MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2004 Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

Referencing materials by
Nathan Companez, Aamir Cheema, Arun Konagurthu and Lloyd Allison

GROUP OF EIGHT AUSTRALIA

Faculty of Information Technology, Monash University

Ready?

Agenda

- The Graph data structure
- Graph Traversal algorithms

Agenda

- The Graph data structure
  - Introduction
  - Representation
- Graph Traversal algorithms

Agenda

- The Graph data structure
  - Introduction
  - Representation
- Graph Traversal algorithms
  - Breadth First Search (BFS)
  - Depth First Search (DFS)
  - Dijkstra's shortest distance

Agenda

- The Graph data structure
  – Introduction
  – Representation
- Graph Traversal algorithms
  – Breadth First Search (BFS)
  – Depth First Search (DFS)

Basic for many graph-algorithms

MONASH University

Let us begin…

- Master race of all data structure

Graph

Introduction

- Master race of all data structure
  - Everything can be represented as a graph

Graph
Introduction

- Master race of all data structure
  - Everything can be represented as a graph
  - Everything can be reduced to a graph

- Master race of all data structure
  - Everything can be represented as a graph
    - Tree is a type of graph
    - Database is a graph
  - Everything can be reduced to a graph

- Master race of all data structure
  - Everything can be represented as a graph
    - Tree is a type of graph
    - Database is a graph
  - Everything can be reduced to a graph
    - You will see this in FIT2014 TOC

Graph
Introduction

- Master race of all data structure
  - Everything can be represented as a graph
    - Tree is a type of graph
    - Database is a graph
  - Everything can be reduced to a graph
    - You will see this in FIT2014 TOC
  - A lot of problems can be solved with a graph
    - The longest common subsequence (LCS) we saw in tutorial?

- Master race of all data structure
  - Everything can be represented as a graph
    - Tree is a type of graph
    - Database is a graph
  - Everything can be reduced to a graph
    - You will see this in FIT2014 TOC
  - A lot of problems can be solved with a graph
    - The longest common subsequence (LCS) we saw in tutorial? Solved by building a directed acyclic graph (DAG)

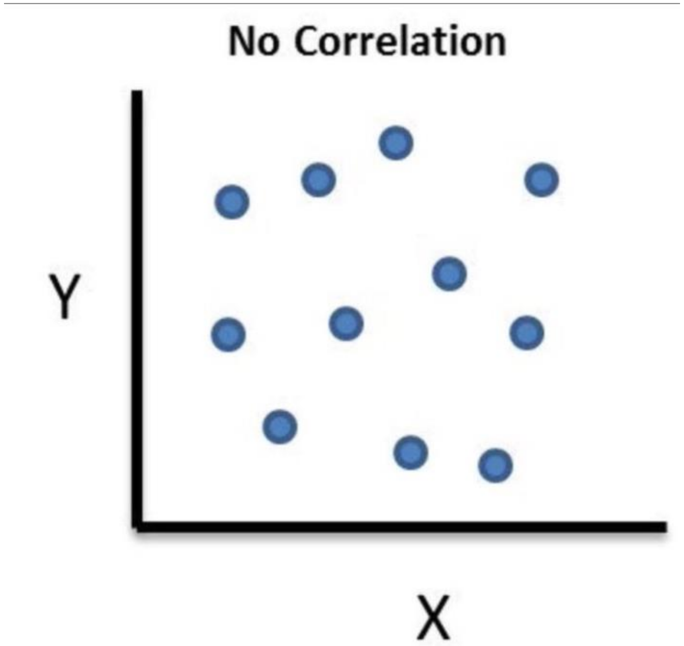
- Master race of all data structure
  - Everything can be represented as a graph
    - Tree is a type of graph
    - Database is a graph
  - Everything can be reduced to a graph
    - You will see this in FIT2014 TOC
  - A lot of problems can be solved with a graph
    - The longest common subsequence (LCS) we saw in tutorial? Solved by building a directed acyclic graph (DAG)
- So what is a graph?
  - Have you seen it before?

Graph
Introduction

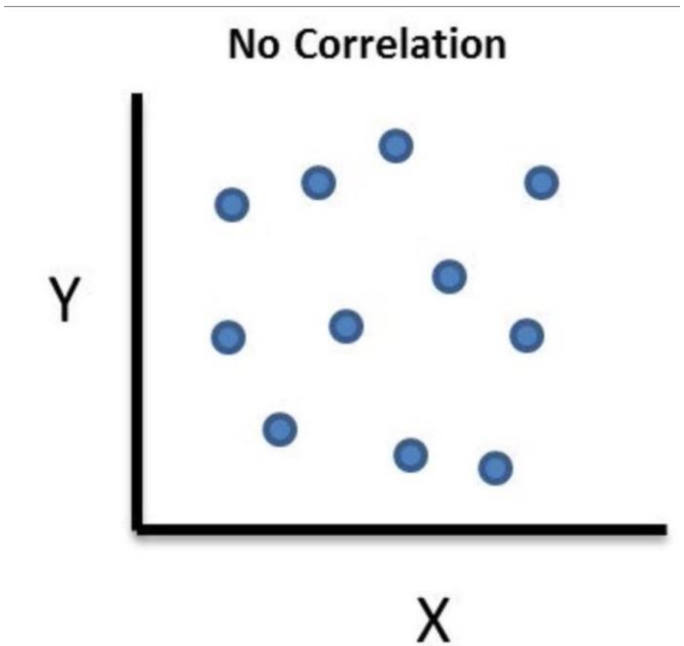
- So what is a graph?
  - Have you seen it before?

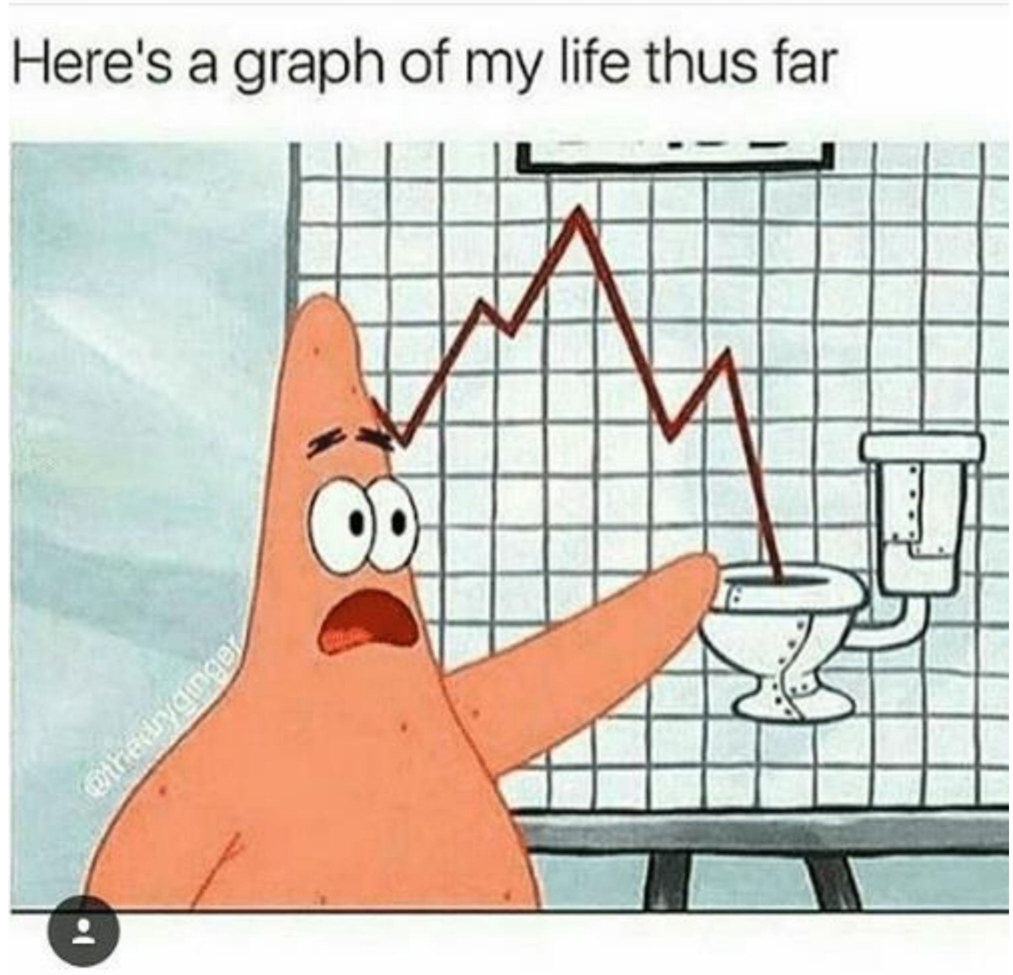
- So what is a graph?
  - Have you seen it before?
  - Was Nickleback right?

- So what is a graph?
  - Have you seen it before?
  - Was Nickleback right?



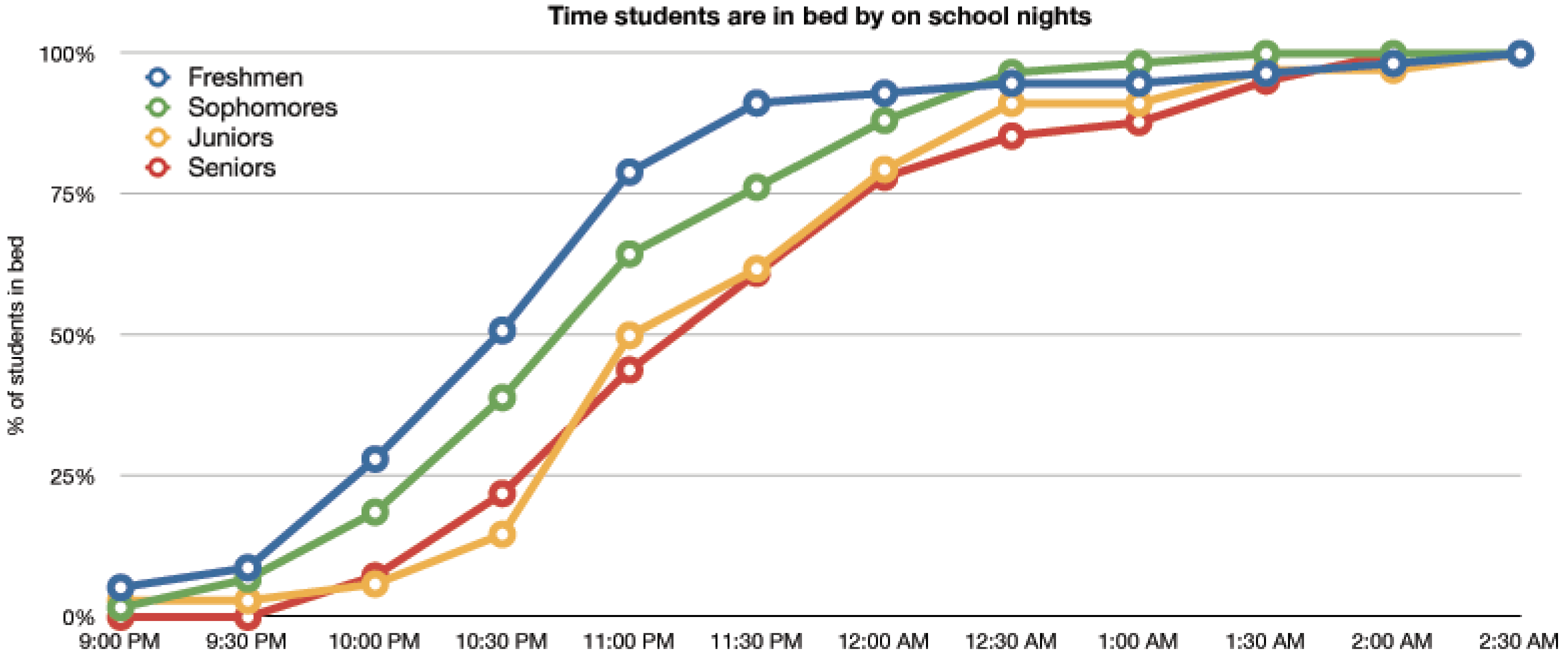
- So what is a graph?
  - Have you seen it before?
  - Was Nickleback right?





- So what is a graph?
- A graph contains
  - Vertex (Vertices)
  - Edge (Edges)

- So what is a graph?
- A graph contains
  - Vertex (Vertices)
  - Edge (Edges)



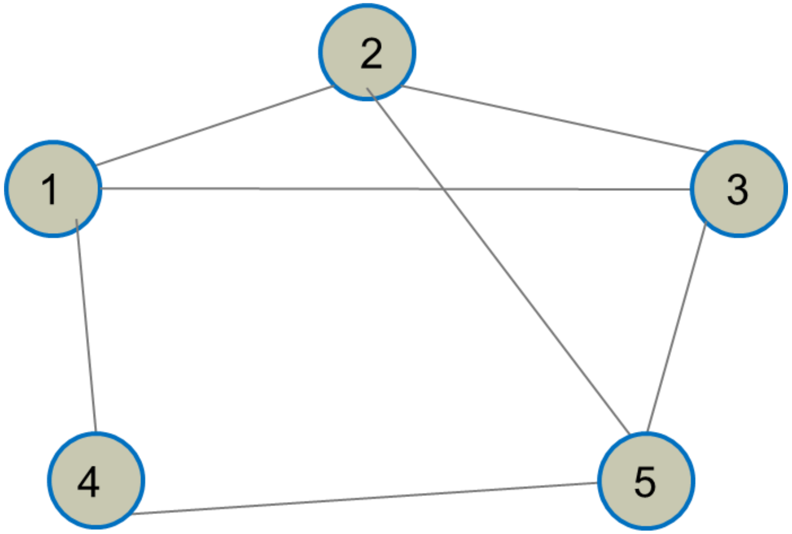
- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
  - Edge (Edges)
    - Link between points

- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
    - Also known as nodes
  - Edge (Edges)
    - Link between points
    - Also known as link

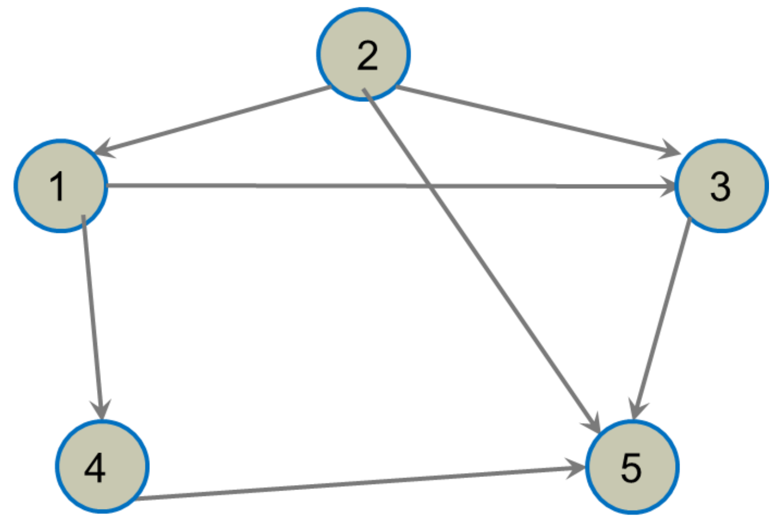
Questions?

- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
    - Also known as nodes
  - Edge (Edges)
    - Link between points
    - Also known as link
    - These links can be directed or undirected

- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
    - Also known as nodes
  - Edge (Edges)
    - Link between points
    - Also known as link
    - These links can be directed or undirected

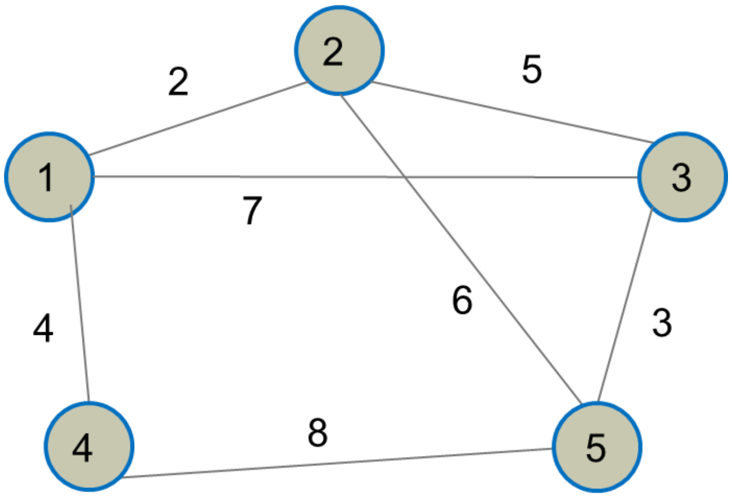


- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
    - Also known as nodes
  - Edge (Edges)
    - Link between points
    - Also known as link
    - These links can be directed or undirected



Graph
Introduction

- So what is a graph?
- A graph contains
  - Vertex (Vertices)
    - Points
    - Also known as nodes
  - Edge (Edges)
    - Link between points
    - Also known as link
    - These links can be directed or undirected
    - These links can be weighted or unweighted



Questions?

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E

Formal definitions

- A graph, G=(V,E)
  – Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  – u and v are both vertices

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  - u and v are both vertices
  - If directed, the edge goes from u to v

- A graph, $G=(V,E)$
  - Contains set of vertices V and a set of Edges E
- An edge, $e=(u,v)$
  - u and v are both vertices
  - If directed, the edge goes from u to v
- A weighted graph, $G=(V,E,W)$

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  - u and v are both vertices
  - If directed, the edge goes from u to v
- A weighted graph, G=(V,E,W)
  - W are the weights

Graph
Formal definitions

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  - u and v are both vertices
  - If directed, the edge goes from u to v
  - If weighted, e=(u,v,w)
- A weighted graph, G=(V,E,W)
  - W are the weights

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  - u and v are both vertices
  - If directed, the edge goes from u to v
  - If weighted, e=(u,v,w)
- A weighted graph, G=(V,E,W)
  - W are the weights

- Simple graph
  - No self-edges arrow from itself to itself
  - No multi-edges between vertices
    no a to b, then b to a

- A graph, G=(V,E)
  - Contains set of vertices V and a set of Edges E
- An edge, e=(u,v)
  - u and v are both vertices
  - If directed, the edge goes from u to v
  - If weighted, e=(u,v,w)
- A weighted graph, G=(V,E,W)
  - W are the weights
- Simple graph
  - No self-edges (known as loops also)
  - No multi-edges between vertices

Questions?

- Let say we have graph, G=(V,E)
  - |V| is the number of vertices
  - |E| is the number of edges

- Let say we have graph, G=(V,E)
  - |V| is the number of vertices
  - |E| is the number of edges

- Maximum number of edges?
  - Directed graph?

Graph
Properties

- Let say we have graph, G=(V,E)
  - |V| is the number of vertices
  - |E| is the number of edges

- Maximum number of edges?
  - Directed graph? V(V-1) = O(V^2)

    [V] * ([V] -1)

- Let say we have graph, G=(V,E)
  - |V| is the number of vertices
  - |E| is the number of edges

  undirected graph [V] = [E]
- Maximum number of edges?
  - Directed graph? V(V-1) = O(V^2)
  - Undirected graph?

- Let say we have graph, G=(V,E)
  - |V| is the number of vertices
  - |E| is the number of edges

- Maximum number of edges?
  - Directed graph? V(V-1) = O(V^2)
  - Undirected graph? V(V-1)/2 = O(V^2)

- Let say we have graph, $G=(V,E)$
  - $|V|$ is the number of vertices
  - $|E|$ is the number of edges

- Maximum number of edges?
  - Directed graph? $V(V-1) = O(V^2)$
  - Undirected graph? $V(V-1)/2 = O(V^2)$

- A graph is called sparse if $E \ll V^2$
- A graph is called dense if $E \approx V^2$

Monash University

Questions?

- We saw earlier how we can represent anything as a graph…

Graph

Importance

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.

Graph
Importance

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!
    - Later, we will see how to build a optimal tree from a graph =)

Graph
Importance

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!
    - Later, we will see how to build a optimal tree from a graph =)
  - The World Wide Web (WWW) are in fact a graph

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!
    - Later, we will see how to build a optimal tree from a graph =)
  - The World Wide Web (WWW) are in fact a graph
    - Each webpage is a vertex
    - Each hyperlink is an edge

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!
    - Later, we will see how to build a optimal tree from a graph =)
  - The World Wide Web (WWW) are in fact a graph
    - Each webpage is a vertex
    - Each hyperlink is an edge
    - Google's PageRank is a graph algorithm

- We saw earlier how we can represent anything as a graph…
  - Trees are in fact graphs.
    - Single vertex with no incoming edge is the root.
    - A binary tree has vertices with a maximum of 2 outgoing edge only.
    - There is no cycle!
    - Later, we will see how to build a optimal tree from a graph =)
  - The World Wide Web (WWW) are in fact a graph
    - Each webpage is a vertex
    - Each hyperlink is an edge
    - Google's PageRank is a graph algorithm
      - Traversal through webpages and propagate authority
      - You can code it yourself, it is easy!

Questions?

Graph
Representation

- How do we represent graphs?

Graph
Representation

- How do we represent graphs? 2 possible way!

