Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 8: Introduction to Graphs and Shortest Path Algorithms

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Announcements

- Assignment 3 was released
  - Due: 10 May 2019 23:55:00 AEST

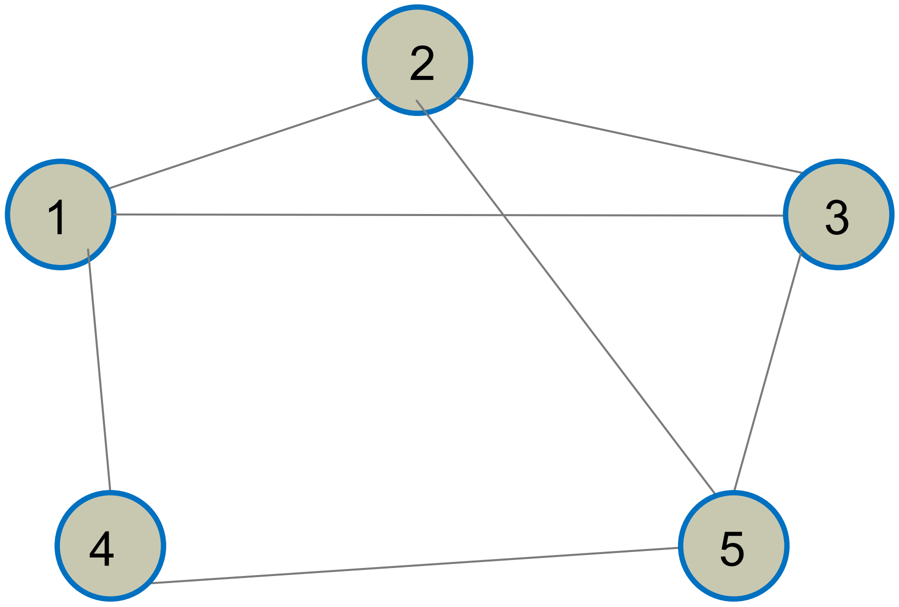
Recommended reading

- Unit notes: Chapters 12&13
- Cormen et al. Introduction to Algorithms.
  - Section 22.1 Representation of graphs
  - Section 22.2 Breadth-First Search
  - Section 24.2 Dijkstra's algorithm
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/Directed/

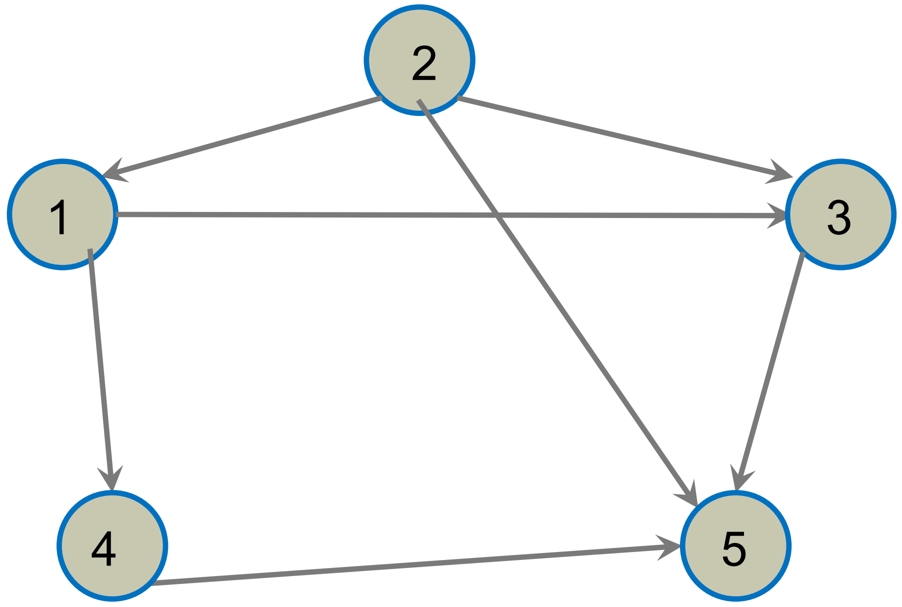
Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

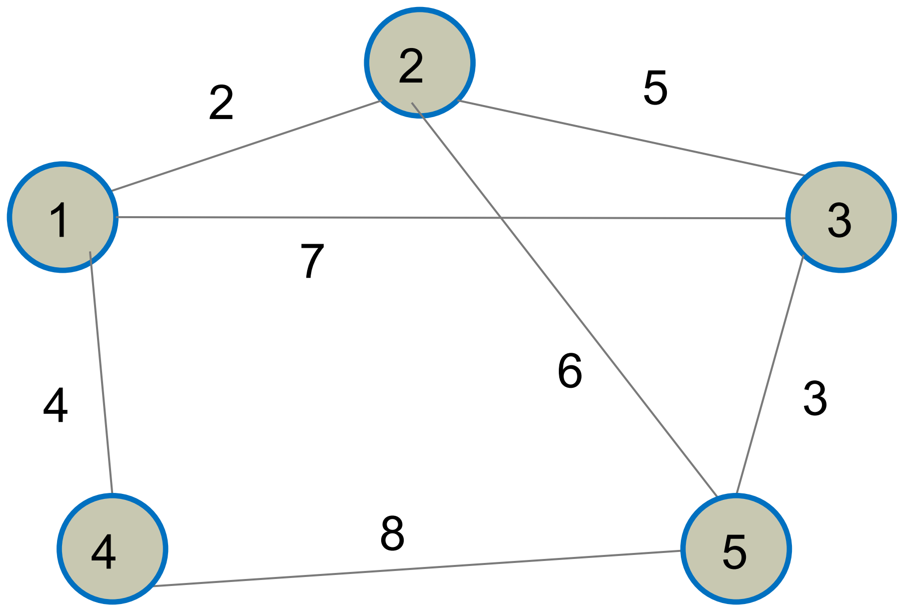
Undirected Graph - Example



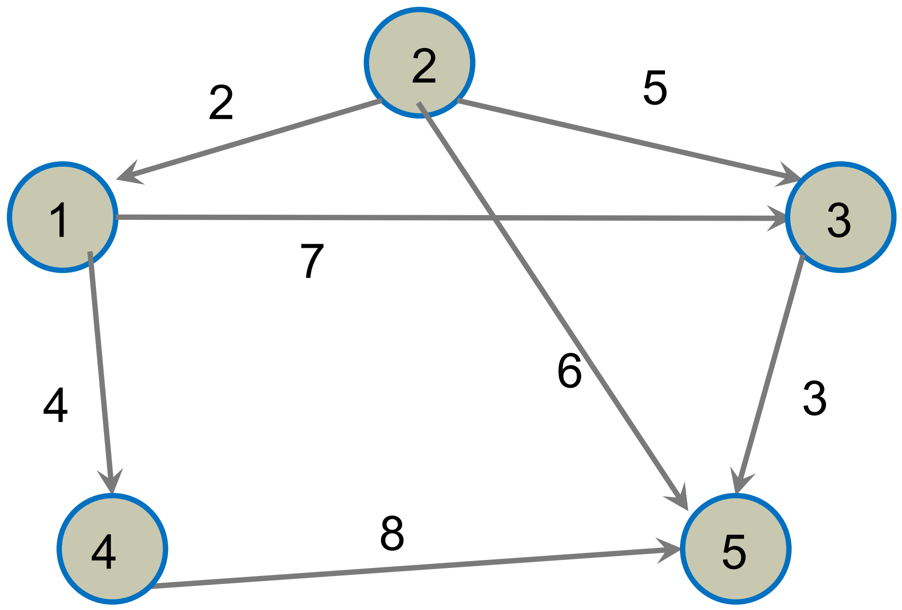
Directed Graph - Example



Undirected Weighted Graph - Example



Directed Weighted Graph - Example



Graphs – 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).
- A weighted graph is represented as G = (V, E, W) where W represents weights for the edges and each edge e is represented as (u, v, w) where w is the weight for the edge (u, v).
- A graph is called a simple graph if it does not have loops AND does not contain multiple edges b/w same pair of vertices.
- In this unit, we focus on simple graphs.

Some Graph Properties

Let G be a graph.

