SIM GLOBAL EDUCATION

UNIVERSITY OF WOLLONGONG AUSTRALIA

CSCI203 – Algorithms and Data Structures

Greedy Algorithm

Sionggo Japit

sjapit@uow.edu.au

2 January 2023

Outline

- Greedy Algorithm
  - Making Changes with fewest coins
  - Shortest Distance
    - Dijkstra's Algorithm
  - Minimum Spanning Tree
    - Kruskal's Algorithm
    - Prim's Algorithm
  - Knapsack Problem

Objective

Upon completion of this topic, you should:

- Have an understanding of what a Greedy Algorithm is.
- Be able to differentiate the different categories of greedy algorithms
  - Describe how coin-changing algorithm works.
  - Describe finding shortest-distance algorithm, such as Dijkstra's algorithm.
  - Describe the two minimum spanning tree algorithms – Kruskal's and Prim's minimum spanning tree algorithms.
  - Describe subset sum algorithm such as Knapsack.


Greedy Algorithms

Greedy Algorithms

Greedy Algorithms

- Greedy algorithm solves optimization problems by making locally optimal choices and hoping that these choices lead to a globally optimal solution . 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)


Making changes with fewest coins

Making Changes With Fewest Coins

Example 1: making change

- Given we have $2, $1, 50¢, 20¢, 10¢, 5¢ and 1¢ 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 largest coin value possible, repeatedly do so until the amount is paid.
- E.g. to pay $17.97 we pay 8 $2 coins, 1 $1 coin, 1 50¢ coin, 2 20¢ coins, 1 5¢ coin and 2 1¢ coins(15 coins total).

Making Changes With Fewest Coins

- This is the optimal solution (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 12¢ coin would result in 15¢ being made from 1 12¢ and 3 1¢ (4 coins) instead of 1 10¢ and 1 5¢ coin (2 coins).

Greedy Algorithms

How Greedy Algorithms Work?

- 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

Greedy Algorithms

- 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

Shortest Path

Shortest Path

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
- The most popular shortest path algorithm is Dijkstra's Algorithm

Dijkstra's Algorithm

Dijkstra's Algorithm

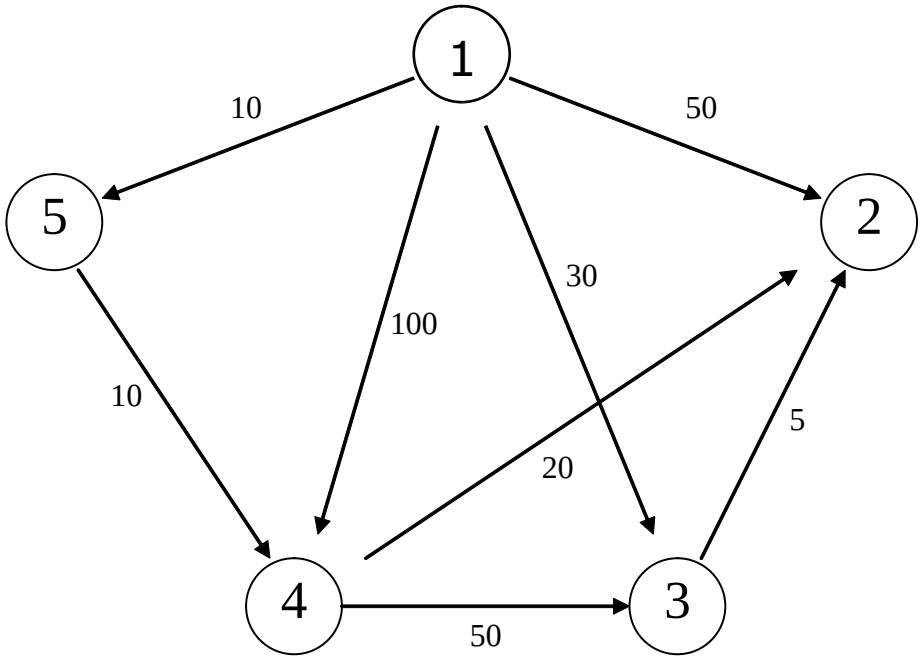
- Dijkstra's algorithm finds the shortest paths from a given node to all other nodes in a graph
  - Initially,
    - Mark the given node as *source* (path length is zero)
    - For each out-edge, set the distance in each neighboring node equal to the *cost* (length) of the out-edge, and set its *predecessor* to the initially given node (source)

Dijkstra's Algorithm

- Repeatedly (until all nodes are chosen),
  - Find a node from the candidate nodes containing the smallest distance
  - Mark the new node as *chosen*
  - For each node adjacent to the new node (chosen node), examine its neighbors to see whether their estimated distance can be reduced (distance to chosen node plus cost of out-edge)
    - If so, also reset the predecessor of the new node (chosen node)

Dijkstra's Algorithm – An Example

For example, find the shortest path from Node 1 to all other nodes.



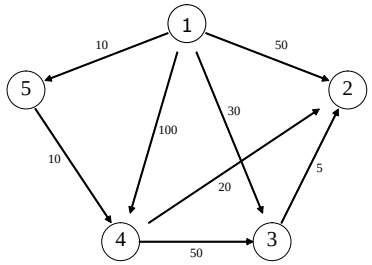
Dijkstra's Algorithm – An Example

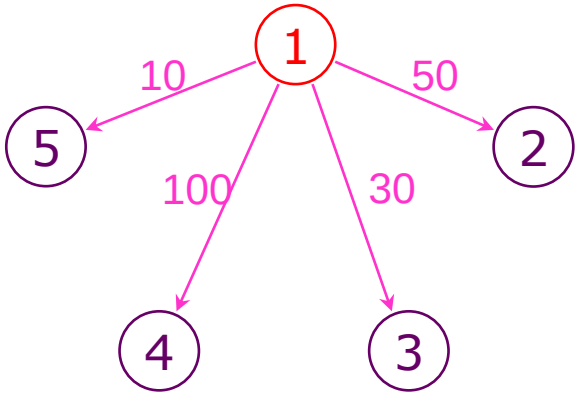
Dijkstra's Algorithm: An Example

- Step 0

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

D = [50, 30, 100, 10]





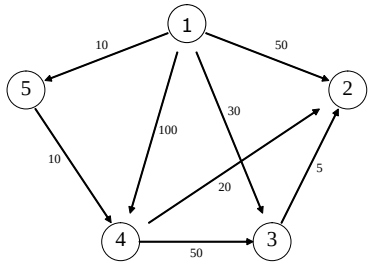
Dijkstra's Algorithm – An Example

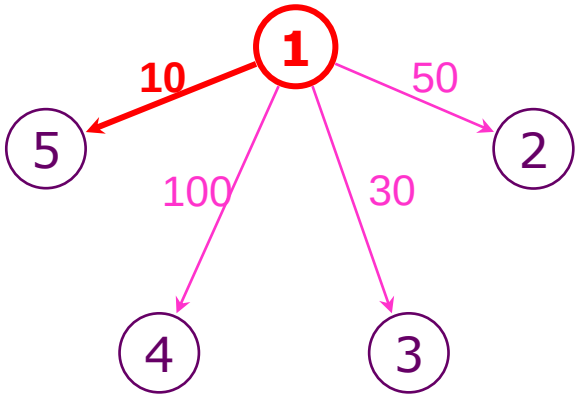
Dijkstra's Algorithm: An Example

- Step 0

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

D = [50, 30, 100, **10**]





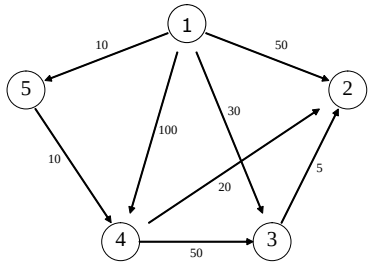
Dijkstra's Algorithm – An Example

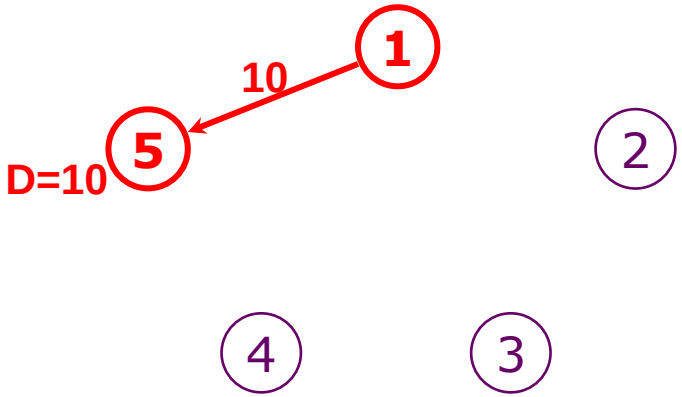
Dijkstra's Algorithm: An Example

- Step 0

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

D = [50, 30, 100, **10**]





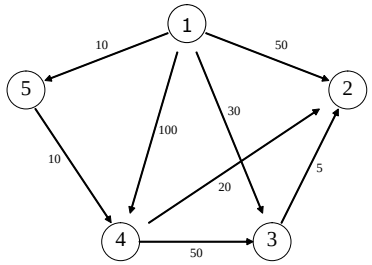
Dijkstra's Algorithm – An Example

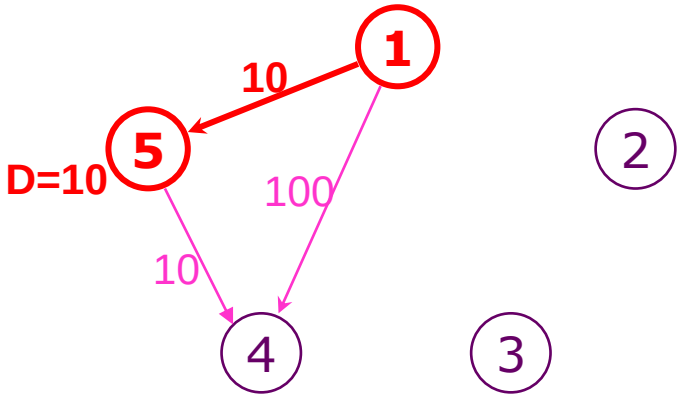
Dijkstra's Algorithm: An Example

- Step 1

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

D = [50, 30, 100, **10**]





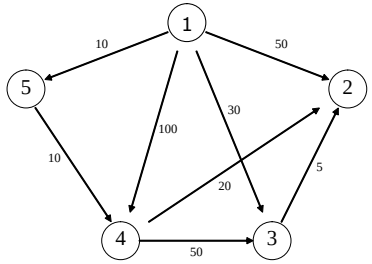
Dijkstra's Algorithm – An Example

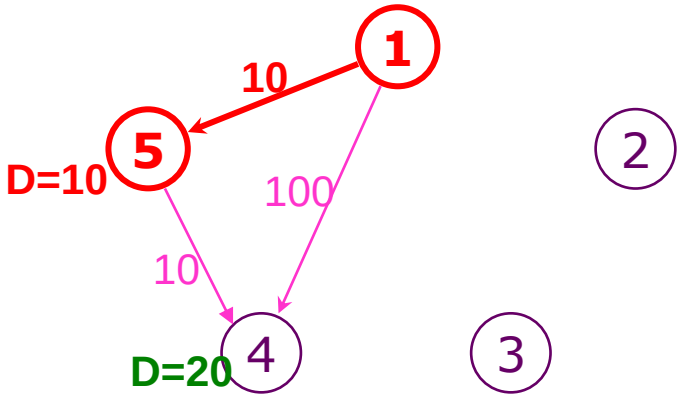
Dijkstra's Algorithm: An Example

- Step 1

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

D = [50, 30, **20**, **10**]





