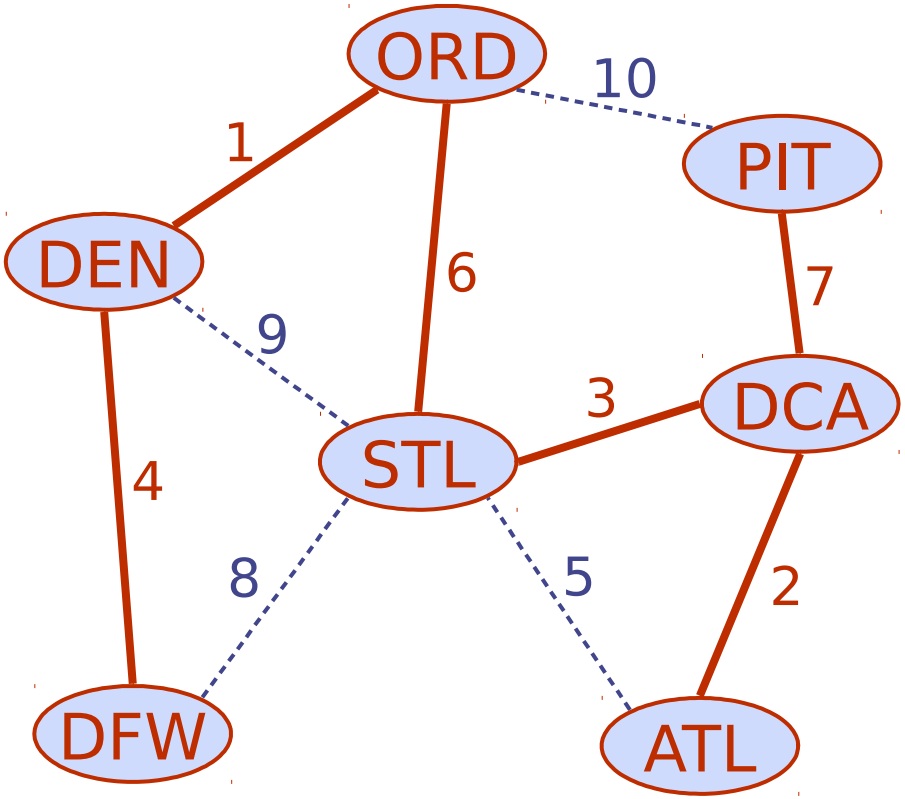
Alberi minimi ricoprenti (Minimum Spanning Tree)

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Albero minimo ricoprente

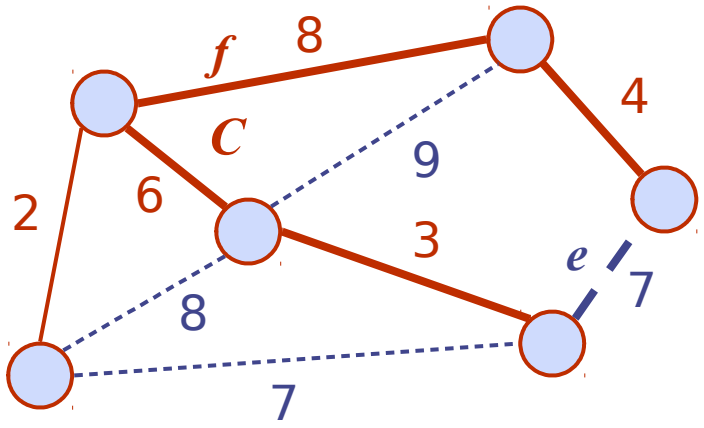
- Consideriamo grafi *non diretti*
- Sottografo ricoprente di G
  - Sottografo che contiene tutti i vertici di G
- Albero ricoprente
  - Qualunque albero che copre tutti i vertici di G
- Albero minimo ricoprente
  - Dato un grafo G (in generale pesato)
  - Albero ricoprente di peso totale minimo
- Applicazioni
  - Reti di comunicazione



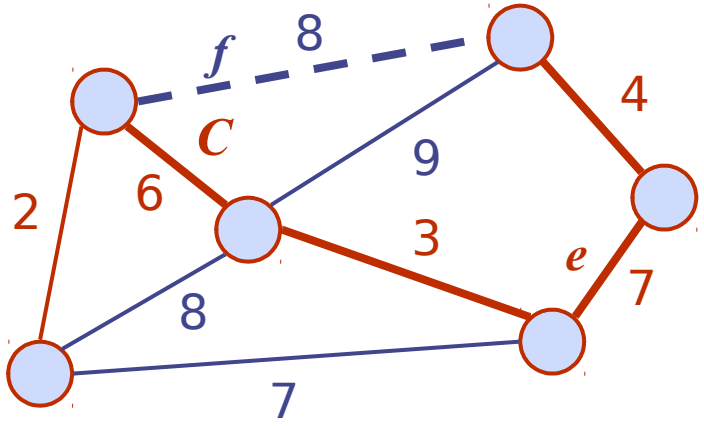
Proprietà

Proprietà di ciclo

- *T*: un MST di *G* *(albero peso minimo)*
- *e*: arco di *G* non in *T*
- *C*: ciclo presente in *T* U *{e}*
- Per ogni arco *f* di *C*
  - w(f) ≤ w(e)
- Prova
  - Per contraddizione
  - Se w(f) > w(e) → nuovo albero di peso



