Chapter 6:
Graphs

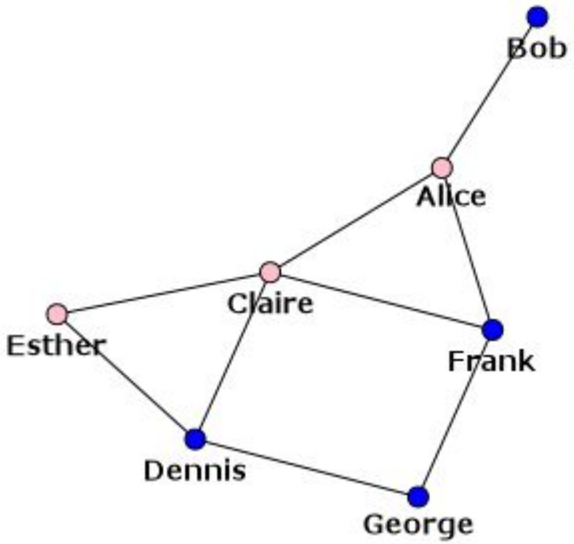
Contents

Graphs

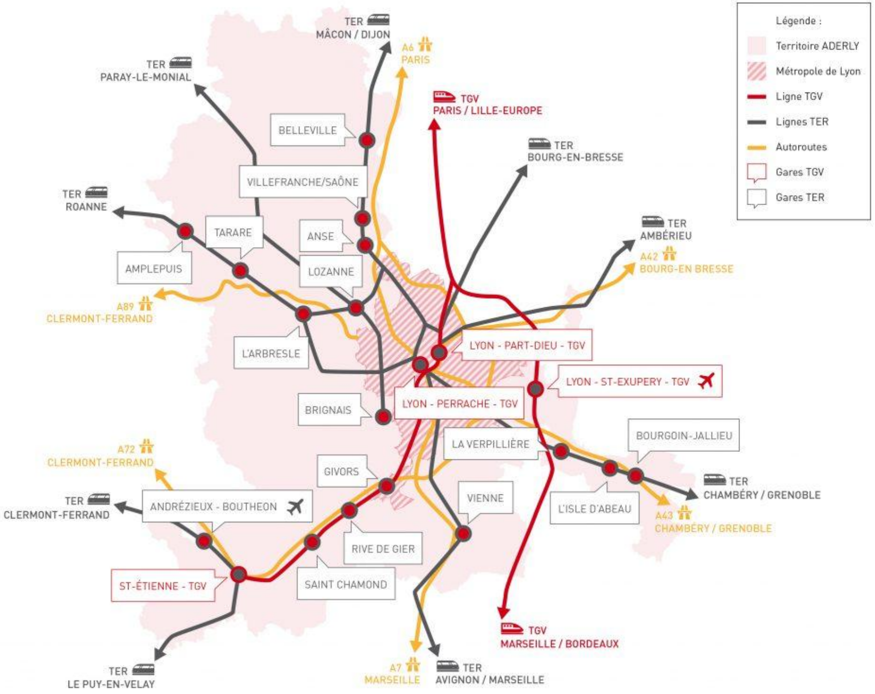
- Data structure used to model a large variety of data
- Examples:
  - Networks such as computer networks, road networks, bus routes, maps, social networks etc.
  - Representing relational data
  - Representing molecular structures
  - Resource allocation in computer systems
  - Probabilistic models etc.

Examples: Networks

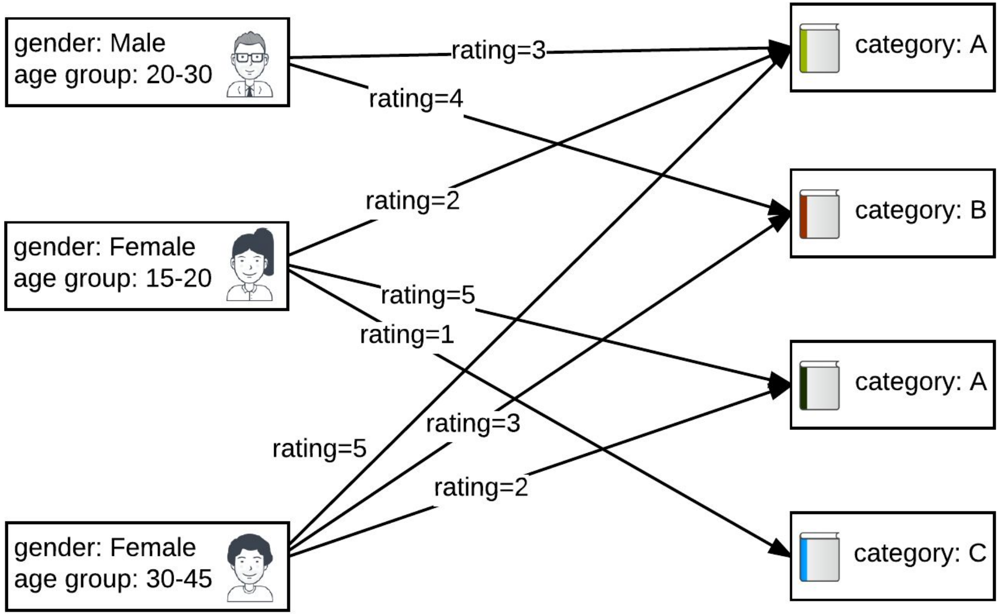
Social network



Road / transportation network



Examples: Representing relational data



User

user_id	gender	age_group
user_1	Male	20-30
user_2	Female	15-20
user_3	Female	30-45

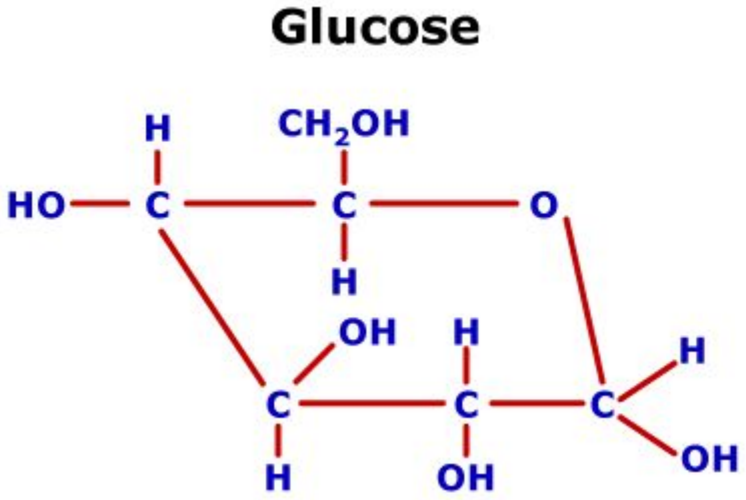
Book

book_id	category
book_1	A
book_2	B
book_3	A
book_4	C

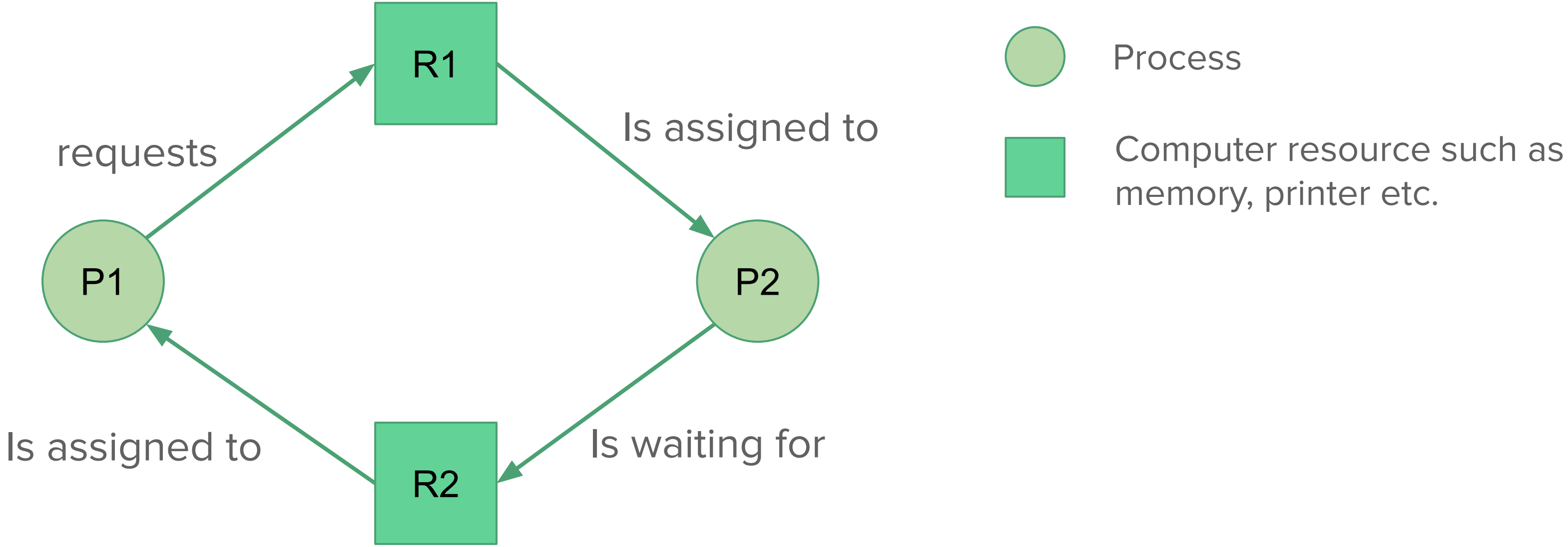
Rating

book_id	user_id	rating
book_1	user_1	3
book_1	user_2	2
book_1	user_3	4
...	...	...
book_4	user_2	1

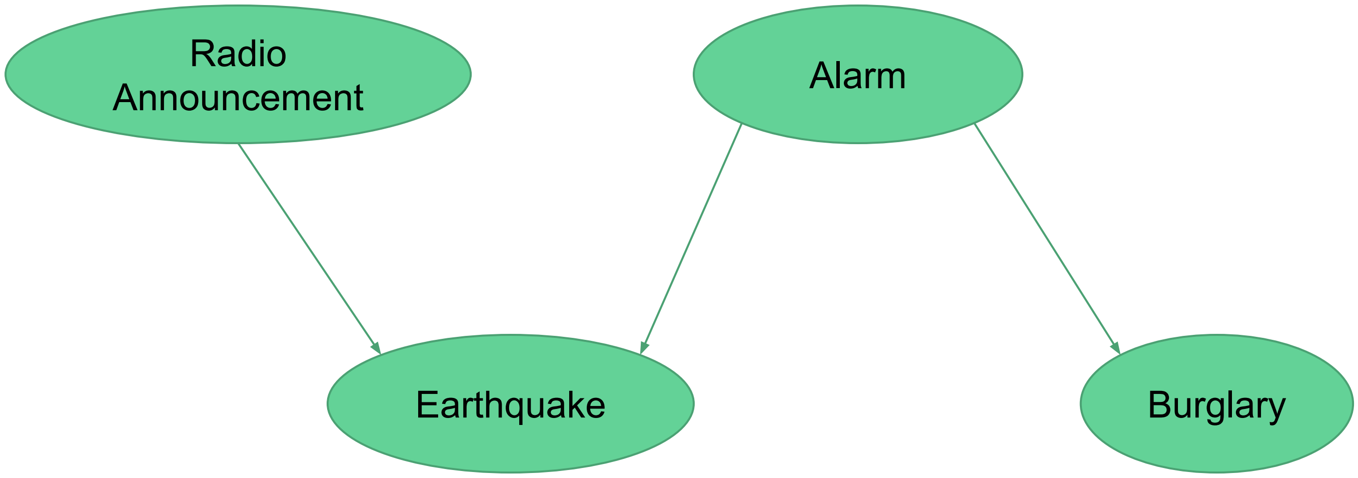
Examples: Representing molecular structure



Examples: Resource allocation

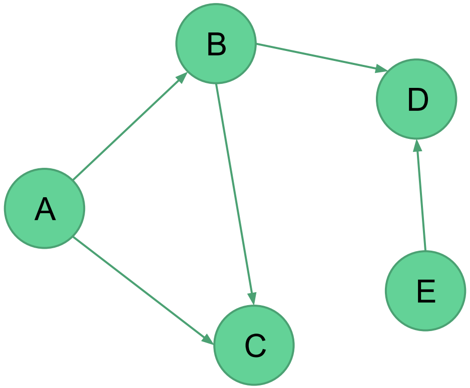


Examples: Probabilistic models



Graphs

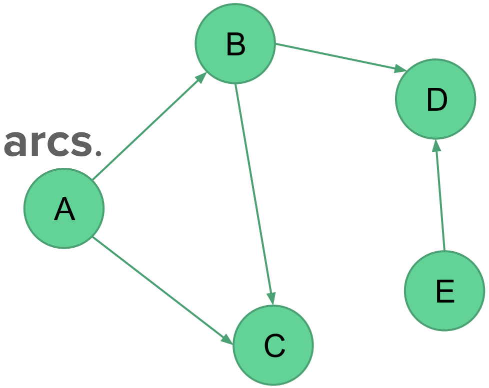
A graph is a collection of **nodes**, called **vertices**, and line segments, called **arcs** or **edges**, that connect pairs of nodes.



Graphs

A graph is a collection of **nodes**, called **vertices**, and line segments, called **arcs** or **edges**, that connect pairs of nodes.

Mathematically, a graph, G, consists of two sets, V and E.
V is a finite, nonempty set of **vertices**.
E is a set of pairs of vertices; these pairs are called **edges** or **arcs**.



G = (V, E) will represent a graph

V(G) or **G.V** will represent the set of vertices

E(G) or **G.E** will represent the set of edges

Graphs

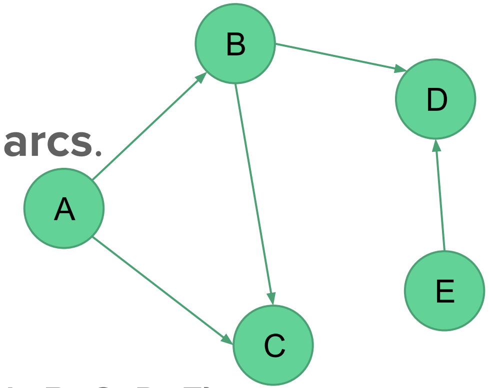
A graph is a collection of **nodes**, called **vertices**, and line segments, called **arcs** or **edges**, that connect pairs of nodes.

