Data Structures and Algorithms

Chapter 8

Algorithms for Weighted Graphs

Werner Nutt

Acknowledgments

- The course follows the book “Introduction to Algorithms‘”, by **Cormen, Leiserson, Rivest and Stein**, MIT Press [CLRST]. Many examples displayed in these slides are taken from their book.
- These slides are based on those developed by Michael Böhlen for this course.

(See http://www.inf.unibz.it/dis/teaching/DSA/)

- The slides also include a number of additions made by Roberto Sebastiani and Kurt Ranalter when they taught later editions of this course

(See http://disi.unitn.it/~rseba/DIDATTICA/dsa2011_BZ//)

DSA, Chapter 8: Overview

1. Weighted Graphs
2. Shortest Paths
   – Dijkstra’s algorithm
3. Minimum Spanning Trees
   – Greedy Choice Theorem
   – Prim’s algorithm

DSA, Chapter 8: Overview

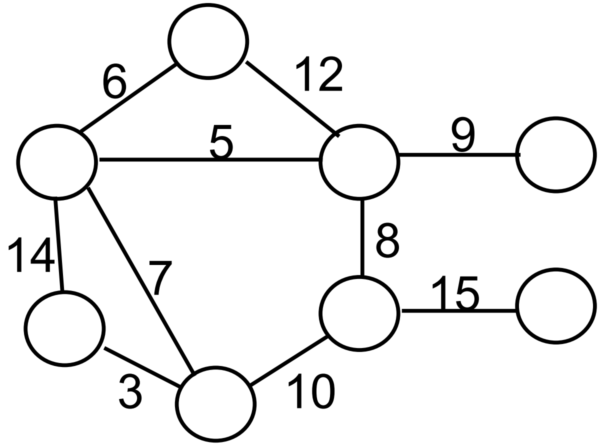
1. Weighted Graphs
2. Shortest Paths
   – Dijkstra’s algorithm
3. Minimum Spanning Trees
   – Greedy Choice Theorem
   – Prim’s algorithm

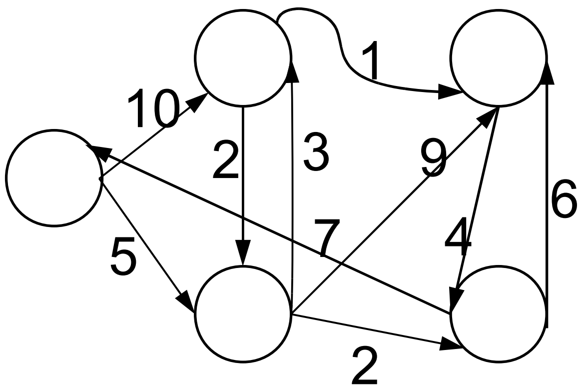
Weighted Graphs

- May be *directed* or *undirected* graphs $G = (V,E)$
- Have a weight function

  $w : E \rightarrow R$

  which assigns cost or length or other values to edges





DSA, Chapter 8: Overview

1. Weighted Graphs
2. Shortest Paths
   – Dijkstra’s algorithm
3. Minimum Spanning Trees
   – Greedy Choice Theorem
   – Prim’s algorithm

Shortest Path

- We generalize *distance* to the weighted setting
- We consider a digraph $G = (V,E)$ with weight function $w: E \to R$ (assigning real values to edges)
- The weight of path $p = v_1 \to v_2 \to \ldots \to v_k$ is

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

- Shortest path = a path of minimum weight (cost)
- Applications
  - static/dynamic network routing
  - robot motion planning
  - map/route generation in traffic

Shortest-Path Problems

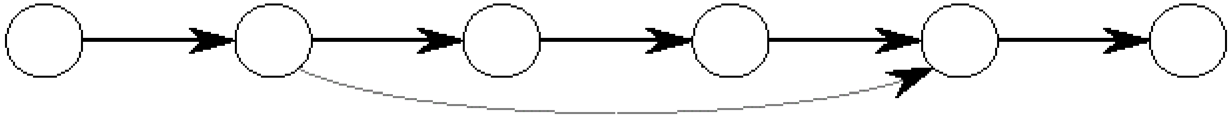
- Single-source. Find a shortest path from a given source (vertex *s*) to each of the vertices.
- Single-pair. Given two vertices, find a shortest path between them. Solution to single-source problem solves this problem efficiently, too.
- All-pairs. Find shortest-paths for every pair of vertices. Dynamic programming algorithm.
- Unweighted shortest-paths – BFS.

Optimal Substructure

Theorem: Subpaths of shortest paths are shortest paths.

Proof:

If some subpath were not the shortest path, one could substitute the shorter subpath and create a shorter total path.



Negative Weights and Cycles

Observations:

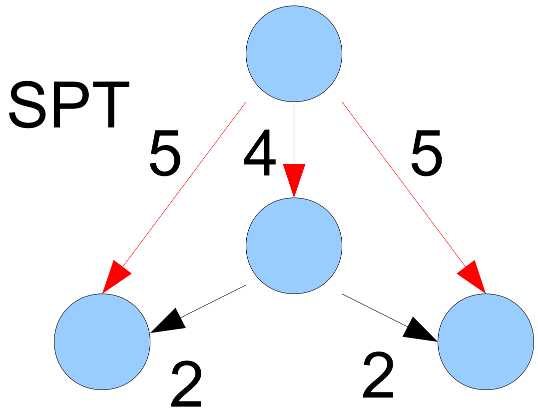
- Negative edges are OK,
  as long as there are no *negative weight cycles*
  (otherwise, paths with arbitrary small "lengths"
  would be possible).
- Shortest-paths can have no cycles
  (otherwise we could improve them by removing cycles).

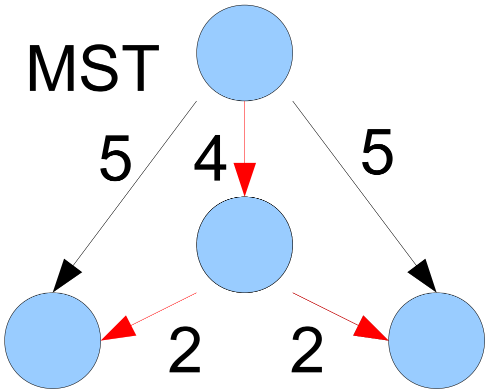
  Any shortest path in graph G can be no longer than $n - 1$ edges, where n is the number of vertices.