End of Step 0

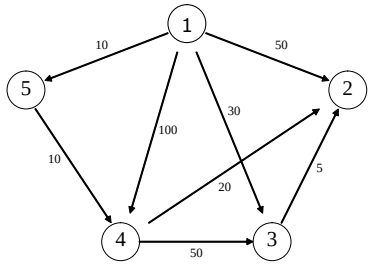
Dijkstra's Algorithm – An Example

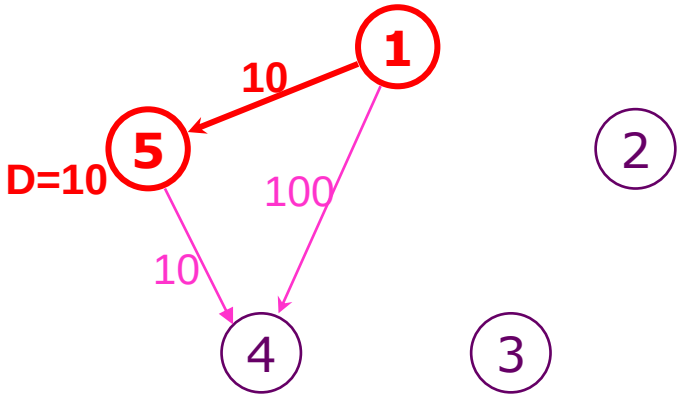
Dijkstra's Algorithm: An Example

- Step 1

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

D = [50, 30, 20, **10**]





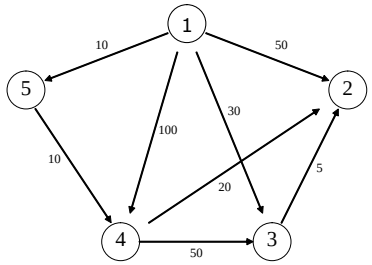
Dijkstra's Algorithm – An Example

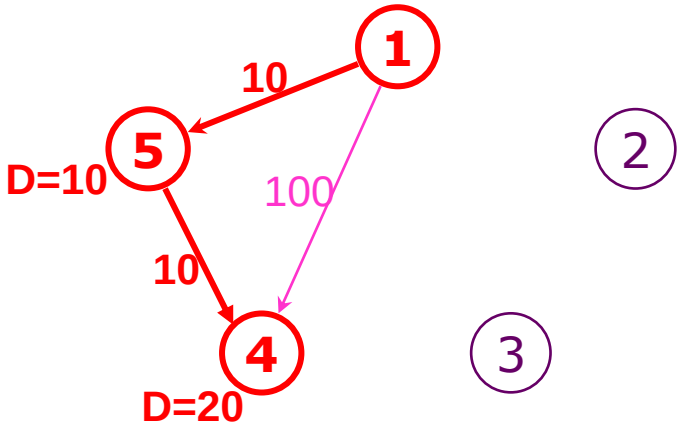
Dijkstra's Algorithm: An Example

- Step 1

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

D = [50, 30, **20**, **10**]





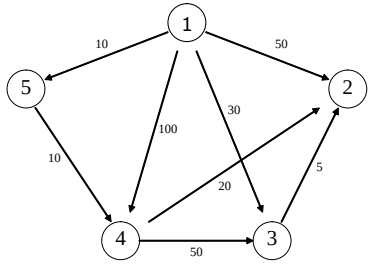
Dijkstra's Algorithm – An Example

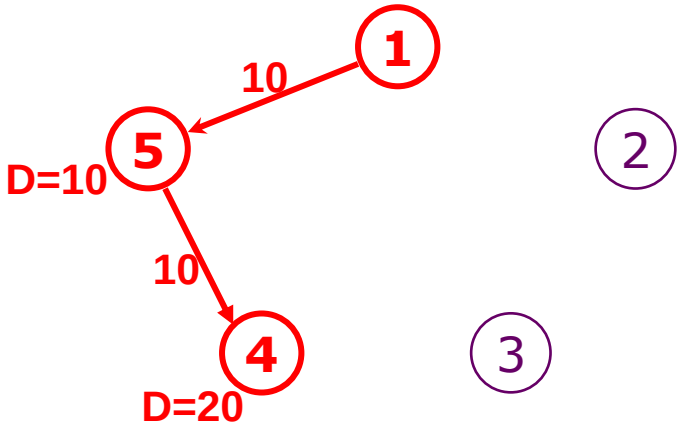
Dijkstra's Algorithm: An Example

- Step 1

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

D = [50, 30, **20**, **10**]





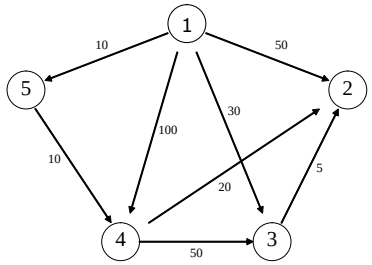
Dijkstra's Algorithm – An Example

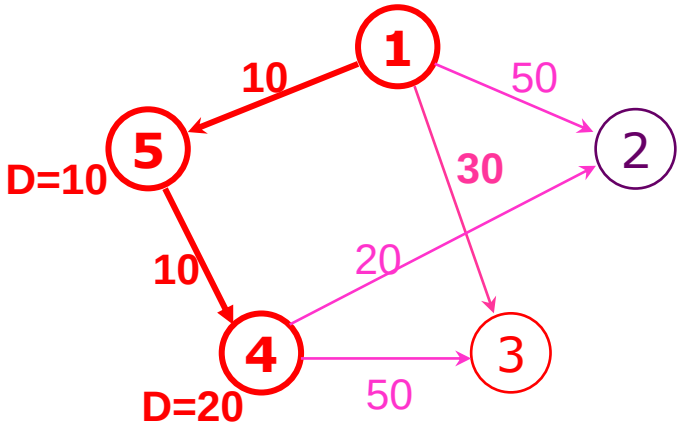
Dijkstra's Algorithm: An Example

- Step 2

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

D = [50, 30, **20**, **10**]





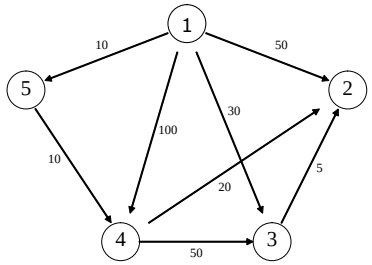
Dijkstra's Algorithm – An Example

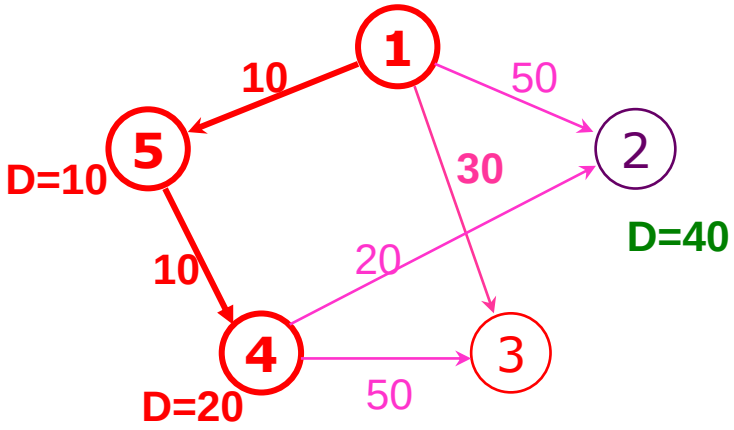
Dijkstra's Algorithm: An Example

- Step 2

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

D = [40, 30, **20**, **10**]





End of Step 1

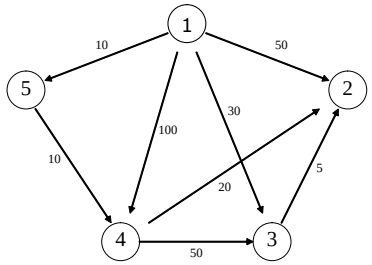
Dijkstra's Algorithm – An Example

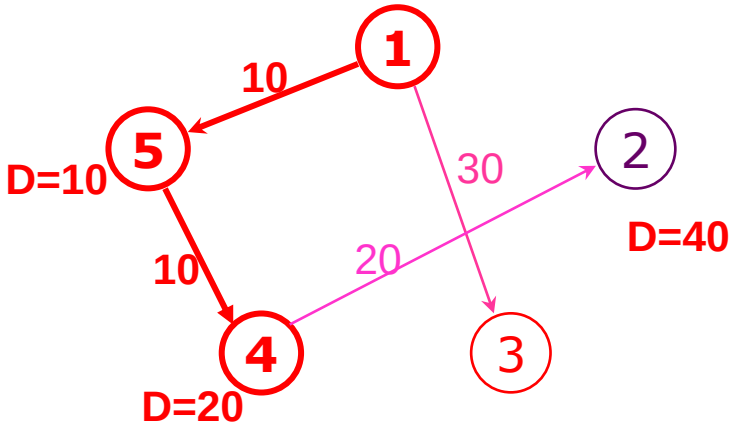
Dijkstra's Algorithm: An Example

- Step 2

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

D = [40, 30, **20**, **10**]





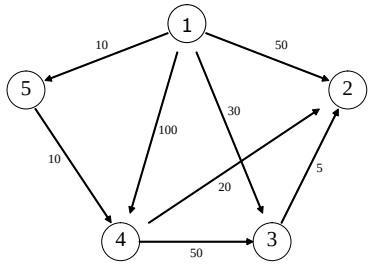
Dijkstra's Algorithm – An Example

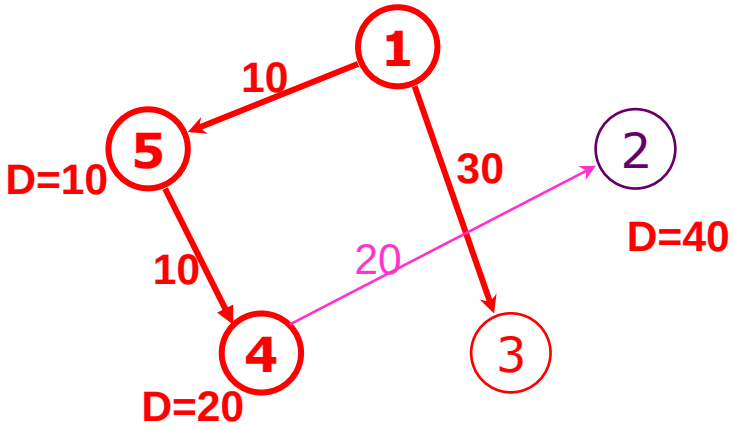
Dijkstra's Algorithm: An Example

- Step 2

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

D = [40, **30**, **20**, **10**]





