JICSCI803
Algorithms and Data Structures
March to June 2020

Highlights of Lecture 07

Greedy Algorithms

Characters of Greedy Algorithms

– These algorithms work by taking what seems to be the best decision at each step
– No backtracking is done (once a choice is made we are stuck with it)
– Easy to design
– Easy to implement
– Efficient (when they work)

Example 1: Making Change

Problem： Given we have \$2, \$1, 50c, 20c, 10c, 5c and 1c coins; what is the best (fewest coins) way to pay any given amount?

- The greedy approach is to pay as much as possible using the larges coin value possible, repeatedly until the amount is paid.
- E.g. to pay \$17.97 we pay 8 \$2 coins, 1 \$1 coin, 1 50c coin, 2 20c coins, 1 5c coin and 2 1c coins(15 coins total).
- This is the optimal solution in required number of coins (although this is harder to prove than you might think).
- Note that this algorithm will not work with an arbitrary set of coin values.
- Adding a 12c coin would result in 15c being made from 1 12c and 3 1c (4 coins) instead of 1 10c and 1 5c coin (2 coins).

Greedy Algorithms: selected or rejected method

- We start with a set of candidates which have not yet been considered for the solution
- As we proceed, we construct two further sets:
  - Candidates that have been considered and selected
  - Candidates that have been considered and rejected
- At each step we check to see if we have reached a solution
- At each step we also check to see if a solution can be reached at all
- At each step we select the best acceptable candidate from the unconsidered set and move it into the selected set
- We also move any unacceptable candidates into the rejected set

Example 2: Shortest Path

- Let $G = (N, E)$ be a connected, directed graph consisting of a set of nodes N and a set of directed edges E.
- Each edge has a length, the distance from the node at one end of the edge to the node at the other end.
- One node is designated the source node
- The problem is to find the shortest path from the source node to each of the other nodes

Example 2: Shortest Path

Application

- In a graph in which edges have costs ..
- Find the shortest path from a source to a destination
- *Surprisingly ..*
  - While finding the shortest path from a source to one destination,
  - *we can find the shortest paths to all over destinations as well!*
- Common algorithm for
  - single-source shortest paths
- is due to Edsger Dijkstra

Dijkstra's Algorithm—DS design

For a graph,

$$G = (V, E)$$

Dijkstra's algorithm keeps *two* sets of vertices:

S Vertices whose shortest paths have already been determined

Q=V-S Remainder

Also

d Best estimates of shortest path to each vertex

π Predecessors for each vertex

Predecessor Sub-graph

Array of vertex indices, $\pi[j]$, $j = 1 .. |V|$

$\pi[j]$ contains the predecessor for node j

All j's predecessors is are $\pi[\pi[j]]$, and so on ....

The edges in the predecessor subgraph are

$$(\pi[j], j)$$

Dijkstra's Algorithm - Operation

Initialise d and π

For each vertex, j, in V

$d_j = \infty$

Initial estimates are all ∞

π_j = nil

No connections

Source distance, $d_s = 0$

Set S to empty

While V-S is not empty

Sort V-S based on d

Add u, the closest vertex in V-S, to S

Add s first!

Relax all the vertices still in V-S connected to u

Dijkstra's Algorithm - Operation

The Relaxation process

Relax the node v attached to node u

Edge cost matrix

```
relax( Node u, Node v, double w[][] )
    if d[v] > d[u] + w[u,v] then
       d[v] := d[u] + w[u,v]
       π[v] := u
```

If the current best estimate to v is greater than the path through u ..

Update the estimate to v

Make v's predecessor point to u

Dijkstra's Algorithm - Full

Given a graph, g, and a source, s

```
shortest_paths( Graph g, Node s )
  initialise_single_source( g, s )
  S := { 0 }       /*Make S empty*/
  Q := Vertices(g) /*Put the vertices in a PQ*/
  while not Empty(Q)
      u := ExtractCheapest( Q );
      AddNode( S, u ); /* Add u to S */
      for each vertex v in Adjacent( u )
          relax( u, v, w )
```

Dijkstra's Algorithm - Initialise

Given a graph, g, and a source, s

Initialise **d, π, S,** vertex **Q**

```
shortest_paths( Graph g, Node s )
    initialise_single_source( g, s )
    S := { 0 }         /* Make S empty */
    Q := Vertices( g ) /* Put the vertices in a PQ */
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )
```

Dijkstra's Algorithm - Loop

The Shortest Paths algorithm

Given a graph, g, and a source, s

```
shortest_paths(                          s )
    initialise_                          g, s )
    S := { 0 }                        Make S empty */
    Q := Vertices( g ) /* Put the vertices in a PQ */
    while not Empty(Q)
        u := ExtractCheapest( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )
```

While there are still nodes in Q

Greedy!

Dijkstra's Algorithm - Relax neighbours

The Shortest Paths algorithm

Given a graph, g, and a source, s

Update the estimate of the shortest paths to all nodes attached to u

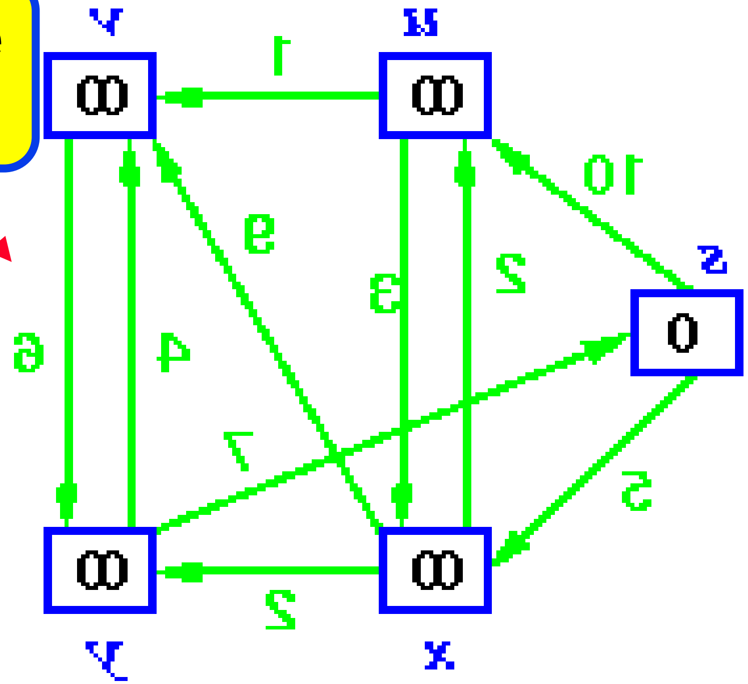
```
shortest_paths( G ... )
    initialise_si... s )
    S := { 0 }   ...empty */
    Q := Vertices( g ) /* Put the vertices in a PQ */
    while not Empty(Q)
        u := ExtractChe...st( Q );
        AddNode( S, u ); /* Add u to S */
        for each vertex v in Adjacent( u )
            relax( u, v, w )
```

Greedy!