Mathematically, a graph, G, consists of two sets, V and E.
V is a finite, nonempty set of **vertices**.
E is a set of pairs of vertices; these pairs are called **edges** or **arcs**.

G = (V, E) will represent a graph

V(G) or **G.V** will represent the set of vertices

E(G) or **G.E** will represent the set of edges



V(G) = {A, B, C, D, E}
E(G) = {(A, B), (B, C), (B, D), (A, C), (E, D)}

Graphs

Iterating over the vertices:

For each vertex v $\in$ G.V do ...

For v $\in$ G.V do ...

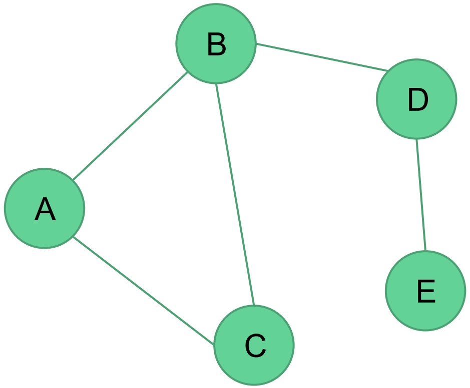
Iterating over the edges:

For each edge (u, v) $\in$ G.E do ...

For (u, v) $\in$ G.E do ...

Undirected graphs

- In an undirected graph, there is **no direction (arrow head)** on any of the **edges** i.e., the pair of vertices representing any edge is **unordered**
- The pairs (u, v) and (v, u) represent the same edge
- An undirected edge is written as {u, v}



V(G) = {A, B, C, D, E}

E(G) = {{A, B}, {B, C}, {B, D}, {A, C}, {E, D}}

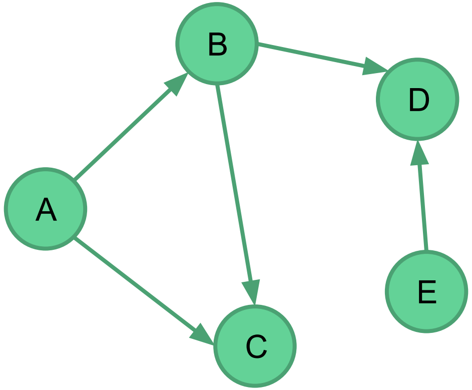
Or

V(G) = {A, B, C, D, E}

E(G) = {{B, A}, {C, B}, {B, D}, {A, C}, {E, D}}

Directed graph (Digraph)

- In a directed graph, each edge, also called an **arc**, **has a direction (arrow head)** to its **successor**
- Each edge is represented by a **directed pair** (u, v) or < u, v >; u is the tail and v is the head of the edge
- The flow along the arcs between two vertices can follow only the indicated direction



V(G) = {A, B, C, D, E}

E(G) = {<A, B>, <B, C>, <B, D>, <A, C>, <E, D>}

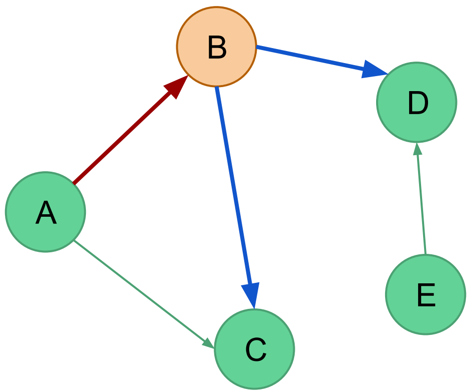
Graph terminologies

Outdegree of a vertex (in a digraph):

The number of arcs leaving the vertex

Indegree of a vertex (in a digraph):

The number of arcs entering the vertex



Outdegree of B, $d_{out}(B) = 2$

Indegree of B, $d_{in}(B) = 1$

Degree of B, $d(B) = 2 + 1 = 3$

Graph terminologies

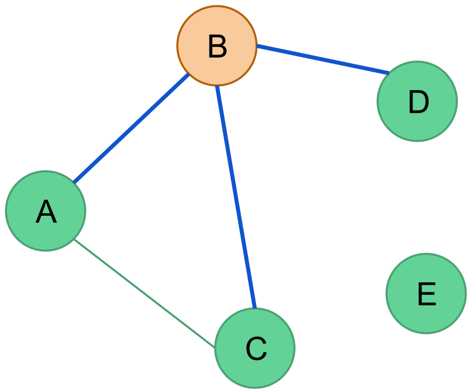
Degree of a vertex:

Total number of edges incident to the vertex

i.e., both incoming and outgoing arcs in a digraph

Isolated vertex:

A vertex whose degree is 0



Degree of B, d(B) = 3

Degree of E, d(E) = 0

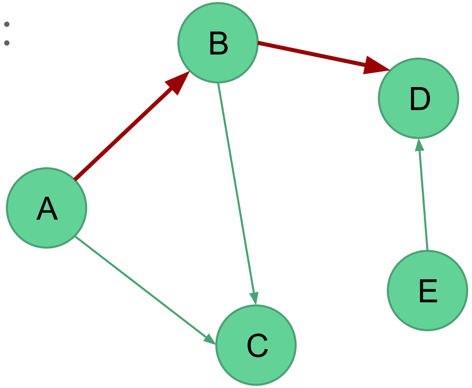
E is an isolated vertex

Graph terminologies

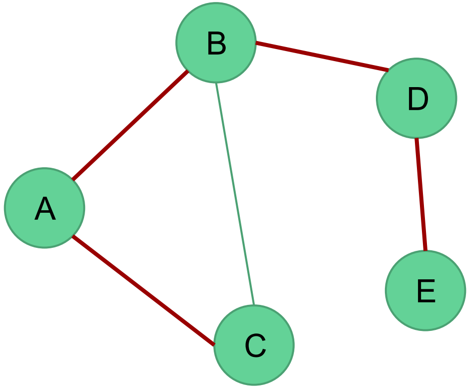
Path:

A sequence of vertices in which each vertex is **adjacent** to the next one

Examples:



ABD



CABDE

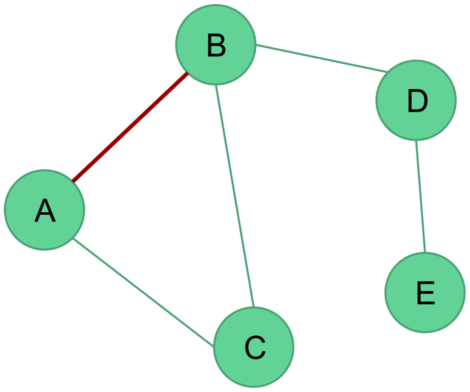
In an undirected graph, we can travel in any direction whereas in a digraph, we cannot

Graph terminologies

Adjacent vertices:

Two vertices in a graph are said to be adjacent vertices (or **neighbours**) if there is a path of length 1 connecting them

Example:



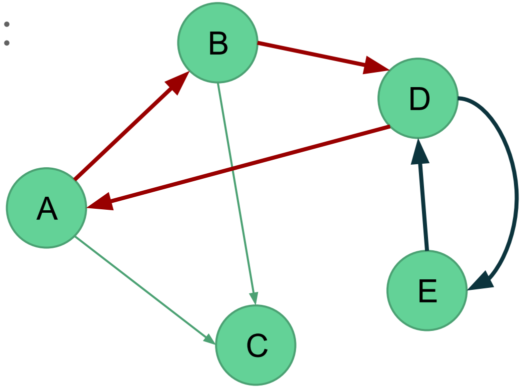
A and B are adjacent vertices because there is a path AB, whose length is 1
But C and E are not adjacent even though there is a path from C to E

Graph terminologies

Cycle:

A path consisting of at least three vertices that starts and ends with the same vertex

Examples:

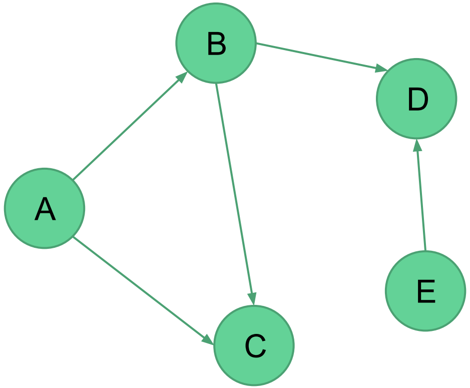


ABDA and DED are cycles

Graph terminologies

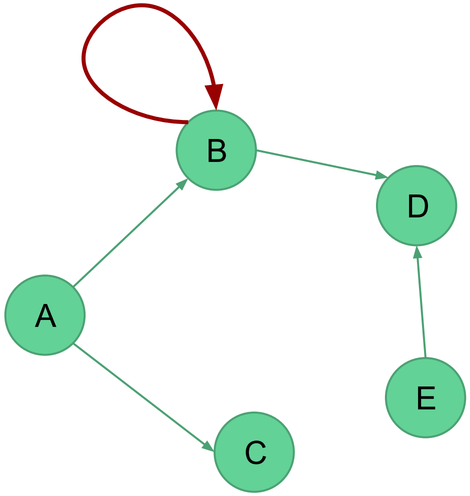
Acyclic graph:

A graph **without any cycle**



Loop:

A special case of a cycle in which a single arc begins and ends with the same vertex



Graph terminologies

Simple graph:

A simple graph has **no loop** or **multiple edges**

Balanced graph:

A graph is balanced if $d_{in}(v) = d_{out}(v)$ for all nodes

Exercise

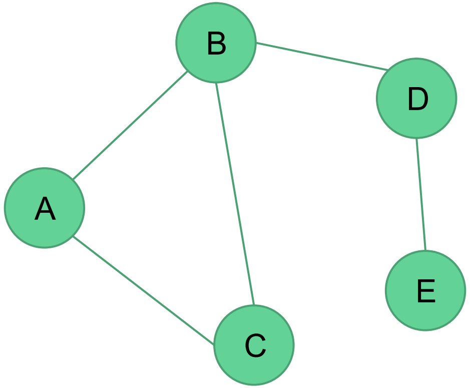
Types of graphs

- Directed graph
- Undirected graph
- Acyclic graph
- Connected graph
- Disjoint graph
- Complete graph
- Weighted graph
- Multigraph
- Null graph

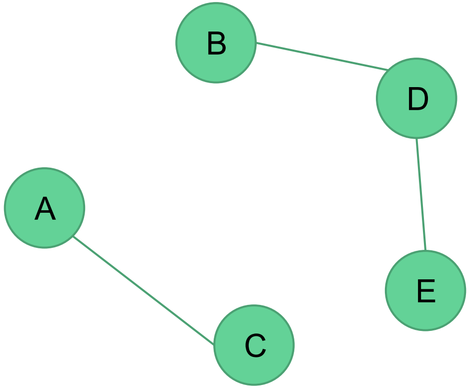
Connected graph

A graph is said to be **connected** if there is a path (ignoring direction) between every pair of vertex.

A graph is a **disjoint** graph if it is not connected.



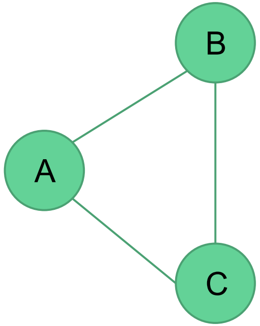
Connected graph

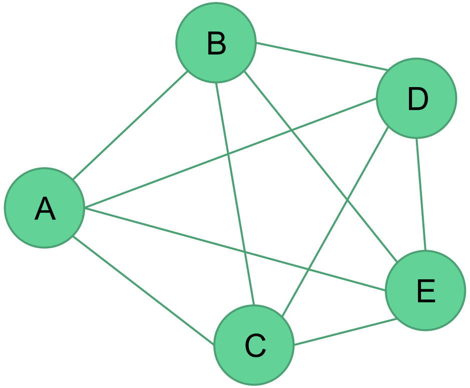


Disjoint graph

Complete graph

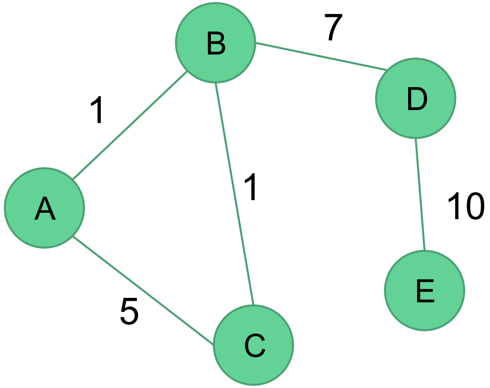
An undirected graph in which every pair of distinct vertices is connected by a unique edge





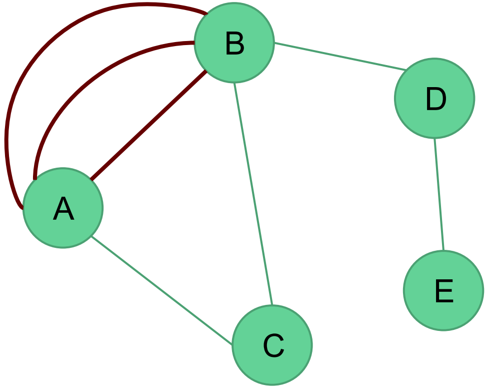
Weighted graph

- A graph where each edge has an assigned number
- Depending on the context in which such graphs are used, the number assigned to an edge is called its **weight, cost, distance, length** etc.
- A weighted graph is also called a network



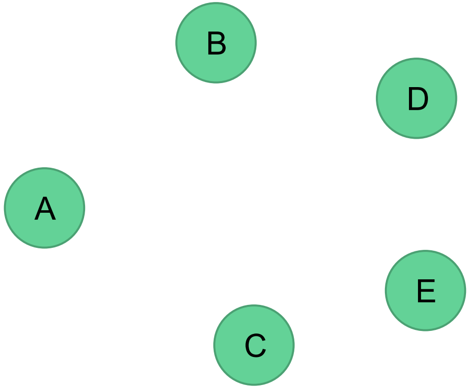
Multigraph

Graph with more than one edge between the same two vertices



Null graph

A graph containing only vertices and no edges

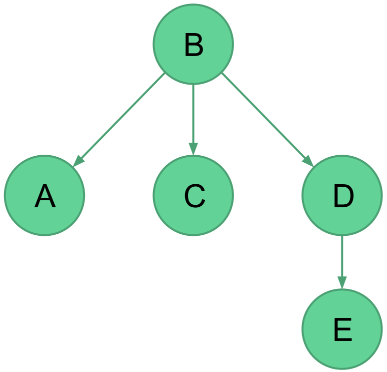


Graphs vs Trees

Trees are special cases of graphs

A tree is an **acyclic connected graph** with exactly one root node and each vertex has **only one predecessor**

