FIT2004 Week 8 Lecture

Introduction to Graphs and Shortest Path Algorithms

1. Use of Graphs

- Graphs are extremely useful for modelling. For example:
  - Transportation networks: map of routes of an airline, rail network, ...
  - Communication networks: the connections between different Internet service providers, wireless ad-hoc networks, ...
  - Information networks: the connections between different webpages using links (Google PageRank algorithm for determining the relative importance of each website), ...
  - Social networks: the persons could be the nodes and the edges represent friendship (Facebook), or the nodes could represent companies and persons, and the edges financial relationships among them, etc. Properties of the graphs representing social networks are often used to find influencers, target ads,...
  - Dependency networks: prerequisites in a course map

2. Graph Formal Notations

- A graph G = (V, E) is defined using a set of vertices V and a set of edges E.
- An edge e is represented as e = (u, v) where u and v are two vertices
- For undirected graphs, (u, v) = (v, u) because there is no sense of direction. For a directed graph, (u, v) represents an edge **from** u **to** v and (u, v) ≠ (v, u).
- We will slightly abuse notation and use V (instead of |V|) for the number of vertices and E (instead of |E|) for the number of edges when what is meant is clear from the context.
- A weighted graph is represented as G = (V, E) and each edge (u, v) has an associated weight w.
- A graph is called a **simple graph** if it does not have loops AND does not contain multiple edges between same pair of vertices.
- **In this unit, we focus on simple graphs with a finite number of vertices.**

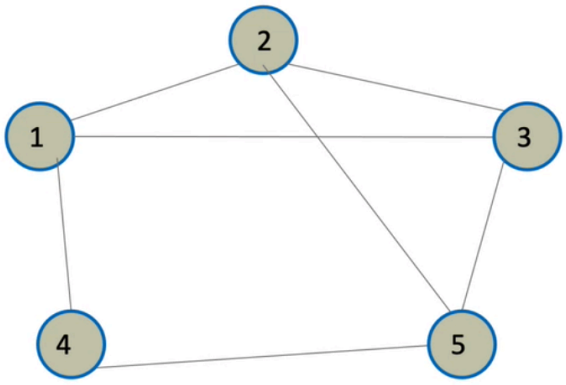
3. Graph --- Connected Components

- A vertex v is **reachable** from u if there is a path in the graph that starts in u and ends v.
- In an undirected graph, reachability is an equivalence relation:
  - Reflexive: each u node is reachable from itself.
  - Symmetric: if v is reachable from u, then u is reachable from v.
  - Transitive: if v is reachable from u, and u is reachable from w, then v is reachable from w.
- The set of vertices reachable from u defines the **connected component** of G containing u.

- For any two nodes u and v, their connected components are either identical or disjoint.

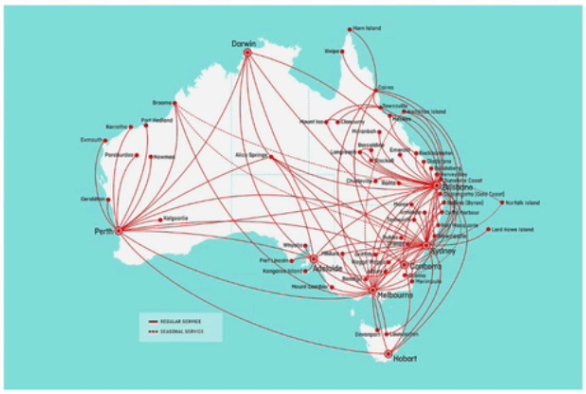
- An undirected graph is **connected** if all vertices are part of a single connected component.

- In other words, for any pair of vertices u and v, there is a path between them.



4. Connected Component --- Importance

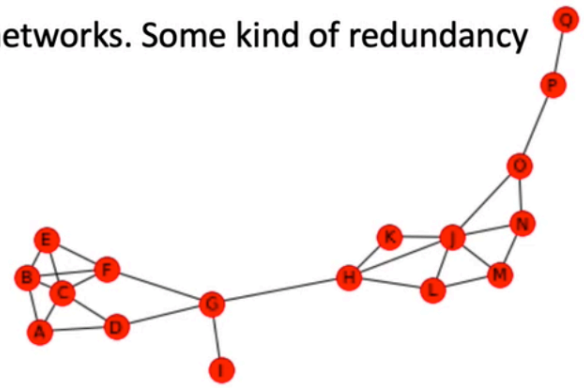
- Why is that an important concept in practice?
  - Example: Airlines normally want their air routes to form a connected graph.



  - Getting a connected graph is often an important consideration when designing communication and transportation networks.
- Hubs: Often it is not viable to have pairwise connections between all nodes; but one still wants to have paths, without many intermediary nodes, between every pair of nodes.

5. Redundancy or Not

- In this graph the edge (g, h) is quite critical as any problem in the network that eliminates this edge would break the connected graph into "large" disjoint connected components.
- This can be quite bad in networks. Some kind of redundancy is often desirable.