Dijkstra’s Algorithm - Operation

Initial Graph

Source Mark 0

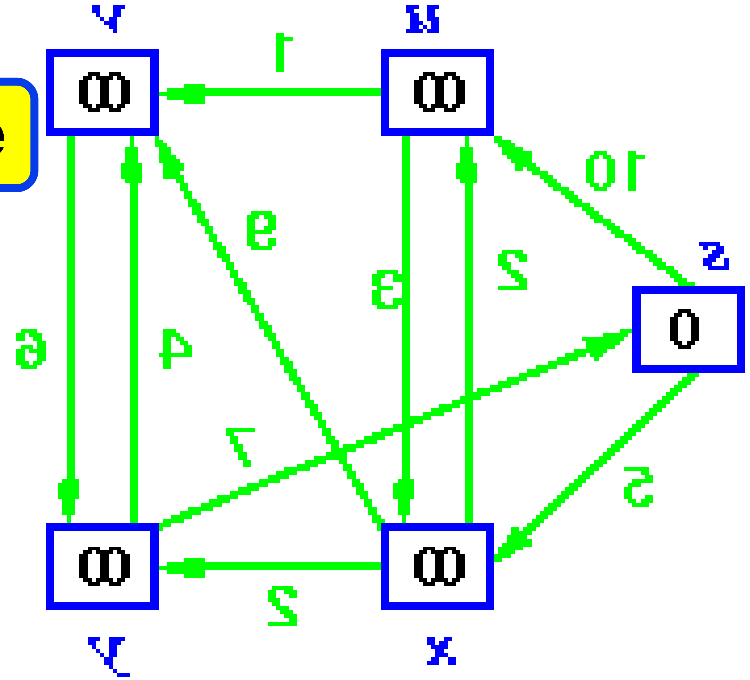


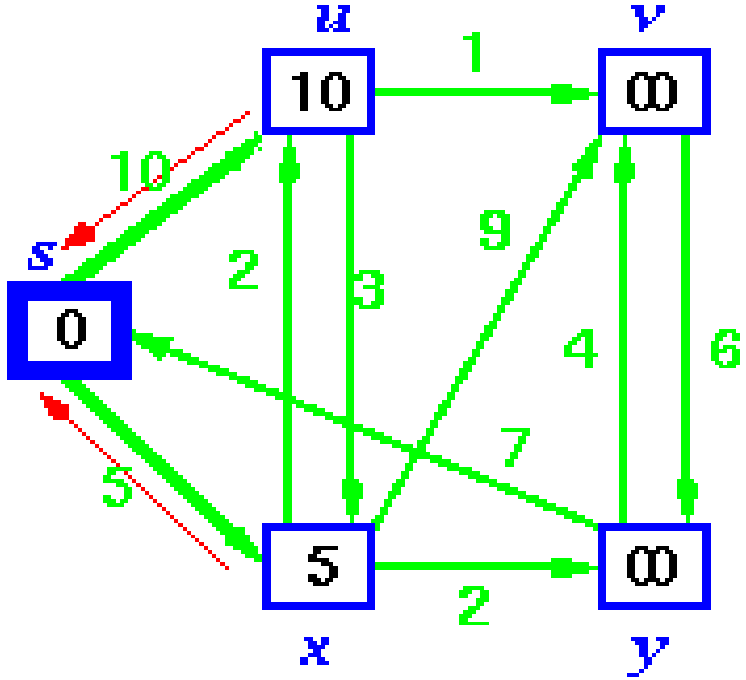
Distance to all nodes marked ∞

Dijkstra's Algorithm - Operation

Initial Graph

Source

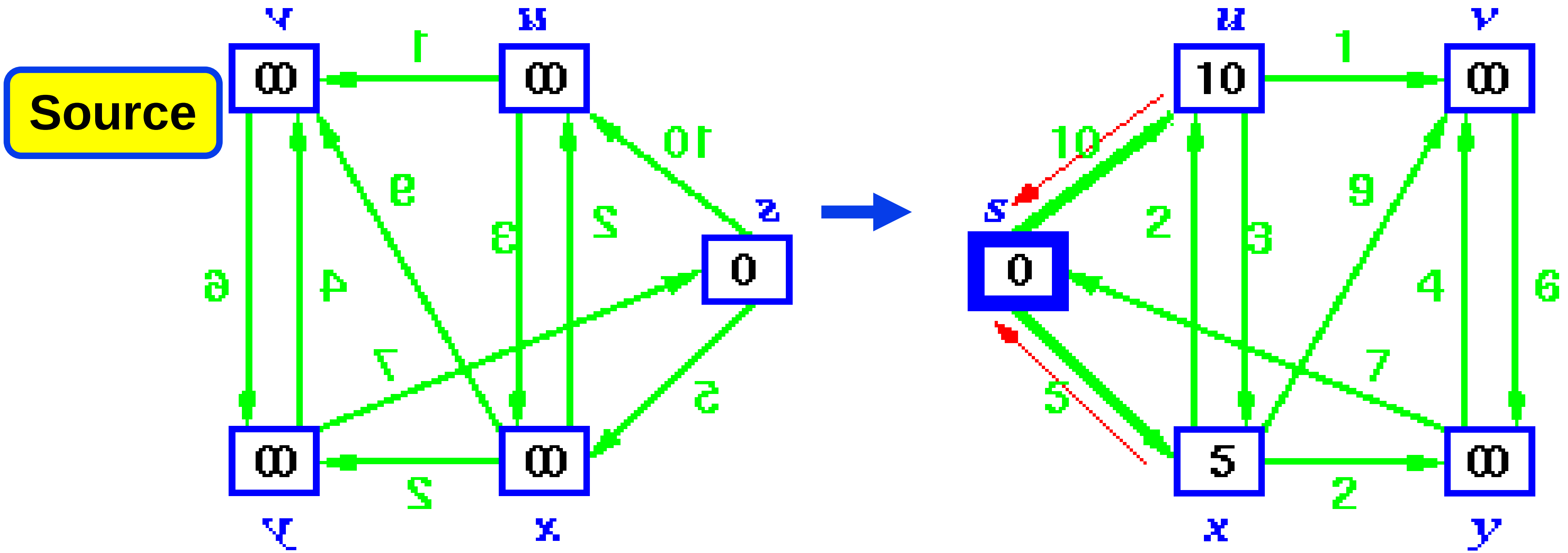




Relax vertices adjacent to source

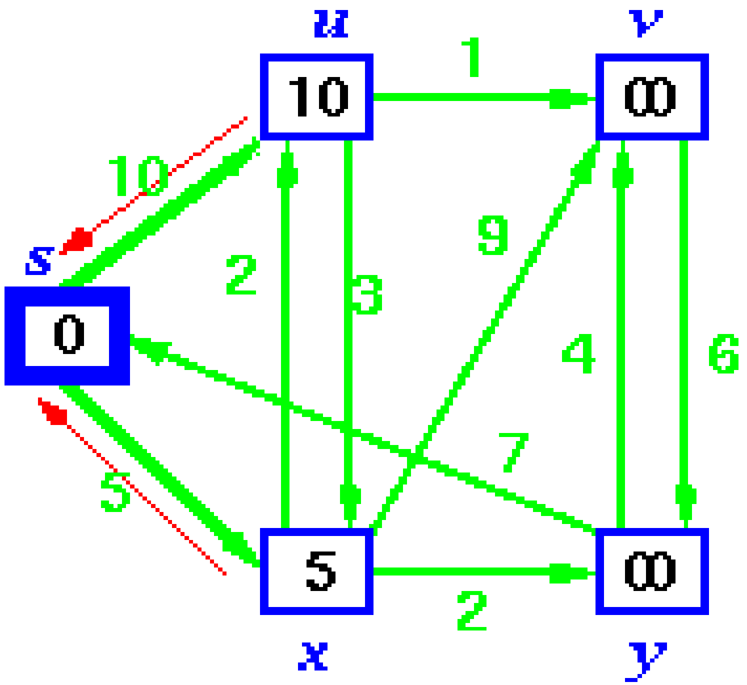
Dijkstra’s Algorithm - Operation

Initial Graph

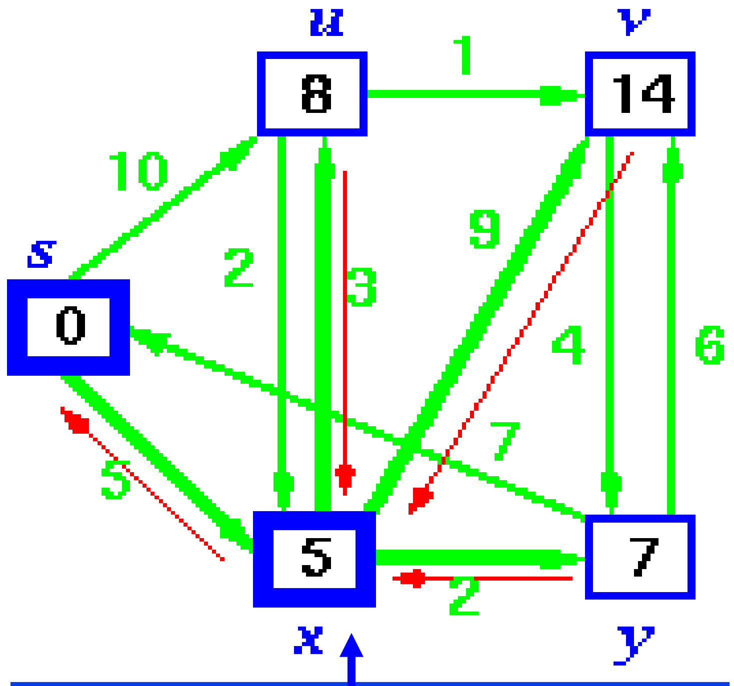


Red arrows show pre-decessors

Dijkstra's Algorithm - Operation



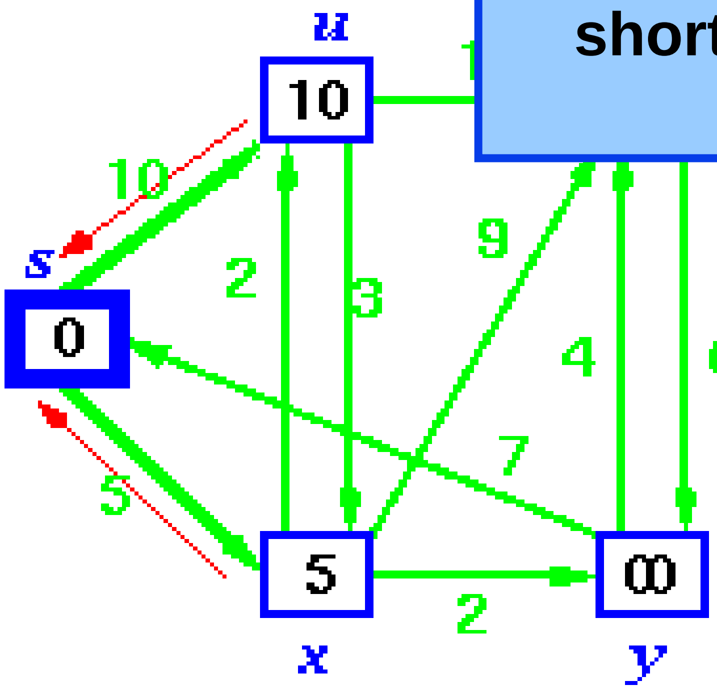
Source is now in *S*



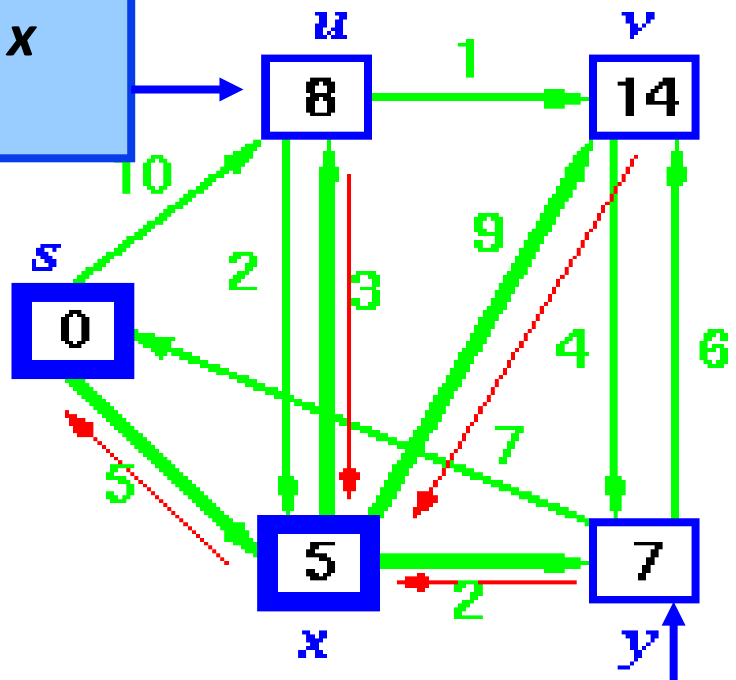
Sort vertices and choose closest

Dijkstra's Algorithm - Operation

Relax *u* because a shorter path via *x* exists

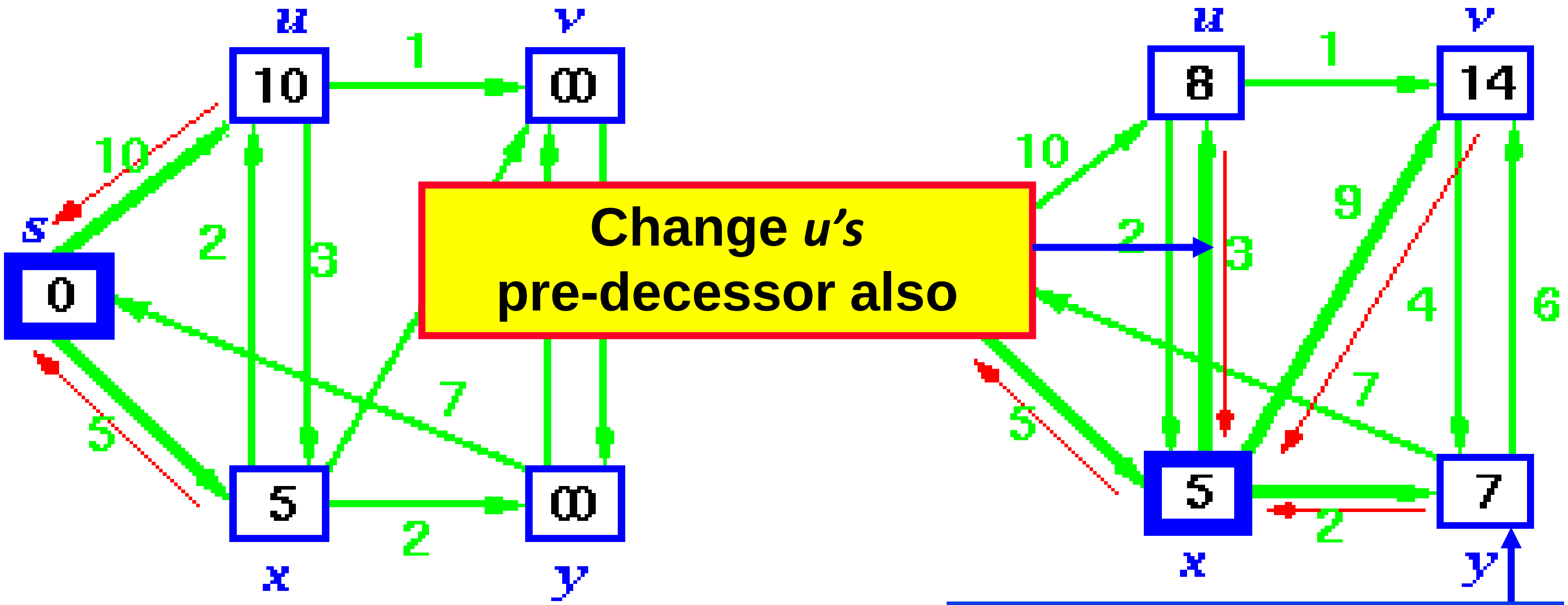


Source is now in *S*



Relax *y* because a shorter path via *x* exists

Dijkstra's Algorithm - Operation

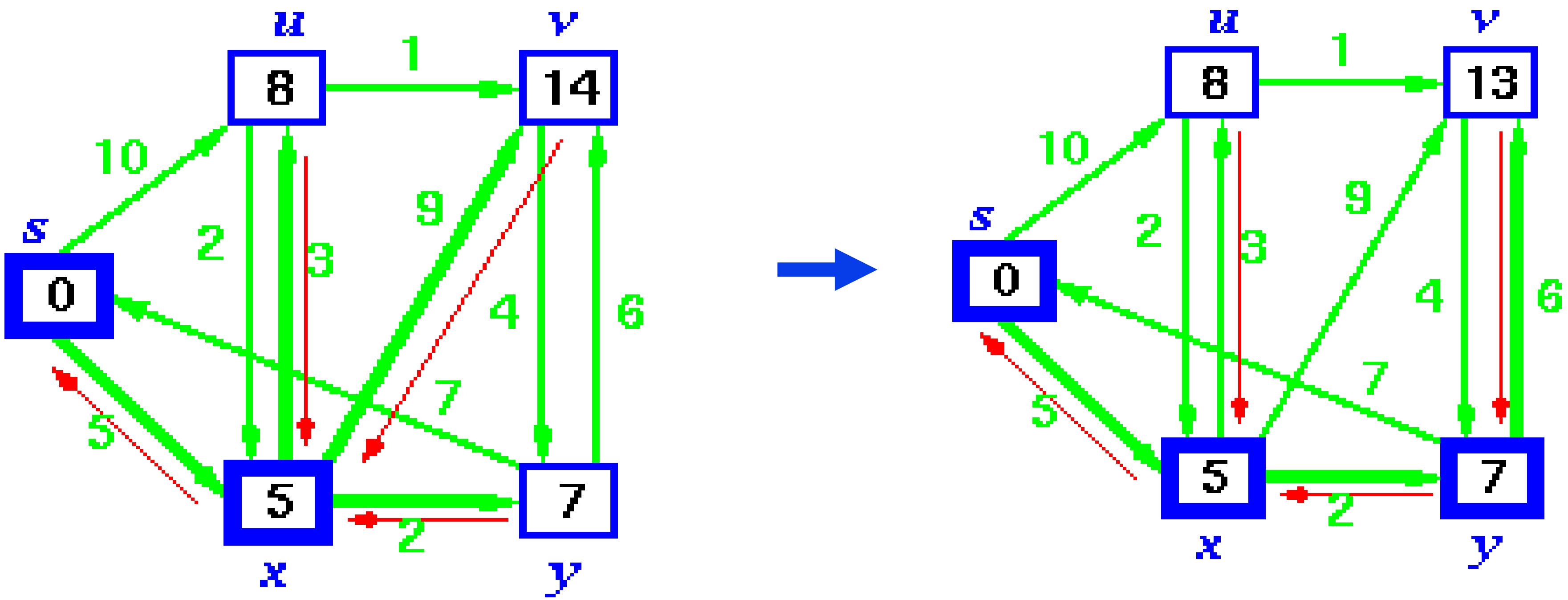


Change *u's* pre-decessor also

Source is now in *S*

Relax *y* because a shorter path via *x* exists

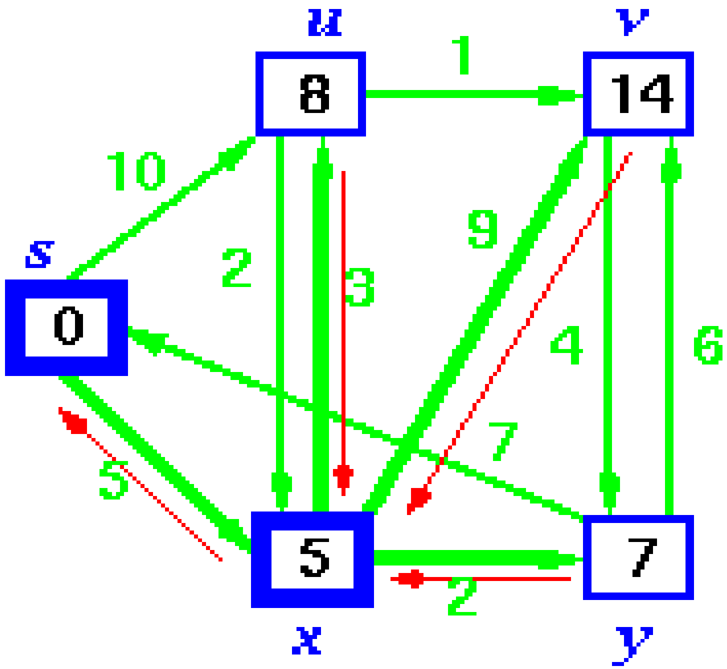
Dijkstra’s Algorithm - Operation



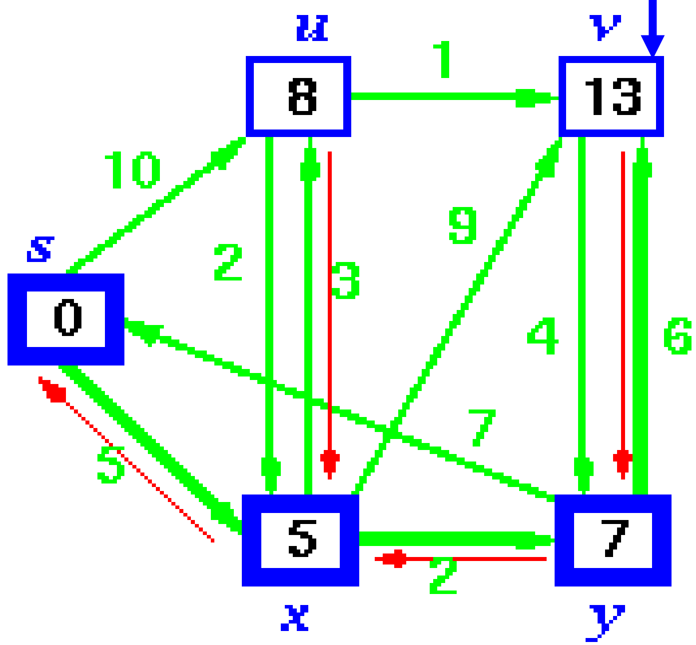
S is now { *s*, *x* }

Sort vertices and choose closest

Dijkstra's Algorithm - Operation



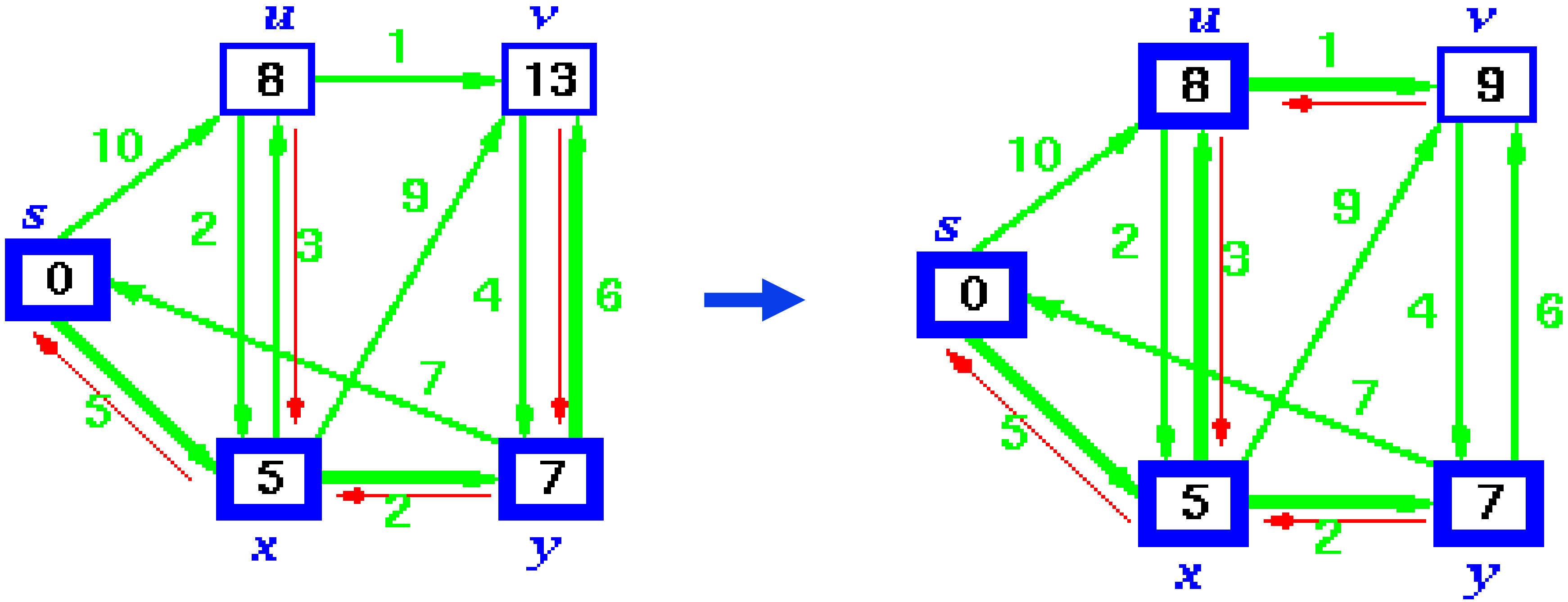
S is now { *s*, *x* }



Relax *v* because a shorter path via *y* exists

Sort vertices and choose closest

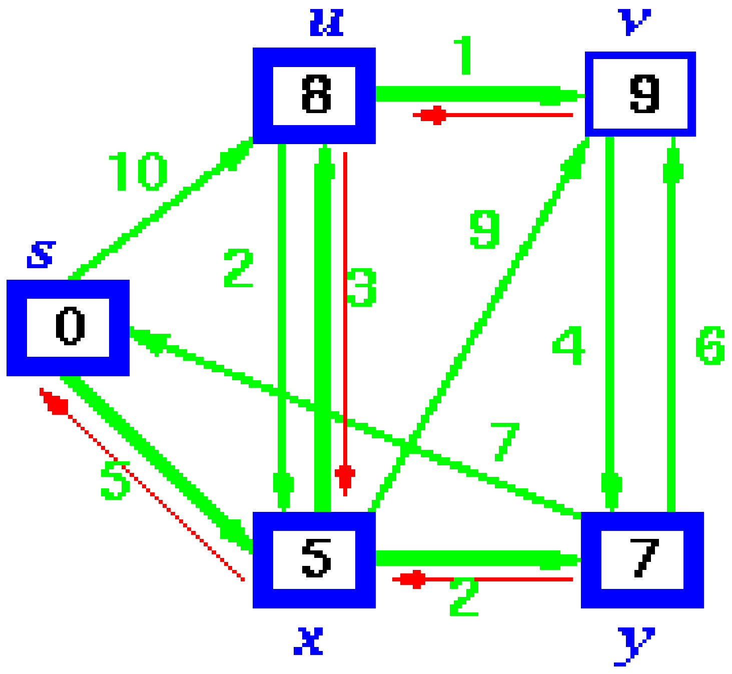
Dijkstra's Algorithm - Operation



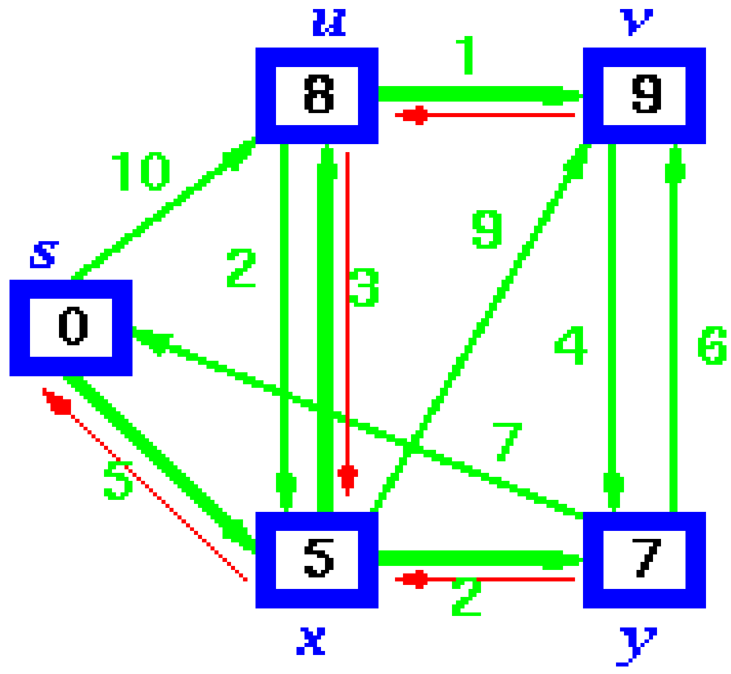
S is now { *s, x, y* }

Sort vertices and choose closest, *u*

Dijkstra's Algorithm - Operation

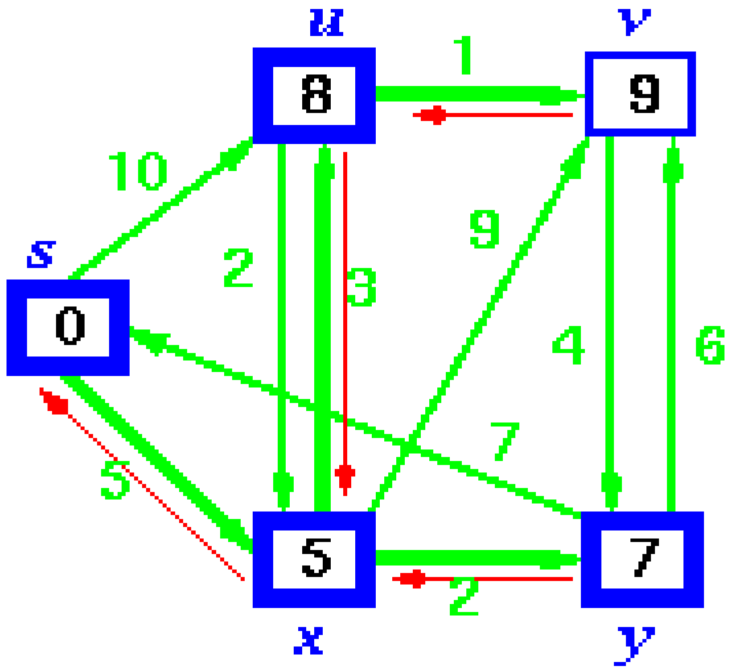


S is now { *s, x, y, u* }

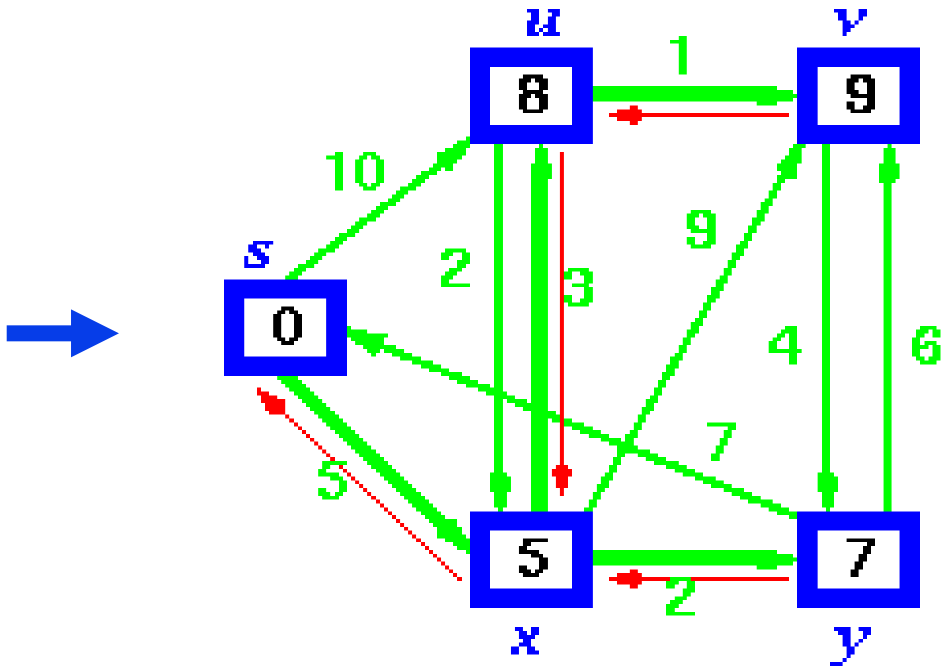


Finally add *v*

Dijkstra's Algorithm - Operation



S is now { *s, x, y, u* }



Pre-decessors show shortest paths sub-graph

Dijkstra's Algorithm - Proof

- Greedy Algorithm
  - Proof by contradiction test
- Lemma 1
  - Shortest paths are composed of shortest paths
- Proof
  - If there was a shorter path than any sub-path, then substitution of that path would make the whole path shorter

Dijkstra’s Algorithm – Correctness Proof

Denote

$\delta(s,v)$ - the cost of the shortest path from s to v

Lemma 2

If $s \rightarrow \ldots \rightarrow u \rightarrow v$ is a shortest path from s to v, then after u has been added to S and relax(u,v,w[][]) called, $d[v] = \delta(s,v)$ and $d[v]$ is not changed thereafter.

Proof

Follows from the fact that at all times $d[v] \geq \delta(s,v)$

See Cormen (or any other text) for the details.

Dijkstra's Algorithm - Proof

Using Lemma 2

- After running Dijkstra's algorithm, we assert
- $d[v] = \delta(s,v)$ for all v

Proof *(by contradiction)*

- Suppose that u is the first vertex added to S for which
- $d[u] \neq \delta(s,u)$
- Note
  - v is not s because $d[s] = 0$
  - There must be a path $s \rightarrow \ldots \rightarrow u$,
  - otherwise $d[u]$ would be ∞
  - Since there's a path, there must be a shortest path

Dijkstra’s Algorithm - Proof

Proof *(by contradiction)*

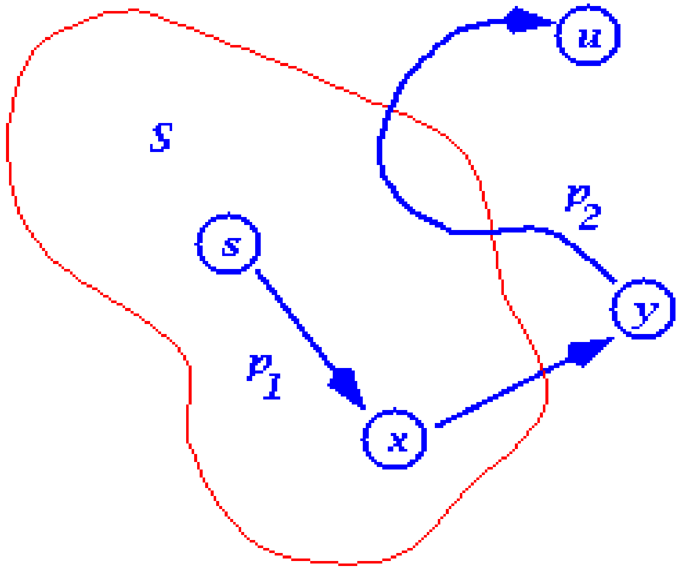
Suppose that u is the first vertex added to S for which $d[u] \neq \delta(s,u)$

Let $s \rightarrow x \rightarrow y \rightarrow u$ be the shortest path $s \rightarrow u$,

where x is in S and y is the first outside S

When x was added to S, $d[x] = \delta(s,x)$

Edge $x \rightarrow y$ was relaxed at that time, so $d[y] = \delta(s,y)$



Proof *(by contradiction)*

Edge $x \rightarrow y$ was relaxed at that time,

so $d[y] = \quad \delta(s,y)$

$\quad\quad\quad \leq \ \delta(s,u) \ \leq \ d[u]$

But, when we chose u,

both u and y where in V-S,

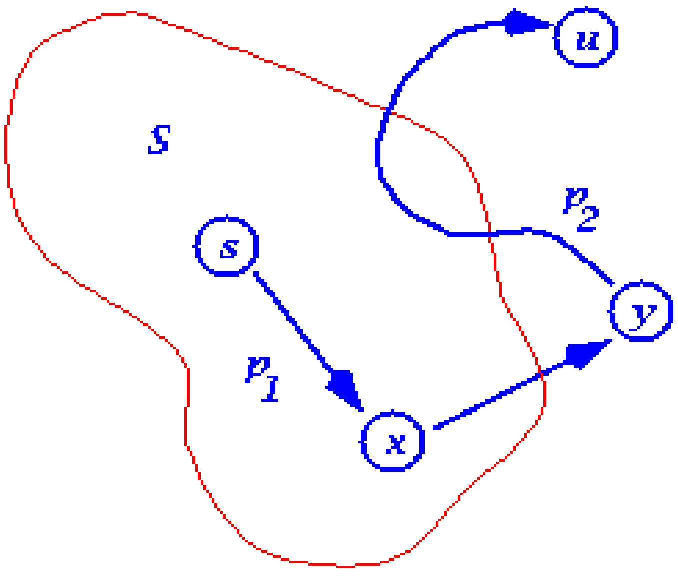
so $d[u] \leq d[y]$

(otherwise we would have chosen y)

Thus the inequalities must be equalities

$\therefore \ d[y] = \delta(s,y) = \delta(s,u) \ = d[u]$

And our hypothesis ($d[u] \neq \delta(s,u)$) is contradicted!

