Curtin
University of Technology

Design and Analysis of Algorithms

Lecture 6

Greedy Algorithms

Curtin University of Technology

Topics

- Greedy principle
- Activity Selection problem
- 0/1 Knapsack problem
- Minimum-Cost Spanning Trees
- Single-source Shortest Path

Curtin
University of Technology

The Greedy Method - Principle

The problem:

We want to find a **feasible solution** that either **maximizes** or **minimizes** a given objective solution.

- A solution is called an **optimal solution** if it finds a **minimum** or a **maximum**
  - It is easy to determine a **feasible solution** but not necessarily an optimal solution.
- The greedy method solves this problem in **stages**
  - At each stage, a decision is made considering inputs in an order determined by the **selection procedure** which may be based on an **optimization** measure.

Curtin
University of Technology

The Greedy Method - Principle

- The greedy algorithm always makes the **choice** that looks **best** at the moment.
  - For each decision point in the greedy algorithm, the choice that seems best at the moment is chosen
  - It makes a **local optimal** choice that may lead to a **global optimal** choice.
- Greedy algorithms **do not** always give optimal solutions
  - However for many problems they do.

Curtin University of Technology

Steps in greedy algorithm design

1. Cast the optimization problem as one in which we make a choice and is left with one **sub-problem** to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that greedy choice is always safe.
3. Demonstrate that, having made the greedy choice, what remains is a sub-problem with the property that if we combine an optimal solution to the sub-problem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Curtin
University of Technology

Greedy method - Example

An activity-selection problem [Cormen et al.,]

Schedule a resource among several competing activities *S*.

- $S = \{1, 2, 3, \ldots, n\}$ is the set of n proposed activities
- The activities share a resource, which can be used by only one activity at a time, *e.g.*, a Tennis Court, a Lecture Hall, a processor *etc*.
- Each activity i has a **start time** s_i and a **finish time** f_i, where $s_i \leq f_i$
- When selected, an activity i takes place during time (s_i, f_i)
- Activities i and j are **compatible** if $s_i \geq f_j$ or $s_j \geq f_i$

The activity-selection problem: Select the maximum-size set of mutually compatible activities

Example:

S = { (0, 6), (5, 7), (1, 4), (12, 14), (3, 5), (3, 8), (5, 9), (8, 12), (6, 10), (8, 11), (0, 14)}

- Activities (0, 6) and (5, 7) are not compatible
- Activities (0, 6), (6, 10), and (12, 14) are compatible
- Activities (5, 7), (1, 4), (12, 14), and (8, 12) are compatible → maximum size

Curtin University of Technology

GREEDY_ACTIVITY_SELECTOR

- The input activities are **ordered in increasing finishing times**: $f_1 \leq f_2 \leq f_3 \ldots \leq f_n$
- The input can also be sorted in **decreasing starting time**: $s_n \geq s_{n\text{-}1} \geq \ldots \geq s_2 \geq s_1$
- Either input can be sorted in $O(n \lg n)$ time

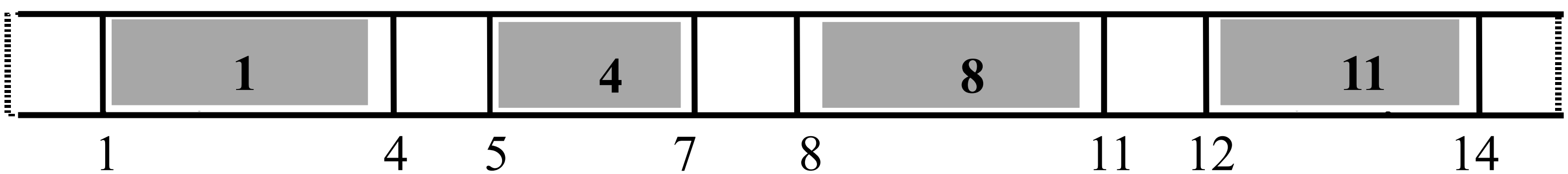
GREEDY_ACTIVITY_SELECTOR (*s*, *f*)

1. $n \leftarrow length\ [S]$
2. $A \leftarrow \{1\}$
3. $j \leftarrow 1;$
4. **for** $i \leftarrow 2$ to n
5. **do if** $s_i \geq f_j$
6. **then** $A \leftarrow A \cup \{i\}$
7. $j \leftarrow i$
8. **return** A

Example (input is ordered in increasing finishing time)

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	0	14
11	12	14

- Choose activity 1 as it has the least finish time: A = {1}
  - Activities 2 and 3 are not compatible; $s_2 < f_1$ & $s_3 < f_1$
- Choose activity 4: $s_4 > f_1$, add to the set A; A = {1, 4}
  - Activities 5, 6, 7 are incompatible
- Choose 8: A = {1, 4, 8}
  - Activities 9, 10 are incompatible
- Choose 11: A = {1, 4, 8, 11}
- The algorithm can schedule a set of n activities in $\Theta(n)$ time.
- Note A = {10} is **not** optimal.



The 0/1 Knapsack problem

Given input:

- n items each with a weight w_i and a profit p_i
- A maximum capacity c

Problem: Select a set of items that has the maximum total profit, but their total weight must not exceed c, i.e.,

Maximize the sum of profit: $P = \sum p_i$

without violating the weight constraint: $W = \sum w_i \leq c$

- 0/1 → for each item, either select the item, or not select it
  - Cannot fractionally select any item, e.g., ½ or ¼ item

Curtin University of Technology

Greedy Choice Example

Input: $n = 3$, $c = 30$; $w = \{20, 15, 15\}$; $p = \{40, 25, 25\}$

Greedy Solution:

Choice 1: Greedy by **weight**: $S = \{2, 3\}$, $P = 50$, $W = 30$

Select the item with the smallest weight first

Choice 2: Greedy by **profit**: $S = \{1\}$, $P = 40$, $W = 20$

Select the item with the largest profit first

Choice 3: Greedy by $\boldsymbol{p_i/w_i}$: $S = \{1\}$, $P = 40$, $W = 20$

- Order → 40/20, 25/15, 25/15

- Greedy approach does not always produce optimal solution for 0/1 Knapsack.
  - It is optimal for fractional knapsack problem

Curtin University of Technology

Greedy Graph Algorithms

We will show some other example problems in which the greedy approach can produce **optimal solutions**:

- Minimum cost spanning trees (MCST)
  - Cormen *et al* Ch 24 (23), p. 498 (561)
- Single-source Shortest paths
  - Cormen *et al* Ch 25 (24), p. 514 (580)

Typical Application

Consider a network of computers (WAN)

- Computers are connected by different mediums (cables, satellites, *etc.)*
- Links have different weights, e.g., costs, times