We use V to denote the number of vertices in the graph

We use E to denote the number of edges in the graph

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

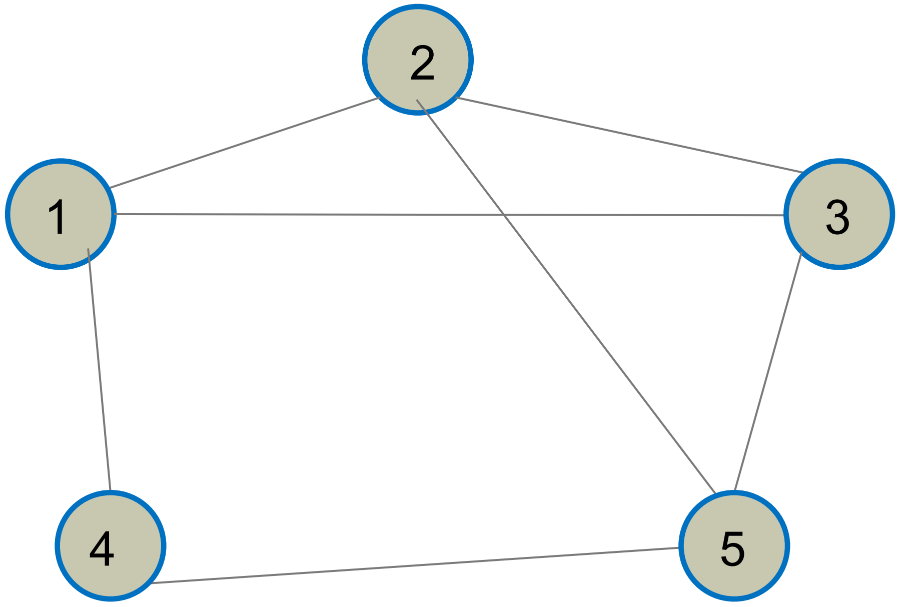
- A graph is called **sparse** if $E \ll V^2$ (<< means significantly smaller than)
- A graph is called **dense** if $E \approx V^2$

Representing Graphs

Adjacency Matrix:

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

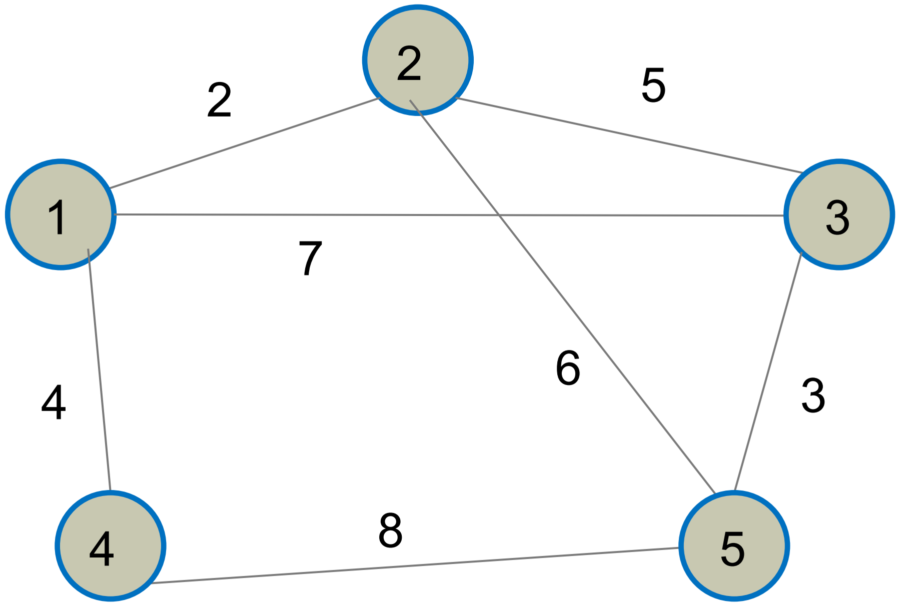


Representing Graphs

Adjacency Matrix:

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	



Representing Graphs

Adjacency Matrix:

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.

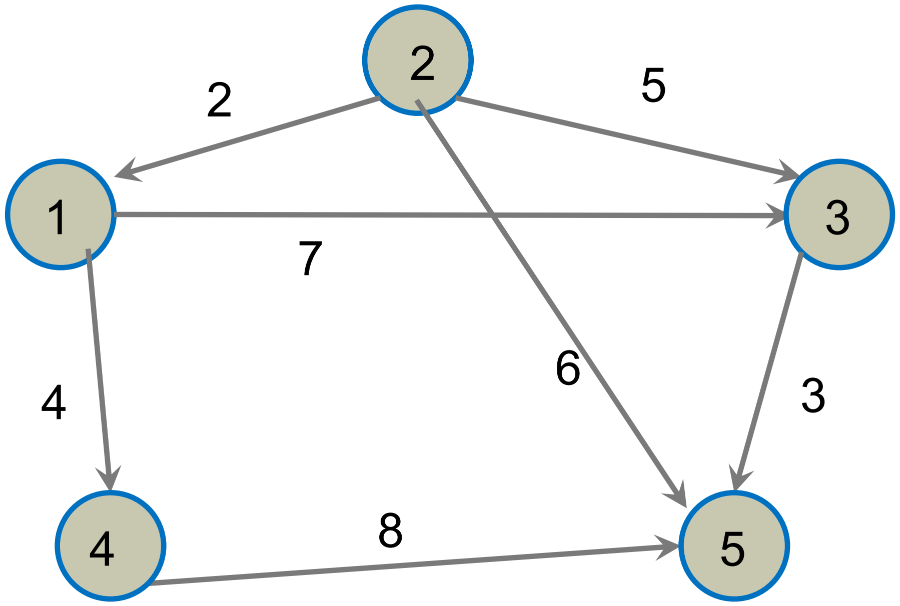
Space Complexity: O(V^2) regardless of the number of edges

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)

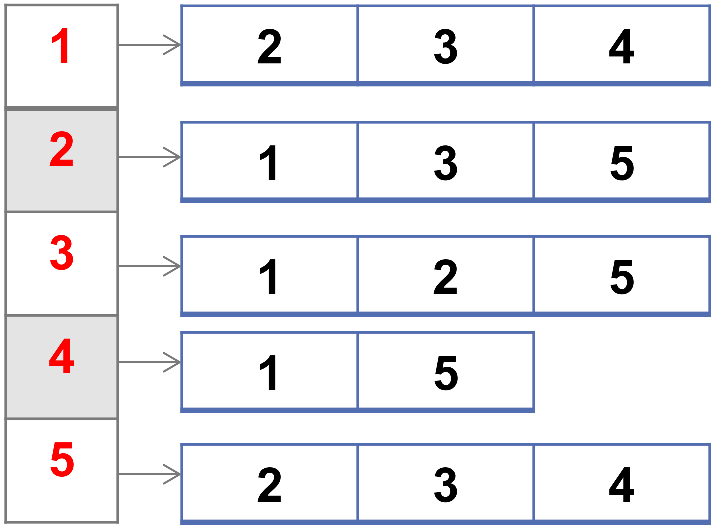
	1	2	3	4	5
1			7	4	
2	2		5		6
3					3
4					8
5					

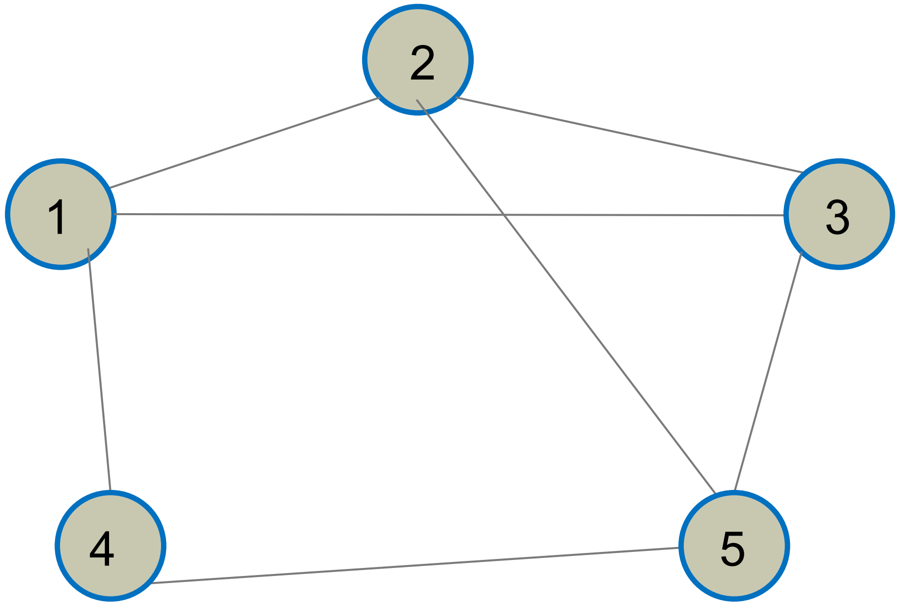


Representing Graphs

Adjacency List:

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



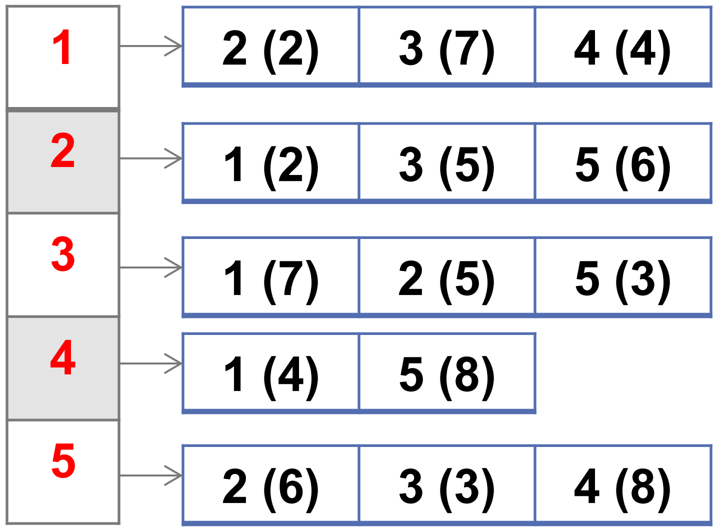


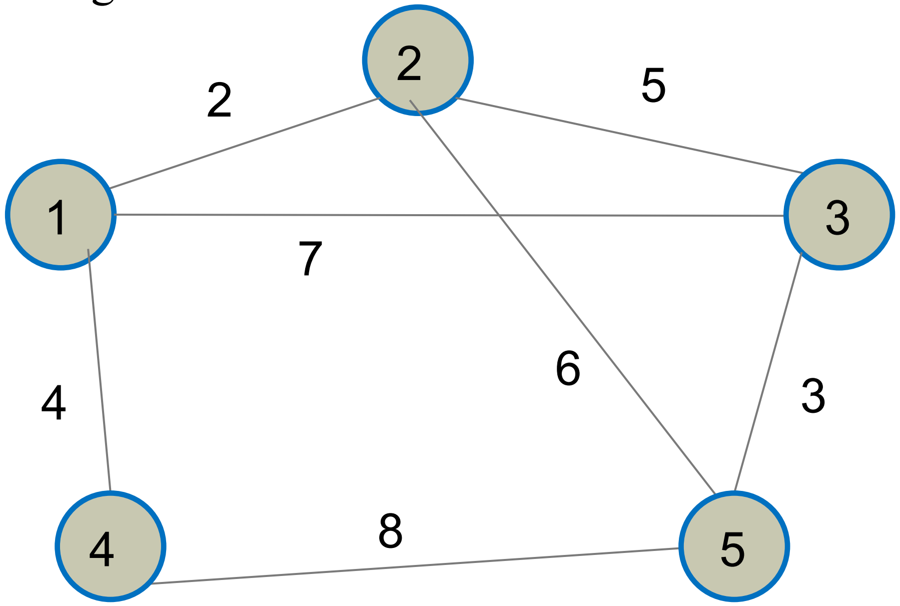
Representing Graphs

Adjacency List:

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.





Representing Graphs

Adjacency List:

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:

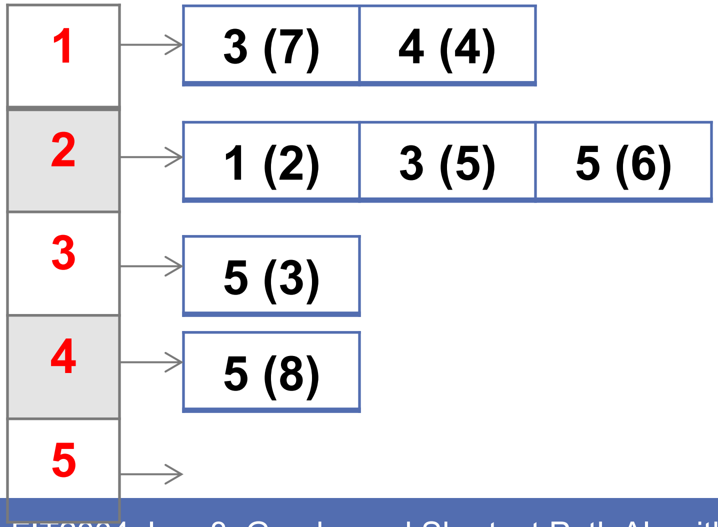
- O(V + E)

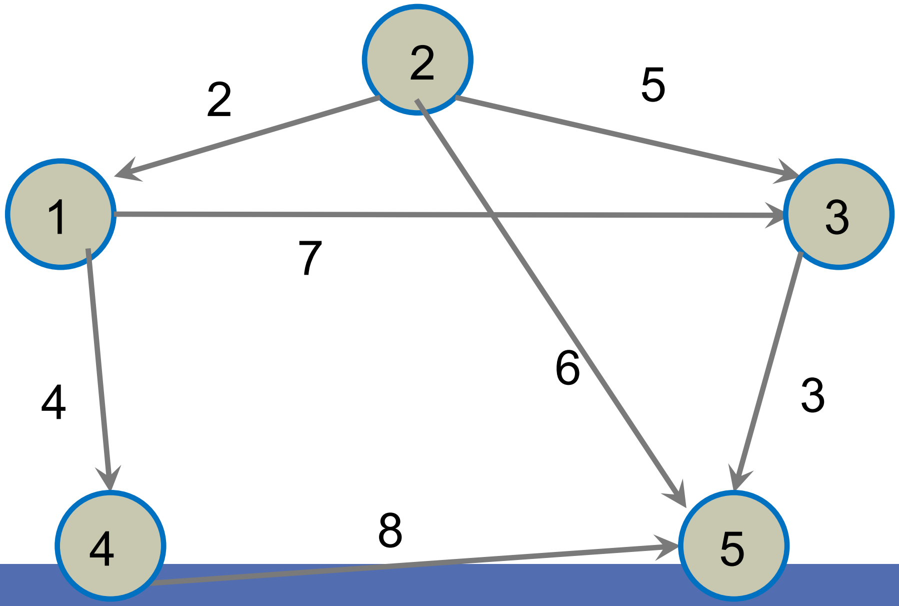
Time complexity of checking if a particular edge exists:

- 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)





Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

Graph Traversal

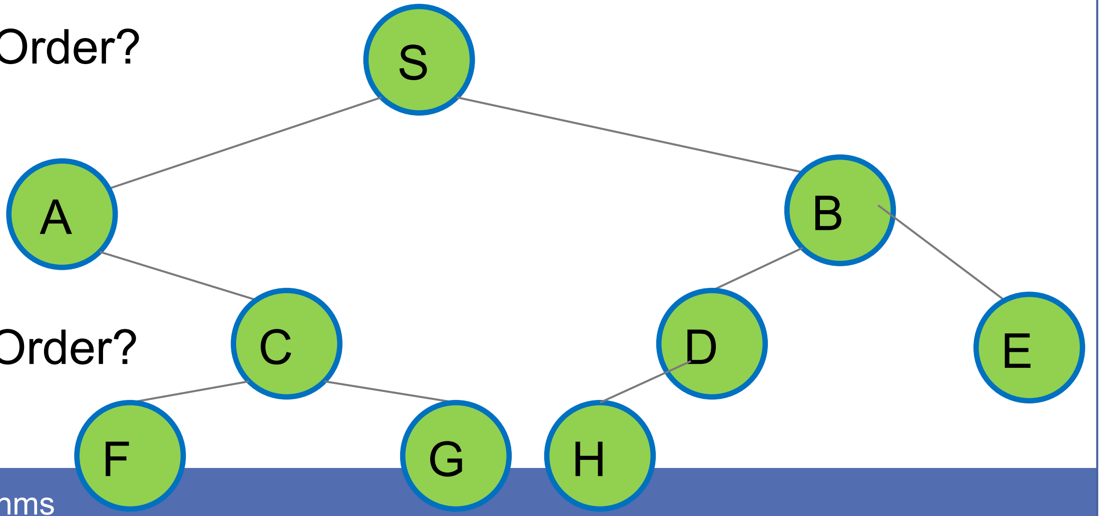
Graph traversal algorithms traverse (visit) the nodes of a graph starting from a source vertex.

We will look into two algorithms:

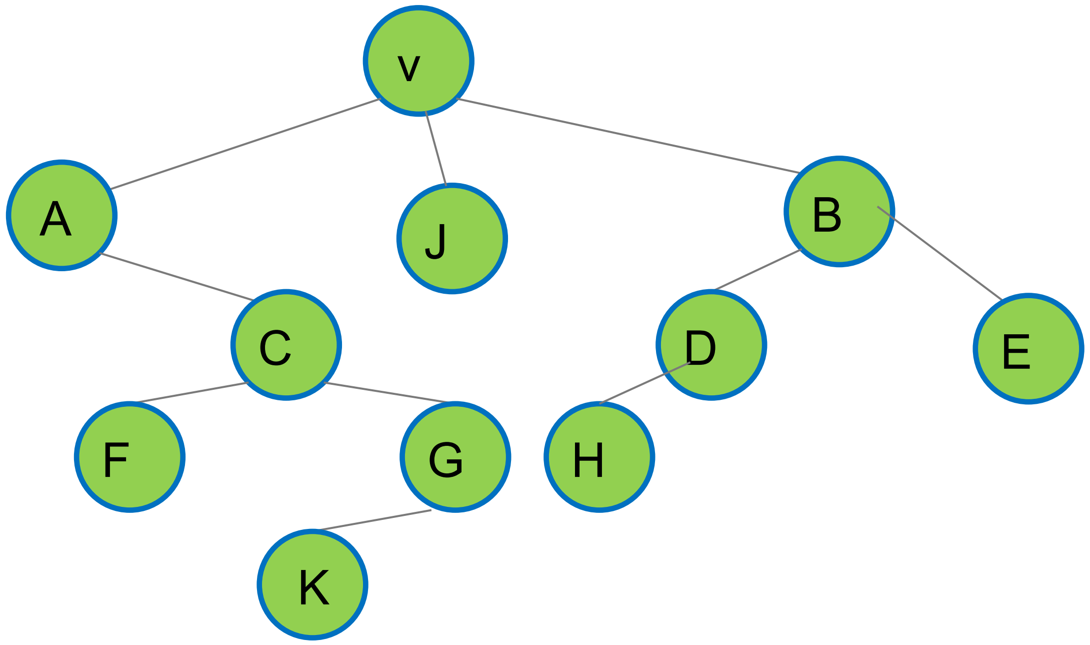
- Breadth First Search (BFS)
  - traverses the graph uniformly from the source vertex
  - i.e., all vertices that are k edges away from the source vertex are visited before all vertices that are k+1 edges away from source
  - In the tree below, one possible BFS order is S, A, B, C, D, E, F, G, H.
- Depth First Search (DFS)
  - traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the tree, one possible DFS order is: S, A, C, F, G, B, E, D, H.