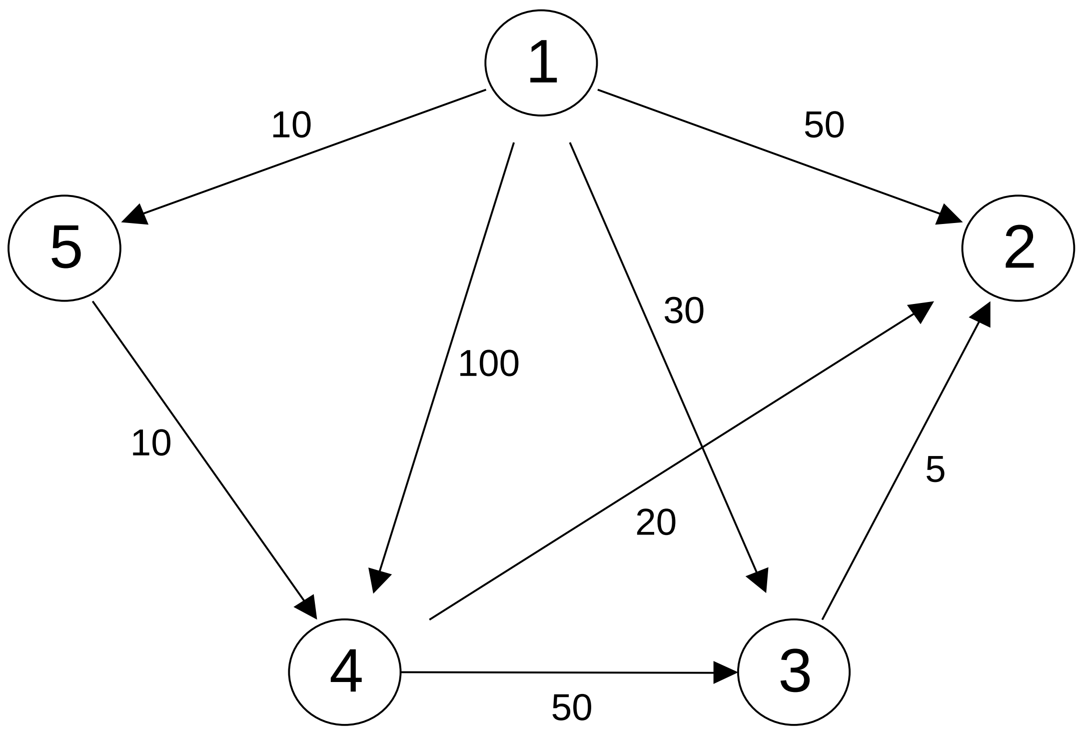


Dijkstra's Algorithm - Time Complexity

- Dijkstra's Algorithm
  - Key step is sort on the edges
  - Complexity is
    - $O((|E|+|V|)\log|V|)$ *or*
    - $O(n^2 \log n)$
    - for a dense graph with $n = |V|$

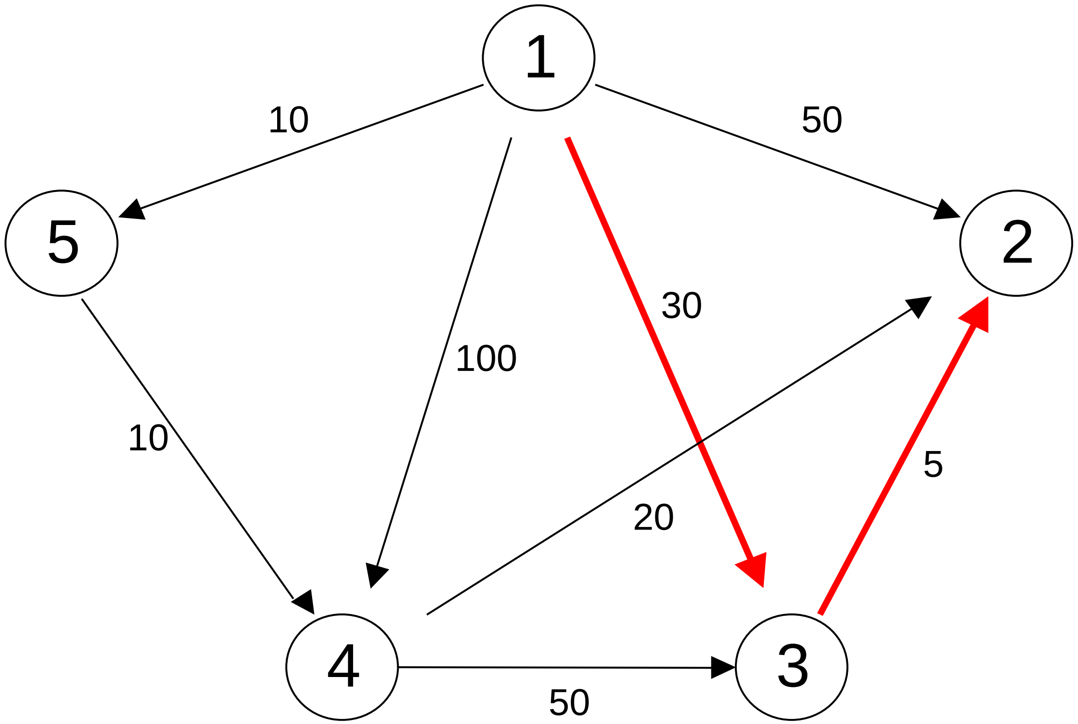
Example 2: Shortest Path

Node 1 is Source



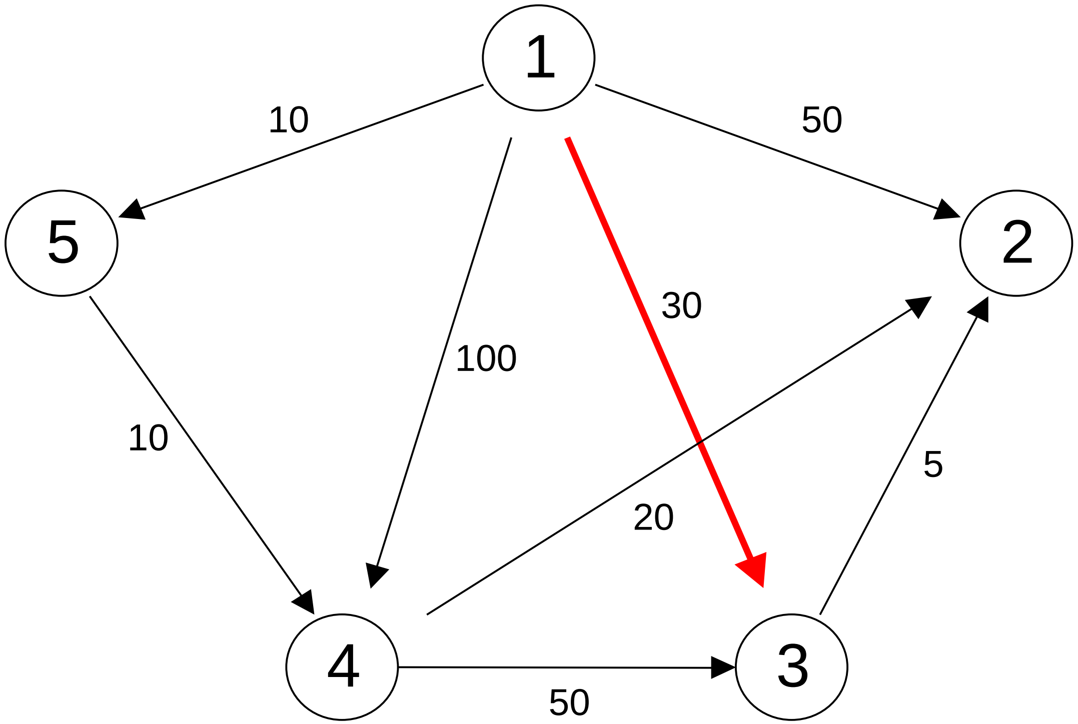
Example 2: Shortest Path

– 1 to 2, length 35



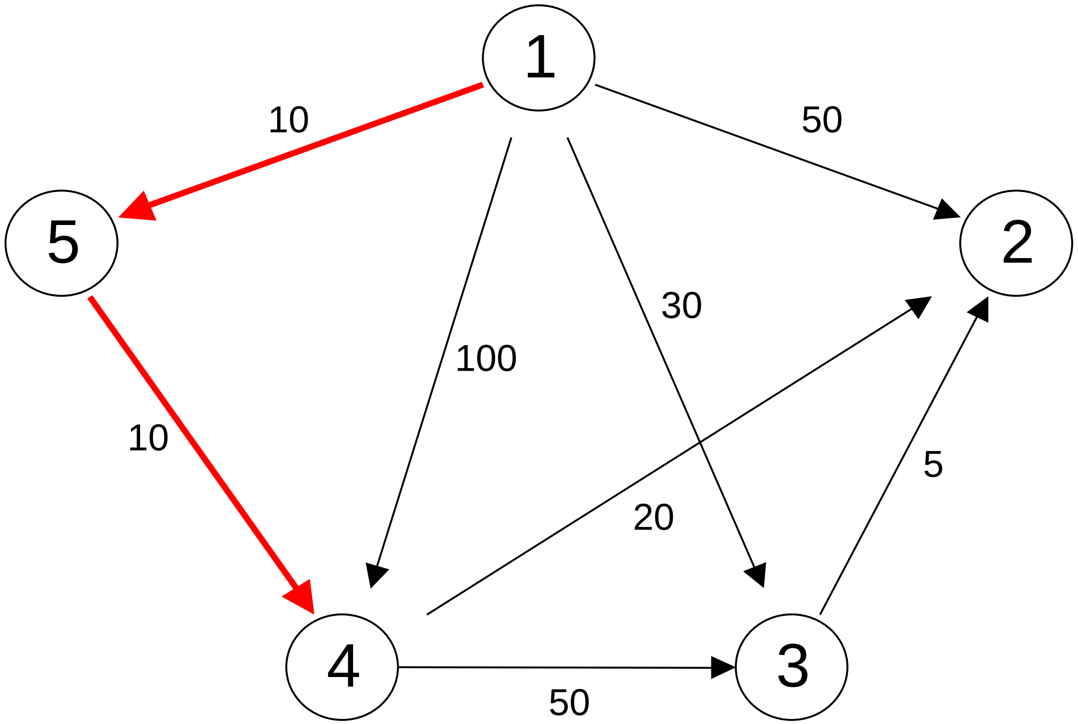
Example 2: Shortest Path

– 1 to 3 length 30



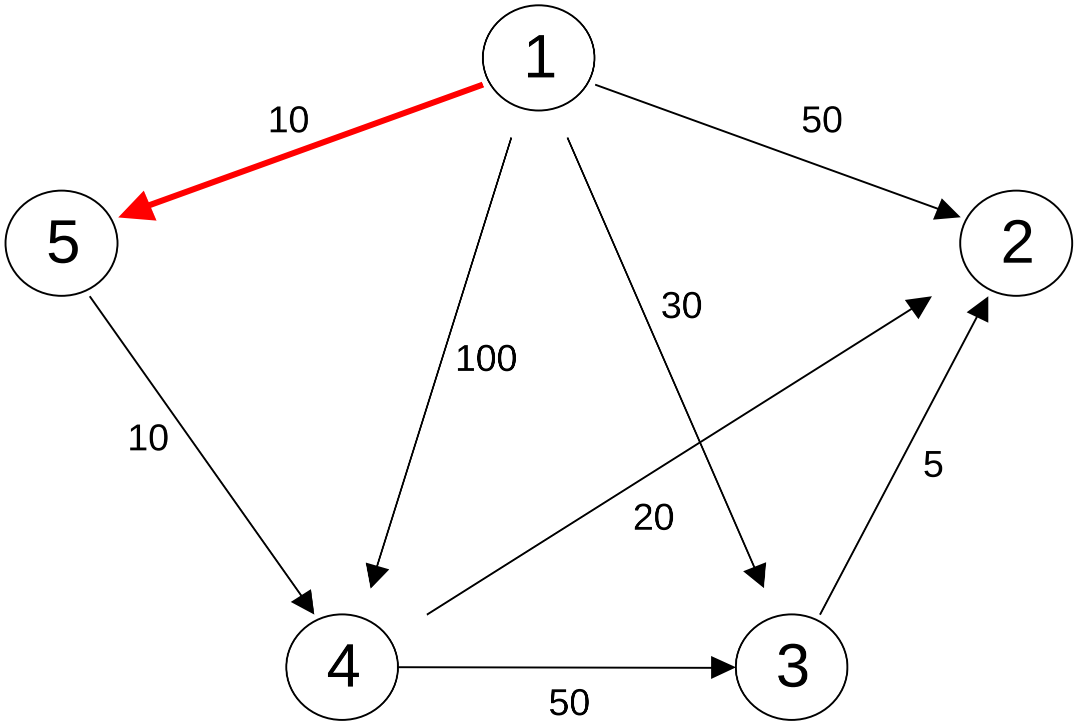
Example 2: Shortest Path

– 1 to 4 length 20



Example 2: Shortest Path

– 1 to 5 length 10



Example 2: Shortest Path

–Dijkstra's Algorithm

–Uses two sets of nodes *S* and *C*

–At each iteration *S* contains the set of nodes that have already been chosen

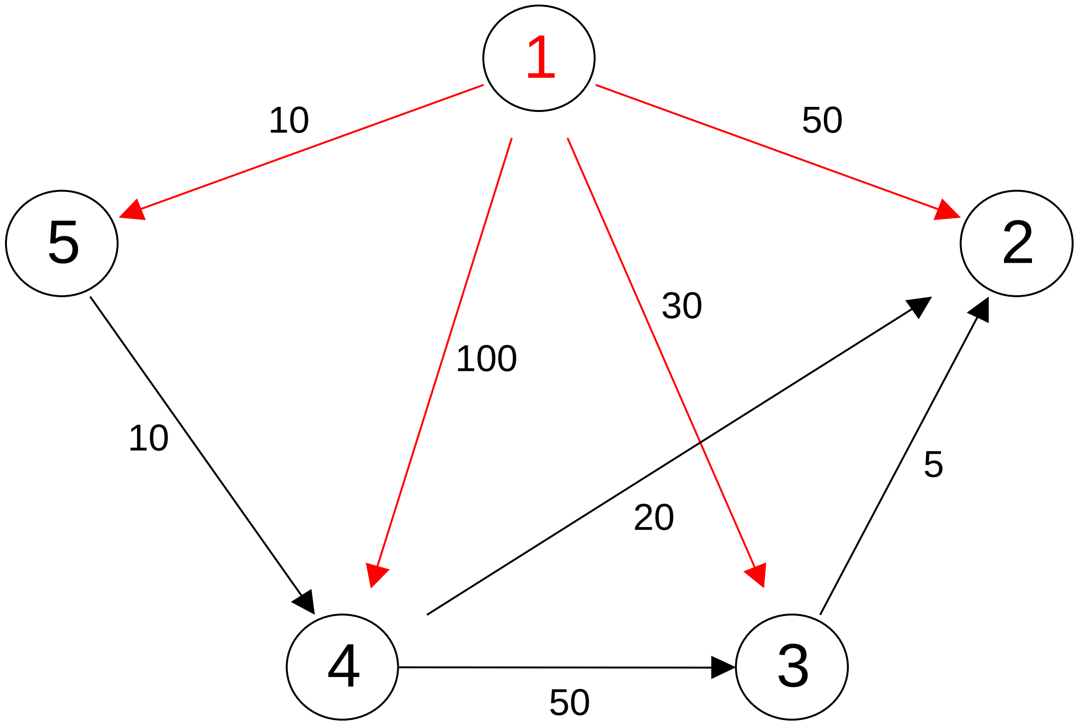
–At each iteration *C* contains the set of nodes that have not yet been chosen

–At each step we move the node which is cheapest to reach from C to *S*

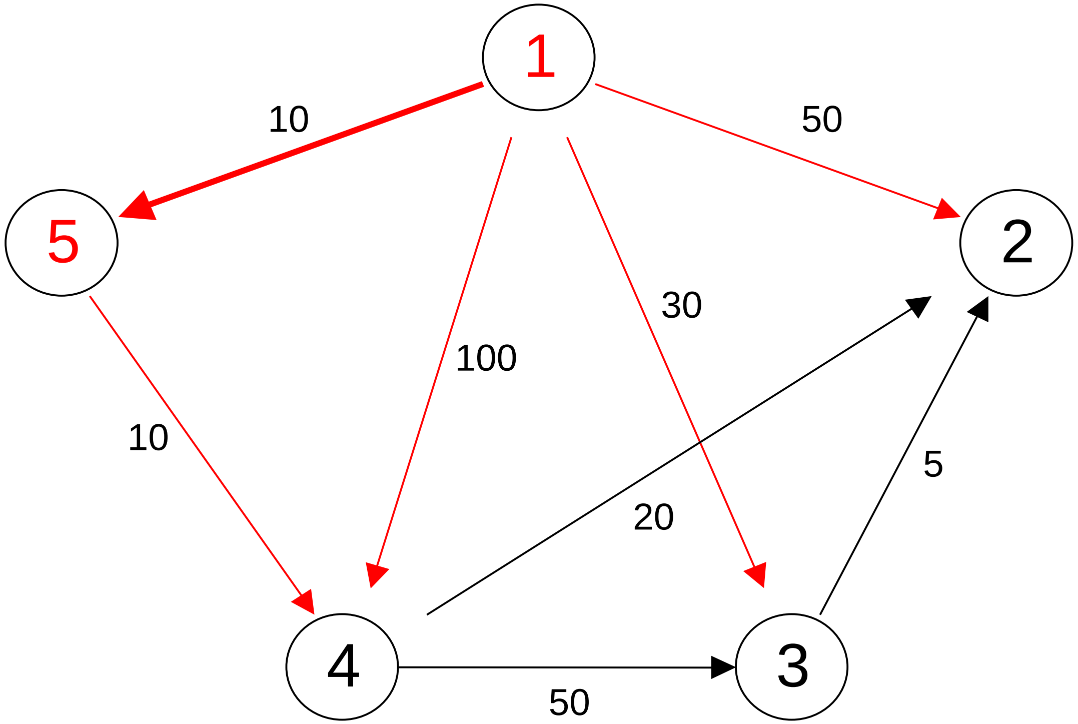
–An array *D* contains the shortest path so far from the source to each node

Example 2: Shortest Path

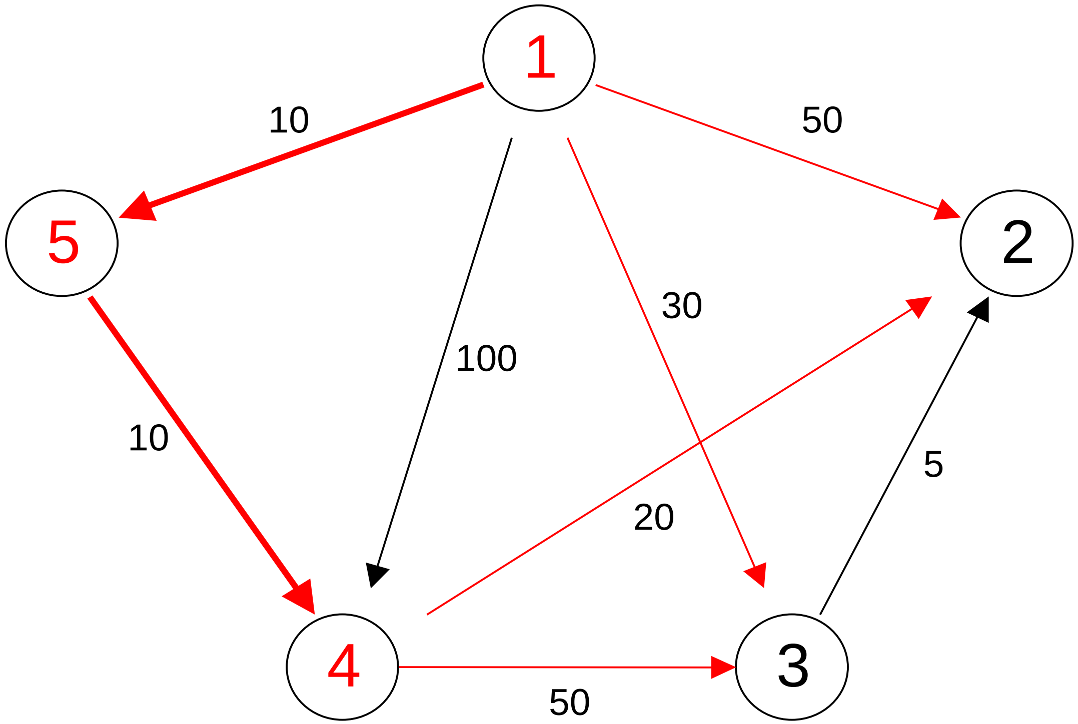
- Dijkstra's Algorithm: An Example
  – Step 0 S = {1} C = {2, 3, 4, 5} D = [50, 30, 100, 10]



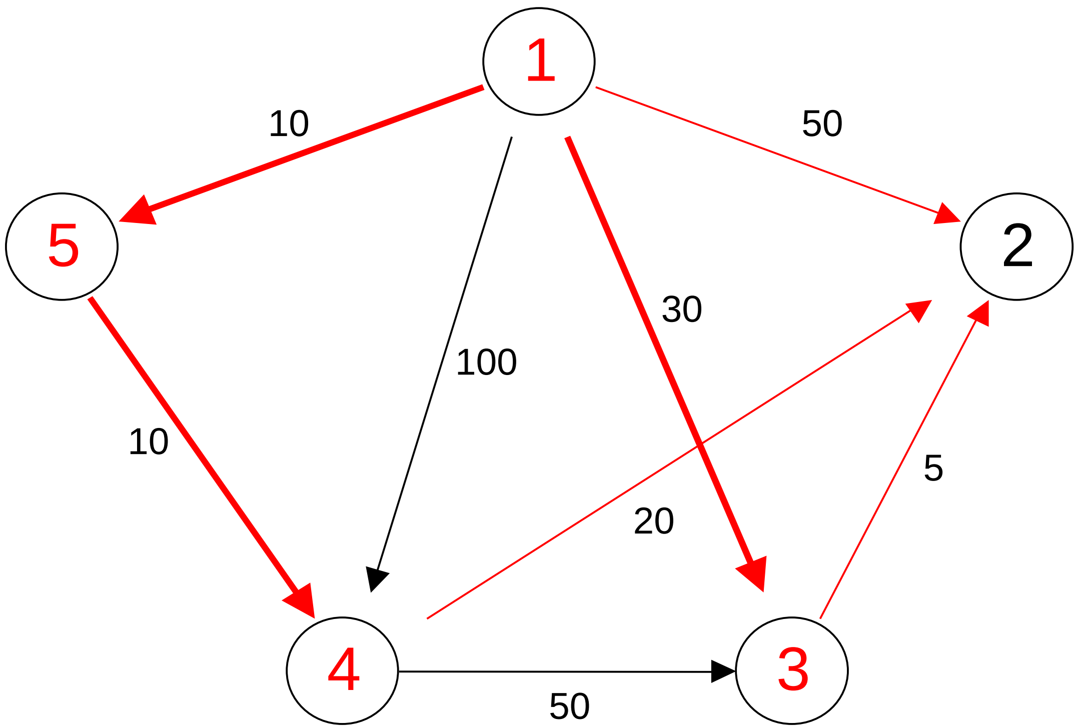
- Dijkstra's Algorithm: An Example
  - Step 1 S = {1,5} C = {2, 3, 4} D = [50, 30, 20, 10]