In graphs, nodes do not have any clear parent-child relationship like in trees. Instead, nodes are called neighbours if they are connected by an edge



V(G) = {A, B, C, D, E}
E(G) = {<B, A>, <B, C>, <B, D>, <D, E>}

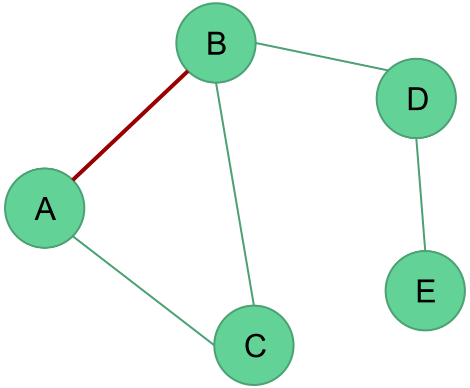
Graph representation

- There are variety of ways to represent graphs
- Two common ways:
  - Adjacency matrix
  - Incidence matrix
  - Adjacency list

Adjacency matrix

An adjacency matrix of a graph G = (V, E) is a binary |V| x |V| matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if there exists an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

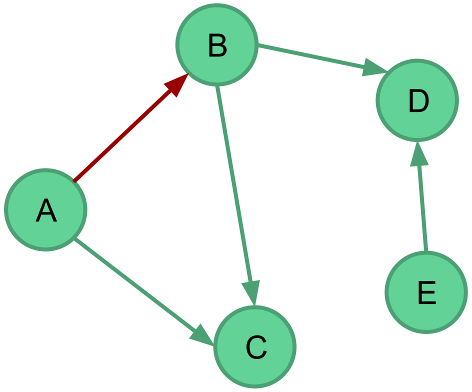


	A	B	C	D	E
A	0	**1**	1	0	0
B	**1**	0	1	1	0
C	1	1	0	0	0
D	0	1	0	0	1
E	0	0	0	1	0

Adjacency matrix

An adjacency matrix of a graph G = (V, E) is a binary |V| x |V| matrix such that

$$a_{ij} = \begin{cases} 1 & \text{if there exists an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$



	A	B	C	D	E
A	0	**1**	1	0	0
B	**0**	0	1	1	0
C	0	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

Exercise

Adjacency matrix

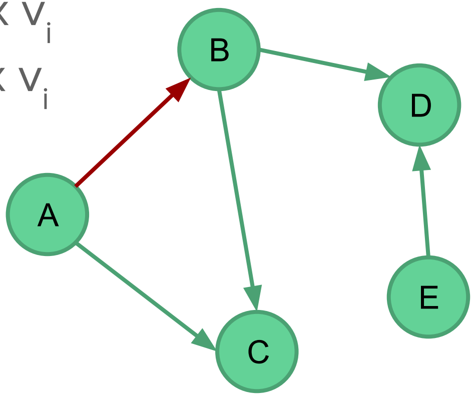
Limitations

- The size of the graph must be known in advance
- Only one edge can be stored between any two vertices
- In sparse graphs, most of the elements in the matrix will be 0, i.e. when $|E| \ll |V|^2$

Incidence matrix

An incidence matrix of a graph G = (V, E) is a |V| x |E| matrix such that

$$a_{ij} = \begin{cases} -1 & \text{if edge } e_j \text{ leaves vertex } v_i \\ 1 & \text{if edge } e_j \text{ enters vertex } v_i \\ 0 & \text{otherwise} \end{cases}$$



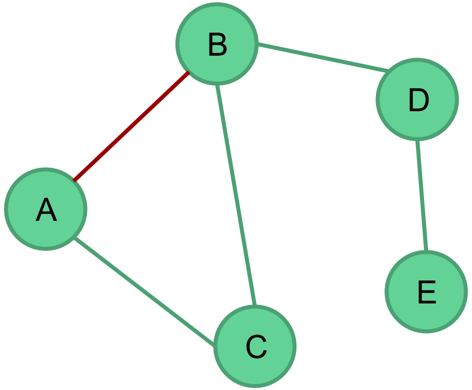
	AB	AC	BC	BD	ED
A	-1	-1	0	0	0
B	1	0	-1	-1	0
C	0	1	1	0	0
D	0	0	0	1	1
E	0	0	0	0	-1

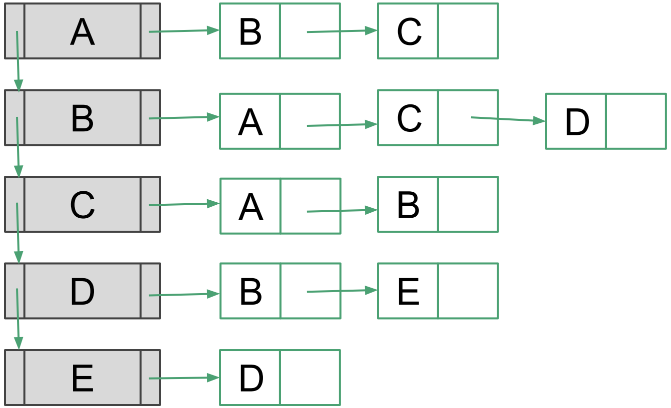
For an undirected graph,

a_{ij} = 1 if e_j is incident with v_i

Adjacency list

In the adjacency list representation, we use a table or a list to store the vertices and a two-dimensional linked list to store the edges

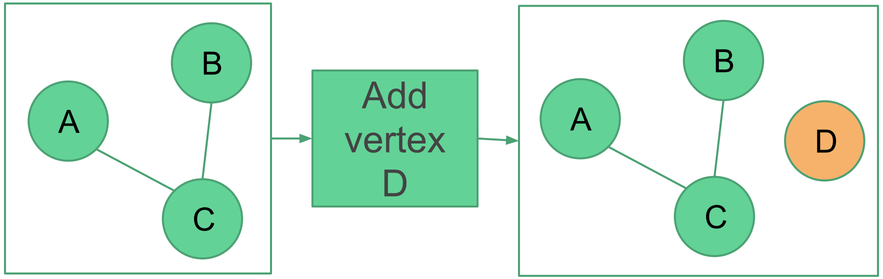




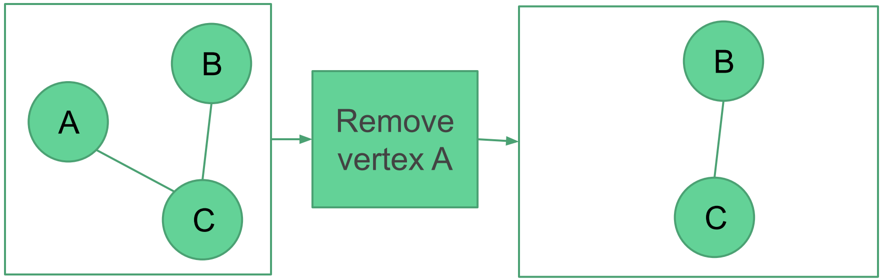
Exercise

Operations on graphs

1. **Add a vertex to a graph**
   When a vertex is added, the graph becomes disjoint as the new vertex is not connected to any other vertex yet.

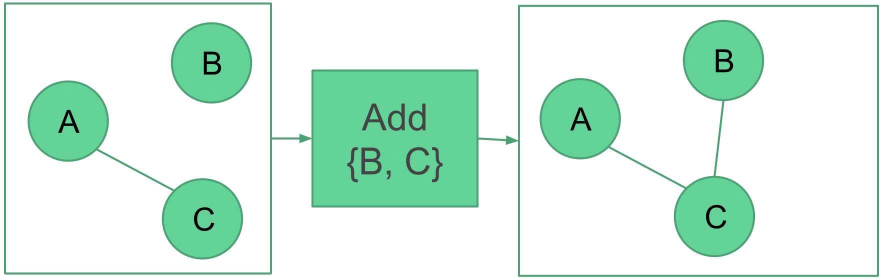


2. **Remove a vertex from a graph**
   When a vertex is removed, all connecting edges are also removed

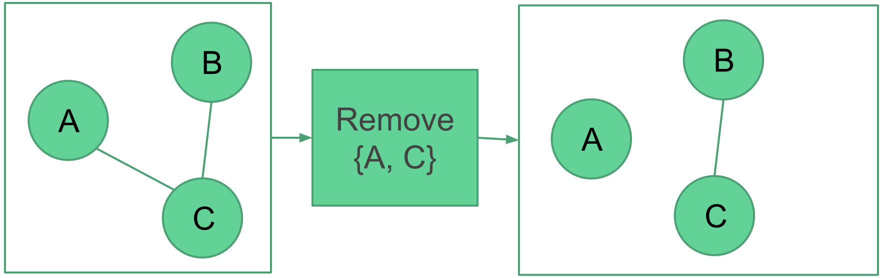


Operations on graphs

3. **Add an edge to a graph**



4. **Remove an edge from a graph**



Traversal techniques

Graph traversal

Process of visiting each vertex in a graph

Given a graph, $G = (V, E)$, and a vertex, $v \in V(G)$, visit all vertices in G that are reachable from v

2 ways of doing this:

1. Depth-first search (DFS)
2. Breadth-first search (BFS)

Depth-first search (DFS)

Process all descendants of a vertex before we move to an adjacent vertex.

DFS in a graph is similar to DFS in a tree. Since graphs may contain cycles unlike trees, we may come to the same node again. To avoid processing a node more than once, we keep track of visited nodes.

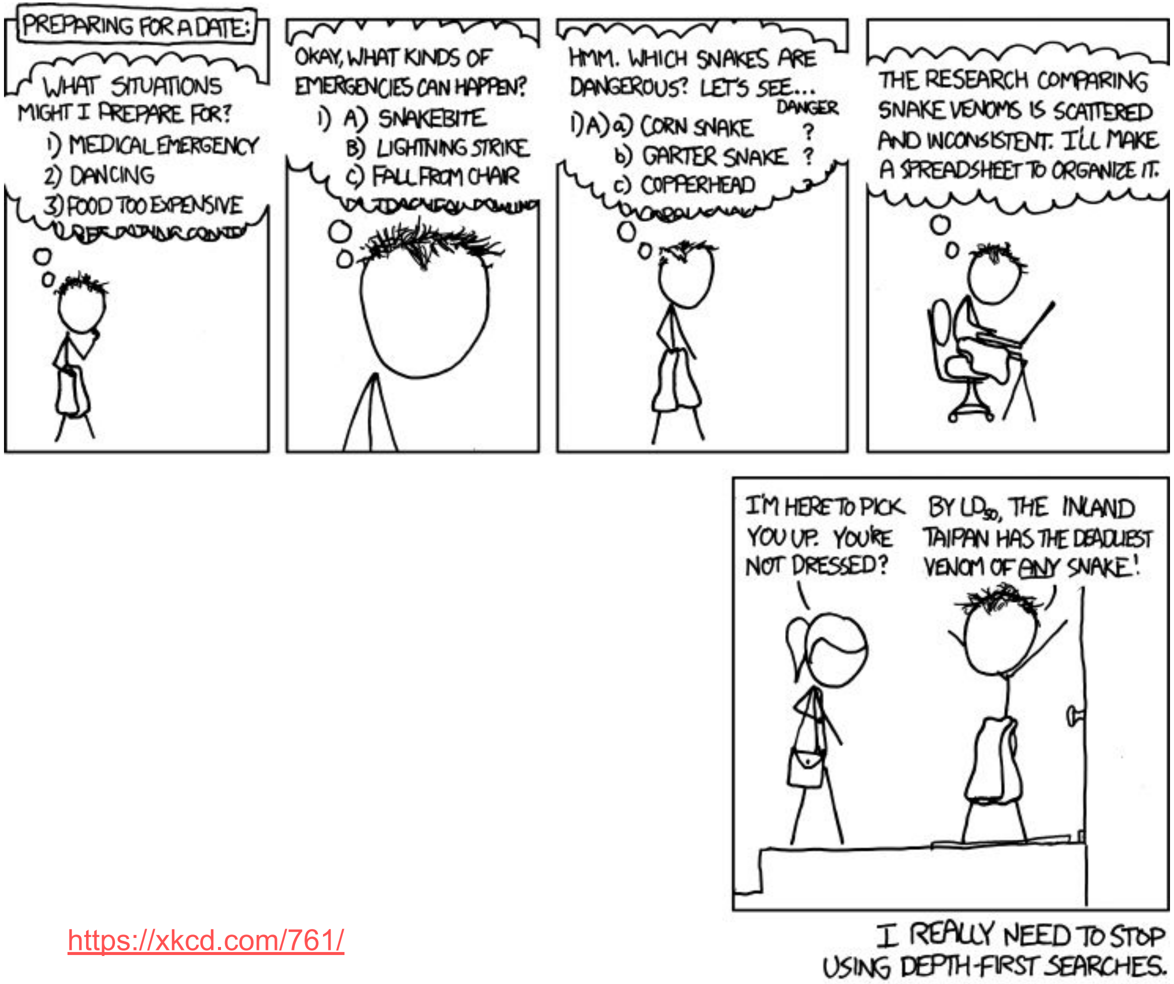
Uses a stack data structure to perform the search.