Sostituendo *e* a *f* abbiamo un albero di peso minore

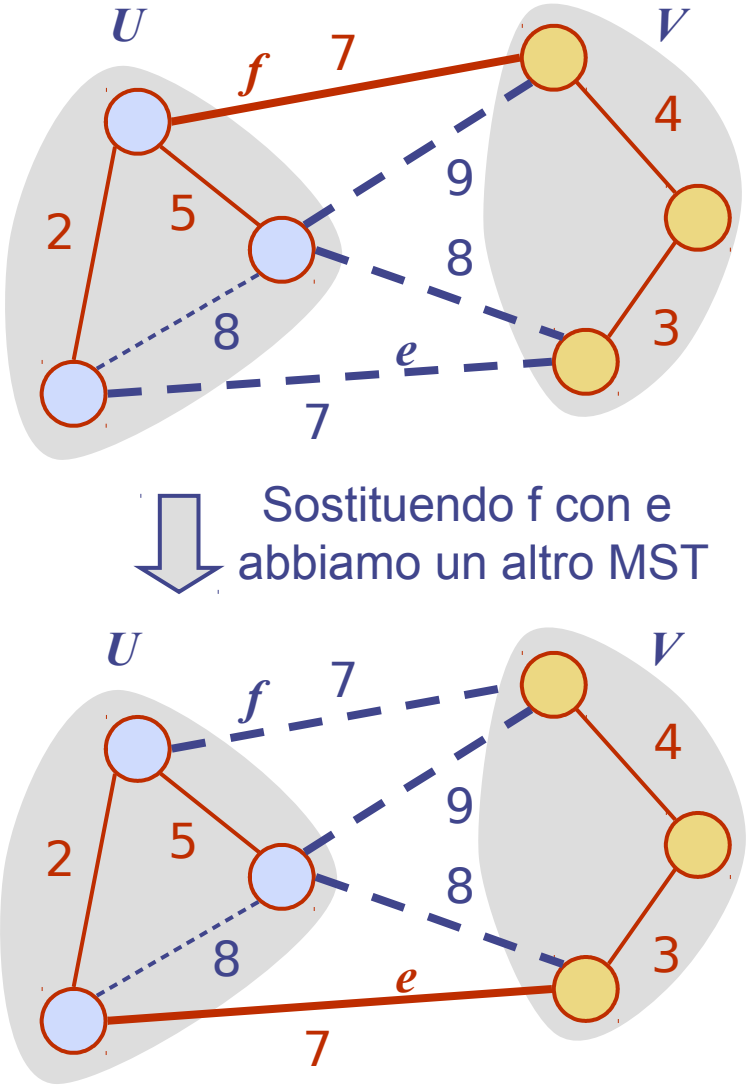


Proprietà di taglio

- Taglio: partizione dell'insieme dei vertici V del grafo in due sottoinsiemi V_1 e V_2
- Arco del taglio: qualsiasi arco con un estremo in V_1 e l'altro in V_2
- Proprietà di taglio
  - Sia e in arco di peso minimo del taglio (V_1, V_2)
  - Esiste un albero minimo ricoprente che contiene e

Prova

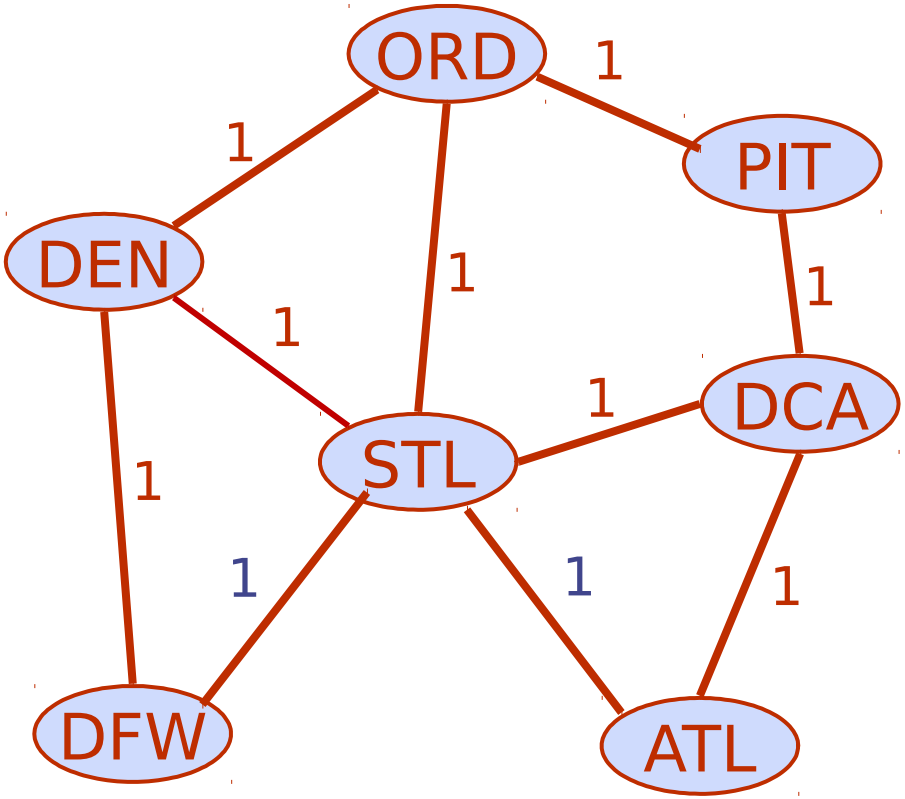
- Sia T un MST di G
- Se T non contiene e si consideri il ciclo C presente in T U {e}
- Sia f un arco di C appartenente al taglio
- Per la proprietà di ciclo
  - w(f) ≤ w(e)
  - Ma e è minimo → w(f) = w(e)



