

Faculty of Information Technology, Monash University

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

FIT2004, S2/2016

Week 8: Introduction to Graphs and Shortest Path Problems

Lecturer: Muhammad Aamir Cheema

ACKNOWLEDGMENTS

The slides are based on the material developed by [Arun Konagurthu](#) and [Lloyd Allison](#).

Announcements

- Assessment week 10 to be released by this weekend
 - Due: 03-Oct-2016 10:00:00 AM
 - The submissions will be passed through MOSS for plagiarism detection
 - Most probably a variant of shortest path problem and Burrows-Wheeler Transform. Therefore, attempt the lab questions for week 08 after/before you are interviewed.
- Round 1: top-3 contestants
 - Alexxaurus: Alex Phu Hung Ong (1st position)
 - Patra3: Khoa Phan Anh Tran (1st position)
 - Wzha246: Wenyu Zhao (1st position)

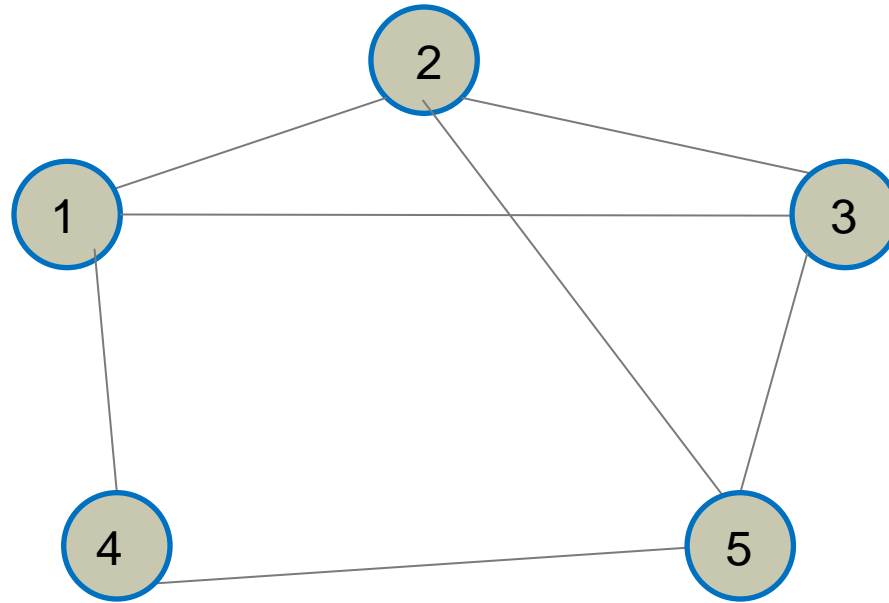
Overview

- Introduction to Graphs
- Shortest Path Problem
 - What is the shortest path between different vertices of a graph?
 - ✦ Breadth-First Search (BFS) on unweighted graphs
 - ✦ Dijkstra's Algorithm on (non-negative) weighted graphs