- Adding edges that join distinct connected components can sometimes also have bad consequences. E.g., quarantine measures often try to avoid a disease from reaching a disease-free connected component of a social network.

6. Some Graph Properties

Let G be a graph.

- The minimum number of edges in a connected undirected graph
  - V-1 = O(V)
- The maximum number edges in an undirected graph
  - $V(V - 1)/2 = O(V^2)$

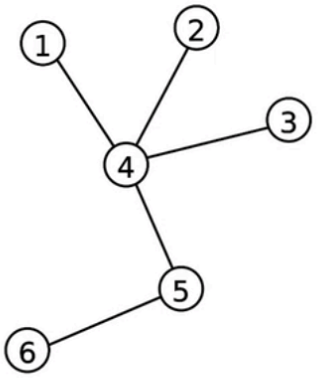
- A graph is called **sparse** if $E << V^2$ (<< means significantly smaller than)
- A graph is called **dense** if $E \approx V^2$

V(V-1)/2: each node, connected to all the other nodes [also for connected graph]

Sparse: 稀疏的　dense: 稠密的

7. Tree --- Definitions

- Let G=(V, E) be an undirected graph. G is a tree if it satisfies any of the following equivalent conditions:
  - G is connected and acyclic (i.e., contains no cycles).
  - G is connected and has V-1 edges.
  - G is acyclic and has V-1 edges.
  - G is acyclic, but a cycle is formed if any edge is added to G.
  - G is connected, but would become disconnected if any single edge is removed from G.



- In other words, if any of the above conditions is satisfied for an undirected graph G, then G is a tree (and all the other conditions will also hold).

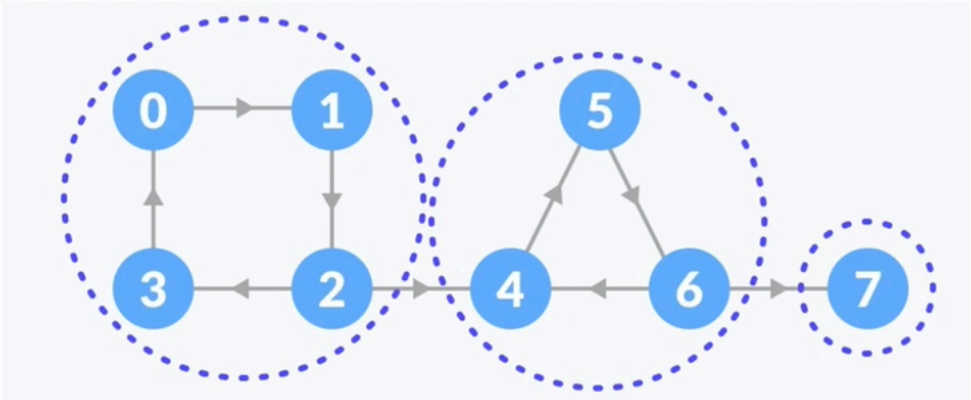
8. Mutually Reachable and Strongly-Connected Component

- In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric (i.e., possibly there could be a path from u to v, but no path from v to u).

- Vertices u and v are called **mutually reachable** if there are paths from u to v and from v to u.

- Mutual reachability is an equivalence relation and decomposes the graph into **strongly-connected components** (for any two vertices u and v, their strong components are either identical or disjoint).

- A directed graph with 3 strongly-connected components:



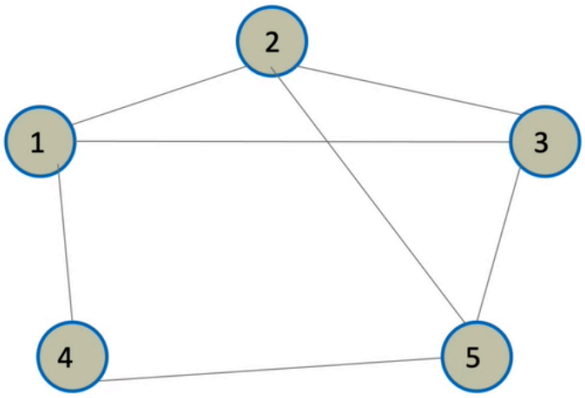
- A directed graph is **strongly connected** if for every pair of vertices u and v of G, there are paths from u to v and from v to u.

- I.e., the graph only has one strongly-connected component.

Adjacency Matrix (Undirected Graph):

Create a V x V matrix M and store T (true) for M[i][j] if there exists an edge between i-th and j-th vertex. Otherwise, store F (false).