Possono esstere molti MST

In generale in un grafo possiamo avere molti Minimum Spanning Trees

Nel grafo a destra sono moltissimi



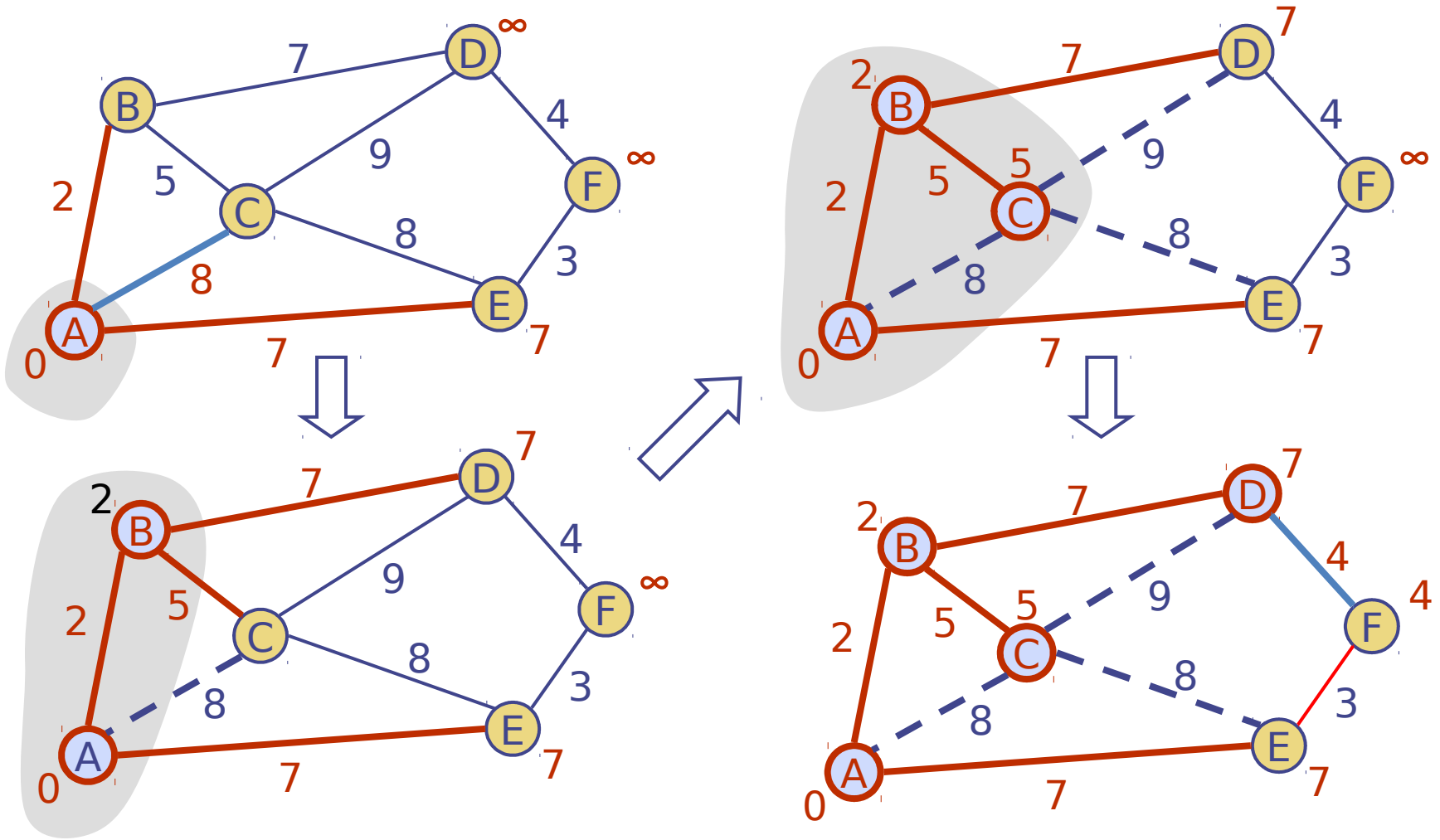
Algoritmo di Prim-Jarnik

Algoritmo di Prim-Jarnik

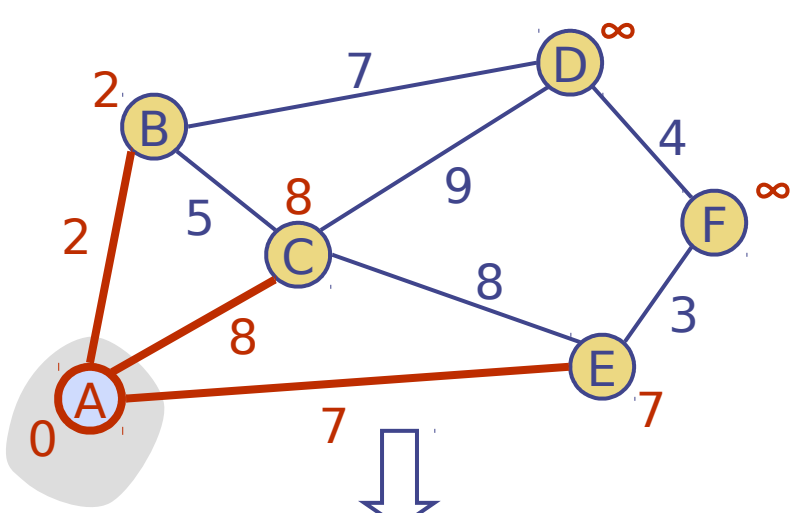
- Abbiamo visto che il problema di trovare cammini minimi si può risolvere con l'algoritmo di Djikstra
- L'algoritmo di Dijkstra definisce una soluzione iniziale che è costituita dal solo nodo iniziale della ricerca
- Ad ogni iterazione l'algoritmo di Djikstra estende il miglior cammino trovato dal vertice iniziale aggiungendo un nuovo arco (e un nuovo nodo alla soluzione)