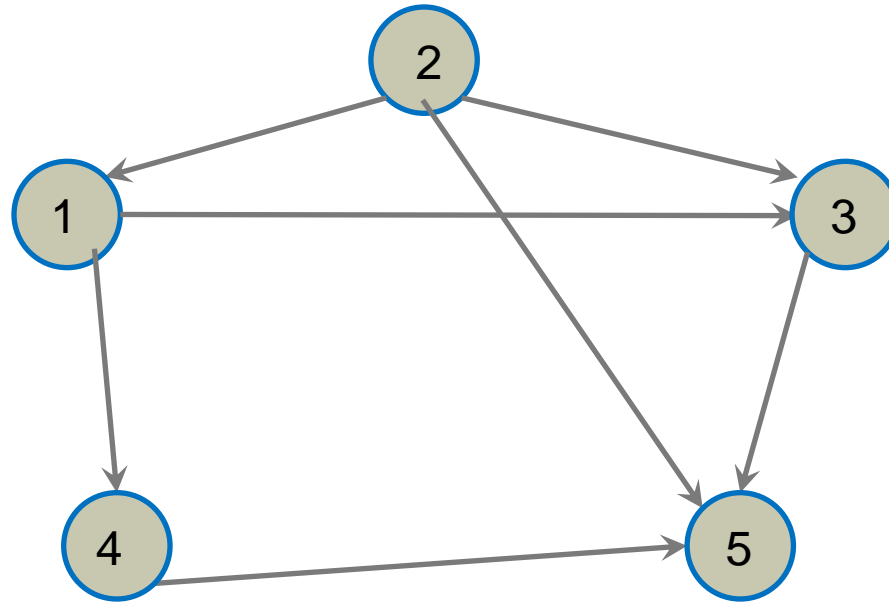
Recommended reading

- 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/>

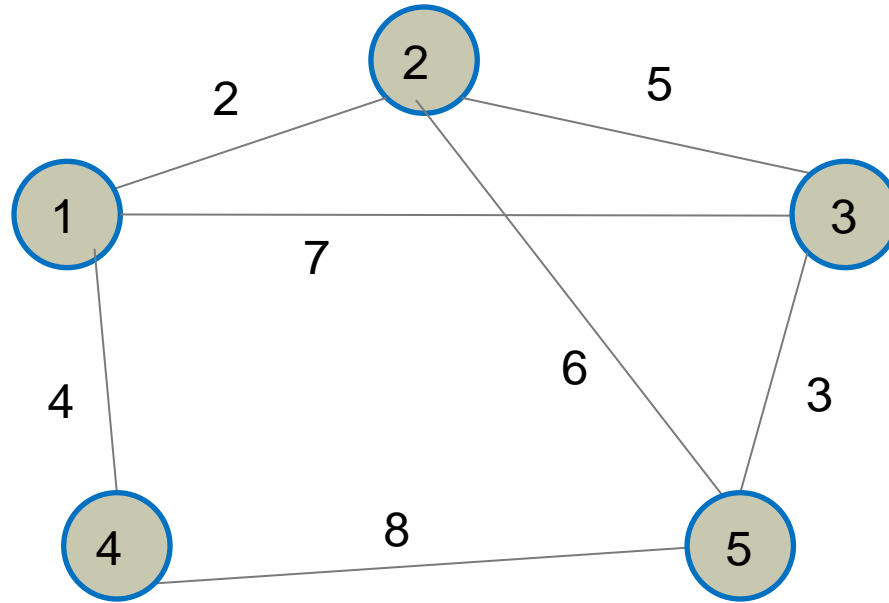
Undirected Graph - Example



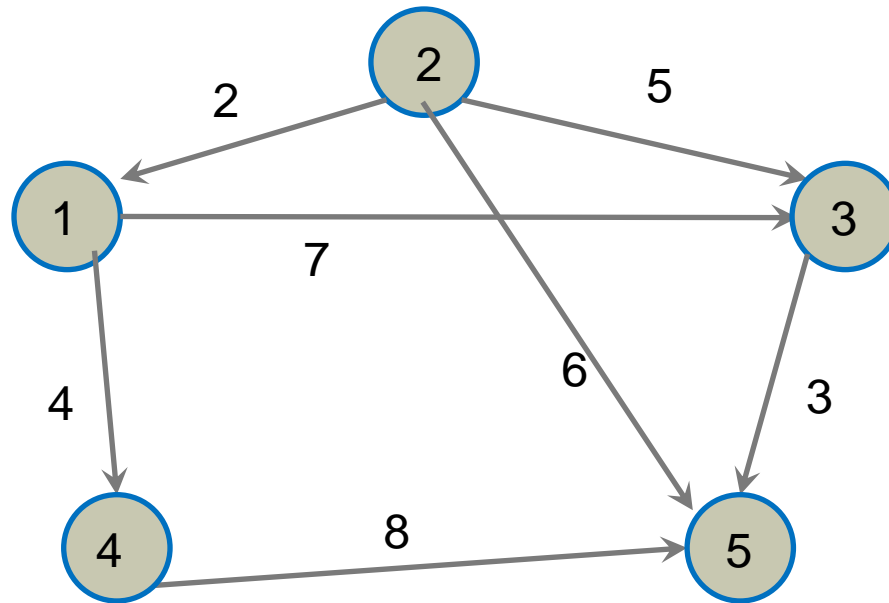
Directed Graph - Example



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 $(u, v) \neq (v, u)$.
- 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) .

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

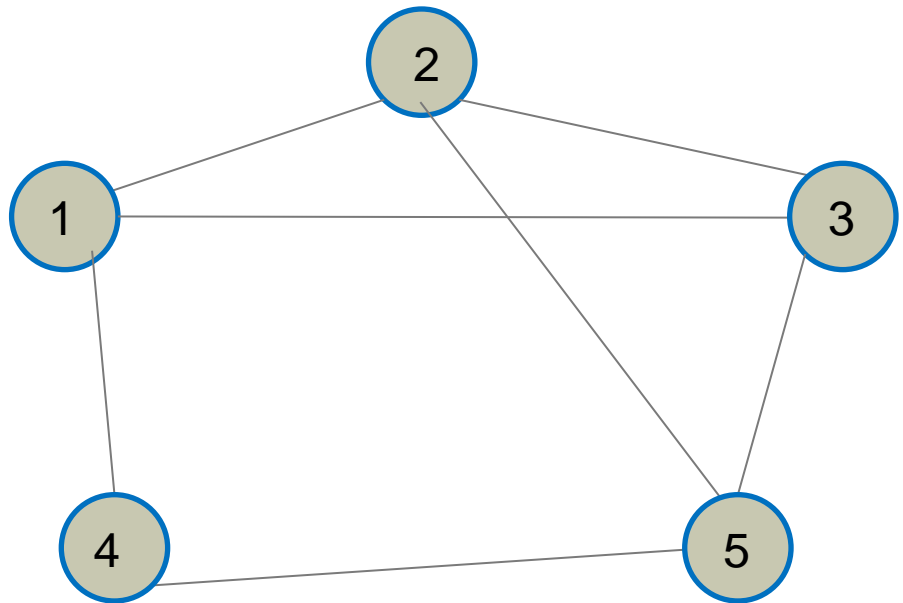
- The maximum number of edges in a directed graph (excluding self edges)
 - $V(V - 1) = O(V^2)$
- The maximum number edges in an undirected graph (excluding self edges)
 - $V(V - 1)/2 = O(V^2)$
- A graph is called **sparse** if $E \ll V^2$
- A graph is called **dense** if $E \approx V^2$