- Dijkstra’s Algorithm: An Example
  - Step 2   S = {1,4,5}   C = {2, 3}   D = [40, 30, 20, 10]



- Dijkstra’s Algorithm: An Example
  - Step 3 S = {1,3,4,5} C = {2} D = [35, 30, 20, 10]



Dijkstra’s Algorithm

```
Function Dijkstra(L[1..n, 1..n]): array [2..n]
  array D[2..n]
  C = {2, 3, …, n}
  for i = 2 to n do
    D[i] = L[1, i]
  repeat n – 2 times
    v = the index of the minimum D[v] not yet selected
    remove v from C      // and implicitly add v to S
    for each w
        D[w] = min(D[w], D[v] + L[v, w])
  return D
```

Dijkstra’s Algorithm (recorded paths)

```
Function Dijkstra(L[1..n, 1..n]): array [2..n]
   array D[2..n], P[2..n]
   C = {2, 3, …, n}
   for i = 2 to n do
      D[i] = L[1, i]
      P[i] = 1
   repeat n – 2 times
     v = the index of the minimum D[v] not yet selected
     remove v from C        // and implicitly add v to S
     for each w ∈ C do
         if (D[w] > D[v] + L[v, w]) then
              D[w] = D[v] + L[v, w]
              P[w] = v

    return D, P
```

Dijkstra's Algorithm: at start

L =

∞	∞	∞	∞	∞
50	∞	5	20	∞
30	∞	∞	50	∞
100	∞	∞	∞	10
10	∞	∞	∞	∞

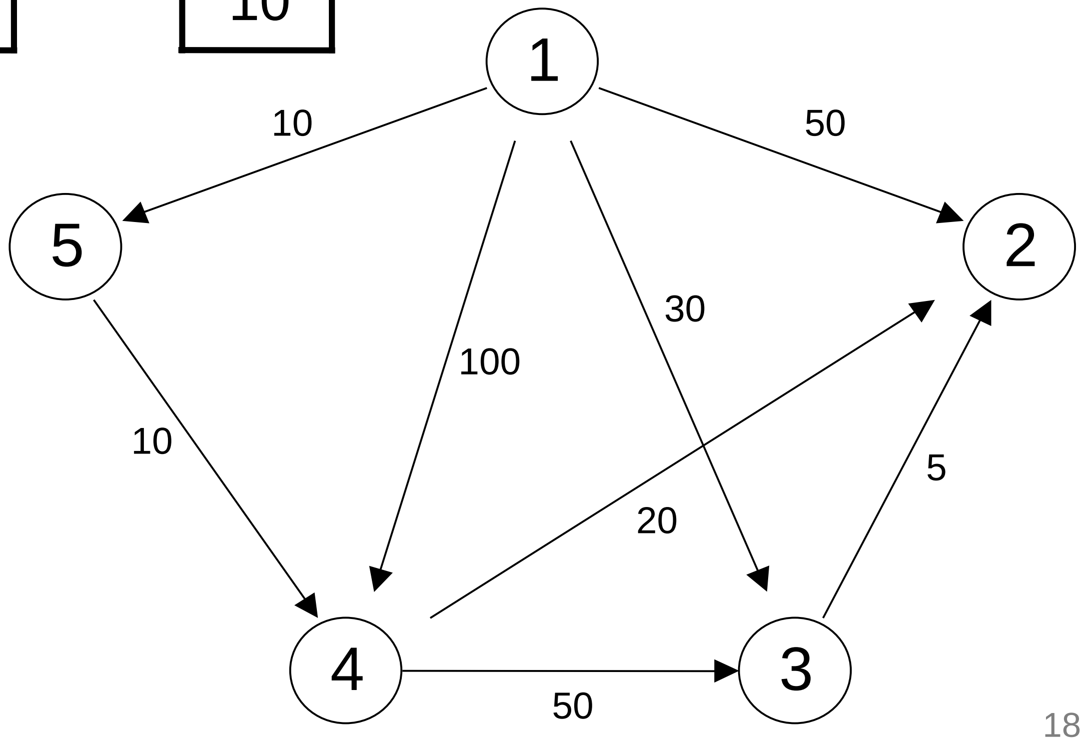
P =

1
1
1
1

D =

50
30
100
10

C = {2, 3, 4, 5}



Dijkstra's Algorithm: at start

L =

∞	∞	∞	∞	∞
50	∞	5	20	∞
30	∞	∞	50	∞
100	∞	∞	∞	10
10	∞	∞	∞	∞

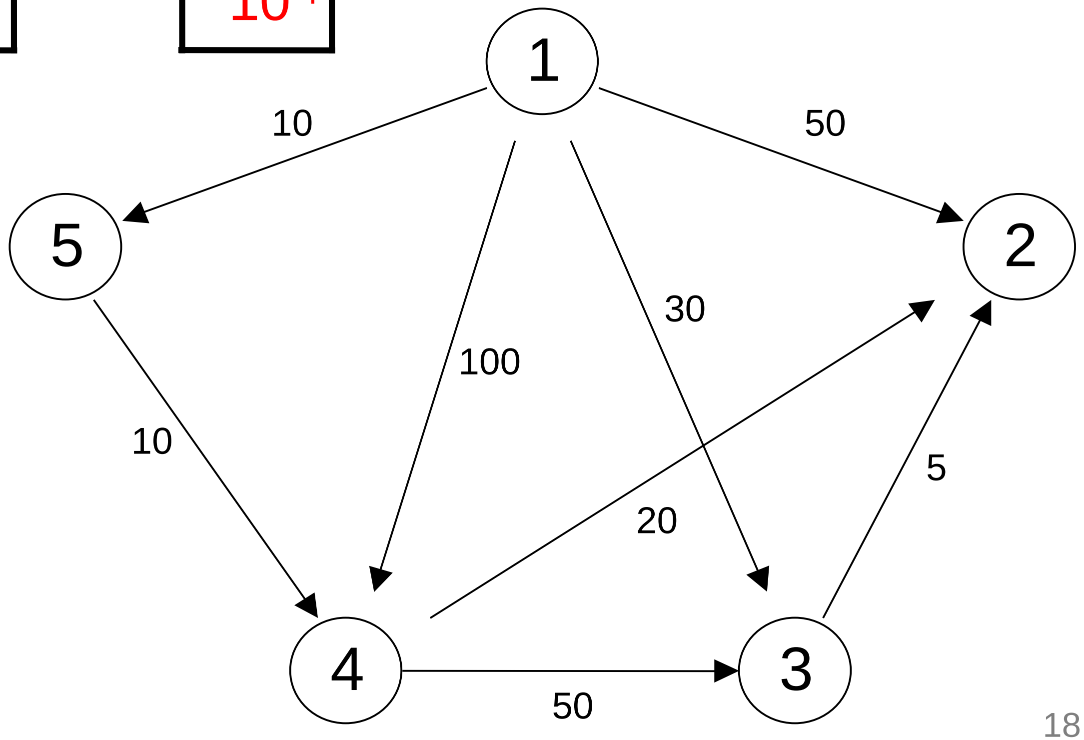
P =

1
1
1
1

D =

50
30
100
10

v = 5
C = {2, 3, 4, 5} => {2, 3, 4}
S = {1}



Dijkstra's Algorithm: after iteration 1

L =

∞	∞	∞	∞	∞
50	∞	5	20	∞
30	∞	∞	50	∞
100	∞	∞	∞	10
10	∞	∞	∞	∞

P =

1
1
5
1

D =

50
30
20
10

v = 4
C = {2, 3, 4}
S = {1,5}

Dijkstra’s Algorithm: after iteration 2

L =

∞	∞	∞	∞	∞
50	∞	5	20	∞
30	∞	∞	50	∞
100	∞	∞	∞	10
10	∞	∞	∞	∞

P =

4
1
5
1

D =

40
30
20
10

v = 3
C = {2, 3}
S = {1,4,5}

Dijkstra’s Algorithm: after iteration 3

– L =

∞	∞	∞	∞	∞
50	∞	5	20	∞
30	∞	∞	50	∞
100	∞	∞	∞	10
10	∞	∞	∞	∞

P =

3
1
5
1

D =

35
30
20
10

v = 2
C = {2}
S = {1,3,4,5}

Dijkstra’s Algorithm: Recorded Paths

What does it mean?

P =

3
1
5
1

Node 1 is source.

The Predecessor of Node 2 is Node 3.

The Predecessor of Node 3 is Node 1 (source).

The Predecessor of Node 4 is Node 5

The Predecessor of Node 5 is Node 1 (source).

Dijkstra's Algorithm: Recorded Paths

To 2 – 1, 3, 2

To 3 – 1, 3

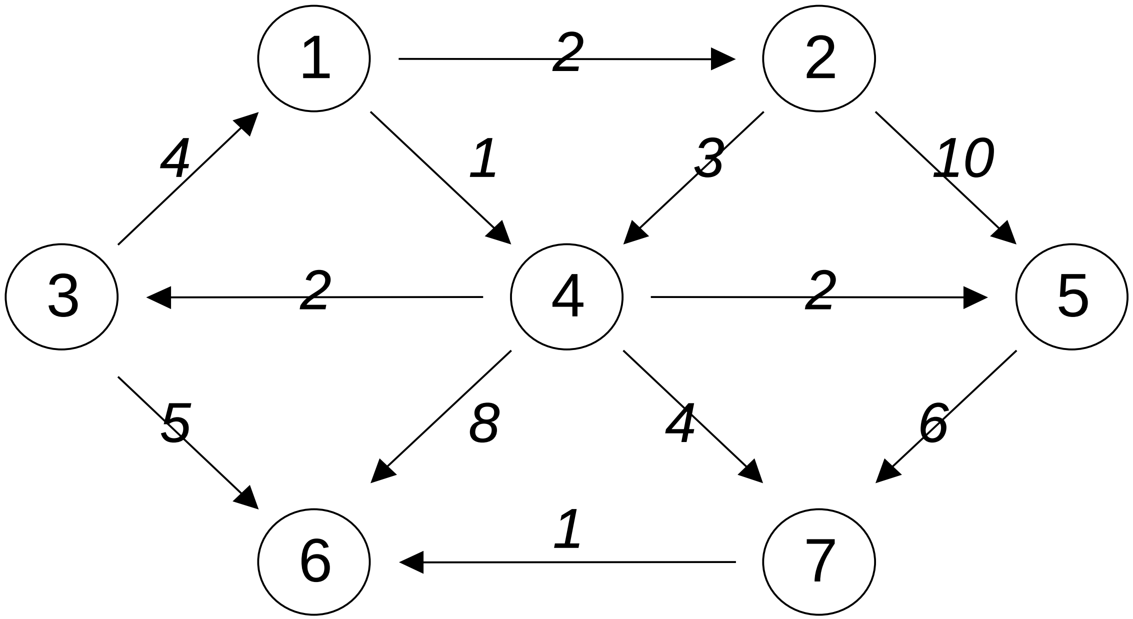
To 4 – 1, 5, 4

To 5 – 1, 5

P =

3
1
5
1

- Dijkstra’s Algorithm: Another Example



- Dijkstra’s Algorithm: At start

L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
1
1
1
1
1

D =

2
∞
1
∞
∞
∞

v = 4
C = {2, 3, 4, 5, 6, 7}
S = {1}

- Dijkstra's Algorithm: After step 1
  - L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

2
3
1
3
9
5

v = 2
C = {2, 3, 5, 6, 7}
S = {1, 4}

- Dijkstra's Algorithm: After step 2
  - L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

2
3
1
3
9
5

  -

v = 5
C = { 3, 5, 6, 7}
S = {1， 2， 4}

- Dijkstra’s Algorithm: After step 3
  - L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

2
3
1
3
9
5

v = 3
C = { 3, 6, 7}
S = {1， 2， 4， 5}

- Dijkstra's Algorithm: After step 4
  - L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
3
4

D =

2
3
1
3
8
5

v = 7
C = {6, 7}
S = {1, 2, 3, 4, 5}

- Dijkstra's Algorithm: After step 5 – done
  - L =

∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
7
4

D =

2
3
1
3
6
5

v = 7
C = {6}
S = {1, 2, 3, 4, 5, 7}

- Dijkstra’s Algorithm: After step 5 – done
  - Paths

    To 2: 1, 2

    To 3: 1, 4, 3

    To 4: 1, 4

    To 5: 1, 4, 5

    To 6: 1, 4, 7, 6

    To 7: 1, 4, 7

P =

1
4
1
4
7
4

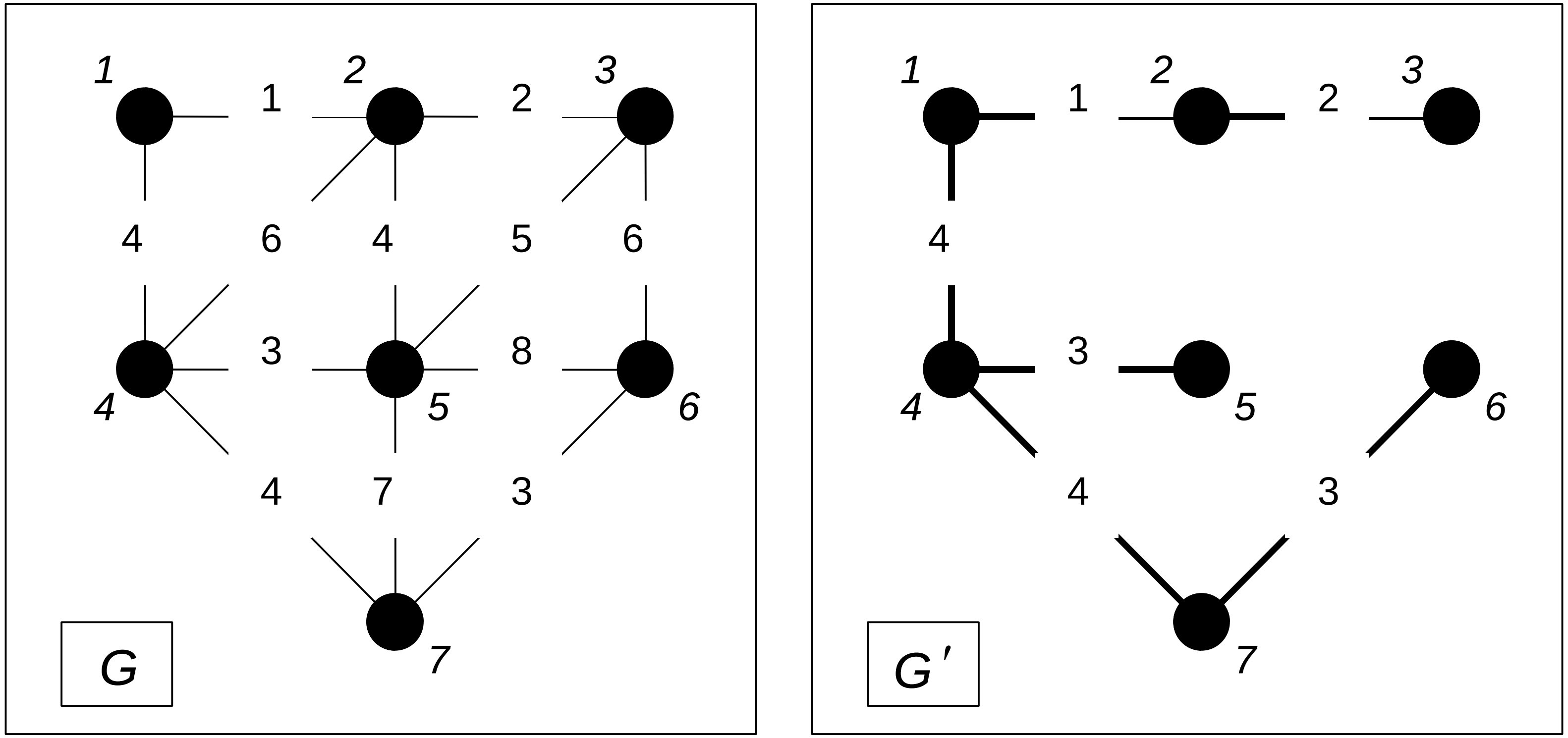
D =

2
3
1
3
6
5

Example 3: Minimum Spanning Tree

- Greedy Algorithms
  - Example 3: Minimum Spanning Tree
    - Let G = (N, E) be a connected, undirected graph consisting of a set of nodes, N, and a set of edges E.
    - Each edge has a length, the distance from the node at one end of the edge to the node at the other end.
    - The problem is to find a subset, S, of the edges of G such that the graph G'= (N, S) is still connected and that the total length of the edges in S is minimized.
    - G' is called the minimum spanning tree for the graph G

Example 3: Minimum Spanning Tree



Example 3: Minimum Spanning Tree

– Two possible paths of attack seem possible:

- Start with an empty set S and select at each stage the shortest edge that has been neither selected nor rejected.

- Start at a given node and at each stage select into S shortest edge that extends the graph to a new node

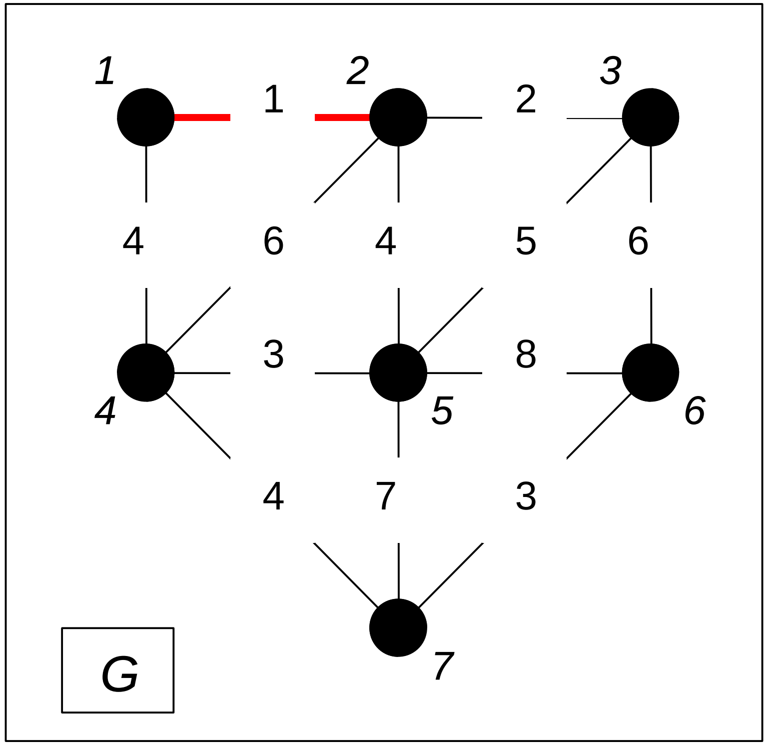
– Strangely, both approaches work

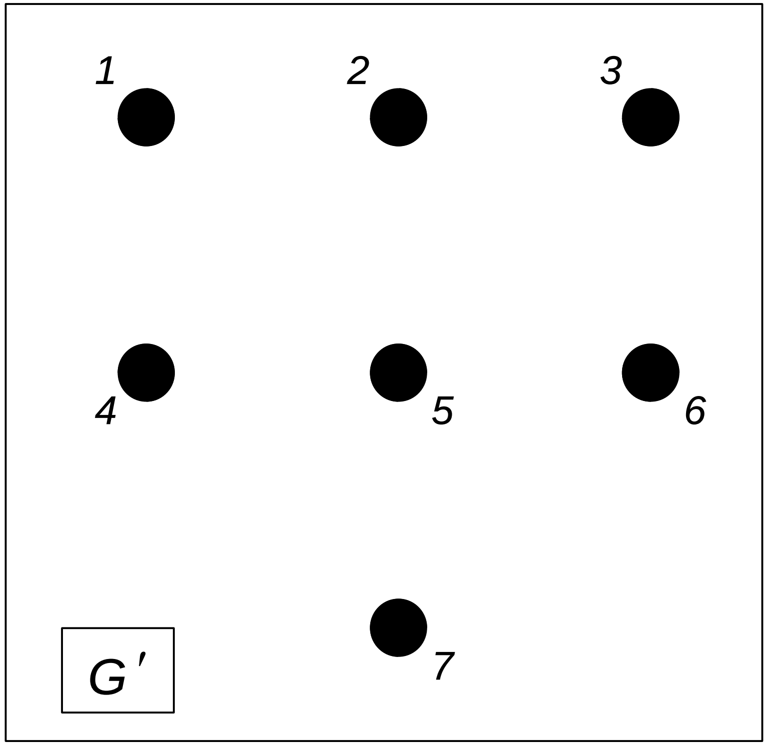
Kruskal’s Algorithm

– Start with an initially empty set of edges S.
– Add edges to S
– At each step add the shortest edge to S which increases the connectedness of the graph.
– Reject a candidate edge if it does not effect the connectedness of S.
– Stop when the graph is connected.