Shortest Path Tree

- The result of the algorithms is a *shortest path tree (SPT).* For each vertex *v,* it
  - records a shortest path from the start vertex *s* to *v*;
  - *v*.pred is the predecessor of *v* on this shortest path
  - *v*.dist is the shortest path length from *s* to *v*

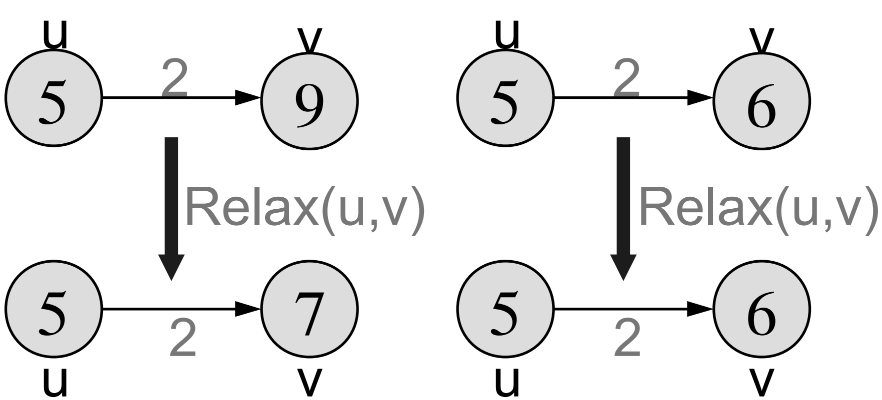
- *Note: SPT is different from minimum spanning tree (MST)!*





Relaxation

- For each vertex *v* in the graph, we maintain *v*.**dist**, the estimate of the shortest path from *s. It is* initialized to ∞ at the start.
- Relaxing an edge (*u*,*v*) means testing whether we can improve the shortest path to *v* found so far by going through *u.*




```
Relax (u,v)
if v.dist > u.dist + w(u,v) then
    v.dist := u.dist + w(u,v)
    v.pred := u
```

Dijkstra's Algorithm

- Assumption: non-negative edge weights
- Greedy, similar to Prim's algorithm for MST
- Like breadth-first search
  (if all weights = 1, one can simply use BFS)
- Use *Q*, a priority queue with keys *v*.dist
  (BFS used FIFO queue, here we use a PQ,
  which is re-organized whenever some dist decreases)
- Basic idea
  - maintain a set *S* of solved vertices
  - at each step, select a "closest" vertex *u*,
    add it to *S*, and
    relax all edges from *u*

Priority Queues

- A priority queue maintains a set *S* of elements, each with an associated key value.
- We need a PQ to support the following operations
  - **init**(*VertexSet S*)
  - *Vertex* **extractMin**()
  - **modifyKey**(*Vertex v, Key k*)
- To choose how to implement a PQ, we need to count how many times these operations are performed.

Dijkstra's Algorithm: Pseudo Code

Input: Graph *G*, start vertex *s*

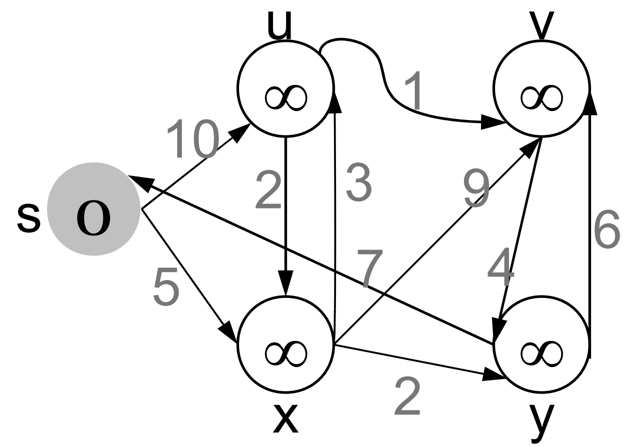
```
Dijkstra(G,s) do
for u ∈ G.V
    u.dist := ∞
    u.pred := NULL
s.dist := 0
Q := new PriorityQueue
Q.init(G.V) // initialize priority queue Q
while not Q.isEmpty() do
    u := Q.extractMin()
    for v ∈ u.adj do
        if v in Q and u.dist+w(u,v) < v.dist
            then Q.modifyKey(v,u.dist+w(u,v))
                 v.pred := u
```

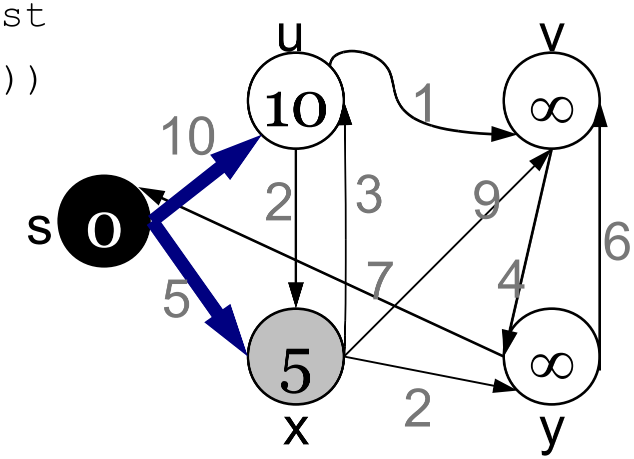
initialize graph

relax edges

Dijkstra's Algorithm: Example/1

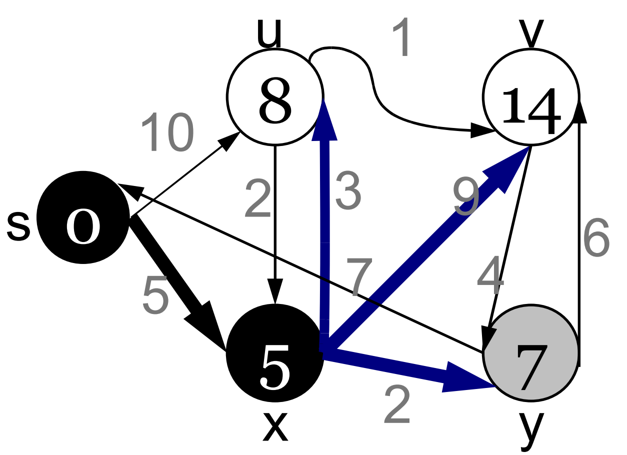
```
Dijkstra(G,s)
for u ∈ G.V do
   u.dist := ∞
   u.pred := NULL
s.dist := 0
Q := new PriorityQueue
Q.init(G.V)
while not Q.isEmpty() do
   u := Q.extractMin()
   for v ∈ u.adj do
      if v in Q and u.dist+w(u,v) < v.dist
         then Q.modifyKey(v,u.dist+w(u,v))
              v.pred := u
```

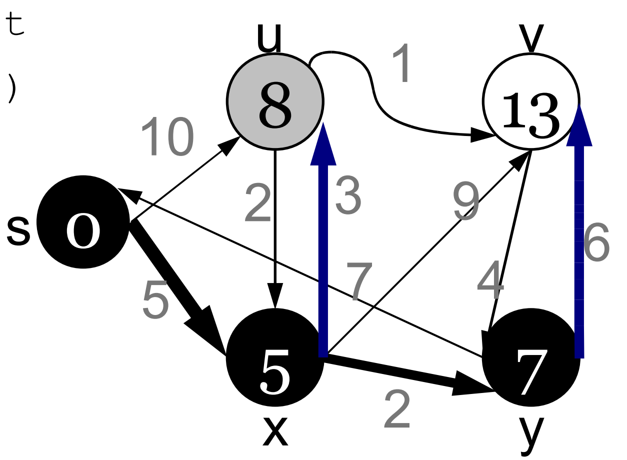




Dijkstra’s Algorithm: Example/2

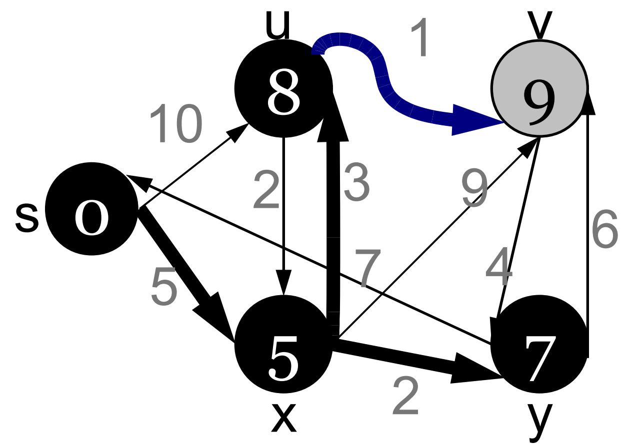
```
Dijkstra(G,s)
for u ∈ G.V do
   u.dist := ∞
   u.pred := NULL
s.dist := 0
Q := new PriorityQueue
Q.init(G.V)
while not Q.isEmpty() do
   u := Q.extractMin()
   for v ∈ u.adj do
      if v in Q and u.dist+w(u,v) < v.dist
         then Q.modifyKey(v,u.dist+w(u,v))
              v.pred := u
```

