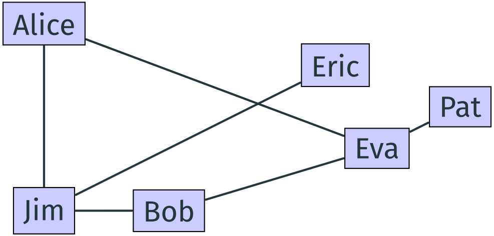
Graphs

Dr. Anirban Ghosh

School of Computing
University of North Florida

What is a graph?



Vertices. {Alice, Jim, Bob, Eva, Eric, Pat}

Edges. { {Alice, Jim}, {Jim, Bob}, {Alice, Eva}, {Jim, Eric}, {Bob, Eva}, {Eva, Pat} }

Definition

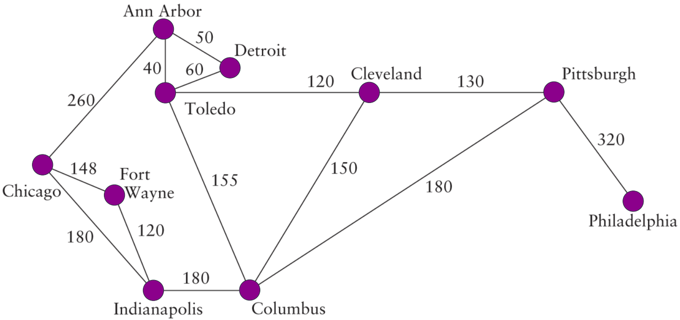
A **graph** is an abstract data type used to represent relationships that exist between pairs of objects of a certain type. The objects are known as the **vertices** and the relationships are known as the **edges**. Two objects have an edge between them if they have a relationship between them.

Social network graphs

There is a unique vertex for every user. An edge between two users exists if they are connected (friends).

Computer network graphs

There is a unique vertex for every machine/server. An edge between two machines/servers exists if they are directly connected in the network.



☞ This is a **weighted graph** since every edge has a real-number weight associated with it

Road network graphs

There is a unique vertex for every town/city. An edge between two cities/towns exists if there is direct road between them.

Flight network graphs

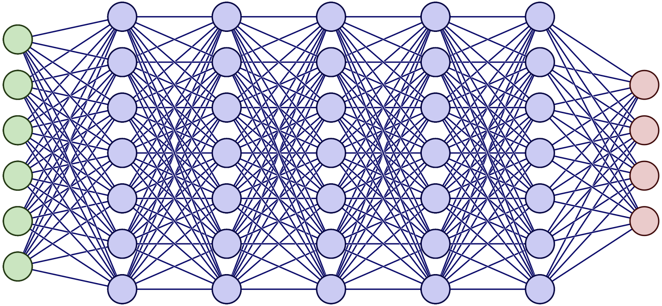
There is a unique vertex for every airport. An edge between two airports exists if there is an airline route between them.

Applications

Used for modeling in

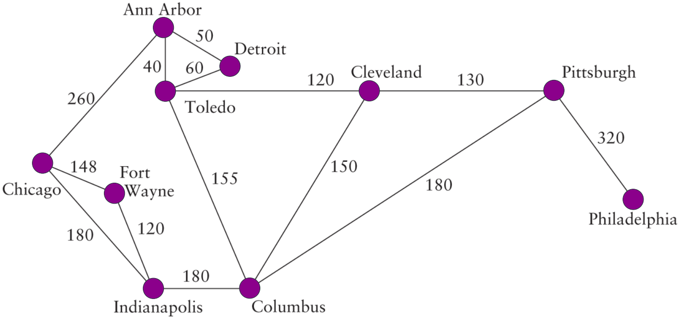
- Social networks
- Road and airline networks
- Computer networks
- Robotics
- Operating systems
- Distributed systems
- AI and machine learning
- Databases
- Software testing

$\vdots$



A deep neural network (used in Machine Learning)

Degree



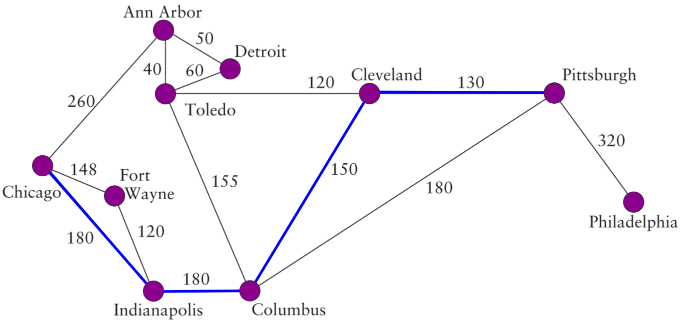
Degree of a vertex

Degree of a vertex is the number of edges incident on it.

Examples. degree(Columbus) = 4, degree(Philadelphia) = 1

Degree of a graph

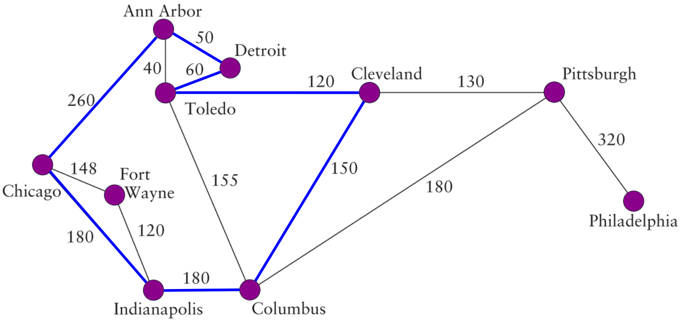
Degree of a graph is the maximum degree of a vertex in it. Degree of the above graph is 4.



Path

A simple path in a graph is a sequence of vertices in which each successive vertex is adjacent to its predecessor (an edge exists between them) and vertices do not repeat in the sequence except that the first and last vertices may be same

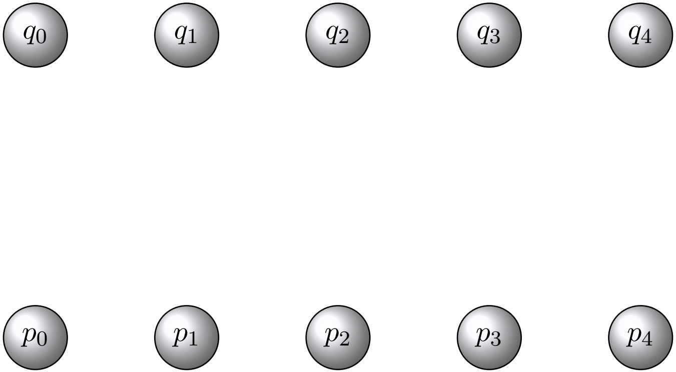
Cycle



Path

A cycle is a simple path in which the first and last vertices are the same

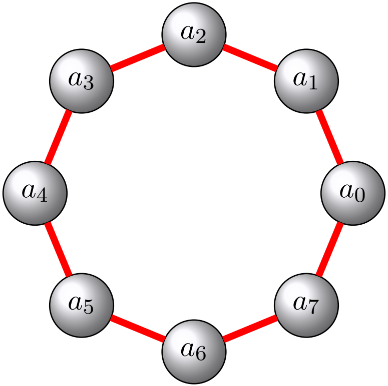
Empty graph (no edges)



Fact

Empty graphs have zero edges.