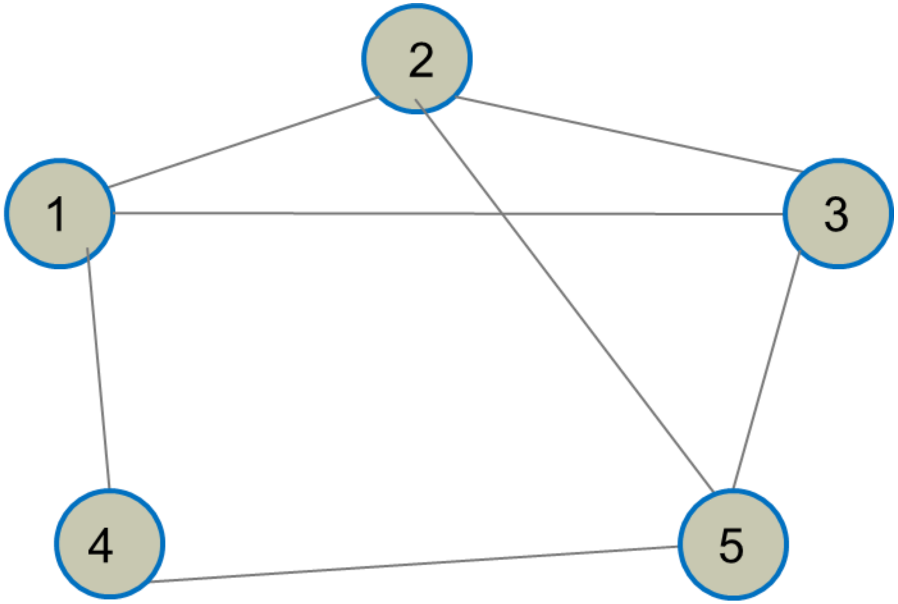
Graph
Representation

- How do we represent graphs? 2 possible way!
  - Adjacency matrix
  - Adjacency list prefer this a lot of time

- Adjacency matrix
  - Store edge information in a matrix

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted

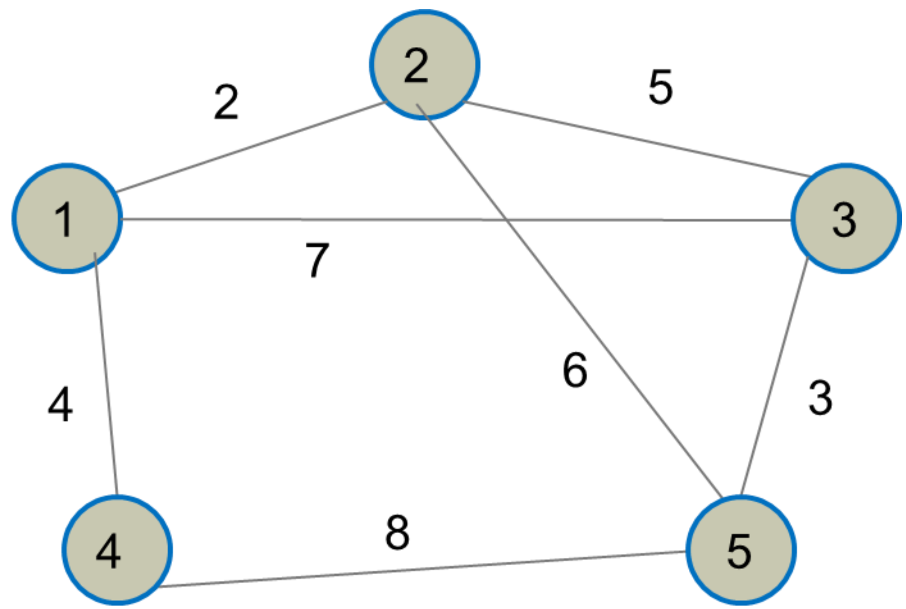
	1	2	3	4	5
1	F	T	T	T	F
2	T	F	T	F	T
3	T	T	F	F	T
4	T	F	F	F	T
5	F	T	T	T	F



Graph Representation

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted

	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
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?
    - O(V^2) as we need the matrix no matter what

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?
    - O(V^2) as we need the matrix no matter what
  - Time complexity?

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?
    - O(V^2) as we need the matrix no matter what
  - Time complexity?
    - O(1) to check if an edge exist

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?
    - O(V^2) as we need the matrix no matter what
  - Time complexity?
    - O(1) to check if an edge exist
    - O(V) to check/ traverse all adjacent vertices

- Adjacency matrix
  - Store edge information in a matrix
    - True/ False or 1/0 for unweighted
    - Weight for weighted
  - Space complexity?

    good-dense graph, bad-sparse graph
    - O(V^2) as we need the matrix no matter what
  - Time complexity?
    - O(1) to check if an edge exist
    - O(V) to check/ traverse all adjacent vertices
      - Adjacent = neighbour

        have a edge

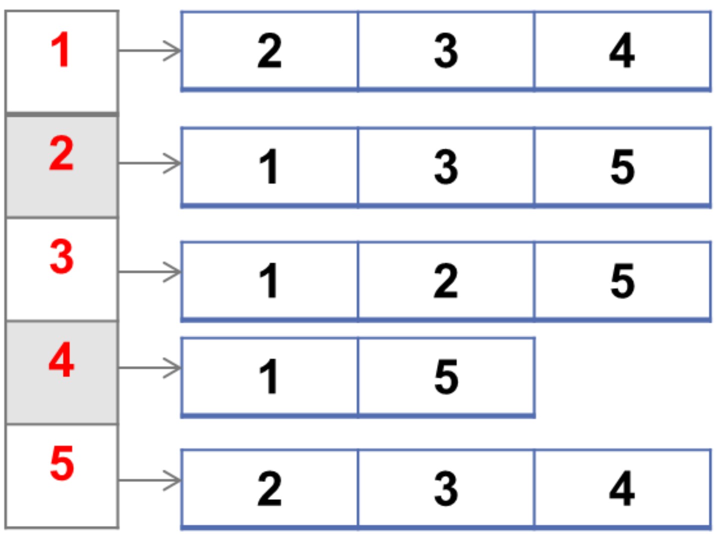
- Adjacency list

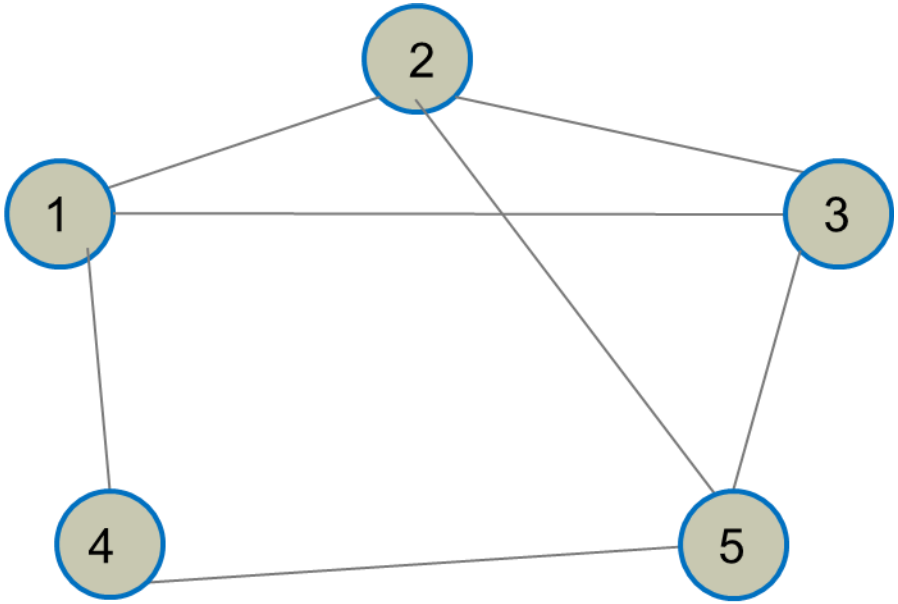
Graph
Representation

- Adjacency list
  - Array to store all Vertex object

- Adjacency list
  – Array to store all Vertex object
  – Each vertex store a list of edges from it

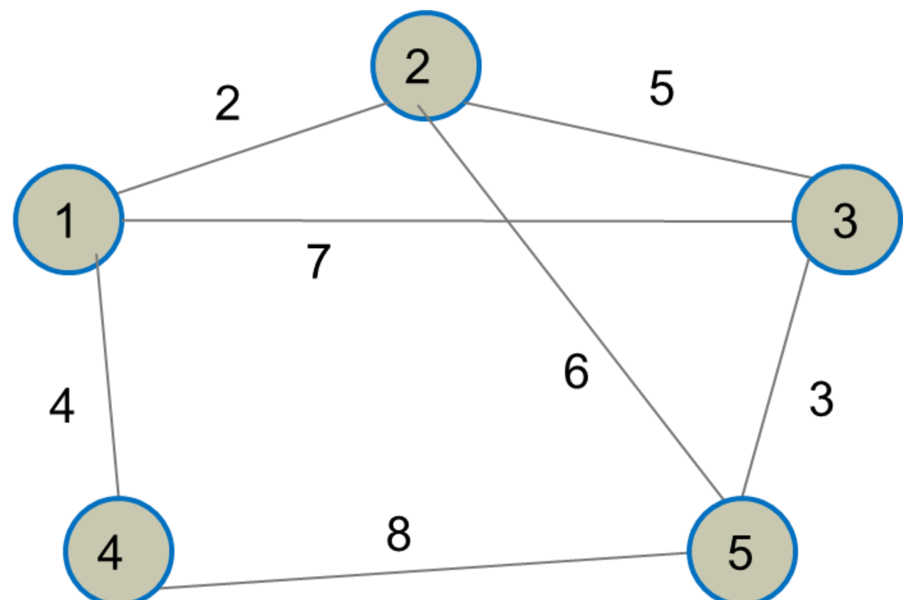
- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it





- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With 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)



Graph
Representation

- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With the weights
  - Space complexity?

- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With the weights
  - Space complexity?
    - O(V+E). Storing V vertices (as an array) and then total of E edges

MONASH University

- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With the weights
  - Space complexity?
    - O(V+E). Storing V vertices (as an array) and then total of E edges
      not O(VE) E = V[V-1), only link to every other vertices
  - Time complexity?

Graph

Representation

- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With the weights
  - Space complexity?
    - O(V+E). Storing V vertices (as an array) and then total of E edges
  - Time complexity?

    binary search
    - O(log V) to check if an edge exist if the edges are sorted
    - O(X) to retrieve all of the adjacent vertices of a vertex

      when traverse use adjacency list

    since graph can be sparse, adjacency matrix would need to loop through until biggest vertex
    but Adjacency List has length of list, number of links to other vertices

MONASH University

- Adjacency list
  - Array to store all Vertex object
  - Each vertex store a list of edges from it
    - With the weights
  - Space complexity?
    - O(V+E). Storing V vertices (as an array) and then total of E edges
  - Time complexity?
    - O(log V) to check if an edge exist if the edges are sorted
      - But you can't use binary search on linked list!
      - So this is still O(X) but you can terminate earlier once you reach a bigger vertex
    - O(X) to retrieve all of the adjacent vertices of a vertex
      - Where X = number of adjacent vertices (output-sensitive complexity)

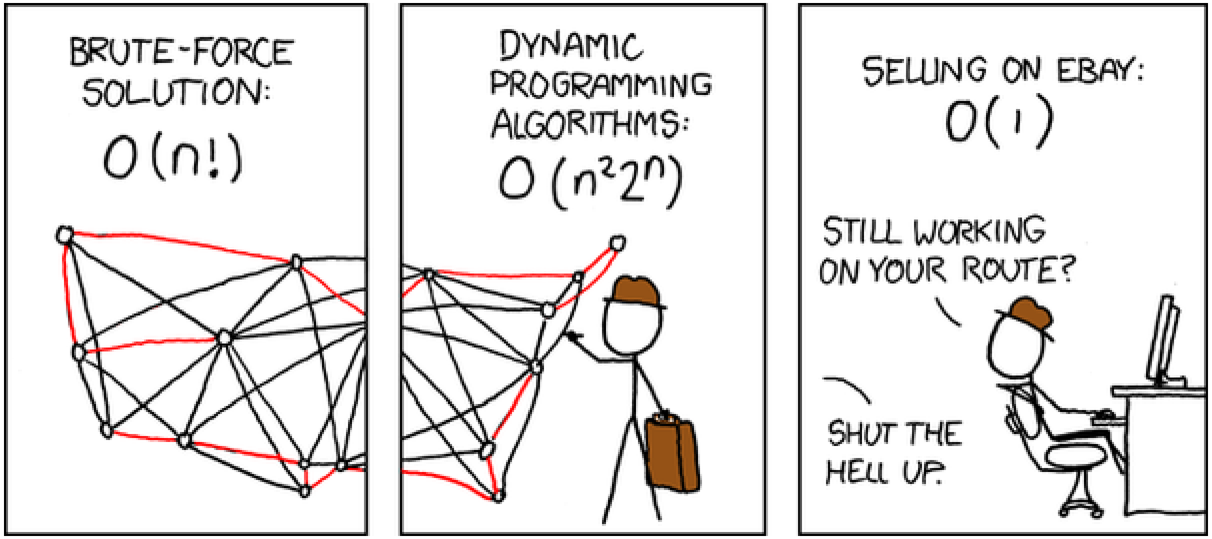
Questions?

- Going from a place to another

Graph
Traversal

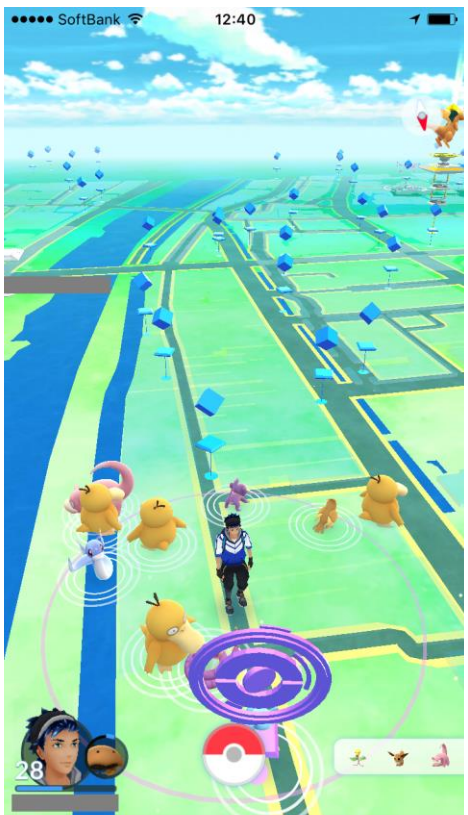
- Going from a place (source vertex) to another

- Going from a place (source vertex) to another or everywhere!



Graph
Traversal

- Going from a place (source vertex) to another or everywhere!



Graph
Traversal

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

- Breadth-First Search (BFS)
  - Going wide
- Depth-First Search (DFS)
  - Going deep

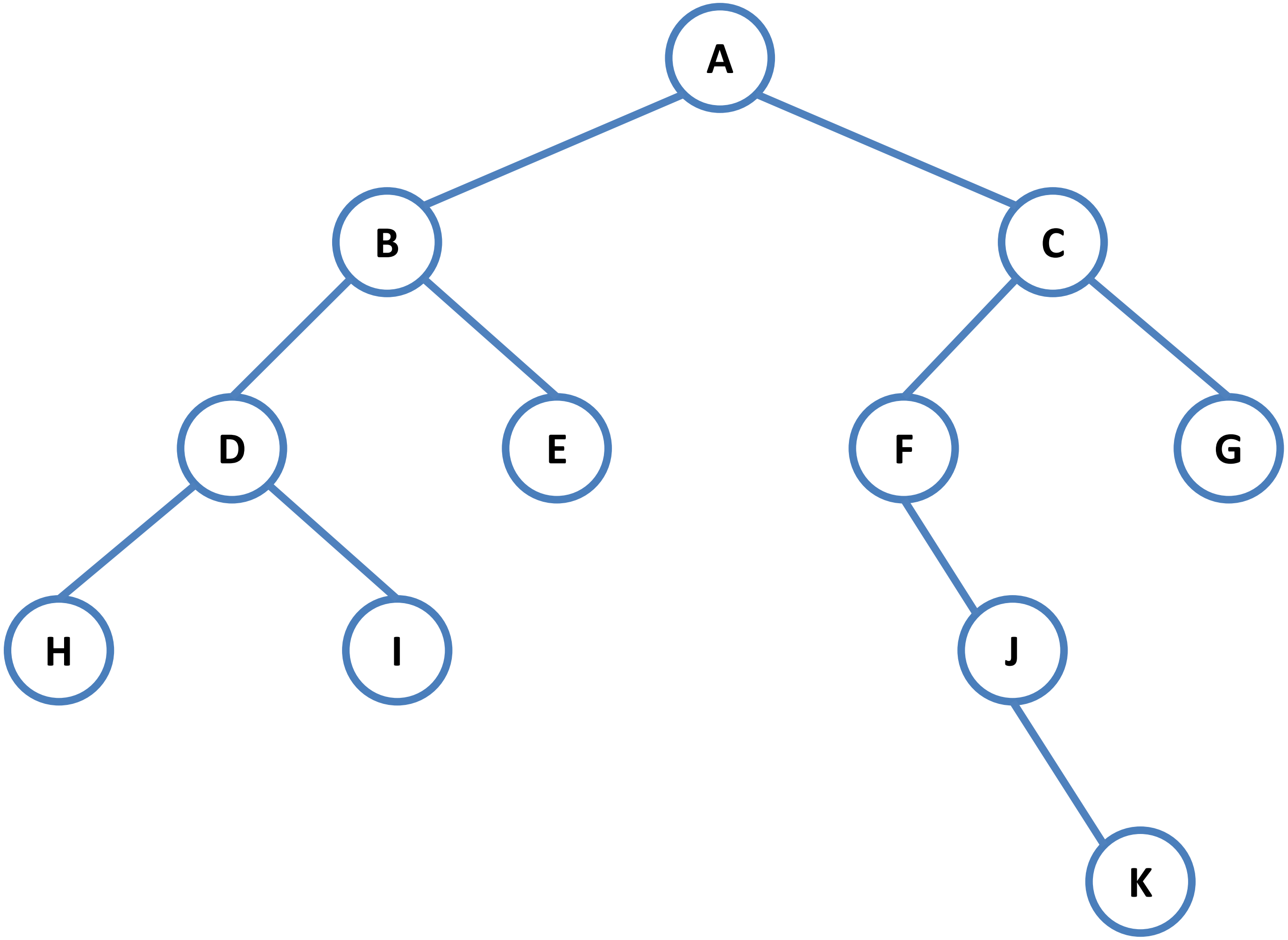
- Breadth-First Search (BFS)
  - Going wide
- Depth-First Search (DFS)
  - Going deep

- Let us begin with a tree first

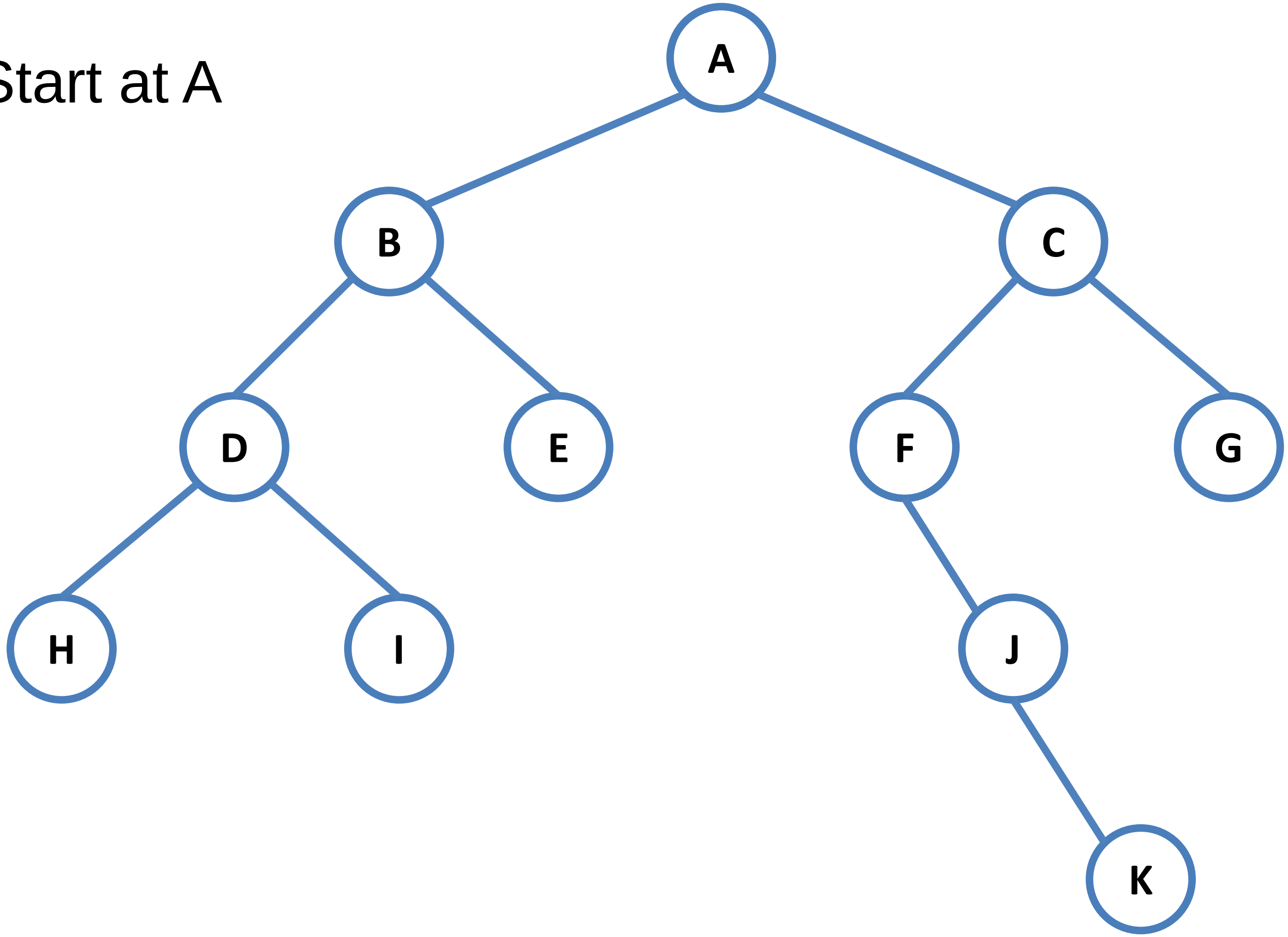
- Breadth-First Search (BFS)
  – Going wide
- Depth-First Search (DFS)
  – Going deep