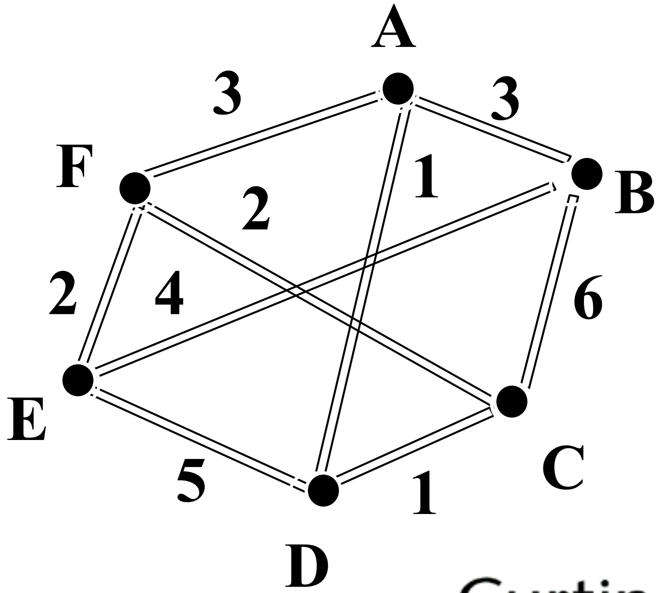
Problem: What paths should a broadcast message from my computer follow to minimize cost?

Cost = sum of individual transmission costs

Curtin University of Technology

MCST: Solution

- The WAN is modeled as a ***weighted*** graph $G(V, E)$
  - Each computer is a node in the set of nodes V
    - Vertices = computers / routers
  - Set E contains edges = connections
    - Each edge has a positive real weight
    - Weight = cost of transmission along that edge



MCST (cont.)

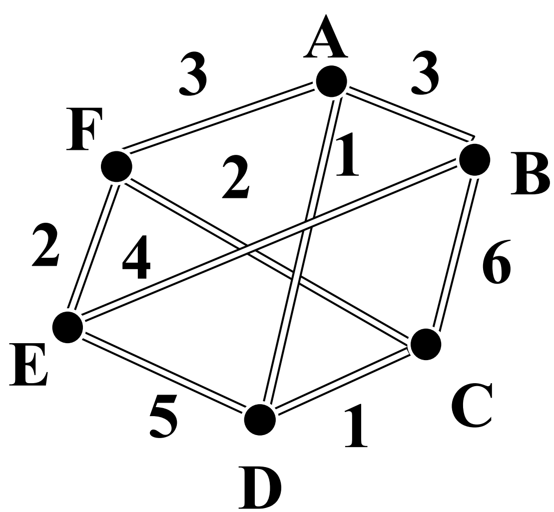
Definition:

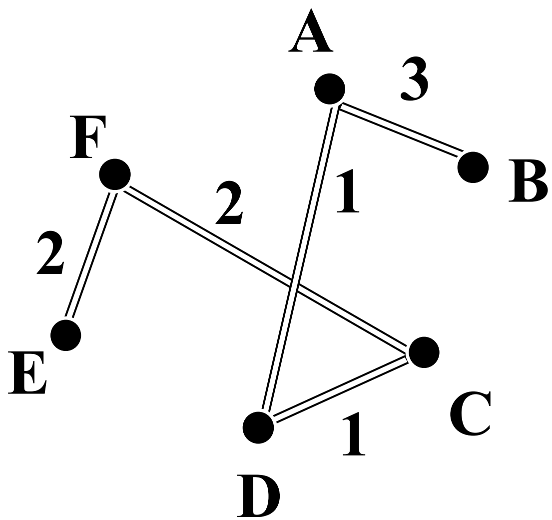
- A ***spanning tree*** is a tree (*i.e.*, connected graph with no cycles) that connects all *nodes* in V
- The ***cost*** of a spanning tree is the sum of edge weights
- A minimum cost spanning tree (**MCST)** is the tree amongst all spanning trees with the smallest weight (minimum cost)

Curtin University of Technology

MCST (cont.)

- MCST will have the smallest *cost*
- Cost is sum of *edge costs*
- Edge cost is transmission time or **$**
- So, sending a message along an MCST path will save cost





Growing MCST the greedy way

INPUT: Connected weighted graph G

OUTPUT: MCST T

Algorithm:

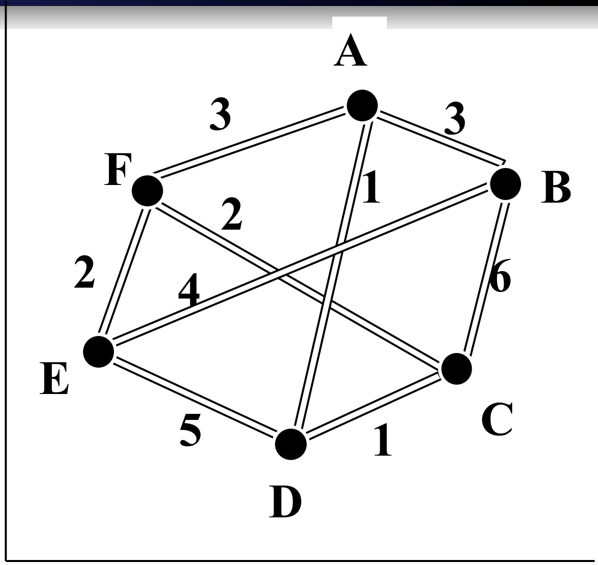
while more edges **do**

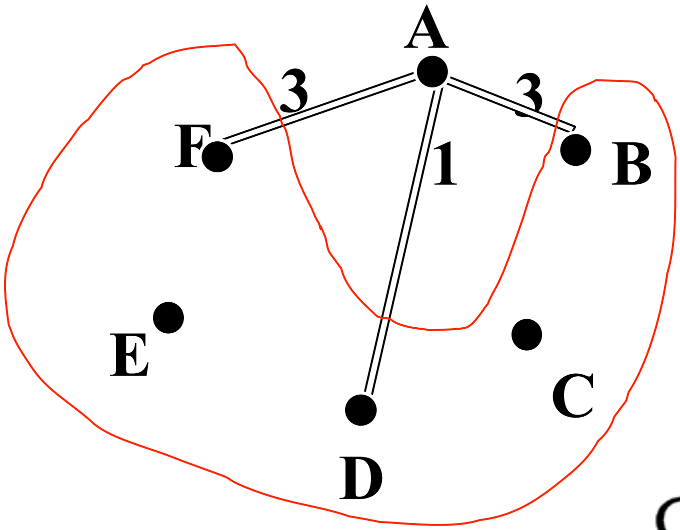
1. Take edge of least weight
2. If it won't make a cycle in T, add it to T

Generic example

Choose vertex A, V={B,C,D,E,F}

Candidate edges = (A,F) (A,D) (A,B)