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

Is S, A, C, G, F, B, D, E, H a DFS Order?
No!



Graph Traversal



Outline

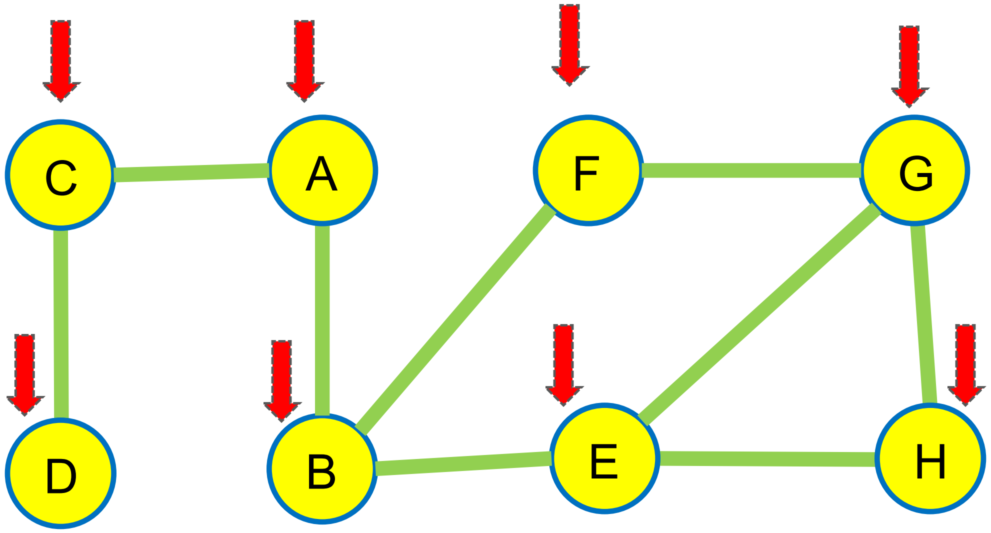
1. Introduction to Graphs
2. Graph Traversal Algorithms
    A. Breadth First Search (BFS)
    B. Depth First Search (DFS)
    C. Applications
3. Shortest Path Problem
    A. Breadth First Search (for unweighted graphs)
    B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

Breadth First Search (BFS)

- Initialize a list called Discovered and insert the source node A in it
- While Discovered is not empty
  - Get the first vertex v from the Discovered List
  - For each adjacent edge (v,u)
    - If u is not discovered or visited
      - Insert u at the end of Discovered list
  - Move v from Discovered to Visited

Discovered: ●

Visited: ●



Discovered:

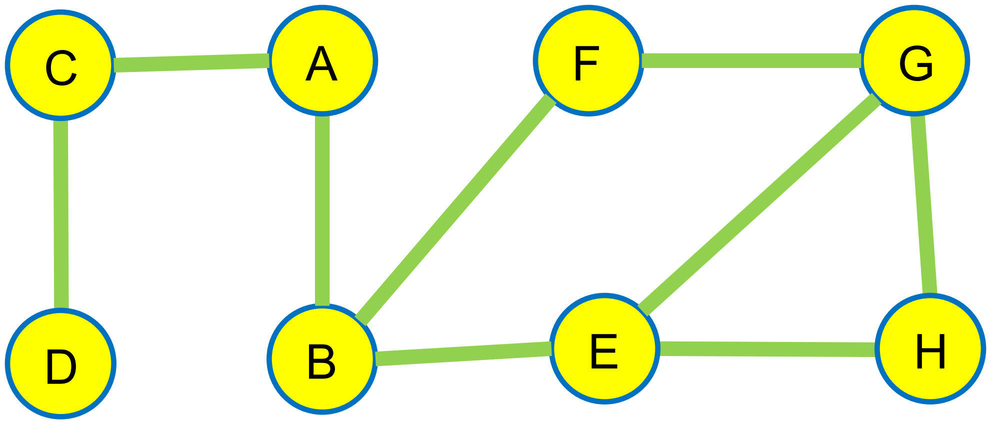
A	B	C	E	F	D	G	H

Visited:

A	B	C	E	F	D	G	H

Breadth First Search (BFS)

- Initialize a list called Discovered and insert the source node A in it
- While Discovered is not empty
  - Get the first vertex v from the Discovered List
  - For each edge (v,u)
    - If u is not discovered or visited
      - Insert u at the end of Discovered list
  - Move v from Discovered to Visited



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)

Outline

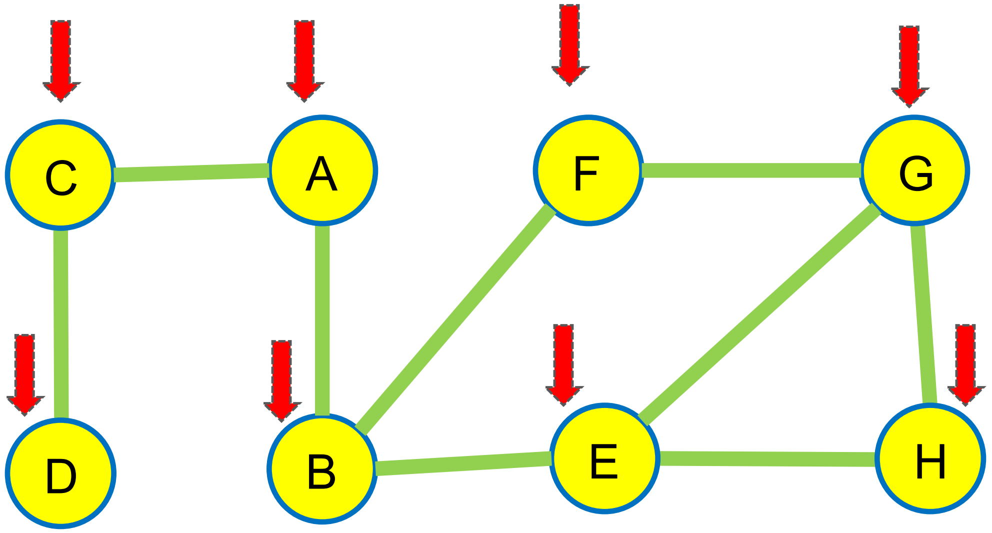
1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

Depth First Search (DFS)

- DFS(A)

- function DFS(v):
  - Mark u as Visited
  - For each adjacent edge (v,u)
    - If u is not visited
      - DFS(u)

Visited: ●

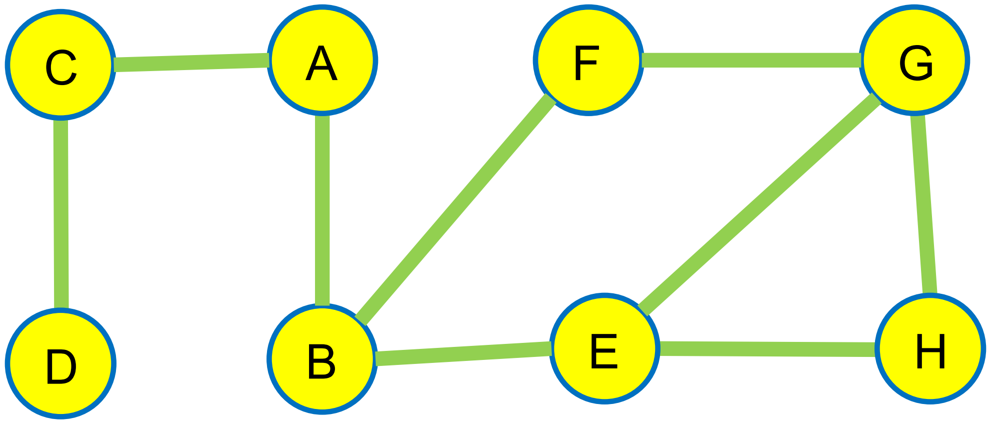


Visited:	A	B	E	G	H	F	C	D

Depth First Search (DFS)

- DFS(A)

- function DFS(v):
  - Mark u as Visited
  - For each edge (v,u)
    - If u is not visited
      - DFS(u)



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)

Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra's algorithm (for weighted graphs with non-negative weights)

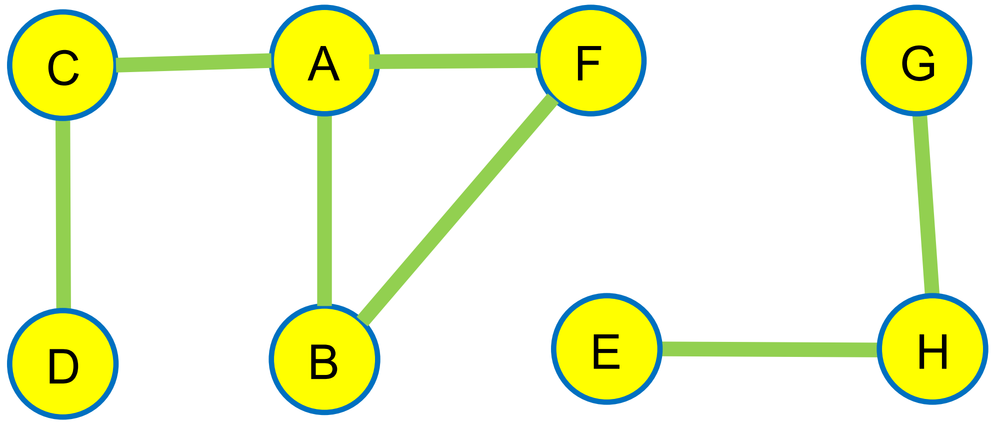
Applications of DFS and BFS

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
- Finding cycles
- Topological sort (week 11)
- Shortest paths on unweighted graphs
- …

More details are given in unit notes and tutorials



Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

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 sources 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.