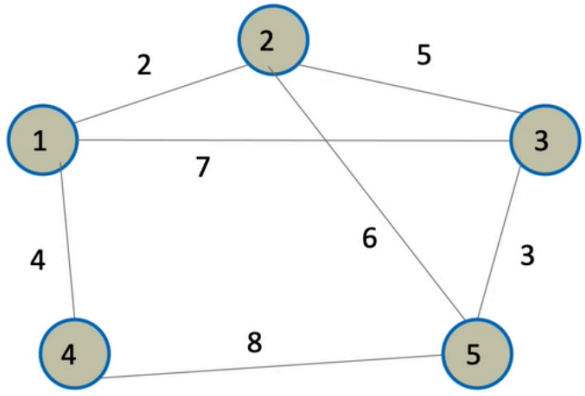
	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



Adjacency Matrix (Undirected Weighted Graph):

Create a V x V matrix M and store **weight** at M[i][j] only if there exists an edge **between** i-th and j-th vertex.

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



Adjacency Matrix (Directed Weighted Graph):

Create a V x V matrix M and store weight at M[i][j] only if there exists an edge **from** i-th **to** j-th vertex.

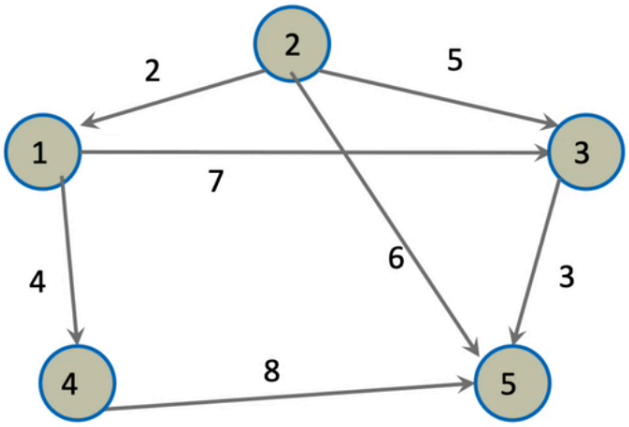
Space Complexity: $O(V^2)$ regardless of the number of edges

Time Complexity of checking if an edge exits: O(1)

Time Complexity of retrieving all neigbhbors (adjacent vertices) of a given vertex:

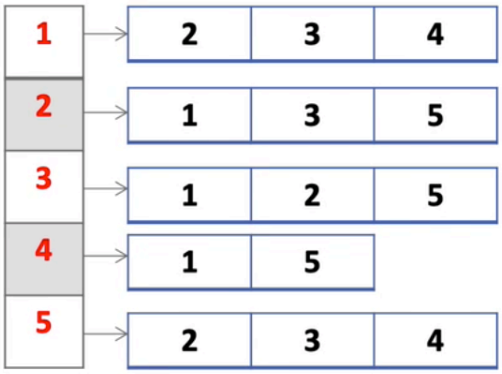
O(V) regardless of the number of neighbors (unless additional pointers are stored)

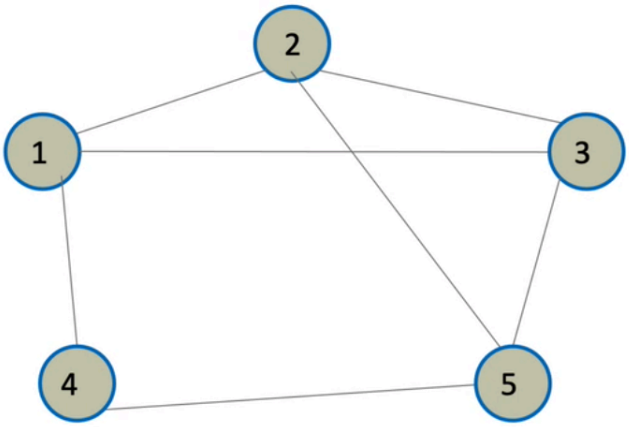
	1	2	3	4	5
1			7	4	
2	2		5		6
3					3
4					8
5					



Adjacency List (Undirected Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex.



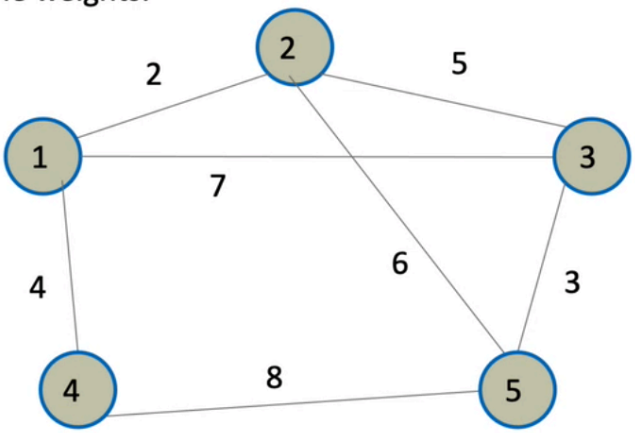


Adjacency List (Undirected Weighted Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex **along with the weights**.

The numbers in parenthesis correspond to the weights.

1	2 (2) → 3 (7) → 4 (4)
2	1 (2) → 3 (5) → 5 (6)
3	1 (7) → 2 (5) → 5 (3)
4	1 (4) → 5 (8)
5	2 (6) → 3 (3) → 4 (8)



Adjacency List (Directed Weighted Graph):

Create an array of size V. At each V[i], store the list of vertices adjacent to the i-th vertex **along with the weights**.