- Let us begin with a tree first
  – Recall a tree is a graph without cycles

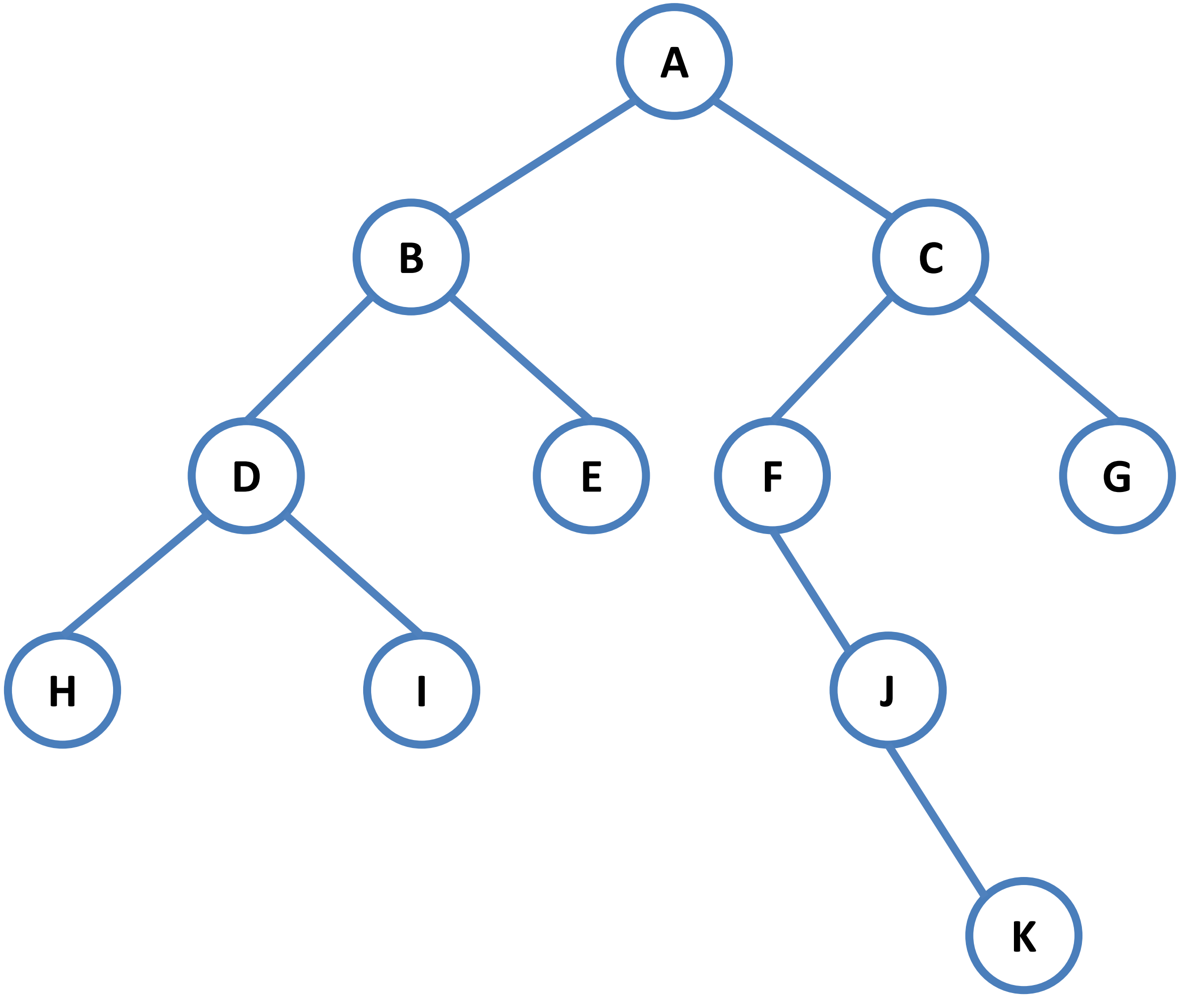
Graph
Traversal



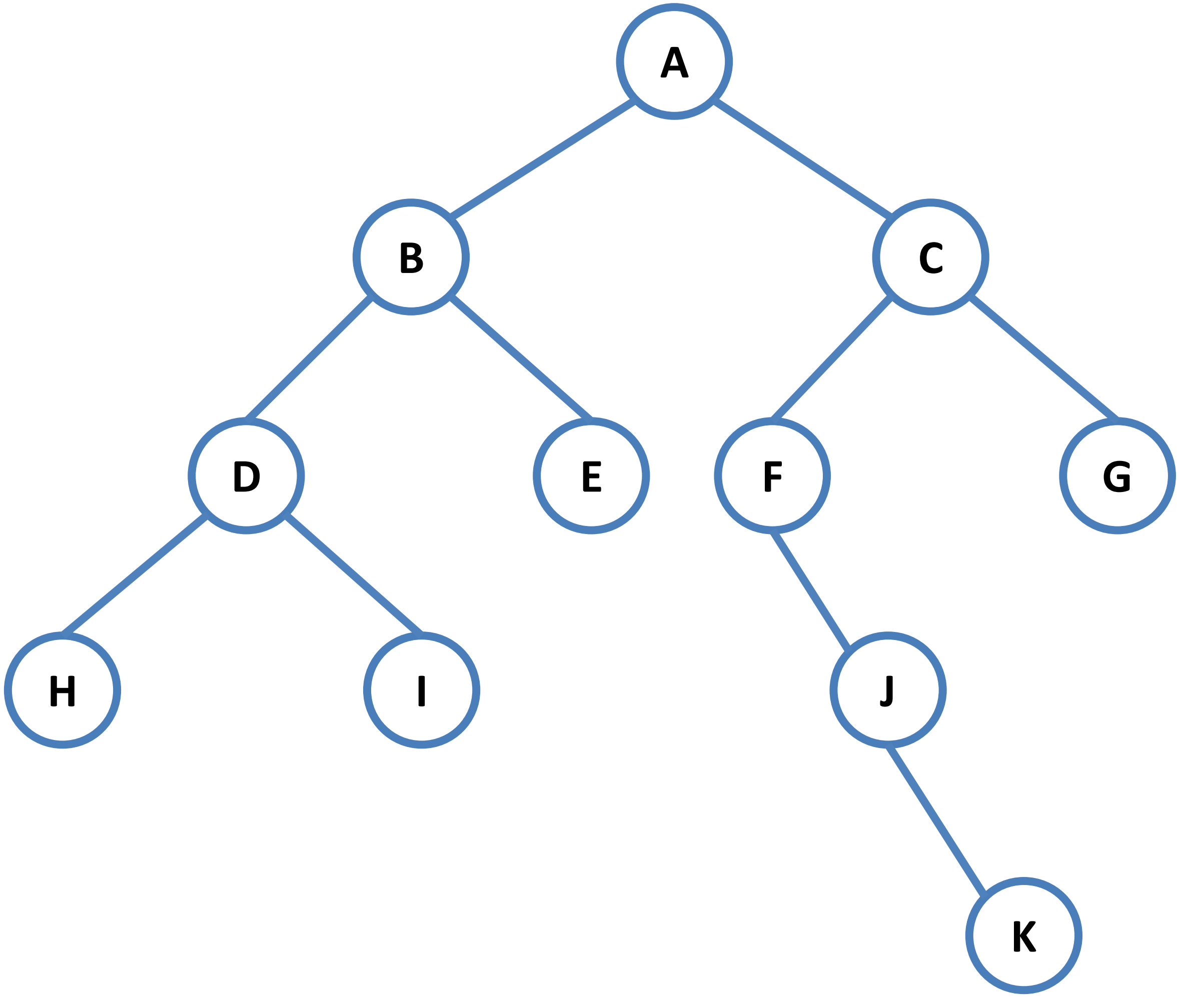
- Start at A



- Start at A

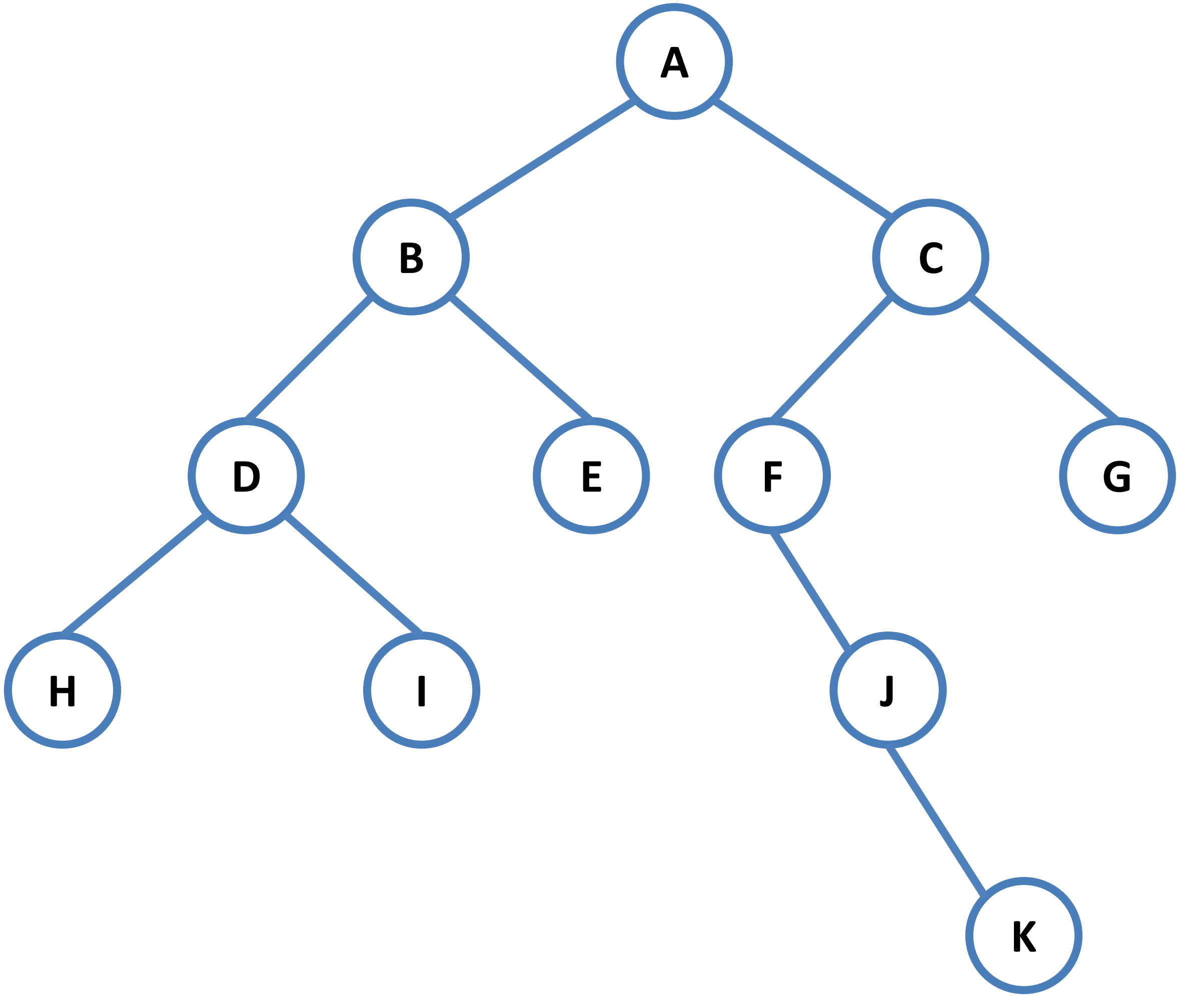


- Start at A, BFS



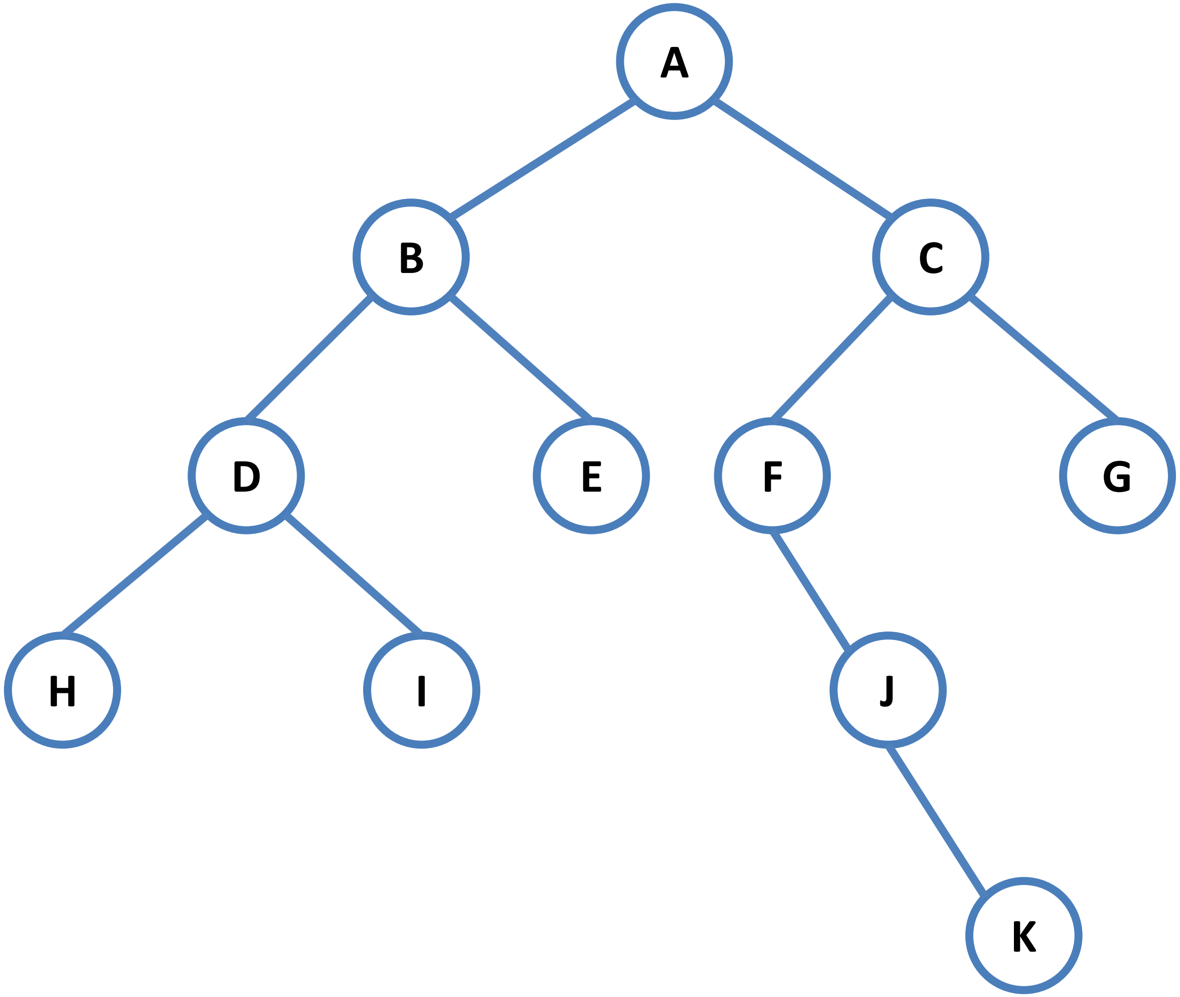
Graph
Traversal

- Start at A, BFS
- A

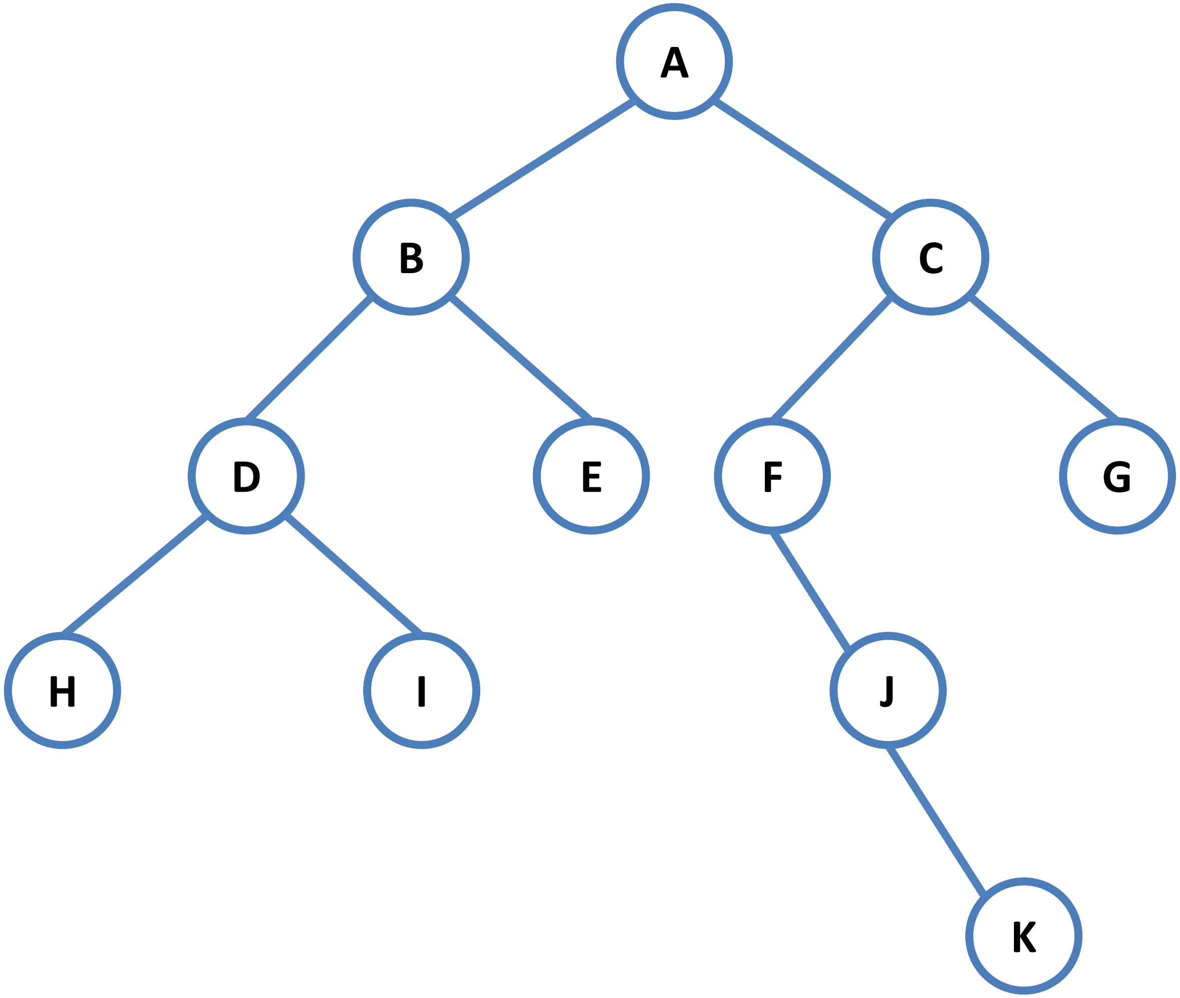


MONASH University

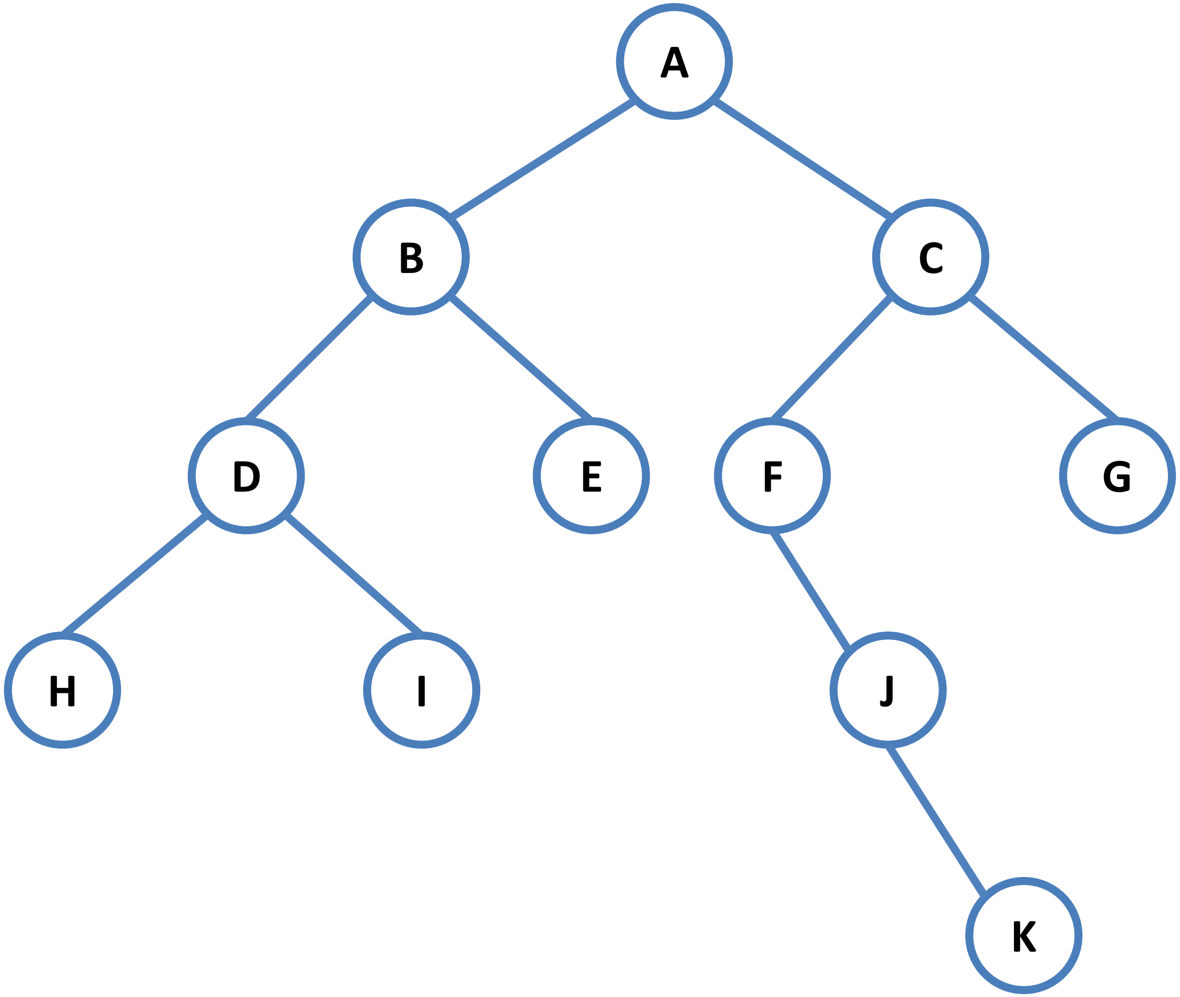
- Start at A, BFS
- A
- B



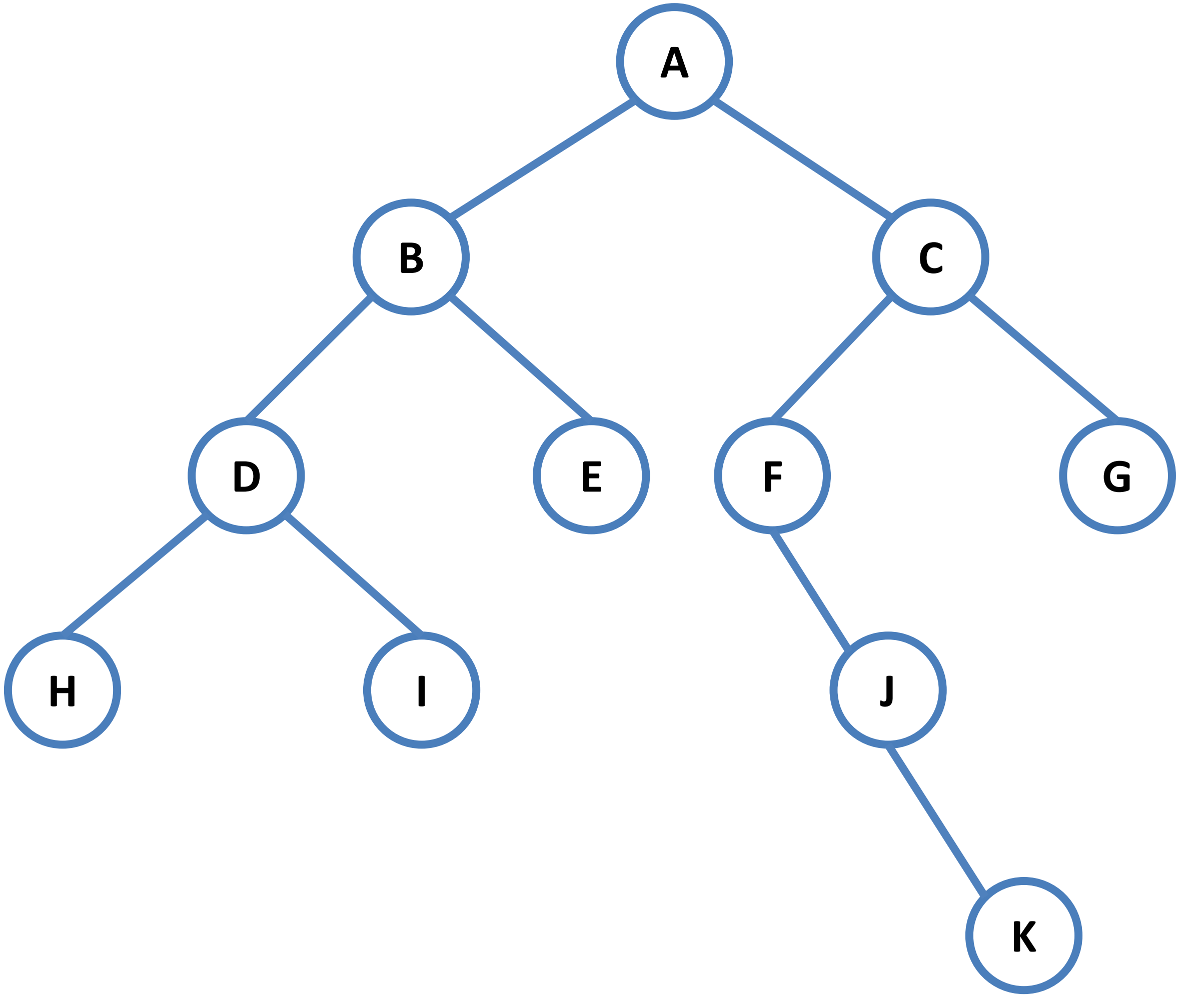