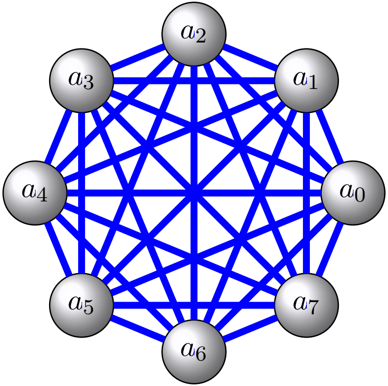
A cycle graph



Fact

Every cycle graph on n vertices has exactly n edges.

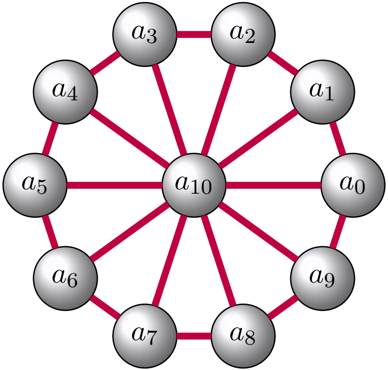
A complete graph



Fact

A complete graph on n vertices has $C(n,2) = \frac{n(n-1)}{2}$ edges.

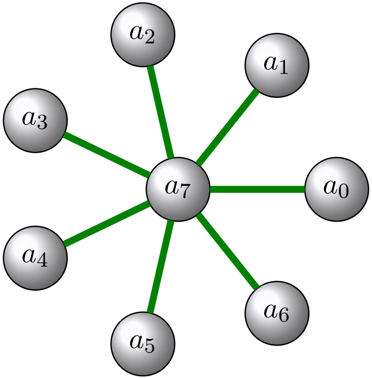
A wheel graph



Fact

A wheel graph on n vertices has $2n-2$ edges since there are $n-1$ edges in the outter cycle and $n-1$ spokes.

A star graph



Fact

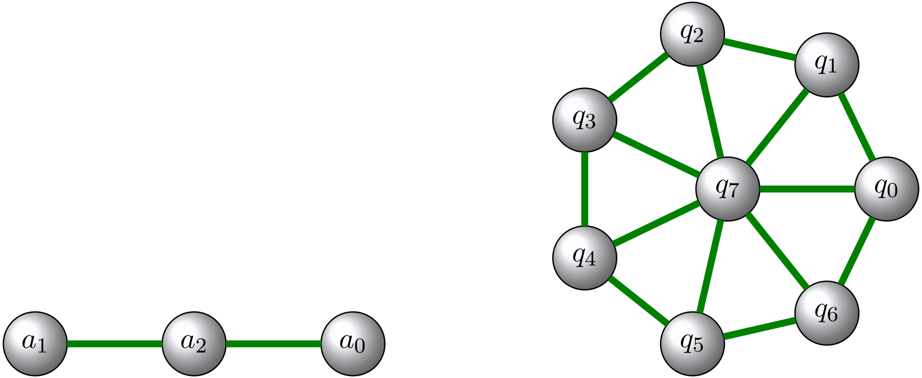
A star graph on n vertices has $n-1$ edges.

A path graph

Fact

Every path graph on n vertices has exactly $n-1$ edges.

Disconnected graphs

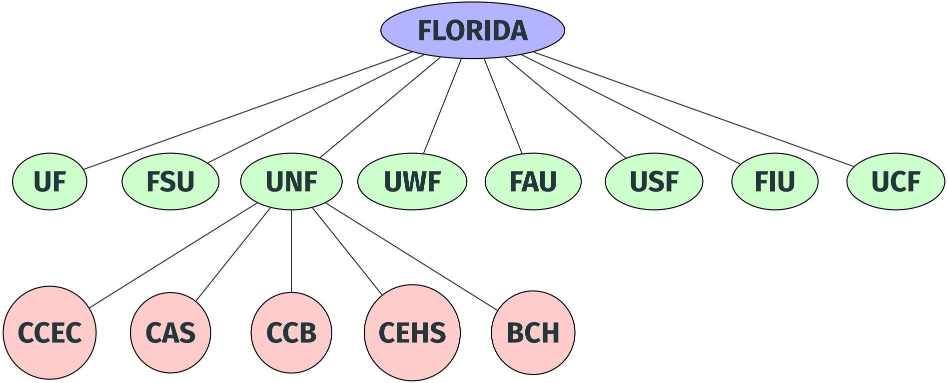


A **component** is a subgraph in which there is path between every pair of vertices

The above graph has 2 components $\{a_1, a_2, a_0\}$ and $\{q_0, \ldots, q_7\}$

A graph is **disconnected** if it has 2 or more components

A tree



A **tree** is a connected graph without a cycle

In a tree, every pair of vertices has exactly one path between them

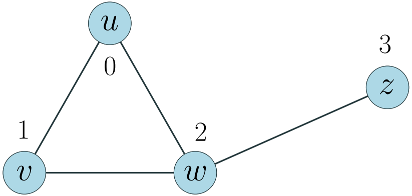
A tree with n vertices has exactly $n - 1$ edges

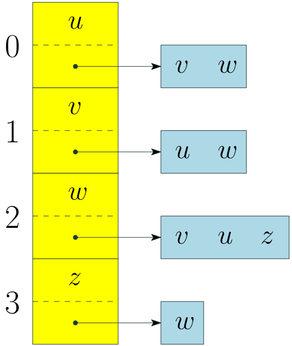
How to represent graphs

The popular ways to represent graphs:

1. Adjacency List
2. Adjacency Map
3. Adjacency Matrix

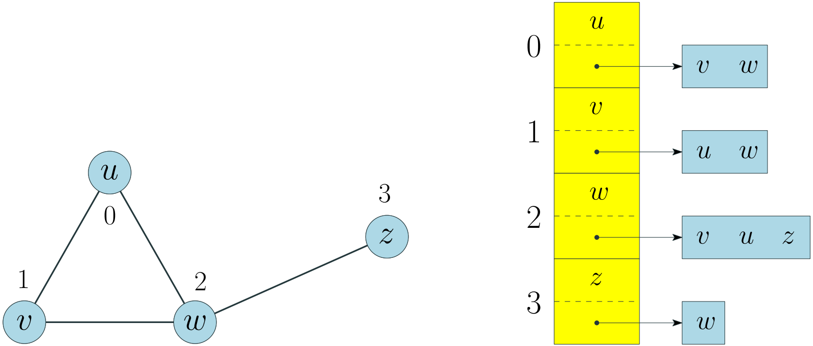
Adjacency list





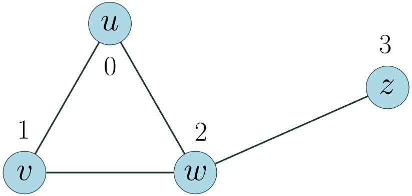
- Every vertex stores a list of its neighbors; implemented using an ArrayList or a linked-list.
- Verification for the existence of an edge $\{i, j\}$ in the graph runs slow; takes $O(n)$ time, where n is the number of vertices, since in the worst-case the whole neighbor list of vertex i may need to be searched to check if j is present in it
- Needs $O(m+n)$ storage space, where n is the number of vertices and m is the number of edges, since the vertical array takes up $O(n)$ space and the total space taken by the n lists is $O(m)$