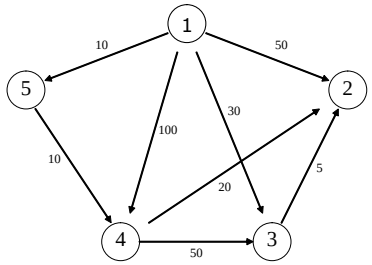
Dijkstra's Algorithm – An Example

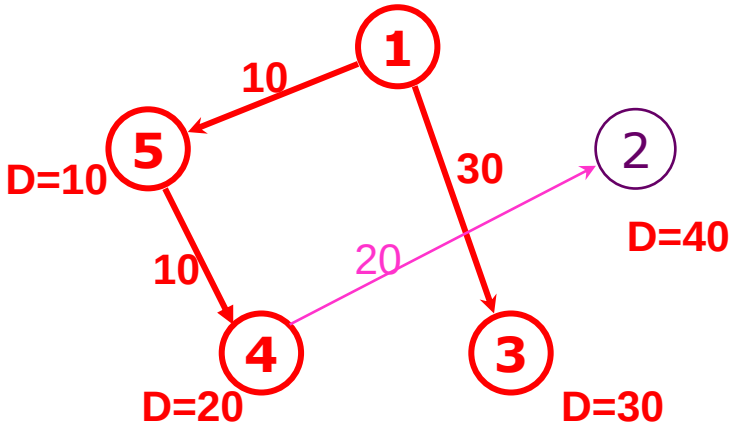
Dijkstra's Algorithm: An Example

- Step 2

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

D = [40, **30**, **20**, **10**]





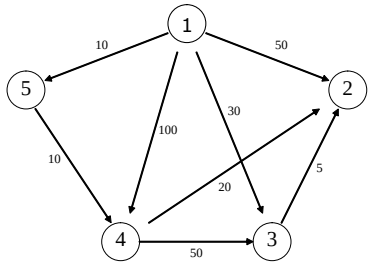
Dijkstra's Algorithm – An Example

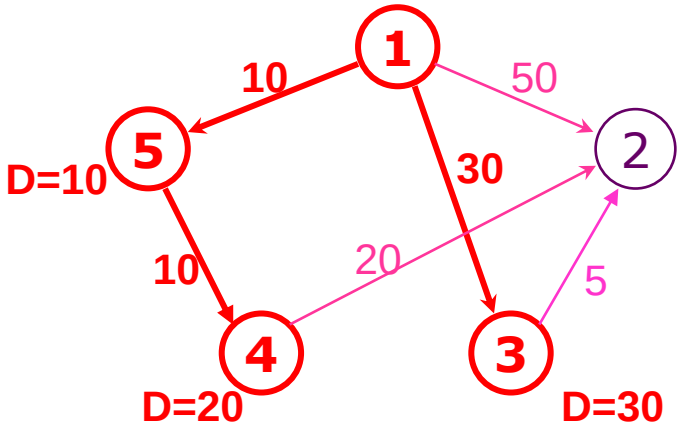
Dijkstra's Algorithm: An Example

- Step 3

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

D = [40, **30**, **20**, **10**]





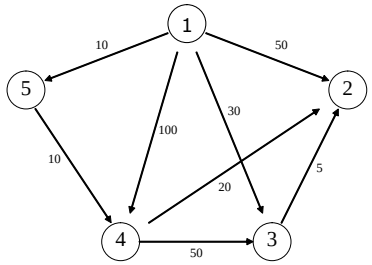
Dijkstra's Algorithm – An Example

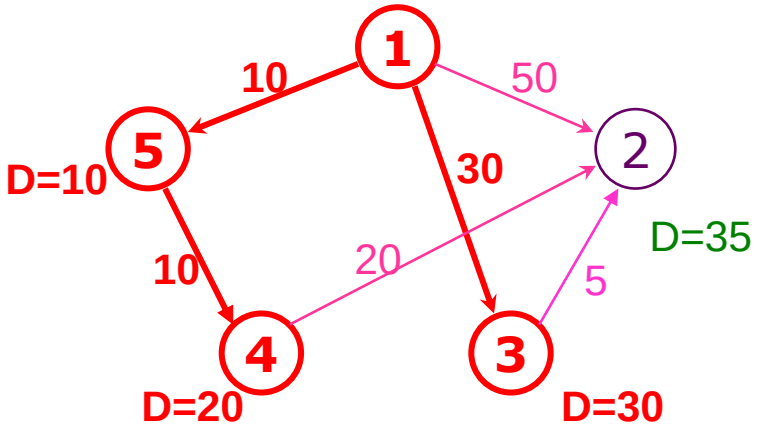
Dijkstra's Algorithm: An Example

- Step 3

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

D = [35, **30**, **20**, **10**]





End of Step 2

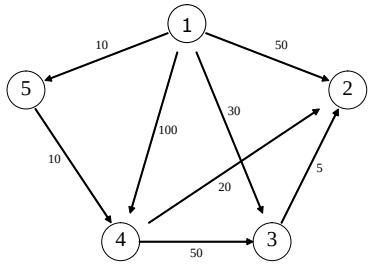
Dijkstra's Algorithm – An Example

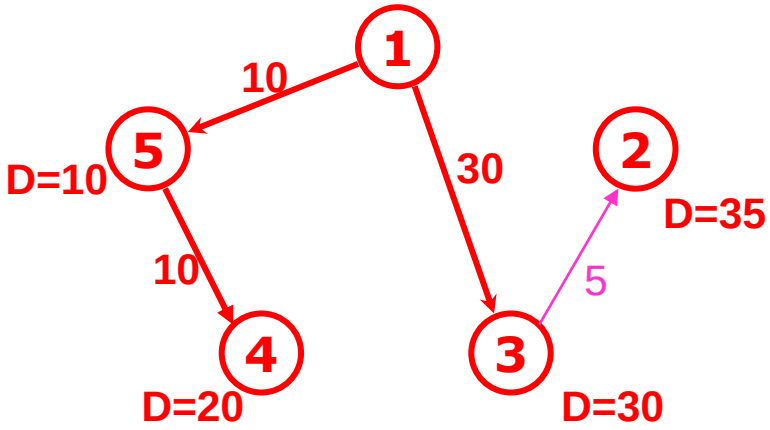
Dijkstra's Algorithm: An Example

- Step 3

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

D = [35, **30**, **20**, **10**]





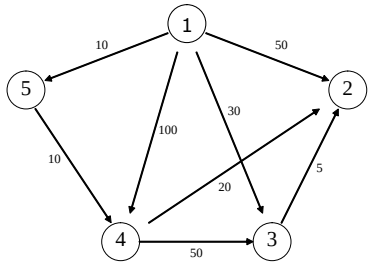
Dijkstra's Algorithm – An Example

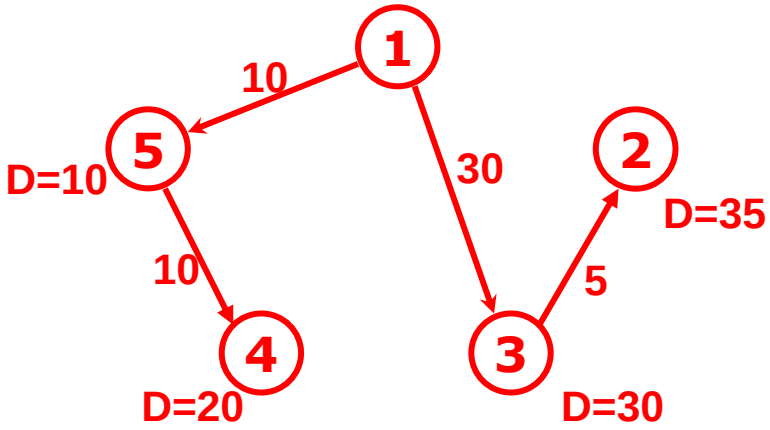
Dijkstra's Algorithm: An Example

- Step 3

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

D = [**35**, **30**, **20**, **10**]





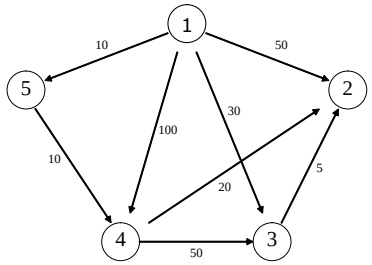
Dijkstra's Algorithm – An Example

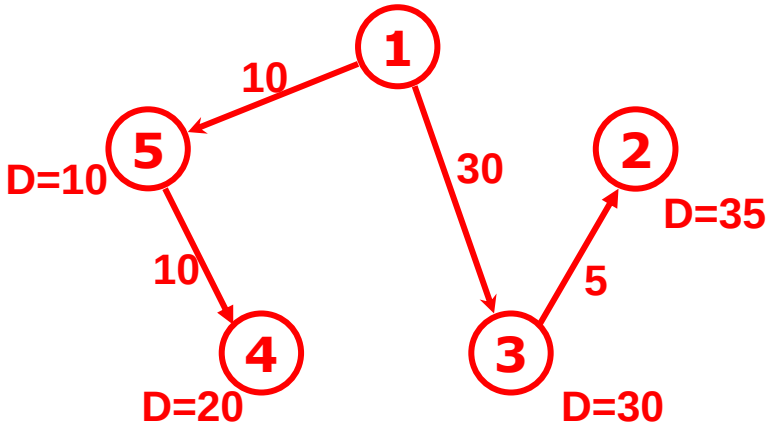
Dijkstra's Algorithm: An Example

- Step 3

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

D = [**35**, **30**, **20**, **10**]