- Start at A, BFS
- A
- B
- C



- Start at A, BFS
- A
- B
- C
- D



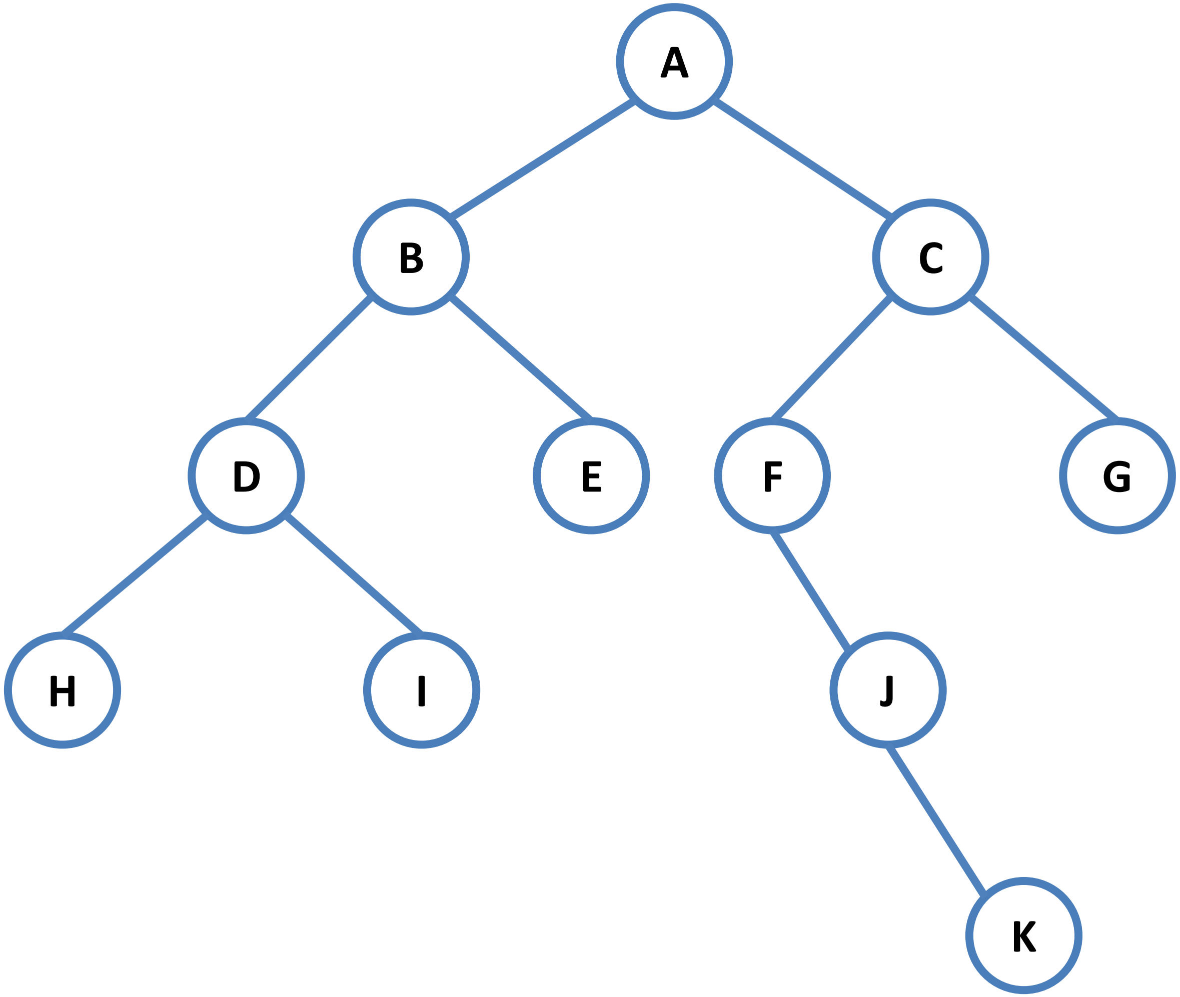
- Start at A, BFS
- A
- B
- C
- D
- E



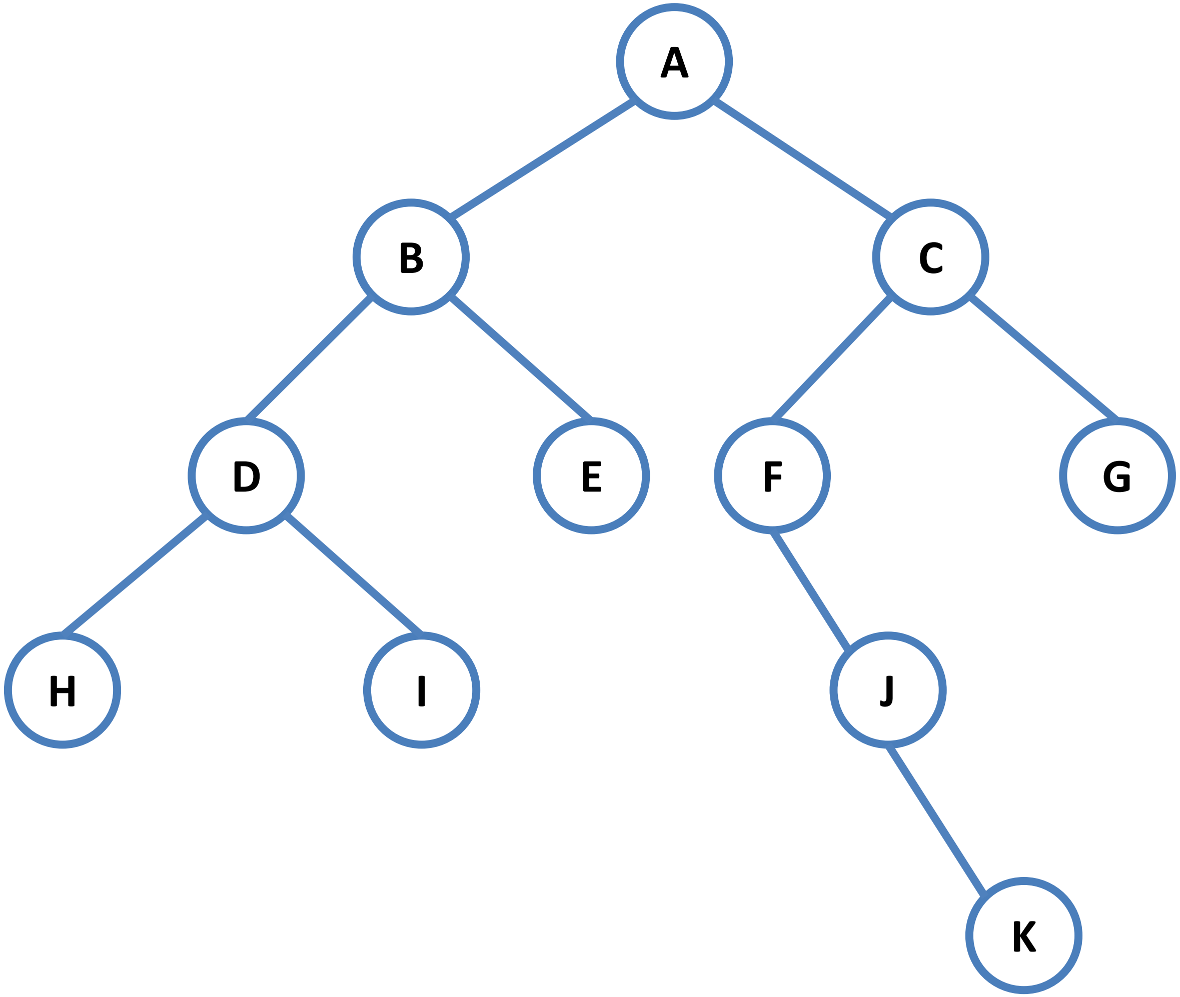
Graph

Traversal

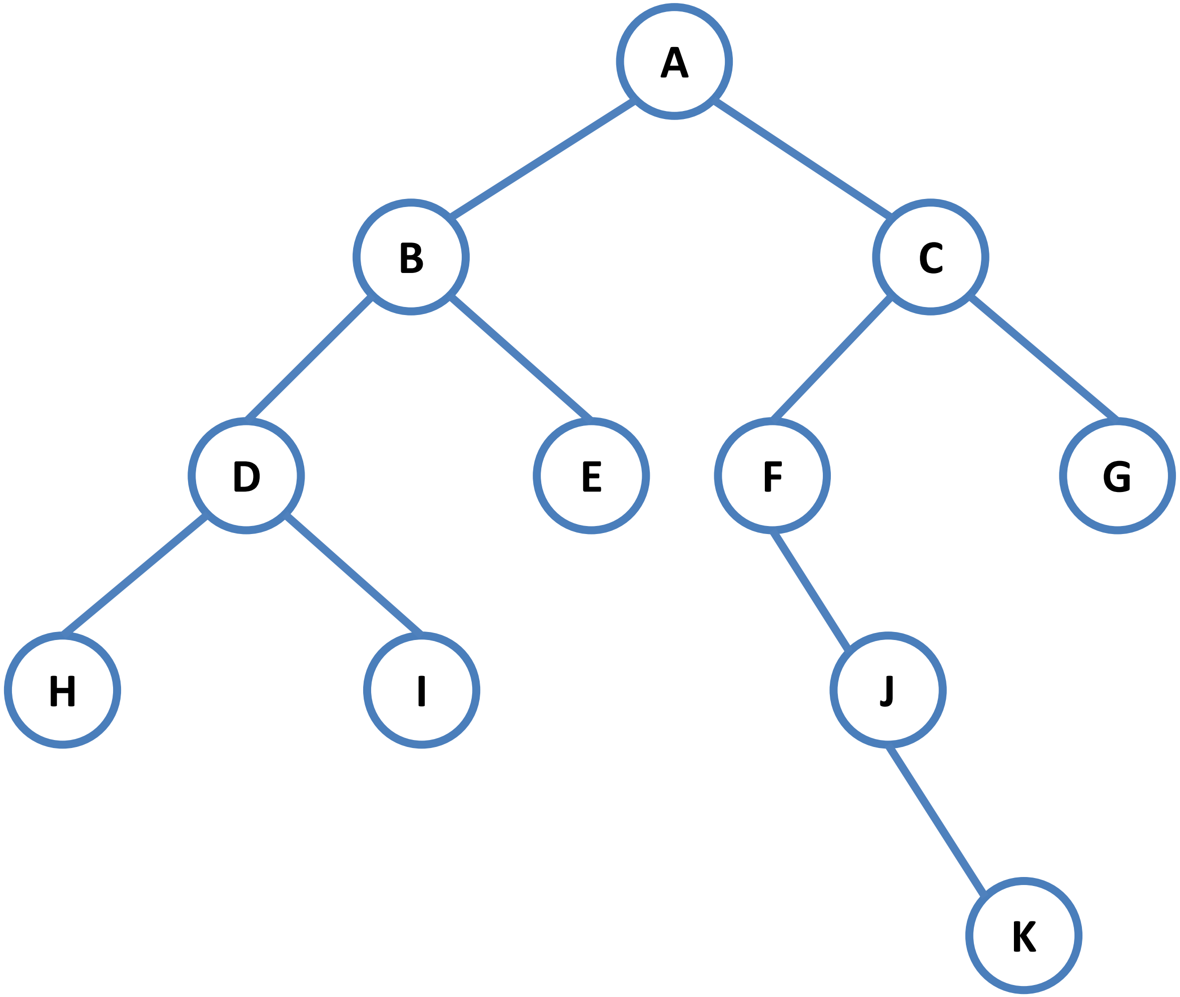
- Start at A, BFS
- A
- B
- C
- D
- E
- F



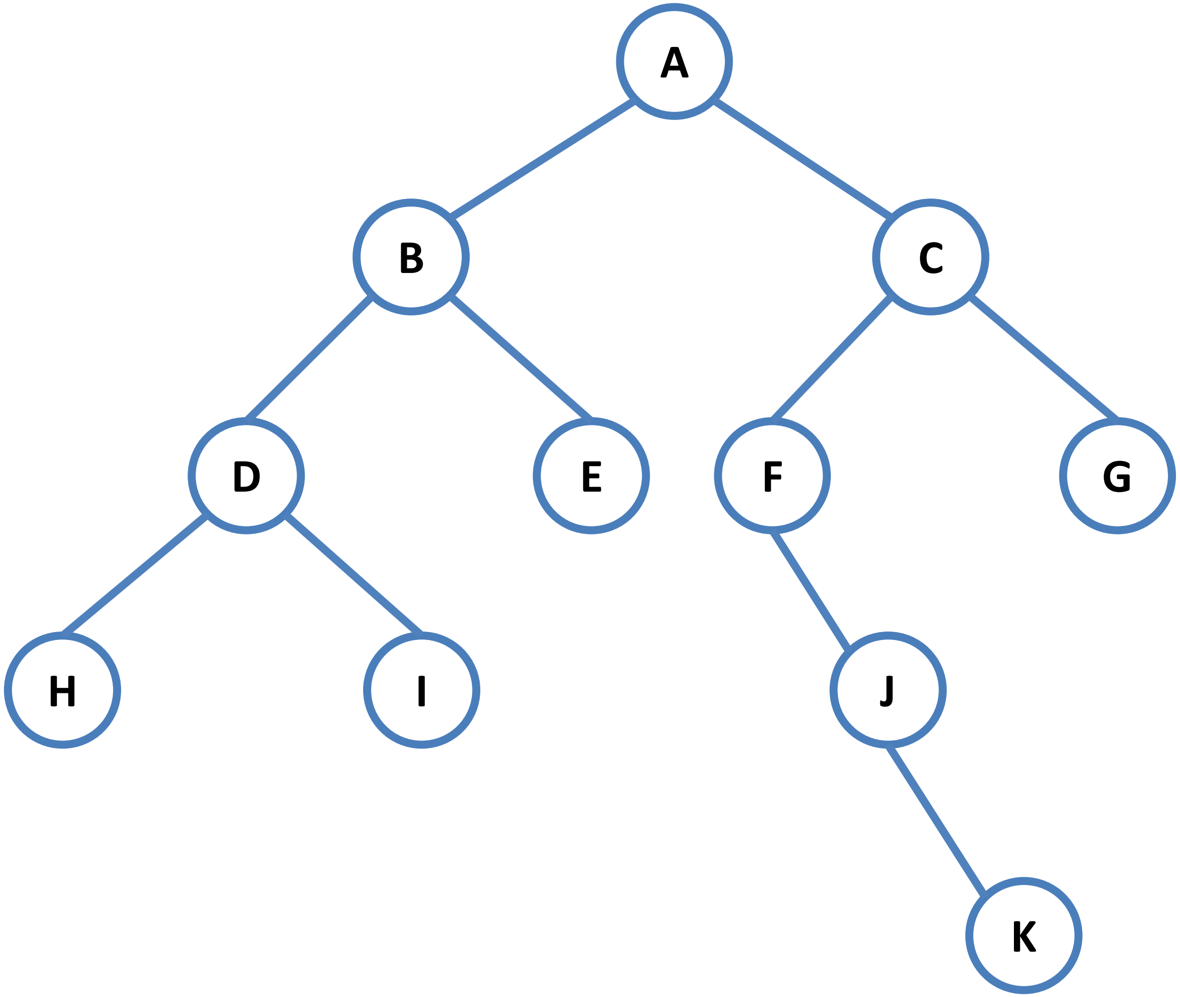
- Start at A, BFS
- A
- B
- C
- D
- E
- F
- G



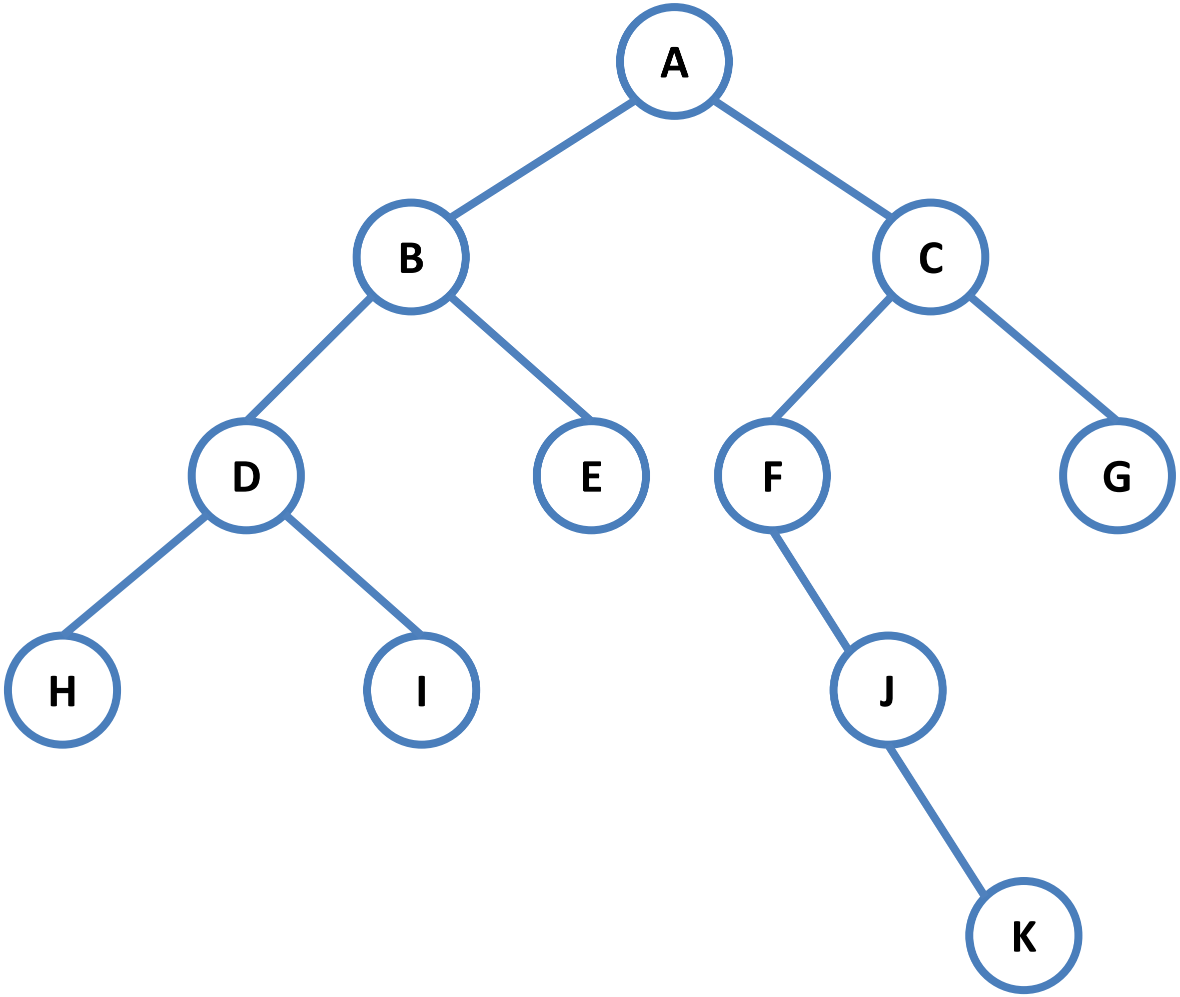
- Start at A, BFS
- A
- B
- C
- D
- E
- F
- G
- … and so on…