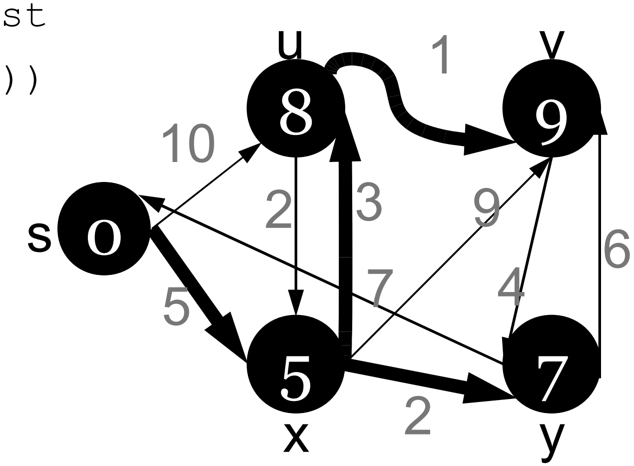




Dijkstra's Algorithm: Example/3

```
Dijkstra(G,s)
for u ∈ G.V do
   u.dist := ∞
   u.pred := NULL
s.dist := 0
Q := new PriorityQueue
Q.init(G.V)
while not Q.isEmpty() do
   u := Q.extractMin()
   for v ∈ u.adj do
      if v in Q and u.dist+w(u,v) < v.dist
         then Q.modifyKey(v,u.dist+w(u,v))
              v.pred := u
```





Notation

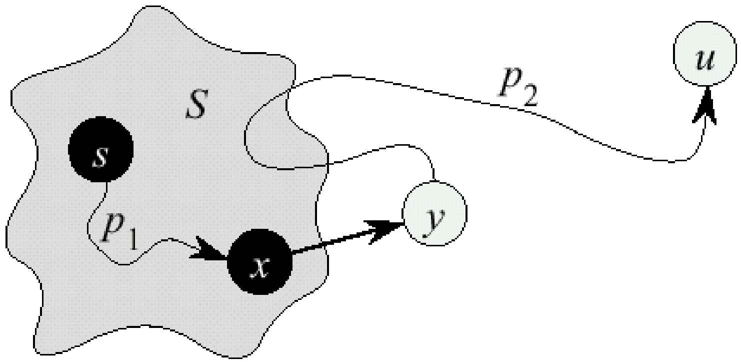
For any nodes u, v in $G = (V,E)$, we define

$$\delta(u,v) = \text{minimal length of a path from } u \text{ to } v$$

We call $\delta(u,v)$ the distance from u to v

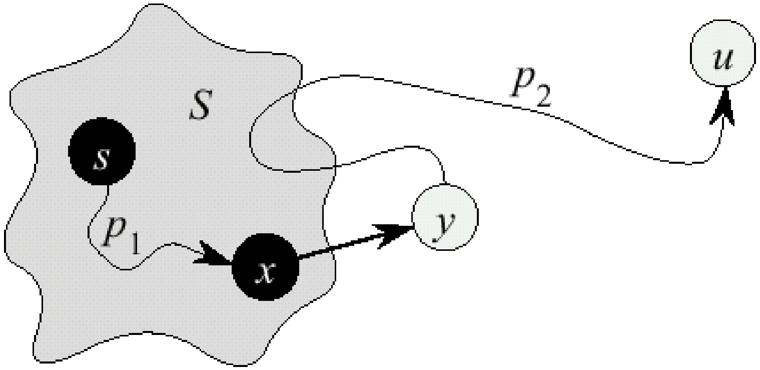
Dijkstra's Algorithm: Correctness/1

- We prove that *whenever* u is *added* to the set S of *solved vertices*, *then* u.dist = $\delta(s,u)$, i.e., dist is minimum.
- Proof (by contradiction)
  - Initially $\forall v$: v.dist $\geq \delta(s,v)$
  - Let u be the **first** vertex such that there is a shorter path than u.dist, i.e., u.dist > $\delta(s,u)$
  - We will show that this assumption leads to a contradiction



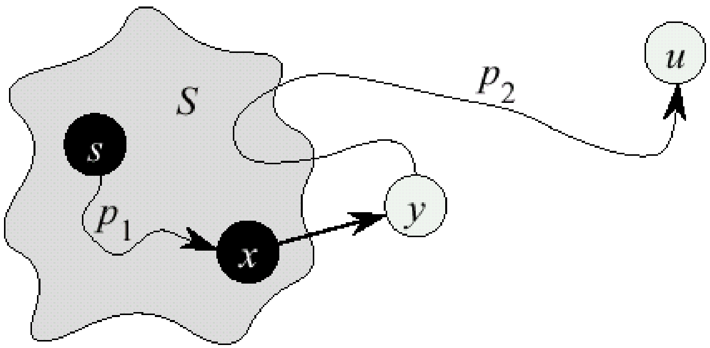
Dijkstra's Algorithm: Correctness/2

- Let *y* be the first vertex in *V* \ *S* on the actual shortest path from *s* to *u*, then it must be that *y*.dist = δ(*s*,*y*) because
  - *x*.dist is set correctly for *y*'s predecessor *x*∈*S* on the shortest path (by choice of *u* as the first vertex for which dist is set incorrectly)
  - when the algorithm inserted *x* into *S*, it relaxed the edge (*x*,*y*), setting *y*.dist to the correct value



Dijkstra’s Algorithm: Correctness/3

u.dist > $\delta(s,u)$	initial assumption
= $\delta(s,y) + \delta(y,u)$	optimal substructure
= y.dist + $\delta(y,u)$	*correctness of y.dist*
≥ *y.dist*	*no negative weights*



- But u.dist > y.dist ⇒ algorithm would have chosen y (from the PQ) to process next, not u
  ⇒ contradiction
- Thus, u.dist = $\delta(s,u)$ at time of insertion of u into S, and Dijkstra's algorithm is correct

Implementation Issues

We highlight the operations on the priority queue

```
Dijkstra(G,s) do
for u ∈ G.V
    u.dist := ∞
    u.pred := NULL
s.dist := 0
Q := new PriorityQueue
Q.init(G.V) // initialize priority queue Q
while not Q.isEmpty() do
    u := Q.extractMin()
    for v ∈ u.adj do
        if v in Q and u.dist+w(u,v) < v.dist
            then Q.modifyKey(v,u.dist+w(u,v))
                 v.pred := u
```

initialize graph

relax edges

Priotity Queue Operations

We can implement priority queues as

- simple arrays
- heaps.