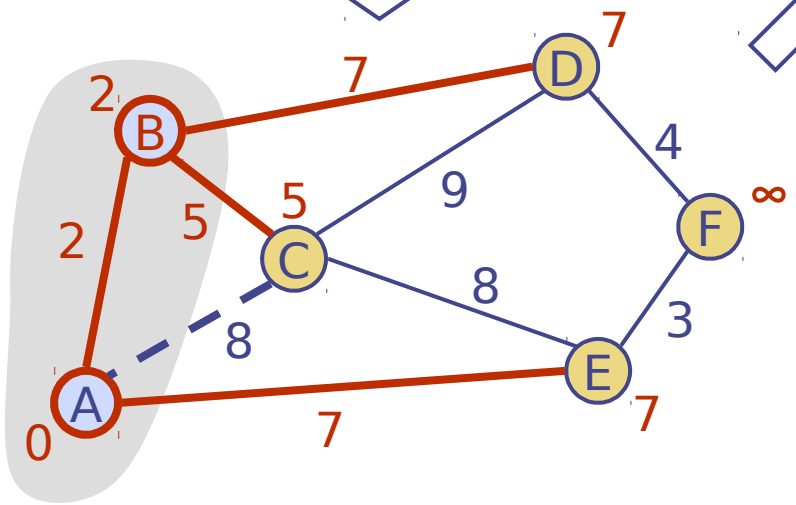
- Dato un grafo G=(V,E), l'algoritmo di Prim-Jarnik definisce una soluzione iniziale T di un albero con un solo vertice iniziale scelto arbitrariamente e nessun arco
- Ad ogni iterazione l'algoritmo di Prim-Jarnik aggiunge alla soluzione T l'arco di peso minimo da T (e il nodo trovato che estende T) di peso minimo fra tutti gli archi che connettono nodi di T a nodi di V-T