Space Complexity:

- O(V + E)

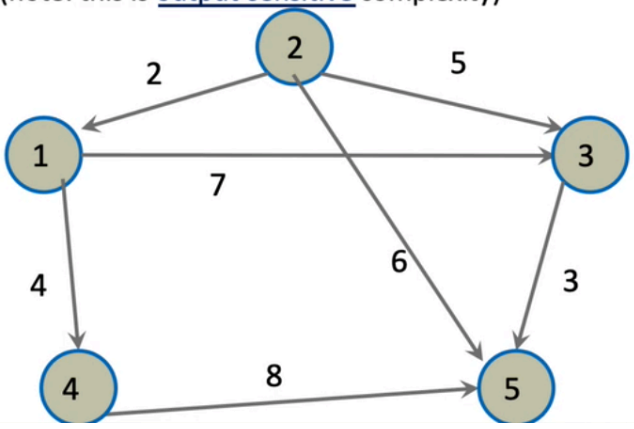
Time complexity of checking if a particular edge exists:

- O(log V) assuming each adjacency list is a sorted array on vertex IDs

Time complexity of retrieving all adjacent vertices of a given vertex:

- O(X) where X is the number of adjacent vertices (note: this is output-sensitive complexity)

1	3 (7) → 4 (4)
2	1 (2) → 3 (5) → 5 (6)
3	5 (3)
4	5 (8)
5	



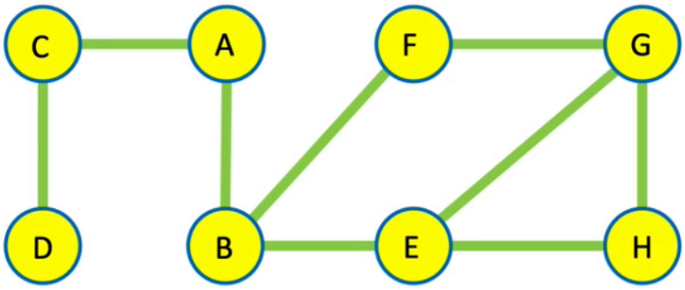
10. Graph Traversal

Graph traversal algorithms traverse (visit) all nodes of a graph.

They are very important in the design of numerous algorithms.

We will look into two algorithms that traverse a connected component from a graph starting from a source vertex:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)



Both of them visit the vertices exactly once.

They visit vertices in different orders.

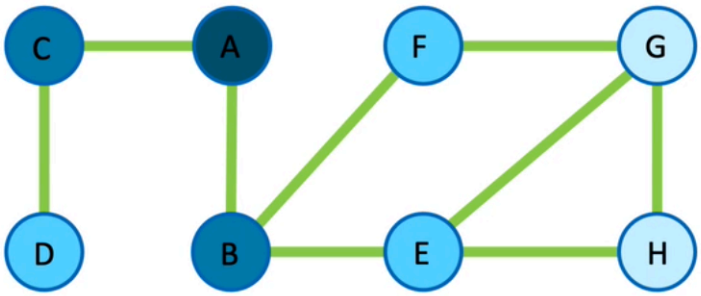
If a graph has more than one connected component, they can be repeatedly called (on unvisited nodes) until all graph nodes are marked as visited.

Each one has properties that makes it useful for certain kinds of graph problems.

11. BFS --- Breadth First Search

- Breadth-First Search (BFS)
  - Traverses the graph uniformly from the source vertex
  - i.e., all vertices that are k edges away from the source vertex are visited before all vertices that are k+1 edges away from source
  - In the graph below, if A is the source, then one possible BFS order is:
  - A, C, B, D, E, F, G, H

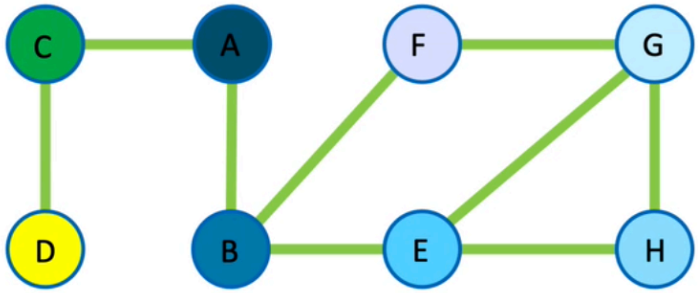
Is A, B, C, D, E, F, G, H a BFS Order?
Yes!