End of Step 3

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

Dijkstra's Algorithm

- 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 ∈ C do
        D[w] = min(D[w], D[v] + L[v, w])
return D
```

Initialis e D

There are n-1 nodes to visit. Visit the nearest node from 1->5 first.

If there exists a shorter path to reach the rest of the unvisited nodes (w) from the current path v then update the cost to reach them via this path.

The existing cost to reach w

The new cost to reach w from the shortest path

Dijkstra's Algorithm

Dijkstra's Algorithm

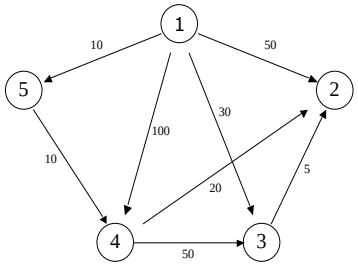
- Function Dijkstra(L[1..n, 1..n]): array [2..n]
  array D[2..n]
  C = {2, 3, …, n}
  for i = 2 to n do
    D[i] = L[1, i]
  repeat n – 2 times
    v = the index of the minimum D[v] not yet selected
    remove v from C // and implicitly add v to S
    for each w $\in$ C do
      D[w] = min(D[w], D[v]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

Dijkstra's Algorithm

Dijkstra's Algorithm (modified to record 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 $\in$ 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



Dijkstra's Algorithm: at start

L =

P =

D =

From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	10
5	10	∞	∞	∞	∞

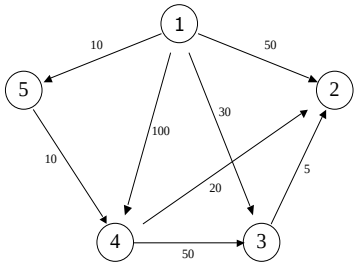
	V	
1	2	50
1	3	30
1	4	100
1	5	10

V =

C = {2, 3, 4, 5}

S = {1}

Dijkstra's Algorithm



Dijkstra's Algorithm: at start

L =

P =

D =

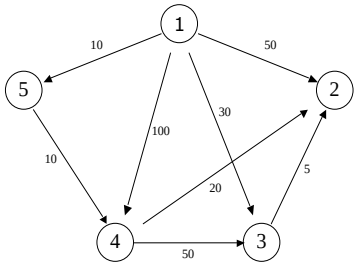
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	10
5	10	∞	∞	∞	∞

	V	
1	2	50
1	3	30
1	4	100
1	5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: at start

L =

P =

D =

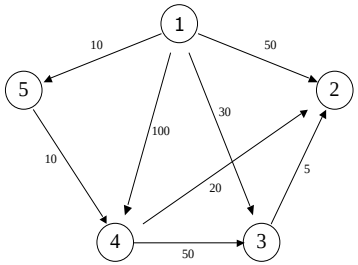
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	~~10~~
5	10	∞	∞	∞	∞

	V	
1	2	50
1	3	30
1	4	100
1	5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: at start

L =

P =

D =

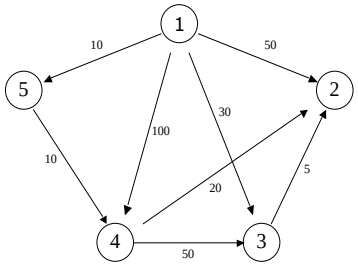
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	~~10~~
5	10	∞	∞	∞	∞

	V	
1	2	50
1	3	30
5	4	20
1	5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: after iteration 1

L =

P =

D =

From:
To:

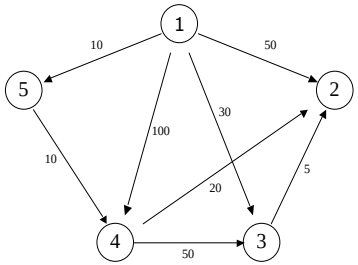
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

1	
1	
5	
1	

V	
2	50
3	30
4	20
5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: start of iteration 2

L =

P =

D =

From:
To:

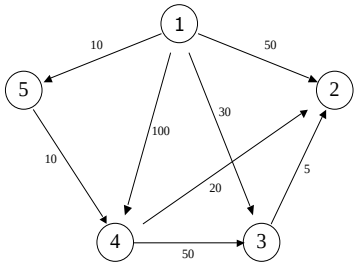
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	20	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

V	
2	1
3	1
4	5
5	1

2	50
3	30
4	**20**
5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: iteration 2

L =

P =

D =

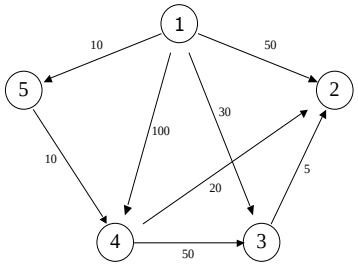
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	~~20~~	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

	V	
1	2	50
1	3	30
5	4	20
1	5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: iteration 2

L =

P =

D =

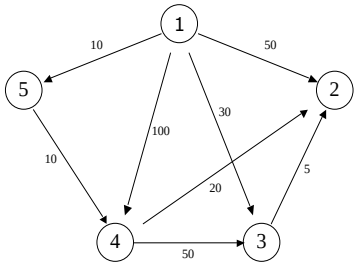
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	~~20~~	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