Depth-first search (DFS)

Basic idea:

1. Start by putting any one of the graph's vertices (starting vertex) on top of a stack.
2. Pop the topmost item from the stack and add it to the visited list.
3. Push the popped vertex’s unvisited neighbors into the top of stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

DFS



https://xkcd.com/761/

Depth-first search (DFS)

Algorithm: (Recursive) **DFS(G, s)**
Input: A graph, G, and a starting vertex, s
Output: A sequence of processed vertices

Steps:

1. mark(s); // Mark s as visited
2. $\forall (s, v) \in E(G)$
   a. DFS (G, v)

Depth-first search (DFS)

Algorithm: (Iterative) **DFS(G, s)**

Input: A graph, G, and a starting vertex, s

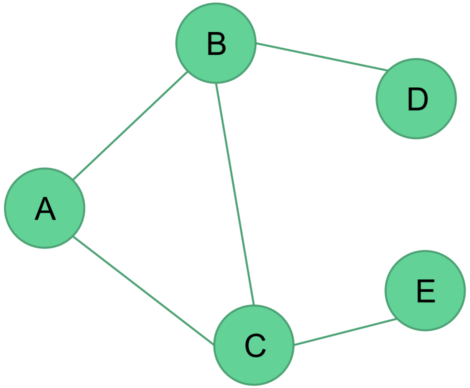
Output: A sequence of processed vertices

Steps:

1. mark(s); // Mark s as visited
2. $L \coloneqq \{s\}$ // Push s into the stack
3. **while** $L \neq \varnothing$ **do**
   a. $u \coloneqq \text{last}(L)$ // Top of the stack
   b. **if** $\exists (u, v)$ such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. else
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. endif
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A

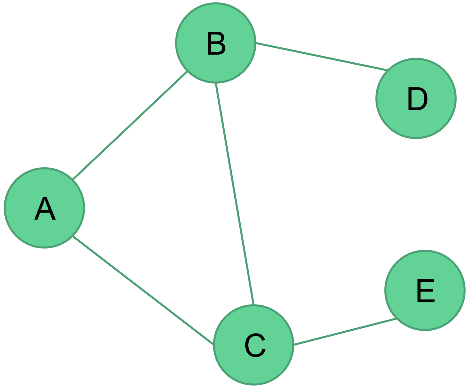


Stack

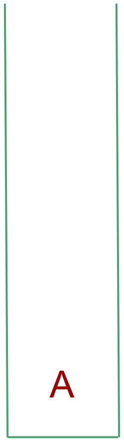
Visited vertices

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



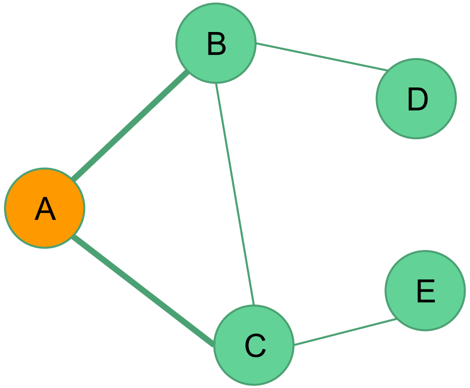
Visited vertices

A				

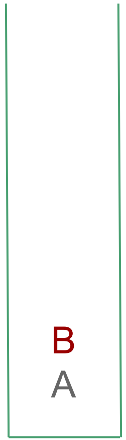
```
mark(s);      // Mark s as visited
L ≔ {s}       // Push s into the stack
```

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



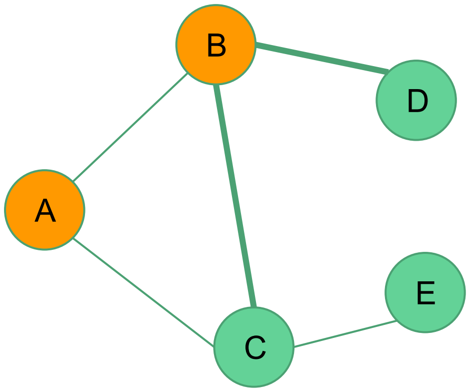
Visited vertices

A	B			

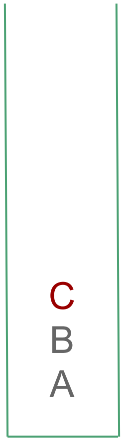
3. **while** L $\neq \varnothing$ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** $\exists$ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L $\cup$ {v}
   c. **else**
      i. L ≔ L \ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



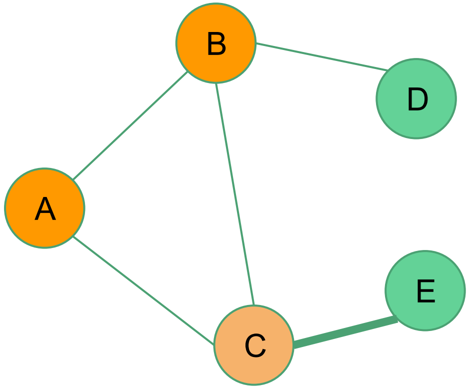
Visited vertices

A	B	C		

3. **while** $L \neq \varnothing$ **do**
   a. $u \coloneqq \mathrm{last}(L)$ // Top of the stack
   b. **if** $\exists (u, v)$ such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. **else**
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack

E
C
B
A

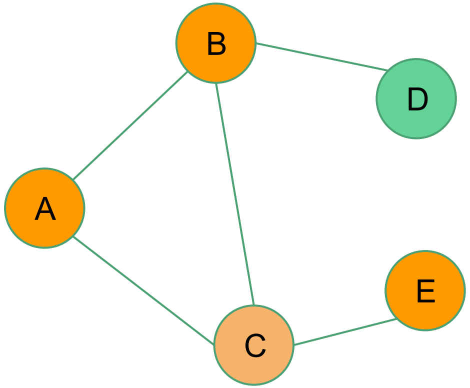
Visited vertices

A	B	C	E	

3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack

C
B
A

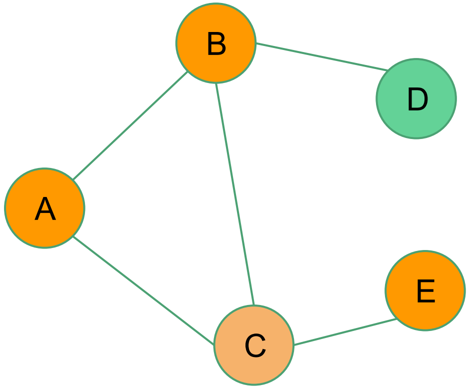
Visited vertices

A	B	C	E	

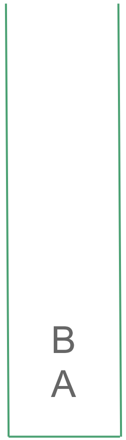
```
3.    while L ≠ ∅ do
   a.    u ≔ last(L)     // Top of the stack
   b.    if ∃ (u, v) such that v is unmarked then
         // Find neighbors of u
            i.    choose v of the smallest index;
            ii.   mark(v); L ≔ L ∪ {v}
   c.    else
            i.    L ≔ L ∖ {u}  // Pop from the stack
   d.    endif
4.    endwhile
```

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



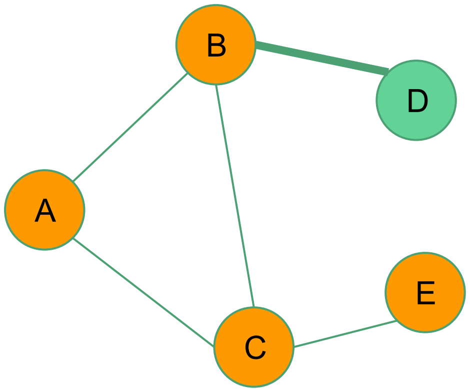
Visited vertices

A	B	C	E	

3. **while** $L \neq \varnothing$ **do**
    a. $u \coloneqq \mathrm{last}(L)$ // Top of the stack
    b. **if** $\exists\,(u, v)$ such that v is unmarked **then**
    // Find neighbors of u
        i. choose v of the smallest index;
        ii. mark(v); $L \coloneqq L \cup \{v\}$
    c. **else**
        i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
    d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack

D
B
A

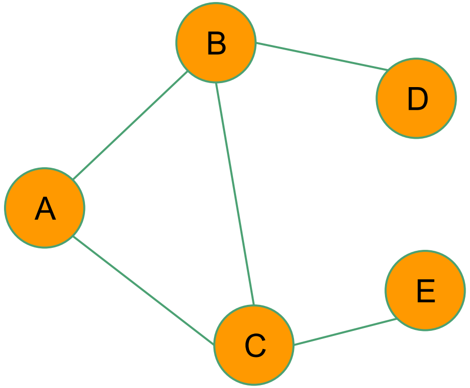
Visited vertices

A	B	C	E	D

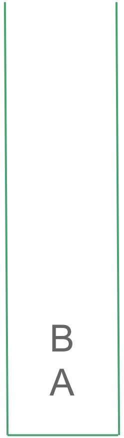
3. **while** L $\neq \varnothing$ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** $\exists$ (u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L $\cup$ {v}
   c. **else**
      i. L ≔ L $\setminus$ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



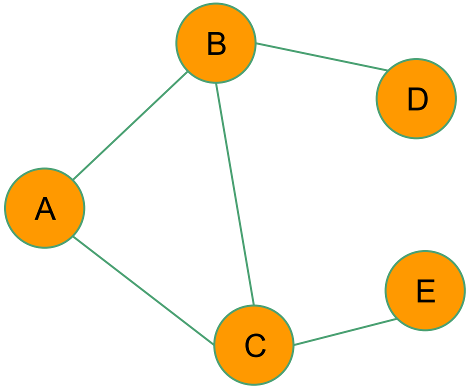
Visited vertices

A	B	C	E	D

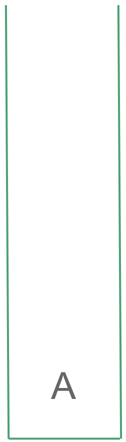
3. **while** L $\neq \emptyset$ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** $\exists$(u, v) such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L $\cup$ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A



Stack



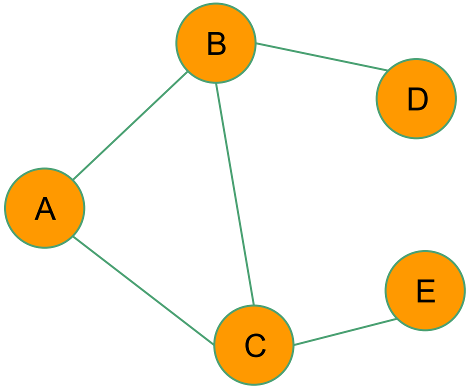
Visited vertices

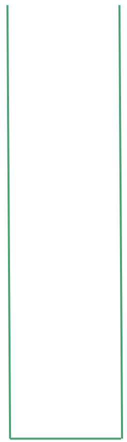
A	B	C	E	D

3. **while** $L \neq \varnothing$ **do**
   a. $u \coloneqq last(L)$ // Top of the stack
   b. **if** $\exists\,(u, v)$ such that v is unmarked **then**
      // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. **else**
      i. $L \coloneqq L \setminus \{u\}$ // Pop from the stack
   d. **endif**
4. **endwhile**

Depth-first search (DFS)

Example: Perform DFS on the following graph starting from A





Visited vertices

A	B	C	E	D

3. **while** L ≠ ∅ **do**
   a. u ≔ last(L) // Top of the stack
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. else
      i. L ≔ L∖ {u} // Pop from the stack
   d. endif
4. endwhile

Applications of DFS

- Finding a minimum spanning tree for unweighted graphs
- Detecting a cycle in the graph
- Finding a path from one node to another
- Topological ordering: determining the order of compilation tasks, resolving symbol dependencies in linkers etc.
- Solving problems with only one solution, such as maze
- etc.

Breadth-first search (BFS)

Process all adjacent vertices of a vertex before going to the next level.

Uses a queue data structure to perform the search.

Breadth-first search (BFS)

Basic idea:

1. Start by putting any one of the graph's vertices (starting vertex) at the back of a queue.
2. Dequeue the queue (take the vertex at the front of the queue) and add it to the visited list.
3. Enqueue the dequeued vertex’s unvisited neighbours to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Breadth-first search (BFS)

Algorithm: **BFS(G, s)**

Input: A graph, G, and a starting vertex, s

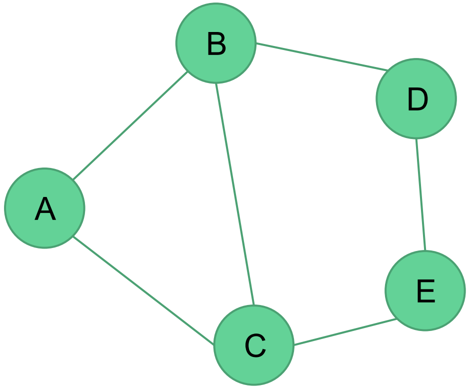
Output: A sequence of processed vertices

Steps:

1. mark(s); // Mark s as visited
2. L ≔ {s} // Push s into the queue
3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. else
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. endif
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A				

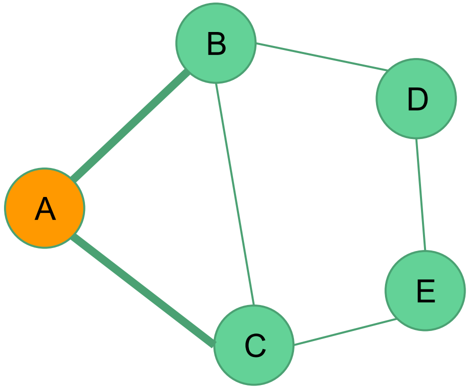
List

A				

```
mark(s);     // Mark s as visited
L ≔ {s}      // Push s into the queue
```

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B			

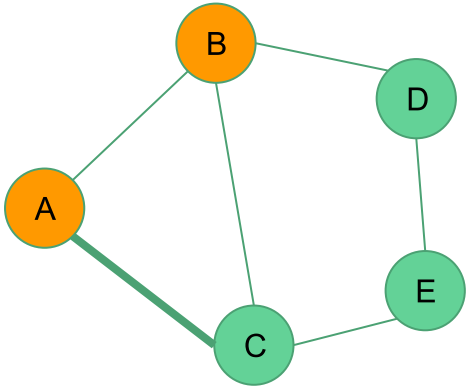
List

A	B			

```
3.  while L ≠ ∅ do
    a.  u ≔ first(L)    // Front of the queue
    b.  if ∃ (u, v) such that v is unmarked
        then    // Find neighbors of u
        i.   choose v of the smallest index;
        ii.  mark(v); L ≔ L ∪ {v}
    c.  else
        i.   L ≔ L ∖ {u}  // Dequeue the
             queue
    d.  endif
4.  endwhile
```