Shortest Path Algorithms

- Breadth First Search – (Unweighted graphs)
- Dijkstra's Algorithm – (Weighted graphs with only non-negative weights)
- Bellman Ford Algorithm – (Weighted graphs including negative weights)

Outline

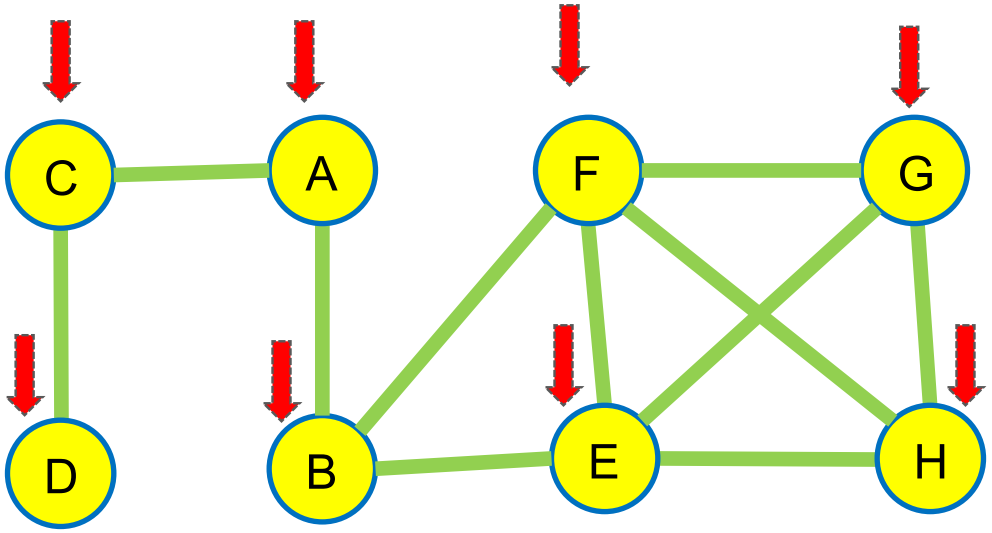
1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. Breadth First Search (BFS)
   B. Depth First Search (DFS)
   C. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

Breadth First Search (BFS)

- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
  - Get the first vertex v from the Discovered List
  - For each edge (v,u)
    - If u is not discovered or finalized
      - u.distance = v.distance + 1
      - Insert u at the end of Discovered list
  - Move v from Discovered to Finalized

Discovered: ●

Finalized: ●



Discovered:	A,0	B,1	C,1	E,2	F,2	D,2	G,3	H,3
Finalized:	A,0	B,1	C,1	E,2	F,2	D,2	G,3	H,3

Breadth First Search (BFS)

- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
  - Get the first vertex v from the Discovered List
  - For each adjacent vertex u of v
    - If u is not discovered or finalized
      - u.distance = v.distance + 1
      - Insert u at the end of Discovered list
  - Move v from Discovered to Finalized

Can standard DFS be used to compute shortest distances?

No! Try it yourself!

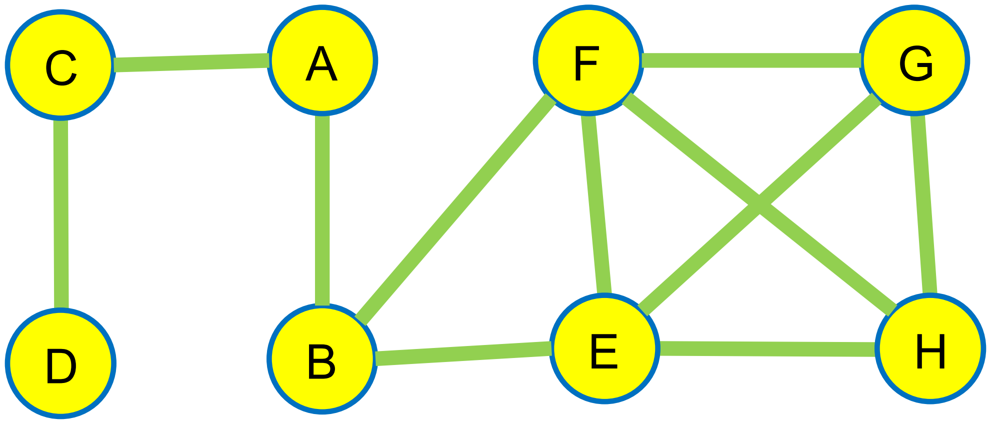
Assuming adjacency list representation.

Time Complexity:

O(V + E)

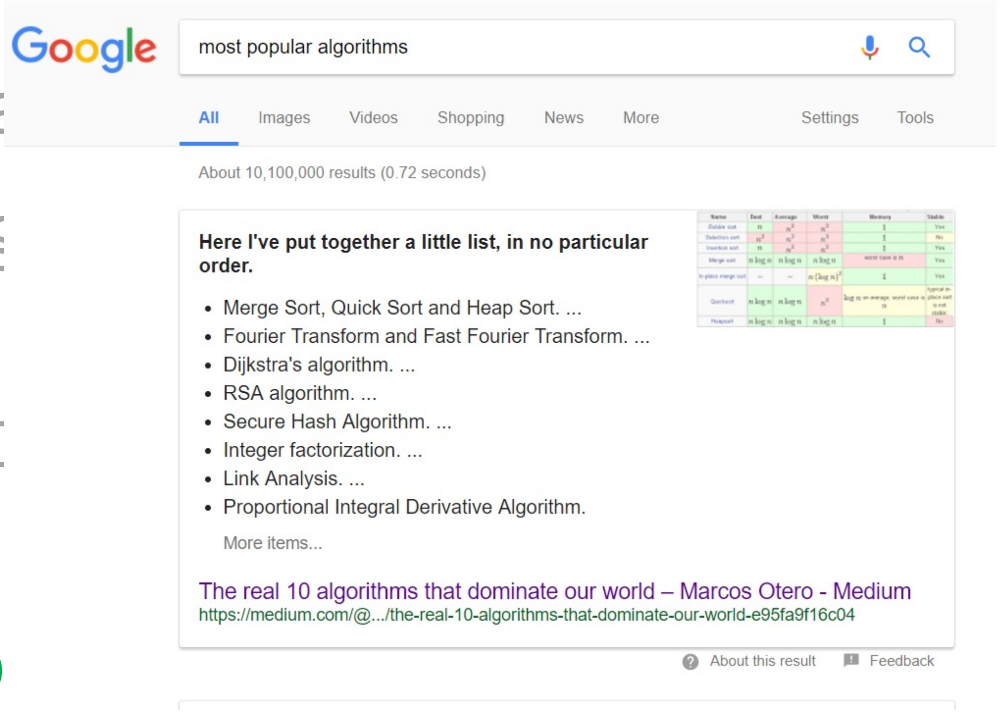
Space Complexity:

O(V + E)



Outline

1. Introduction to Gra
2. Graph Traversal Alg
   - A. Breadth First Search
   - B. Depth First Search (I
   - C. Applications
3. Shortest Path Prob
   - A. Breadth First Search (for unweighted graphs)
   - B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)



Google

most popular algorithms

About 10,100,000 results (0.72 seconds)

Here I've put together a little list, in no particular order.

- Merge Sort, Quick Sort and Heap Sort. ...
- Fourier Transform and Fast Fourier Transform. ...
- Dijkstra's algorithm. ...
- RSA algorithm. ...
- Secure Hash Algorithm. ...
- Integer factorization. ...
- Link Analysis. ...
- Proportional Integral Derivative Algorithm.

More items...