Representing Graphs

Adjacency Matrix:

Create a $V \times V$ 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

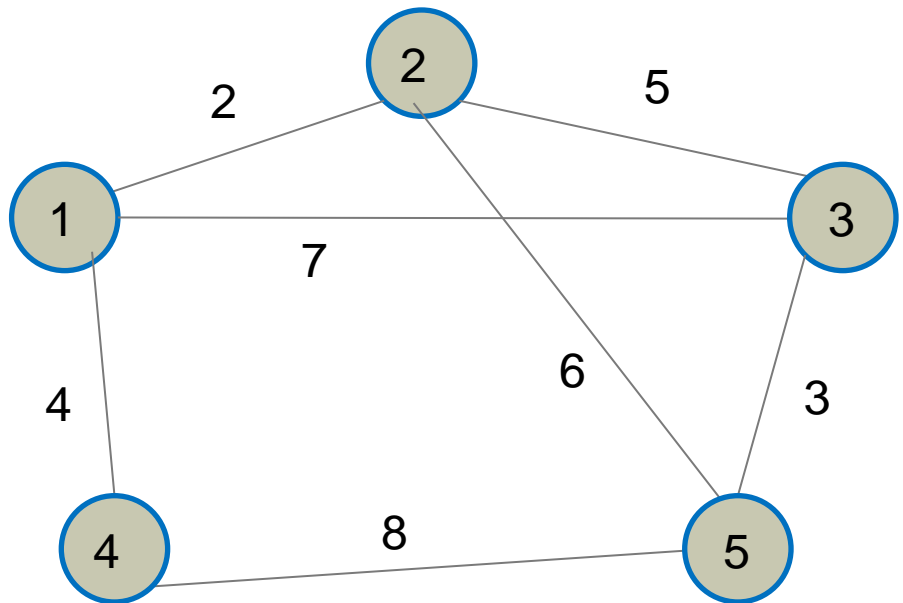


Representing Graphs

Adjacency Matrix:

Create a $V \times V$ 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:

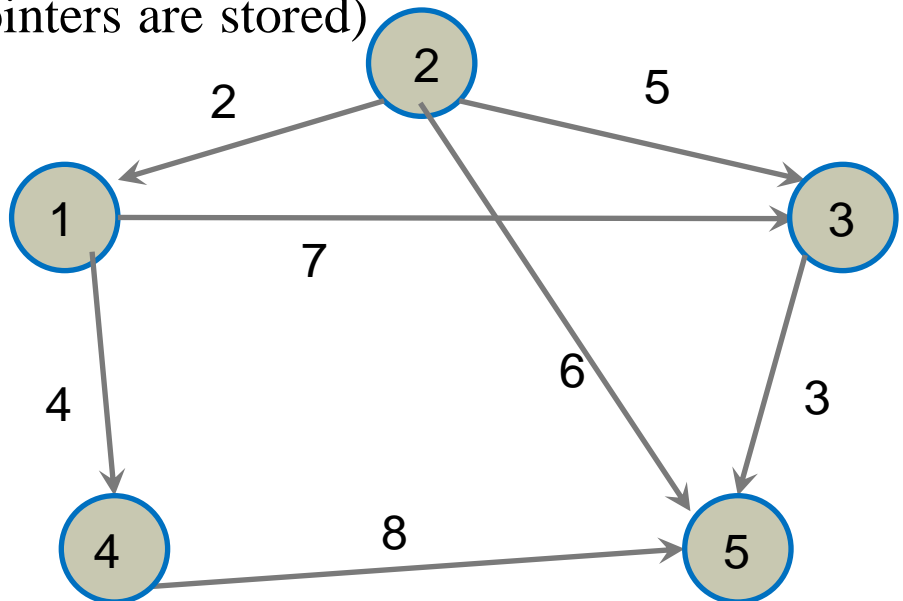
Create a $V \times V$ matrix M and store weight at $M[i][j]$ only if there exists an edge **from** i -th **to** j -th vertex.