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C		

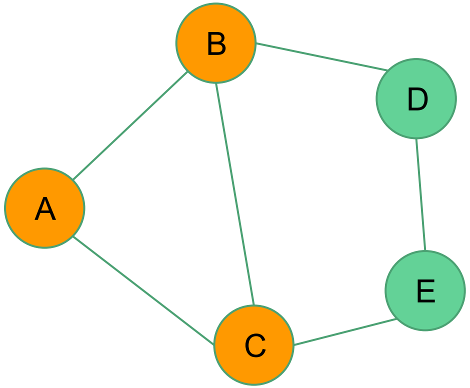
List

A	B	C		

3. **while** L $\neq \varnothing$ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C		

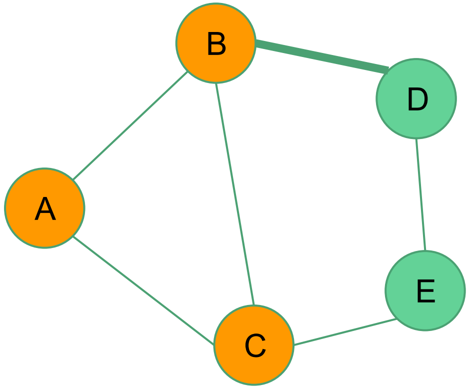
List

B	C			

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	

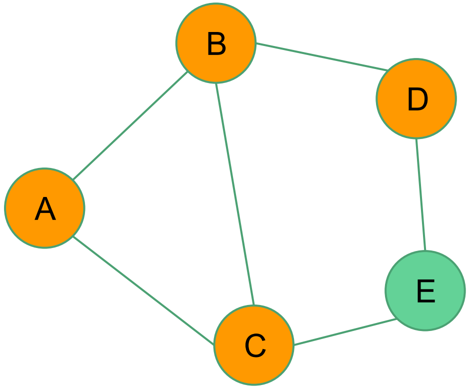
List

B	C	D		

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	

List

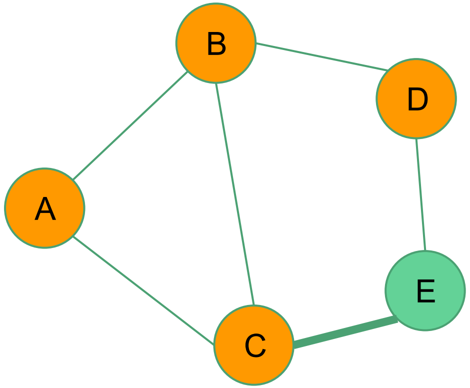
C	D			

3. **while** L ≠ ∅ **do**
   - a. u ≔ first(L) // Front of the queue
   - b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
     - i. choose v of the smallest index;
     - ii. mark(v); L ≔ L ∪ {v}
   - **c.** **else**
     - i. L ≔ L ∖ {u} // Dequeue the queue
   - **d.** **endif**

4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

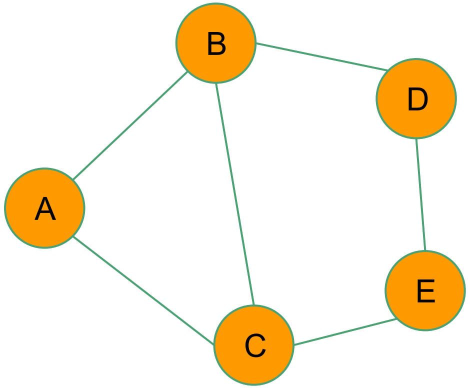
List

C	D	E		

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. else
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. endif
4. endwhile

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

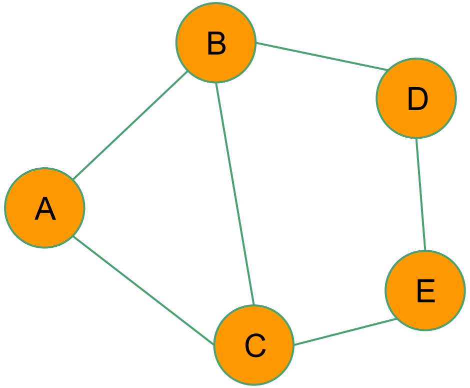
List

D	E			

3. **while** L ≠ ∅ **do**
   a. u ≔ first(L) // Front of the queue
   b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); L ≔ L ∪ {v}
   c. **else**
      i. L ≔ L ∖ {u} // Dequeue the queue
   d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

A	B	C	D	E

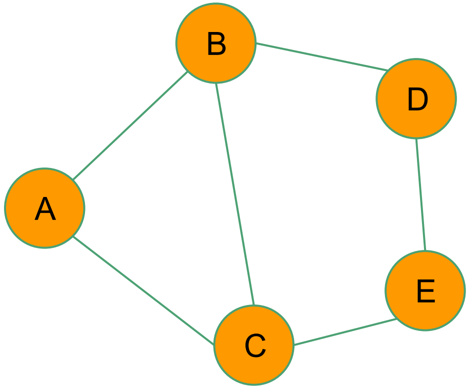
List

E				

3. **while** L ≠ ∅ **do**
    a. u ≔ first(L) // Front of the queue
    b. **if** ∃ (u, v) such that v is unmarked **then** // Find neighbors of u
        i. choose v of the smallest index;
        ii. mark(v); L ≔ L ∪ {v}
    c. **else**
        i. L ≔ L ∖ {u} // Dequeue the queue
    d. **endif**
4. **endwhile**

Breadth-first search (BFS)

Example: Perform BFS on the following graph starting from A



Visited vertices

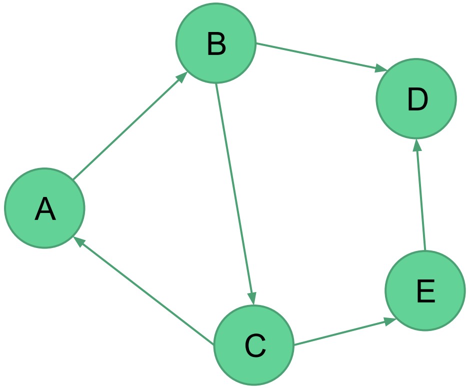
A	B	C	D	E

List

3. **while** $L \neq \emptyset$ **do**
   a. $u \coloneqq \text{first}(L)$ // Front of the queue
   b. **if** $\exists\,(u, v)$ such that v is unmarked **then** // Find neighbors of u
      i. choose v of the smallest index;
      ii. mark(v); $L \coloneqq L \cup \{v\}$
   c. **else**
      i. $L \coloneqq L \setminus \{u\}$ // Dequeue the queue
   d. **endif**
4. **endwhile**

Exercise

Perform DFS and BFS on the following graph starting from A



Hint: For directed graphs, when exploring a vertex v, we only want to look at edges (v,w) going out of v; we ignore the other edges coming into v.

Applications of BFS

- Finding a minimum spanning tree for unweighted graphs
- Web crawler: Begin from a starting page and follow all links from this page and keep doing the same
- Social networks: Find people within a given distance ‘k’ from a person
- Finding the shortest path to another node
- GPS navigation systems: Finding the direction to reach from one place to another
- etc.

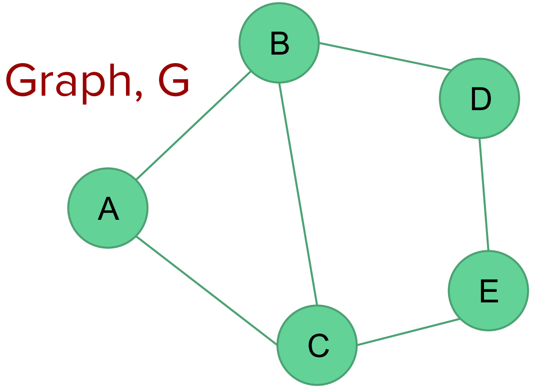
Minimum spanning tree

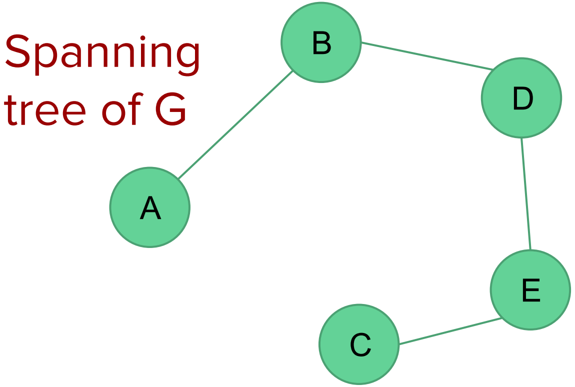
Spanning tree

A spanning tree of a connected graph G is a tree that consists solely of edges in G and that includes all of the vertices in G.

Our solution to generate a spanning tree must satisfy the following constraints:

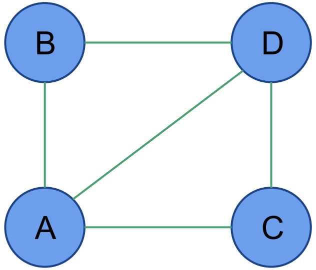
1. We must use only edges within the graph
2. We must use exactly n-1 edges
3. We may not use edges that would produce a cycle



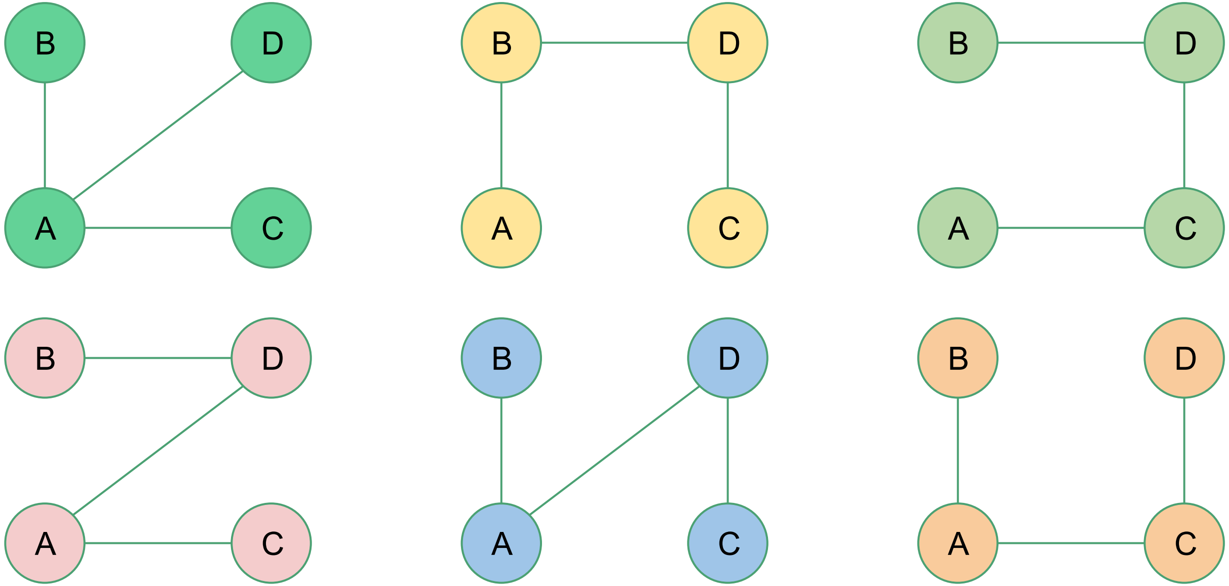


Spanning tree

A single graph can have many spanning trees.



Graph G



Spanning trees of G

Spanning tree

A spanning tree can be generated using a DFS or a BFS. The spanning tree is formed from those edges traversed during the search.

- If a breadth first search is used, the resulting spanning tree is called a **breadth first spanning tree**.
- If a depth first search is used, it is called **depth first spanning tree**.

For a disconnected / disjoint graph, a **spanning forest** is defined.

Minimum spanning tree (MST)

A **minimum spanning tree of a weighted graph** is a **spanning tree of least weight**, i.e. a spanning tree in which the total weight of the edges is guaranteed to be the minimum of all possible trees in the graph.

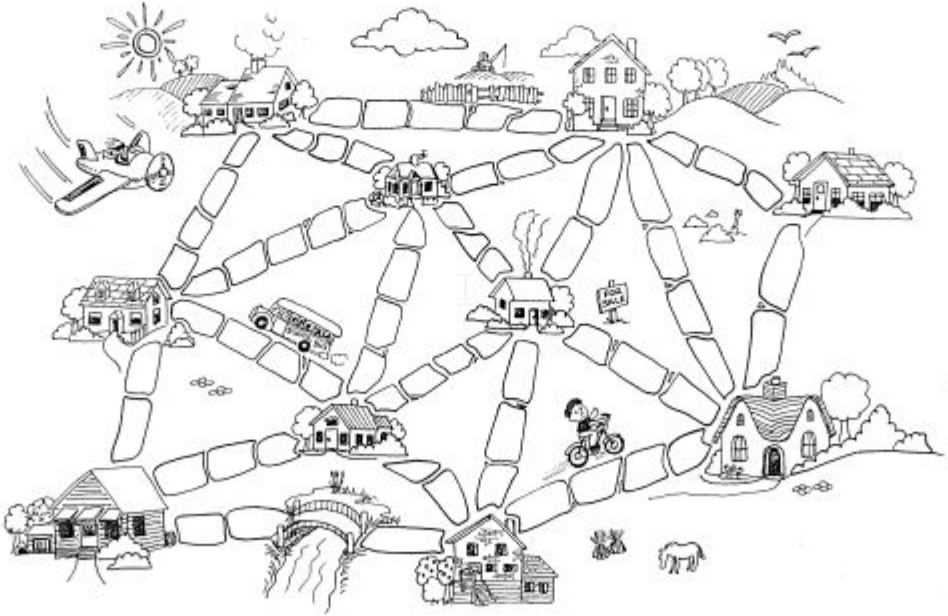
If the weights in the network/graph are unique, there is only one MST

If there are duplicate weights, there may be one or more MSTs

Application: Network design, Muddy city problem

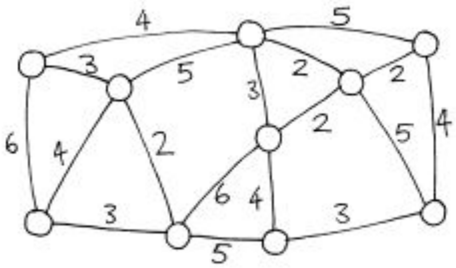
Muddy city problem

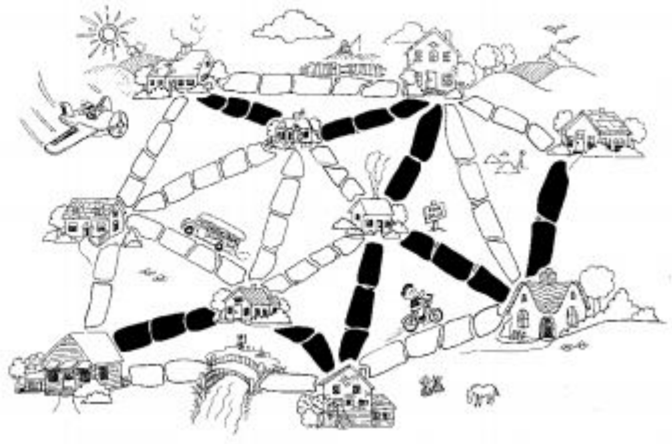
- A city with no paved road
- The mayor of the city decided to pave some of the streets with the following two conditions:
    1. Enough streets must be paved so that it is possible for everyone to travel from their house to anyone else's house only along paved roads, and
    2. The paving should cost as little as possible.