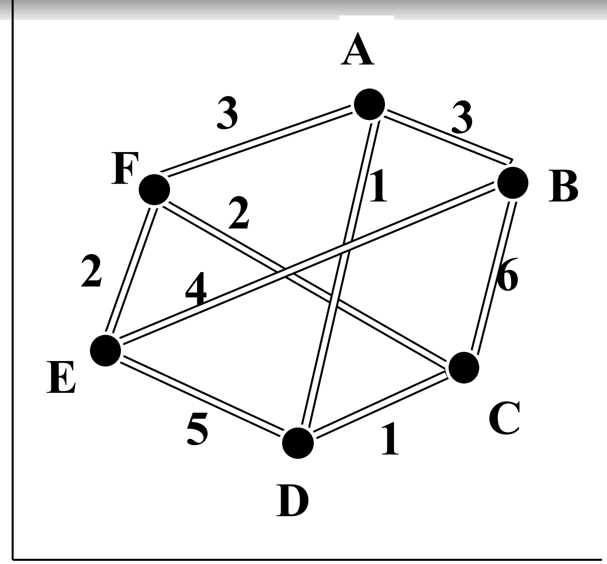
Generic example

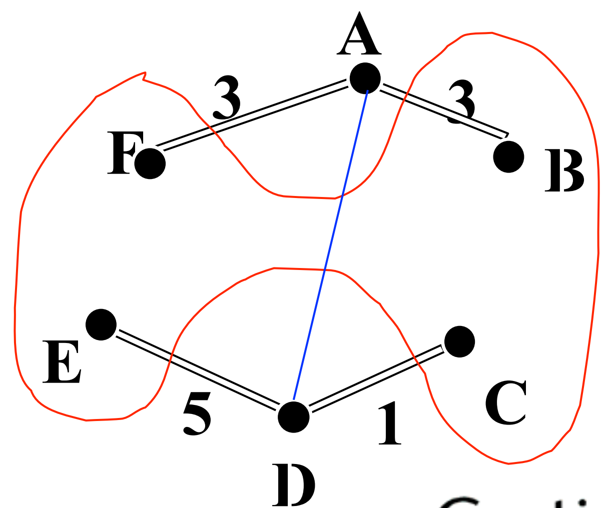
Choose vertex A, V={B,C,D,E,F}

Candidate edges = (A,F) (A,D) (A,B)

Choose edge (A,D) V={B,C,E,F}

Candidate edges = (A,F) (A,B) (D,C) (D,E)





Generic example

Choose vertex A, V={B,C,D,E,F}

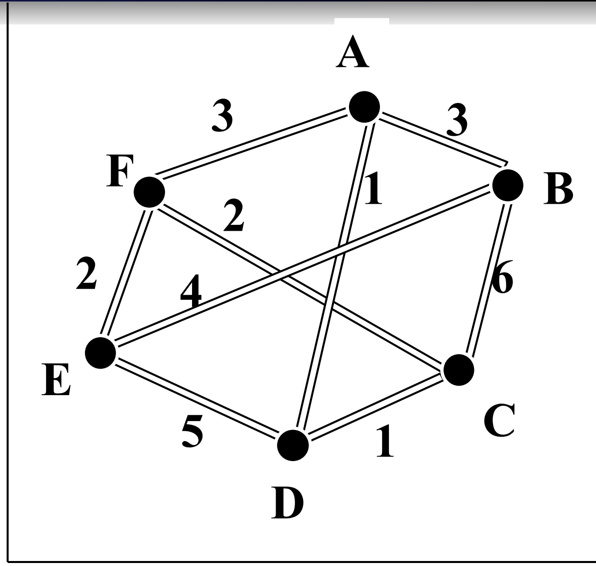
Candidate edges = (A,F) (A,D) (A,B)

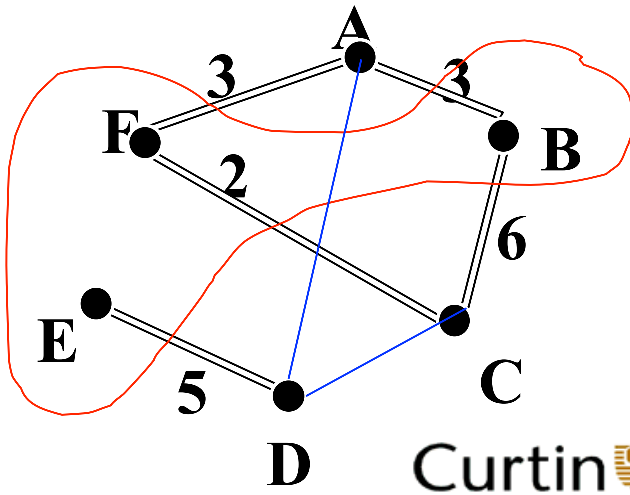
Choose edge (A,D) V={B,C,E,F}

Candidate edges = (A,F) (A,B) (D,C) (D,E)

Choose edge (D,C) V={B,E,F}

Candidate edges = (A,F) (A,B) (D,E) (C,F) (C,B)





Generic example

Choose vertex A, V={B,C,D,E,F}

Candidate edges = (A,F) (A,D) (A,B)

Choose edge (A,D) V={B,C,E,F}

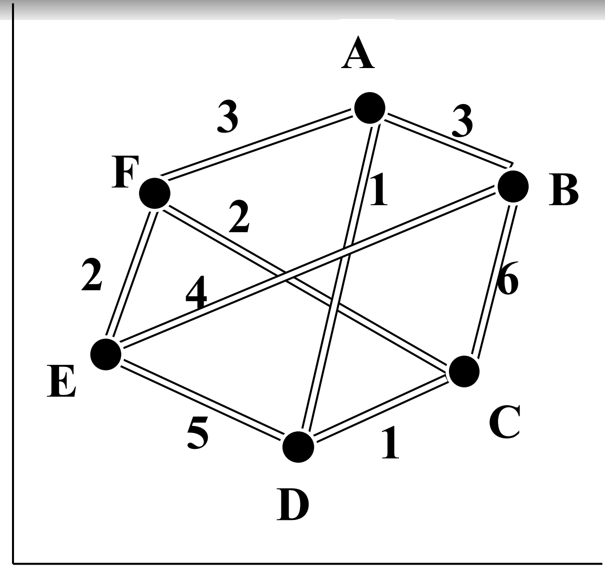
Candidate edges = (A,F) (A,B) (D,C) (D,E)

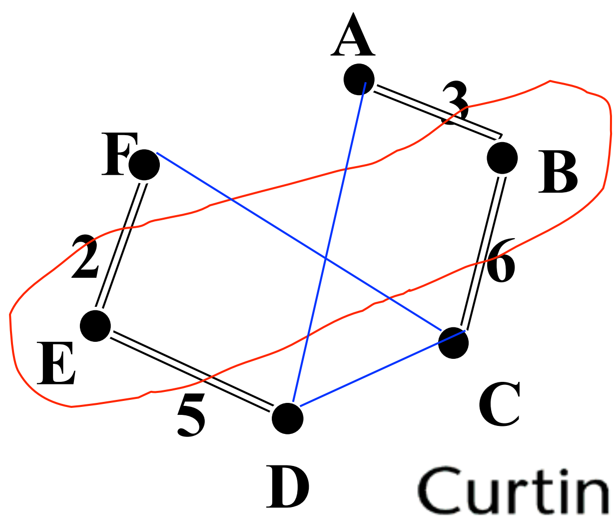
Choose edge (D,C) V={B,E,F}

Candidate edges = (A,F) (A,B) (D,E) (C,F) (C,B)