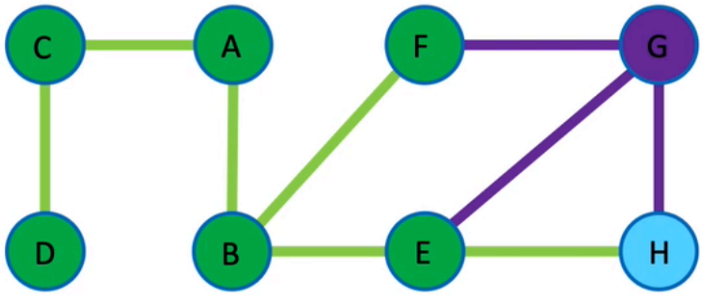
12. DFS --- Depth First Search

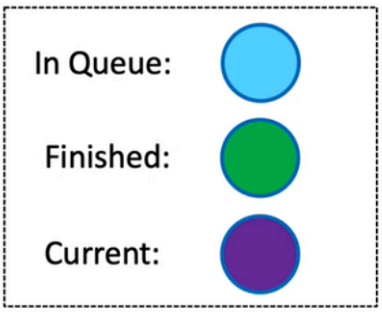
- Depth-First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: A, B, F, G, H, E, C, D

Is A, B, E, H, F, G, C, D as possible DFS order?
No!



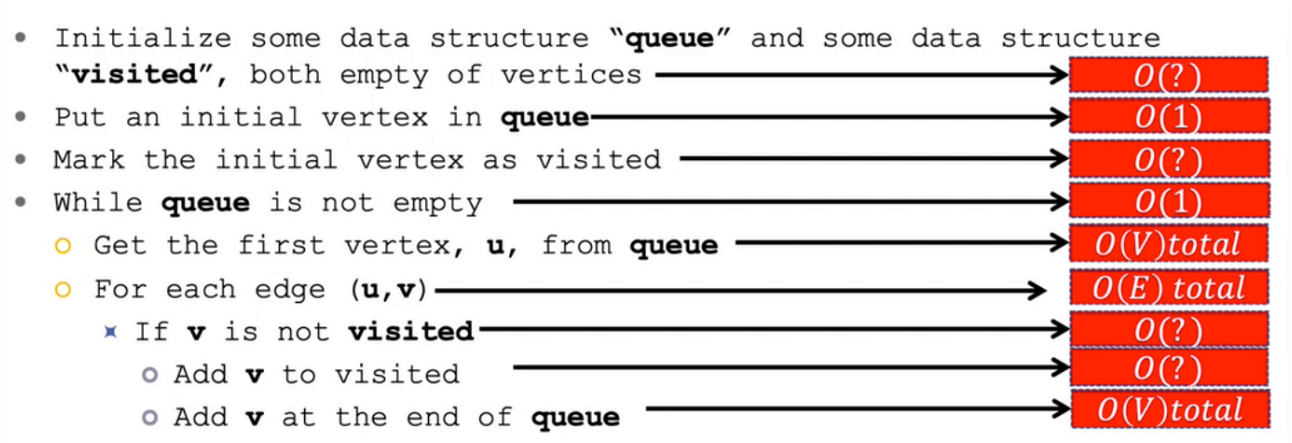
13. BFS --- Demo





Current:	G
Queue:	H
Finished:	A B C E F D

14. BFS Complexity Analysis



Assuming adjacency list representation.

Time Complexity:

- O(V*insert to visited + E*lookup on visited)
- Visited is just a bit list, indexed by vertex ID
- Lookup and insert are both O(1)

Assuming adjacency list representation.

Time Complexity:

- O(V * 1 + E * 1) = O(V+E)

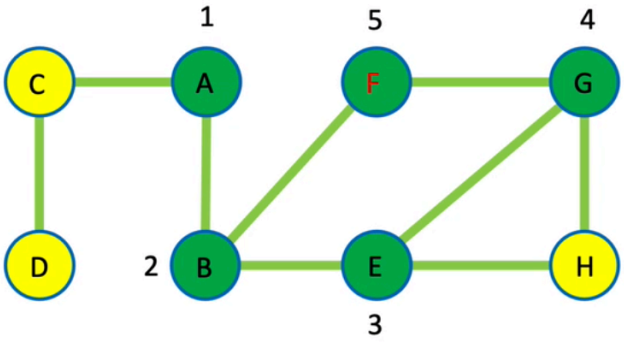
Space Complexity:

- O(V+E)

Algorithm 55 Generic breadth-first search

```
1:  function BFS(G = (V, E), s)
2:      visited[1..n] = false
3:      visited[s] = true
4:      queue = Queue()
5:      queue.push(s)
6:      while queue is not empty do
7:          u = queue.pop()
8:          for each vertex v adjacent to u do
9:              if not visited[v] then
10:                 visited[v] = true
11:                 queue.push(v)
```

15. DFS -- Depth First Search Analysis



Visited:

- F is a dead end
- Go back to the last active node (G)

Current: F

Visited:

A	B	C	D	E	F	G	H
1	1	0	0	1	1	1	0

Algorithm 52 Generic depth-first search

```
1:  // Driver function that calls DFS until everything has been visited
2:  function TRAVERSE(G = (V, E))
3:      visited[1..n] = false
4:      for each vertex u = 1 to n do
5:          if not visited[u] then
6:              DFS(u)
7:
8:  function DFS(u)
9:      visited[u] = true
10:     for each vertex v adjacent to u do
11:         if not visited[v] then
12:             DFS(v)
```

Assuming adjacency list representation.