Name	Best	Average	Worst	Memory	Stable
Bubble sort	n	n^2	n^2	1	Yes
Selection sort	n^2	n^2	n^2	1	No
Insertion sort	n	n^2	n^2	1	Yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	worst case is n	Yes
In-place merge sort	—	—	$n (\log n)^2$	1	Yes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ on average, worst case is n	typical in-place sort is not stable
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No

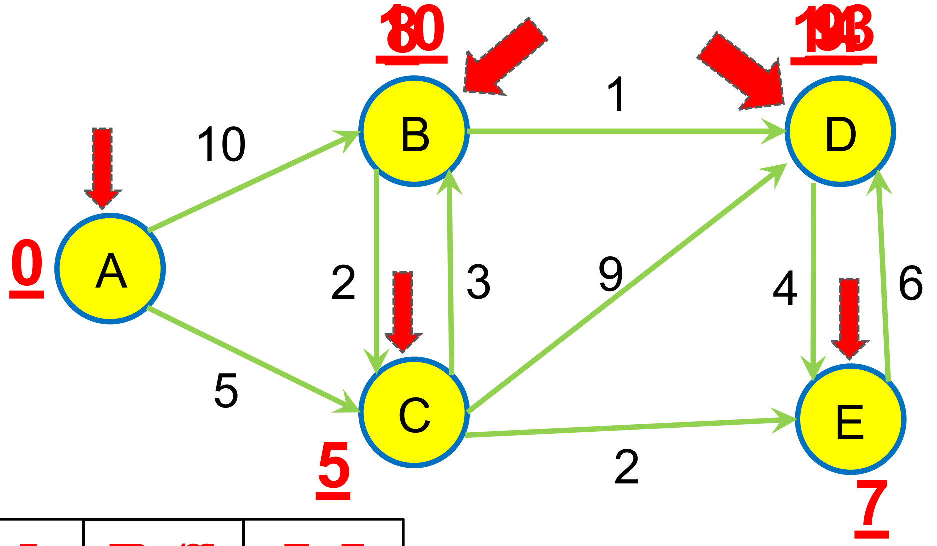
The real 10 algorithms that dominate our world – Marcos Otero - Medium
https://medium.com/@.../the-real-10-algorithms-that-dominate-our-world-e95fa9f16c04

Dijkstra's Algorithm

- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
  - Get the vertex v from the Discovered List **with smallest distance**
  - For each outgoing edge (v, u, w) of v
    - If u is not in Discovered/Finalized
      - Insert u in Discovered with distance **v.distance + w**
    - Else If u.distance > v.distance + w
      - update the distance of u in Discovered to v.distance + w
  - Move v from Discovered to Finalized

Discovered: ●

Finalized: ●



Discovered:

A, 0	B,10 B,8	C,5	D, 14 D, 13 D, 9	E, 7

Finalized:

A,0	C,5	E, 7	B,8	D, 9

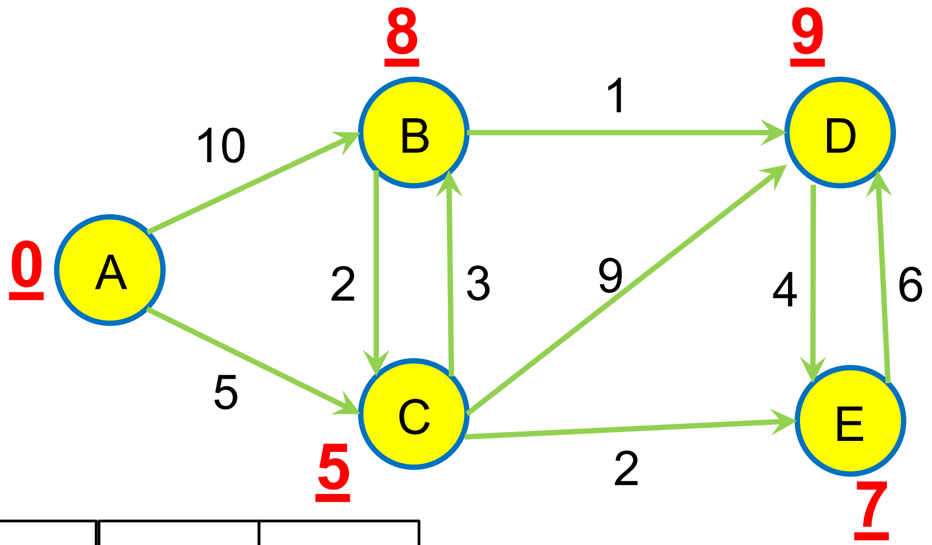
Dijkstra's Algorithm

- Initialize a list called Discovered and insert the source vertex A in it with distance 0
- While Discovered is not empty
  - Get the vertex v from the Discovered List with smallest distance
  - For each outgoing edge (v, u, w) of v
    - If u is not in Discovered or Finalized
      - Insert u in Discovered with distance v.distance + w
    - Else If u.distance > v.distance + w
      - If u is not finalized, update the distance of u in Discovered to v.distance + w
  - Move v from Discovered to Finalized

Assume Discovered is implemented as an array (direct-addressing) where i-th vertex is at index i.

Time Complexity:

- Each edge visited once → O(E)
- While loop executes O(V) times
  - Find the vertex with smallest distance: O(V)
- Total cost: $O(E + V^2) = O(V^2)$



Finalized:	A,0	C,5	E, 7	B,8	D, 9

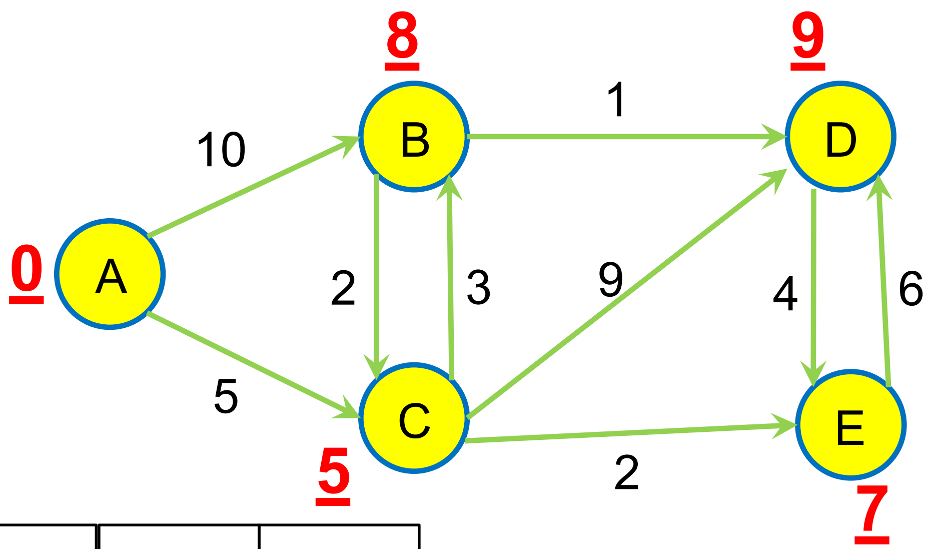
Dijkstra's Algorithm

- Initialize a list called Discovered and insert the source vertex A in it with distance 0
- While Discovered is not empty
    - Get the vertex v from the Discovered List with smallest distance
    - For each outgoing edge (v, u, w) of v
        - If u is not in Discovered or Finalized
            - Insert u in Discovered with distance v.distance + w
        - Else If u.distance > v.distance + w
            - If u is not finalized, update the distance of u in Discovered to v.distance + w
    - Move v from Discovered to Finalized

Using a min-heap to implement Discovered.

Time Complexity:

- While loop executed O(V) times
    - Get the vertex with smallest distance: O(1)
    - Removing vertex with smallest distance: O(log V)
- Each edge is visited once: O(E)
    - Updating the distance of a vertex: ?
    - Checking if u is finalized/discovered : ?



Finalized:	A,0	C,5	E, 7	B,8	D, 9

Dijkstra's Algorithm using min-heap

Required additional structure:

- Create an array called Vertices.
- Vertices[i] will record the location of i-th vertex in the min-heap
  - -1 if the vertex is finalized
  - -2 if the vertex is not discovered yet

Checking if a vertex v is discovered or finalized in O(1)

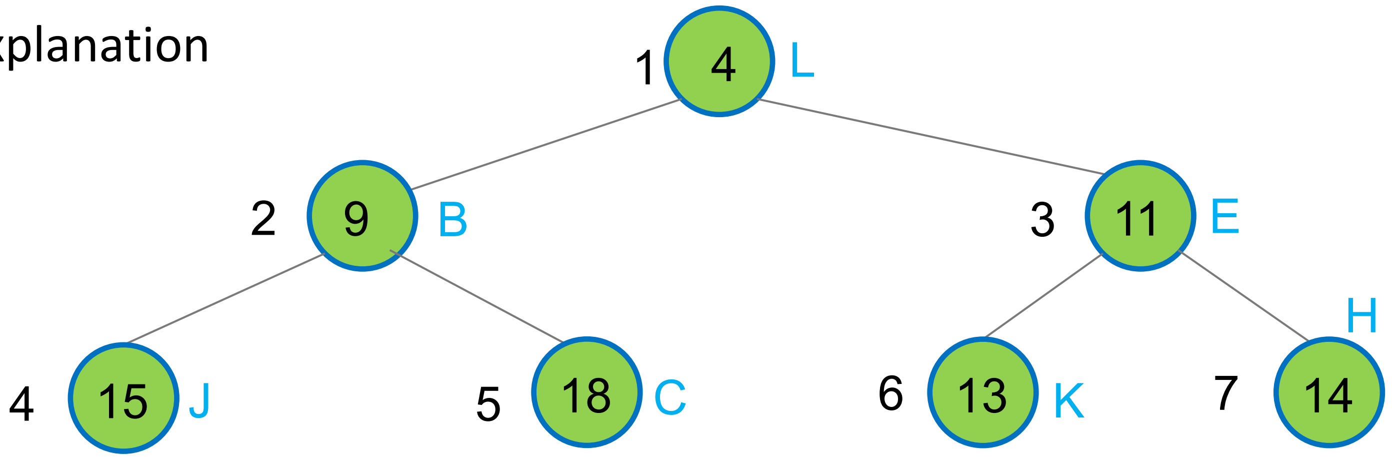
- v is finalized if Vertices[v] == -1
- v is in discovered if Vertices[v] >0

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

- Let j = Vertices[v], i.e., j is the location of v in min-heap
- Update (i.e., decrease) the key of element at min-heap[j]
- Now upHeap this element (by recursively swapping with parent)
  - 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

Dijkstra's Algorithm using min-heap

Watch MULO for explanation



Min-heap

L	B	E	J	C	K	H
4	9	11	15	18	13	14
1	2	3	4	5	6	7

Suppose K's distance is to be updated to 7.