Choose edge (F,C) V={B,E}

Candidate edges = (A,B) (D,E) (C,B) (E,F)





Generic example

Choose vertex A, V={B,C,D,E,F}

Candidate edges = (A,F) (A,D) (A,B)

Choose edge (A,D) V={B,C,E,F}

Candidate edges = (A,F) (A,B) (D,C) (D,E)

Choose edge (D,C) V={B,E,F}

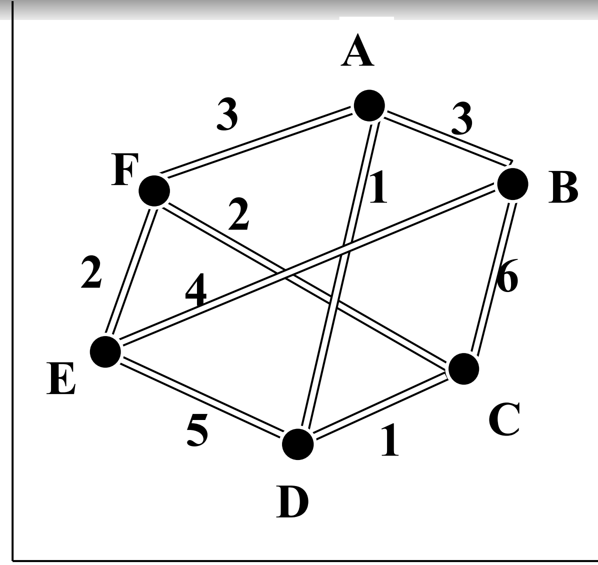
Candidate edges = (A,F) (A,B) (D,E) (C,F) (C,B)

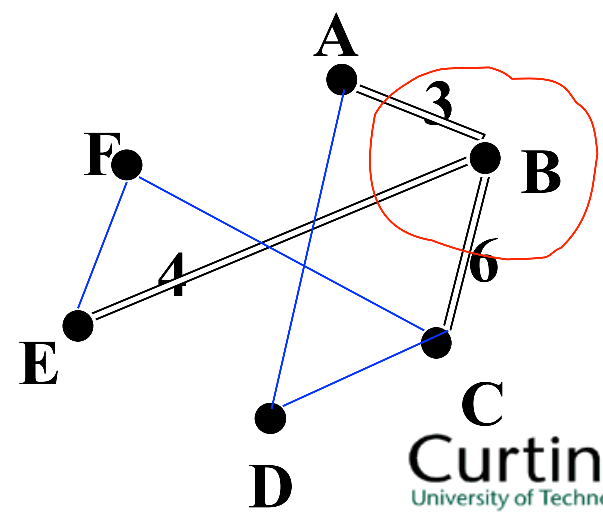
Choose edge (F,C) V={B,E}

Candidate edges = (A,B) (D,E) (C,B) (E,F)

Choose edge (E,F) V={B}

Candidate edges = (A,B) (C,B) (B,E)





Generic example

Choose vertex A, V={B,C,D,E,F}

Candidate edges = (A,F) (A,D) (A,B)

Choose edge (A,D) V={B,C,E,F}

Candidate edges = (A,F) (A,B) (D,C) (D,E)

Choose edge (D,C) V={B,E,F}

Candidate edges = (A,F) (A,B) (D,E) (C,F) (C,B)

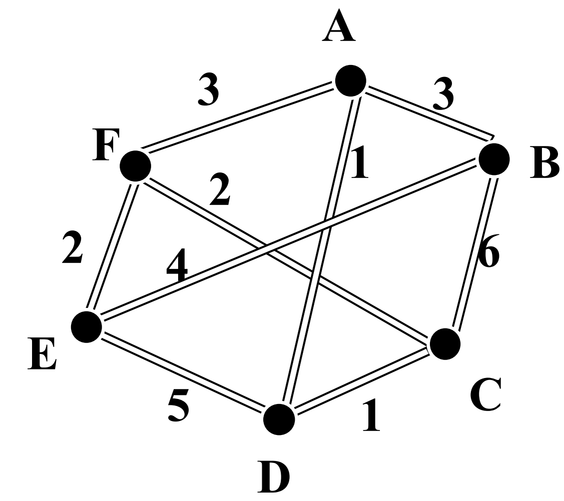
Choose edge (F,C) V={B,E}

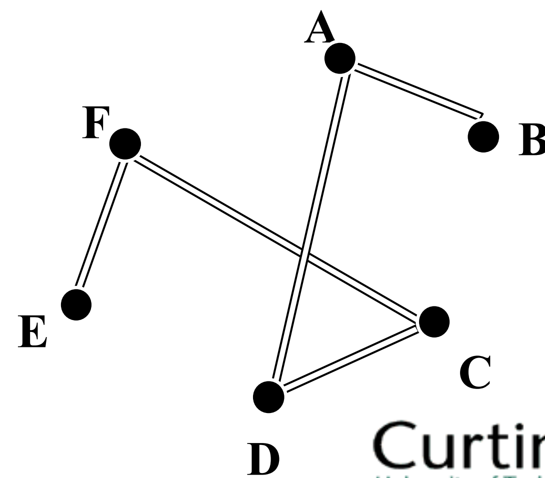
Candidate edges = (A,B) (D,E) (C,B) (E,F)

Choose edge (E,F) V={B}

Candidate edges = (A,B) (C,B) (B,E)

Choose edge (A,B) V={}





How to build MCST?

Two well-known algorithms:

- Kruskal algorithm
- Prim algorithm

Kruskal Algorithm

Input: An undirected graph $G(V, E)$ with a cost function c on the edges
Output: T the minimum cost spanning tree for G

1. $T = \{\}$, VS = {}
2. **for** each vertex $v \in V$ **do**
3. VS = VS ∪ {v} // Initially, each node is in a set W
4. Sort the edges of E in non-decreasing order of weight c
5. **for** each edge $(v, w) \in E$, taken in non-decreasing order by weight c, **do**
6. **if** v and w are in disjoint sets W1 and W2 in VS **then**
7. W1 = W1 ∪ W2
8. VS = VS – W2 // Remove W2 from VS
9. $T = T \cup (v, w)$
10. **return** T

- VS is a set of disjoint-sets of vertices
  - Initially each vertex is in a set by itself in VS
- Each set W in VS represents a connected set of vertices forming a spanning tree

Kruskal (cont.)