	V	
~~4~~	2	40
1	3	30
5	4	20
1	5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: after iteration 2

L =

P =

D =

From:
To:

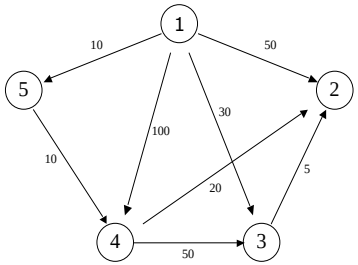
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

4	
1	
5	
1	

V	
2	40
3	30
4	**20**
5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: start of iteration 3

L =

P =

D =

From:
To:

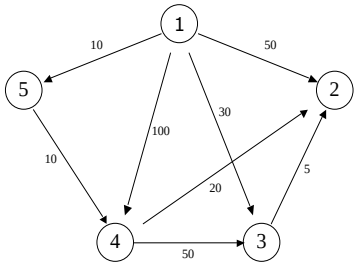
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

4	
1	
5	
1	

V	
2	40
3	**30**
4	**20**
5	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: iteration 3

L =

P =

D =

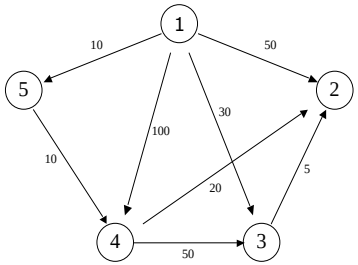
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

V		
2	4	40
3	1	**30**
4	5	**20**
5	1	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: iteration 3

L =

P =

D =

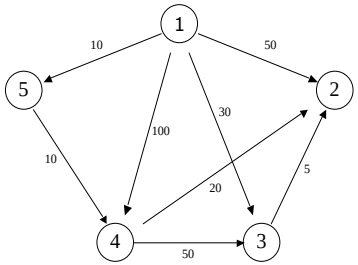
From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	5	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

	3	35
V	1	**30**
	5	**20**
	1	**10**

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

Dijkstra's Algorithm



Dijkstra's Algorithm: after iteration 3

L =

P =

D =

From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	**5**	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

3	
1	
5	
1	

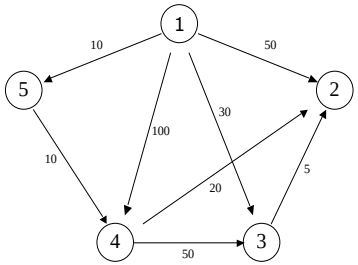
V	
2	35
3	**30**
4	**20**
5	**10**

V =

C = {2}

S = {1, 4, 5, 3}

Dijkstra's Algorithm



Dijkstra's Algorithm: start iteration 4

L =

P =

D =

From:
To:

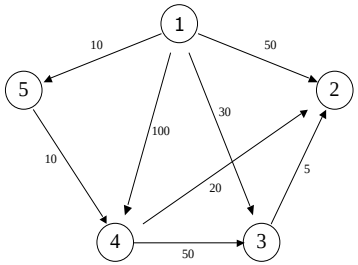
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	**5**	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

V	
2	**35**
3	**30**
4	**20**
5	**10**

3
1
5
1

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

Dijkstra's Algorithm



Dijkstra's Algorithm: after iteration 4

L =

P =

D =

From:
To:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	50	∞	**5**	**20**	∞
3	30	∞	∞	50	∞
4	100	∞	∞	∞	**10**
5	10	∞	∞	∞	∞

	V
3	2
1	3
5	4
1	5

35
30
20
10

V =

C = {}

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

Dijkstra's Algorithm

Dijkstra's Algorithm:

- Paths:

To 2 – 1, 3, 2

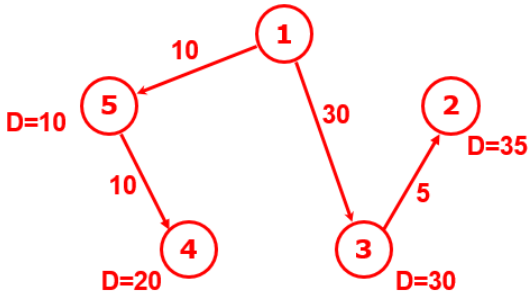
To 3 – 1, 3

To 4 – 1, 5, 4

To 5 – 1, 5

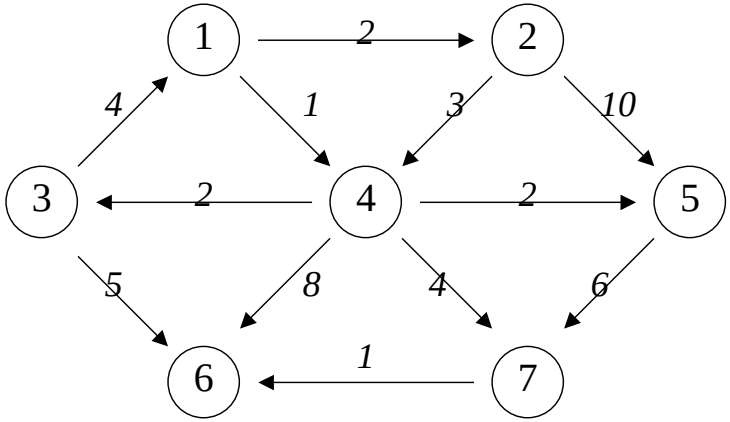
P =

V	
2	3
3	1
4	5
5	1



Dijkstra’s Algorithm

- Dijkstra’s Algorithm: Another Example



Dijkstra's Algorithm

- Dijkstra's Algorithm: At start
  - L =

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8		∞	1
∞	∞	∞	4	6	∞	∞

P =

1
1
1
1
1
1

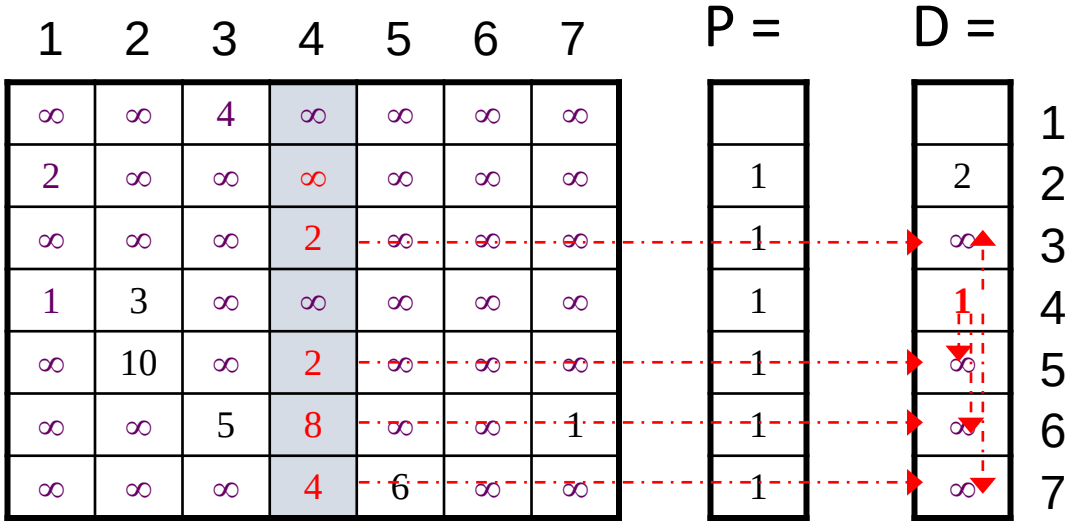
D =

	1
2	2
∞	3
1	4
∞	5
∞	6
∞	7

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

Dijkstra's Algorithm

- Dijkstra's Algorithm: At start
  - L =



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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

	1
2	2
3	3
1	4
3	5
9	6
5	7

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

	1
2	2
3	3
1	4
3	5
9	6
5	7

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

	1
2	2
3	3
1	4
3	5
9	6
5	7

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

D =

	1
2	2
3	3
1	4
3	5
9	6
5	7