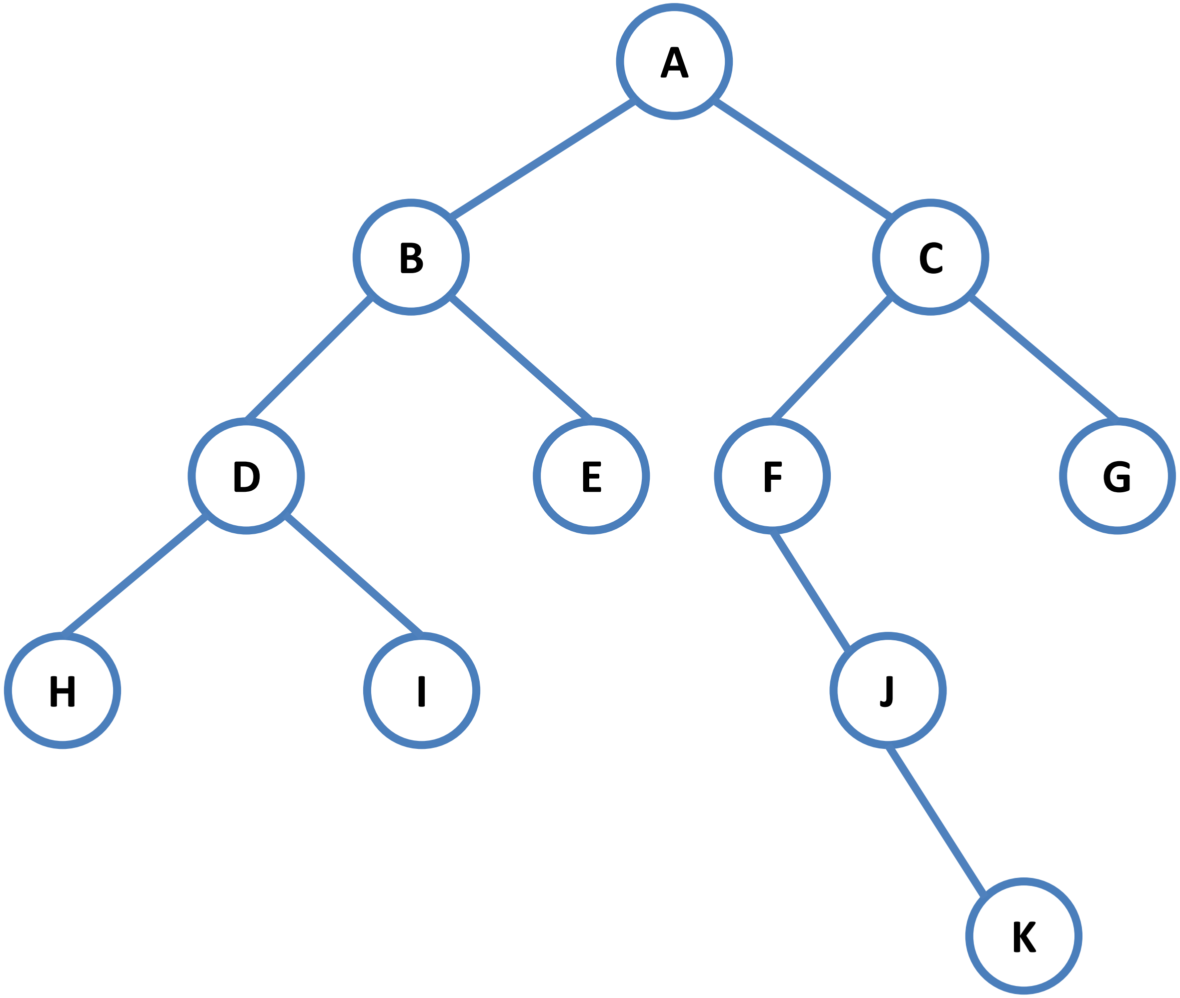
- Start at A



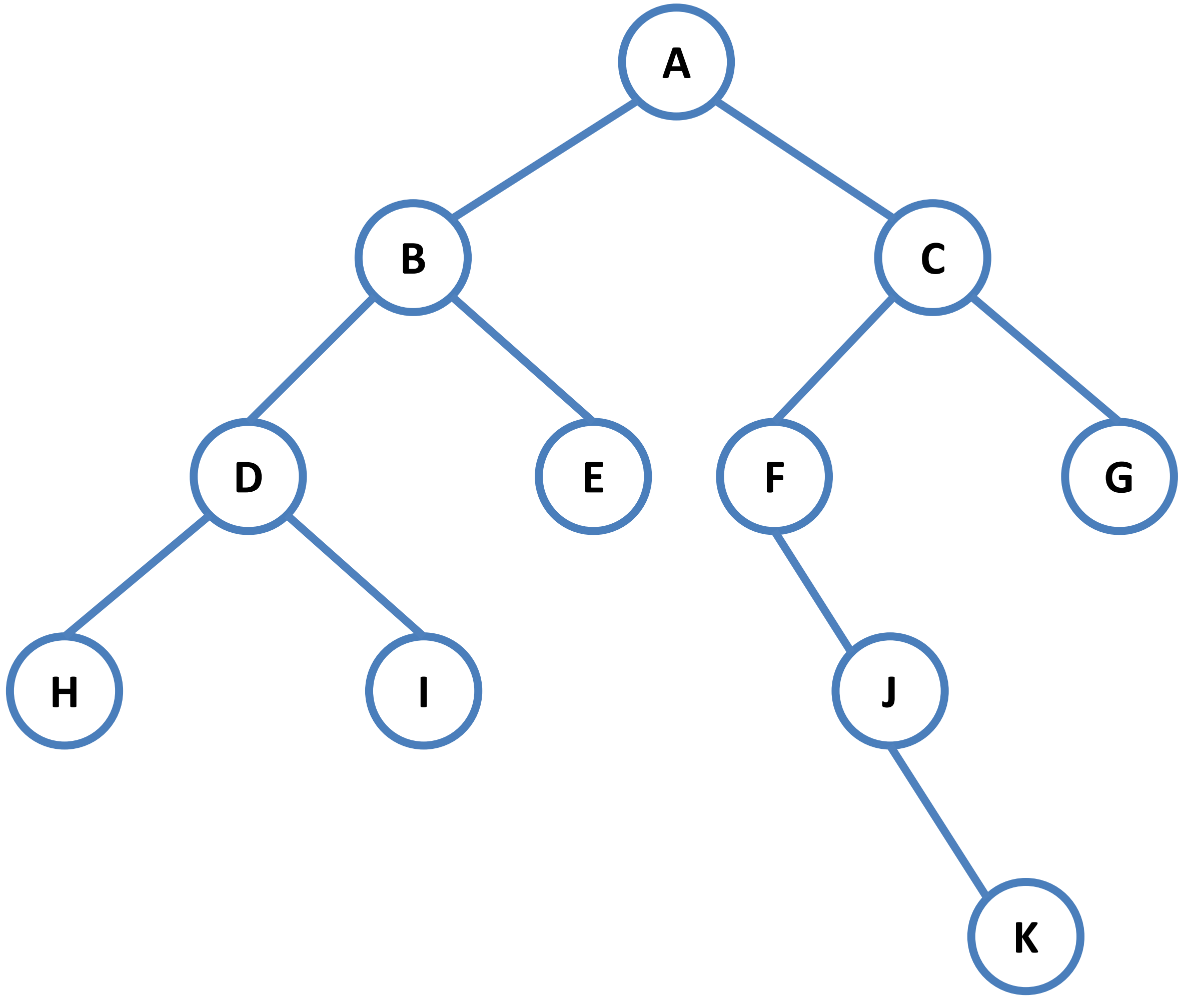
- Start at A, DFS



- Start at A, DFS
- A

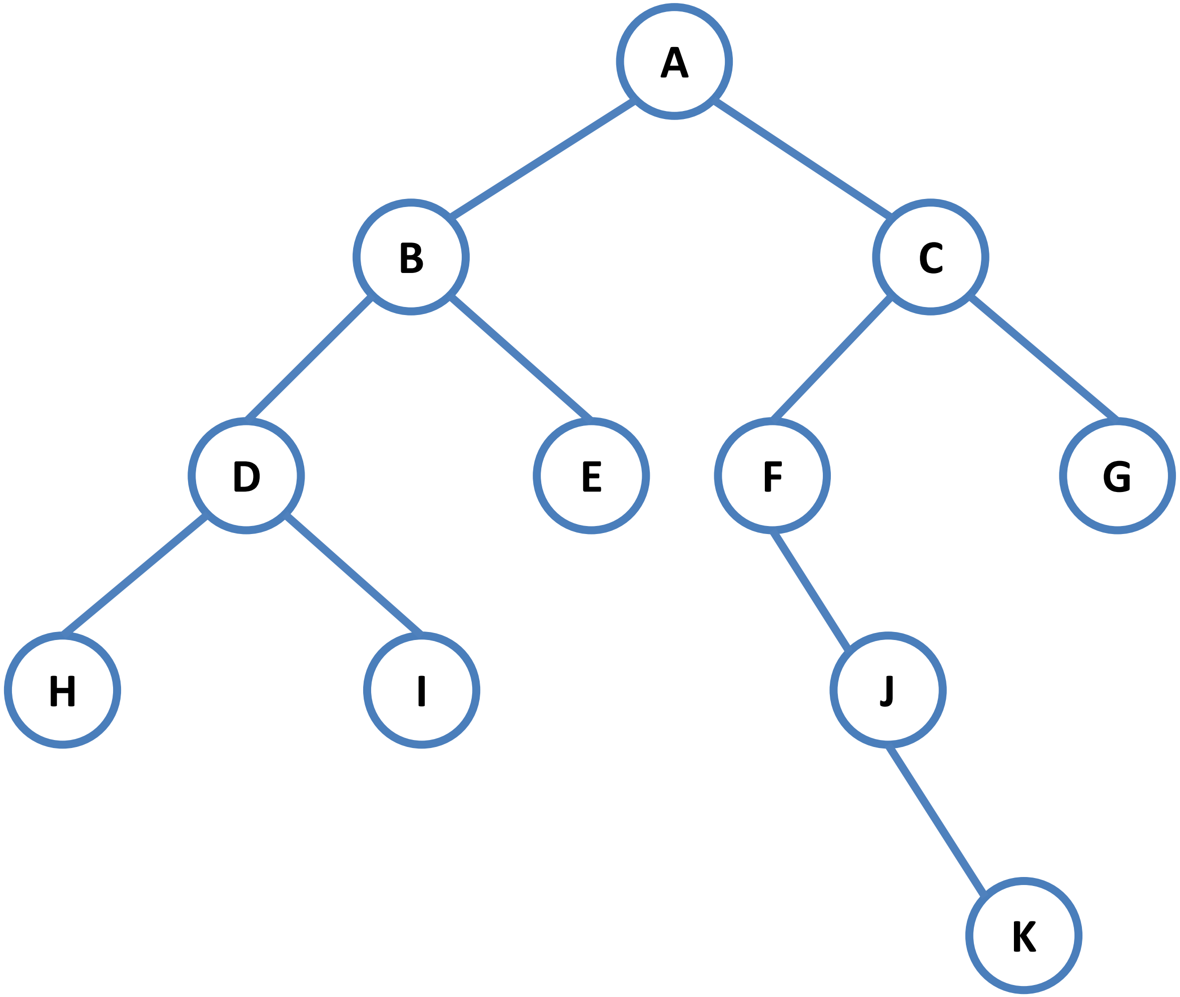


- Start at A, DFS
- A
- B

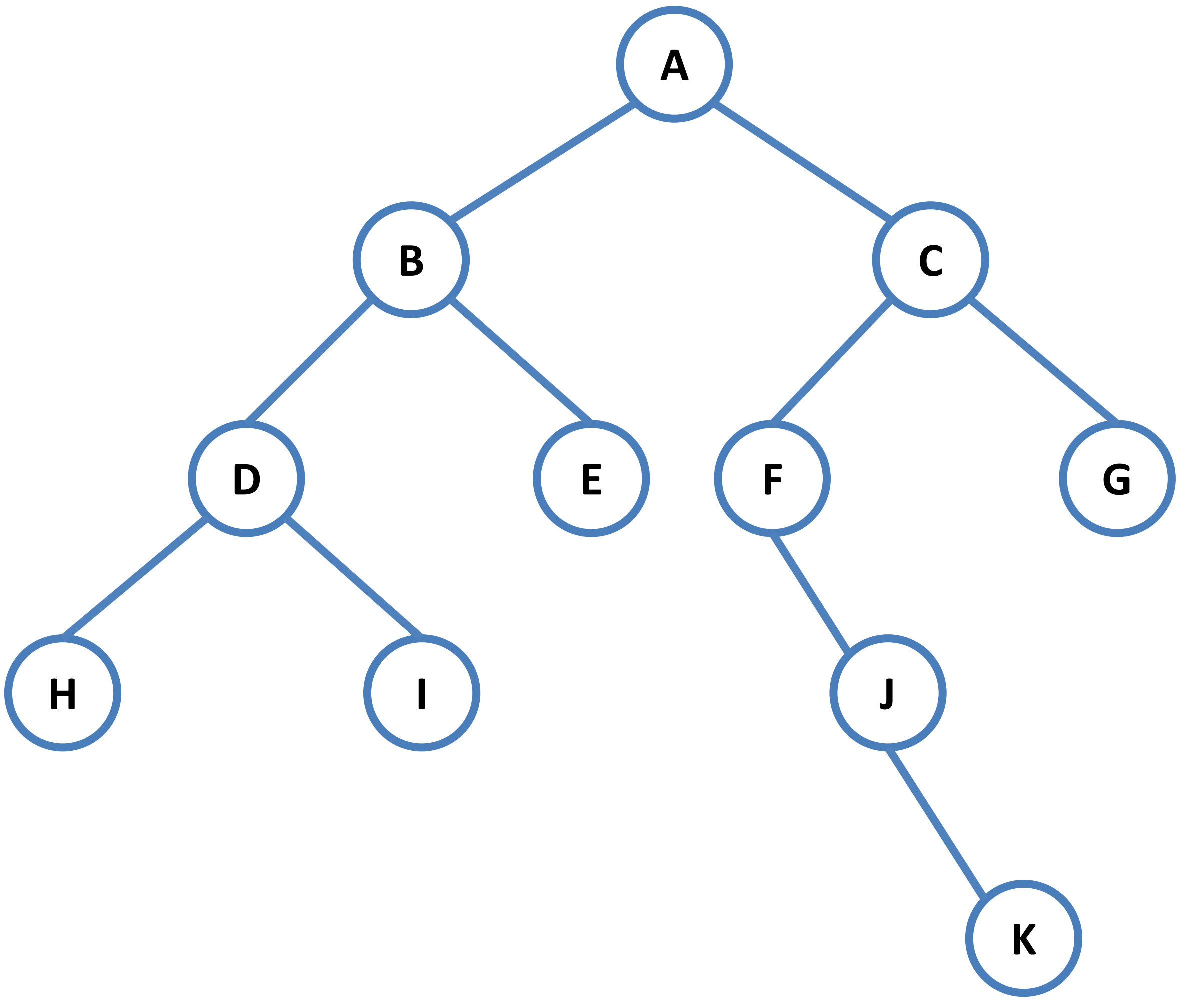


Graph
Traversal

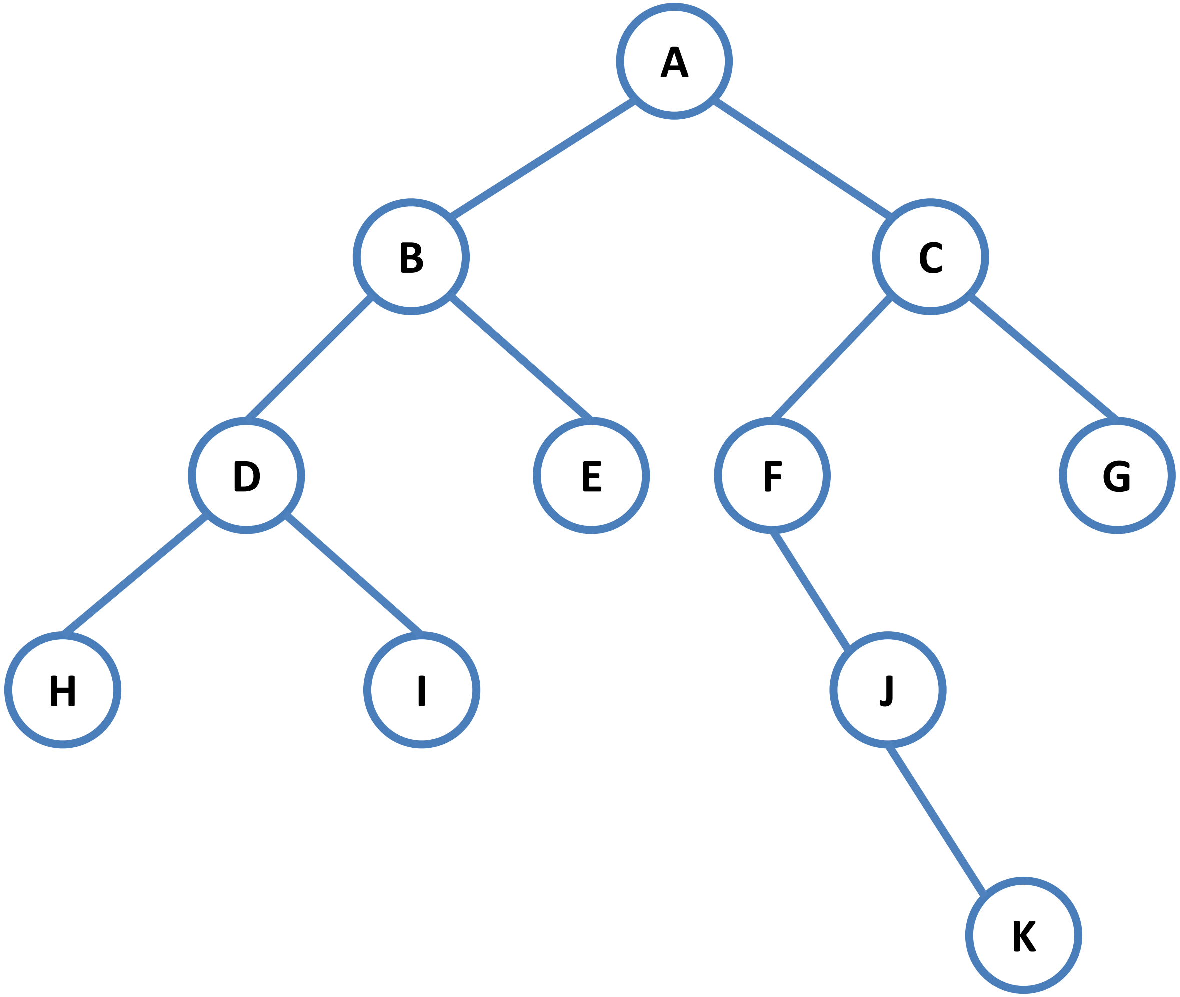
- Start at A, DFS
- A
- B
- D, go deep!