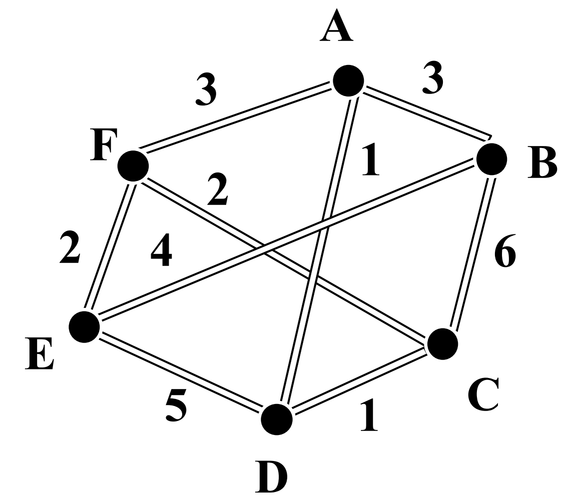
- Each vertex *i* is labeled as being in a set W_i
  - Initially each is in its own set
- Each set of vertices forms a MCST for the sub-graph it connects
- It is **safe** to join two vertices from different sets, i.e., it does not create a cycle
- The edge that joins the sets goes into *T*
- *T* is MCST because of greedy nature

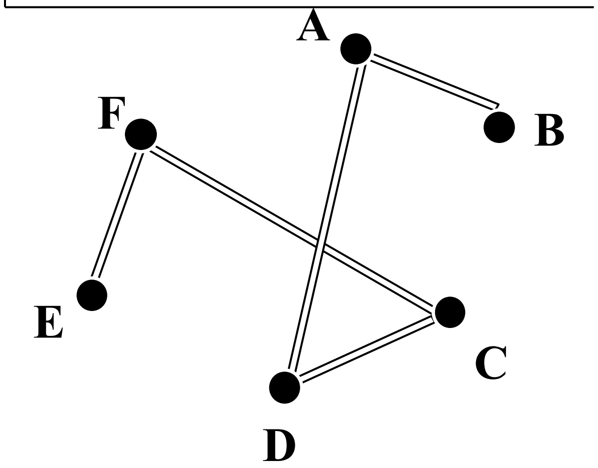
Kruskal example

Consider the graph shown earlier

VS = {{A},{B},{C},{D},{E},{F}}

1	(A,D)	merge	{A,D} {B} {C} {E} {F}
1	(C,D)	merge	{A,C,D} {B} {E} {F}
2	(C,F)	merge	{A,C,D,F} {B} {E}
2	(E,F)	merge	{A,C,D,E,F} {B}
3	(A,F)	reject	{A,C,D,E,F} {B}
3	(A,B)	merge	{A,C,D,E,F,B}
4	(B,E)	reject	{A,C,D,E,F,B}
5	(D,E)	reject	{A,C,D,E,F,B}
6	(B,C)	reject	{A,C,D,E,F,B}





Analysis of Kruskal

Input: An undirected graph $G(V,E)$ with a cost function c on the edges

Output: T the minimum cost spanning tree for G

$T \leftarrow \{\}$; VS $\leftarrow \{\}$

for each vertex $v \in V[G]$ **do**

  VS $\leftarrow$ VS $\cup \{v\}$

Sort the edges of E in non-decreasing order of weight c **O(E log E)**

for each edge $(v, w) \in E$, taken in nondecreasing order by weight c **do**

  if v and w are in disjoint sets W1 and W2 in VS **then**

    W1 $\leftarrow$ W1 $\cup$ W2   **O(???)**

    VS $\leftarrow$ VS – W2   **O(1)**

    $T \leftarrow T \cup (v, w)$ **O(1)**

return T **O(1)**

E times

- VS is a set of disjoint-sets of vertices
  - Initially each vertex is in a set by itself in VS
- Each set W in VS represents a connected set of vertices forming a spanning tree

Disjoint sets

Cormen et al, Ch 22 (21), p. 440 (498):

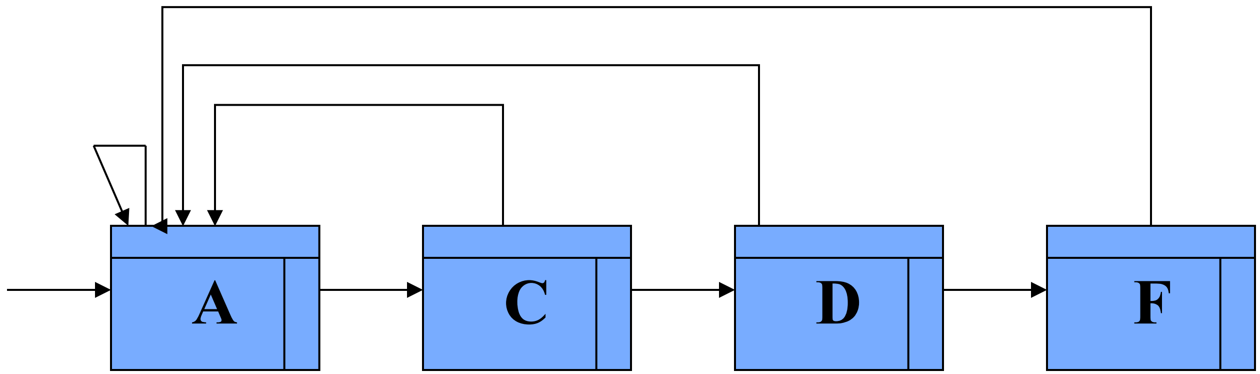
- A disjoint set has each member pointing to a *representative* in the set

<u>Three operators on disjoint set:</u>

- MAKE-SET(v): create a set that contains only node v
- FIND-SET(v): find the representative of node v
- UNION(u, v): merge set with representative u and another set with representative v

How to implement disjoint set?

Linked list disjoint sets



MAKE-SET(v) – $O(1)$

FIND-SET(v) – $O(1)$

UNION(u, v) – $O(min(|u|, |v|))$

In Kruskal's case, UNION is $O(V)$, so…

Curtin University of Technology

Analysis of Kruskal's

Input: An undirected graph $G(V, E)$ with a cost function c on the edges
Output: T the minimum cost spanning tree for G

1. $T \leftarrow \{\}$;		
2. **for** each vertex $v \in V[G]$ **do**	$O(V)$	
3. MAKE-SET(v)	$O(1)$	
4. Sort the edges of E in non-decreasing order of weight c		$O(E \log E)$
5. **for** each edge $(v, w) \in E$, taken in non-decreasing order by weight c **do**		
6. **if** FIND-SET(v) $\neq$ FIND-SET(w) **then**	$O(1)$	E times
7. $T \leftarrow T \cup (v, w)$	$O(1)$	E times
8. UNION (v, w)	$O(V)$	E times
9. **return** T	$O(1)$	

Total = $O(E \log E + EV)$

Curtin University of Technology

Disjoint-set linked-list

Theorem 21.1 (textbook)

Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

Used to compute the complexity of Step 2, and steps 5-8.