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

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

Time Complexity of retrieving all adjacent vertices: $O(V)$ regardless of the number of neighbors (unless additional pointers are stored)

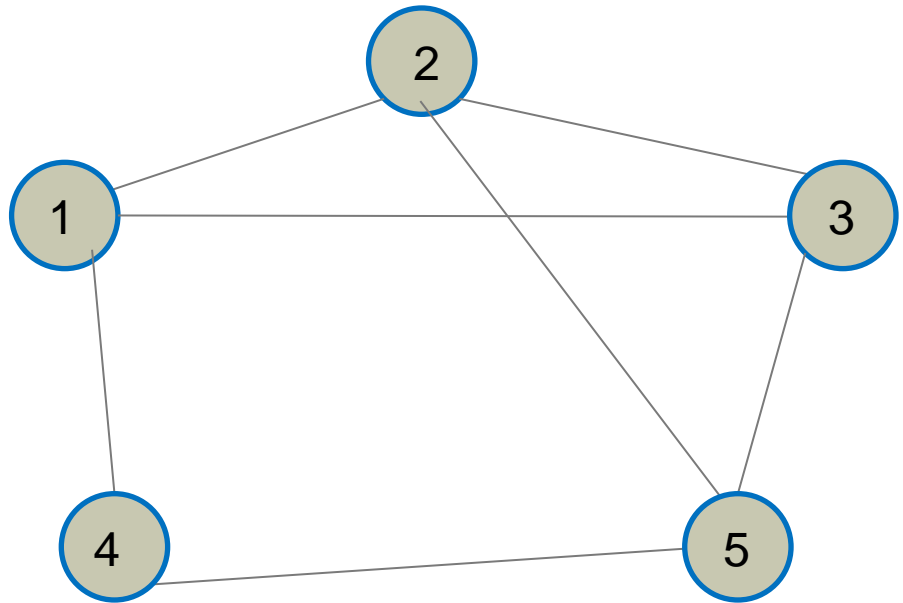
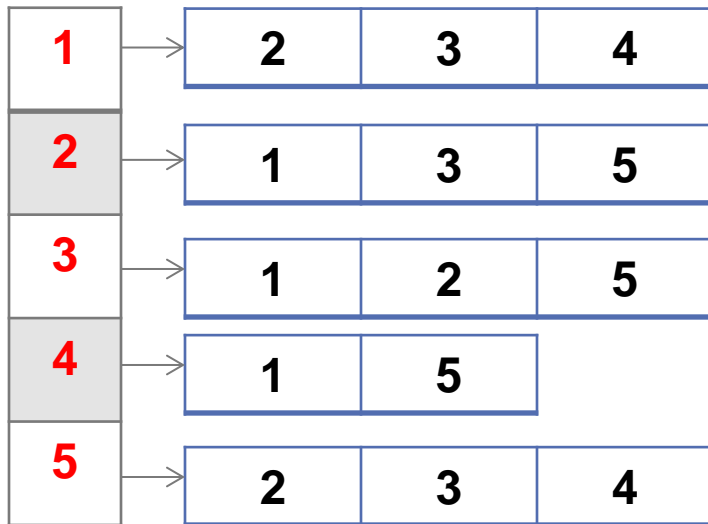
	1	2	3	4	5
1			7	4	
2	2		5		6
3					3
4					8
5					



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.

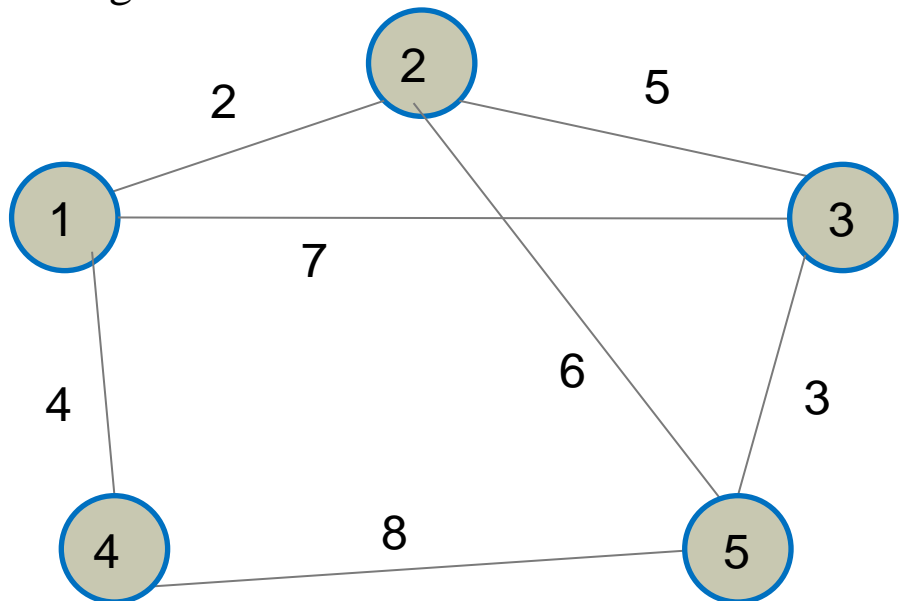
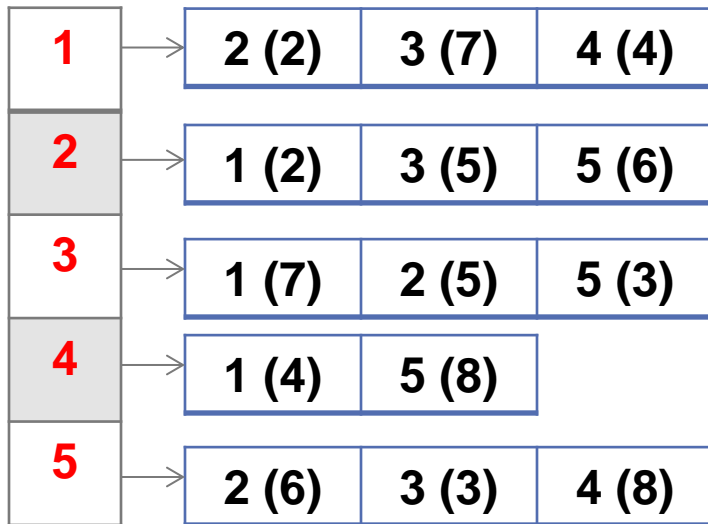