Adjacency map



- Same as adjacency lists except that hashsets or treesets are used to maintain the set of neighbors
- Edge verification takes $O(\log n)$ time if treesets are used; $O(1)$ time on average if hash-sets are used
- Hashsets are popularly used for speedy real-world performance

Adjacency matrix



	0	1	2	3
$u : 0$	0	1	1	0
$v : 1$	1	0	1	0
$w : 2$	1	1	0	1
$z : 3$	0	0	1	0

- A $n \times n$ boolean matrix is used, where n is the number of vertices in the graph
- The cell i, j contains 1 if there is an edge between the vertices i and j, otherwise contains a 0
- Edge verification takes $O(1)$ time since we can directly peek into the cell i, j
- Downside: takes up $O(n^2)$ space since the matrix has $n^2 = O(n^2)$ cells
- For large graphs, adjacency matrices are impractical

How to traverse a graph?

- Breadth-first traversal (BFS)
- Depth-first traversal (DFS)

BFS

- Start at any given vertex, say s and visit it
- Visit the vertices that can be reached from s using exactly one edge
- Visit the vertices that can be reached from s using exactly two edges
- Visit the vertices that can be reached from s using exactly three edges

$\vdots$

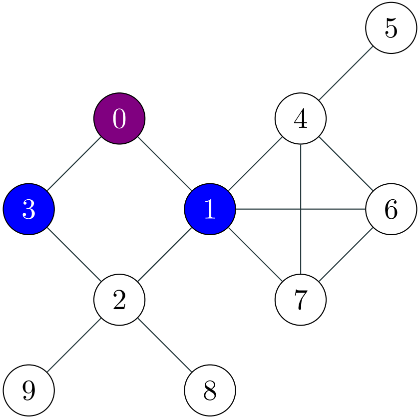
This can be beautifully implemented using a **queue**!

BFS (non-recursive/iterative)

BFS(s), where s is the start vertex

```
Declare a boolean array named discovered of size n;
for every vertex v in the graph do
    discovered[v] = false;
Declare an empty queue Q of vertices;
Q.enqueue(s);
discovered[s] = true;
while Q is not empty do
    u = Q.dequeue();
    VISIT vertex u;
    for every neighbor v of u do
        if discovered[v] == false then
            discovered[v] = true;
            v.parent = u; // since v is just now discovered from u
            Q.enqueue(v);
```

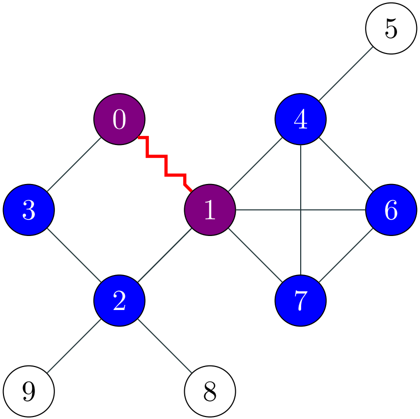
Example



$$Q : 1\ 3$$

BFS traversal sequence: 0

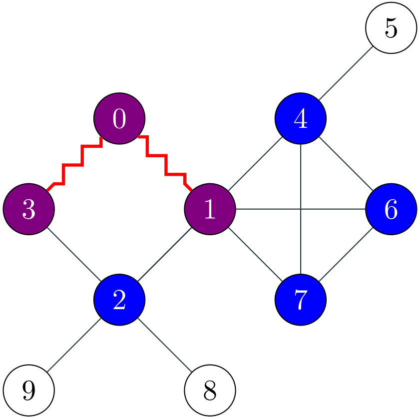
Example



$$Q : 3\ 2\ 4\ 6\ 7$$

BFS traversal sequence: $0, 1$

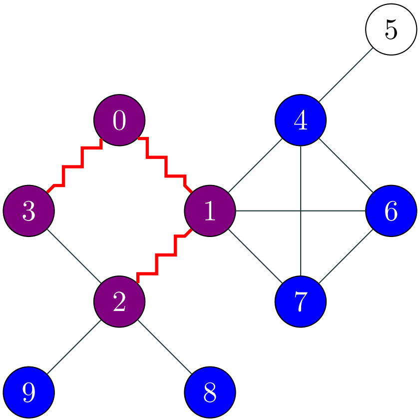
Example



$$Q : 2\ 4\ 6\ 7$$

BFS traversal sequence: $0, 1, 3$

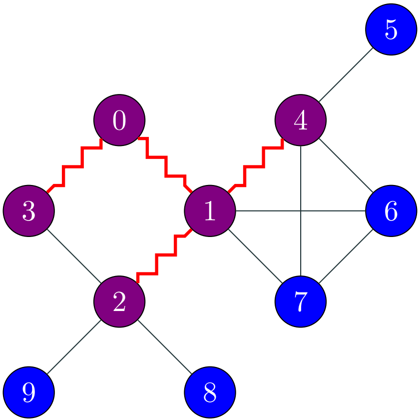
Example



$$Q : 4\ 6\ 7\ 8\ 9$$

BFS traversal sequence: $0, 1, 3, 2$

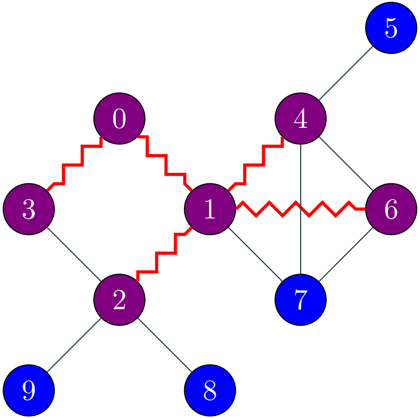
Example



$$Q : 6\ 7\ 8\ 9\ 5$$

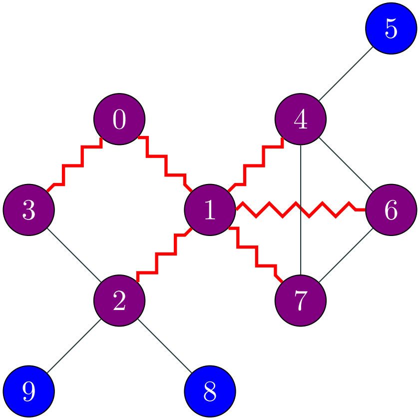
BFS traversal sequence: $0, 1, 3, 2, 4$

Example



$$Q : 7\ 8\ 9\ 5$$

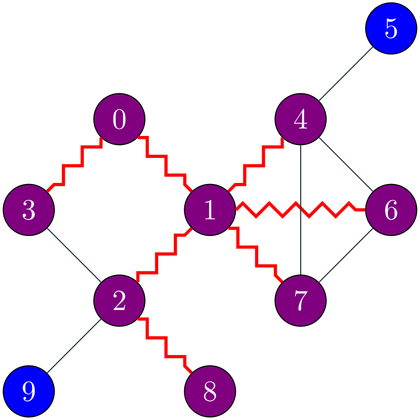
BFS traversal sequence: $0, 1, 3, 2, 4, 6$