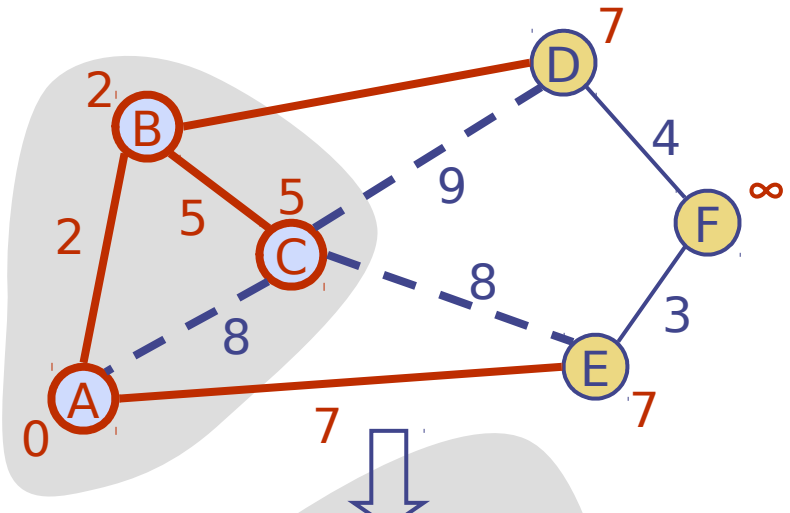
Algoritmo di Prim-Jarnik illustrato

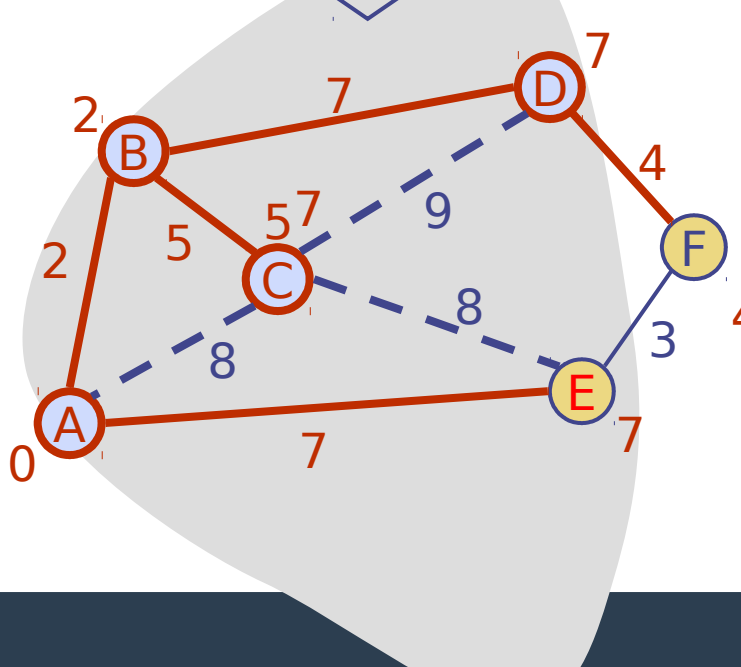


Algoritmo di Prim-Jarnik illustrato



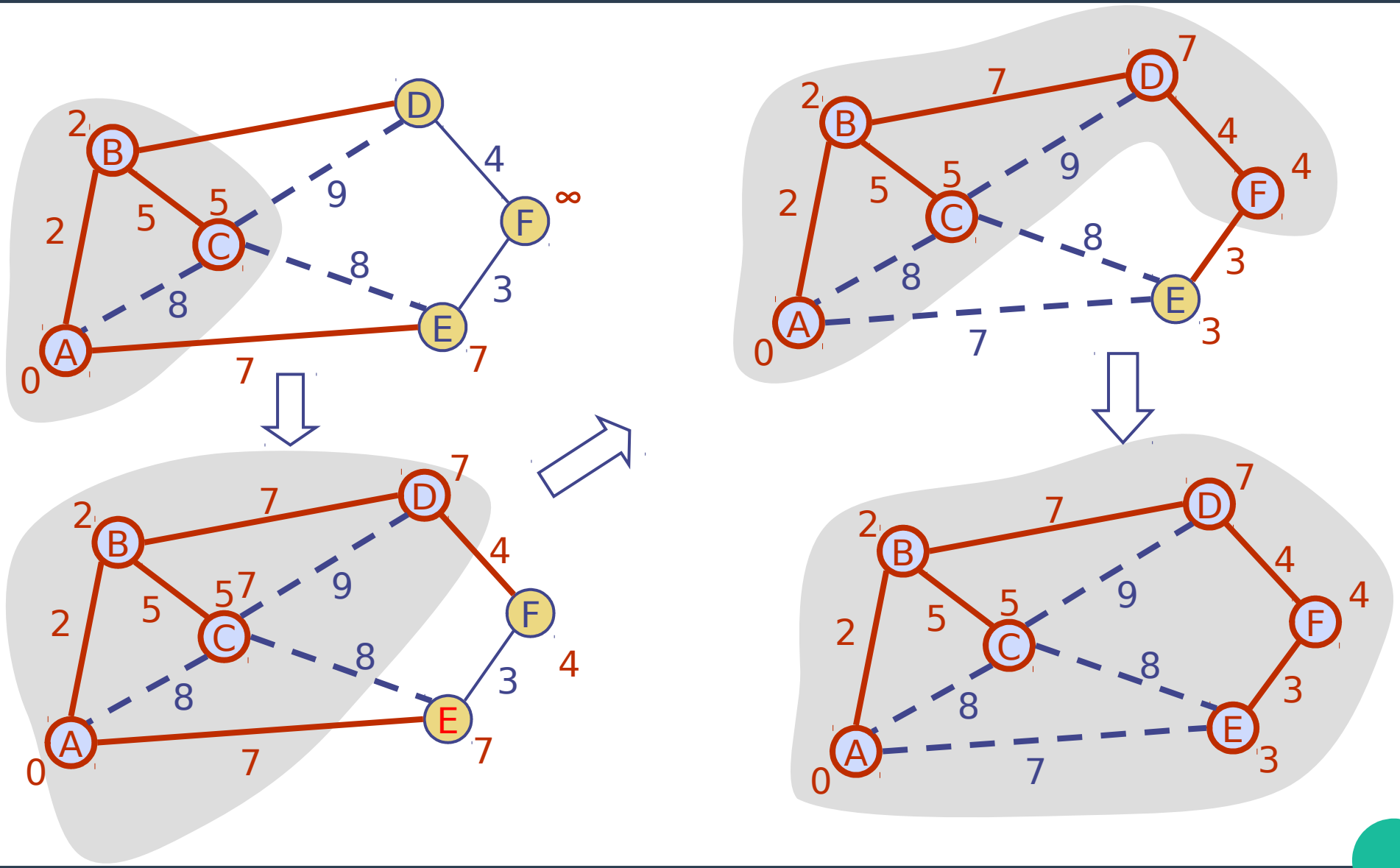






NO!
Ad ogni passo si aggiunge un solo nodo

Algoritmo di Prim-Jarnik illustrato



Algoritmo di Prim-Jarnik: pseudocodice

Algorithm PrimJarnik(G):

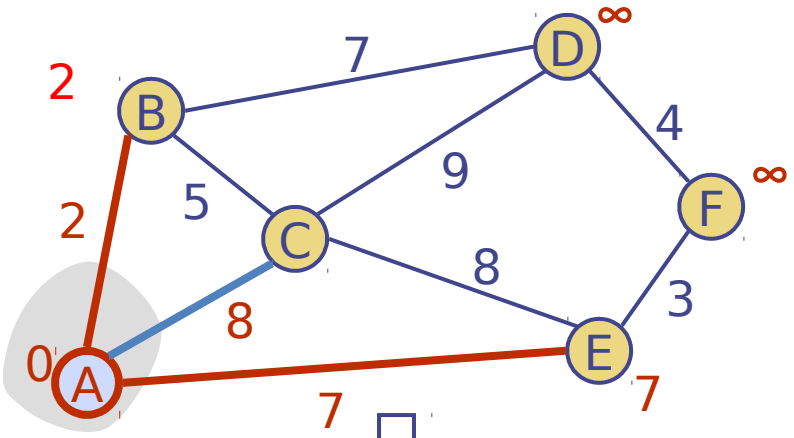
Input: An undirected, weighted, connected graph G with n vertices and m edges

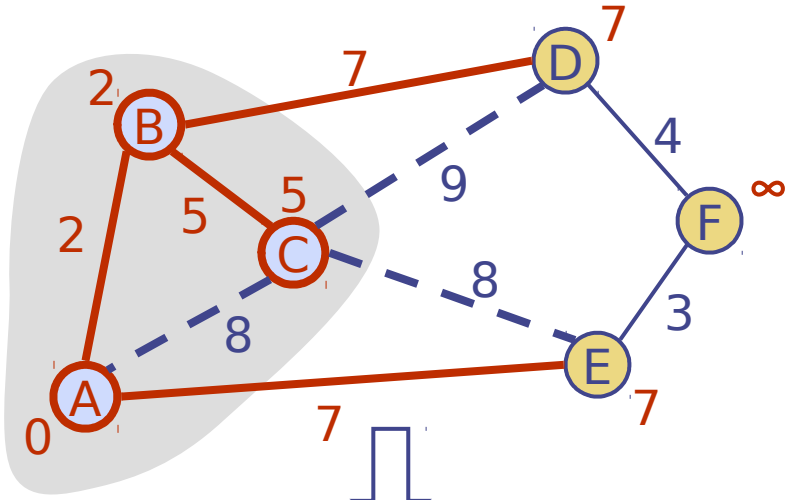
Output: A minimum spanning tree T for G

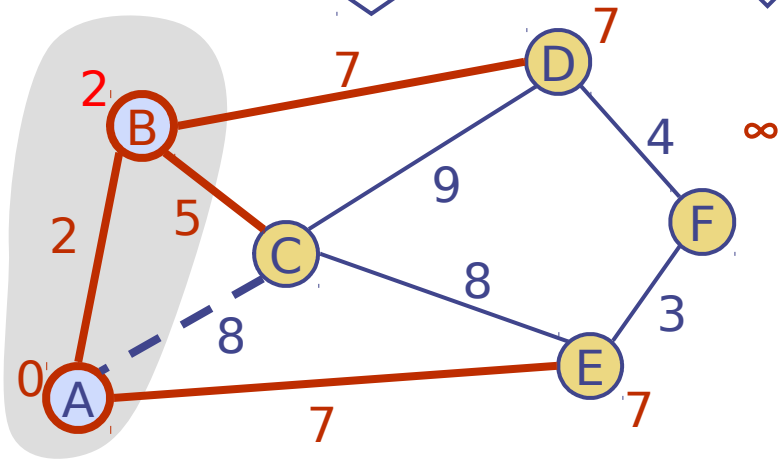
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