In Kruskal: $m = |V| + |E|$ and $n = O(|V|)$; so

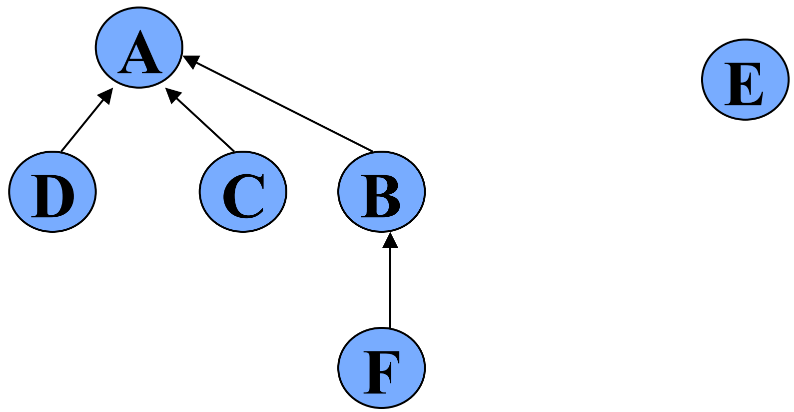
Steps 2, 5-8 + step 4: $O((E + V) + V \lg V + E \lg E) = O(E \lg E)$;

Observe: $|E| < |V|^2 \rightarrow \lg E = O(\lg V)$

So, Kruskal complexity: $O(E \lg V)$

Curtin University of Technology

Forest of disjoint sets



MAKE-SET(v) – $O(1)$
FIND-SET(v) – O(depth of tree)
UNION(u, v) – $O(1)$

Disjoint-set forests

Theorem 21.13 (textbook)

A sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, can be performed on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\ \alpha(n))$.

Used to compute the complexity of Step 2, and steps 5-8

For connected G: $|E| \geq |V|$ -1; $m = O(V + E)$, $n = O(V)$;

So Steps 2, 5-8: $O((V + E)\ \alpha(V)) = O(E\ \alpha(V))$

$\alpha(V) = O(\lg V) = O(\lg E)$; so we obtain $O(E \lg E)$

Kruskal: Steps 3, 5-8, and step 4: $O(E \lg E)$

Observe: $|E| < |V|^2 \rightarrow \lg E = O(\lg V)$

So, Kruskal complexity: $O(E \lg V)$

Curtin University of Technology

Prim algorithm

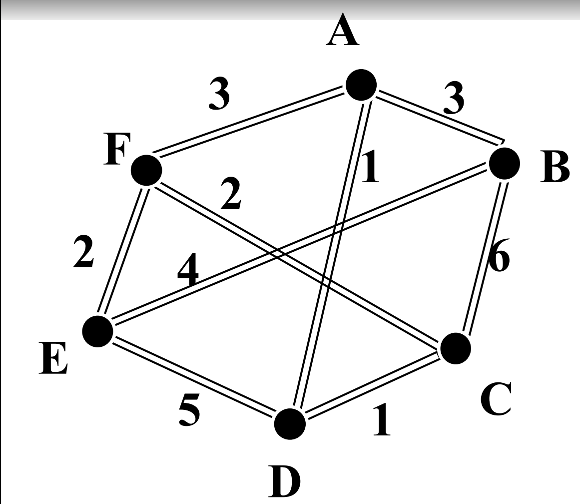
- Unlike Kruskal, Prim grows a single tree greedily, rather than joining sub-trees
- Keep a priority queue (sorted by weight) of *candidate* edges
- A candidate joins a vertex not already in the tree to the tree
- Choose the smallest weight candidate edge at each step
- The key to implementing Prim efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in the tree

Prim - example

- Q is a priority queue sorted by key with elements of the format (v , key[v], parent[v])

  Q: (A,∞,_) , (B,∞,_) , (C,∞,_) , (D,∞,_) , (E,∞,_) , (F,∞,_)

- key[v] is the minimum weight of any edge connecting vertex v to a vertex in the tree
  - Initially all keys = ∞



Prim – example (cont.)

Q: (A,∞,_),(B,∞,_),(C,∞,_),(D,∞,_),(E,∞,_),(F,∞,_)

Choose A, update keys and parents of F, D, B

Q: (D,1,A) , (B,3,A) , (F,3,A) , (C,∞,_) , (E,∞,_)

Choose D, update keys and parents of E, C

Q: (C,1,D) , (B,3,A) , (F,3,A), (E,5,D)

Choose C, update keys and parents of D, F, B

Q: (F,2,C), (B,3,A) , (E,5,D)

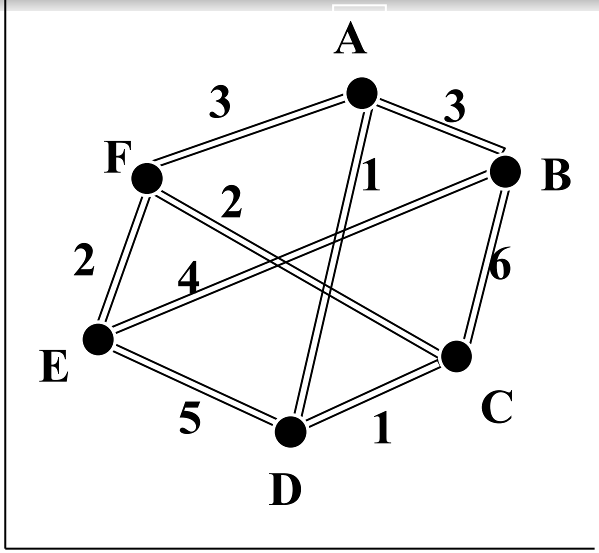
Choose F, update keys and parents of (A), (C), E

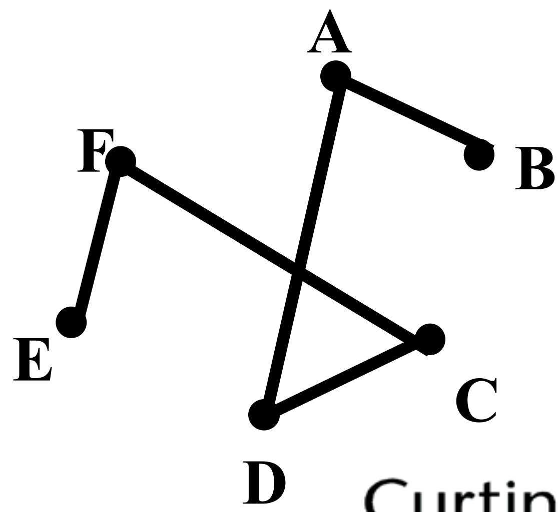
Q: (E,2,F), (B,3,A)

Choose E, update keys and parents of (F), B, (D)

Q: (B,3,A)

Choose B





Prim's algorithm

1. Insert all $v \in V$ into a priority queue Q, each with key of *infinity*. key[v] is the weight of the edge connecting v to the MCST
2. **while** Q is not empty **do**
3. Take out u, the min key vertex, from Q
4. **for** each edge (u, v) **do**
5. **if** weight of (u, v) < key[v] **then**
6. Record u as parent of v
7. Decrease key to weight of (u, v)
8. The final tree is the set of edges $\{ (u, \text{parent}[u]) \}$