- Start at A, DFS
- A
- B
- D
- H

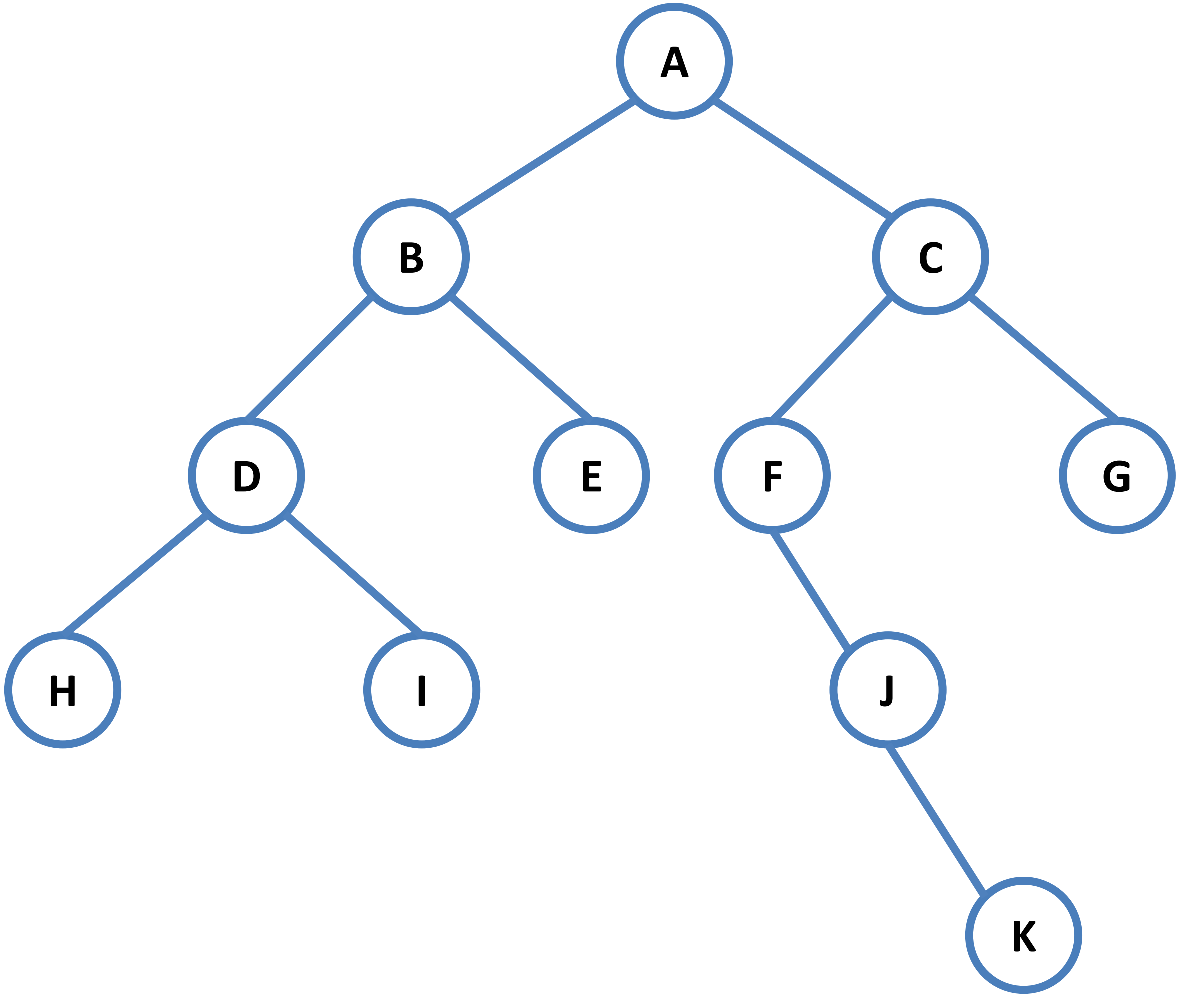


- Start at A, DFS
- A
- B
- D
- H
- can't go deeper

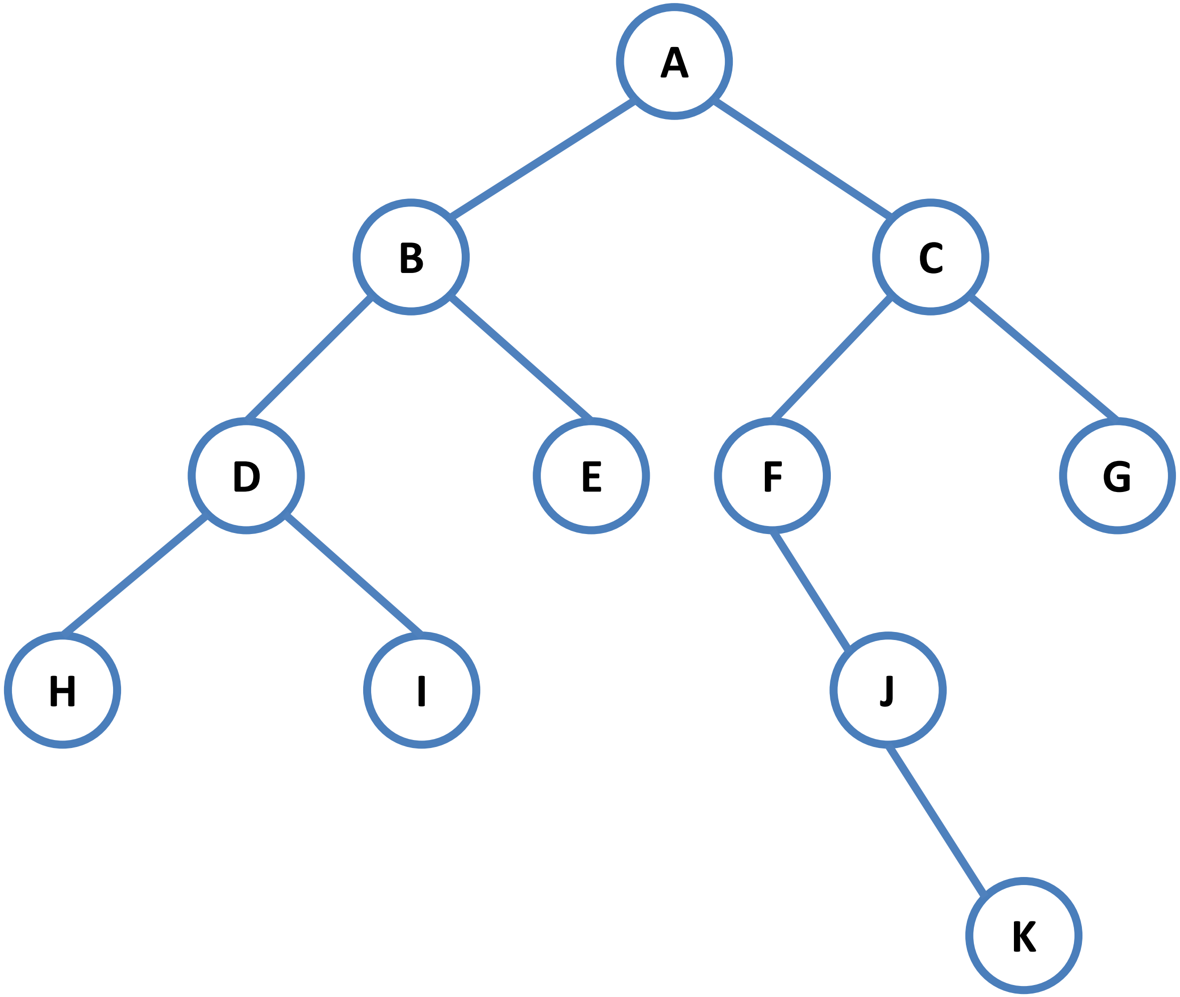


Graph
Traversal

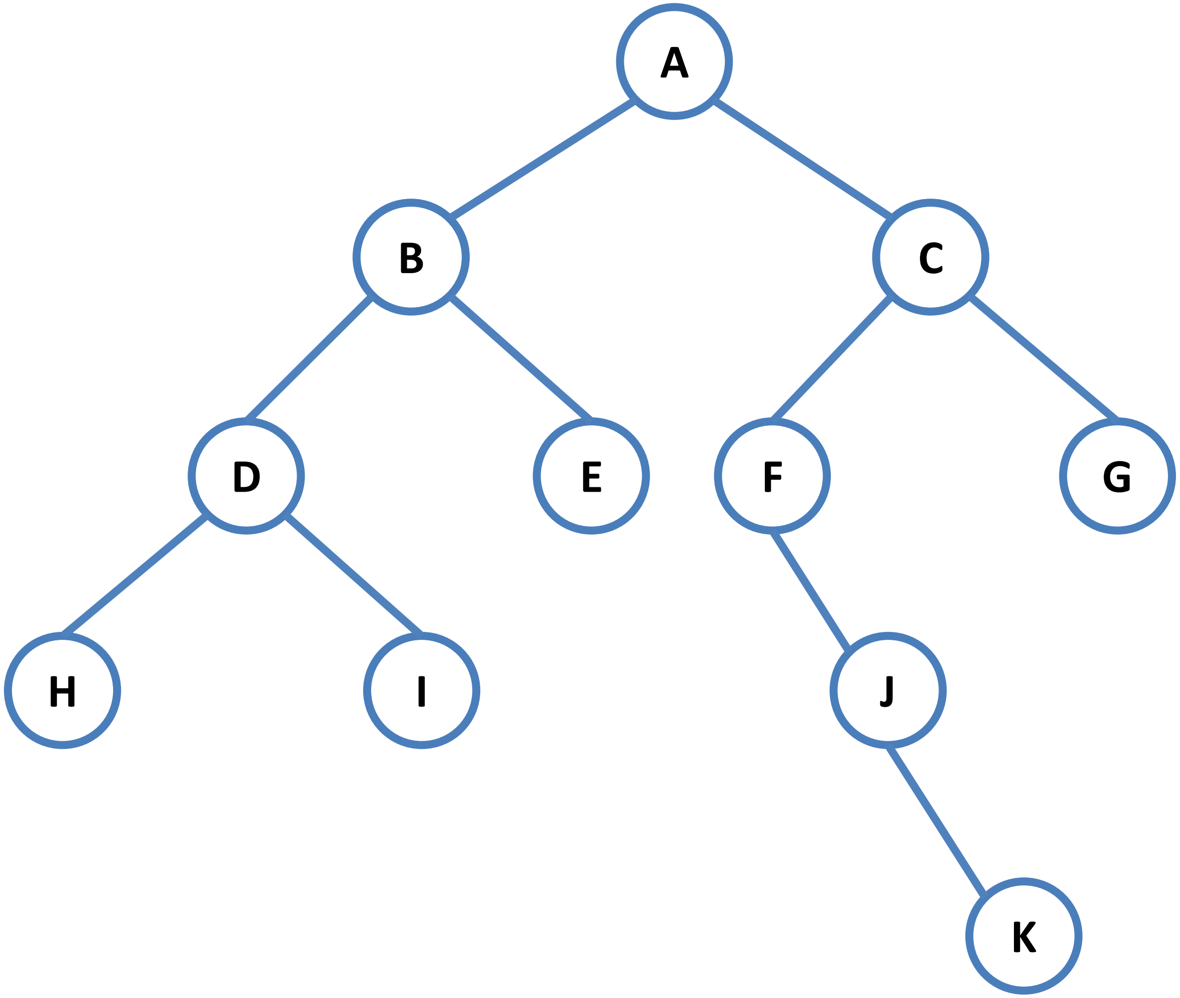
- Start at A, DFS
- A
- B
- D
- H
- I



- Start at A, DFS
- A
- B
- D
- H
- I
- E

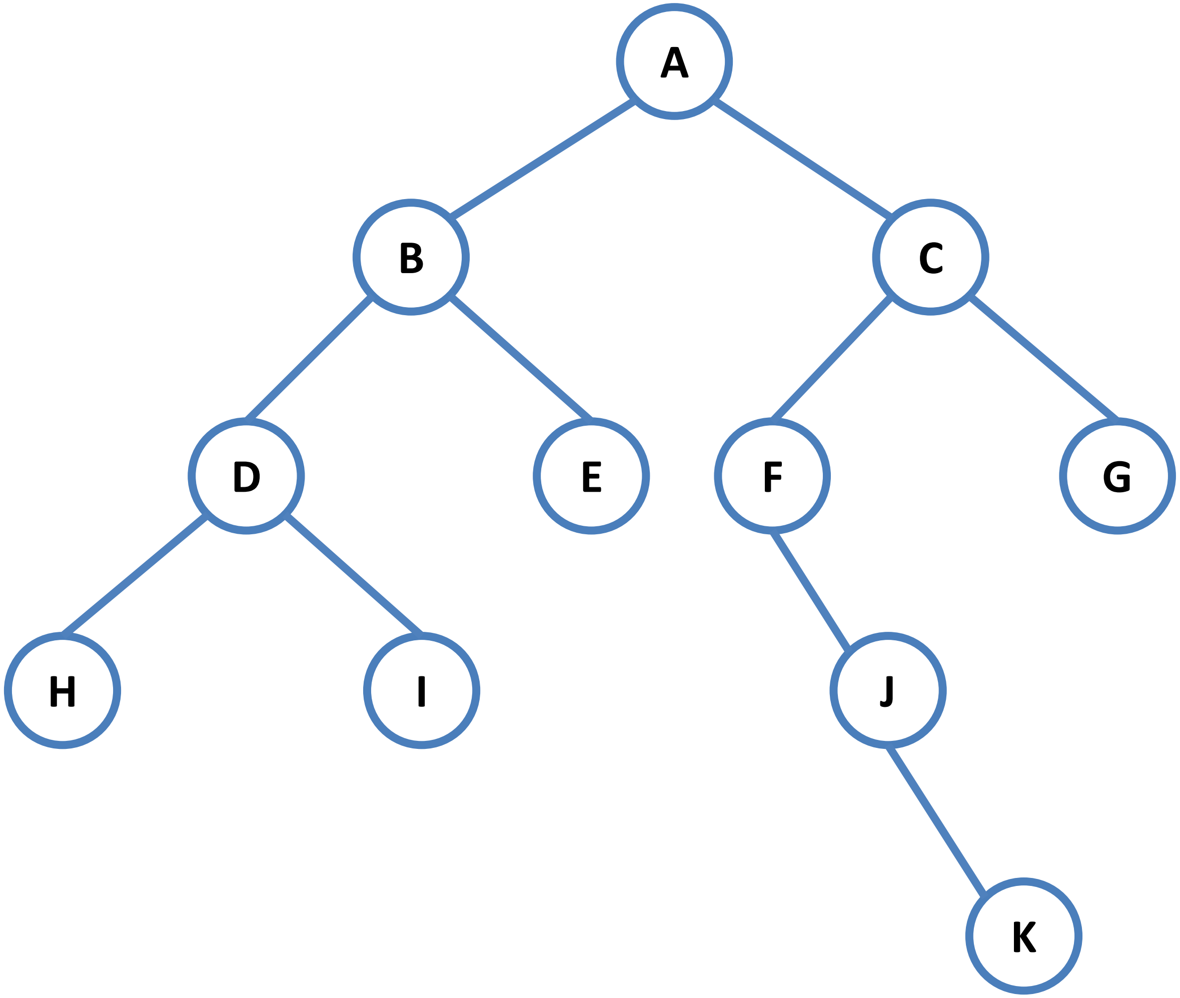


- Start at A, DFS
- A
- B
- D
- H
- I
- E
- C

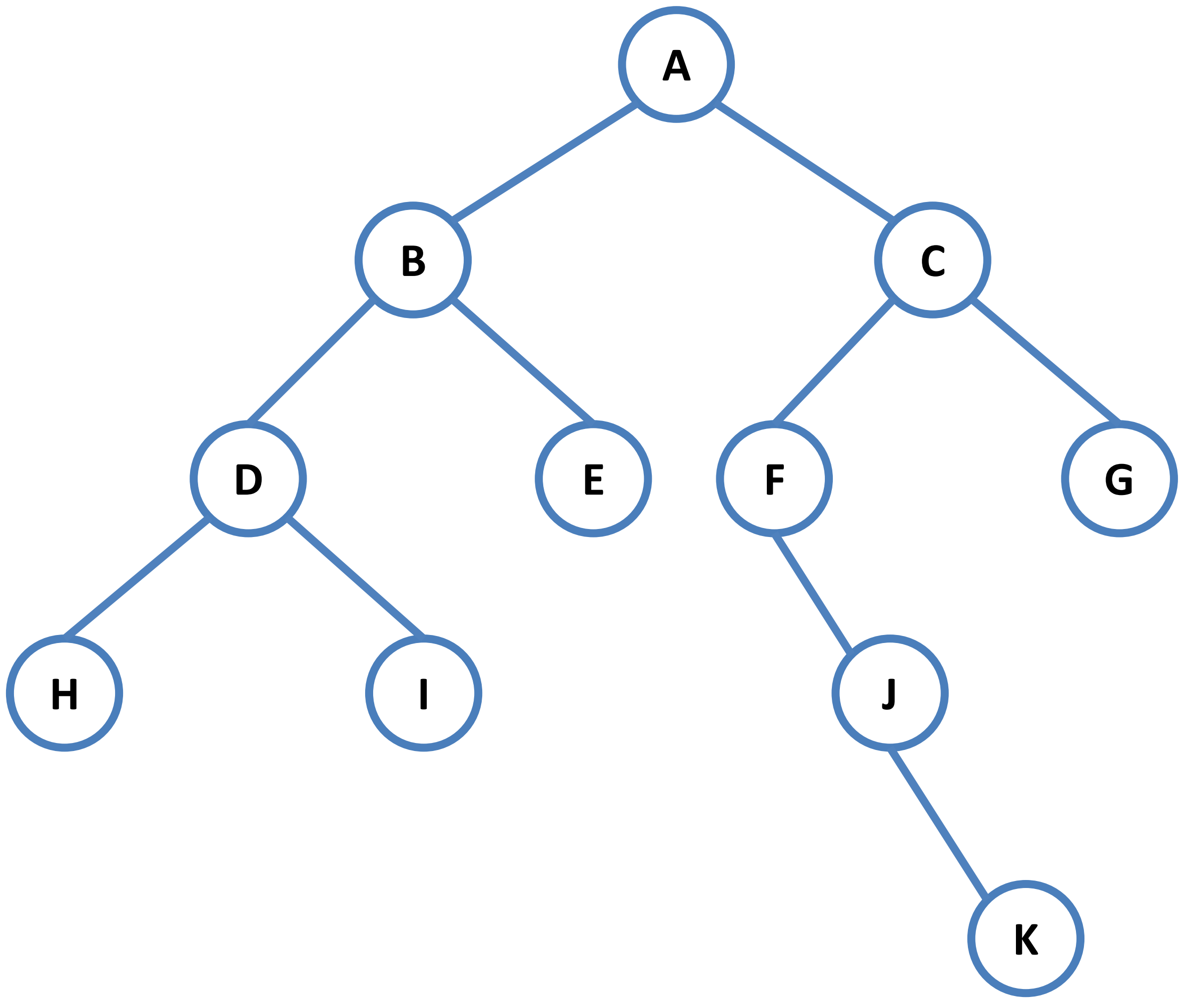


Graph
Traversal

- Start at A, DFS
- A
- B
- D
- H
- I
- E
- C
- F

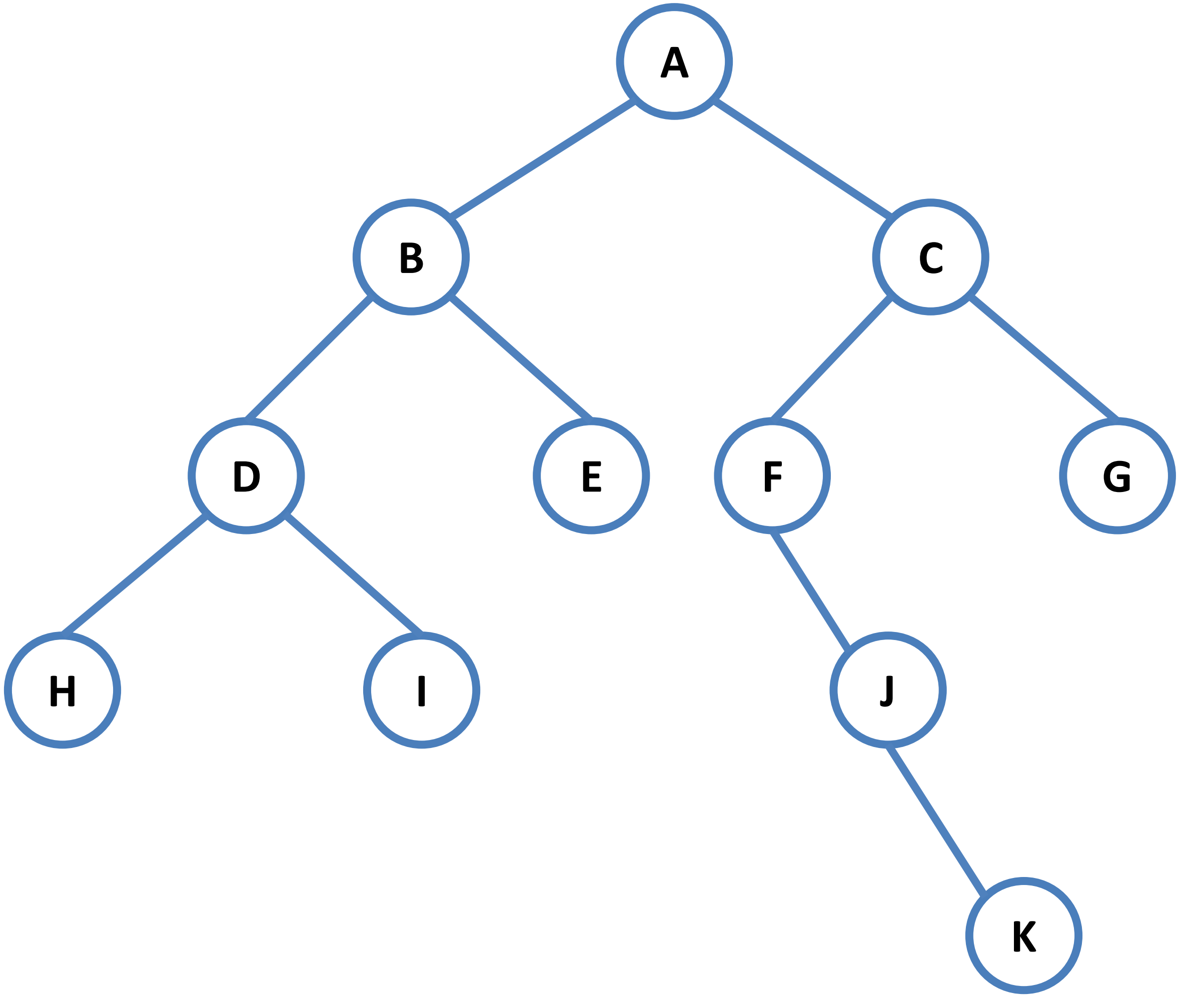


- Start at A, DFS
- A
- B
- D
- H
- I
- E
- C
- F
- J



Graph
Traversal

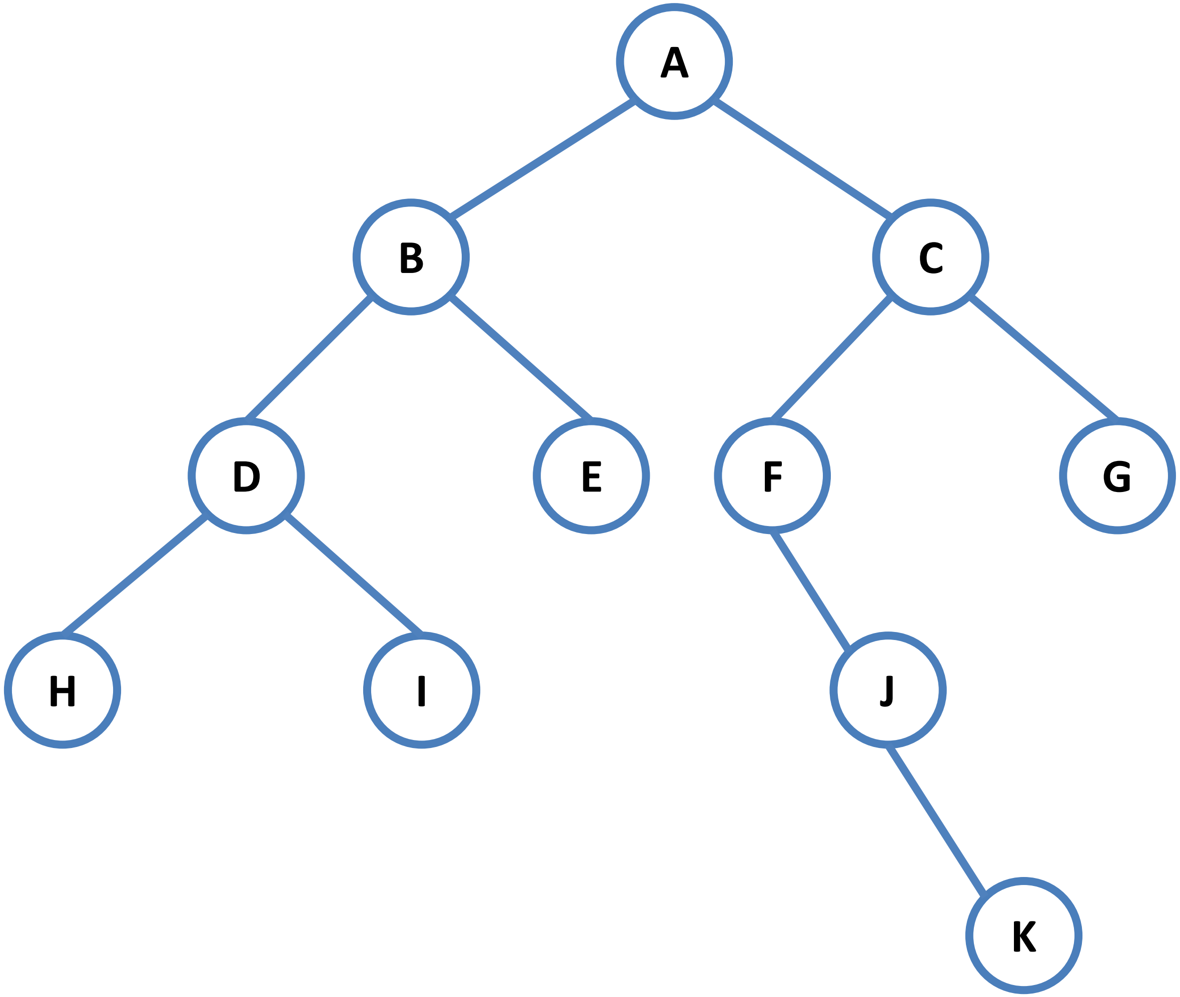
- Start at A, DFS
- A
- B
- D
- H
- I
- E
- C
- F
- J
- K



Graph

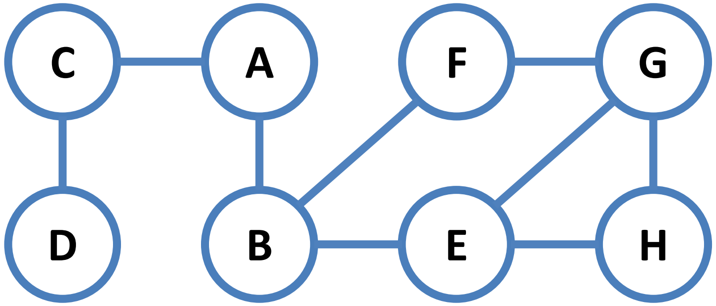
Traversal

- Start at A, DFS
- B
- D
- H
- I
- E
- C
- F
- J
- K
- G, finally

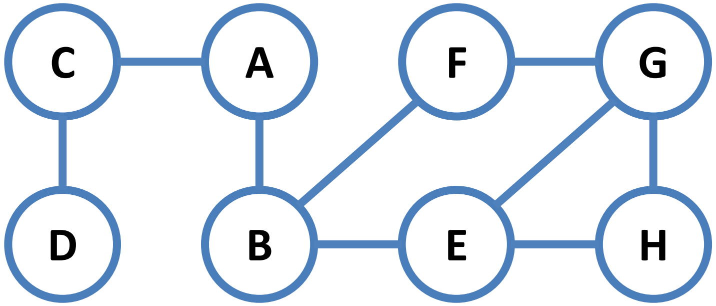


Questions?