$$Q : 8\ 9\ 5$$

BFS traversal sequence: $0, 1, 3, 2, 4, 6, 7$

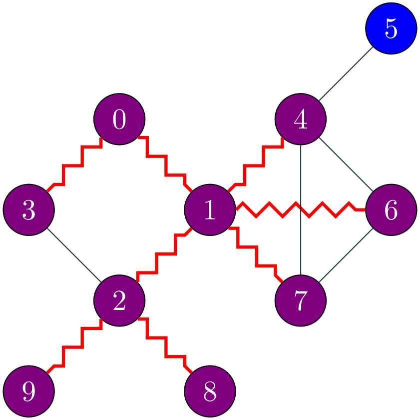
Example



$$Q : 9\ 5$$

BFS traversal sequence: $0, 1, 3, 2, 4, 6, 7, 8$

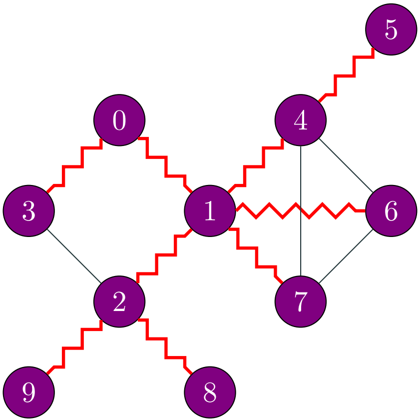
Example



$$Q : 5$$

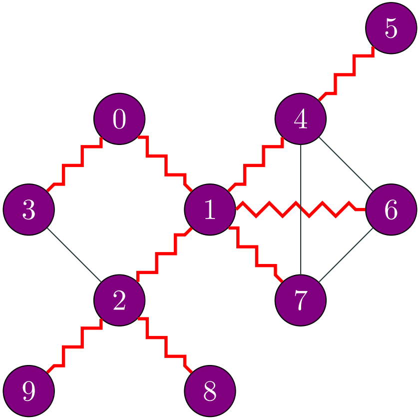
BFS traversal sequence: $0, 1, 3, 2, 4, 6, 7, 8, 9$

Example



$$Q : \emptyset$$

BFS traversal sequence: $0, 1, 3, 2, 4, 6, 7, 8, 9, 5$



BFS traversal sequence: $0, 1, 3, 2, 4, 6, 7, 8, 9, 5$

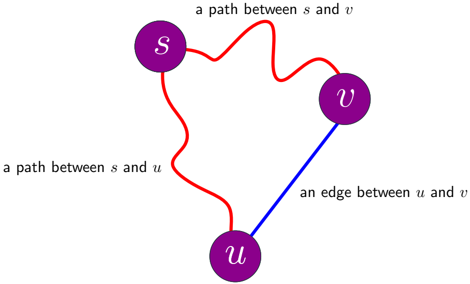
When we visit a vertex u, we put a squiggly edge with u as one of its two vertices and v as the other vertex if u was discovered for the first time from v

Observations about BFS

- If the edges are unweighted, the **BFS tree** (shown using **red** squiggly edges) rooted at the start vertex s contains shortest paths from the start vertex to every other vertex; the shortest path between s and another vertex v is simply the unique path between them in the BFS tree
- If the graph is **disconnected** (made up of multiple components), we get a collection of BFS trees, one for every component
- A graph is **connected** if and only if we get exactly one BFS tree in the end
- Takes $O(n+m)$ time, where n is the number of vertices and m is the number of edges; this includes $O(n)$ time spent for maintaining the queue. Space complexity: $O(n)$ (for maintaining the queue and the `discovered` array)

Observations about BFS

- BFS traversal sequence is **not unique** even if we fix the start vertex; depends on the order we enqueue the vertices
- A cycle is a path that starts from a given vertex and ends at the same vertex. BFS can be used to check **if there is a cycle** in the input graph: *a graph has at least one cycle if there is a neighbor v of a visited vertex u such that v was visited before u*



The path between s to u plus the edge between u, v plus the path between v, s form a cycle

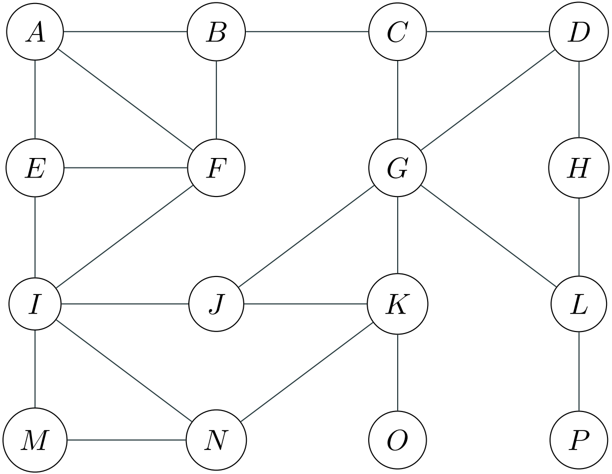
Go deep as much as you can,
then when stuck/nothing new to discover, back up

DFS(u)

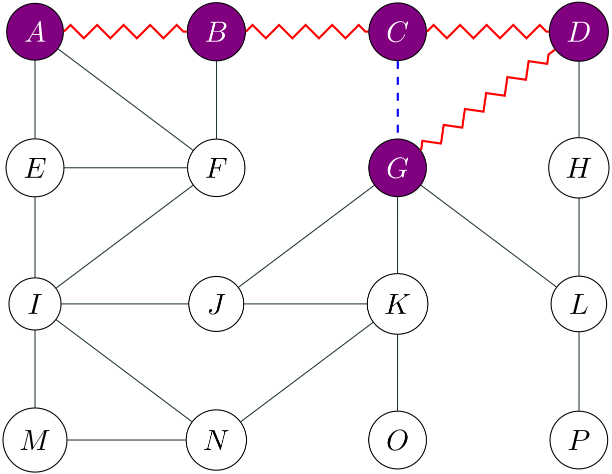
1. **VISIT** vertex u and mark it **`visited`**;
2. **For** each of u's neighbor v
   - If v was not **`visited`** before, recursively call DFS(v) and set v's parent to u;