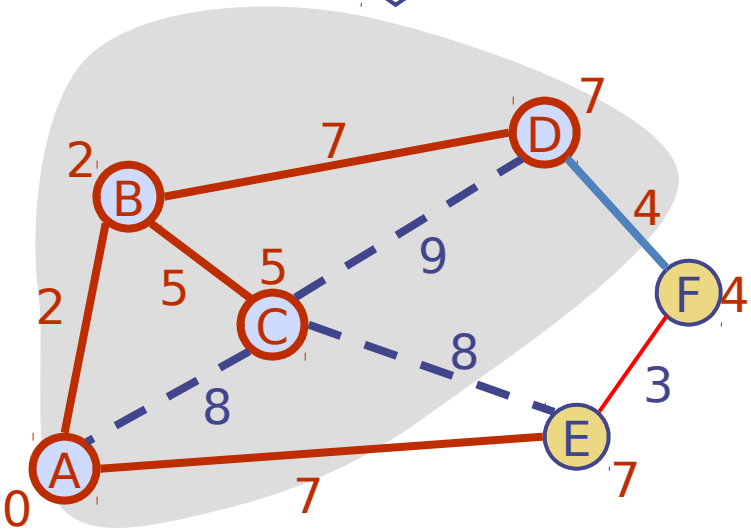
Algoritmo di Prim-Jarnik illustrato

NOTA: Numero vicino al nodo indica la priorità del nodo



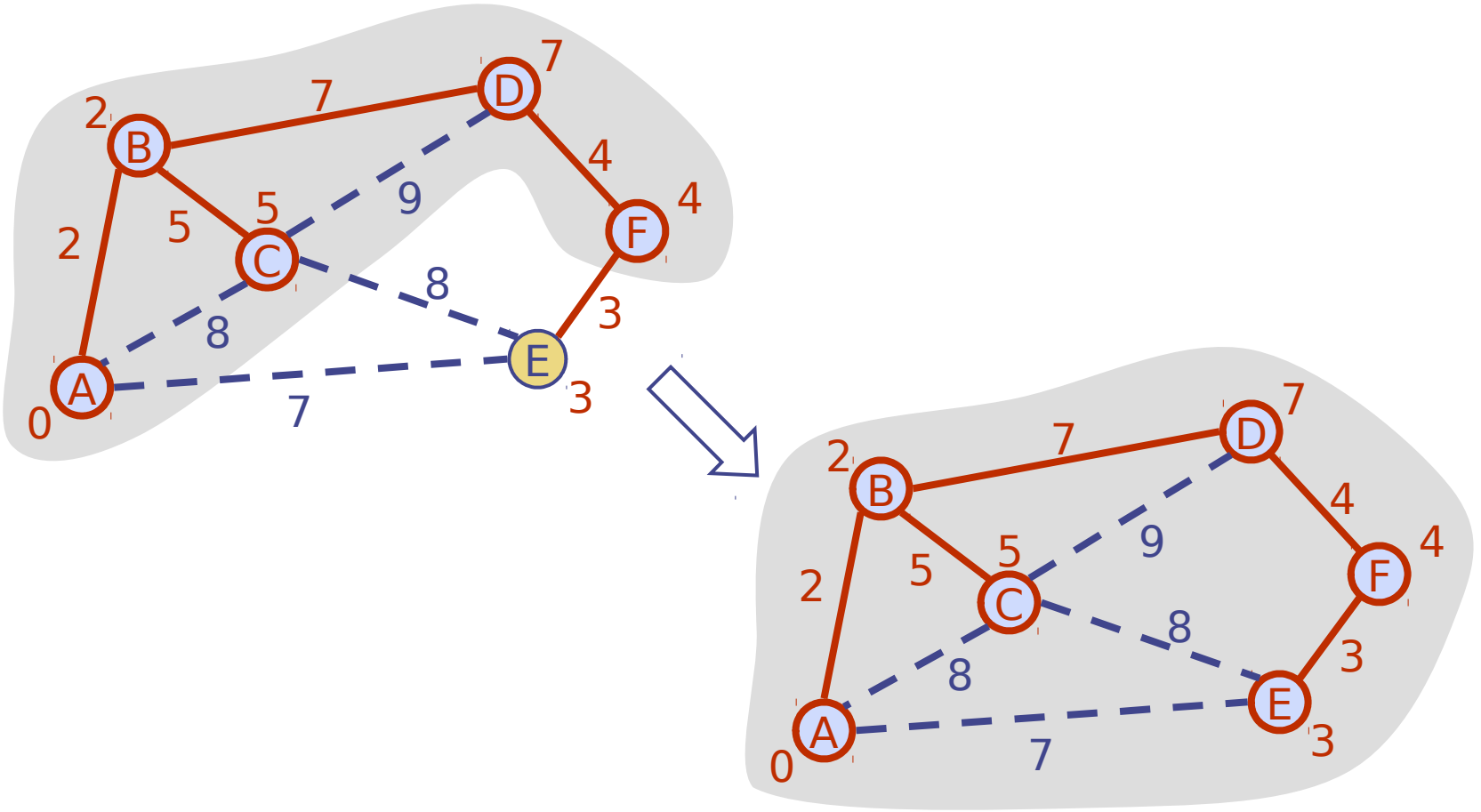






E' stato incluso D ma avremmo potuto includere E

Algoritmo di Prim-Jarnik illustrato



Analisi: complessità

- Costo delle inizializzazioni
  - O(nlog n)
- Costo della generica iterazione del ciclo while (sia u il nodo estratto da Q)
  - O(deg(u)log n) → possibile aggiornamento di Q per ogni vicino di u. (deg(u) è il grado di u – numero di archi di u)
- Costo totale
  - O((n + m)log n)
  - n iterazioni in ciascuna un costo proporzionale al grado del vertice per log(n), e ricordiamo che $\Sigma_u \deg(u) = 2m$