Time Complexity:

- Each vertex visited at most once
- Each edge accessed at most twice (once when u is visited once when v is visited)
- Total cost: O(V+E)

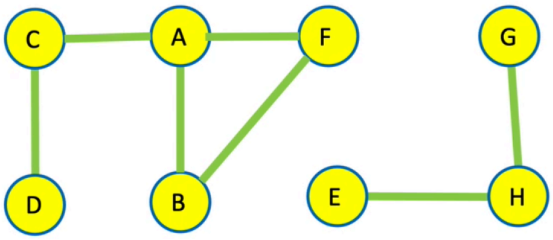
Space Complexity:

- O(V+E)

16. Applications of BFS and DFS

The algorithms we saw can also be applied on directed graphs.
BFS and DFS have a wide variety of applications:

- Reachability
- Finding all connected components
- Testing a graph for bipartiteness
- Finding cycles
- Topological sort (week 12)
- Shortest paths on unweighted graphs



17. Shortest Path Problem

Length of a path:

For unweighted graphs, the length of a path is the number of edges along the path.

For weighted graphs, the length of a path is the sum of weights of the edges along the path.

Single source, single target:

Given a source vertex s and a target vertex t, return the shortest path from s to t.

Single source, all targets:

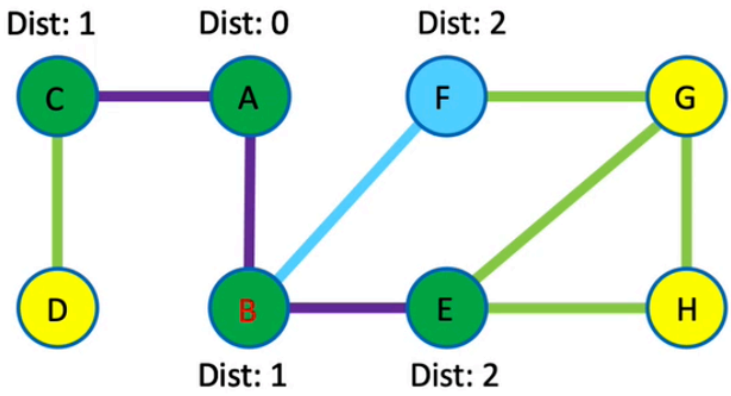
Given a source vertex s, return the shortest paths to every other vertex in the graph.

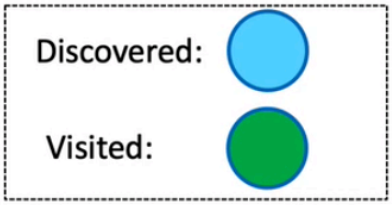
We will focus on single source, all targets problem because the single source, single target problem is subsumed by it.

18. Shortest Path Algorithms

- Breadth-First Search – (Single source, unweighted graphs)
- Dijkstra's Algorithm – (Single Source, weighted graphs with only non-negative weights)
- Bellman-Ford Algorithm – (Single source, weighted graphs including negative weights)
- Floyd-Warshall Algorithm– (All pairs, weighted graphs including negative weights)

19. BFS ---> Single Source, Unweighted Graphs





Current: B

Queue: C E

Visited:

A	B	C	D	E	F	G	H
1	1	1	0	1	0	0	0

Algorithm 56 Single-source shortest paths in an unweighted graph

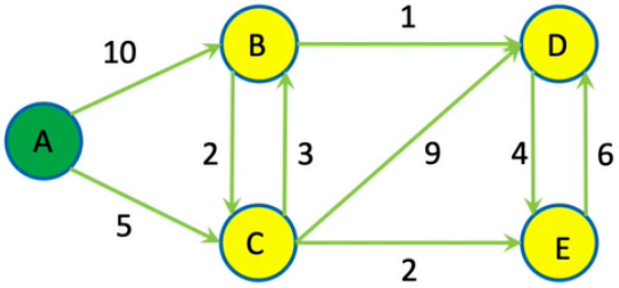
```
1: function BFS(G = (V, E), s)
2:     dist[1..n] = ∞
3:     pred[1..n] = null
4:     queue = Queue()
5:     queue.push(s)
6:     dist[s] = 0
7:     while queue is not empty do
8:         u = queue.pop()
9:         for each vertex v adjacent to u do
10:            if dist[v] = ∞ then
11:                dist[v] = dist[u] + 1
12:                pred[v] = u
13:                queue.push(v)
```

- Note that distances are stored in an O(1) lookup structure.
- Distances are set by lookup at the distance of the current vertex and adding 1.
- Path from s to v can be found by backtracking from v to s using the array pred.
- Complexity is the same as regular BFS, O(V+E).