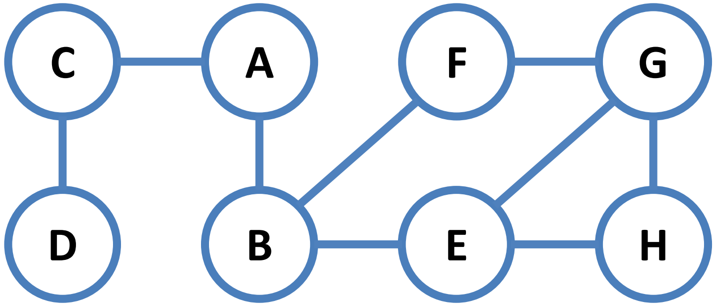
- How would you implement it?



- How would you implement it?
  - Let say we begin from vertex A

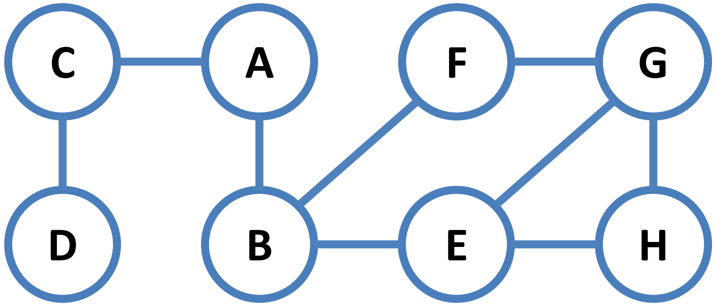


- How would you implement it?
  - Let say we begin from vertex A
  - What is our traversal?



Graph
BFS Implementation

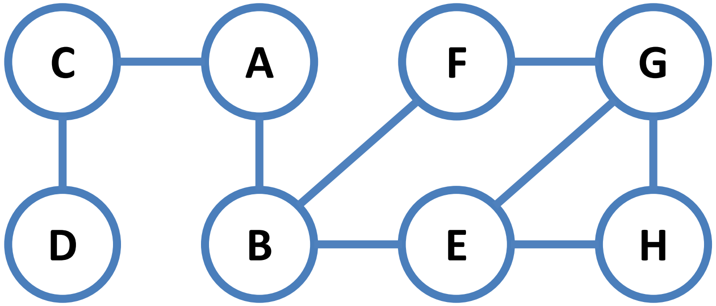
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered



Discovered										
Visited										

Graph
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Start with A

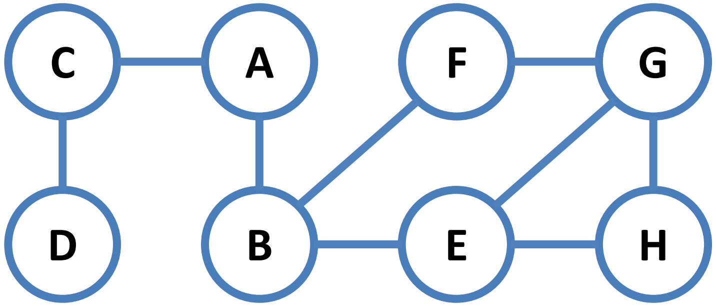


Discovered										
Visited										

Graph

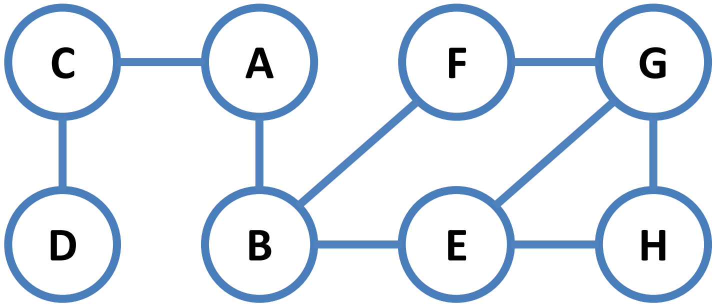
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it



Discovered	A									
Visited										

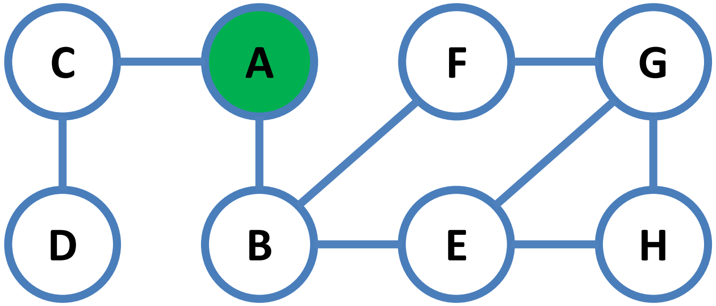
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty



Discovered	A									
Visited										

Graph
BFS Implementation

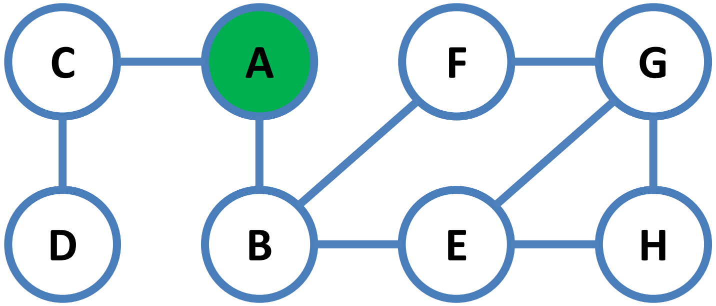
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered



Discovered	A									
Visited										

Graph
BFS Implementation

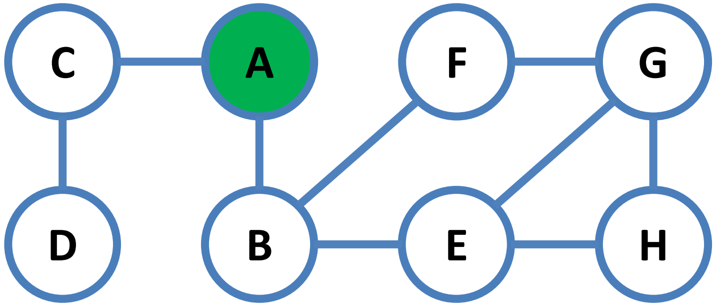
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited



Discovered										
Visited	A									

Graph
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served



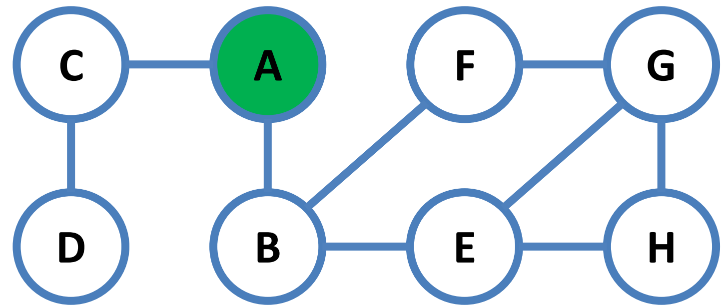
Discovered										
Visited	A									

Graph

BFS Implementation

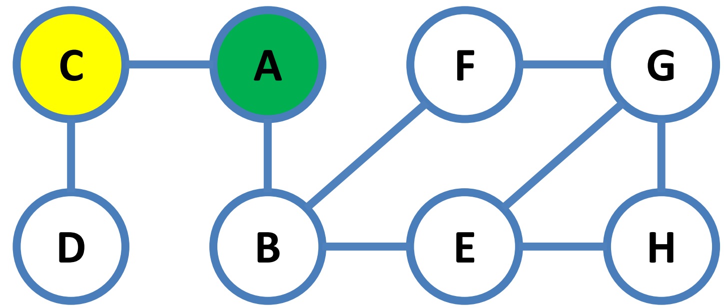
MONASH University

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A									

Graph
BFS Implementation



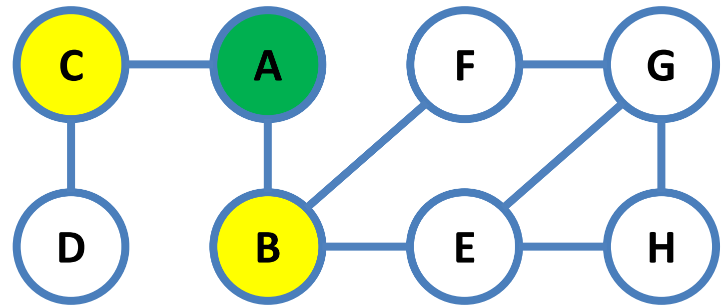
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	C									
Visited	A									

Graph

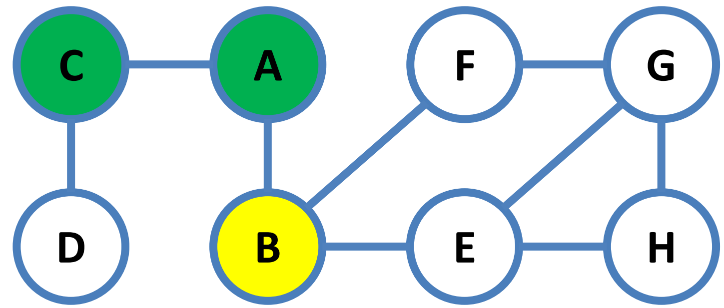
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



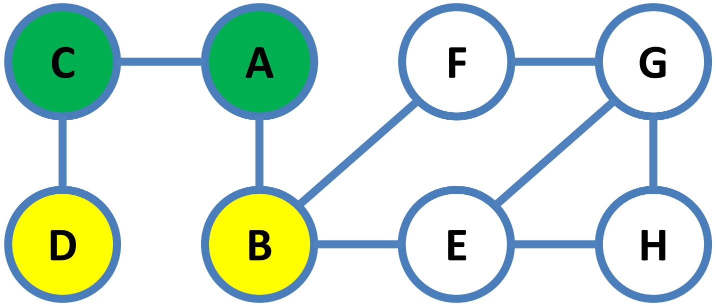
Discovered	C	B								
Visited	A									

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered		B								
Visited	A	C								

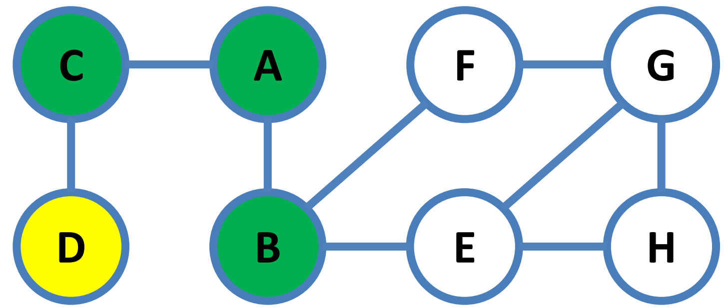
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered		B	D							
Visited	A	C								

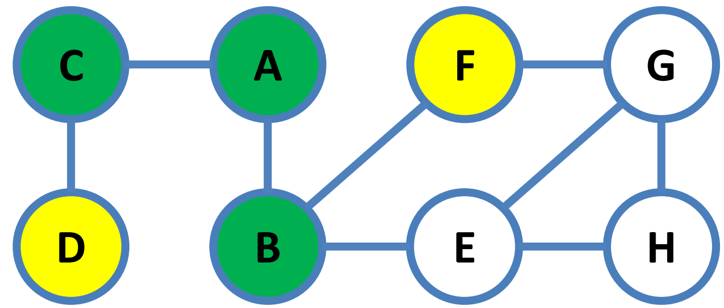
Graph
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



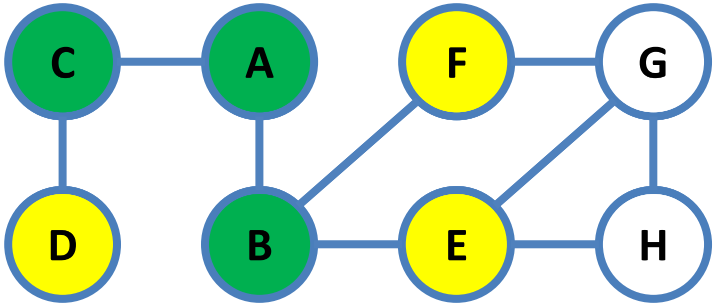
Discovered			D							
Visited	A	C	B							

Graph
BFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

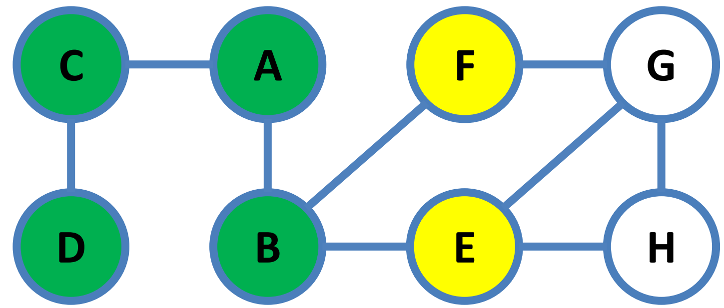
Discovered			**D**	**F**						
Visited	A	C	B							



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered			**D**	**F**	**E**					
Visited	A	C	B							

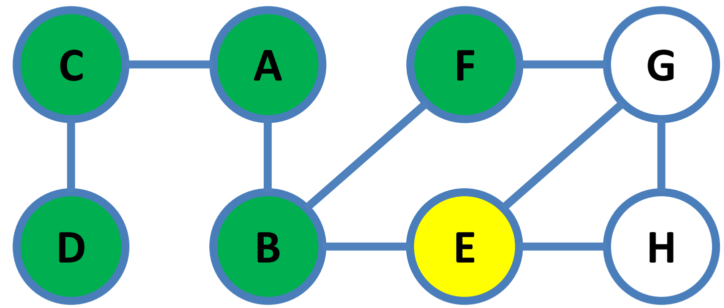
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered				**F**	**E**					
Visited	A	C	B	D						

Graph
BFS Implementation

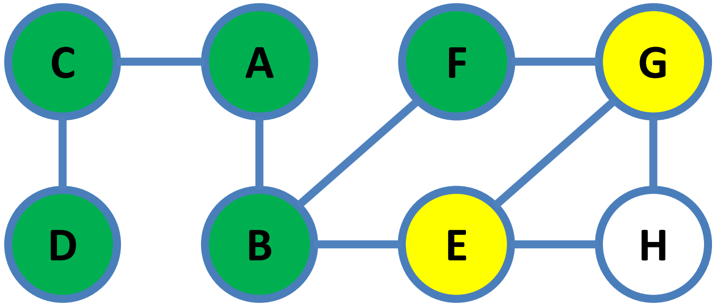
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered					**E**					
Visited	A	C	B	D	F					

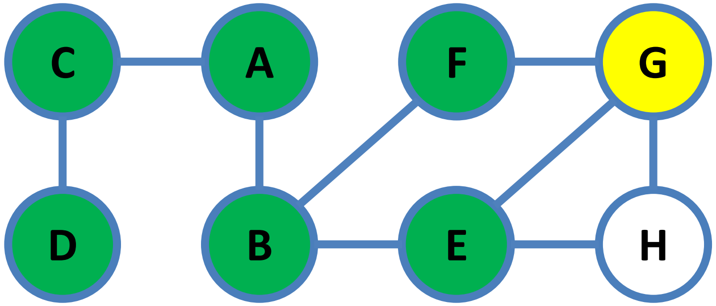
Graph
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered					E	G				
Visited	A	C	B	D	F					

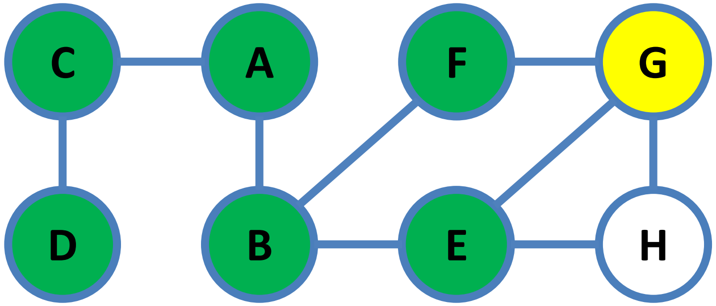
Graph
BFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered						G				
Visited	A	C	B	D	F	E				

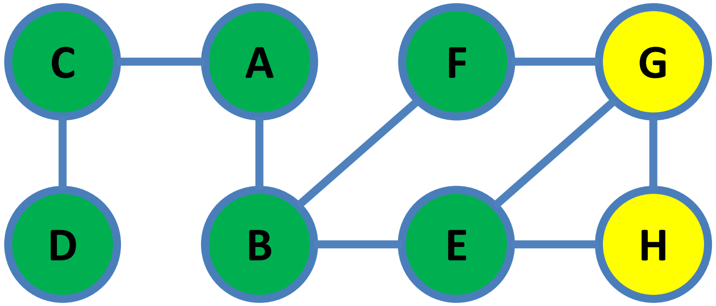
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered						G	G?			
Visited	A	C	B	D	F	E				

Graph
BFS Implementation

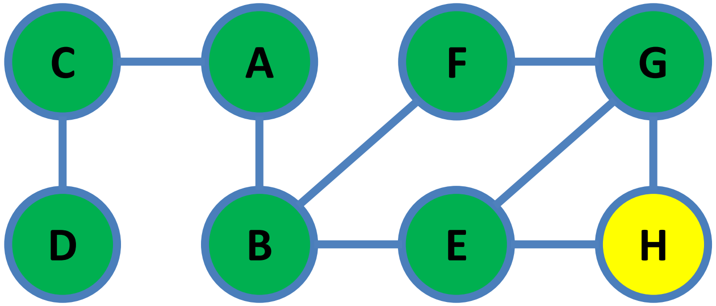
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered						G	H			
Visited	A	C	B	D	F	E				

Graph
BFS Implementation

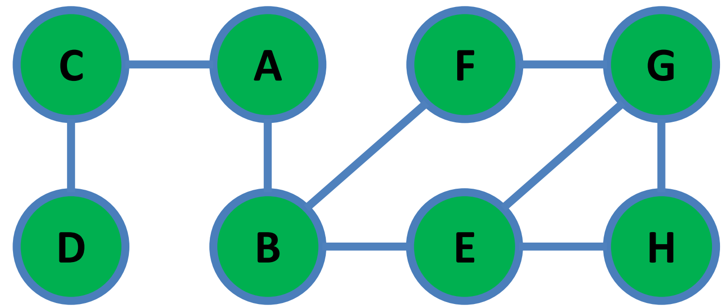
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered							**H**			
Visited	A	C	B	D	F	E	G			

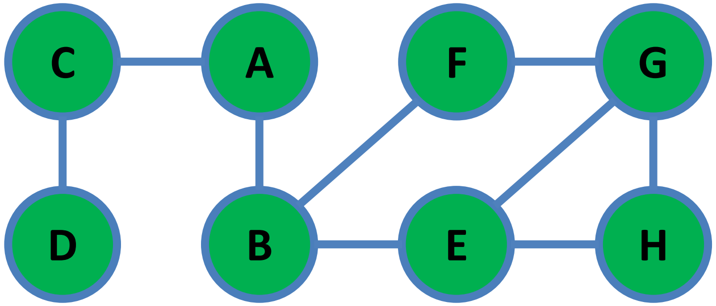
Graph
BFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A	C	B	D	F	E	G	H		

Graph
BFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
- The traversal answer is not unique

Discovered										
Visited	A	C	B	D	F	E	G	H		

- Complexity?

- Complexity?
  - Time is O(V+E)

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v

        directed only visit once

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)
    - V maximum for the discovered queue

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)
    - V maximum for the discovered queue
    - E to stored all of the edges (adjacency list)

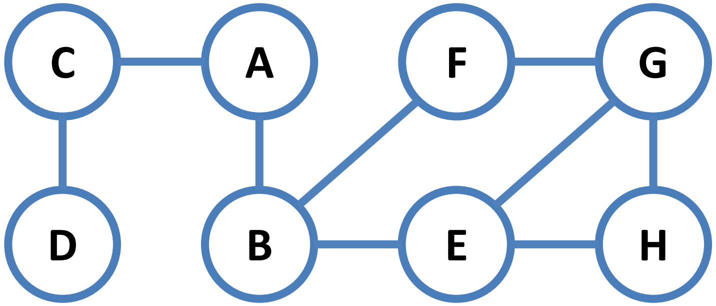
- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)
    - V maximum for the discovered queue
    - E to stored all of the edges (adjacency list)
  - But don’t we need to check the discovered queue for each vertex v?
    - O(V) search through the queue?

- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)
    - V maximum for the discovered queue
    - E to stored all of the edges (adjacency list)
  - But don’t we need to check the discovered queue for each vertex v?
    - O(V) search through the queue?
    - NO! Implement a Node class with self.discovered = True/ False

      in individual vertex

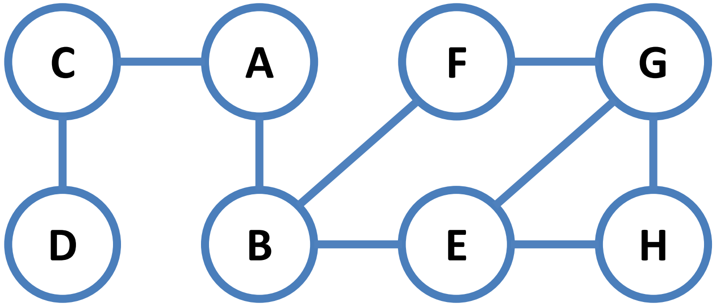
Questions?

- How would you implement it?
  - Let say we begin from vertex A
  - What is our DFS traversal?