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
4
4

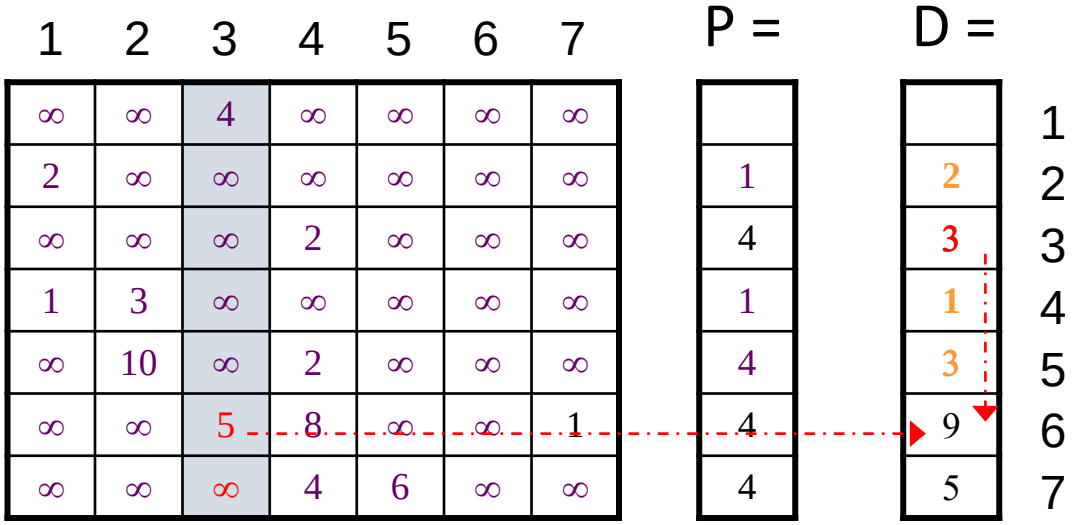
D =

	1
2	2
3	3
1	4
3	5
9	6
5	7

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

Dijkstra's Algorithm

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



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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
3
4

D =

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

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
3
4

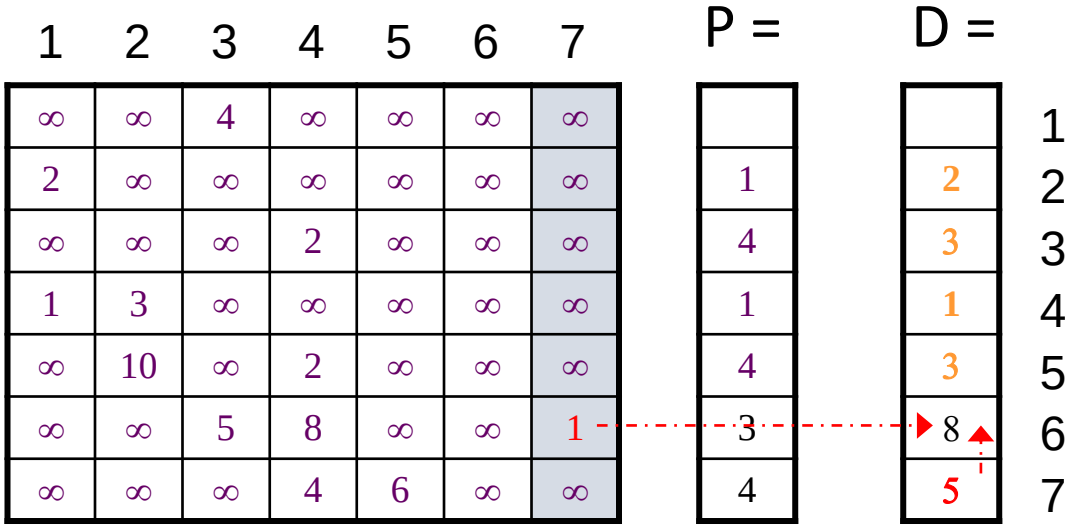
D =

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

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

Dijkstra's Algorithm

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



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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
7
4

D =

	1
2	2
3	3
1	4
3	5
6	6
5	7

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

Dijkstra's Algorithm

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

1	2	3	4	5	6	7
∞	∞	4	∞	∞	∞	∞
2	∞	∞	∞	∞	∞	∞
∞	∞	∞	2	∞	∞	∞
1	3	∞	∞	∞	∞	∞
∞	10	∞	2	∞	∞	∞
∞	∞	5	8	∞	∞	1
∞	∞	∞	4	6	∞	∞

P =

1
4
1
4
7
4

D =

	1
2	2
3	3
1	4
3	5
6	6
5	7

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

Dijkstra's Algorithm

- 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 =

	1
2	2
3	3
1	4
3	5
6	6
5	7

Minimum Spanning Tree

Minimum Spanning Tree

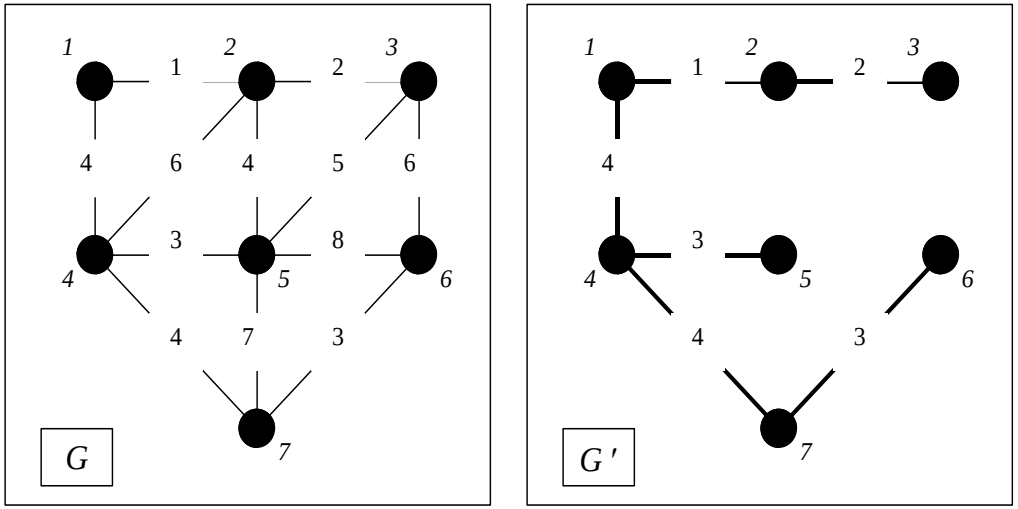
Minimum Spanning Tree

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

Minimum Spanning Tree

Example 3: Minimum Spanning Tree

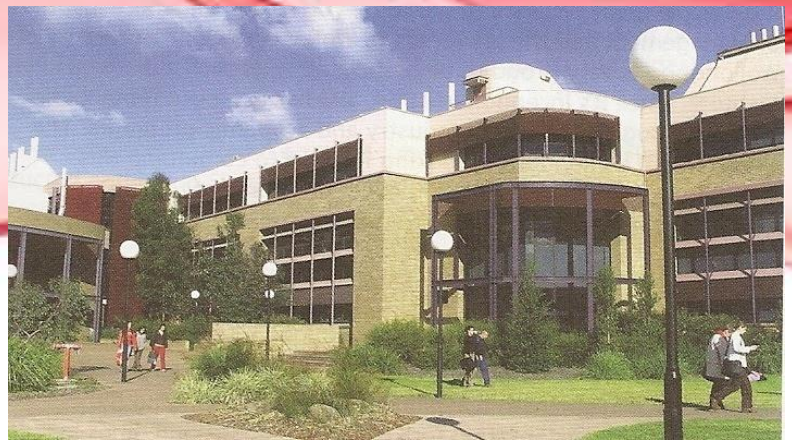


Minimum Spanning Tree

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* the shortest edge that extends the graph to a new node
- Strangely, both approaches work

Kruskal's Algorithm



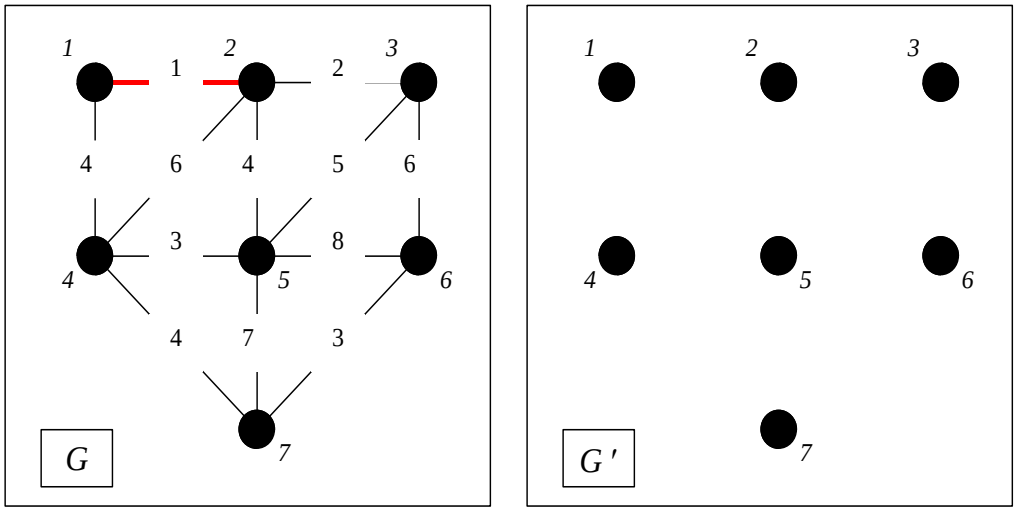
Kruskal's Algorithm

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.

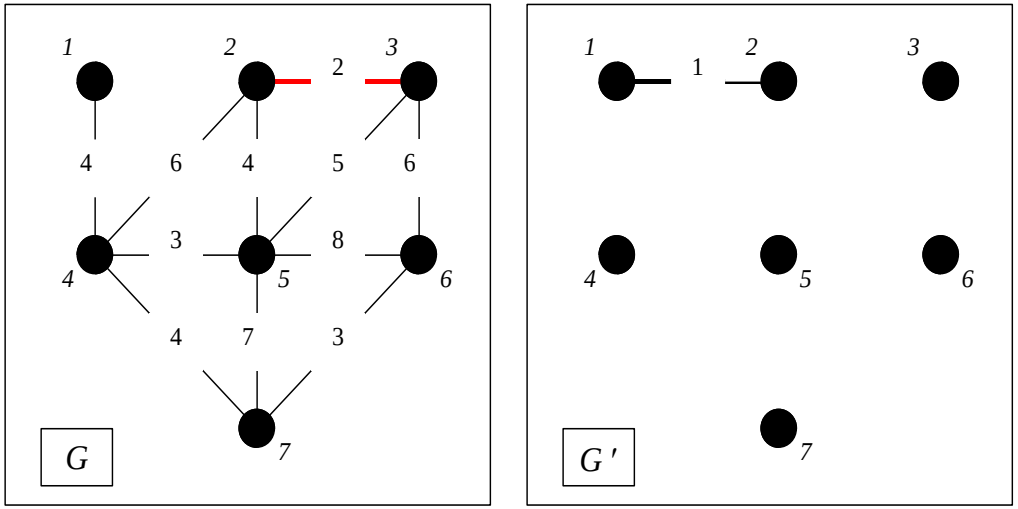
Kruskal's Algorithm

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



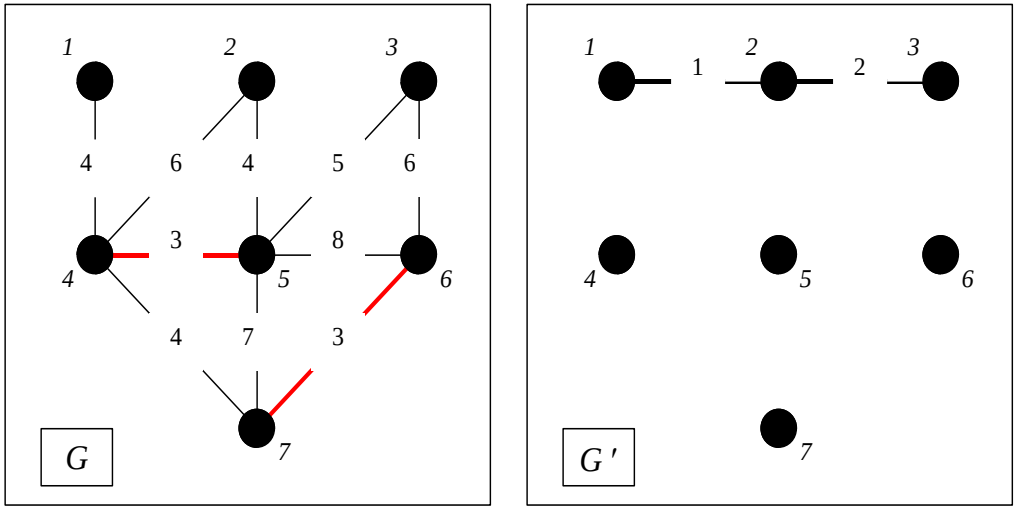
Kruskal's Algorithm

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



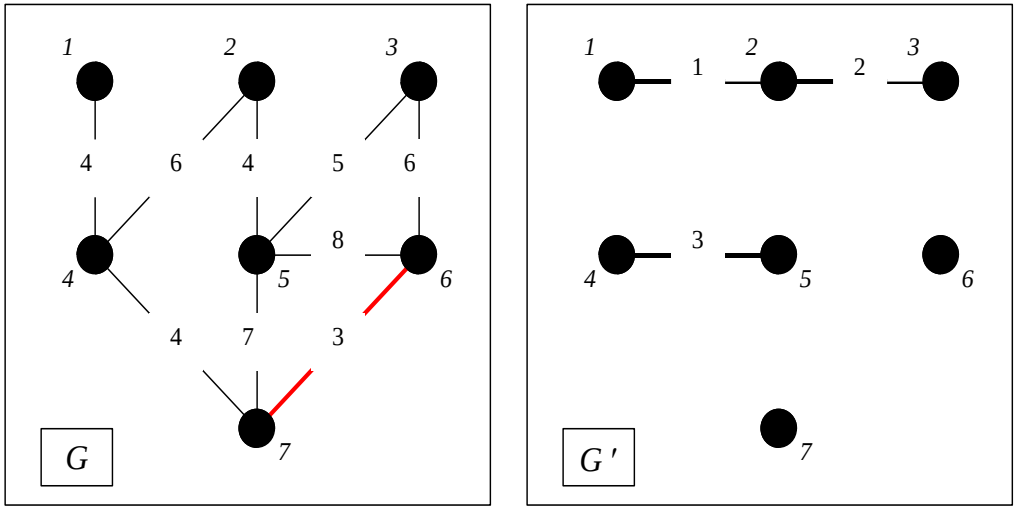
Kruskal's Algorithm

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



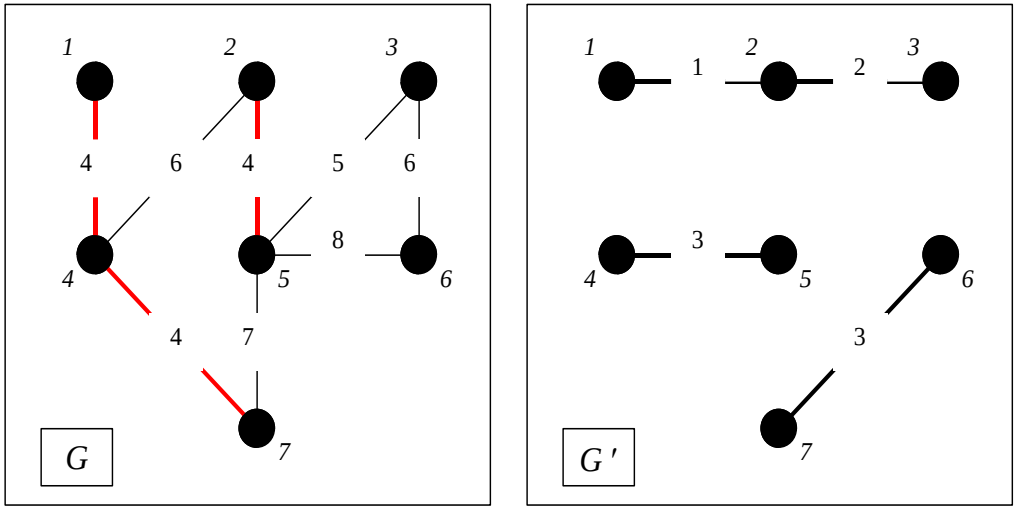
Kruskal's Algorithm

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



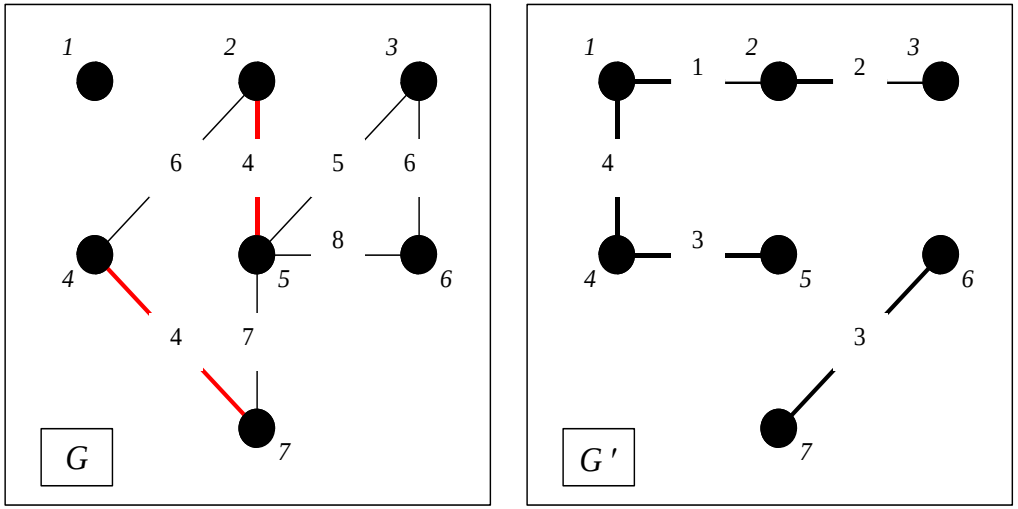
Kruskal's Algorithm

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



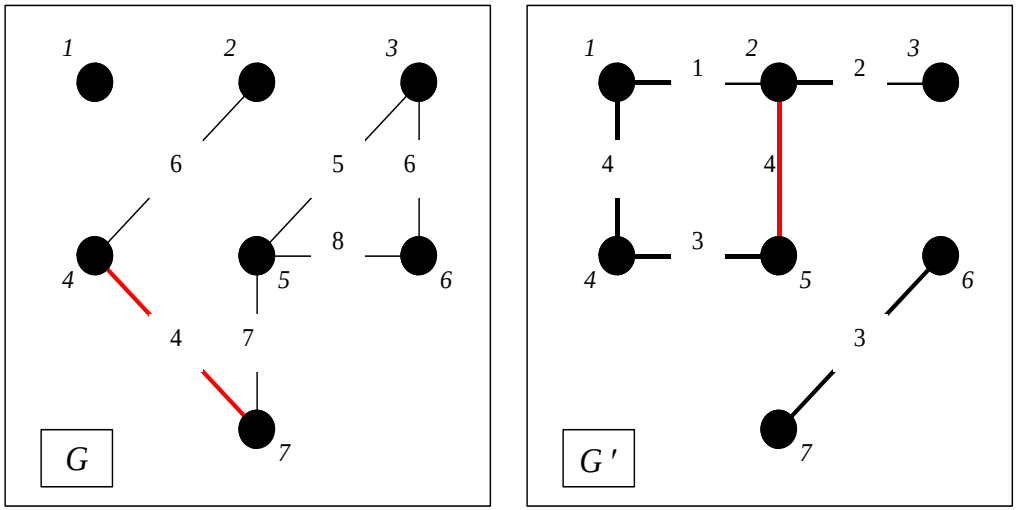
Kruskal's Algorithm

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



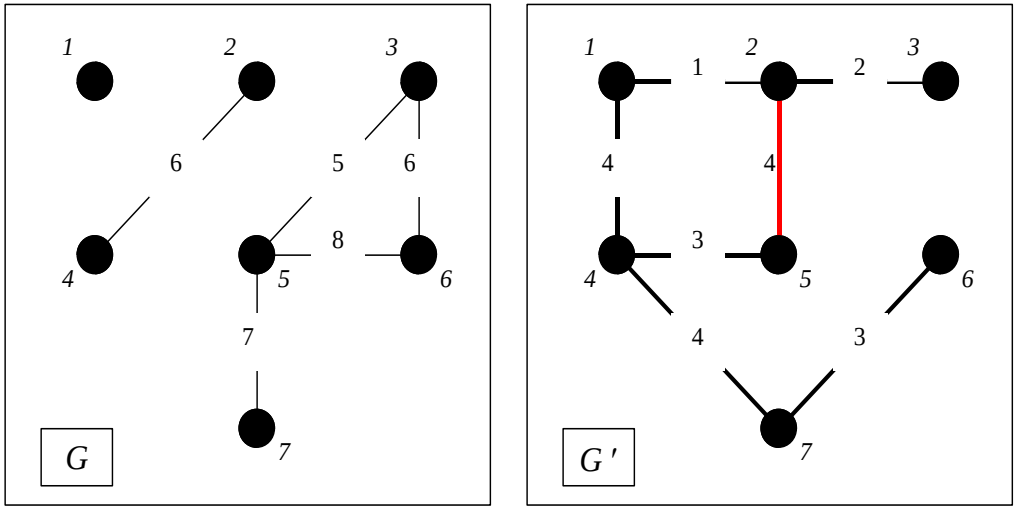
Kruskal's Algorithm

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



Kruskal's Algorithm

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



Kruskal's Algorithm

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

Kruskal's Algorithm

- 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 ¹ vset then
      merge(uset, vset)
      add E[i] to S
until S contains n – 1 edges
return S
```