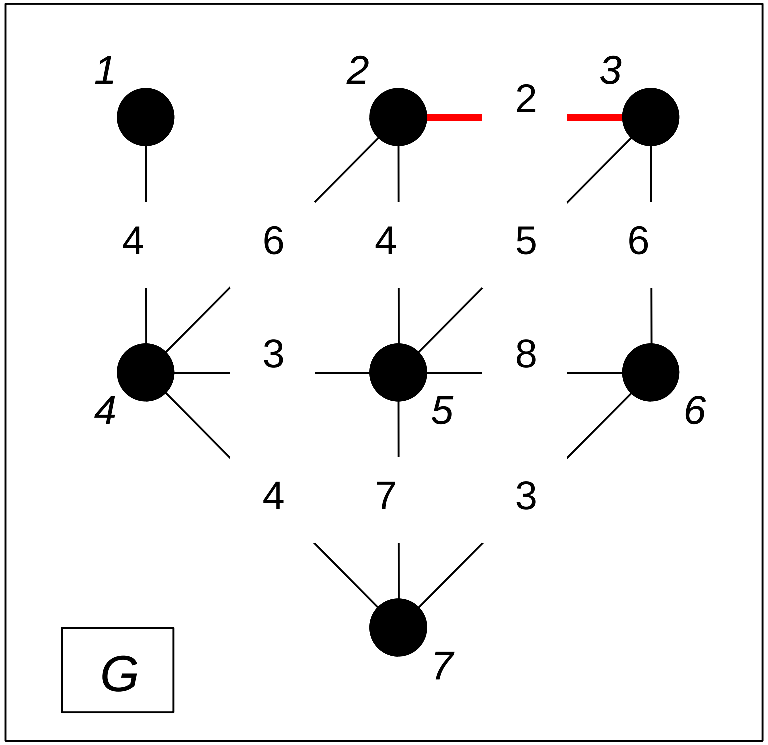
Kruskal’s Algorithm: An Example

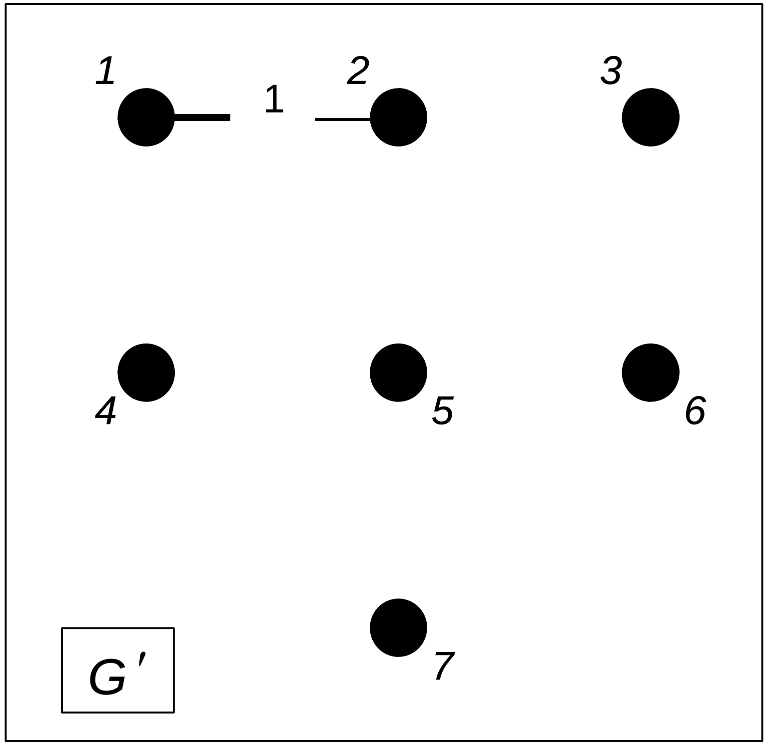
- Kruskal’s Algorithm: An Example
  - Step 0 - {1} {2} {3} {4} {5} {6} {7}



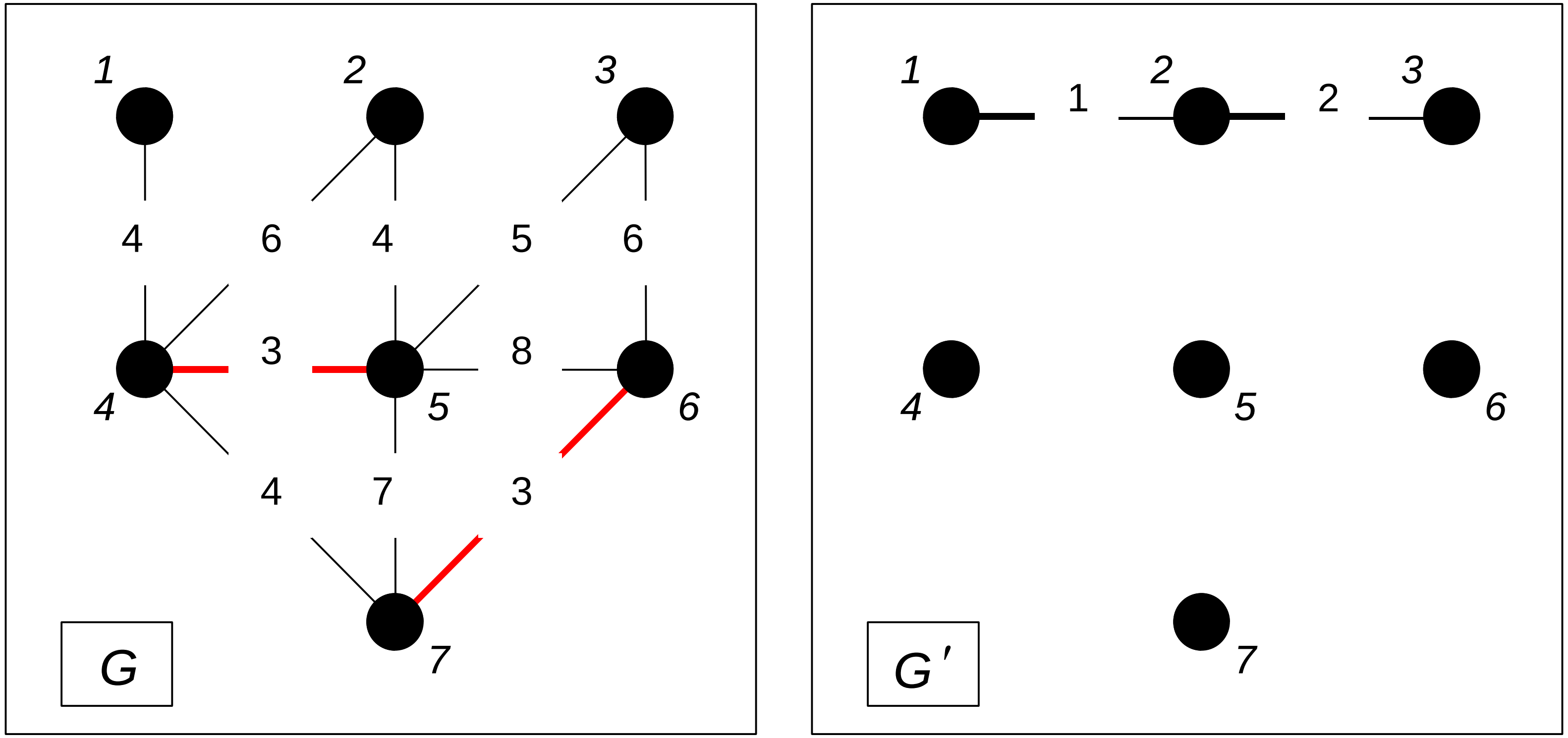


- Kruskal's Algorithm: An Example
  - Step 1 {1,2} {3} {4} {5} {6} {7}

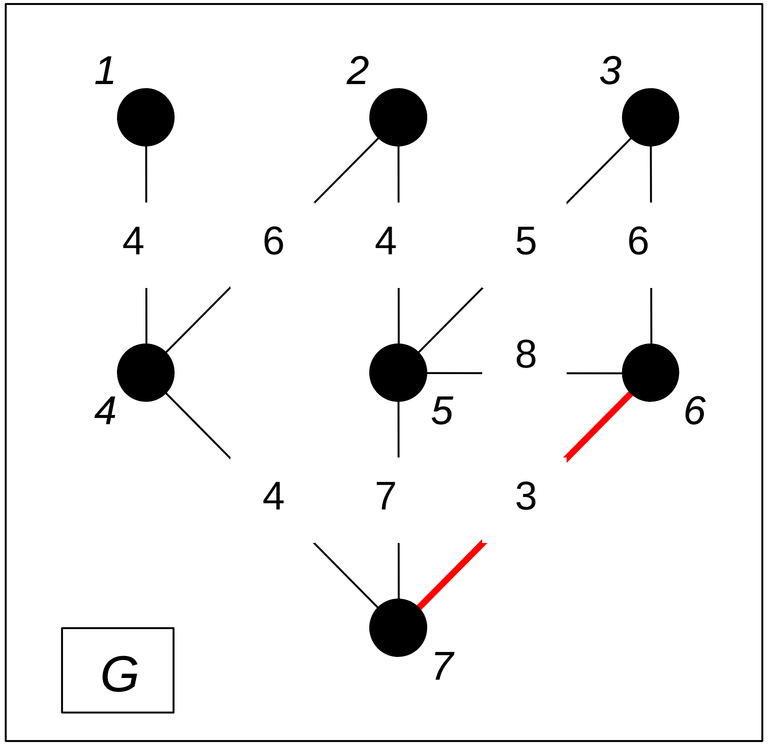


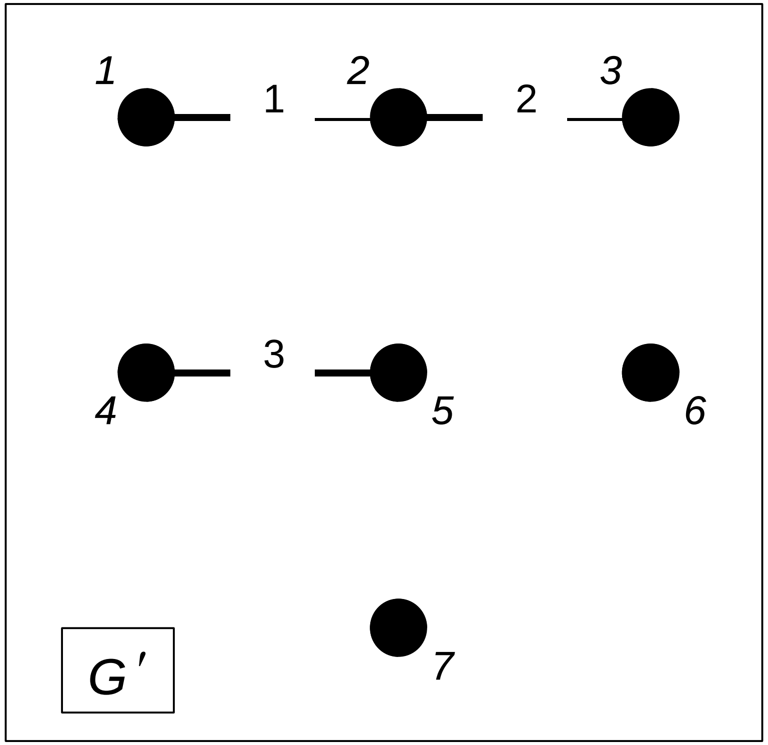


- Kruskal's Algorithm: An Example
  - Step 2 {2,3} {1,2,3} {4} {5} {6} {7}

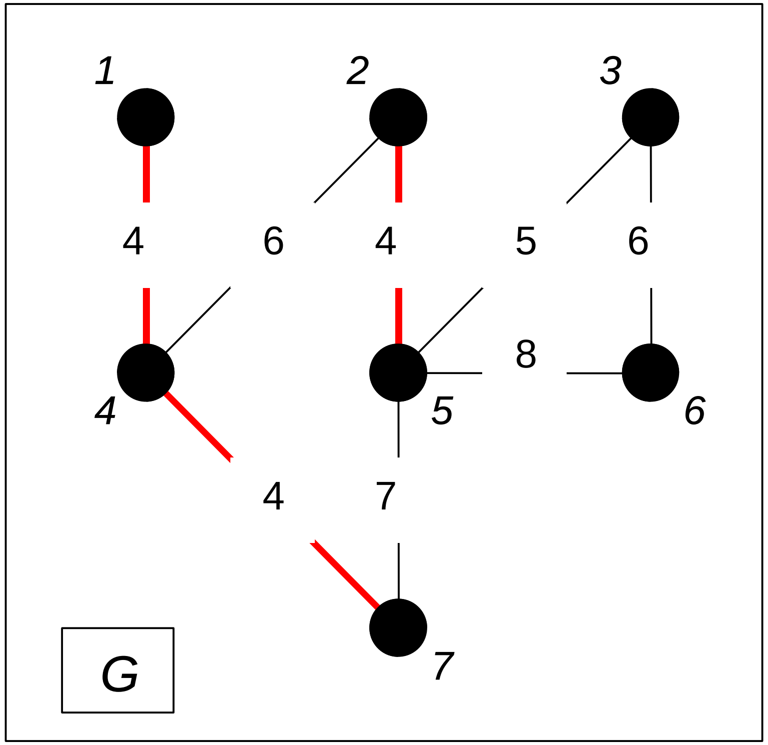


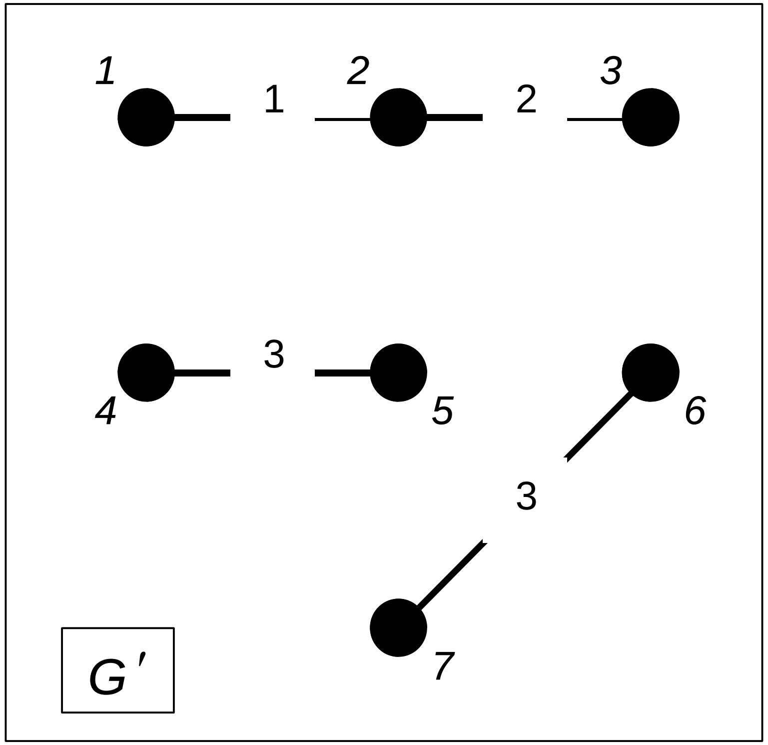
- Kruskal’s Algorithm: An Example
  - Step 3 {4,5} {1,2,3} {4,5} {6} {7}



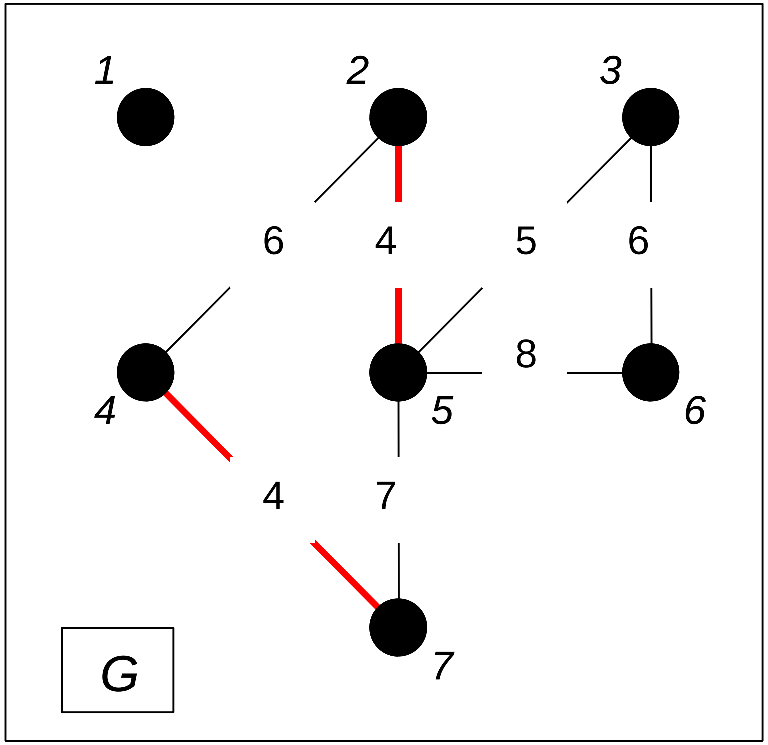


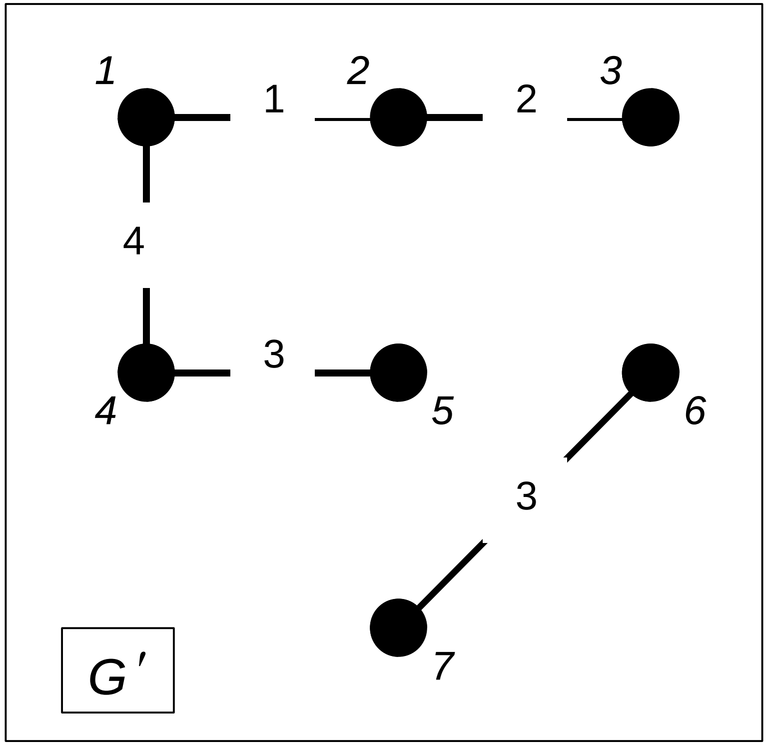
- Kruskal's Algorithm: An Example
    - Step 4 {6,7} {1,2,3} {4,5} {6,7}