Invoke the above algorithm using **DFS**(s), where s is the start vertex

Example

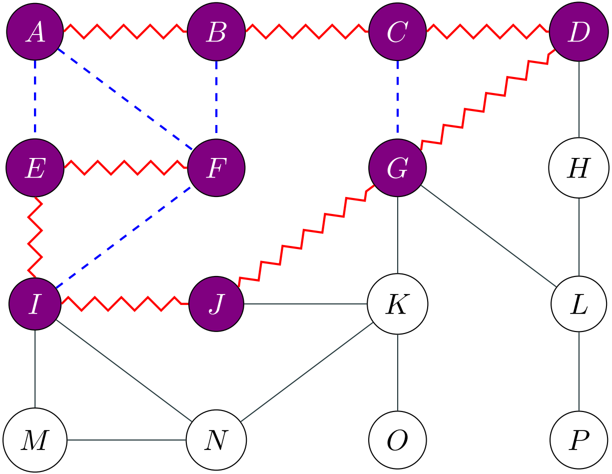


Example



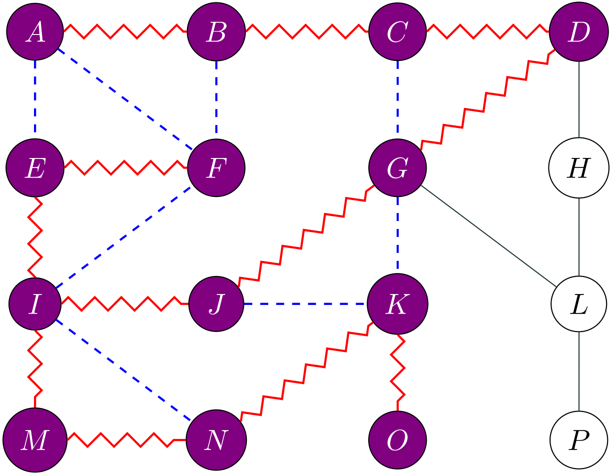
DFS traversal sequence : A, B, C, D, G

Example



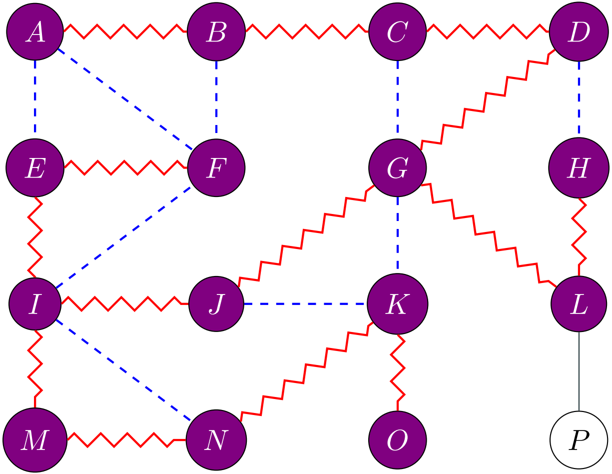
DFS traversal sequence : $A, B, C, D, G, J, I, E, F$

Example



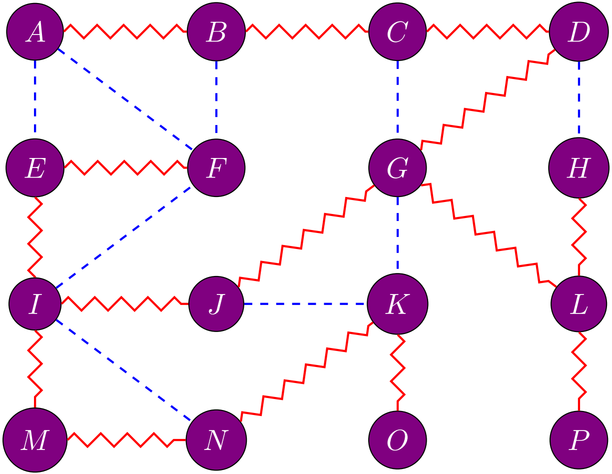
DFS traversal sequence : $A, B, C, D, G, J, I, E, F, M, N, K, O$

Example



DFS traversal sequence : $A, B, C, D, G, J, I, E, F, M, N, K, O, L, H$

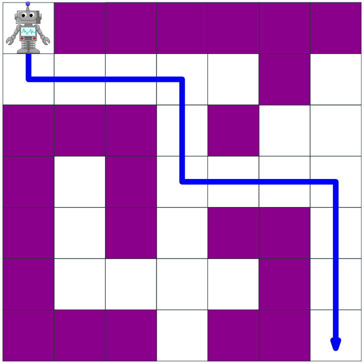
Example

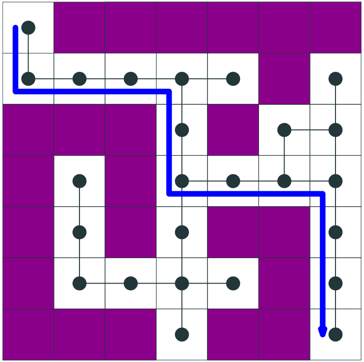


DFS traversal sequence : $A, B, C, D, G, J, I, E, F, M, N, K, O, L, H, P$

Observations about DFS

- We always get a DFS tree (shown using red squiggly edges) at the end (not guaranteed to be a shortest-paths tree like BFS!)
- If the graph is disconnected, we get a collection of DFS trees; one for every component
- If we get only one DFS tree in the end, the graph is connected
- Takes $O(n+m)$ time where n is the number of vertices and m is the number of edges. Space complexity: $O(n)$ for maintaining the stack for recursive calls
- DFS traversal sequence is not unique even if we fix the start vertex; depends on the order we consider the neighbors of every vertex
- Just like BFS, DFS can also be used to check if there is a cycle
- Just like BFS, DFS can also be used to find the connected components

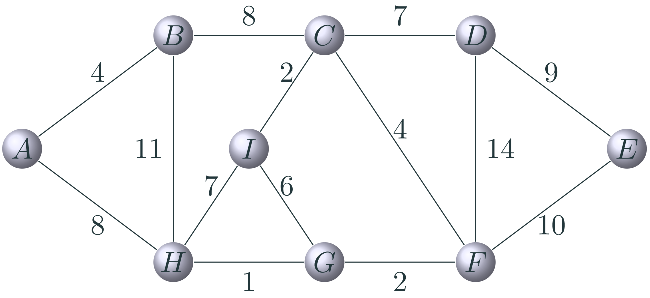




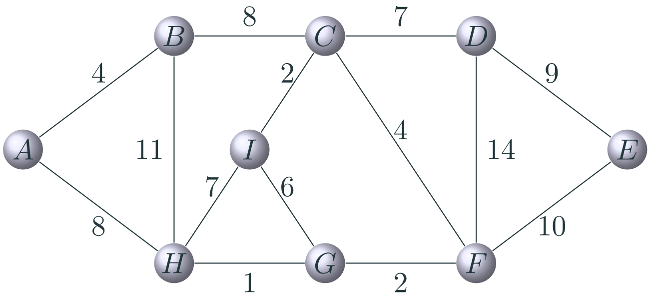
Treat every empty cell as a vertex; two neighbouring empty cells has an edge between them; then run BFS to get a shortest path to the exit (DFS works but shortest path is not guaranteed)