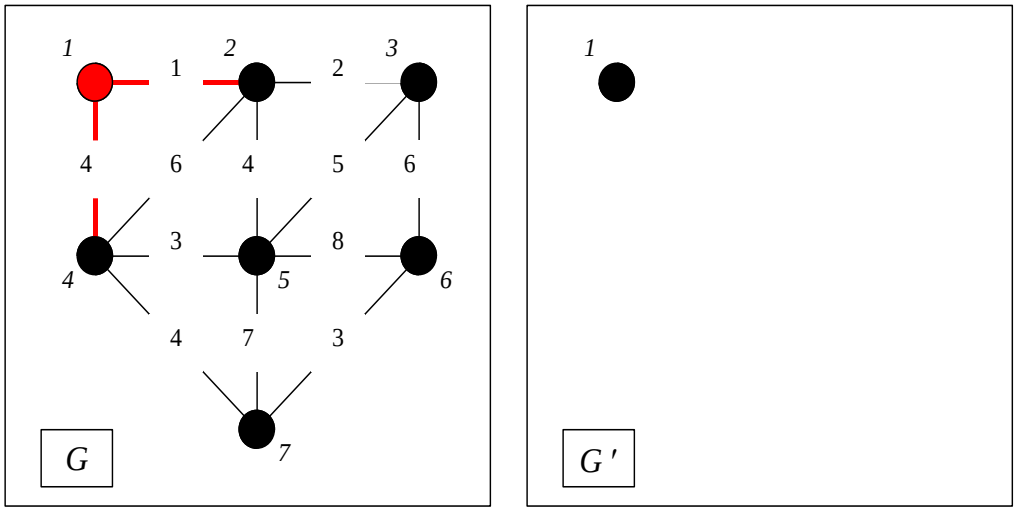
Prim's Algorithm

Prim's Algorithm

- 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*

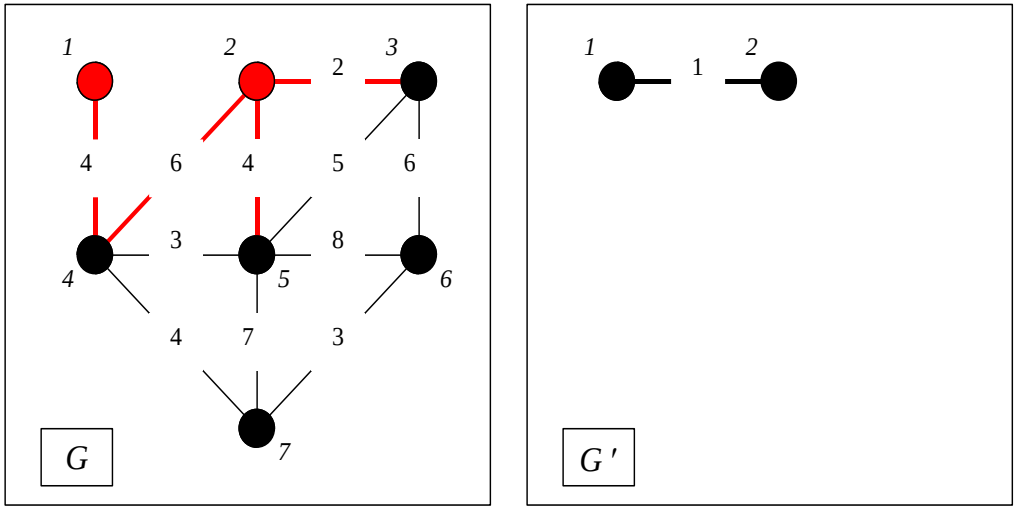
Prim's Algorithm

- Prim's Algorithm: An Example
  - Step 0 - {1}



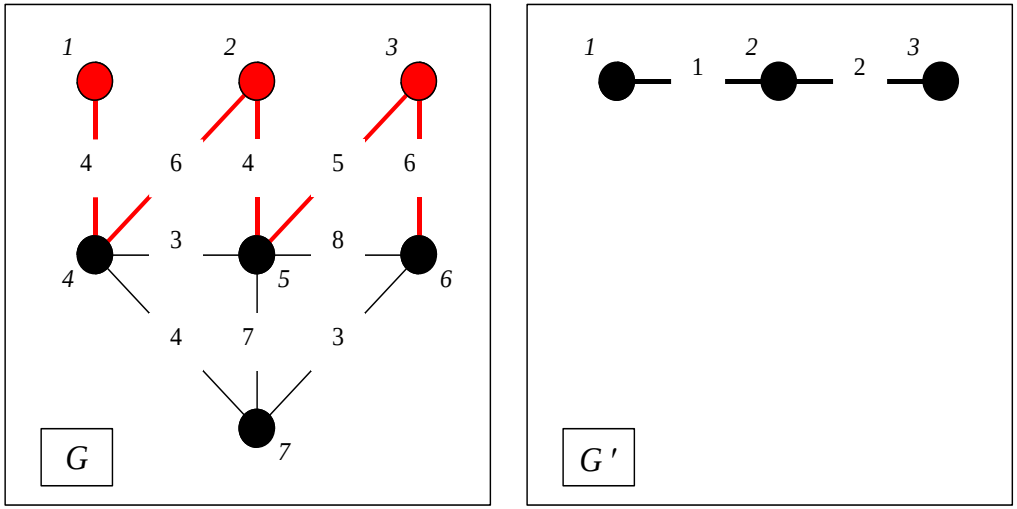
Prim's Algorithm

- Prim's Algorithm: An Example
  - Step 1 {1, 2} {1, 2}



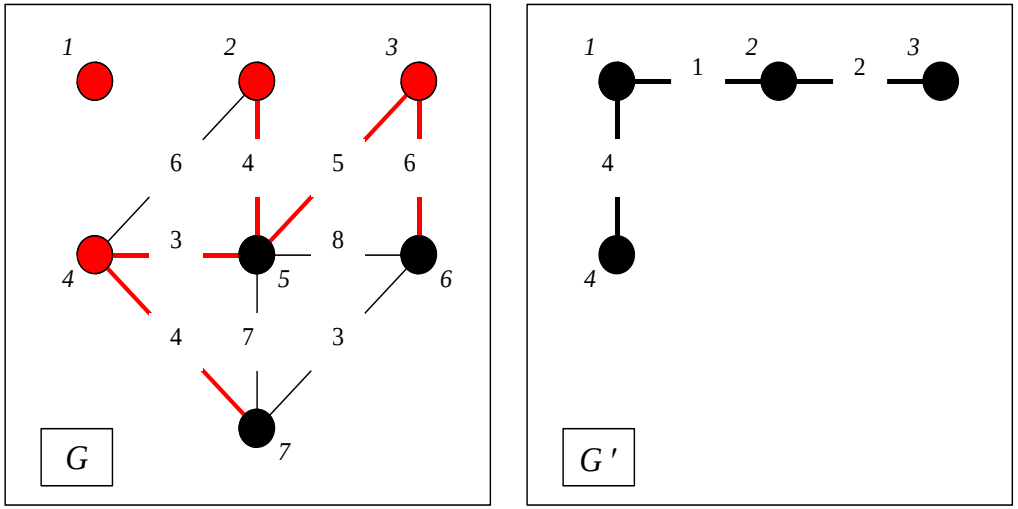
Prim's Algorithm

- Prim's Algorithm: An Example
  - Step 2 {2, 3} {1, 2, 3}



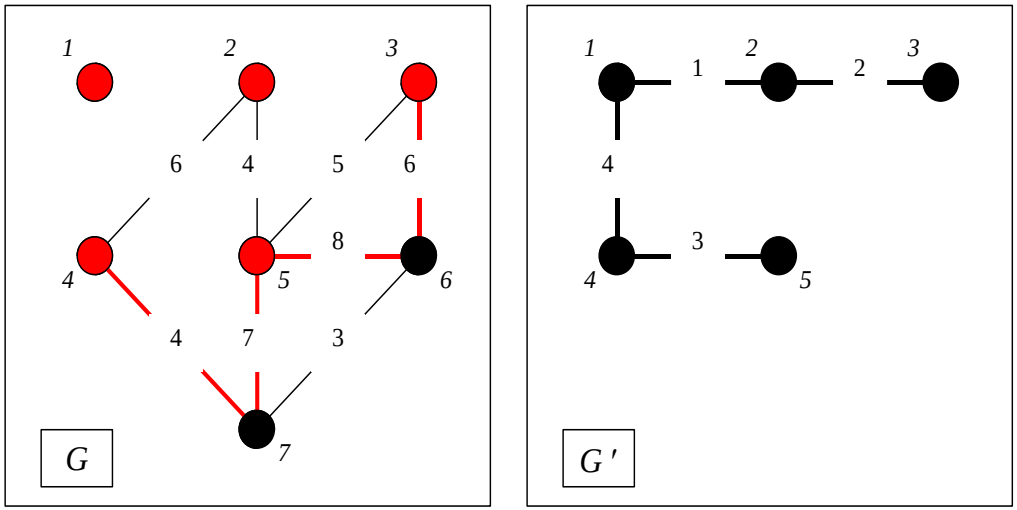
Prim's Algorithm

- Prim's Algorithm: An Example
  - Step 3 {1, 4} {1, 2, 3, 4}



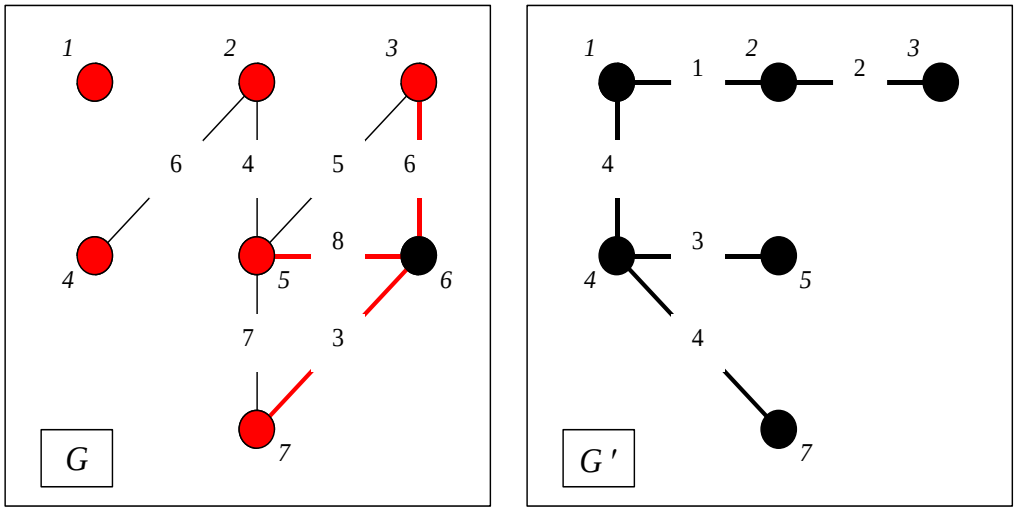
Prim's Algorithm

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



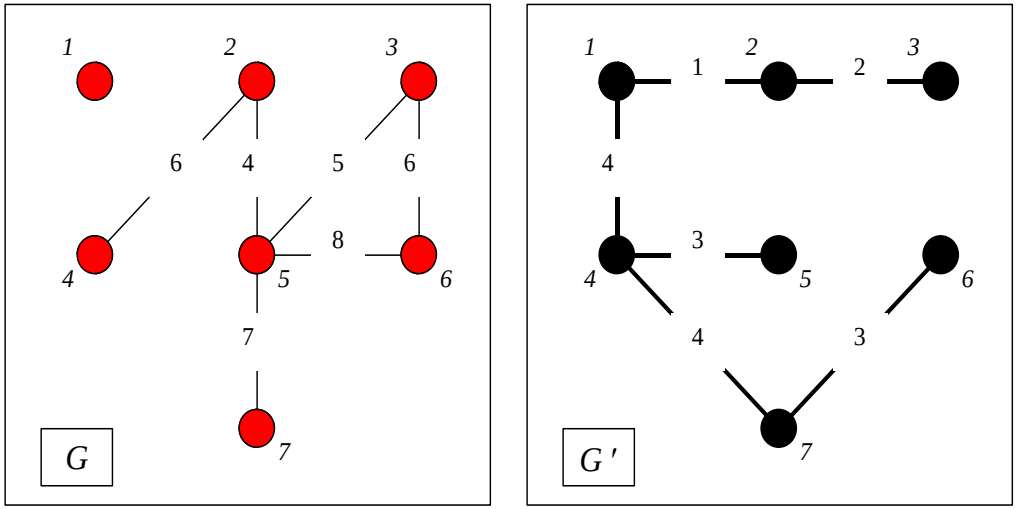
Prim's Algorithm

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



Prim's Algorithm

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



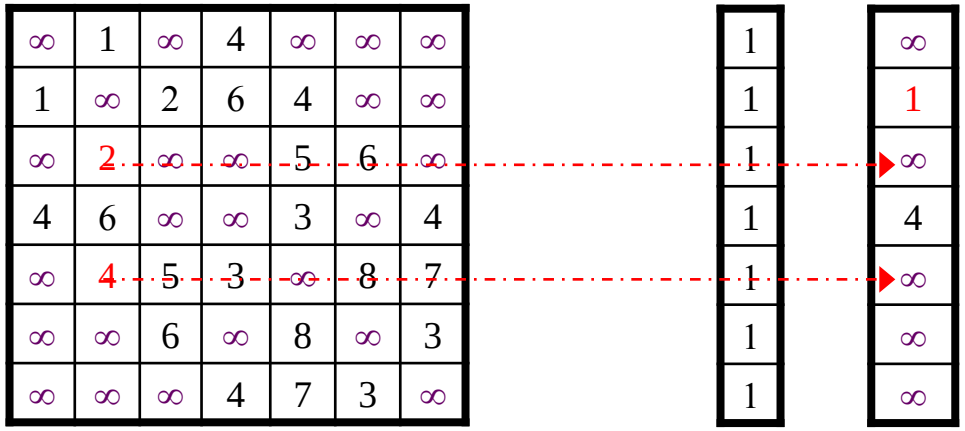
Prim's Algorithm

Prim's Algorithm

- Function Prim(L[1..n, 1..n])
  // L[i, j] is the length of the edge from node i to node j L[I, j] = ∞ if no edge
  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

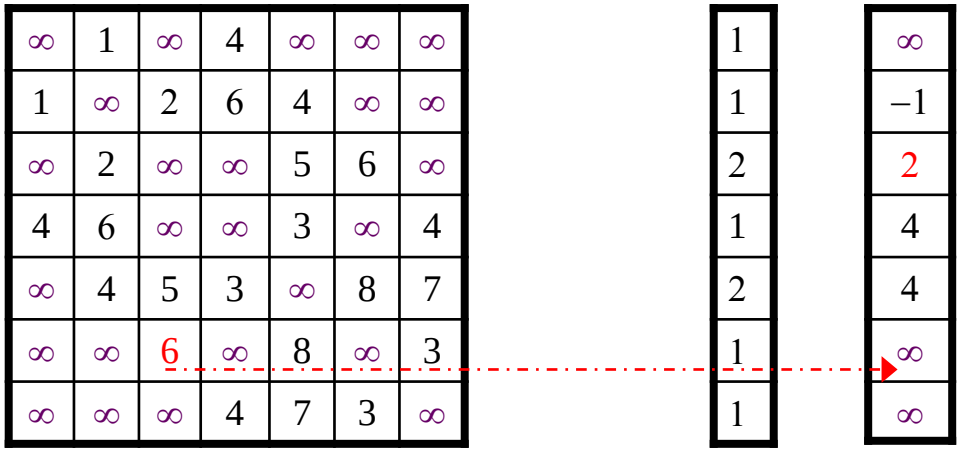
- Prim's Algorithm: at start
  - L = nearest = mindist =



S = {}

Prim's Algorithm

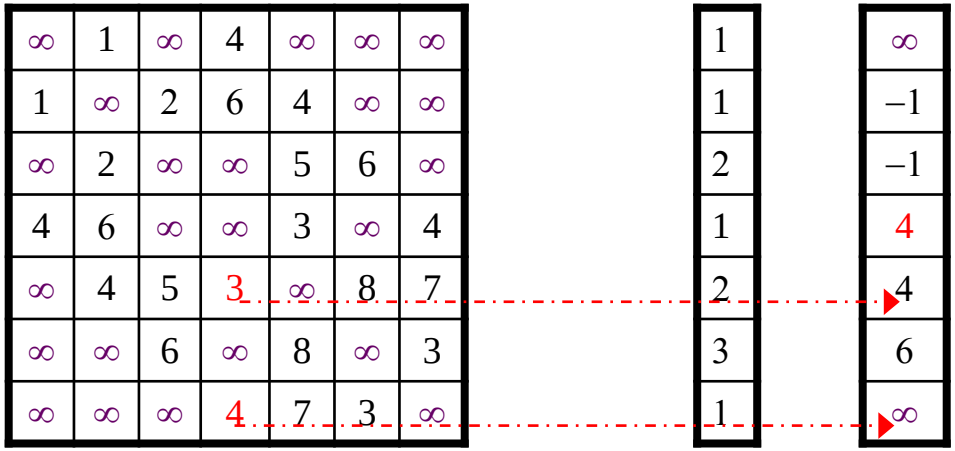
- Prim's Algorithm: after iteration 1
  - L = nearest = mindist =



S = {{1, 2}}

Prim's Algorithm

- Prim's Algorithm: after iteration 2
  - L = nearest = mindist =



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

Prim's Algorithm

- Prim's Algorithm: after iteration 3
  - L = nearest = mindist =

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

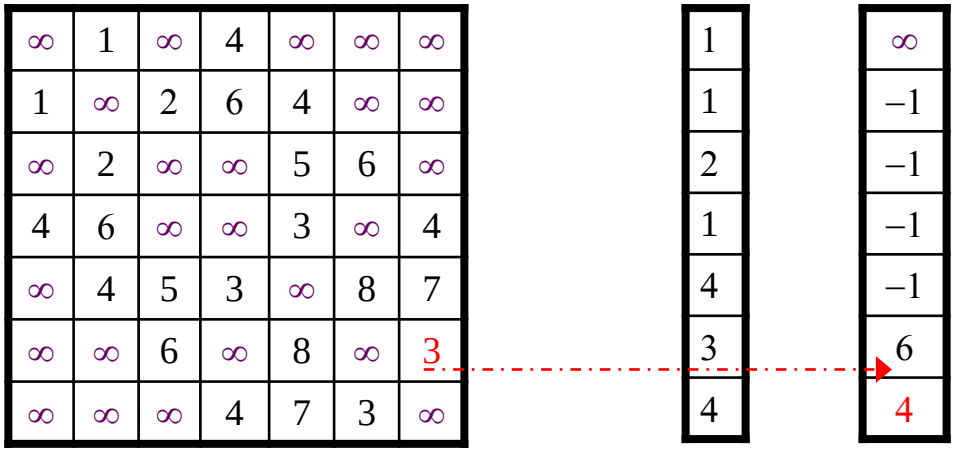
1
1
2
1
4
3
4

∞
–1
–1
–1
3
6
4

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

Prim's Algorithm

- Prim's Algorithm: after iteration 4
  - L = nearest = mindist =



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

Prim's Algorithm

- Prim's Algorithm: after iteration 5
  - L = nearest = mindist =

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

1
1
2
1
4
7
4

∞
–1
–1
–1
–1
3
–1

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

Prim's Algorithm