Representing Graphs

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

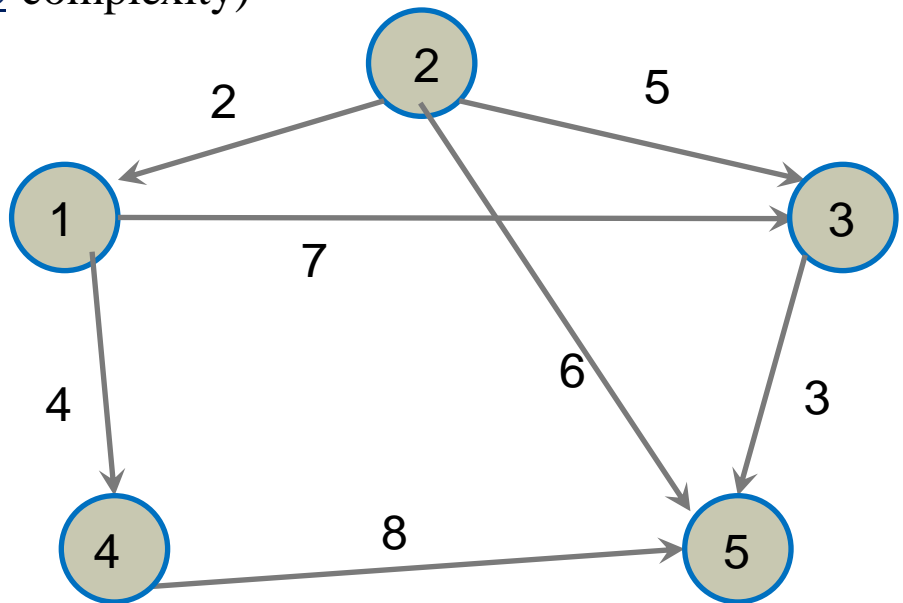
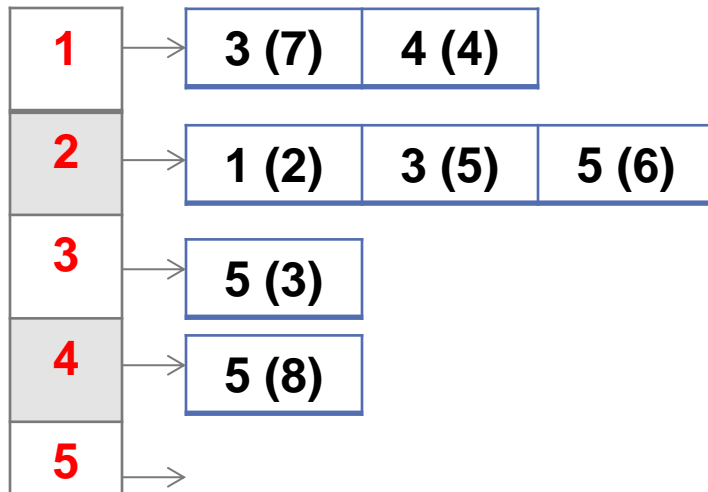
Adjacency List:

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: $O(\log E)$ assuming sorted lists

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



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.

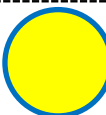
We will focus on single source all targets problem because the single source single target problem is subsumed by it.