Analisi - correttezza

- Supponiamo per semplicità che tutti i pesi degli archi siano diversi tra loro
- Ogni iterazione del ciclo while identifica un taglio
  - I vertici in Q e quelli in V – Q
- Viene sempre selezionato l’arco di peso minimo del taglio
  - Questo arco ***deve*** appartenere all’MST per la proprietà di taglio

Algoritmi di Dijkstra e di Prim-Jarnik: confronto

L'algoritmo di Dijkstra e di Prim-Jarnik sono molto simili: entrambi trovano alberi ricoprenti

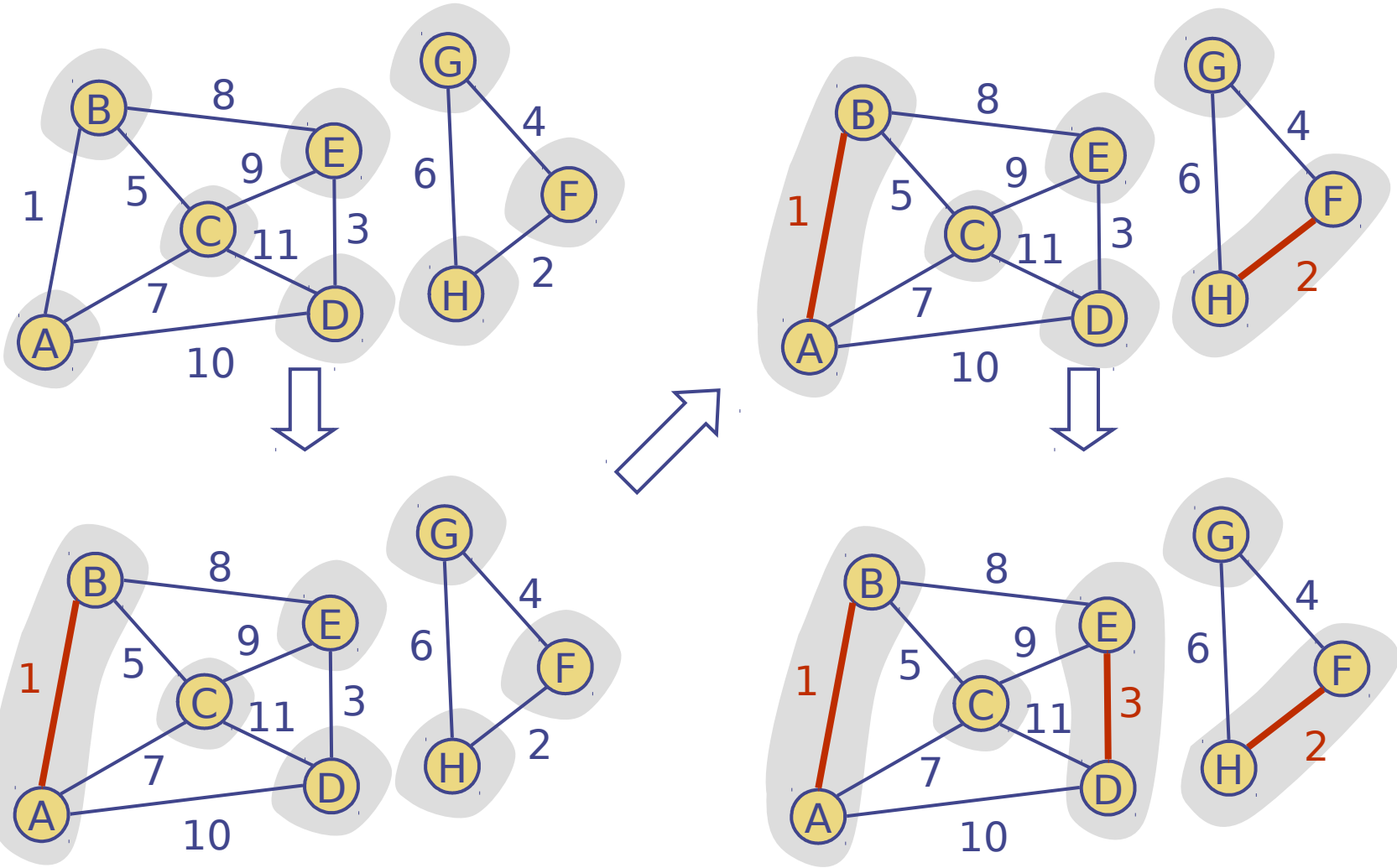
- Esercizio: trova un esempio in cui le soluzioni di Prim e di Dijkstra sono diverse