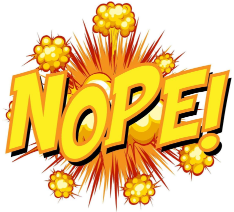
The problem

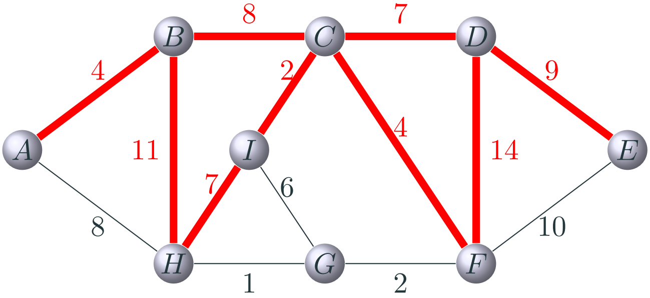
Given an **weighted** (every edge has an weight) **undirected** graph G, find a tree on the vertices of G that has the minimum weight (sum of weight of the edges)



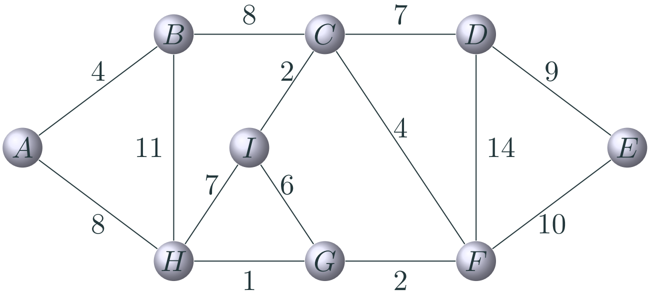
Is this even a candidate solution?

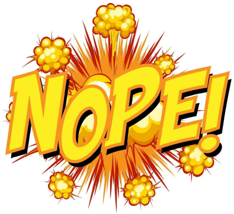


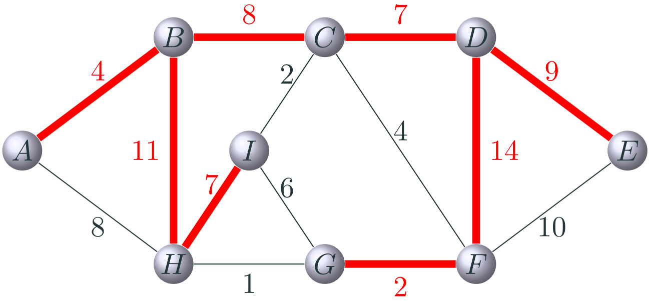




How about this one?



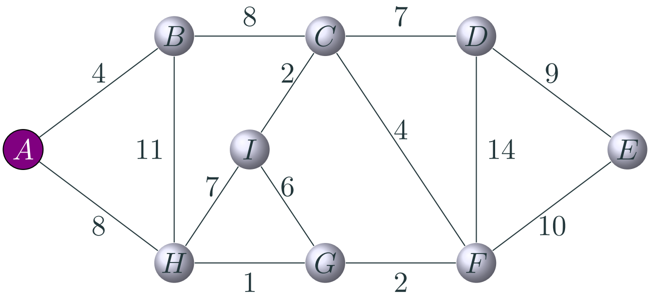




The algorithm

1. Choose any vertex s as the starting vertex;
2. $X = \{s\}$;
3. Let T be an empty set of edges;
4. **while** there is an edge (v, w) with $v \in X, w \notin X$ do
   1. find such an edge (v^*, w^*) having the **minimum cost**;
   2. insert the vertex w^* into X; // w^* was not in X
   3. insert the edge (v^*, w^*) into T;
5. **return** T;