20. Dijkstra's Algorithm ---> Weighted Graphs Without Negative Weights

- Algorithm for solving the single source, all targets shortest path problem on graphs with non-negative weights. Closely related to BFS.
- It keeps track of a set S of nodes whose distance to the source has already been determined.
- Initially S only contains the source node, and it grows by one element at a time until containing all nodes that are reachable from the source node.
- At each iteration, from all nodes that are one edge away from S, add to S the node u that has the smallest distance to the source.
- By doing so, the overall effect is that the nodes will be added to S in increasing order of distance to the source (as we will see soon).

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

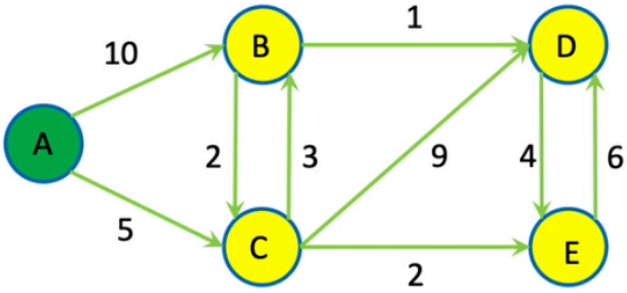
Dist:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Q is a priority queue, where priority is based on distance

Pred and Dist are the usual ID-indexed arrays

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

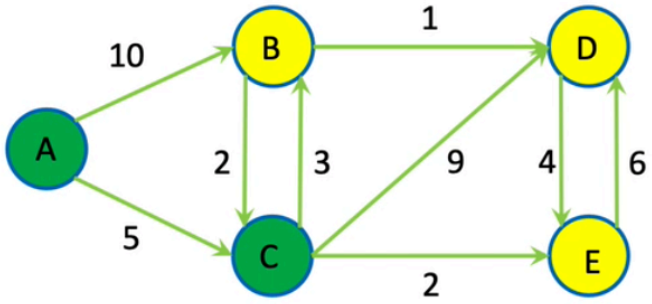
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest dist
- The dist of this vertex is now finalised

u: C



Q:

E	B	D
7	8	14

- Done with C

Pred:

A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Algorithm 61 Dijkstra's algorithm

```
1:  function DIJKSTRA(G = (V, E), s)
2:      dist[1..n] = ∞
3:      pred[1..n] = 0
4:      dist[s] = 0
5:      Q = priority_queue(V[1..n], key(v) = dist[v])
6:      while Q is not empty do
7:          u = Q.pop_min()
8:          for each edge e that is adjacent to u do
9:              // Priority queue keys must be updated if relax improves a distance estimate!
10:             RELAX(e)
11:     return dist[1..n], pred[1..n]
```

Time Complexity:

- Each edge visited once → O(E)
- Relaxation is O(1) since we can find distances and compare them in O(1)
- Updating the priority queue: depends on implementation
- While loop executes O(V) times
  - Find the vertex with smallest distance: depends on priortiy queue implementation
- Total cost: O(E*Q.decrease_key + V*Q.extract_min)

While Loop: V times, each time need to extract the current min list from the queue, i.e., pop()

Each edge visited once, could possibly cause changes in the Queue, i.e., insert()

Required additional structure:

- Create an array called Vertices.
- Vertices[i] will record the location of i-th vertex in the min-heap

Updating the distance of a vertex v in min-heap in O(log V)

- Find the location in the queue (heap) in O(1) using Vertices
- Now do the normal heap-up operation in O(log V)
  - For each swap performed between two vertices x and y during the upHeap
    - Update Vertices[x] and Vertices[y] to record their updated locations in the min-heap

Time Complexity:

- Each edge visited once → O(E)
- Relaxation is O(1) since we can find distances and compare them in O(1)
- Updating the priority queue: O(logV)
- While loop executes O(V) times
  - Find the vertex with smallest distance: O(1)

- Total cost: O(E*Q.decrease_key + V*Q.extract_min)
- Total cost: O(E*logV+ V*logV)

- If the graph is connected, minimum value of E is V-1
- E dominates V

- Total cost O(ElogV)

- O(E log V)
- For dense graphs, $E \approx V^2$
  - O(E log V) → O(V^2 log V) for dense graphs

Dijkstra's using a Fibonacci Heap (not covered in this unit)

- O(E + V log V)
- For dense graphs, $E \approx V^2$
  - O(E + V log V) → $O(V^2)$ for dense graphs

21. Proof of Correctness

Claim: For every vertex v which has been removed from the queue, dist[v] is correct