In both cases,

- initializing takes time $O(n)$
- emptyness checks take time $O(1)$

However, the running times differ for

- ExtractMax()
- ModifyKey

Dijkstra's Algorithm: Running Time

- Extract-Min executed $|V|$ times
- Modify-Key executed $|E|$ times
- Time = $|V| \times T_{\text{Extract-Min}} + |E| \times T_{\text{Modify-Key}}$
- T depends on implementation of Q

Q	T(Extract-Min)	T(Modify-Key)	Total
array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\|E\| \log \|V\|)$

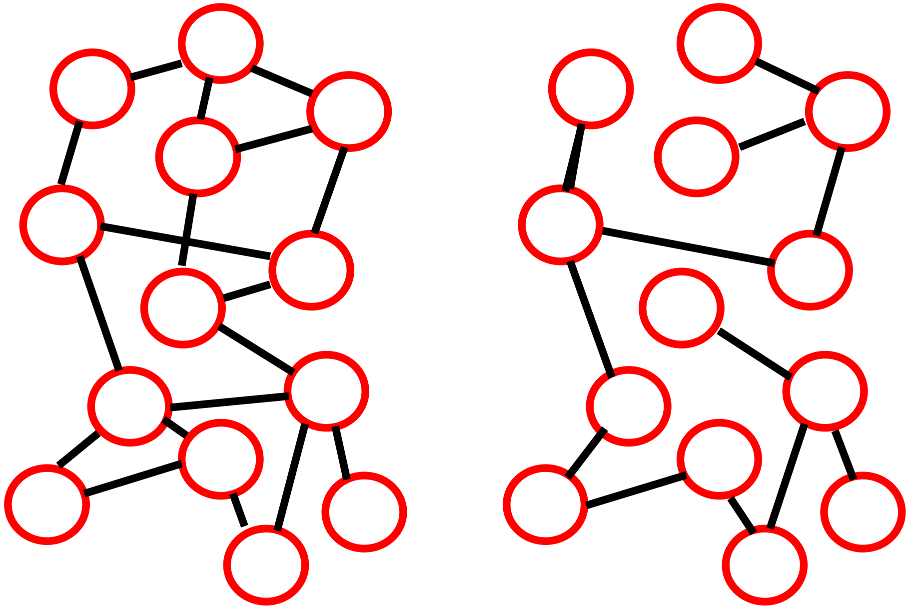
DSA, Chapter 8: Overview

1. Weighted Graphs
2. Shortest Paths
   – Dijkstra’s algorithm
3. Minimum Spanning Trees
   – Greedy Choice Theorem
   – Prim’s algorithm

Spanning Tree

- A spanning tree of *G* is a subgraph which
  - contains all vertices of *G*
  - is a tree

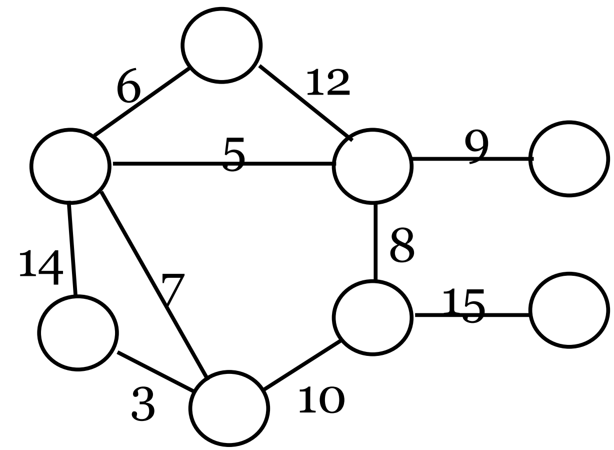
- How many edges are there in a spanning tree, if *V* is the set of vertices?

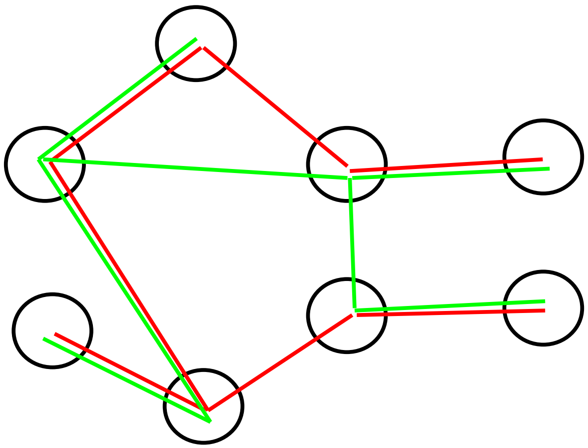


Minimum Spanning Trees

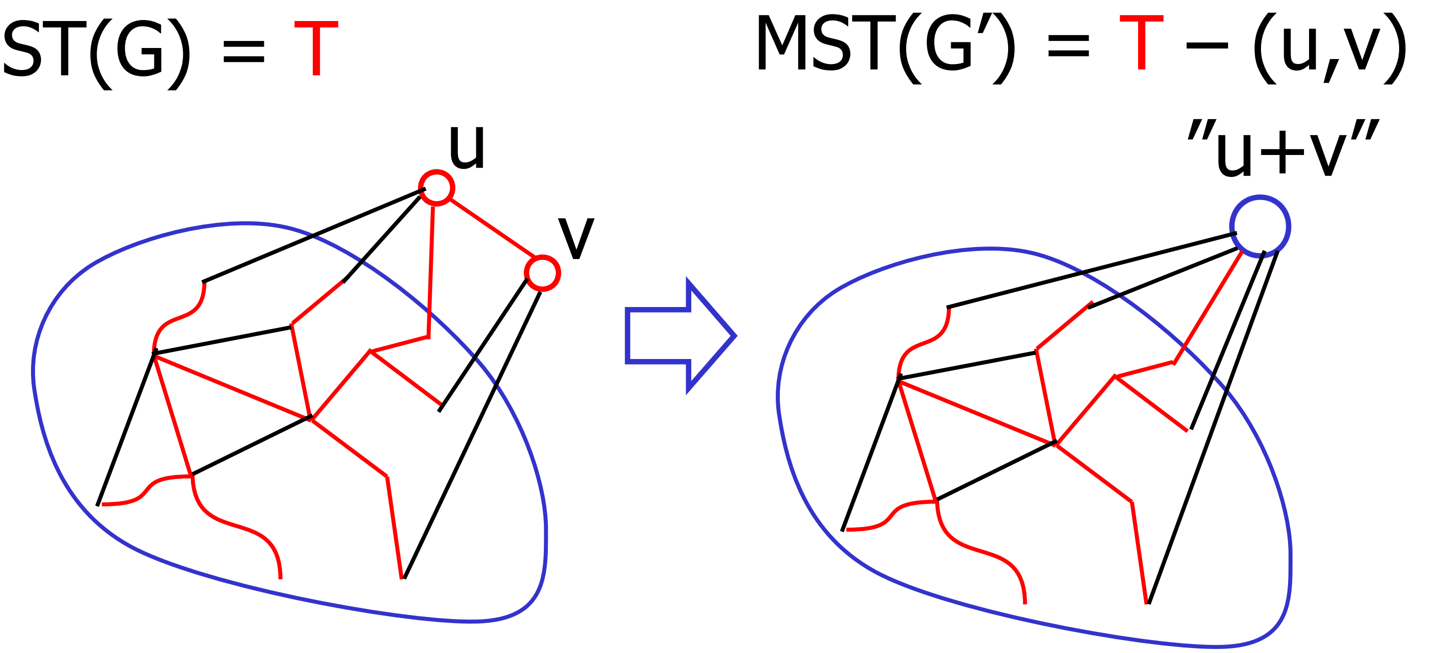
- Undirected, connected graph *G* = (*V*,*E*)
  - Weight function *W*: *E* → *R* (assigning cost or length or other values to edges)
- Spanning tree: tree that connects all vertexes
- Minimum spanning tree (MST): spanning tree *T* that minimizes

$$w(T)=\sum_{(u,v)\in T} w(u,v)$$





Optimal Substructure



Rationale:

If *G'* had a cheaper subtree *T'*,
then we would get a cheaper subtree of *G*: *T'* + (*u*,*v*)

Idea for an Algorithm

- We have to make $|V|-1$ choices (edges of the MST) to arrive at the optimization goal
- After each choice we have a sub-problem that is one vertex smaller than the original problem.
  - A dynamic programming algorithm would consider all possible choices (edges) at each vertex.
  - Goal: at each vertex cheaply determine an edge that definitely belongs to an MST

Greedy Choice

Greedy choice property: locally optimal (greedy) choice yields a globally optimal solution.

Theorem: Let $G = (V, E)$ and $S \subseteq V$. Consider the cut of G formed by S and $V \setminus S$, that is, the partitioning into two disjoint parts.

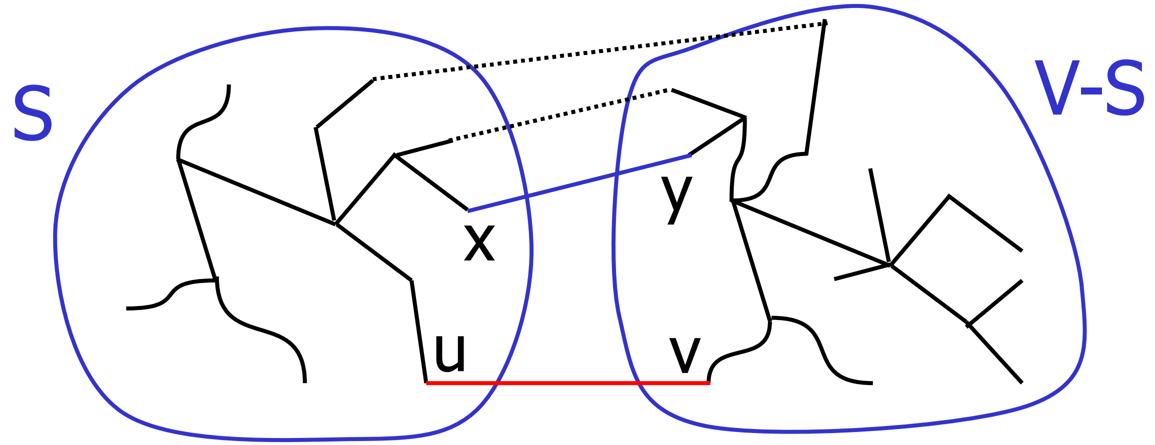
- Suppose (u,v) is a light edge, that is, it is a *min*-weight edge of G that connects S and $V - S$.
- Then (u,v) belongs to *every* MST of G

Greedy Choice/2

Proof:

- Suppose *(u,v)* is light, but (*u*,*v*) ∉ any MST
- Look at the path from *u* to *v* in some MST *T*
- Let (*x, y*) be the first edge on a path from *u* to *v* in *T* that crosses from *S* to *V – S*. Swap (*x, y*) with (*u*,*v*) in *T*.
- This improves cost of *T*

➔ Contradiction (since *T* is supposed to be an MST)



Generic MST Algorithm

```
Generic-MST(G, w)
A := ∅  // Contains edges that belong to a MST
while A does not form a spanning tree do
    find an edge (u,v) that is safe for A
    A := A ∪ {(u,v)}
return A
```

A safe edge is an edge that does not destroy *A*'s property.

```
MoreSpecific-MST(G, w)
1    A := ∅  // Contains edges that belong to a MST
2    while A does not form a spanning tree do
3.1      Make a cut (S, V-S) of G that does not split A
3.2      Take the min-weight edge (u,v) connecting S to V-S
4        A := A ∪ {(u,v)}
5  return A
```

Prim-Jarnik Algorithm

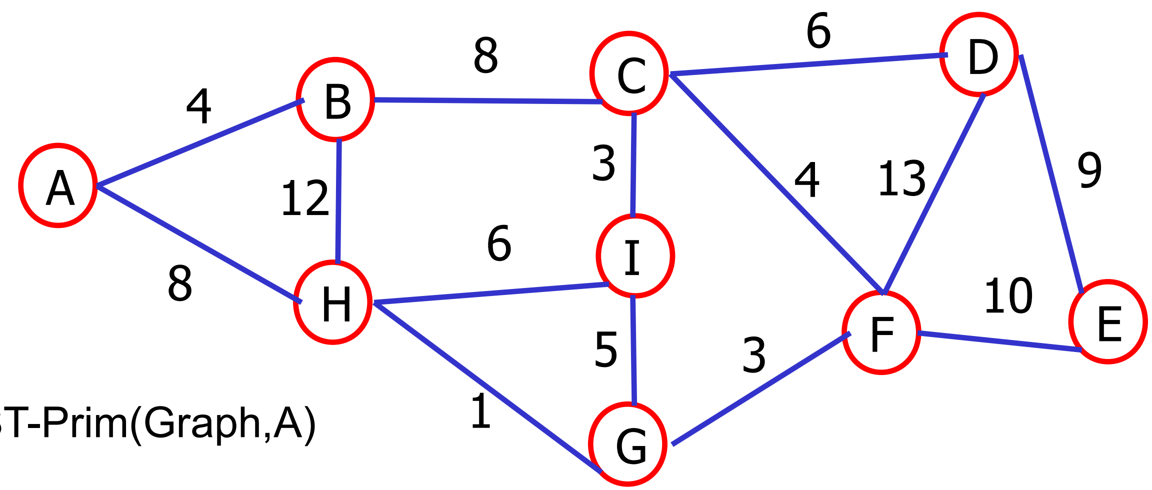
- Vertex-based algorithm
- Grows a single MST T one vertex at a time
- The set *A* covers the portion of T that was already computed
- Annotate all vertices *v* outside of the set *A* with *v*.key as the *current minimum weight of an edge that connects v to a vertex in A* (*v*.key = ∞ if no such edge exists)