Muddy city problem

Solution: Minimum spanning tree





Growing a MST

Problem:

Given a connected, undirected graph $G = (V, E)$ with a weight function $w: E \rightarrow R$, find a minimum spanning tree for G

A greedy strategy:

Grow the minimum spanning tree one edge at a time

- Kruskal's algorithm
- Prim's algorithm

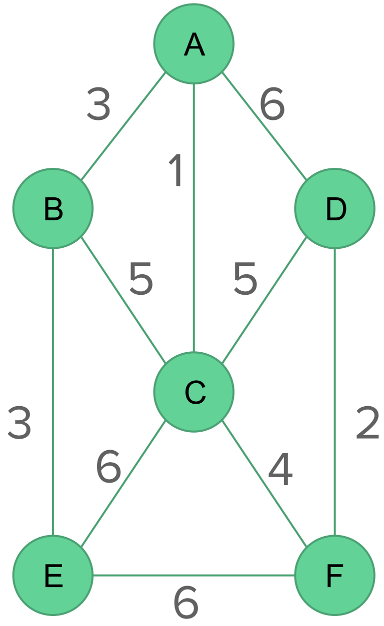
Kruskal’s algorithm

Steps:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle in the spanning tree formed so far. If no cycle is formed, include this edge. Otherwise, discard it.
3. Repeat step#2 until it is a spanning tree.

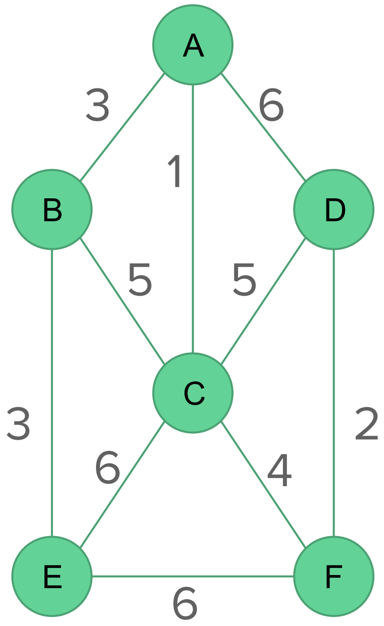
Kruskal's algorithm

Example: Find a minimum spanning tree of the following graph



Kruskal’s algorithm

Step 1: Sort all edges in ascending order by weight

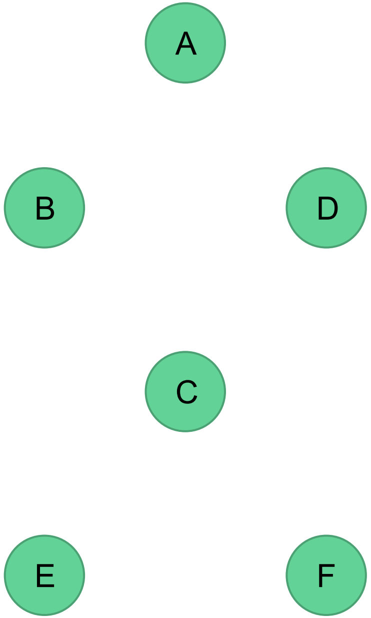


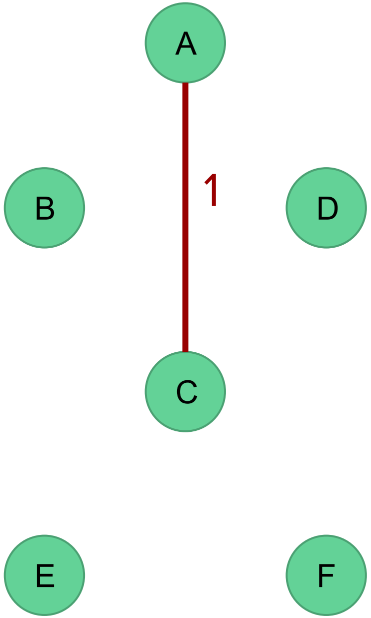
Edge	Weight
(A, C)	1
(D, F)	2
(B, E)	3
(A, B)	3
(C, F)	4

Edge	Weight
(B, C)	5
(C, D)	5
(A, D)	6
(C, E)	6
(E, F)	6

Kruskal's algorithm

Step 2: Add edges in sequence

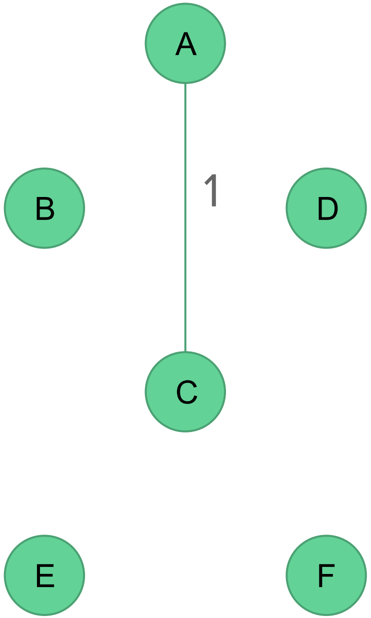


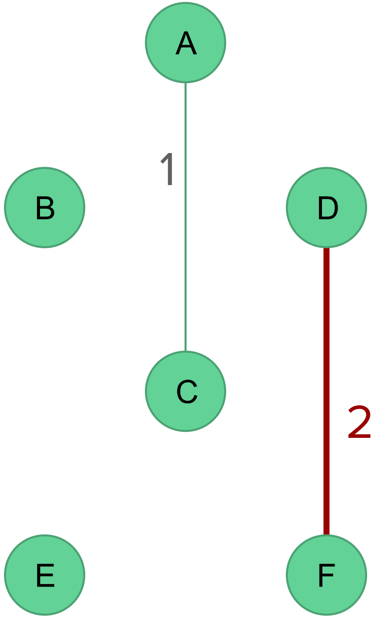


Edge	Weight
(A, C)	**1**
(D, F)	2
(B, E)	3
(A, B)	3
(C, F)	4

Kruskal’s algorithm

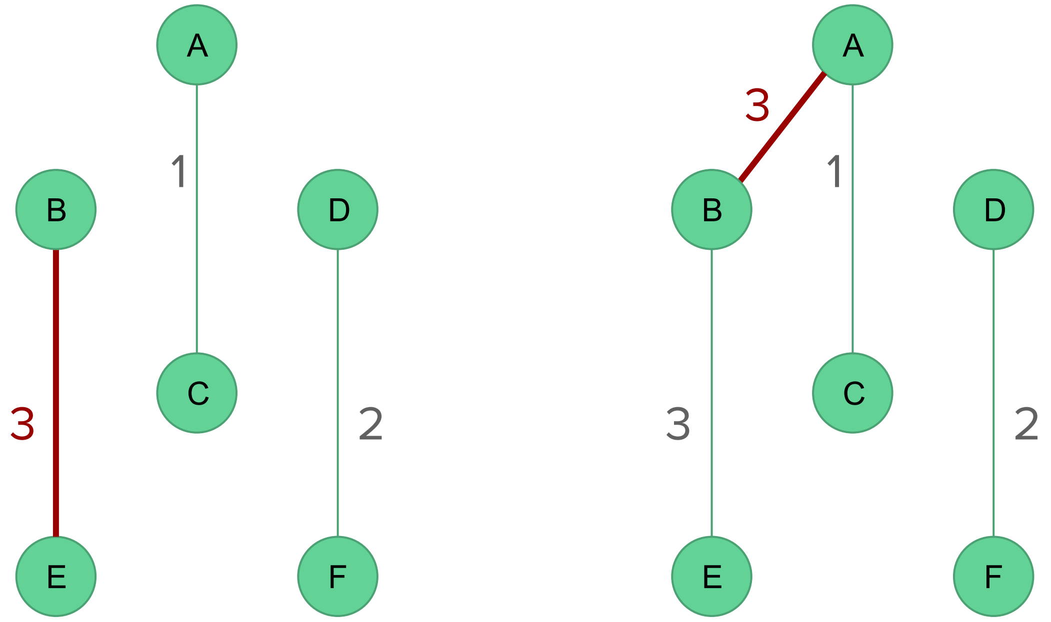
Step 2: Add edges in sequence





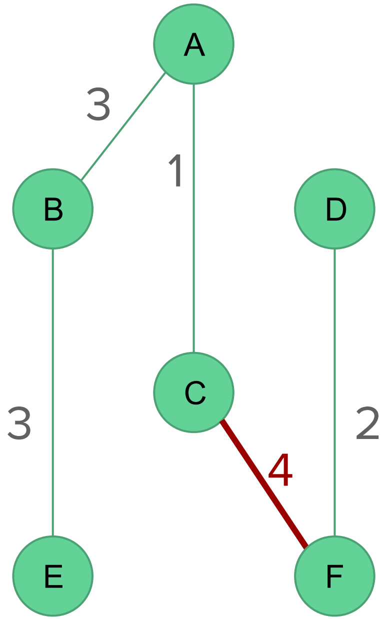
Edge	Weight
(A, C)	1
(D, F)	**2**
(B, E)	3
(A, B)	3
(C, F)	4

Kruskal's algorithm



Edge	Weight
(A, C)	1
(D, F)	2
(B, E)	**3**
(A, B)	**3**
(C, F)	4

Kruskal's algorithm



Edge	Weight
(A, C)	1
(D, F)	2
(A, B)	3
(B, E)	3
(C, F)	**4**

Prim’s algorithm

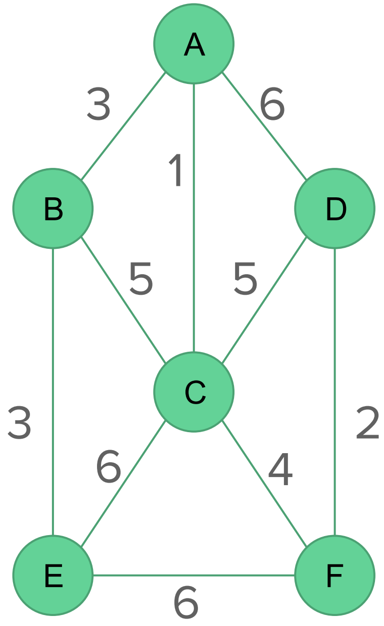
Grows a single tree and adds a light edge (edge with the lowest weight) in each iteration

Steps:

1. Start by picking any vertex to be the root of the tree.
2. While the tree does not contain all vertices in the graph, find shortest edge leaving the tree and add it to the tree

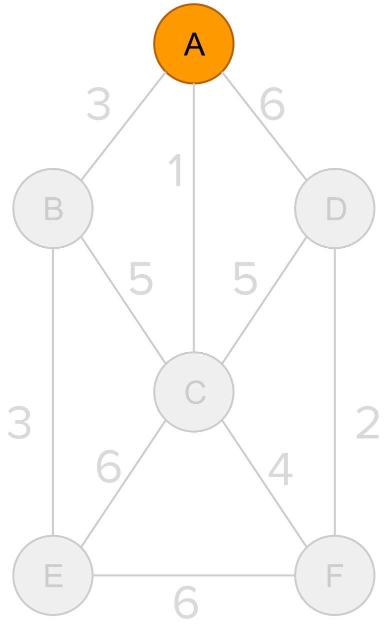
Prim’s algorithm

Example: Find a minimum spanning tree of the following graph



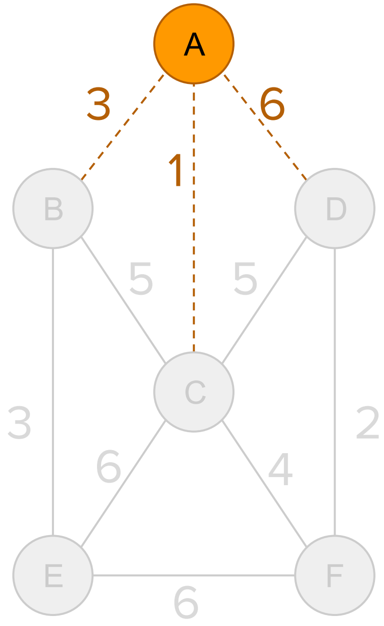
Prim's algorithm

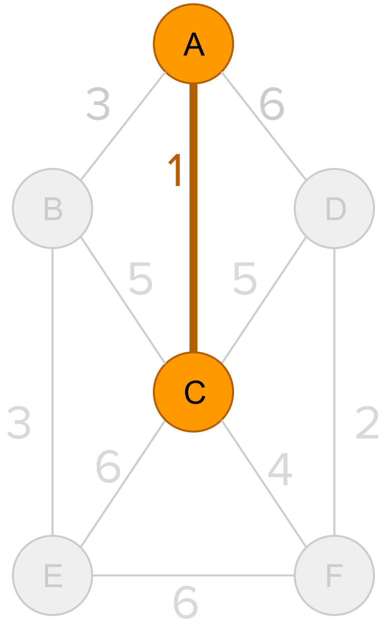
Step 1: Pick any vertex to be the root of the tree. Let's say A will be the root