- Prim's Algorithm: after iteration 6
  - L = nearest = mindist =

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

1
1
2
1
4
7
4

∞
–1
–1
–1
–1
–1
–1

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

Knapsack Problem

Knapsack problem

- Knapsack problem is a combinatoric optimization problem. The algorithm makes use of greedy algorithm approach to find optimal object among a set of n objects in polynomial time, which otherwise requires $n!$ running complexity (NP problem) if all n possible permutation of the objects is tried (brute force.)

Knapsack Problem

Example 4: 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, that is, we can pack a fraction x_i of object i where $0 \leq x_i \leq 1$.
- **Note:** If we are not allowed to break objects this becomes a much harder problem.

Knapsack 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

Knapsack Problem

- 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		

Knapsack Problem

- 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

Knapsack Problem

- 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**

Knapsack 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 2: pick the lightest object

Knapsack Problem

- 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				

Knapsack Problem

- 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			

Knapsack Problem

- 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		

Knapsack Problem

- 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	

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

Knapsack Problem

- 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

Knapsack Problem

- 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		

Knapsack Problem

- 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**		

Knapsack Problem

- 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**		

Knapsack Problem

- 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

Knapsack Problem

- 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

Summary

- Greedy Algorithm
  - Making Changes with fewest coins
  - Shortest Distance
    - Dijkstra's Algorithm
  - Minimum Spanning Tree
    - Kruskal's Algorithm
    - Prim's Algorithm
  - Knapsack Problem