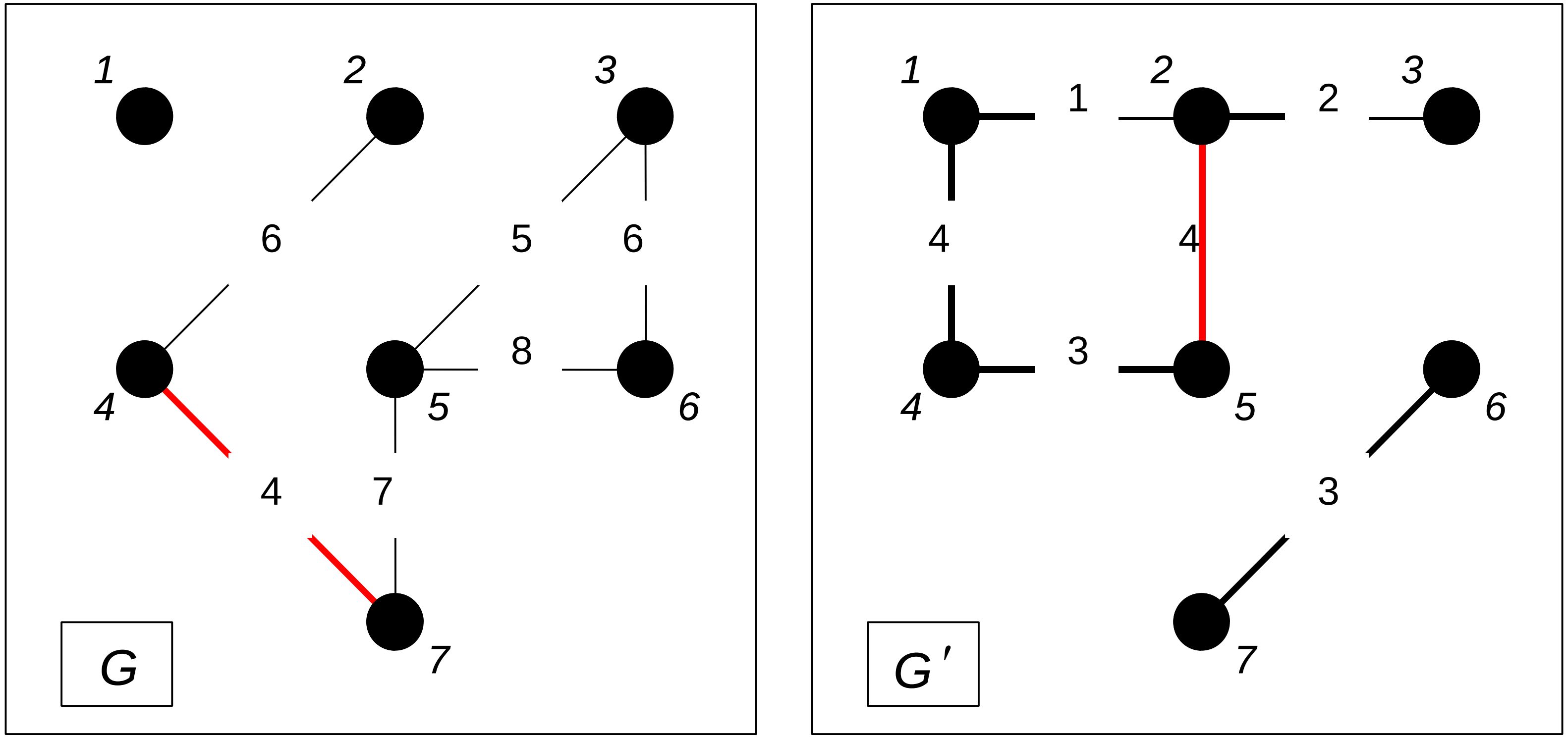


- Kruskal’s Algorithm: An Example
  - Step 5 {1,4} {1,2,3,4,5} {6,7}

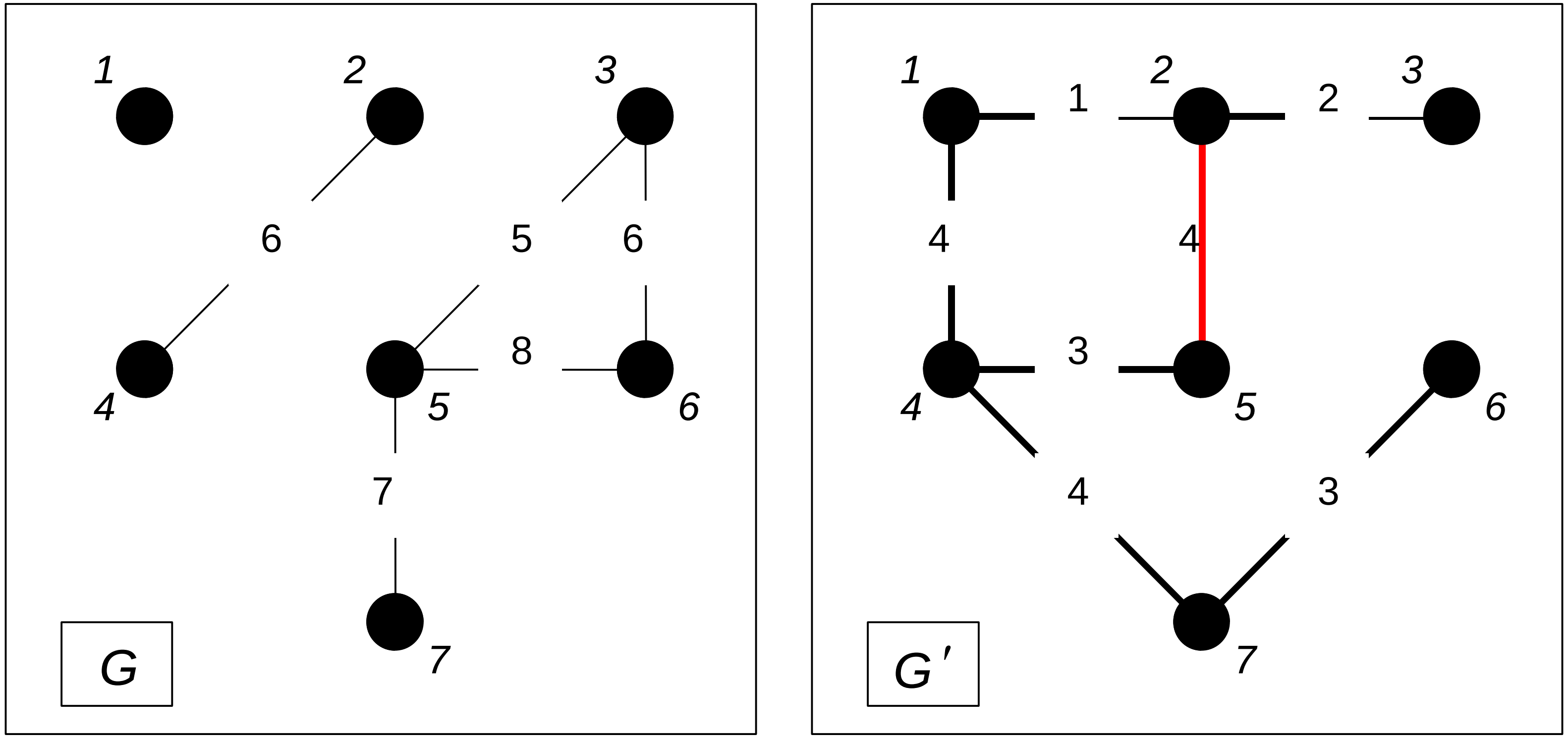




- Kruskal's Algorithm: An Example
  - Step 5 {2,5} {1,2,3,4,5} {6,7} - rejected



- Kruskal's Algorithm: An Example
  - Step 5 {4,7} {1,2,3,4,5,6,7} - done



- Kruskal’s Algorithm
  - type node = record
    node_number: integer
  - type edge = record
    start: ^node
    end: ^node
    length: integer

Kruskal’s Algorithm

```
Function Kruskal(N[1..n]: ^node,
E[1..e]: ^edge)
    sort E by increasing length
    S = {}
    for i =1 to n do
        set[i] = {N[i]}
    i = 0
    repeat
        i = i+1
        u = E[i]^.start
        v = E[i]^.end
        uset = find u in set[]
        vset = find v in set[]
        if uset != vset then
            merge(uset, vset)
            add E[i] to S
    until S contains n – 1 edges
    return S
```

Kruskal’s Algorithm

```
Function Kruskal(N[1..n]: ^node,
E[1..e]: ^edge)
    sort E by increasing length
    S = {}
    for i =1 to n do
        set[i] = {N[i]}
    i = 0
    repeat
        i = i+1
        u = E[i]^.start
        v = E[i]^.end
        uset = find u in set[]
        vset = find v in set[]
        if uset != vset then
            merge(uset, vset)
            add E[i] to S
    until S contains n – 1 edges
    return S
```

Kruskal's Algorithm

KRUSKAL(G):

1 A = ∅

2 **foreach** v ∈ G.V:

3 MAKE-SET(v)

4 **foreach** (u, v) in G.E ordered by weight(u, v), increasing:

5 **if** FIND-SET(u) ≠ FIND-SET(v):

6 A = A ∪ {(u, v)}

7 UNION(u, v)

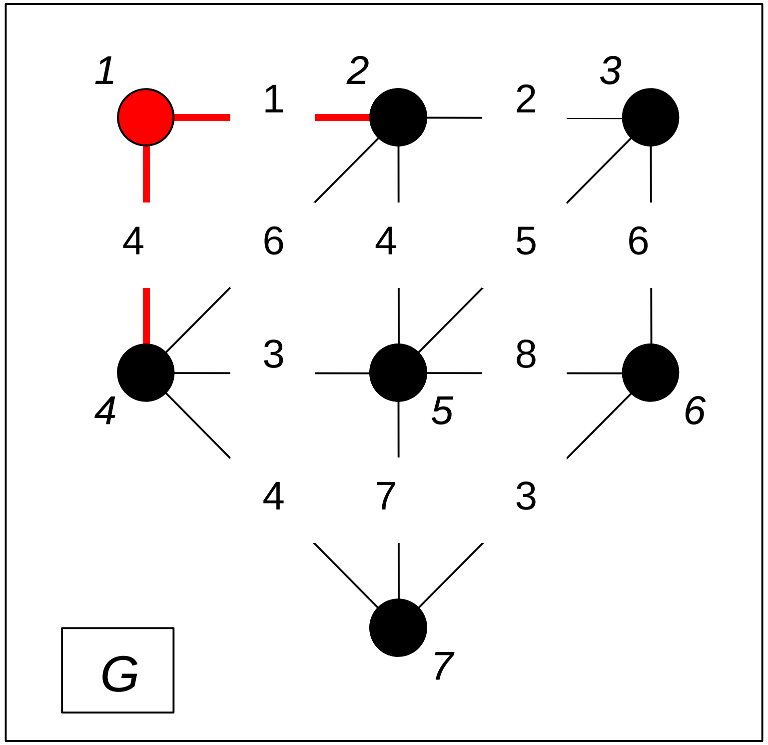
8 **return** A

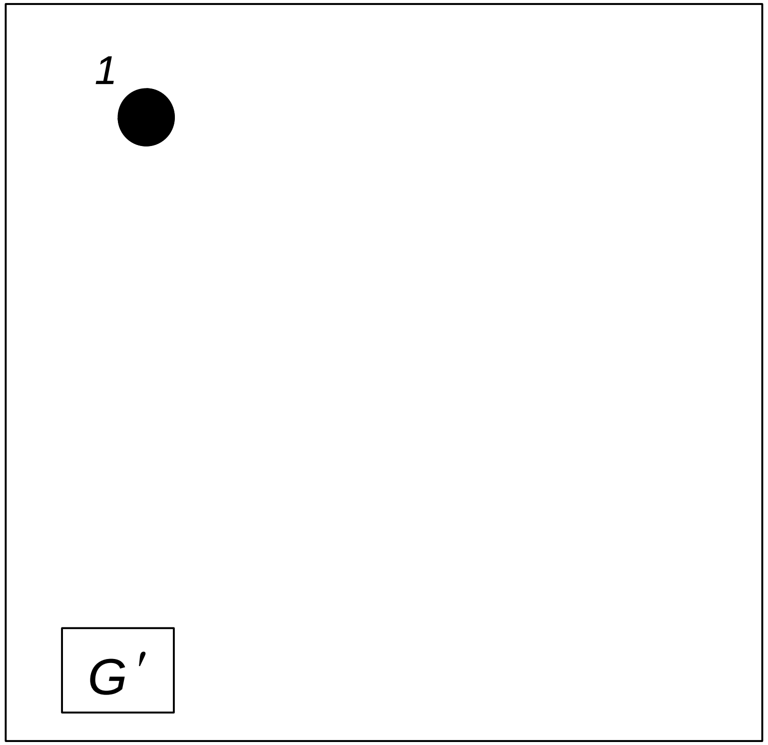
Prim's Algorithm

– Let O be a set of nodes and S a set of edges
  – Initially O contains the first node of N and S is empty
  – At each step look for the shortest edge {u, v} in E such that $u \in O$ and $v \notin O$
  – Add {u, v} to S
  – Add v to O
  – Repeat until O = N
  – Note that, at each step, S is a minimum spanning tree on the nodes in O