Analysis of Prim

1.	Insert all *v* in *V* into a priority queue *Q*, each with weight of *infinity*.	***O(V)***
2.	**while** *Q* is not empty **do**	***V*** **times**
3.	Take out *u*, the min key vertex, from *Q*	***O*(log *V*)**
4.	**for** each edge (*u*,*v*) **do**	***O(E)*** **times in total**
5.	**if** (*v* in *Q*) and weight of (*u*,*v*) < key[*v*] **then**	
6.	Record *u* as parent of *v*	***O*(1)**
7.	Decrease key to weight	***O*(log *V*)**
8.	The final tree is the set of edges { (*u*, parent[*u*]) }	

Total $O(V + V \log V + E \log V) = O(E \log V)$

Curtin University of Technology

Prim's Algorithm (from textbook)

```
MST-PRIM(G, w, r)    // equivalent to the previously discussed algorithm
    for each u ∈ V[G]
        do key[u] ← ∞
           π[u] ← NIL   // π[u] means parent of vertex u
    key[r] ← 0
    Q ← V[G]                        // BUILD-MIN-HEAP
    while Q ≠ ϕ
        do u ← EXTRACT-MIN (Q)
           for each v ∈ Adj[u]
               do if v ∈ Q and w(u,v) < key[v]
                     then π[v] ← u
                          key[v] ← w(u,v)        // DECREASE-KEY
```

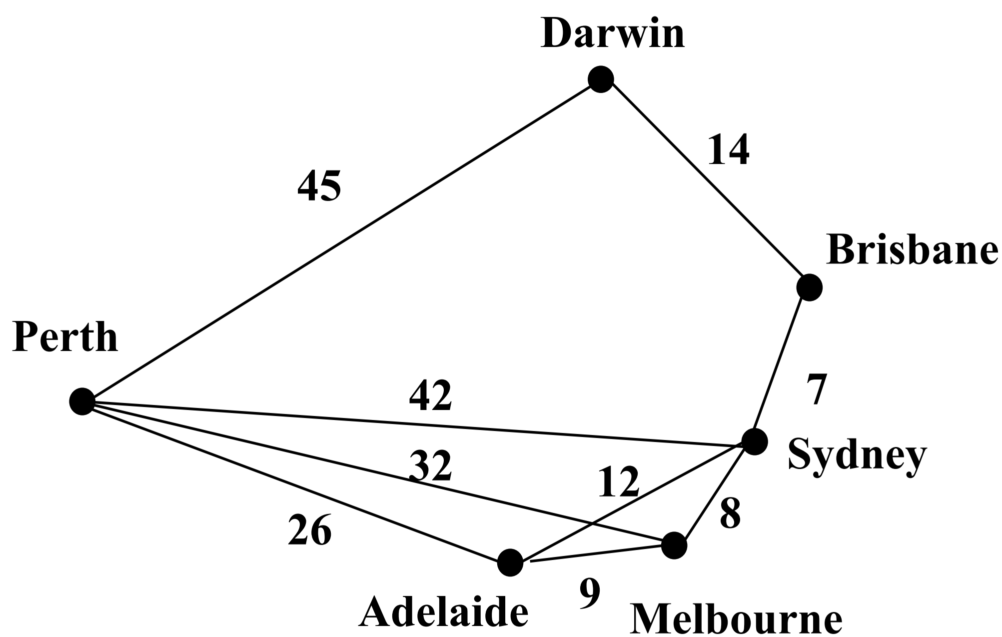
Improved Prim

- Replace the binary heap with a Fibonacci heap
- $O(E + V \log V)$

Shortest Paths

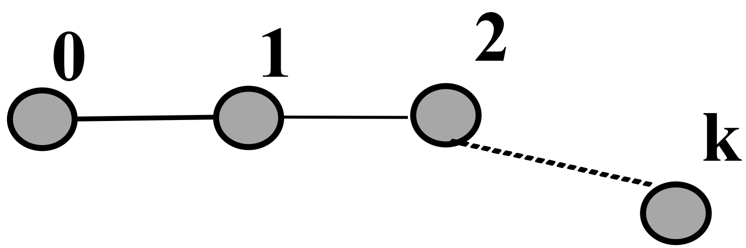
Typical Application:

- A motorist wishes to find the shortest possible route from Perth to Brisbane.
- Given the map of Australia on which the distance between each pair of cities is marked, how can we determine the shortest route?



Shortest paths (cont.)

- In a shortest-path problem, we are given a weighted, directed graph G (V, E), with weights assigned to each edge in the graph.
- The weight of the path $p = (v_0, v_1, v_2, \ldots, v_k)$ is the sum of the weights of its constituent edges:
  - $v_0 \rightarrow v_1 \rightarrow v_2 \ \ldots \rightarrow v_{k\text{-}1} \rightarrow v_k$



- The shortest-path from u to v is given by

$$d(u, v) = \min \{\text{weight } (p)\text{: if there is a path from } u \text{ to } v \}$$
$$= \infty \text{ otherwise}$$

The single-source shortest paths problem

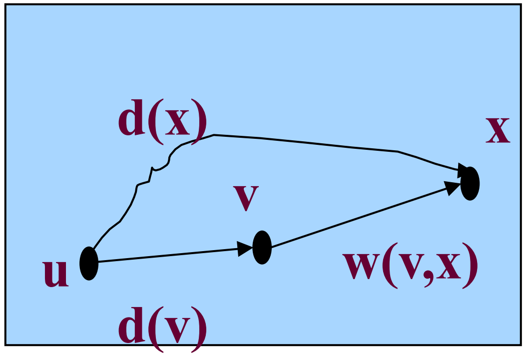
Problem: Given $G\ (V, E)$, find the shortest path from a given vertex $u \in V$ to every vertex $v \in V\ (u \neq v)$

- For each vertex $v \in V$ in the weighted directed graph, $d[v]$ represents the distance from u to v

  Initially, $d[v] = 0$ when $u = v$

  $d[v] = \infty$ if (u, v) is not an edge

  $d[v] =$ weight of edge (u, v) if (u, v) exists



•

•**Dijkstra's Algorithm:** At each step of the algorithm, compute:

$$d[x] = \min \{d[x], d[v] + weight(v, x)\}, \text{ where } v, x \in V$$

- Dijkstra's algorithm is based on the **greedy principle** because at every step we pick the path of least cost.
- Dijkstra's algorithm assumes that all edge weights are non-negative