Prim's algorithm

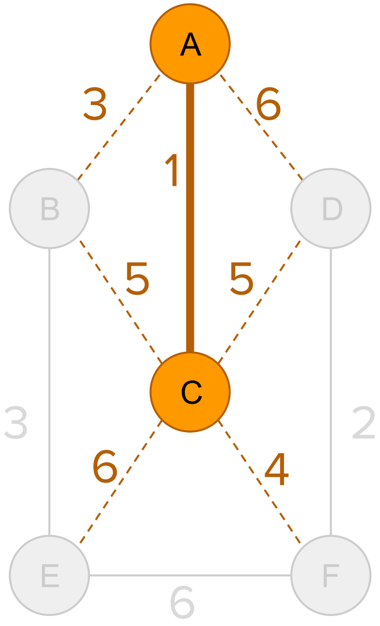
Step 2: Find shortest edge leaving the tree and add it to the tree

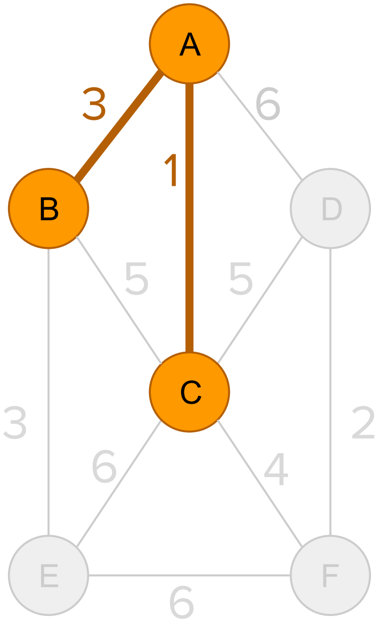




Prim’s algorithm

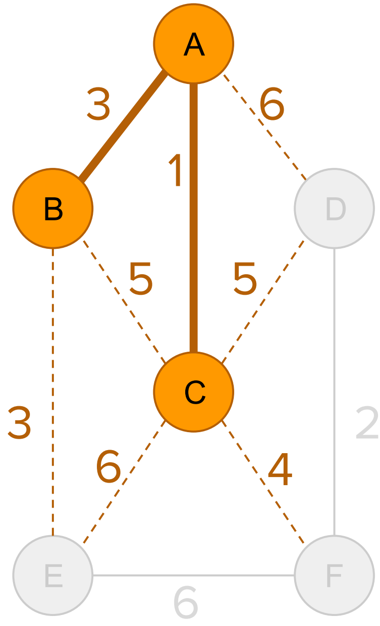
Step 2: Find shortest edge leaving the tree and add it to the tree

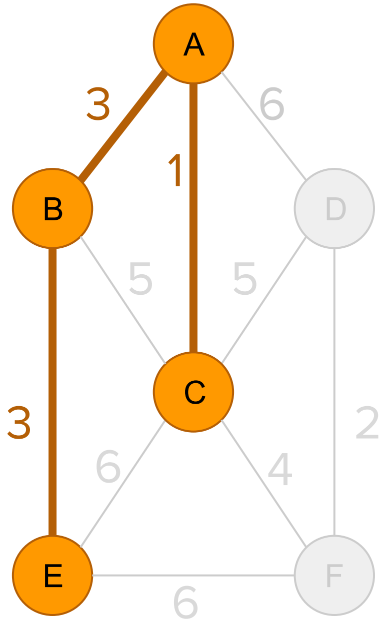




Prim’s algorithm

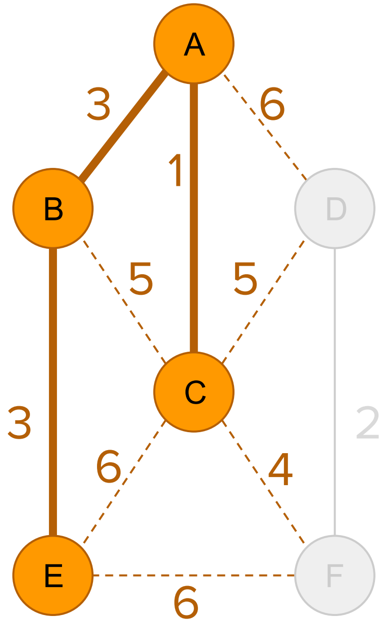
Step 2: Find shortest edge leaving the tree and add it to the tree

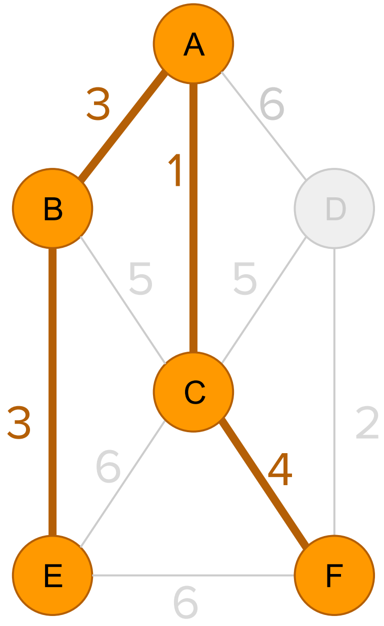




Prim's algorithm

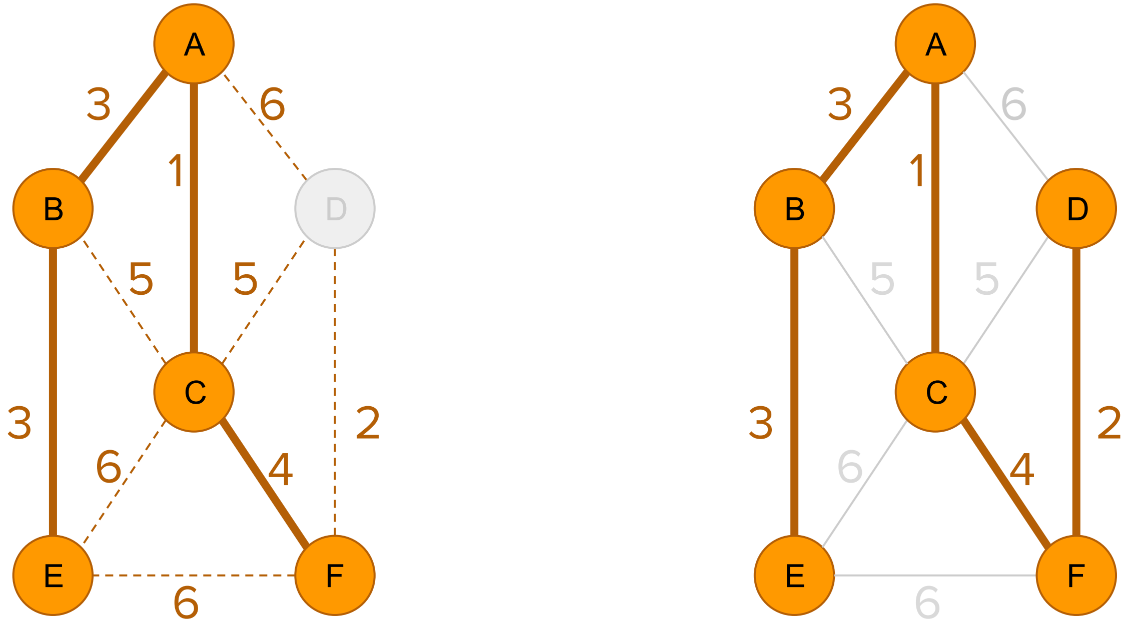
Step 2: Find shortest edge leaving the tree and add it to the tree





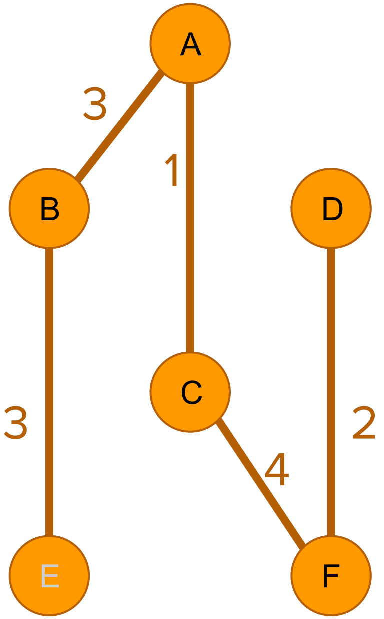
Prim's algorithm

Step 2: Find shortest edge leaving the tree and add it to the tree



Prim’s algorithm

So the spanning tree is



Shortest path algorithms

Shortest path algorithm

Finds the shortest path between two vertices in a graph

Applications:

- Finding the shortest path from one location to another in Google Maps, MapQuest, OpenStreetMap, (KTM Public Route) etc.
- Used by Telephone networks, Cellular networks for routing/connection in communication
- IP routing
- Word ladder problem

Single-source shortest path problem

The problem of finding shortest paths from a source vertex v to all other vertices in the graph.

Optimal substructure of a shortest path

Shortest-path algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

Dijkstra's algorithm

A solution to the single-source shortest path problem in graph theory.

- **Input**: Weighted graph $G = (V, E)$ and source vertex $v \in V$, such that all edge weights are nonnegative
- **Output**: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

Dijkstra’s shortest path algorithm

Steps:

1. Insert the first vertex into the tree
2. From every vertex already in the tree, examine the total path length to all adjacent vertices not in the tree. Selected the edge with the minimum total path weight and insert it into the tree
3. Repeat step 2 until all vertices are in the tree

Dijkstra's shortest path algorithm

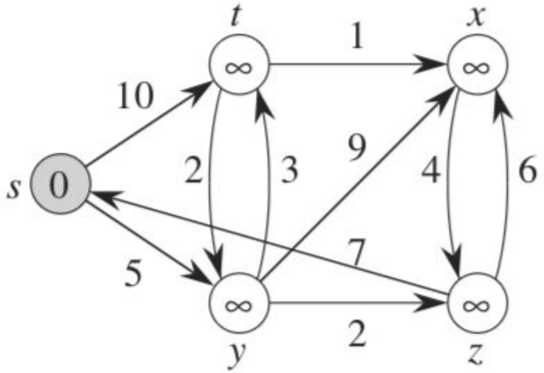
$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$ # Q is a min-priority queue

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while

Dijkstra’s shortest path algorithm: Example

Find the shortest path from s to all other vertices.

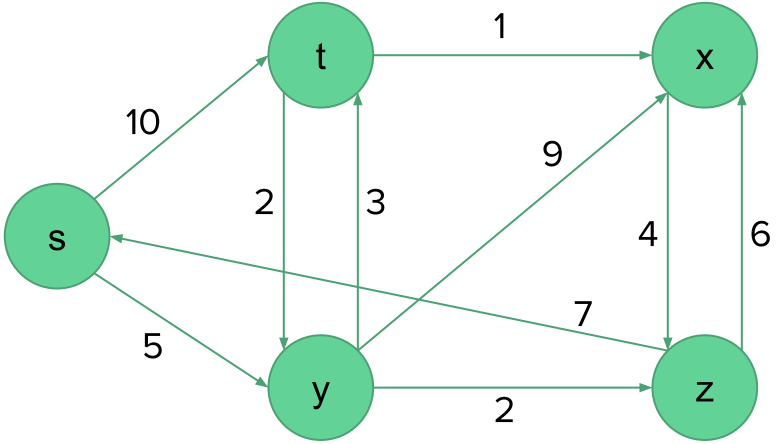


Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$
while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v,u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v,u)$
    end if
  end for
end while

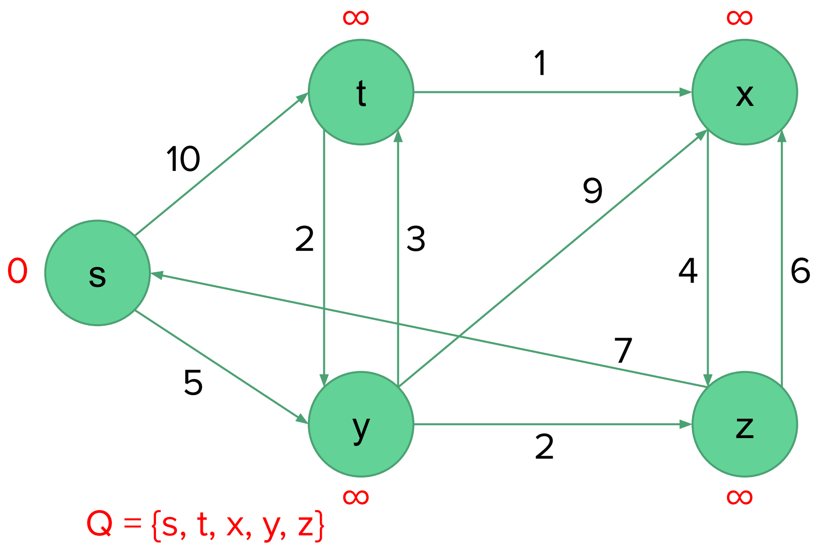


Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$



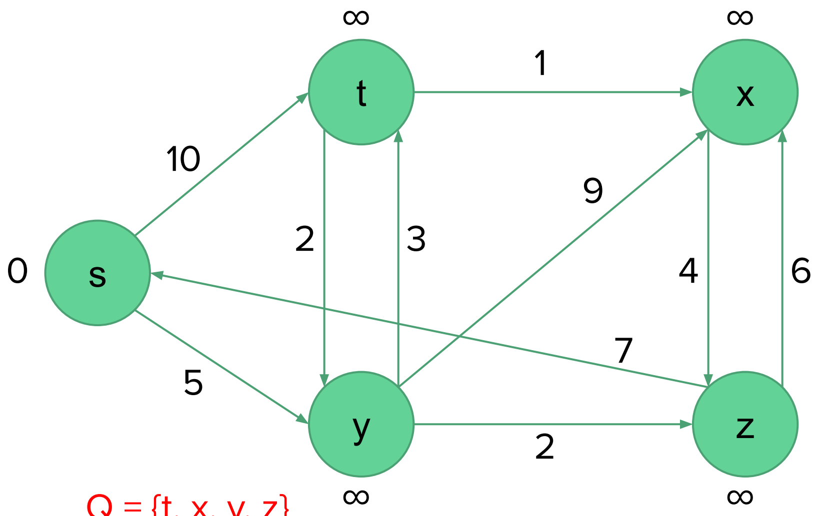
Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while



Q = {t, x, y, z}
v = s
Neighbours of v = {t, y}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