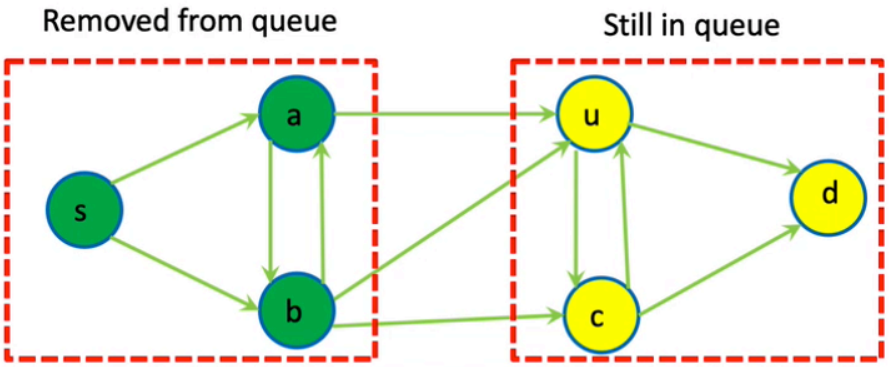
- Notation:
  - V is the set of vertices
  - Q is the set of vertices in the queue
  - S = V / Q = the set of vertices who have been removed from the queue

Base Case

- dist[s] is initialised to 0, which is the shortest distance from s to s (since there are no negative weights)

Inductive Step:

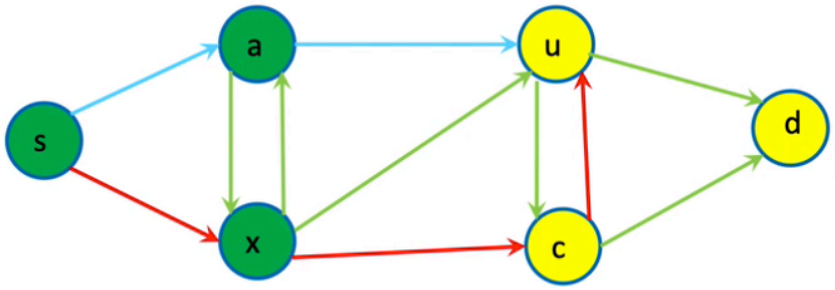
- Assume that the claim holds for all vertices which have been removed from the queue (S)
- Let u be the next vertex which is removed from the queue
- We will show that dist[u] is correct



Inductive Step:

- Suppose (for contradiction) there is a shortest path P, s⇝u with len(P) < dist[u]
- Let x be the furthest vertex on P which is in S (i.e. has been finalised)
- By the inductive hypothesis, dist[x] is correct (since it is in S)

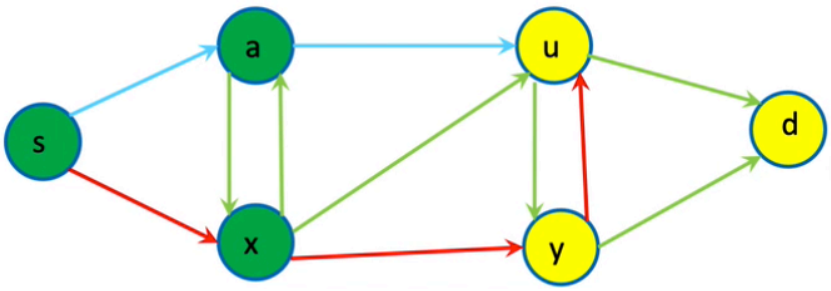
Current path
Assumed shorter path (P)



Inductive Step:

- By the inductive hypothesis, dist[x] is correct (since it is in S)
- Let y be the next vertex on P after x
- len(P) < dist[u] (by assumption)
- Edge weights are non-negative
- len(s⇝y) <= len(P) < dist[u]

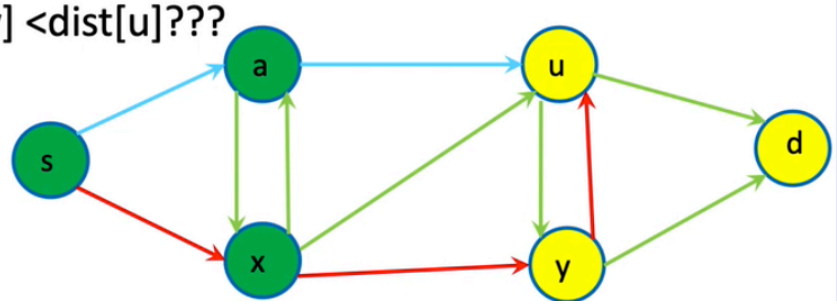
Current path
Assumed shorter path (P)



Inductive Step:

- len(s⇝y) <= len(P) < dist[u]
- Since we said that P (via x and y) is a shortest path…
- dist[y] = len(s⇝y) < dist[u]
- So dist[y] < dist[u]…
- If y ≠ u, why didn't y get removed before u???
- If y = u, how can dist[y] <dist[u]???

Current path
Assumed shorter path (P)



Why two cases: Just suppose Y is the next node on P next to X, so Y could be U

Inductive Step:

- Having obtained a contradiction, we can negate our assumption, namely:
- "Suppose (for contradiction) there is a shorter path P, s⇝u with len(P) < dist[u]."
- So there is no such path and dist[u] is correct.
- So by induction, the distance of every vertex is correct when Dijkstra's algorithm terminates.

Current path
Assumed shorter path (P)