Prim’s Algorithm: An Example

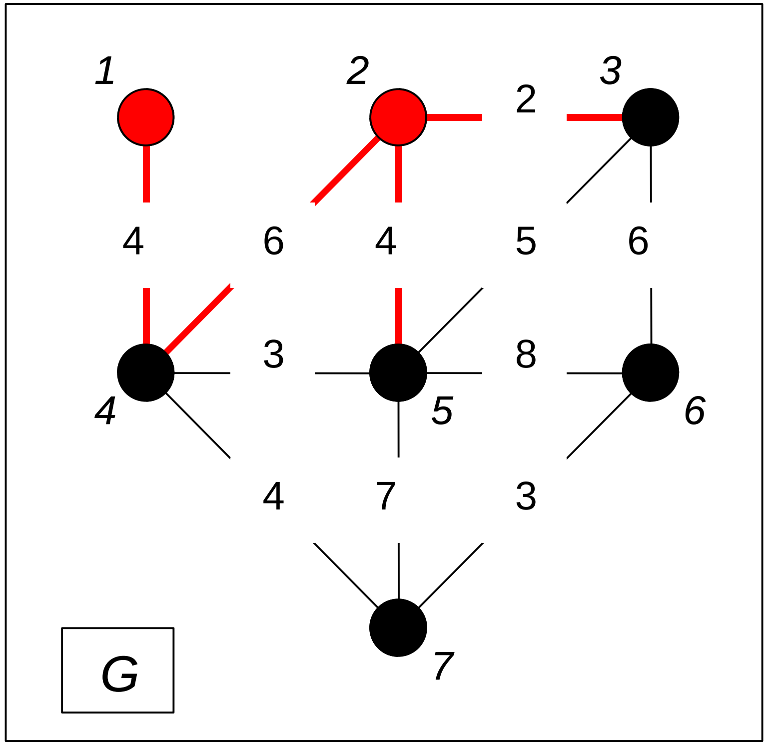
- Step 0 - {1}

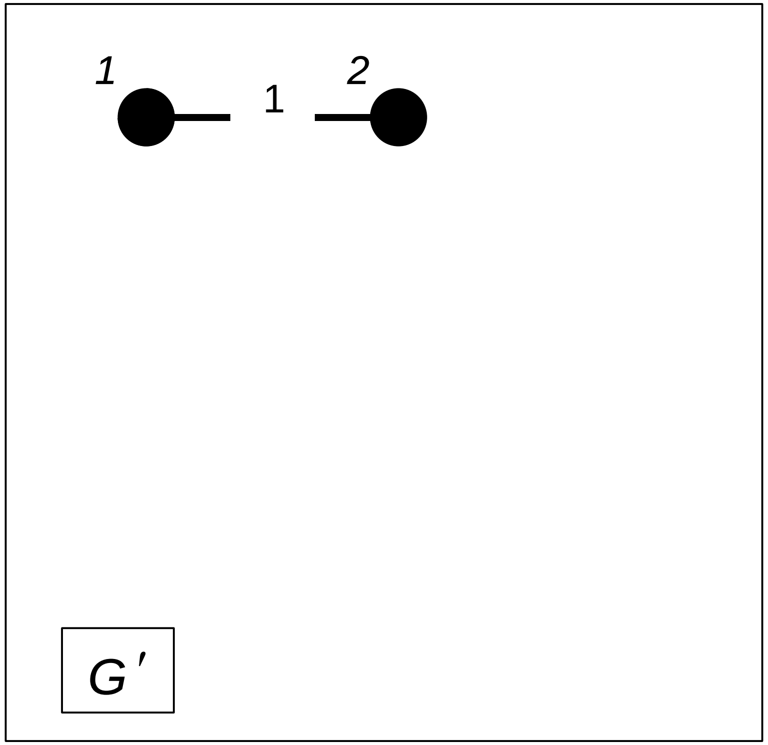




Prim's Algorithm: An Example

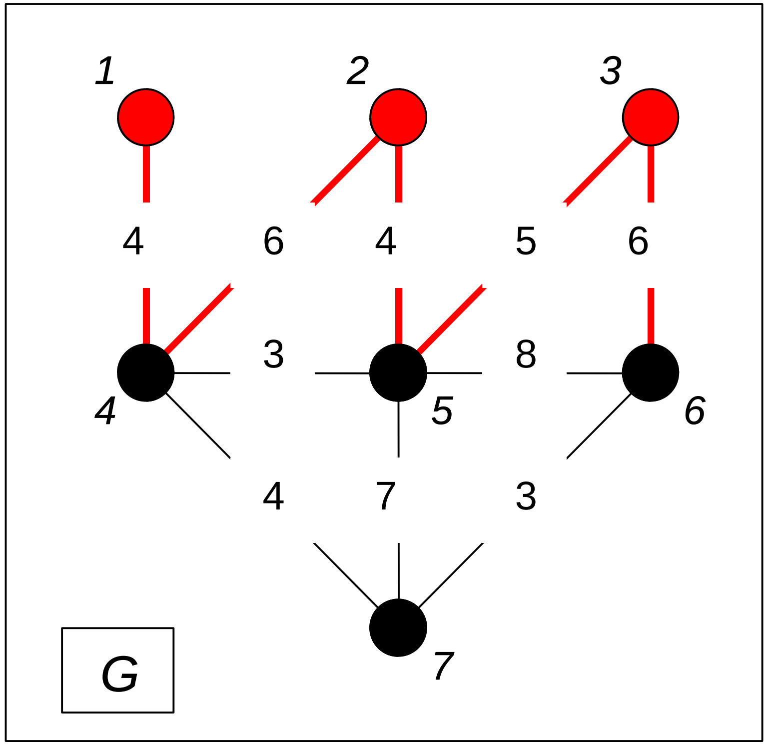
- Step 1 {1, 2} {1, 2}

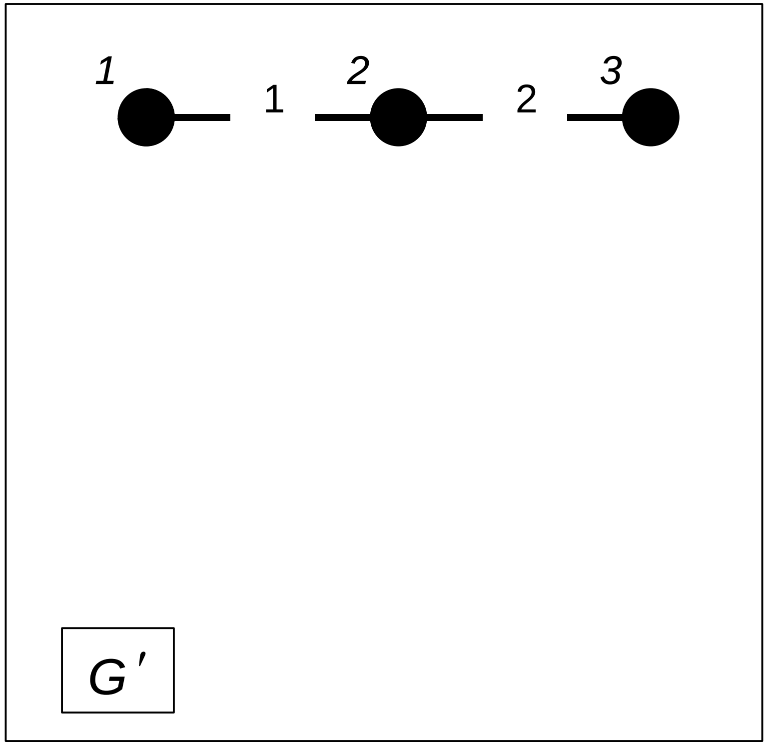




Prim's Algorithm: An Example

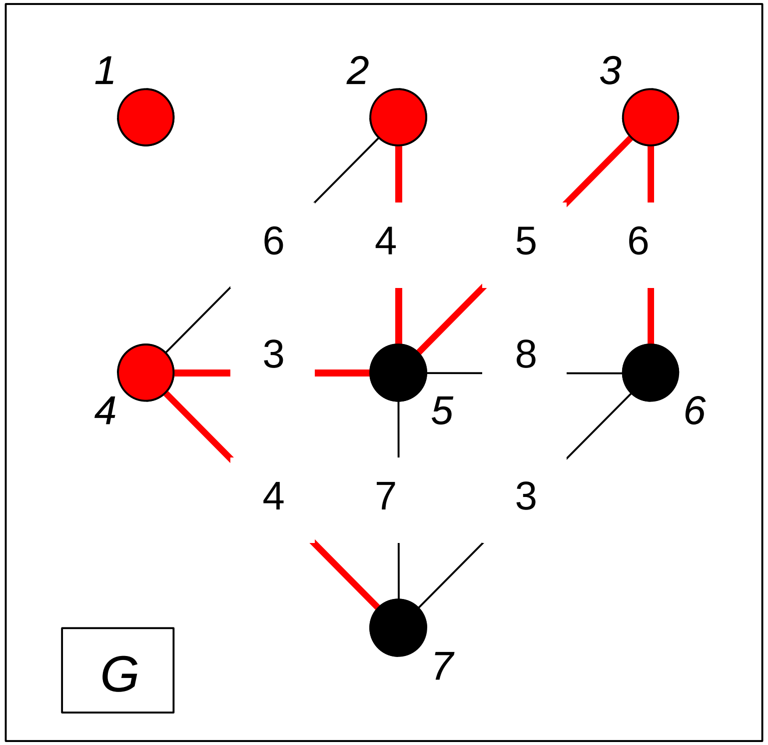
- Step 2 {2, 3} {1, 2, 3}

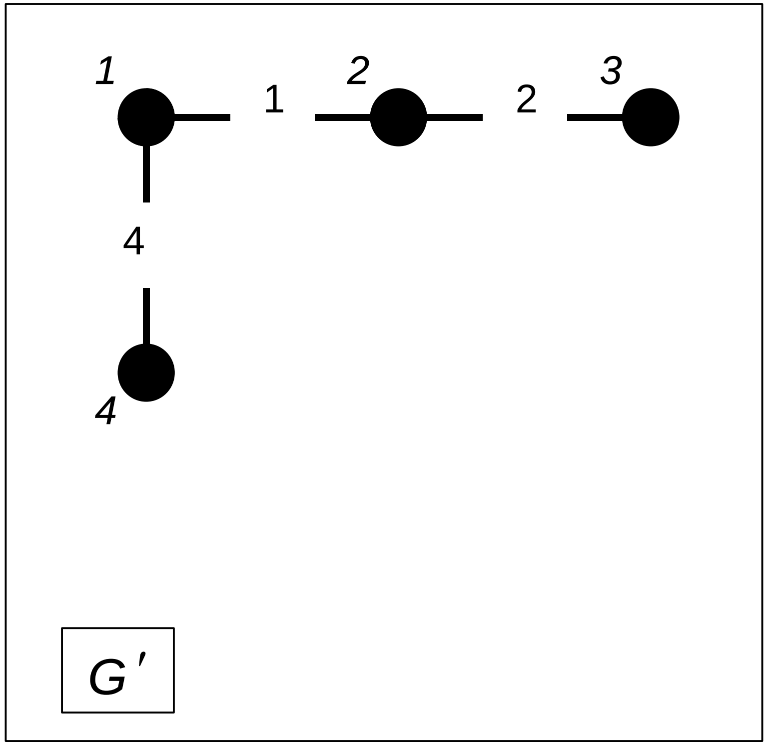




Prim’s Algorithm: An Example

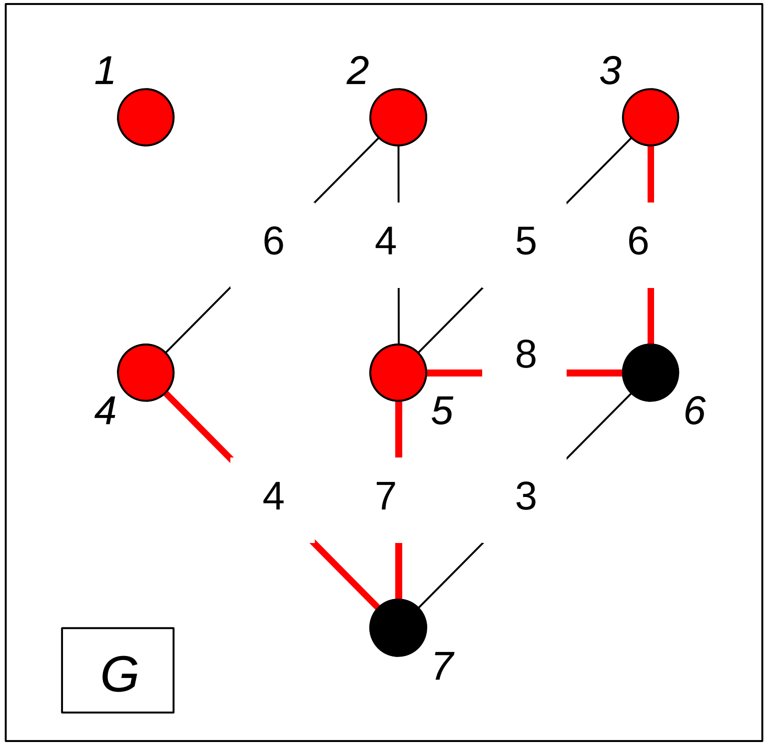
- Step 3 {1, 4} {1, 2, 3, 4}

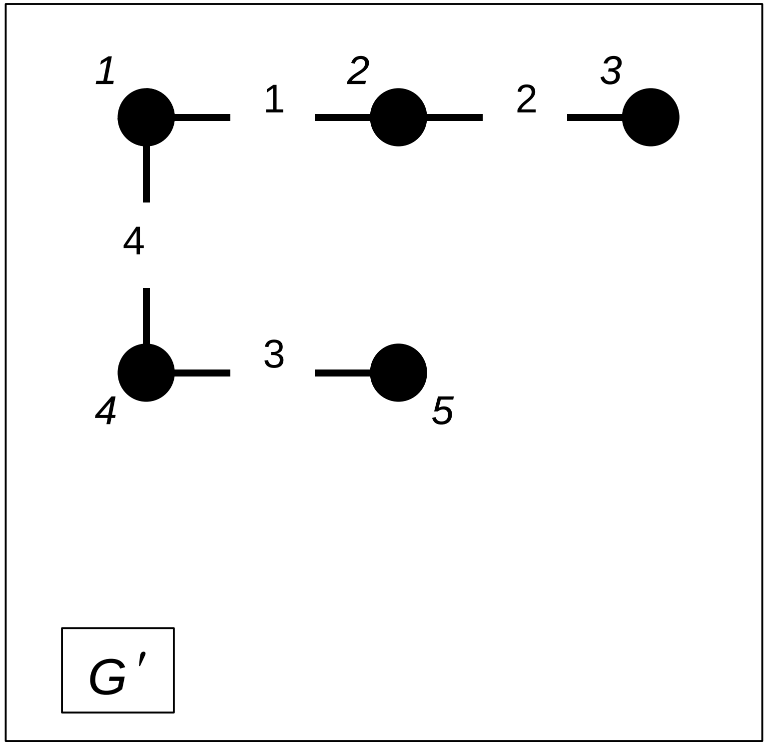




Prim’s Algorithm: An Example

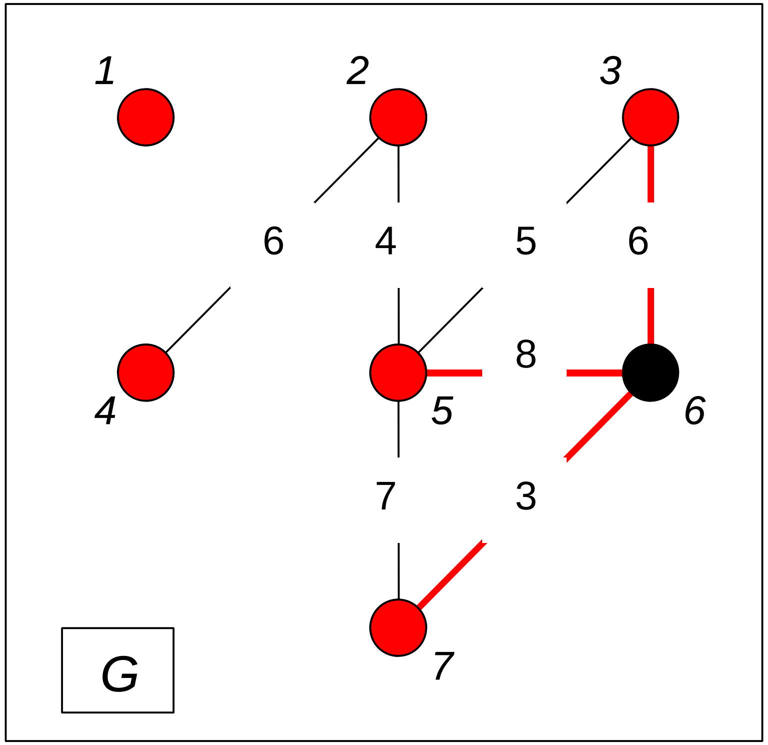
- Step 4   {4, 5}   {1, 2, 3, 4, 5}

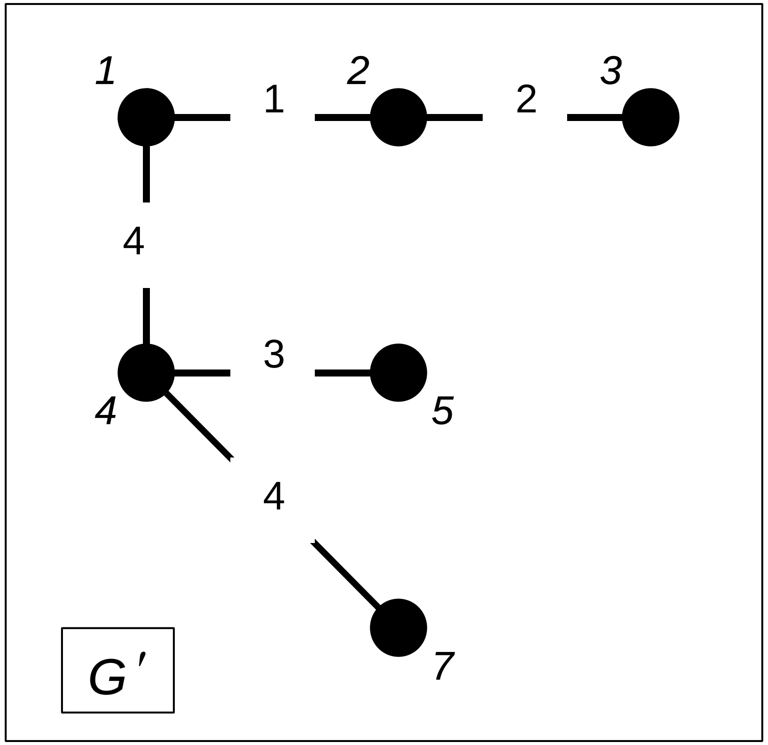




Prim's Algorithm: An Example

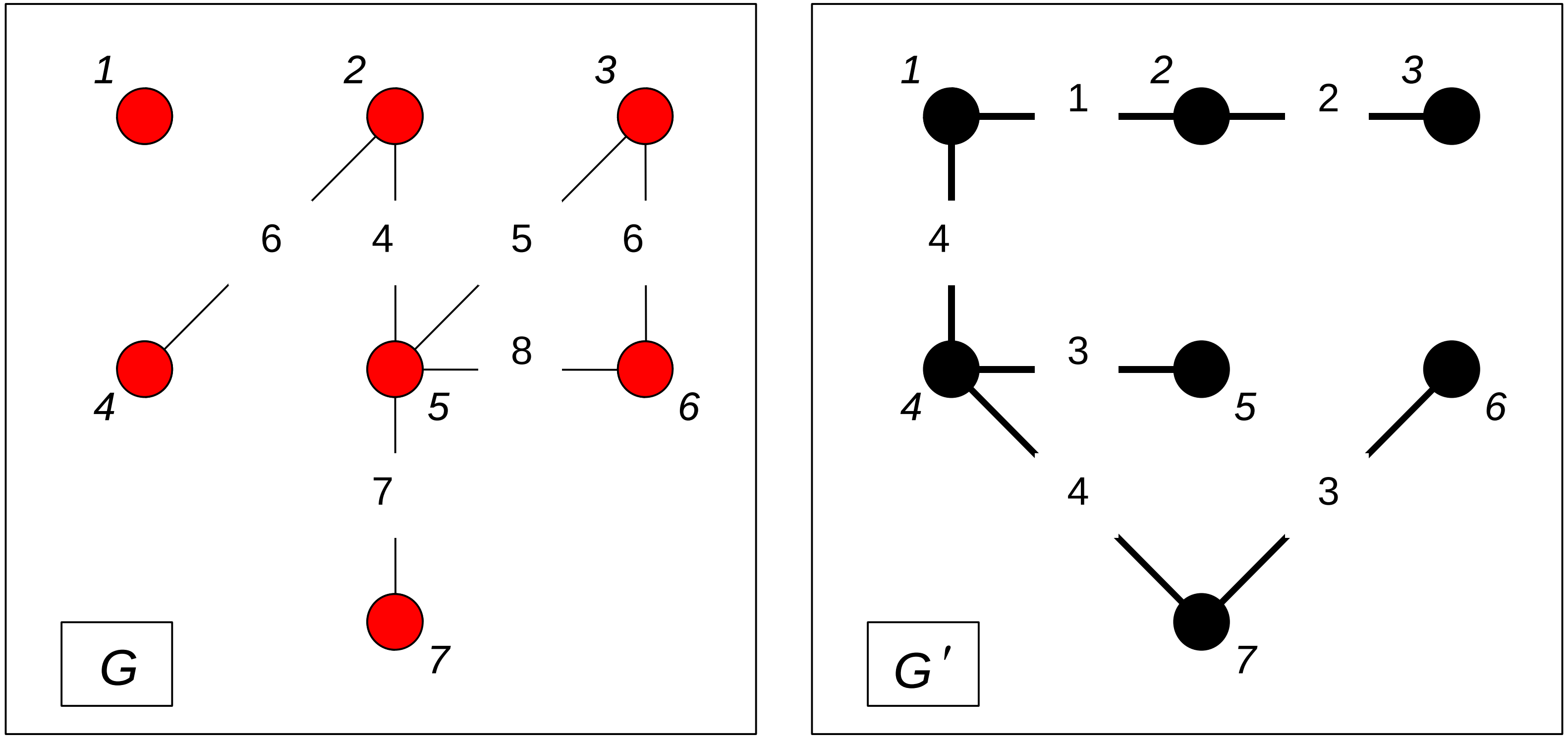
- Step 5 {4, 7} {1, 2, 3, 4, 5, 7}





Prim’s Algorithm: An Example

- Step 5 {7, 6} {1, 2, 3, 4, 5, 6, 7} – done



Prim’s Algorithm

```
Function Prim(L[1..n, 1..n])
      S = {}
      for i = 2 to n do
          nearest[i] = 1
          mindist[i] = L[i, 1]
      repeat n – 1 times
          min = ∞
          for j = 2 to n do
              if 0 ≤ mindist[j] ≤ min then
                   min = mindist[j]
                   k = j
          add {nearest[k], k} to S
          mindist[k] = –1
          for j = 2 to n do
              if L[j, k] < mindist[j] then
                  mindist[j] = L[j, k]
                  nearest[j] = k
      return S
```

Prim's Algorithm: at start

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

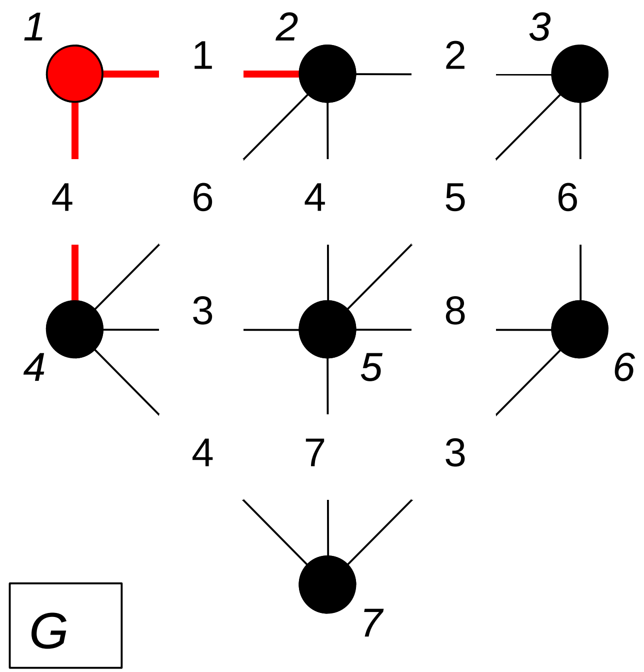
nearest =

1
1
1
1
1
1
1

mindist =

∞
1
∞
4
∞
∞
∞

S = { }



Prim's Algorithm: after iteration 1

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
2
1
1

mindist =

∞
–1
2
4
4
∞
∞

S = {{1, 2}}

Prim's Algorithm: after iteration 2

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
2
3
1

mindist =

∞
–1
–1
4
4
6
∞

S = {{1, 2}, {2, 3}}

Prim's Algorithm: after iteration 3

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
4
3
4

mindist =

∞
–1
–1
–1
3
6
4

S = {{1, 2}, {2, 3}, {1, 4}}

Prim's Algorithm: after iteration 4

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
4
3
4

mindist =

∞
–1
–1
–1
–1
6
4

S = {{1, 2}, {2, 3}, {1, 4}, {4, 5}}

Prim's Algorithm: after iteration 5

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
4
7
4

mindist =

∞
–1
–1
–1
–1
3
–1

S = {{1, 2}, {2, 3}, {1, 4}, {4, 5}, {4, 7}}

Prim’s Algorithm: after iteration 6

L =

∞	1	∞	4	∞	∞	∞
1	∞	2	6	4	∞	∞
∞	2	∞	∞	5	6	∞
4	6	∞	∞	3	∞	4
∞	4	5	3	∞	8	7
∞	∞	6	∞	8	∞	3
∞	∞	∞	4	7	3	∞

nearest =

1
1
2
1
4
7
4

mindist =

∞
–1
–1
–1
–1
–1
–1

S = {{1, 2}, {2, 3}, {1, 4}, {4, 5}, {4, 7}, {7, 6}}

The Simplified Knapsack Problem

- We have a set of n objects and a knapsack.
- Each object has a weight w *i*
- Each object has a value v i
- The knapsack can hold a total weight W
- We must pack the knapsack with the most valuable load.
- We may break an object into smaller pieces if we wish. I.e. we can pack a fraction x i of object i where 0 < x i < 1
- **Note:** If we are not allowed to break objects this becomes a much harder problem.

- The Simplified Knapsack Problem
  - An example:
  - $n = 5$, $W = 100$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Strategy 1: pick the most valuable object

- The Simplified Knapsack Problem
  - Pack as much of the most valuable object as you can
  - $n = 5$, $W = 100$, $V = 66$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i			1.0		

- The Simplified Knapsack Problem
  - Pack as much of the next most valuable object
  - $n = 5$, $W = 100$, $V = 126$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i			1.0		1.0

- The Simplified Knapsack Problem
  - And the next most valuable object
  - $n = 5$, $W = 100$, $V = 146$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i			1.0	0.5	1.0

- The Simplified Knapsack Problem
  - An example:
  - $n = 5$, $W = 100$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Strategy 2: pick the lightest object

- The Simplified Knapsack Problem
  - Pack as much of the lightest object as you can
  - $n = 5$, $W = 100$, $V = 20$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i	1.0				

- The Simplified Knapsack Problem
  - Pack as much of the next lightest object as you can
  - $n = 5$, $W = 100$, $V = 50$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i	1.0	1.0			

- The Simplified Knapsack Problem
  - And the next lightest object
  - $n = 5$, $W = 100$, $V = 116$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i	1.0	1.0	1.0		

- The Simplified Knapsack Problem
  - And, finally, the next lightest object
  - $n = 5$, $W = 100$, $V = 156$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
x_i	1.0	1.0	1.0	1.0	

- The Simplified Knapsack Problem
  - An example:
  - $n = 5$, $W = 100$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60

Strategy 3: pick the object with the highest value per unit weight

- The Simplified Knapsack Problem
  - Calculate the value per unit weight v_i / w_i
  - $n = 5$, $W = 100$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i / w_i	2.0	1.5	2.2	1.0	1.2

- The Simplified Knapsack Problem
  - Pack as much of the best object as you can
  - n = 5, W = 100, V = 66

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i / w_i	2.0	1.5	2.2	1.0	1.2
x_i			1.0		

- The Simplified Knapsack Problem
  - Repeat with the next best object
  - $n = 5$, $W = 100$, $V = 86$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i / w_i	2.0	1.5	2.2	1.0	1.2
x_i	1.0		1.0		

- The Simplified Knapsack Problem
  - And the next best
  - $n = 5$, $W = 100$, $V = 116$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i / w_i	2.0	1.5	2.2	1.0	1.2
x_i	1.0	1.0	1.0		

- The Simplified Knapsack Problem
  - And, finally, the next best
  - $n = 5$, $W = 100$, $V = 164$

Object	1	2	3	4	5
w_i	10	20	30	40	50
v_i	20	30	66	40	60
v_i / w_i	2.0	1.5	2.2	1.0	1.2
x_i	1.0	1.0	1.0		0.8

- The Simplified Knapsack Problem
  - In summary:

Strategy	x_i					Value
Max v_i	0.0	0.0	1.0	0.5	1.0	146
Min w_i	1.0	1.0	1.0	1.0	0.0	156
Max v_i / w_i	1.0	1.0	1.0	0.0	0.8	164

  - Clearly, the last strategy gives the best results

- Greedy Algorithms
  - Example 5: Scheduling – minimum time
    - A single server has n customers to serve
    - The service time for each customer is known in advance = T_i for customer *i*.
    - We want to minimize the average time each customer spends in the queue = T_{av}
    - This is equivalent to spending the least total time – since $T_{av} = (T_1 + T_2 + \cdots + T_n)/n$

- Scheduling – minimum time
  - An example:

    n = 3, t_1= 5, t_2 = 10 , t_3 = 3
  - Try all possible orderings of t_1 and t_2 and t_3

Order	T	
1, 2, 3	5 + (5 + 10) + (5 + 10 + 3)	38
1, 3, 2	5 + (5 + 3) + (5 + 3 + 10)	31
2, 1, 3	10 + (10 + 5) + (10 + 5 +3)	43
2, 3, 1	10 + (10 + 3) + (10 + 3 +5)	41
3, 1, 2	3 + (3 + 5) + (3 + 5 + 10)	29
3, 2, 1	3 + (3 + 10) + (3 + 10 + 5)	34

- Scheduling – minimum time
  - We note that the optimal solution, 29, is obtained by choosing the customers in order of increasing service time.
  - One example does not constitute a proof that the best result is obtained by serving in increasing order.
  - Let us see if we can prove that this is the best strategy.

- Scheduling – minimum time
  - **Theorem:** serving customers in increasing order of service time

minimizes the total time.

  - **Proof:** Let *P* =P_1, P_2, …, P_n be a permutation of customers 1 to n and let s_i = t_{pi} be the service time for the i^{th} customer if customers are served in order *P*.

    The total time for order *P* is

$$
\begin{aligned}
T(P) = \; & s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots \\
& = ns_1 + (n-1)s_2 + (n-2)s_3 + \cdots \\
& = \textstyle\sum_{k=1}^{n} (n-k)s_k
\end{aligned}
$$

- Scheduling – minimum time
  - If we can find customers *a*, *b* < n such that $P_a < P_\mathrm{b}$ and $s_a > s_\mathrm{b}$ we can produce a new permutation P^* by swaping P_a and P_b in the permutation

The total service time for P^* is

$$T(P^*) = (n - P_a + 1)s_b + (n - P_b + 1)s_a + \sum_{k=1}^{n}(n - k + 1)\, s_k$$

$$k \neq P_a, P_b$$

- Scheduling – minimum time
  - The new schedule P^* is better than P because

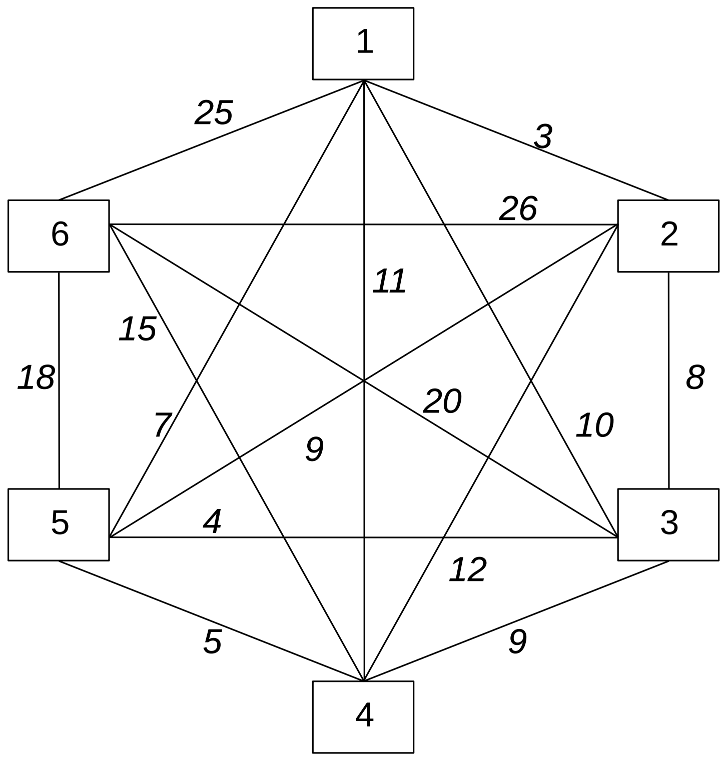
$$T(P) - T(P^*) = (n - P_a + 1)(S_a - S_b) + (n - P_b + 1)(S_b - S_a)$$
$$= (P_a - P_b)(S_a - S_b) > 0$$

because $P_a - P_b$ and $S_a - S_b$

  - Thus, total service time can be improved as long as any customers match the above criteria.
  - No further improvement is possible when customers are served in order of increasing service time.
  - Thus, service time is minimized when customers are served in order of increasing service time.

- Greedy Algorithms
  - Example 7: The Traveling Salesman Problem
    - Let G = (N, E) be a complete, undirected graph consisting of a set of nodes, N, and a set of edges E.
    - Each edge has a length, the distance from the node at one end of the edge to the node at the other end.
    - The problem is to find a subset, S, of the edges of G such that the graph G = (N, S) is still connected , S forms a cycle and that the total length of the edges in S is minimized.
    - If we view the nodes as towns and the edges as roads this is equivalent to finding the shortest round-trip route visiting each town once and returning to the start.
    - Can we find a greedy algorithm to solve this problem?