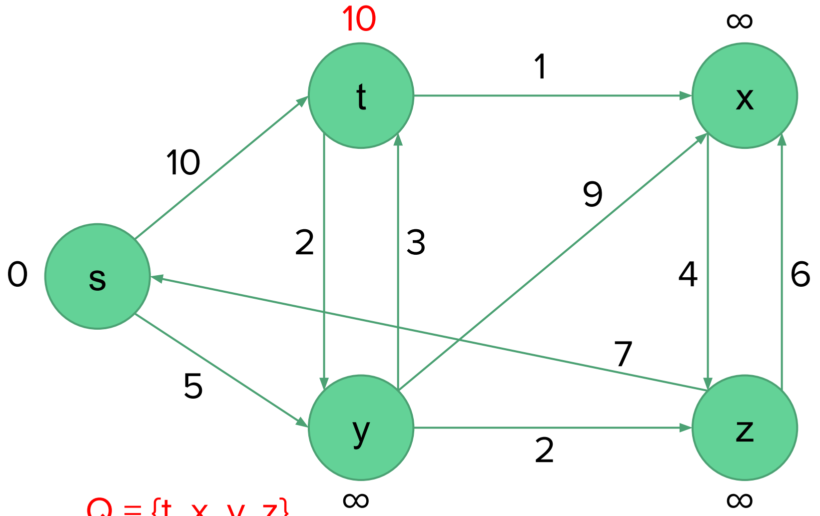
    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

end while



Q = {t, x, y, z}

v = s

Neighbours of v = {t, y}

d(s) + e(s, t) ≤ d(t) True

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

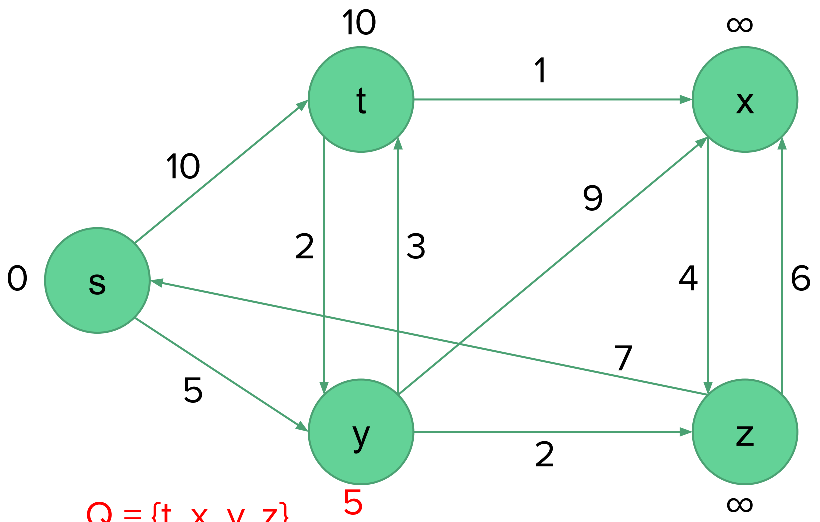
    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

    end if

  end for

end while



Q = {t, x, y, z}

v = s

Neighbours of v = {t, y}

d(s) + e(s, y) ≤ d(y)   True

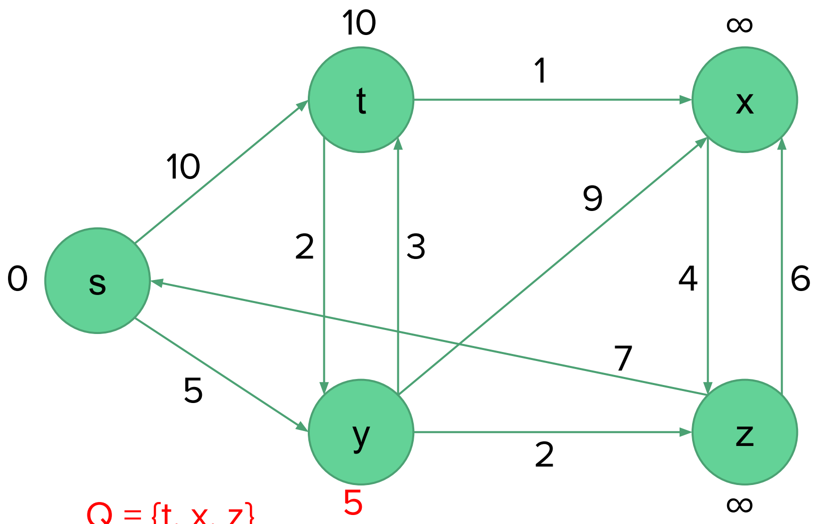
Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v,u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v,u)$
    end if
  end for
end while



Q = {t, x, z}
v = y
Neighbours of v = {t, x, z}

Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

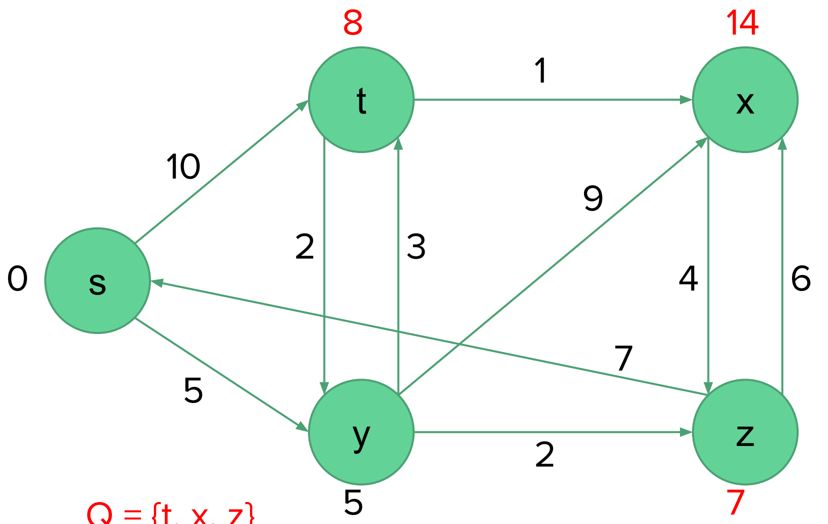
    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

end while



Q = {t, x, z}

v = y

Neighbours of v = {t, x, z}

d(y) + e(y, t) ≤ d(t) True

d(y) + e(y, x) ≤ d(x) True

d(y) + e(y, z) ≤ d(z) True

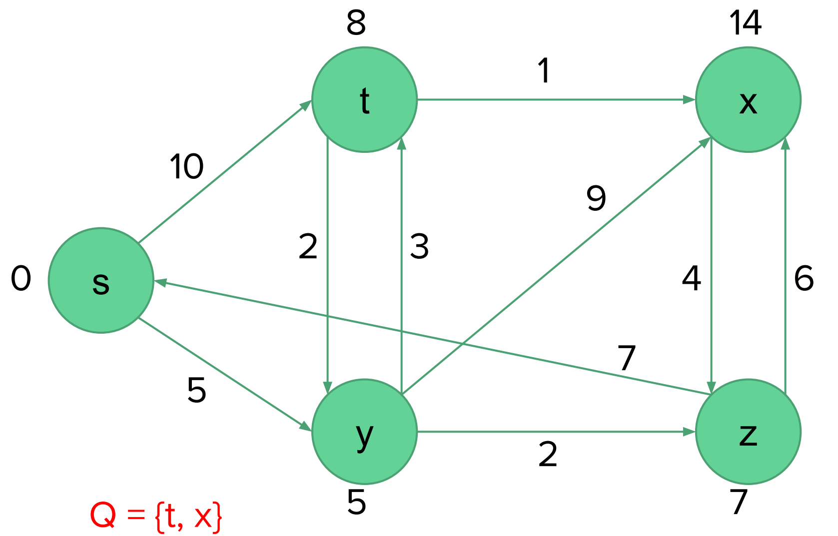
Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

```
while Q not empty do
    v ← Q.pop()
    for all neighbours u of v do
        if d(v) + e(v, u) ≤ d(u) then
            d(u) ← d(v) + e(v, u)
        end if
    end for
end while
```



Q = {t, x}

v = z

Neighbours of v = {x, s}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

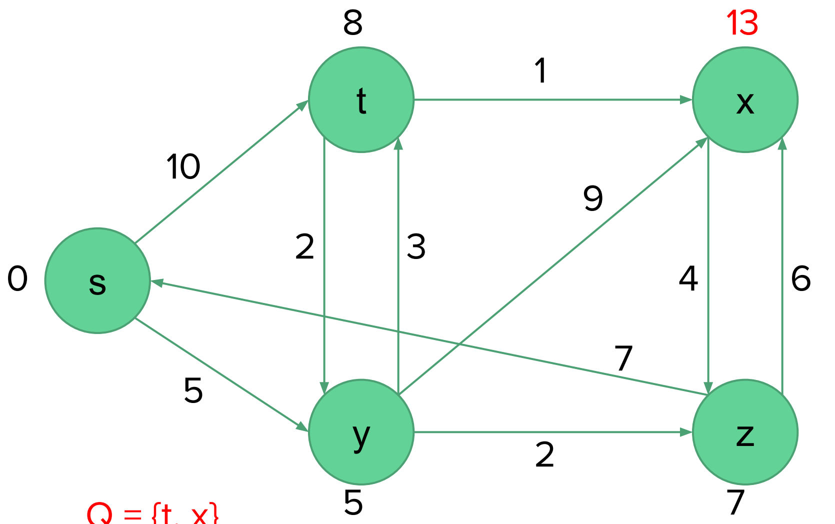
    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

    end if

  end for

end while



Q = {t, x}

v = z

Neighbours of v = {x, s}

d(z) + e(z, x) ≤ d(x) True

d(z) + e(z, s) ≤ d(s) False

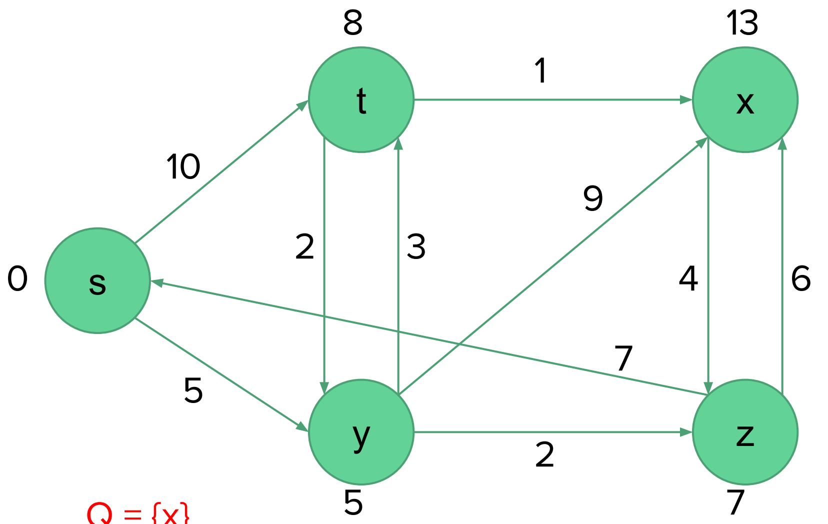
Dijkstra’s shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**
  $v \leftarrow Q.pop()$
  for all neighbours u of v **do**
    if $d(v) + e(v, u) \leq d(u)$ **then**
      $d(u) \leftarrow d(v) + e(v, u)$
    end if
  end for
end while



Q = {x}
v = t
Neighbours of v = {x, y}

Dijkstra's shortest path algorithm: Example

Find the shortest path distance from s to all other vertices.

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $d(v)$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

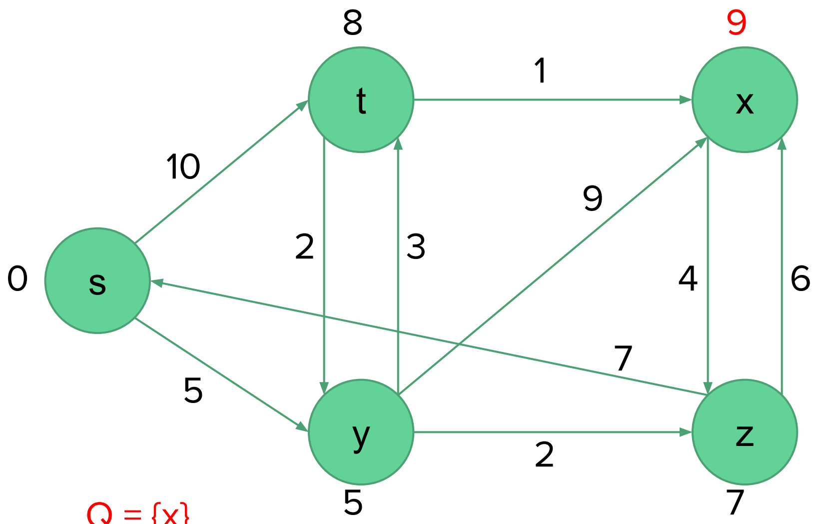
    if $d(v) + e(v,u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v,u)$

    end if

  end for

end while



Q = {x}

v = t

Neighbours of v = {x, y}

d(t) + e(t, x) ≤ d(x) True

d(t) + e(t, y) ≤ d(y) False

Running time

- Depends on the implementation
- The simplest implementation is to store vertices in an array or linked list. This will produce a running time of $O(|V|^2 + |E|)$
- For sparse graphs, or graphs with very few edges and many nodes, it can be implemented more efficiently storing the graph in an adjacency list using a binary heap or priority queue. This will produce a running time of $O((|E| + |V|) \log |V|)$

A* search algorithm

A* (pronounced ‘**A-star**’) is a search algorithm that finds the shortest path between some nodes S and T in a graph

It is a **generalization of Dijkstra's algorithm** that cuts down on the size of the subgraph that must be explored using a **heuristic function**

Suppose we want to get to node T, and we are currently at node v. Informally, a heuristic function h(v) is a function that ‘estimates’ how v is away from T

A* search algorithm

$$d(v) \leftarrow \begin{cases} \infty & \text{if } v \neq S \\ 0 & \text{if } v = S \end{cases}$$

$Q :=$ the set of nodes in V, sorted by $\boxed{d(v) + h(v)}$

while Q not empty **do**

  $v \leftarrow Q.pop()$

  for all neighbours u of v **do**

    if $d(v) + e(v, u) \leq d(u)$ **then**

      $d(u) \leftarrow d(v) + e(v, u)$

    end if

  end for

end while

Dijkstra's algorithm is a special case of A*, when we set h(v) = 0 for all v