Vertices

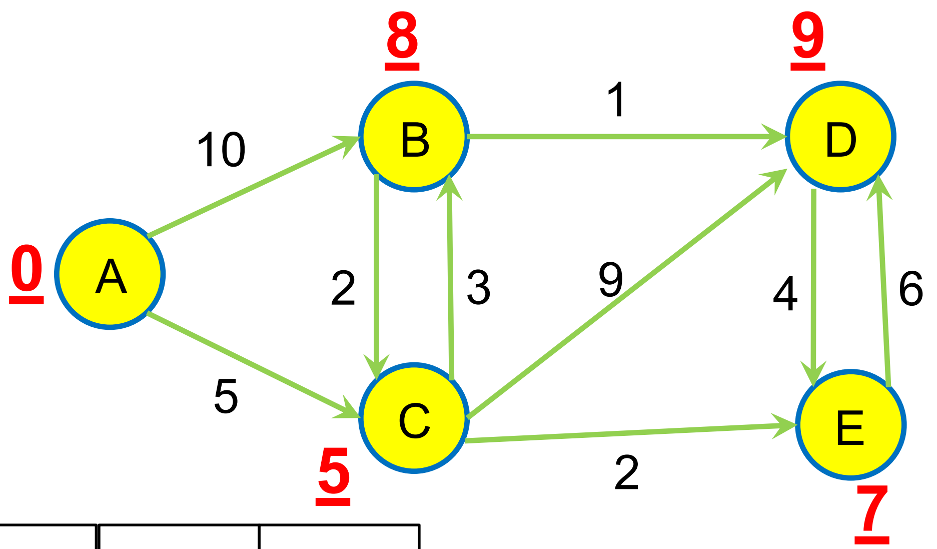
-1	2	5	-2	3	-1	-2	7	-1	4	6	1
A	B	C	D	E	F	G	H	I	J	K	L
1	2	3	4	5	6	7	8	9	10	11	12

Dijkstra's Algorithm

- Initialize a list called Discovered and insert the source vertex A in it with distance 0
- While Discovered is not empty
    - Get the vertex v from the Discovered List with smallest distance
    - For each outgoing edge (v, u, w) of v
        - If u is not in Discovered or Finalized
            - Insert u in Discovered with distance v.distance + w
        - Else If u.distance > v.distance + w
            - If u is not finalized, update the distance of u in Discovered to v.distance + w
    - Move v from Discovered to Finalized

Time Complexity:

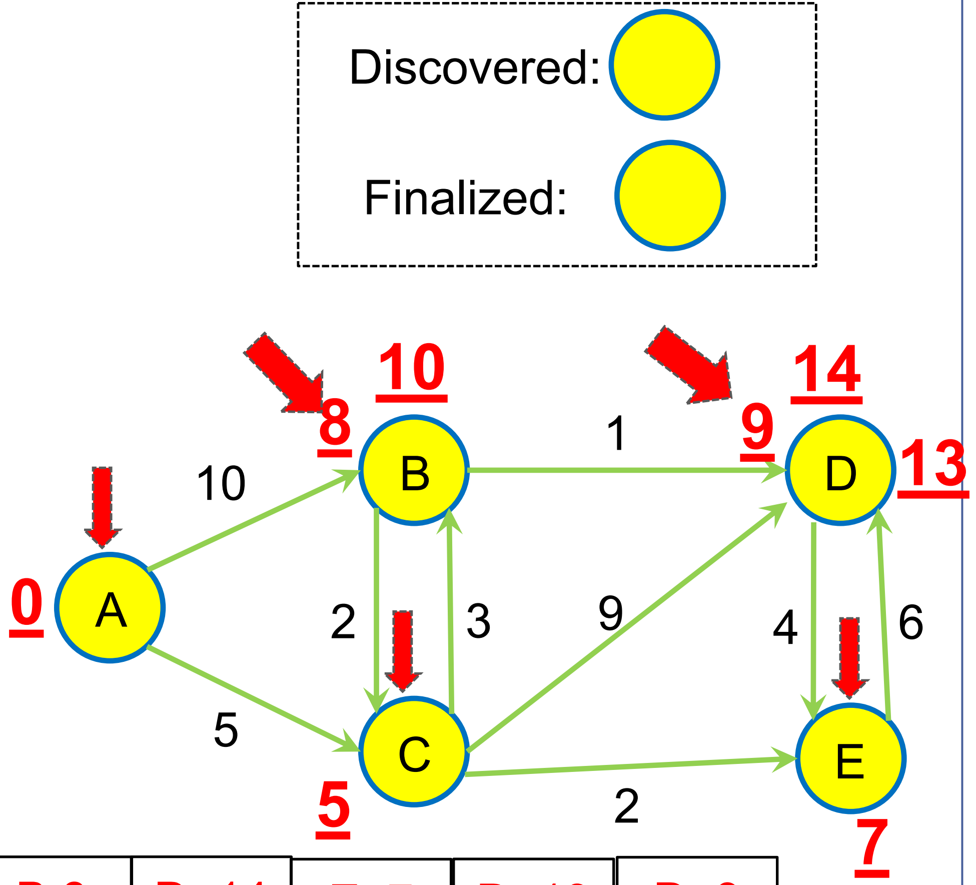
- While loop executed O(V) times
    - Get the vertex with smallest distance: O(1)
    - Removing vertex with smallest distance: O(log V)
- Each edge is visited once: O(E)
    - Updating the distance of a vertex: O(log V)
    - Checking if u is finalized/discovered: O(1)
- Total cost: O(E log V + V log V) → O(E log V) because O(E) >= O(V) for connected graphs.



Finalized:	A,0	C,5	E, 7	B,8	D, 9

Dijkstra’s Algorithm: Alternative implementation

- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
  - Get the vertex v from the Discovered List with smallest distance
  - **If v is not Finalized**
    - For each outgoing edge (v, u, w) of v
      - If u is not Discovered/Finalized
        - Insert u in Discovered with distance **v.distance** + **w**
      - Else If u.distance > v.distance + w
        - ~~update the distance of~~ **insert u in** Discovered to v.distance + w
    - Mark v as Finalized



Discovered:								
	A, 0	B,10	C,5	B,8	D, 14	E, 7	D, 13	D, 9

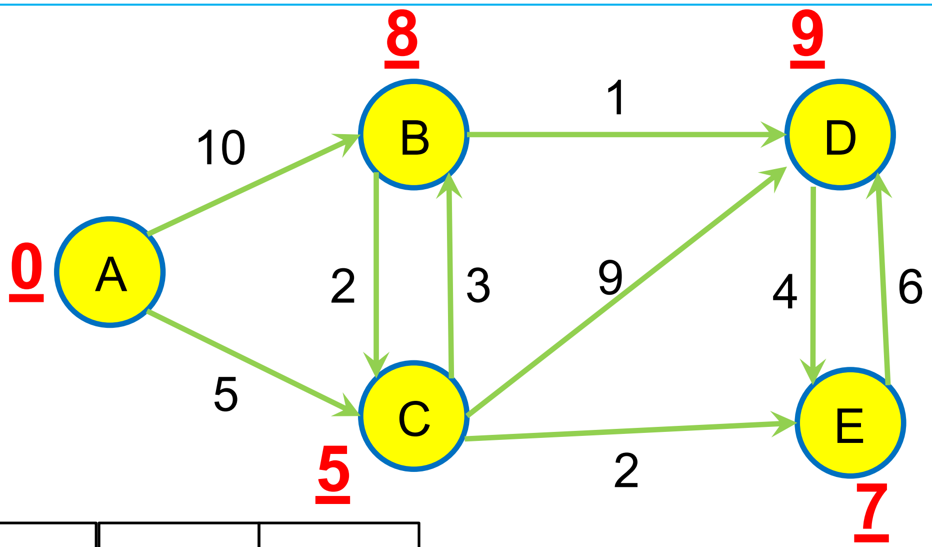
Finalized:					
	A,0	C,5	E, 7	B,8	D, 9

Dijkstra's Algorithm: Alternative implementation

- Initialize a list called Discovered and insert the source vertex A in it with distance 0
- While Discovered is not empty
  - Pop the vertex v from the Discovered List with smallest distance
  - If v is not Finalized
    - For each outgoing edge (v, u, w) of v
      - If u is not in Discovered or Finalized
        - Insert u in Discovered with distance v.distance + w
      - Else If u.distance > v.distance + w
        - ~~update the distance of u~~ insert u in Discovered with v.distance + w
    - Move v from Discovered to Finalized

Time Complexity:

- Total # of entries inserted in Discovered ?
  - O(E)
- While loop executes O(E) times
  - Pop vertex with smallest distance: O(log E)
- Each edge is visited once: O(E)
  - inserting the distance of a vertex: O(log E)
- Total cost: O(E log E)
- Since E <= V^2, O(E log E) → O(E log V^2) → O(E log V)