Prim-Jarnik Algorithm/2

```
MST-Prim(G,s)
for each vertex u ∈ G.V
  u.key := ∞
  u.pred := NULL
s.key := 0
init(Q, G.V) // Q is a priority queue
while not isEmpty(Q)
  u := extractMin(Q) // add u to T
  for each v ∈ u.adj do
    if v ∈ Q and w(u,v) < v.key then
      v.key := w(u,v)
      modifyKey(Q,v)
      v.pred := u
```

updating keys

Prim-Jarnik Example

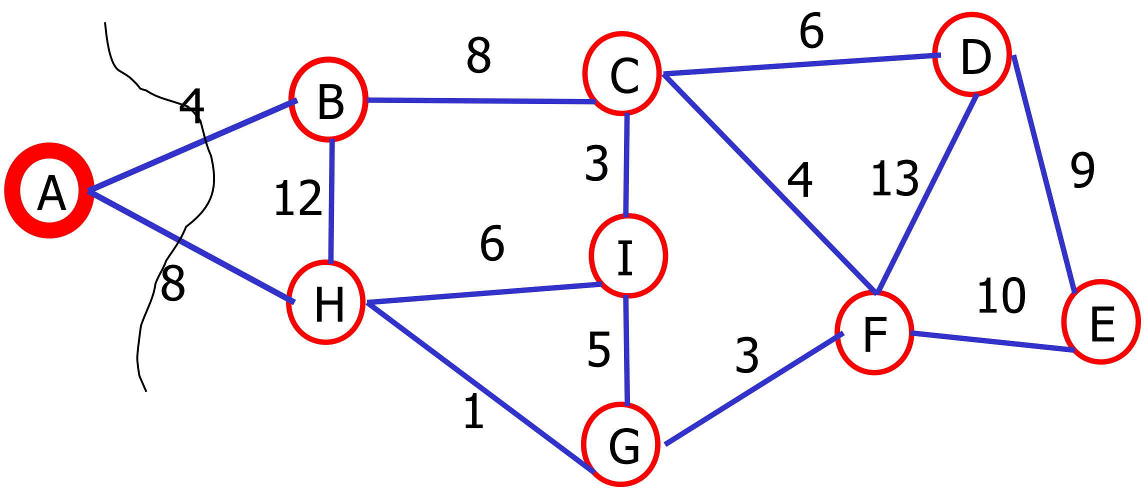


MST-Prim(Graph,A)