Inoltre

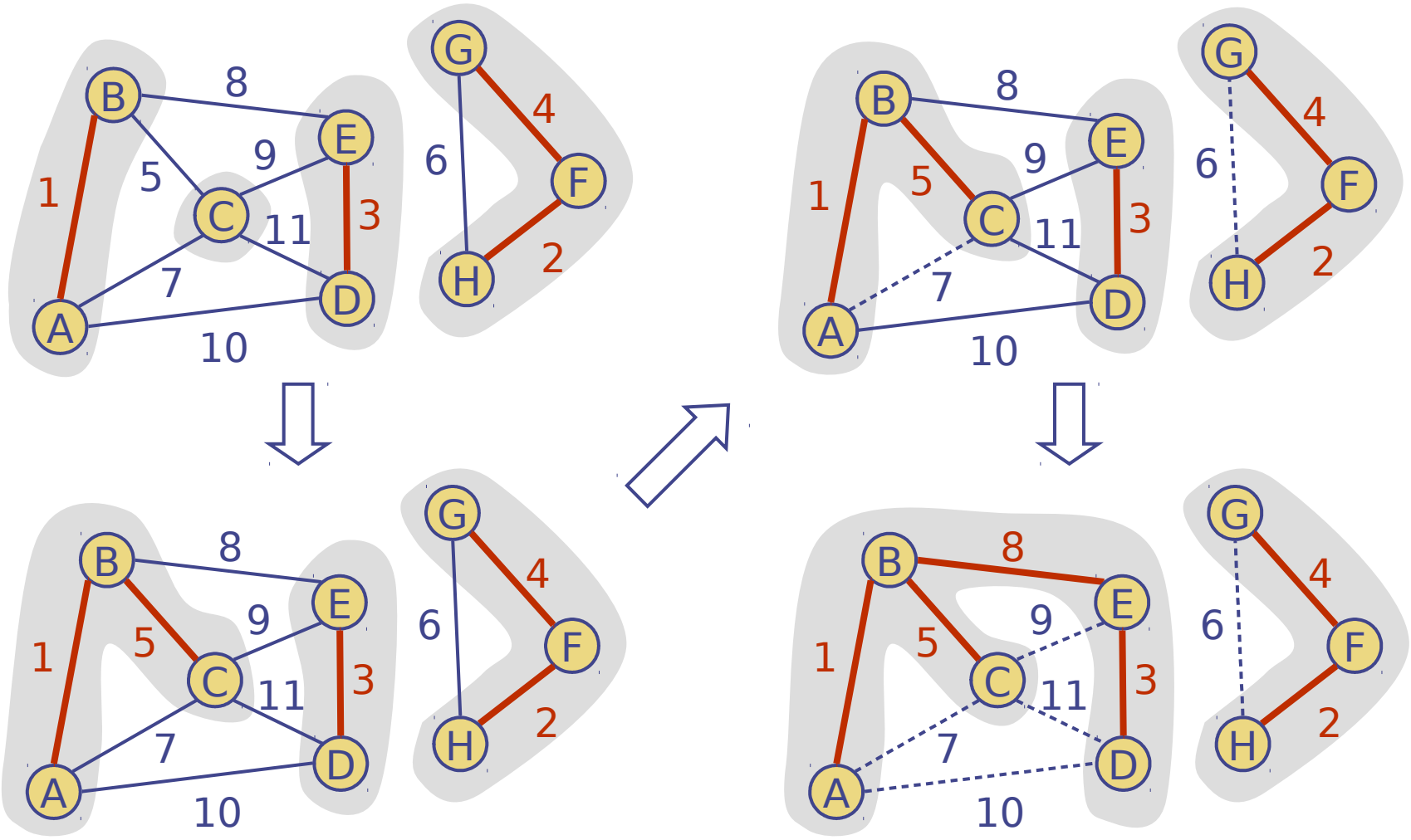
1. Algoritmo di Dijkstra trova il cammino minimo; algoritmo di Prim trova il Minimo Spanning Tree (MST)
2. Algoritmo di Dijkstra opera su grafi diretti e non diretti; algoritmo di Prim opera solo su grafi non diretti
3. Algoritmo di Prim può operare con archi di peso negativo mentre algoritmo di Dijkstra può fornire errori se è presente anche un solo arco di peso negativo

Algoritmo di Kruskal

Algoritmo di Kruskal illustrato



Algoritmo di Kruskal illustrato



Algoritmo di Kruskal: implementazione semplice

1. Ordina gli archi del grafo in ordine crescente di peso
2. Inizializza gli archi del MST T={}
3. For i= 1 to |E|
   if T U {e_i} non ha cicli then aggiungi e_i a T

Nota

il passo 3 è ripetuto m volte (m numero degli archi)

Come eseguire in modo efficiente il passo 3?

- Usare algoritmi di visito (ad esempio BFS, DFS) : costo lineare
- Union-Find permette di eseguire il test in tempo costante!

Astrazione Union-find

- Oggetti
- Insiemi *disgiunti* di oggetti
- Si vogliono implementare efficientemente le seguenti operazioni
  - makeCluster(x): crea l'insieme {x}
  - union(a, b): sostituire agli insiemi contenenti gli oggetti a e b la loro unione
  - find(x): restituisce il *leader* dell'insieme contenente x

Algoritmo di Kruskal: pseudocodice

Algorithm Kruskal(G):

Input: A simple connected weighted graph G with n vertices and m edges

Output: A minimum spanning tree T for G

for each vertex v in G **do**

  Define an elementary cluster $C(v) = \{v\}$.

Initialize a priority queue Q to contain all edges in G, using the weights as keys.

$T = \emptyset$ {T will ultimately contain the edges of the MST}

while T has fewer than $n - 1$ edges **do**

  (u, v) = value returned by Q.remove_min()

  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.

  if $C(u) \neq C(v)$ **then**

    Add edge (u, v) to T.

    Merge $C(u)$ and $C(v)$ into one cluster.

return tree T

Make cluster

Union find

Analisi

- Complessità
  - Inizializzazioni: O(mlog n)
  - Ciclo while: abbiamo una sequenza di al più n - 1 operazioni Union-Find → costo O(n + nlog n)
  - Costo complessivo
    - O((m + n)log n)
- Correttezza
  - Stesso argomento usato per algoritmo di Prim-Jarnik

In pratica qual è il migliore? Non esiste un chiaro vincitore
Esperimenti mostrano che Prim-Jarnik è più veloce nel caso di grafi densi mentre Kruskal è più efficiente in grafi sparsi - |E|= O(n)

Una nota sul costo di esecuzione

Abbiamo visto diversi problemi su grafi (V insieme dei vertici, E insieme archi)

- Verifica se un grafo è connesso: algoritmi di visita del grafo con costo O(|E| + |V|) nel caso peggiore
- Trova cammino minimo in grafi pesati: algoritmo di Dijkstra's con costo O(|E| log|V|) nel caso peggiore
- Trova il minimo albero ricoprente: algoritmi di Prim o di Kruskal's con costo O(|E| log|V|) nel caso peggiore

Di conseguenza questi problemi sono risolubili in tempo accettabile anche per grafi di grandi dimensioni; questi problemi sono considerati ***trattabili***

Abbiamo visto che esistono molti problemi su grafi che sono NP-completi e, per questi problemi (colorazione, insieme indipendente, problema del commesso viaggiatore, ecc) congetturiamo che non esista un algoritmo di soluzione polinomiale; questi problemi sono considerati ***intrattabili***