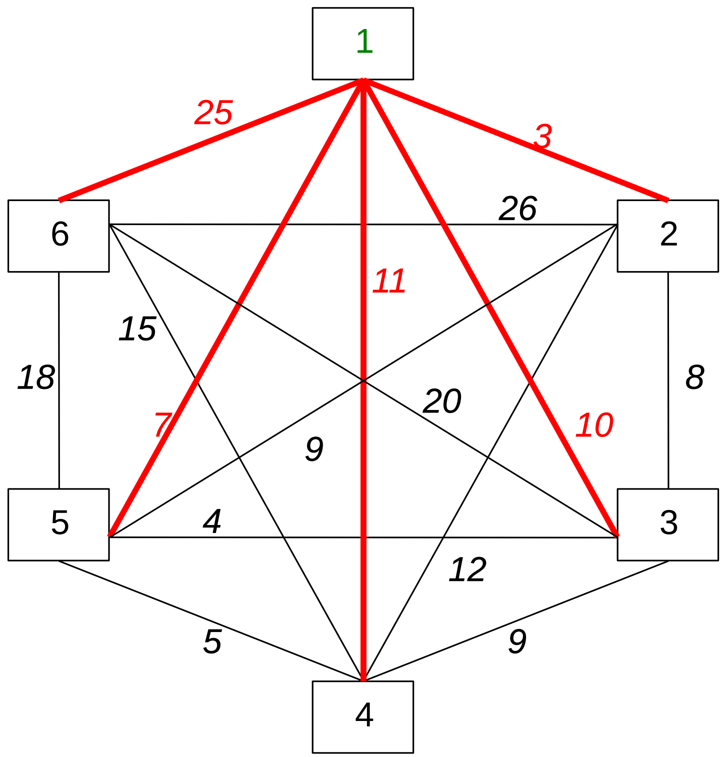
- Example 7: The Traveling Salesman Problem
  - Consider the following map – with distance matrix



0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

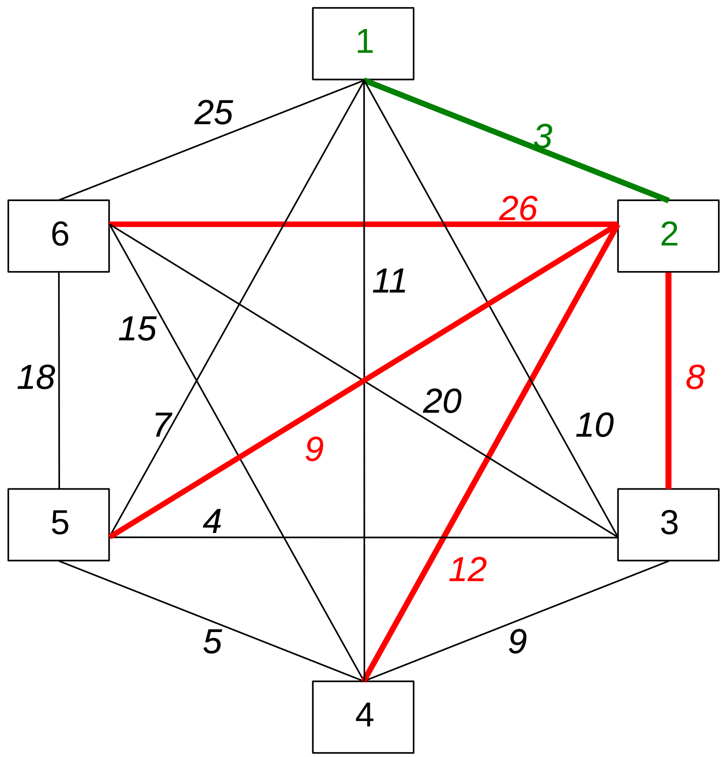
- The Traveling Salesman Problem
  - A greedy algorithm might be:
    - Start at an arbitrary node (node 1)
    - At each step visit the nearest node to the current one
    - When no more nodes are left, go home
  - How good is this algorithm?

- The Traveling Salesman Problem
  - Move to node 1



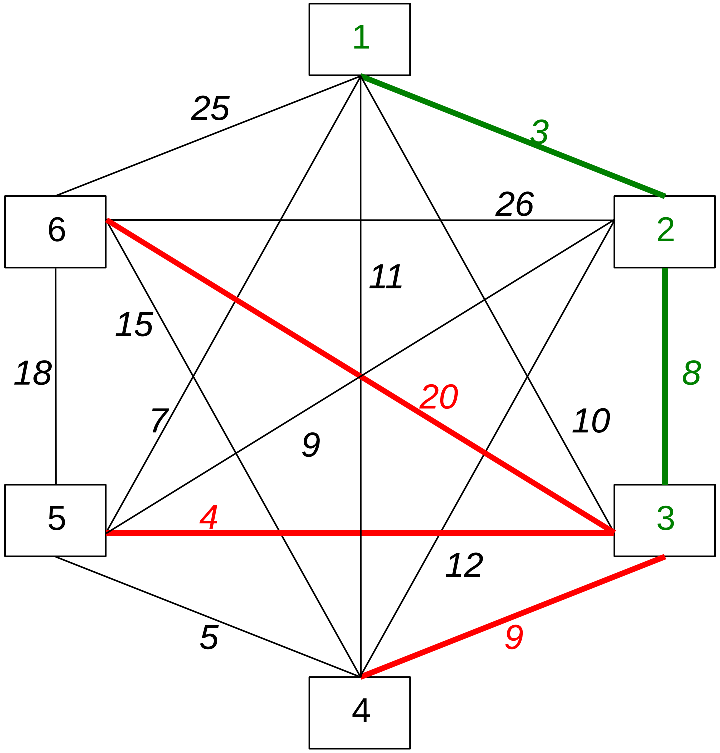
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  – Move to node 2



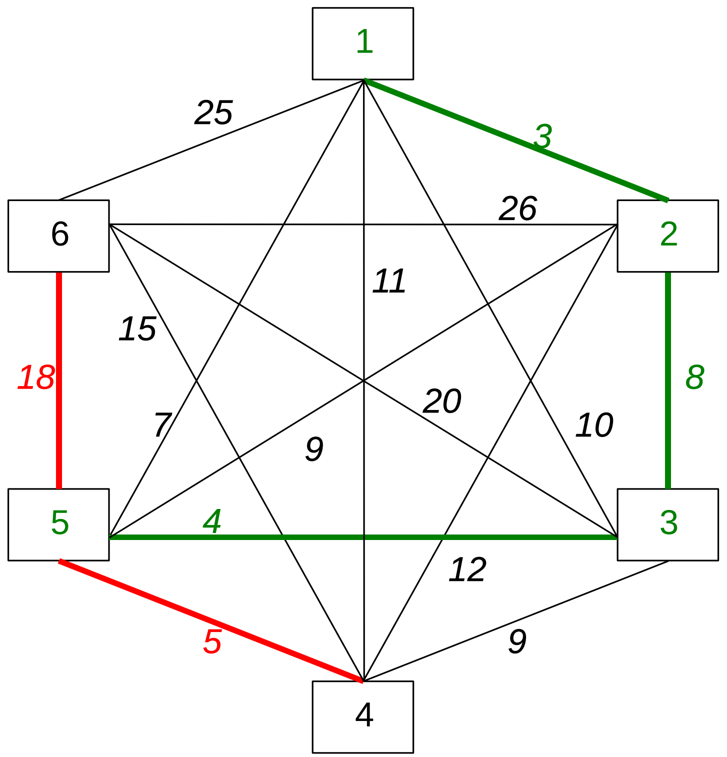
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Move to node 3



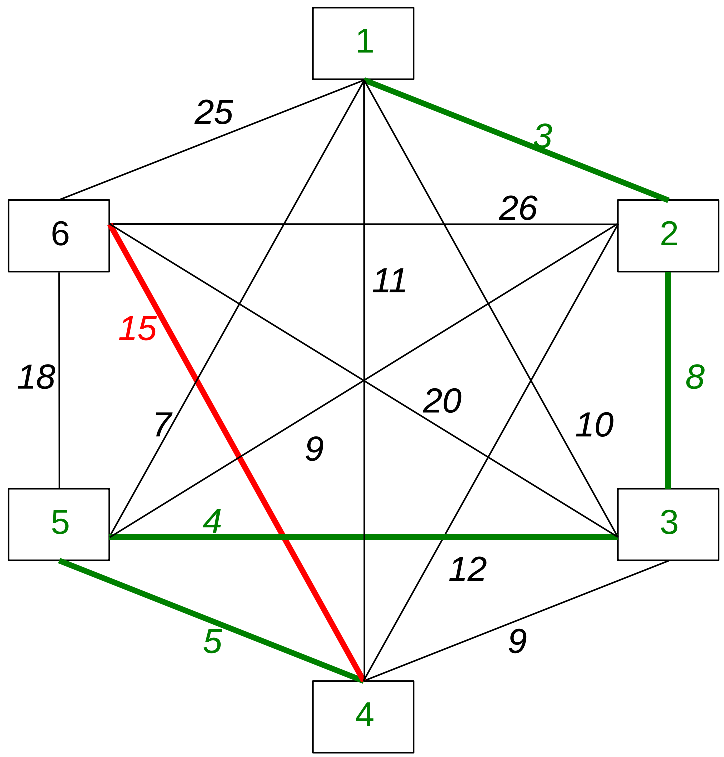
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Move to node 5



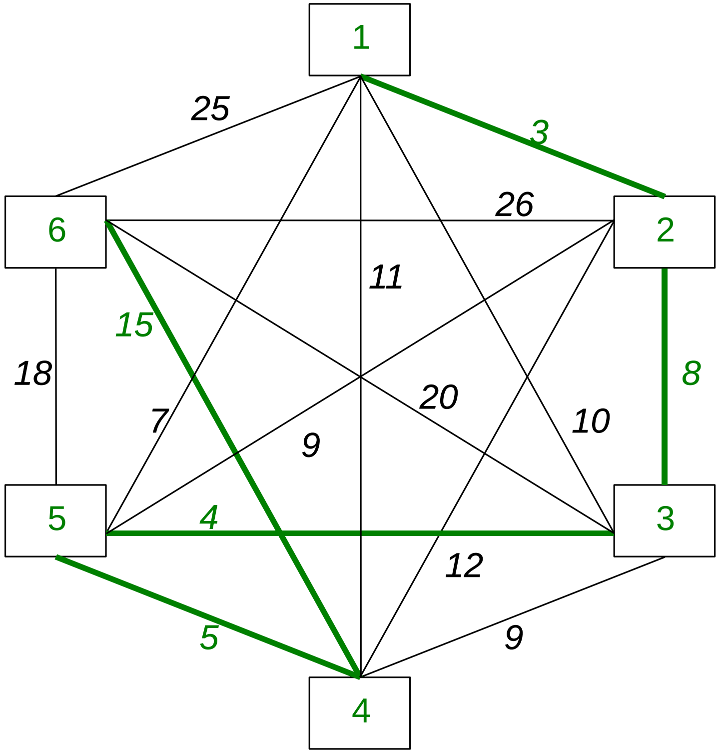
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Move to node 4



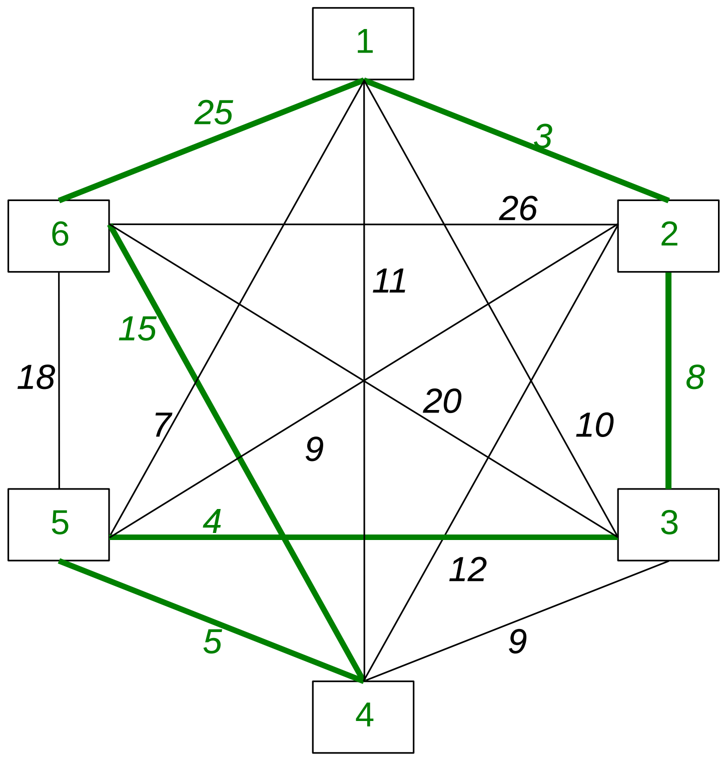
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Move to node 6



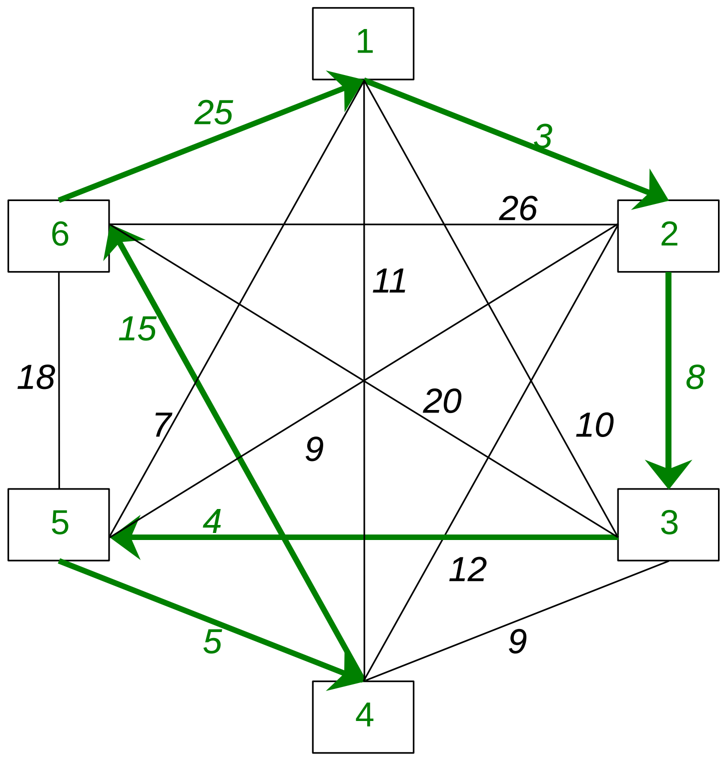
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Move back to node 1



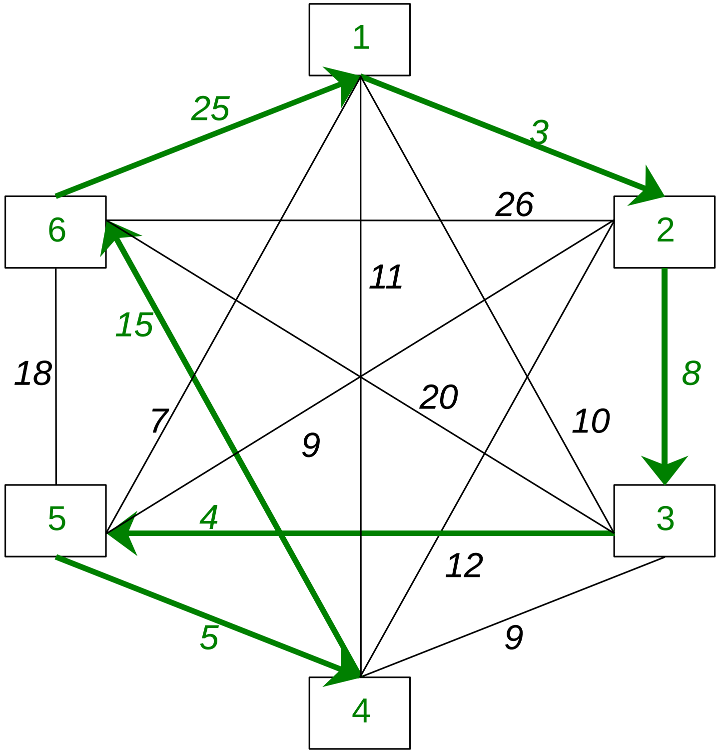
0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- The Traveling Salesman Problem
  - Route is 1 to 2 to 3 to 5 to 4 to 6 to1



0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

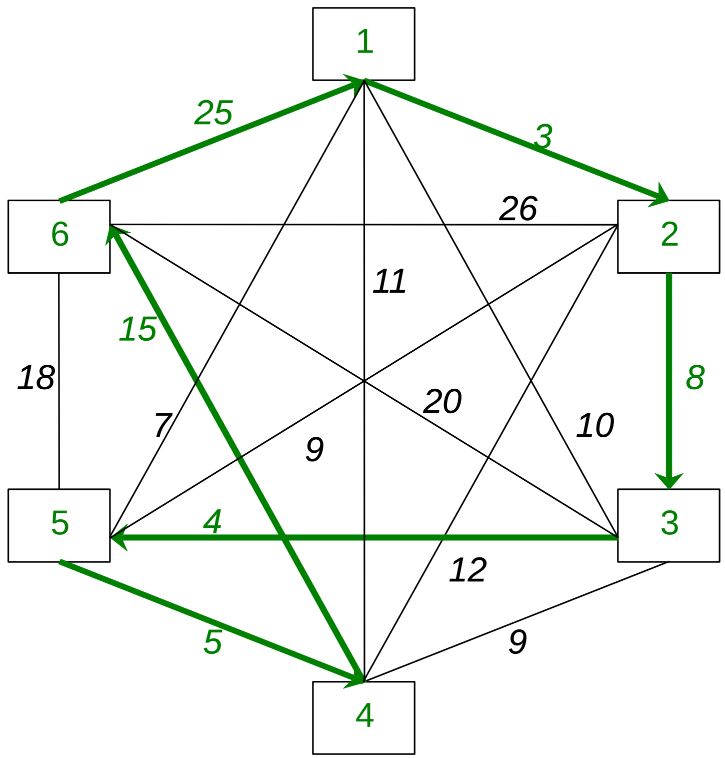
- The Traveling Salesman Problem
  - Total distance is 60

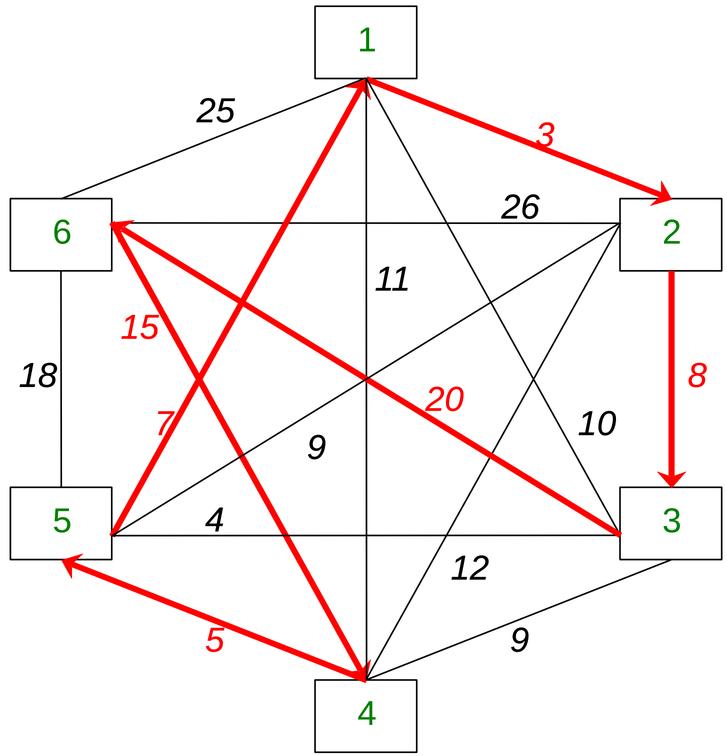


0	3	10	11	7	25
3	0	8	12	9	26
10	8	0	9	4	20
11	12	9	0	5	15
7	9	4	5	0	18
25	26	20	15	18	0

- is this optimal?

- The Traveling Salesman Problem
  - Total distance is 60

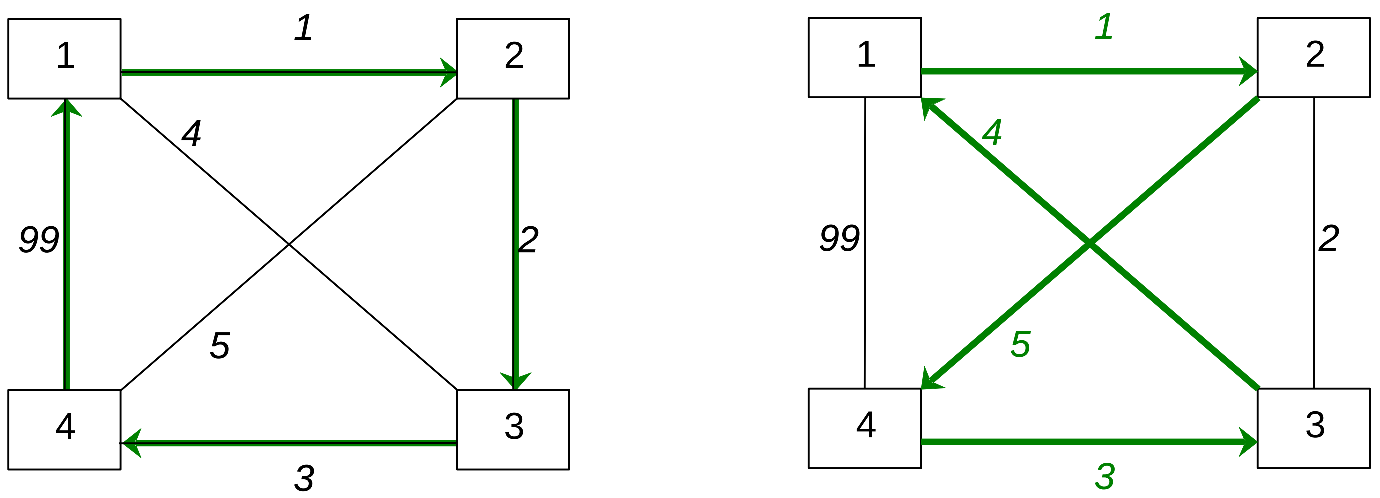




  - is this optimal? No 58 is.

- The Traveling Salesman Problem
  - It is close however.
  - Is this greedy algorithm at least near optimal?
  - Let us look at another problem.

- The Traveling Salesman Problem
  – Consider the following map:
    - The greedy algorithm gives path 1 to 2 to 3 to 4 to 1
    - With distance 105
    - Compared to 13



- Clearly, the greedy algorithm is not even close to optimal in this
  Case !

- Greedy Algorithms
  - Good in a wide range of problem classes
  - Generally, easy to implement
  - Generally, efficient
  - Sometimes not very good at all
  - Clearly, for some sorts of problem we need a different approach from the greedy one
  - Divide-and-Conquer is such an approach

Discussions

1. What is Greedy Strategy.
2. What is the Greedy Algorithm.

Homework

Assignment 2

Implement Prime Algorithm for Minimum Spanning Tree.