A = {}

Q = A-NULL/0, B-NULL/∞, C-NULL/∞, D-NULL/∞, E-NULL/∞, F-NULL/∞, G-NULL/∞, H-NULL/∞, I-NULL/∞

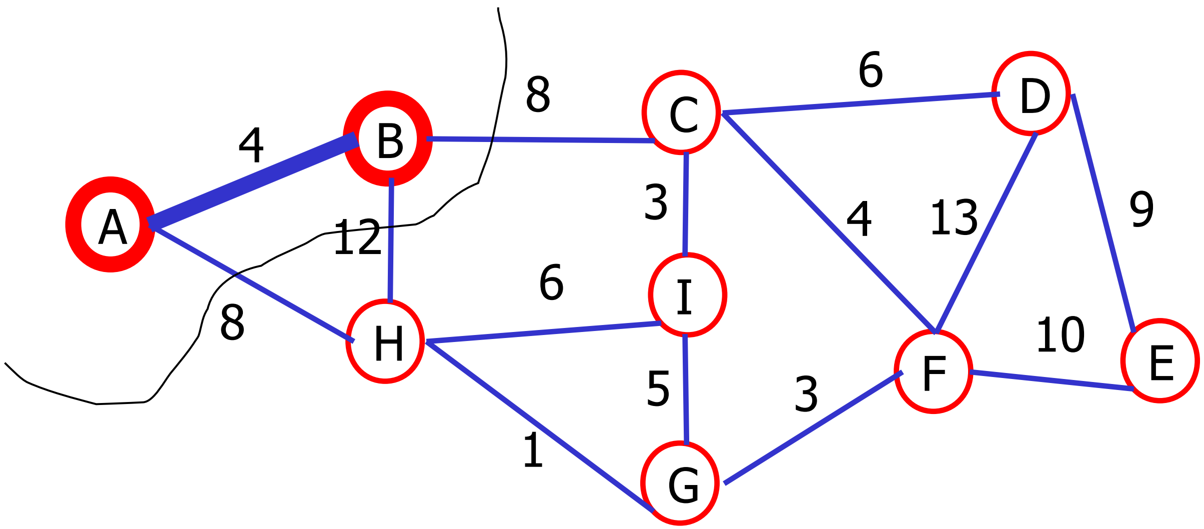
Prim-Jarnik Example/2



A = A-NULL/0

Q = B-A/4, H-A/8, C-NULL/∞, D-NULL/∞, E-NULL/∞, F-NULL/∞, G-NULL/∞, I-NULL/∞

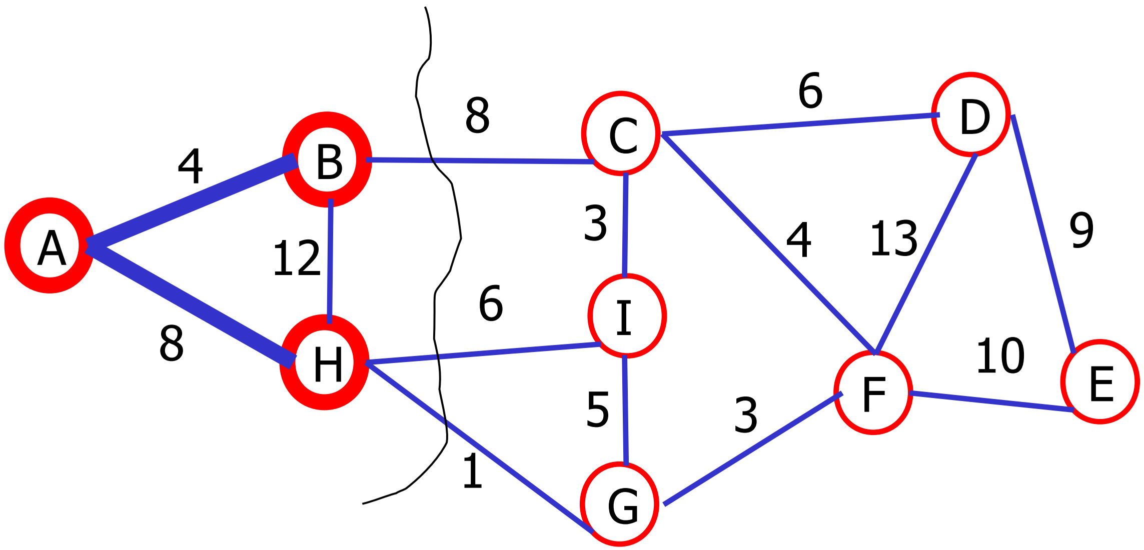
Prim-Jarnik Example/3



A = A-NULL/0, B-A/4

Q = H-A/8, C-B/8, D-NULL/∞, E-NULL/∞,
F-NULL/∞, G-NULL/∞, I-NULL/∞

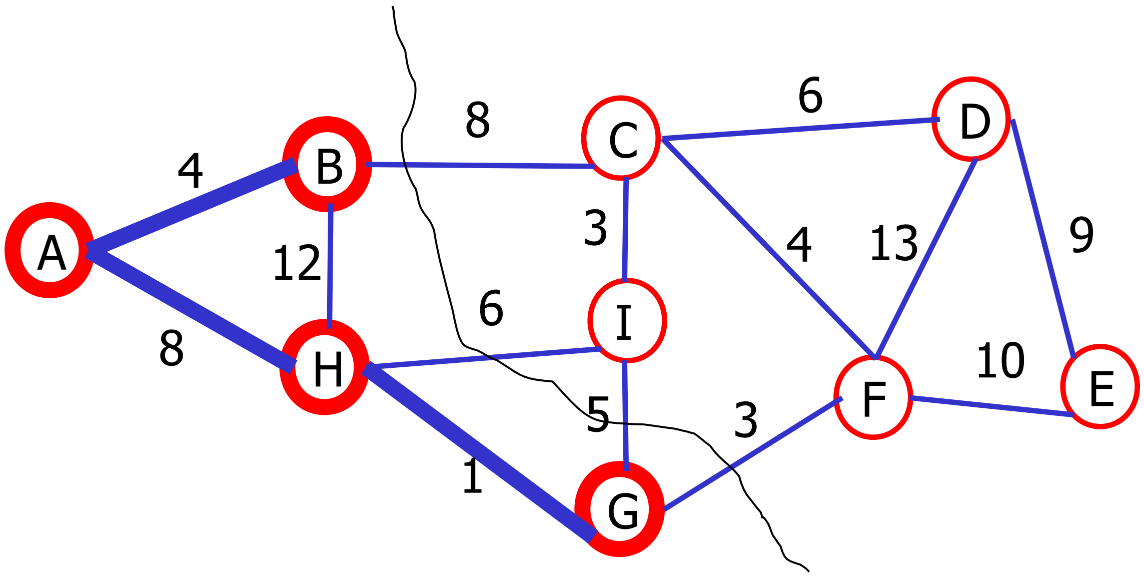
Prim-Jarnik Example/4



A = A-NULL/0, B-A/4, H-A/8

Q = G-H/1, I-H/6, C-B/8, D-NULL/∞, E-NULL/∞, F-NULL/∞

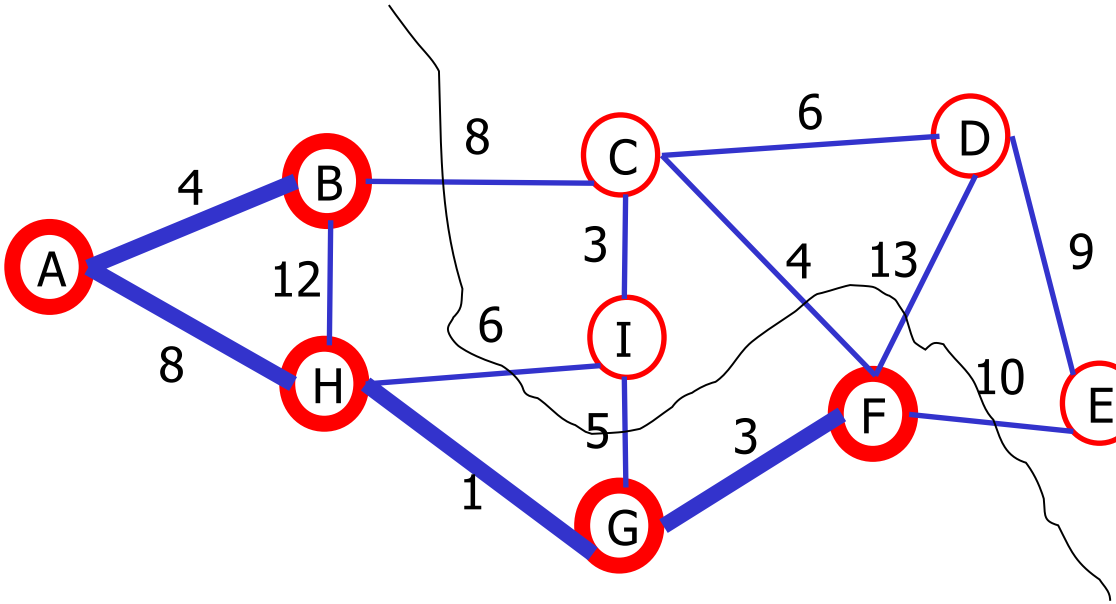
Prim-Jarnik Example/5



A = A-NULL/0, B-A/4, H-A/8, G-H/1

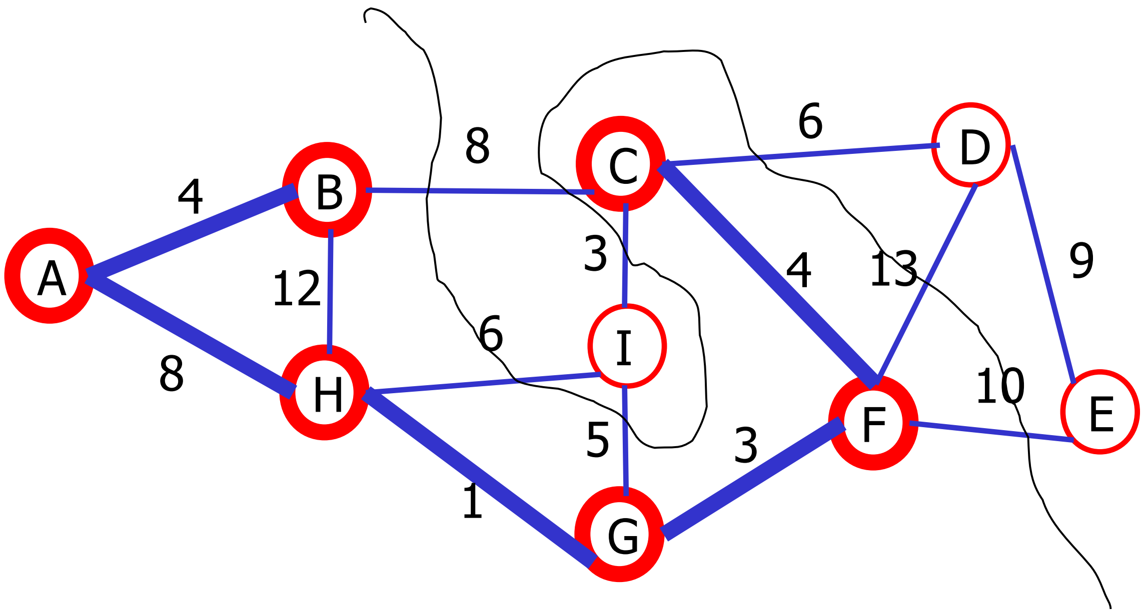
Q = F-G/3, I-G/5, C-B/8, D-NULL/∞, E-NULL/∞