Finalized:	A,0	C,5	E, 7	B,8	D, 9

Time Complexity of Dijkstra's Algorithm

Dijkstra's using an array for Discovered

- $O(V^2)$

Dijkstra's using a min-heap

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

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

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

Proof of Correctness

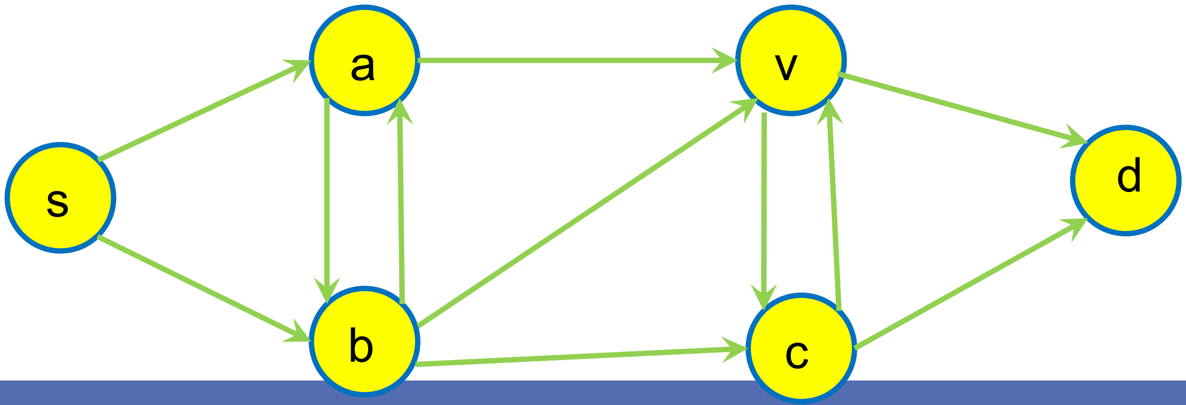
Claim: For every vertex v in Finalized, v.distance is the shortest distance from s to v

Base Case (Finalized has only the source vertex s):

- s.distance = 0 which is the shortest distance from s to s

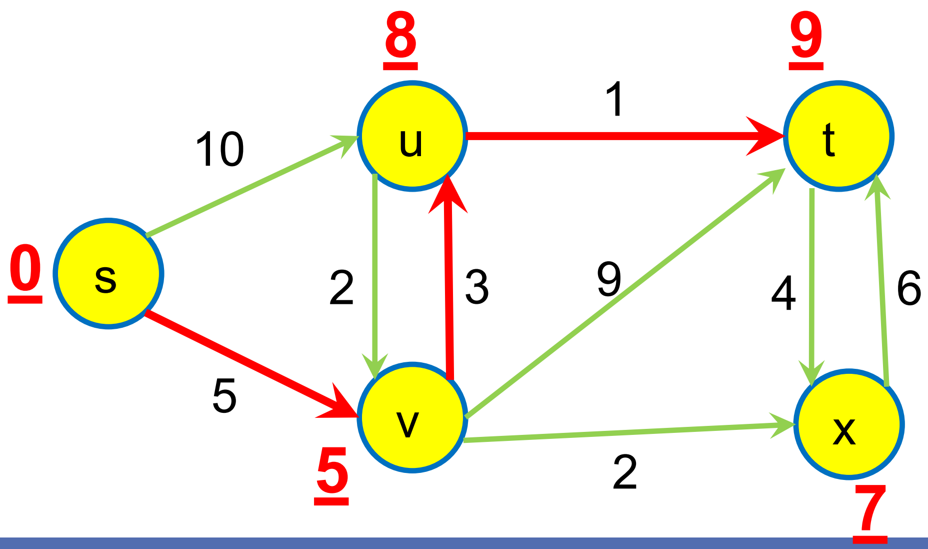
Inductive Step: Assume that the claim holds for all vertices in Finalized. Let v be the vertex with minimum distance in heap (to be finalized in this iteration). We show that v.distance is the shortest distance from s to v

- All “non-finalized” vertices adjacent to any Finalized vertex are in the heap and v has the smallest distance of these.
- Assume that v.distance is NOT the shortest distance from s to v.
  - This implies that there is a path P from s to v that is shorter than v.distance (e.g., P is s → b → c → v)
  - Such a path P must contain at least one vertex that is not Finalized
    - Otherwise v.distance must have been updated to be the length of P (e.g., P cannot be s→a→v)
  - Let c be the first vertex on this path P that is not Finalized
  - v.distance ≤ c.distance because both v and c are present in the heap and v is the root of the min-heap
  - The above implies that v.distance ≤ length of s → b → c ≤ length of P which contradicts the assumption



Single Source Single Target

- Single source single target problem can be solved using the same algorithm except that the algorithm stops as soon as the target vertex t is finalized.
- The algorithms we saw earlier return only the shortest distances
- To get the shortest path
  - When a vertex u is finalized, we also store the previous vertex v that leads to this shortest distance
  - Shortest path then can be recovered easily using this information

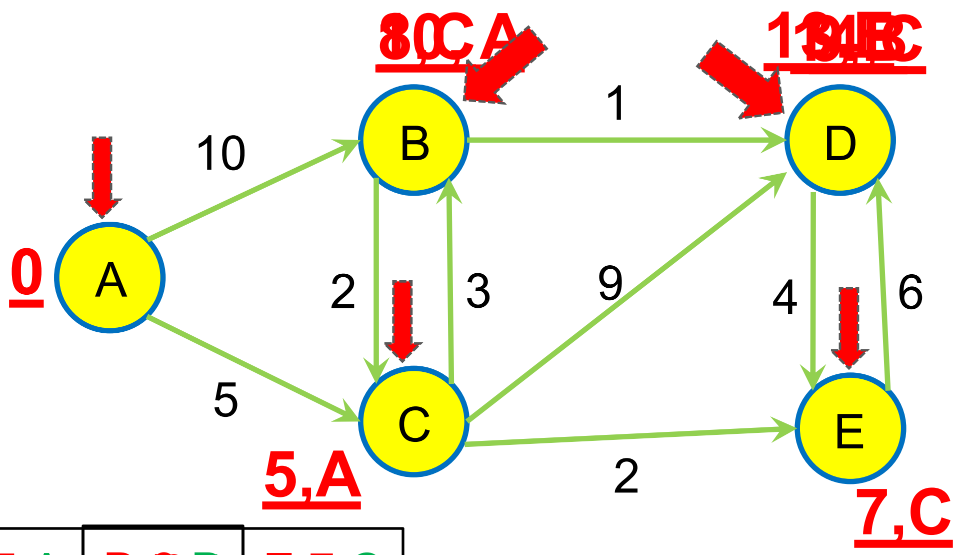


Dijkstra's Algorithm: Recovering Path

- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
  - Get the vertex v from the Discovered List with smallest distance
  - For each outgoing edge (v, u, w) of v
    - If u is not in Discovered or Finalized
      - Insert u in Discovered with distance v.distance + w and prev set as v
    - Else If u.distance > v.distance + w
      - If u is not finalized, update the distance of u in Discovered to v.distance + w and prev set as v
  - Move v from Discovered to Finalized along with its prev

Discovered: ●

Finalized: ●



Discovered:

A, 0	B,8,C	C,5,A	D,9,B	E,7,C

Finalized:

A,0	C,5,A	E, 7,C	B,8,C	D, 9,B

Dijkstra’s Algorithm: Recovering Path

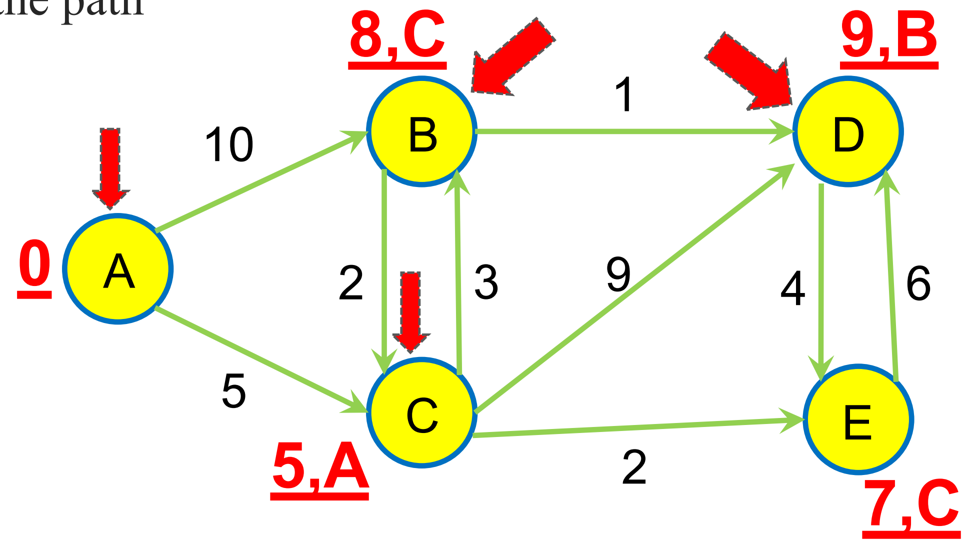
Now, the shortest path from **source** to any **target** vertex.

Example: Path from A to D

- current ← target
- While current != source:
    - Append current.prev before current in the path
    - current = current.prev

Discovered: ●

Finalized: ●



Note: Same ideas can be applied for the alternative implementation to recover path

Shortest Path: A → C → B → D

Summary

Take home message

- Dijkstra's algorithm can be improved significantly using a heap

Things to do (this list is not exhaustive)

- Read more about DFS, BFS and Dijkstra's algorithm and implement these
- Read unit notes

Coming Up Next

- Bellman-Ford, Floyd-Warshall Algorithms and Transitive Closures