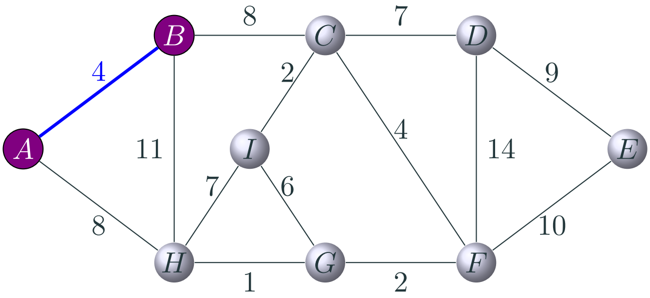
Example



$$X = \{A\}$$

$$T = \emptyset$$

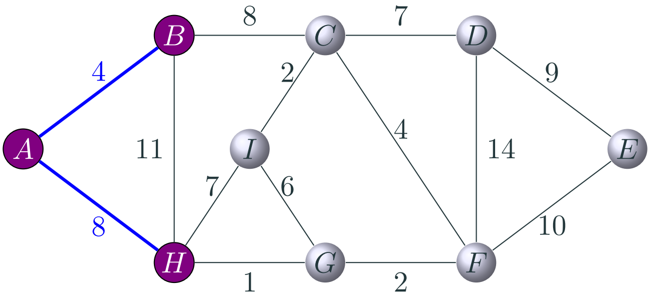
Example



$$X = \{A, B\}$$

$$T = \{(A, B)\}$$

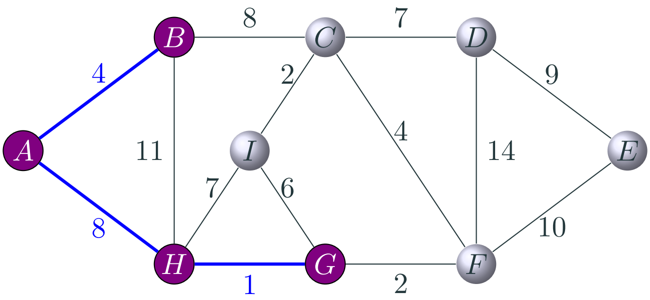
Example



$$X = \{A, B, H\}$$

$$T = \{(A, B), (A, H)\}$$

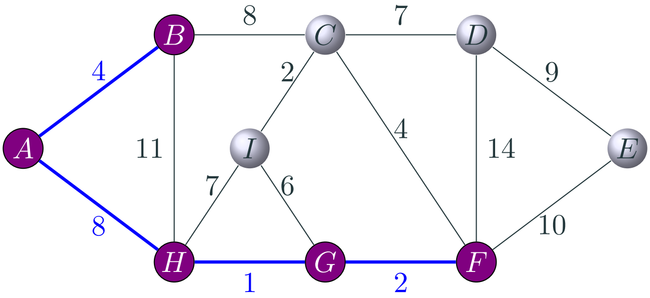
Example



$$X = \{A, B, H, G\}$$

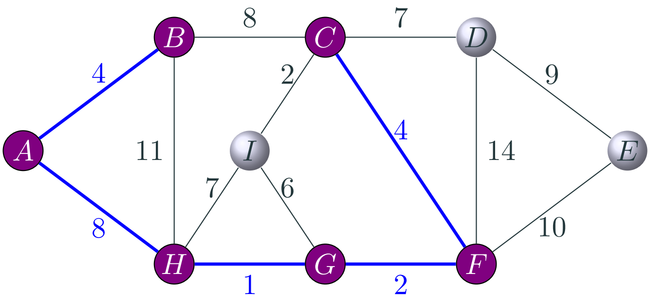
$$T = \{(A, B), (A, H), (H, G)\}$$

Example



$$X = \{A, B, H, G, F\}$$

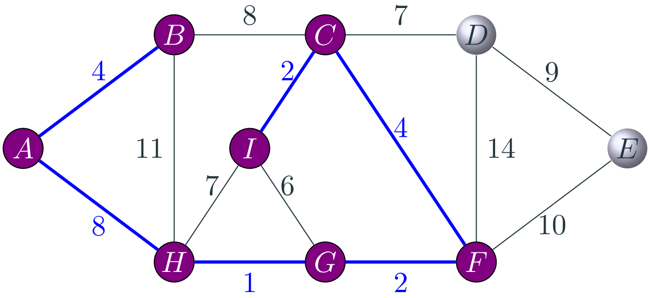
$$T = \{(A, B), (A, H), (H, G), (G, F)\}$$



$$X = \{A, B, H, G, F, C\}$$

$$T = \{(A, B), (A, H), (H, G), (G, F), (F, C)\}$$

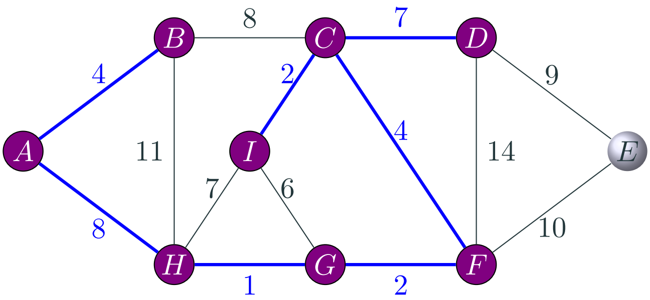
Example



$$X = \{A, B, H, G, F, C, I\}$$

$$T = \{(A, B), (A, H), (H, G), (G, F), (F, C), (C, I)\}$$

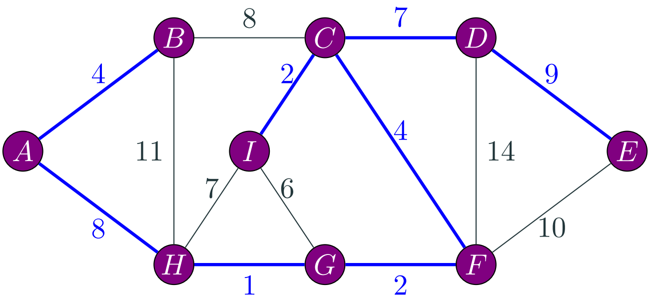
Example



$$X = \{A, B, H, G, F, C, I, D\}$$

$$T = \{(A, B), (A, H), (H, G), (G, F), (F, C), (C, I), (C, D)\}$$

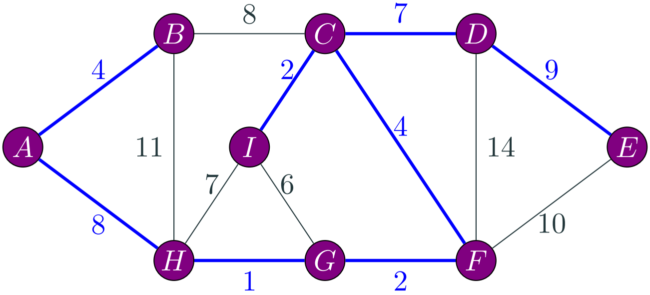
Example



$$X = \{A, B, H, G, F, C, I, D, E\}$$

$$T = \{(A, B), (A, H), (H, G), (G, F), (F, C), (C, I), (C, D), (D, E)\}$$

Example



Weight of the above MST: 37 (sum of the bold blue edge weights)

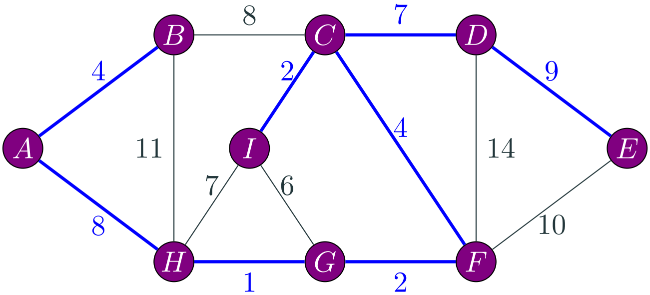
$T = \{(A,B),(A,H),(H,G),(G,F),(F,C),(C,I),(C,D),(D,E)\}$

Note that MSTs **are not** unique!

Time complexity

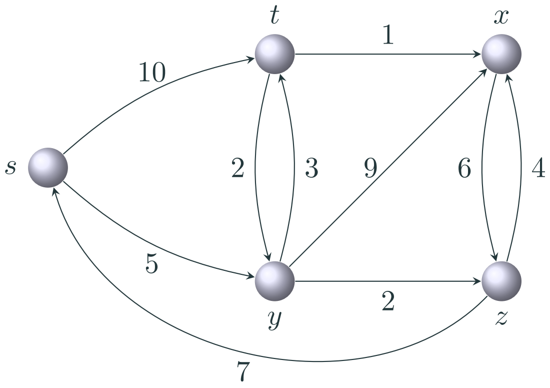
Using a priority queue, Prim's algorithm can be implemented to run in $O(m \log n)$ time, where n is the number of vertices and m is the number of edges

Space complexity: $O(n)$



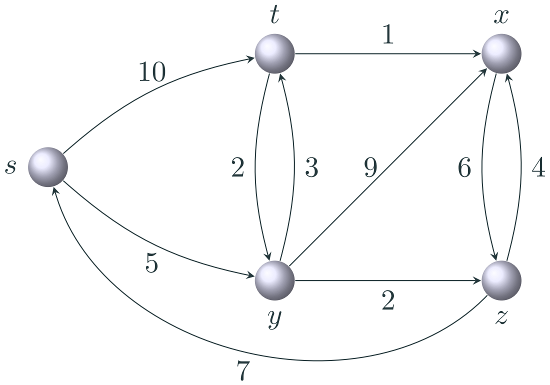
Given a road-network plan on n cities, which roads should you build so that the overall cost is minimized and the n cities remain connected with each other.