Prim-Jarnik Example/6



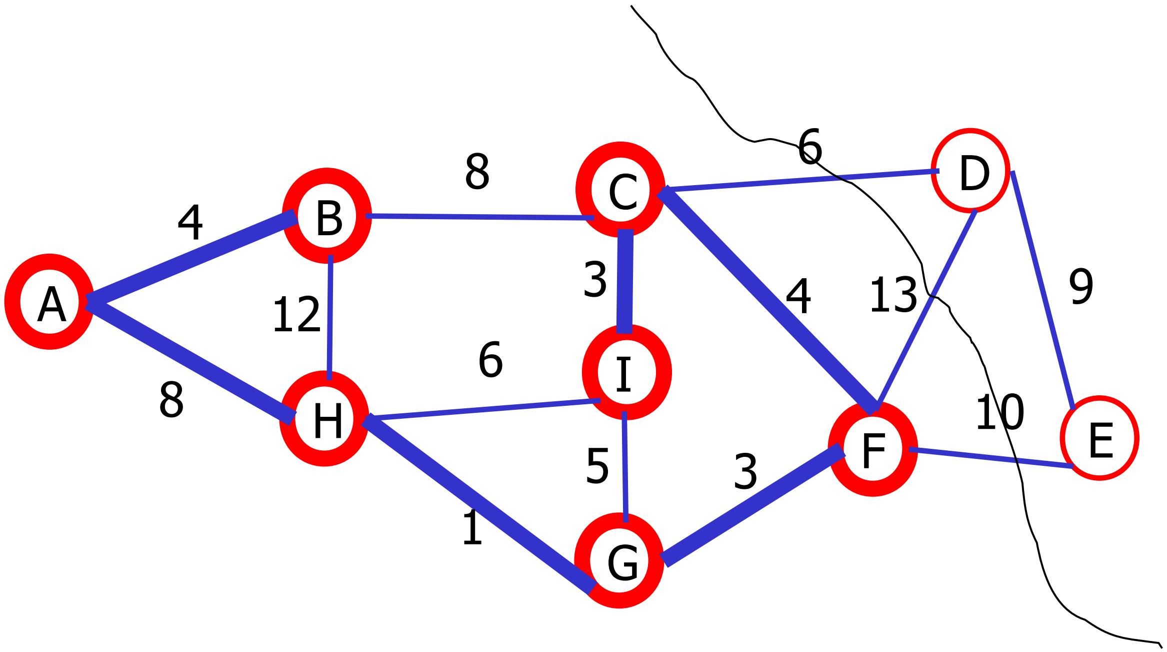
A = A-NULL/0, B-A/4, H-A/8, G-H/1, F-G/3
Q = C-F/4, I-G/5, E-F/10, D-F/13

Prim-Jarnik Example/7



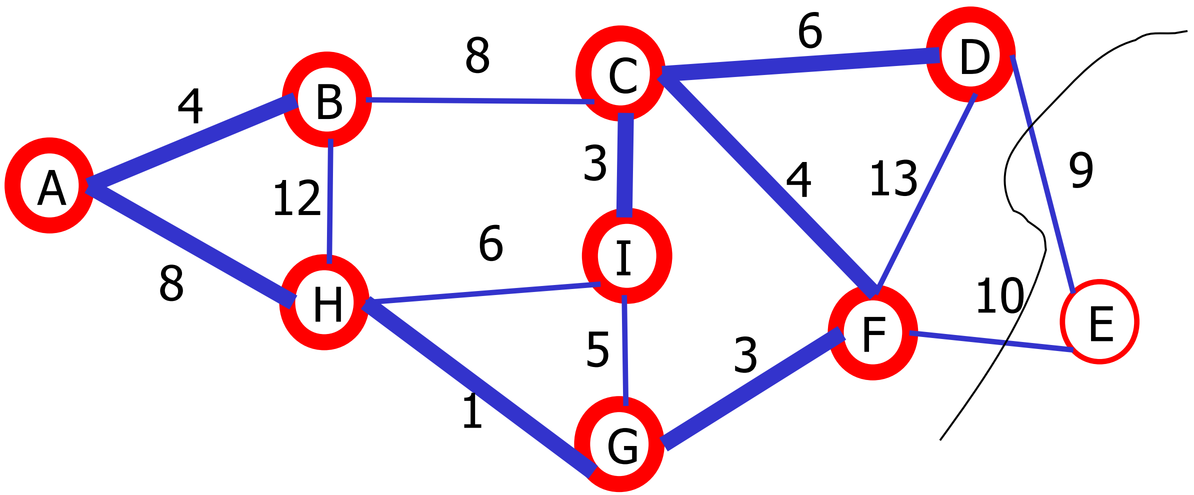
A = A-NULL/0, B-A/4, H-A/8, G-H/1, F-G/3, C-F/4
Q = I-C/3, D-C/6, E-F/10

Prim-Jarnik Example/8



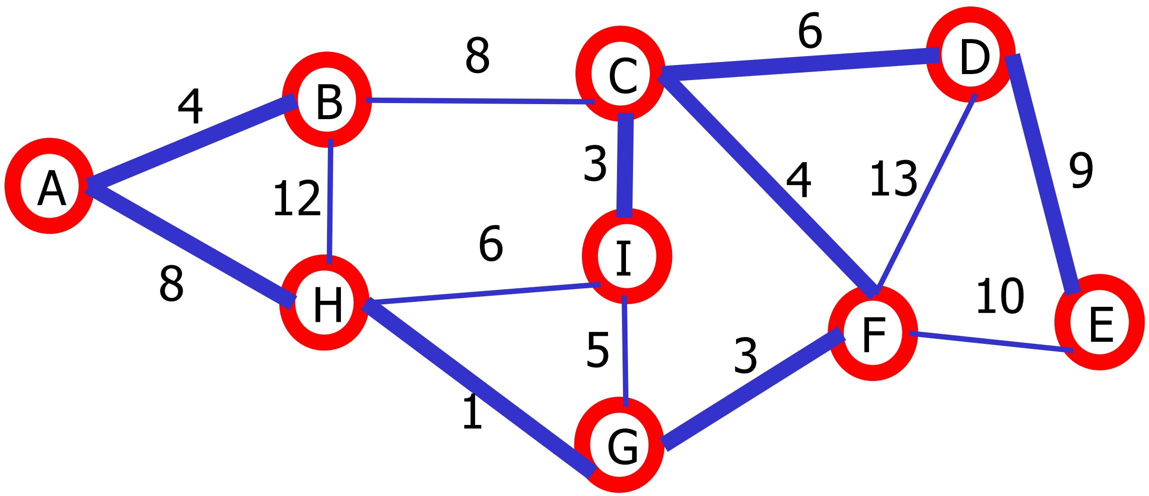
A = A-NULL/0, B-A/4, H-A/8, G-H/1, F-G/3, C-F/4, I-C/3
Q = D-C/6, E-F/10

Prim-Jarnik Example/9



A = A-NULL/0, B-A/4, H-A/8, G-H/1, F-G/3, C-F/4,
I-C/3, D-C/6
Q = E-D/9

Prim-Jarnik Example/10



A = A-NULL/0, B-A/4, H-A/8, G-H/1, F-G/3, C-F/4, I-C/3, D-C/6, E-D/9

Q = {}

Implementation Issues

```
MST-Prim(G,r)
for u ∈ G.V do u.key := ∞; u.pred := NULL
r.key := 0
init(Q, G.V) // Q is a min-priority queue
while not isEmpty(Q) do
  u := extractMin(Q) // add u to T
  for v ∈ u.adj do
    if v ∈ Q and w(u,v) < v.key then
      v.key := w(u,v)
      modifyKey(Q,v)
      v.pred := u
```

Prim-Jarnik Running Time

- Time = $|V| * T(\text{extractMin}) + O(E) * T(\text{modifyKey})$

Q	T(extractMin)	T(modifyKey)	Total
array	$O(V)$	$O(1)$	$O(V^2)$
binary heap	$O(\log V)$	$O(\log V)$	$O(E \log V)$

- $E \geq V-1$, $E < V^2$, $E = O(V^2)$
- Binary heap implementation:
  - Time = $O(V \log V + E \log V) = O(V^2 \log V) = O(E \log V)$

About Greedy Algorithms

- Greedy algorithms make a locally optimal choice (cheapest path, etc).
- In general, a locally optimal choice does not give a globally optimal solution.
- Greedy algorithms can be used to solve optimization problems, if:
  - There is an *optimal substructure*
  - We can prove that a *greedy choice* at each iteration leads to an optimal solution.