Curtin University of Technology

Dijkstra's Single-source shortest path

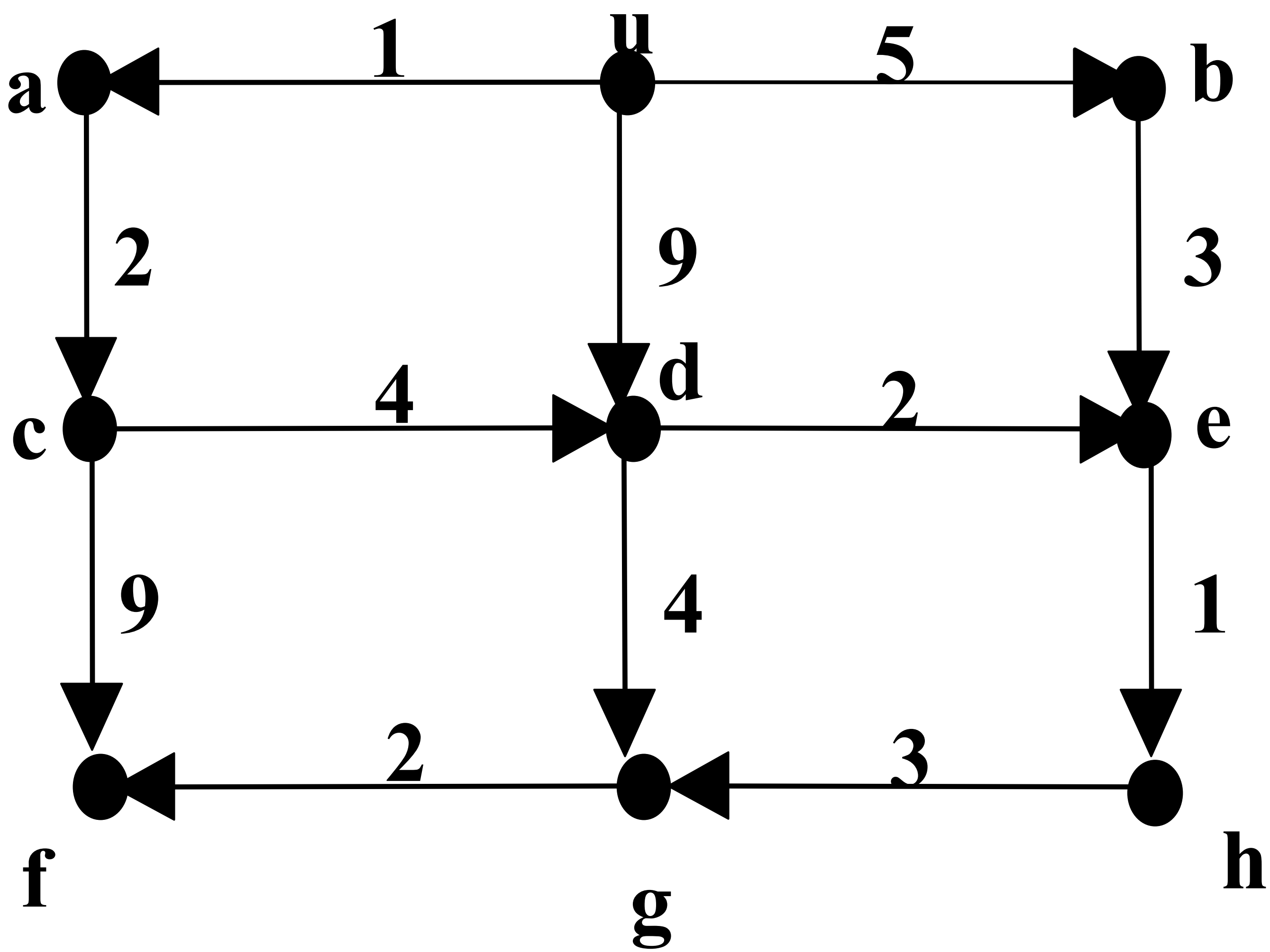
Single-source shortest path_G(V, E, u)
Input: $G = (V, E)$, the weighted directed graph and u the source vertex
Output: for each vertex, v, $d[v]$ is the length of the shortest path from u to v.

mark vertex u
$d[u] \leftarrow 0$
for each unmarked vertex $v \in V$ **do**
  if edge (u, v) exists **then** $d[v] \leftarrow weight\ (u, v)$
  else $d[v] \leftarrow \infty$
while there exists an unmarked vertex **do** // *V vertex*
  let v be an unmarked vertex such that $d[v]$ is minimal //
  mark vertex v
  for all edges (v, x) such that x is unmarked **do** // at most E updates
    if $d[x] > d[v] + weight[v, x]$ **then**
      $d[x] \leftarrow d[v] + weight[v, x]$ // DECREASE-KEY

Complexity of Dijkstra's Algorithm

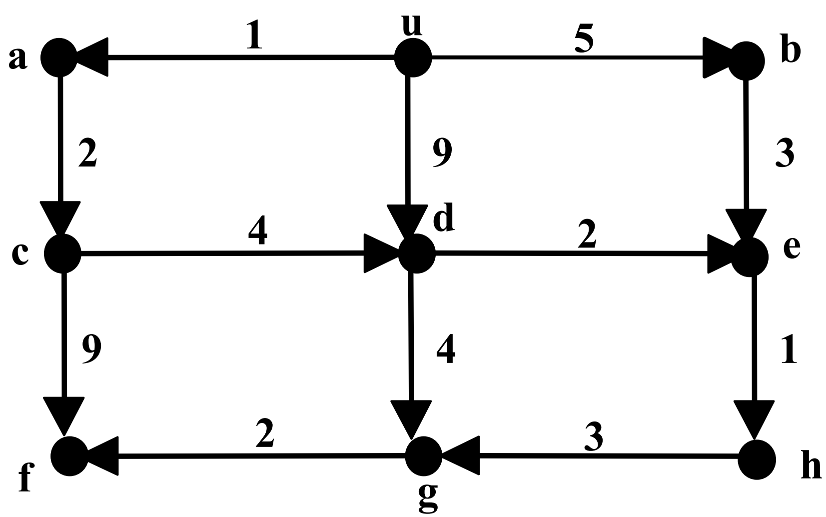
- **Steps 1 and 2:** Θ (1) time
- **Steps 3 to 5:** $O(|V|)$ time
- The vertices are arranged in a heap in order of their paths from *u*
- Updating the length of a path takes $O(\lg V)$ time. // Step 11
- There are $|V|$ iterations, and at most $|E|$ updates
- Therefore the algorithm takes $O((|E|+|V|)\lg|V|)$ time.

Dijkstra - Example



Dijkstra – Example (cont.)

Step #	Vertex to be marked	u	a	b	c	d	e	f	g	h	Unmarked vertices
0	u	0	1	5	∞	9	∞	∞	∞	∞	a,b,c,d,e,f,g,h
1	a	0	1	5	3	9	∞	∞	∞	∞	b,c,d,e,f,g,h
2	c	0	1	5	3	7	∞	12	∞	∞	b,d,e,f,g,h
3	b	0	1	5	3	7	8	12	∞	∞	d,e,f,g,h
4	d	0	1	5	3	7	8	12	11	∞	e,f,g,h
5	e	0	1	5	3	7	8	12	11	9	f,g,h
6	h	0	1	5	3	7	8	12	11	9	g,h
7	g	0	1	5	3	7	8	12	11	9	h
8	f	0	1	5	3	7	8	12	11	9	--------



The End