Shortest Path Algorithms

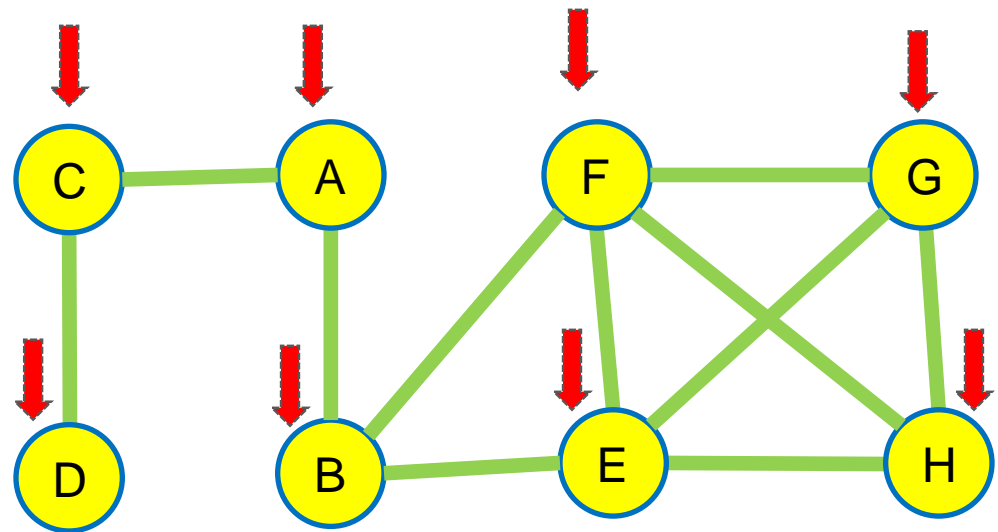
- Breadth First Search – (Unweighted graphs)
- Dijkstra's Algorithm – (Weighted graphs with only non-negative weights)
- Bellman Ford Algorithm – (Weighted graphs including negative weights)

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

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)

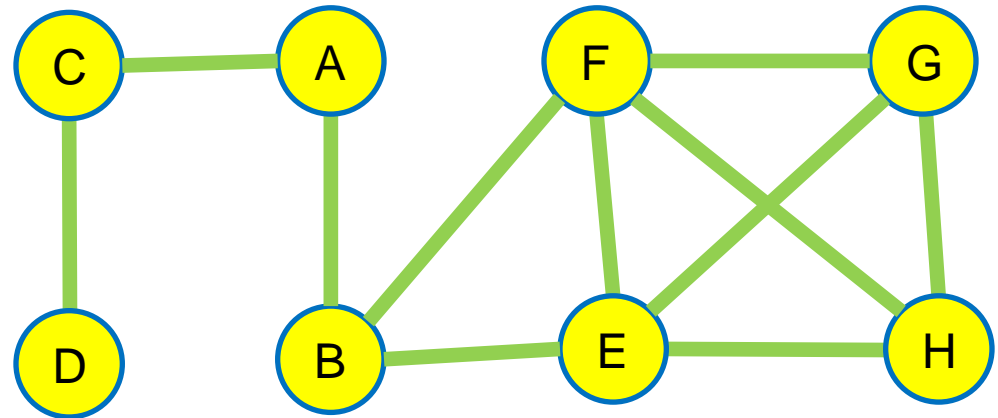
- Initialize a list called Discovered and insert the source node A in it with distance 0
- While Discovered is not empty
 - Get the first vertex v from the Discovered List
 - For each adjacent vertex u of v
 - ✦ If u is not discovered or finalized
 - $u.distance = v.distance + 1$
 - Insert u at the end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

Time Complexity:

$O(V + E)$

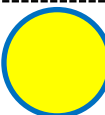
Space Complexity (assuming adjacency list representation):


$O(V + E)$

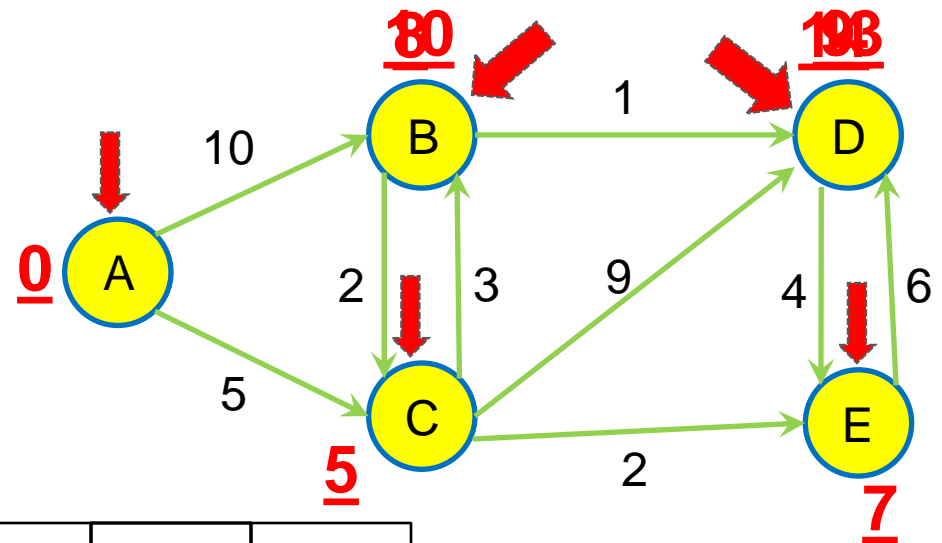


Dijkstra's Algorithm

- 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
 - 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

Discovered: 

Finalized: 



Discovered:

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

Finalized:

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

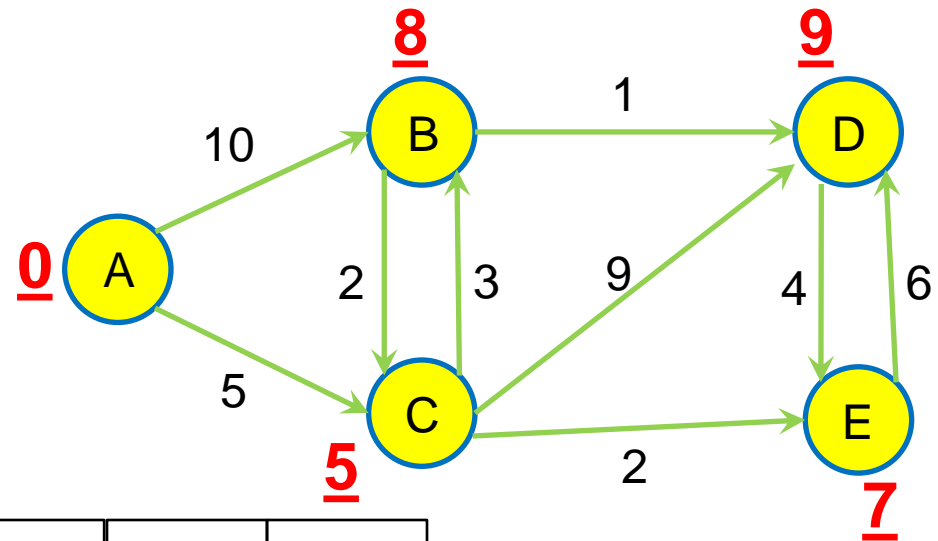
Dijkstra's Algorithm

- 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
 - 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

Suppose we are using a linked list for Discovered.

Time Complexity:

- Each edge visited once $\rightarrow O(E)$
- 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
-----	-----	------	-----	------

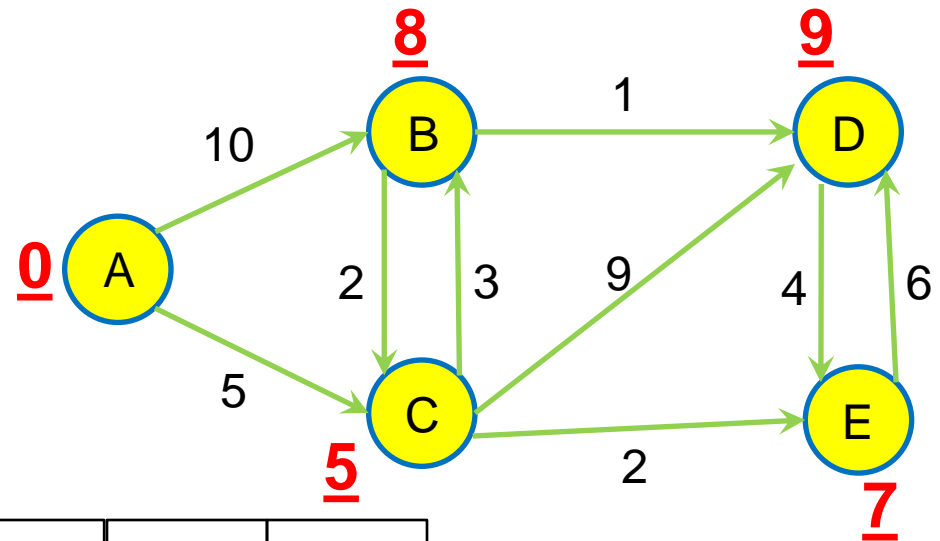
Dijkstra's Algorithm

- 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
 - 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 $O(\log V)$

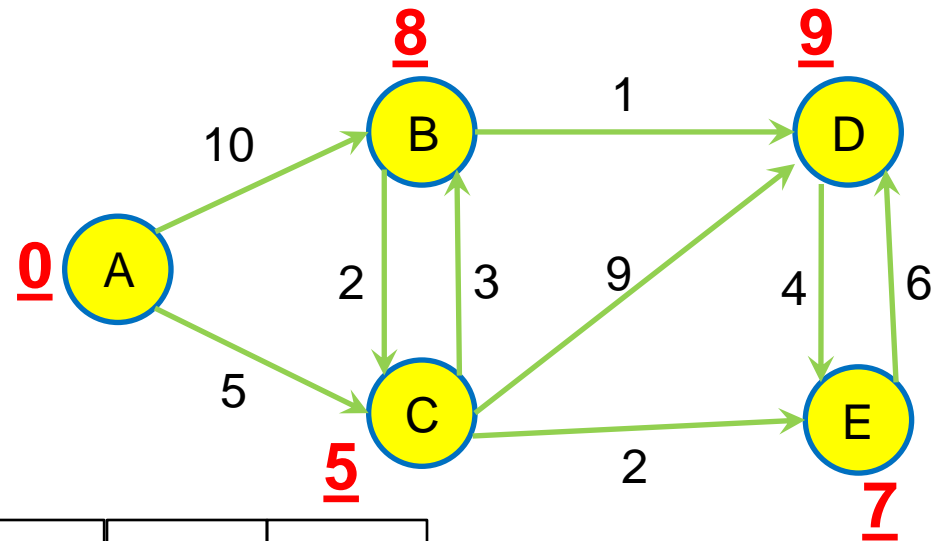
- Let $j = \text{Vertices}[v]$
- Update (i.e., decrease) the key of element at min-heap[j]
- Now upHeap this element (by recursively swapping with parent)
 - For each swap performed between two vertices x and y during the upHeap
 - ✦ Update Vertices[x] and Vertices[y] to record their updated locations in the min-heap