- Edges have directions
- Edges may also have weights



The problem

Given an **weighted directed graph**, find the shortest paths from a given source vertex s to all the other vertices in the graph

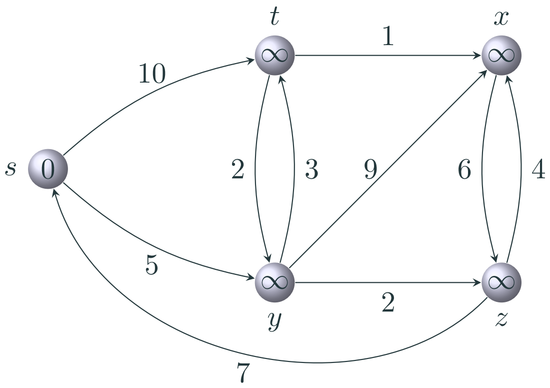


Dijkstra's algorithm

Pseudocode

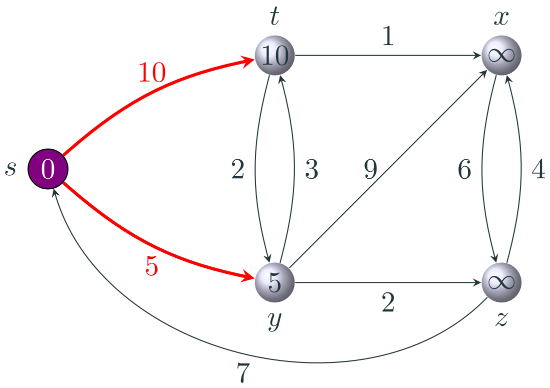
1. For every vertex v except s, set $d[v]$ to ∞;
2. $d[s] = 0$;
3. Create a container of vertices C and put all the vertices in it;
4. **while** C is not empty
   1. extract (return and remove) the vertex u in C that has the minimum d-value;
   2. **for** every neighbour v of u
      1. if $d[v] > d[u] + w(u, v)$ then $[v] = d[u] + w(u, v)$ and set the predecessor of v to u;

☞ $d[v]$ stores the weight of the best path found so far from s to v. When the algorithm is over, all the d-values contain the length of the shortest paths



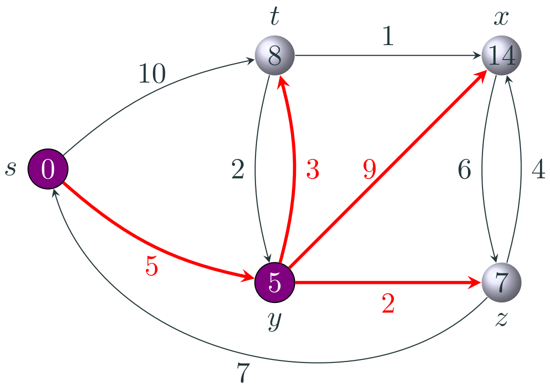
VERTEX	s	t	x	y	z
d	0	∞	∞	∞	∞
PREDECESSOR					

Dijkstra's algorithm

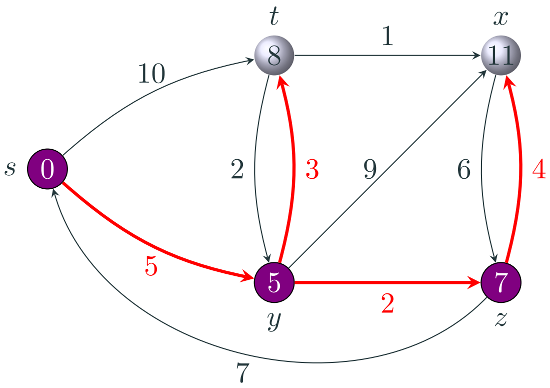


Vertex	s	t	x	y	z
d	0	10	∞	5	∞
Predecessor		s		s	

Dijkstra's algorithm

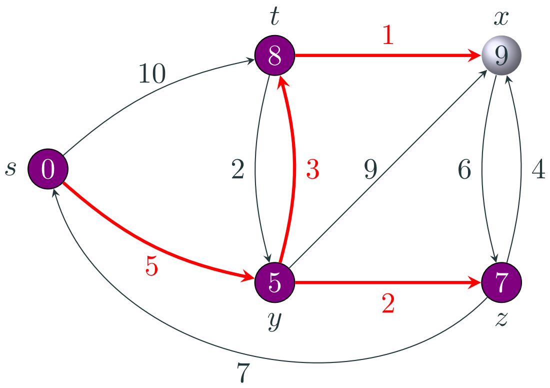


VERTEX	s	t	x	y	z
d	0	8	14	5	7
PREDECESSOR		y	y	s	y



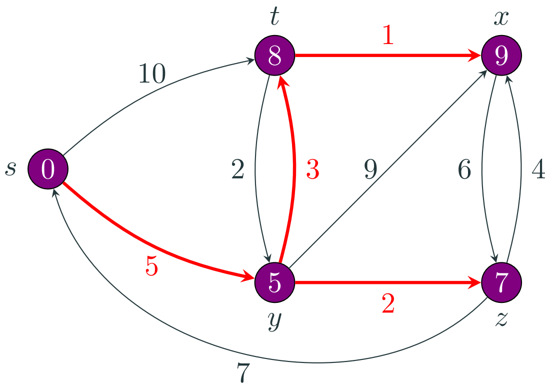
VERTEX	s	t	x	y	z
d	0	8	11	5	7
PREDECESSOR		y	z	s	y

Dijkstra's algorithm



Vertex	s	t	x	y	z
d	0	8	9	5	7
Predecessor		y	t	s	y

Dijkstra's algorithm



VERTEX	s	t	x	y	z
d	0	8	9	5	7
PREDECESSOR		y	t	s	y

Dijkstra's algorithm

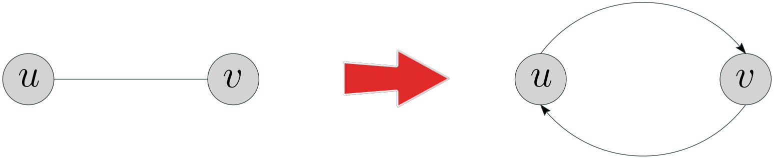
Input: an weighted directed graph and source vertex s

Pseudocode

1. For every vertex v except s, set $d[v]$ to ∞;
2. $d[s] = 0$;
3. Create a container of vertices C and put all the vertices in it;
4. **while** C is not empty
    1. extract (return and remove) the vertex u in C that has the minimum d-value;
    2. **for** every neighbour v of u
        1. if $d[v] > d[u] + w(u, v)$ then $[v] = d[u] + w(u, v)$ and set the predecessor of v to u;

If C is implemented as a plain array, runtime of this algorithm is $O(n^2)$ where n is the number of vertices. If C is implemented as heap, the runtime is $O((m+n) \log n)$ where m is the number of edges. Space complexity: $O(n)$.

Observation



Every undirected graph can be perceived as a directed graph. Replace every undirected edge $\{u, v\}$ in the undirected graph with two directed edges (u, v) and (v, u).

After this transformation, the Dijkstra's algorithm can be used to find shortest paths for the undirected graph

Reference

Chapter 19 from

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html