Dijkstra's Algorithm

- 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
 - 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: $O(E \log V + V \log V)$

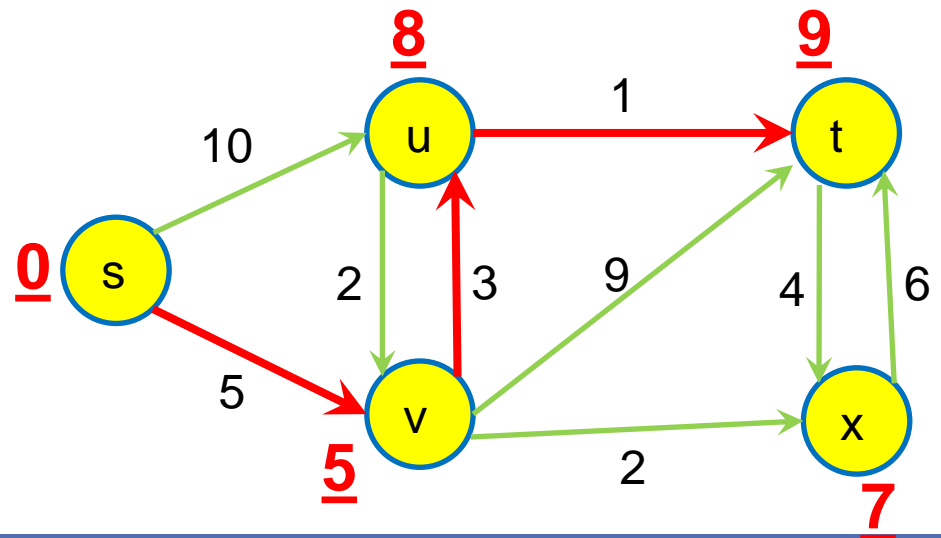


Finalized:

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

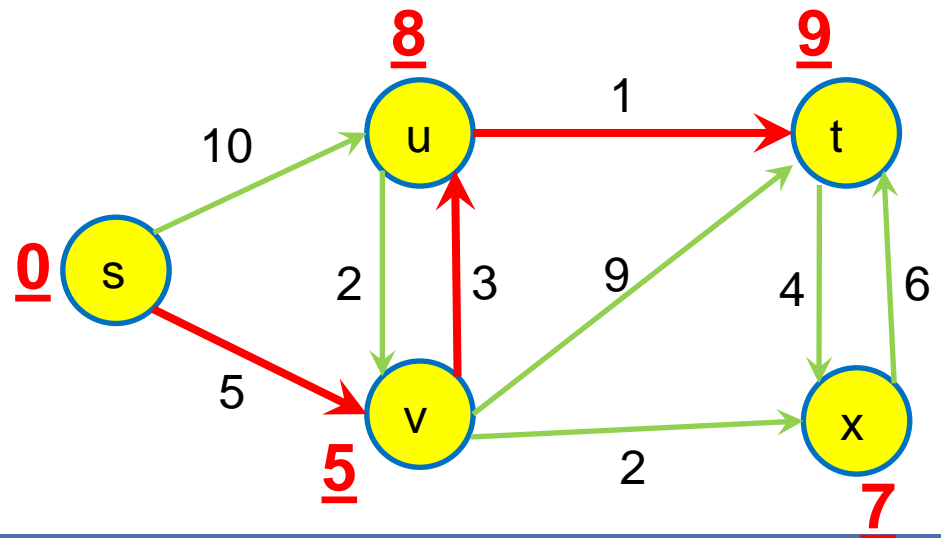
Sketch of the Proof of Correctness

- Suppose $s \rightarrow t$ represents a shortest path from s to t .
- Let u be the last vertex on the shortest path $s \rightarrow t$.
 - Shortest path from s to u $s \rightarrow u$ must be a part of $s \rightarrow t$
 - The shortest path distance of $s \rightarrow u$ is smaller than $s \rightarrow t$.
- Since $s \rightarrow u$ is smaller than $s \rightarrow t$, u is “finalized” before “ t ”.
- Therefore, the shortest path from $s \rightarrow t$ can be obtained by greedily extending previously known shortest paths (i.e., finalized vertices)



Single Source Single Target

- Single source single target problem can be solved using the same algorithms except that the algorithm stops as soon as the target vertex t is finalized.
- The algorithms we saw earlier return only the shortest distances
- The shortest path can be recovered easily by storing, for each vertex u , the previous vertex v that leads to shortest distance



Summary

Take home message

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

Things to do (this list is not exhaustive)

- Read more about BFS and Dijkstra's algorithm and implement these

Coming Up Next

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