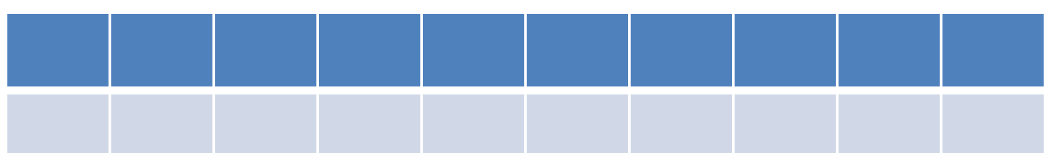
Graph
DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered



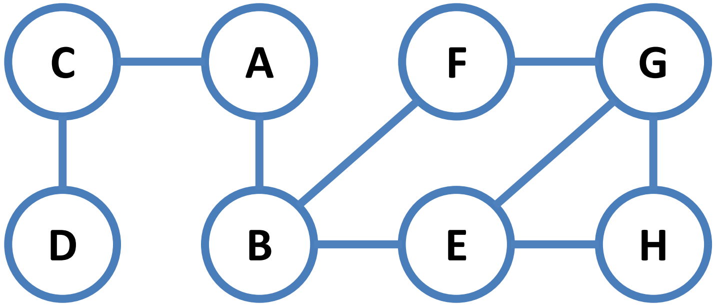
Discovered

Visited



DFS Implementation

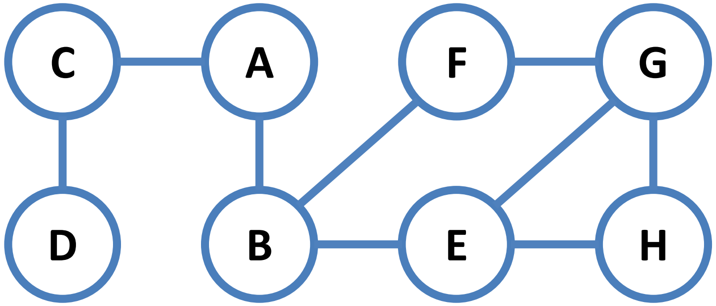
- How would you implement it?
  - Let say we begin from vertex A
  - Have a ~~queue~~ stack for discovered



Discovered										
Visited										

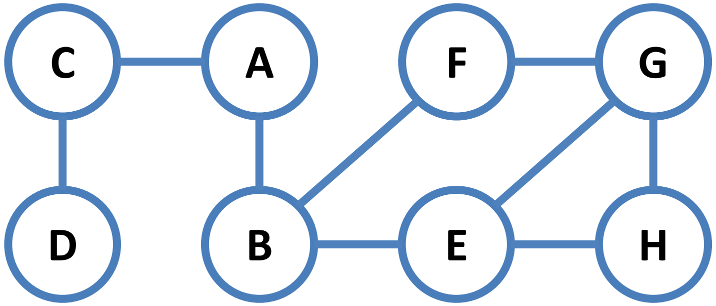
DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it



Discovered	A									
Visited										

Graph
DFS Implementation

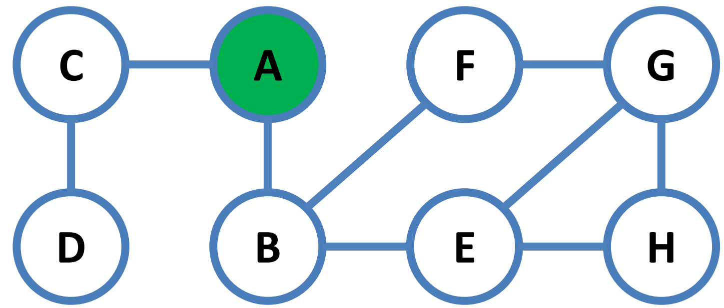


- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	A									
Visited										

Graph
DFS Implementation

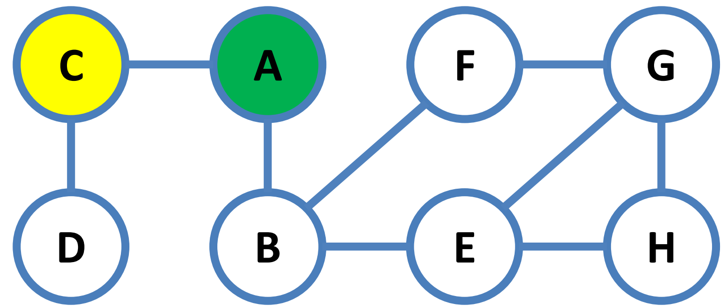
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A									

Graph
DFS Implementation

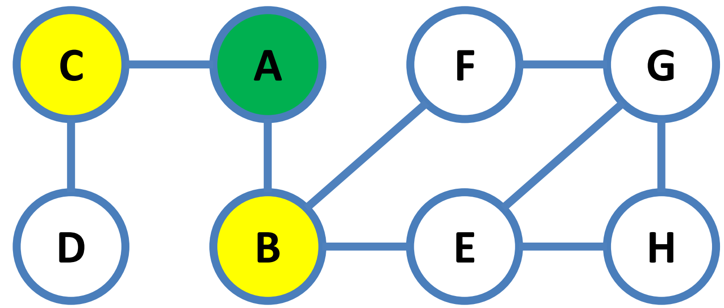
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	C									
Visited	A									

Graph
DFS Implementation

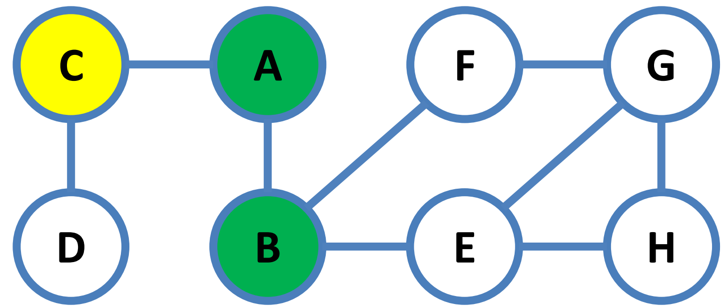
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	C	B								
Visited	A									

Graph
DFS Implementation

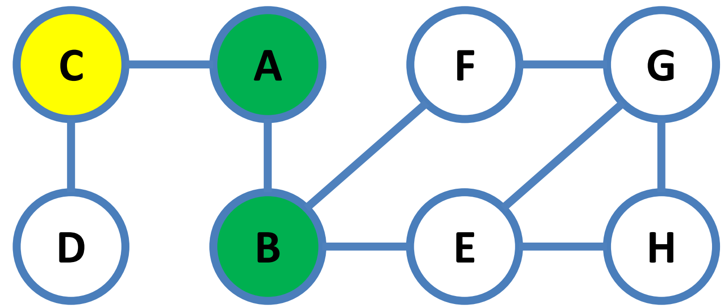
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	C									
Visited	A	B								

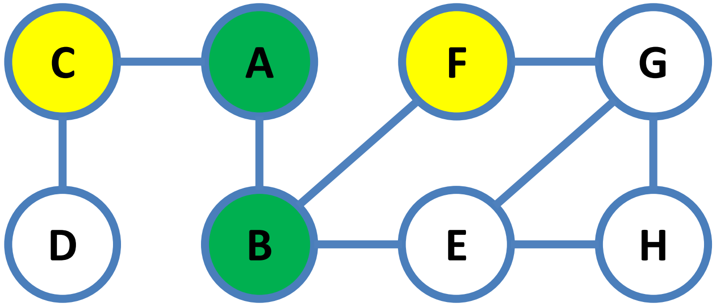
Graph
DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	**C**	**A?**								
Visited	A	B								

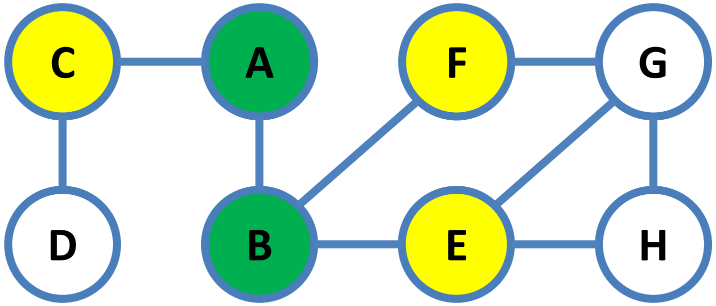
DFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	C	F								
Visited	A	B								

Graph
DFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

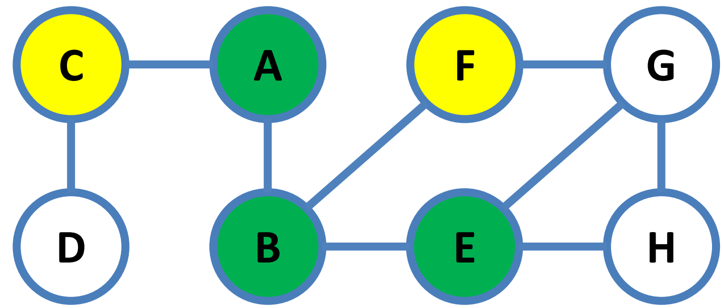
DFS pop B and push to visited (last in first out)
BFS serve C and append to visited (first in first out)

Discovered	C	F	E							
Visited	A	B								

Graph

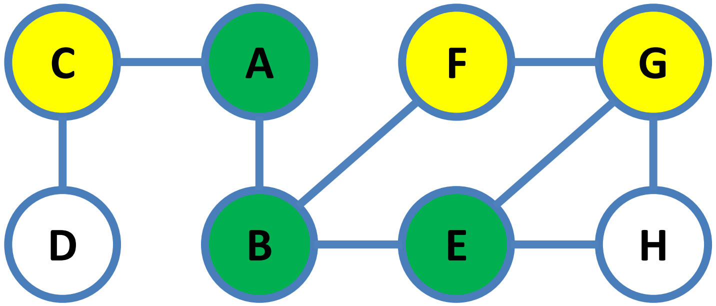
DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



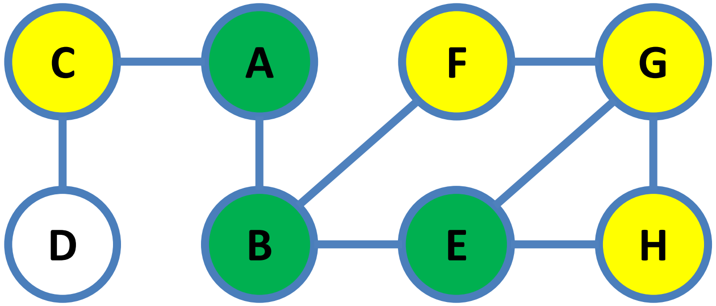
Discovered	**C**	**F**								
Visited	A	B	E							

Graph
DFS Implementation



- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

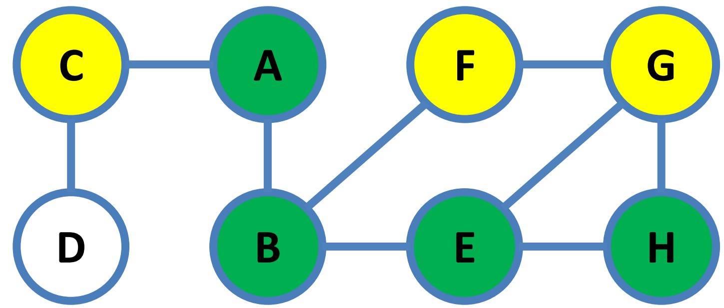
Discovered	**C**	**F**	**G**							
Visited	A	B	E							



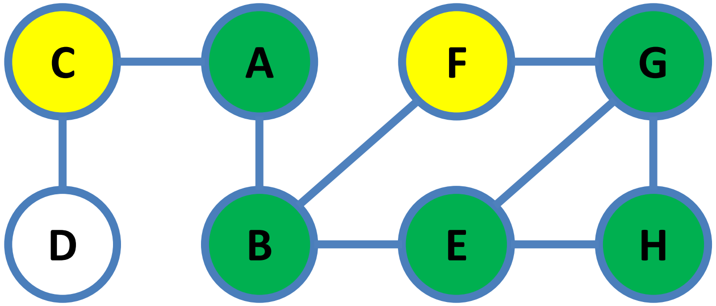
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	C	F	G	H						
Visited	A	B	E							

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	C	F	G							
Visited	A	B	E	H						

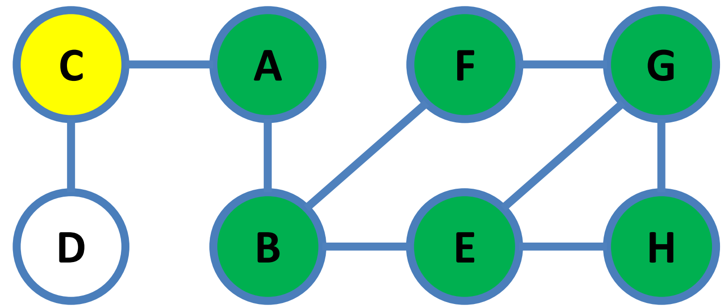


- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	**C**	**F**								
Visited	A	B	E	H	G					

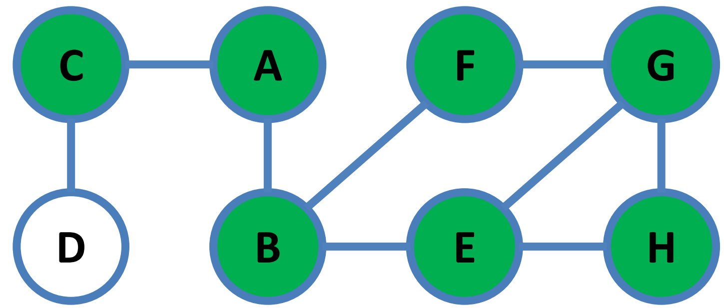
Graph
DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	C									
Visited	A	B	E	H	G	F				

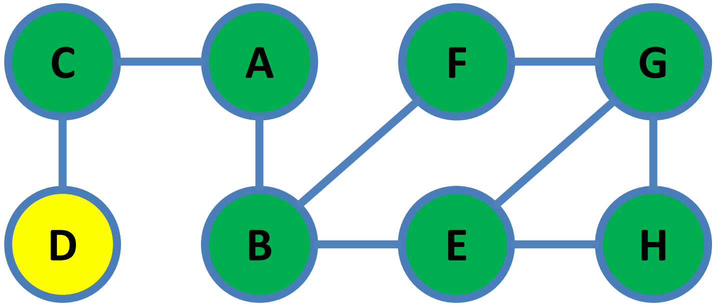
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A	B	E	H	G	F	C			

Graph
DFS Implementation

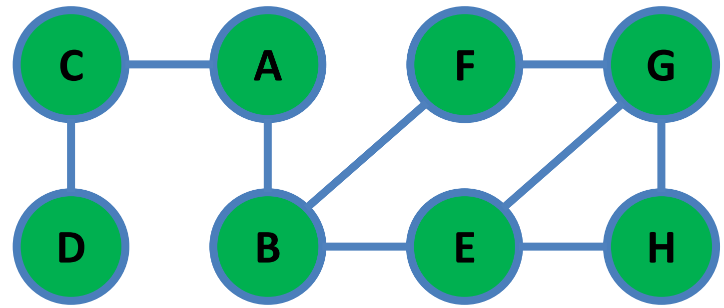
- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered	D									
Visited	A	B	E	H	G	F	C			

DFS Implementation

- How would you implement it?
  - Let say we begin from vertex A
  - Have a stack for discovered
    - Push source (A) into it
  - While discovered is not empty
    - Pop from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A	B	E	H	G	F	C	D		

- Complexity?

DFS Implementation

- Complexity?
  - Time is O(V+E)
    - Explanation same as BFS

- Complexity?
  - Time is O(V+E)
    - Explanation same as BFS
  - Space is O(V+E)
    - Explanation same as DFS

DFS Implementation

- Can you think of another way to implement DFS?

DFS Implementation

- Can you think of another way to implement DFS?
- Recursion!
  - Best way to implement any form of traversal
    - Just like when you implemented tree/ trie traversal

- Can you think of another way to implement DFS?
- Recursion!
  - Best way to implement any form of traversal
    - Just like when you implemented tree/ trie traversal

- Let us just write them all out as a live coding session!

Questions?

- Can you think of another way to implement DFS?
- Recursion!
  - Best way to implement any form of traversal
    - Just like when you implemented tree/ trie traversal

      Recursion stack is the discovered stack in DFS

```
def dfs(current_vertex):
    current_vertex.visited = True
    for next_vertex in current_vertex.adjacent:
        if next_vertex.visited == False:
            dfs(next_vertex)

source_vertex = A
dfs(source_vertex)
```

DFS Implementation

- Can you think of another way to implement DFS?
- Recursion!
  - Best way to implement any form of traversal
    - Just like when you implemented tree/ trie traversal
  - Make sense because we are going depth-first like how recursion does it!

```python
def dfs(current_vertex):
    current_vertex.visited = True
    for next_vertex in current_vertex.adjacent:
        if next_vertex.visited == False:
            dfs(next_vertex)

source_vertex = A
dfs(source_vertex)
```

Questions?

DFS and BFS application?

- As mentioned earlier, it is the basic algorithm for many more complex algorithm… the application include:

DFS and BFS application?

- As mentioned earlier, it is the basic algorithm for many more complex algorithm… the application include:
  - Reachability
  - Finding all connected components
  - Finding cycles
  - Shortest path (brute force)
  - Shortest path (non-brute force) on unweighted graph
  - Topological sort (later on)
  - … and many more!

DFS and BFS application?

- As mentioned earlier, it is the basic algorithm for many more complex algorithm… the application include:
  - Reachability
  - Finding all connected components
  - Finding cycles
  - Shortest path (brute force)
  - Shortest path (non-brute force) on unweighted graph
  - Topological sort (later on)
  - … and many more!

  - We will see more in unit notes and tutorials

Questions?

Break!

- Classical problem

- Classical problem
  - Given a set of locations
  - Given routes between locations
  - Given the distance between locations

- Classical problem
  - Given a set of locations
  - Given routes between locations
  - Given the distance between locations
  - Can you find the shortest distance from a source to a destination?

- Classical problem
  - Given a set of locations
  - Given routes between locations
  - Given the distance between locations
  - Can you find the shortest distance from a source to a destination?
  - What is the path?

- Classical problem
  - Given a set of locations as vertices V
  - Given routes between locations as edges E
  - Given the distance between locations as weights W
  - Can you find the shortest distance from a source to a destination?
  - What is the path?

- Classical problem
  - Given a set of locations as vertices V
  - Given routes between locations as edges E
  - Given the distance between locations as weights W
  - Can you find the shortest distance from a source to a destination?
  - What is the path? This would require backtracking

- Classical problem
  - Given a set of locations as vertices V
  - Given routes between locations as edges E
  - Given the distance between locations as weights W
  - Can you find the shortest distance from a source to a destination?
  - What is the path? This would require backtracking

- If the graph is unweighted?

- Classical problem
  - Given a set of locations as vertices V
  - Given routes between locations as edges E
  - Given the distance between locations as weights W
  - Can you find the shortest distance from a source to a destination?
  - What is the path? This would require backtracking
- If the graph is unweighted?
  - Use BFS from the source!

Graph

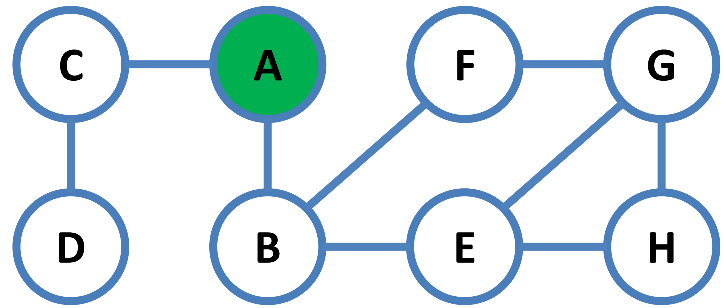
Shortest distance and path

- Classical problem
  - Given a set of locations as vertices V
  - Given routes between locations as edges E
  - Given the distance between locations as weights W
  - Can you find the shortest distance from a source to a destination?
  - What is the path? This would require backtracking

- If the graph is unweighted?
  - Use BFS from the source!
  - Look back our BFS example

    tree going down level by level
    so give shortest distance

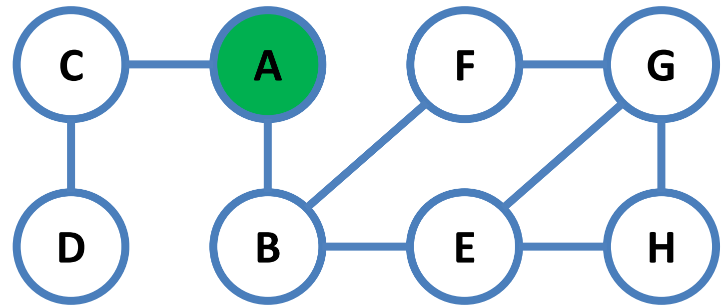
MONASH University



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue

Discovered	A,0									
Visited										

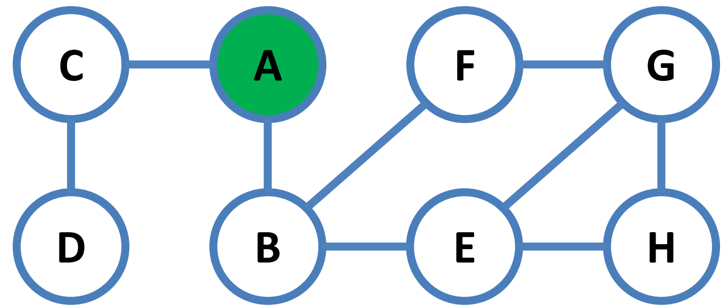
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue



Discovered										
Visited	A,0									

Graph

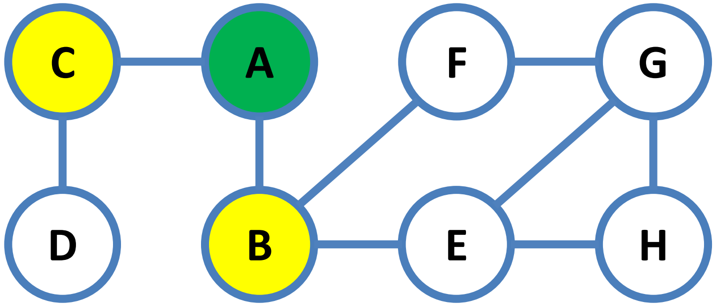
Shortest distance with BFS



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

Discovered										
Visited	A,0									

Shortest distance with BFS



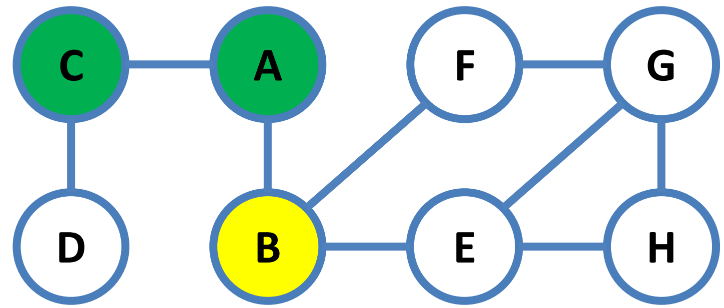
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

Discovered	**C,1**	**B,1**								
Visited	A,0									

Graph

Shortest distance with BFS

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



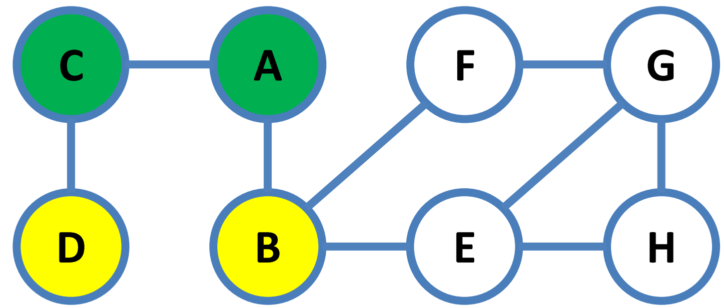
Discovered		**B,1**								
Visited	A,0	C,1								

Graph

Shortest distance with BFS

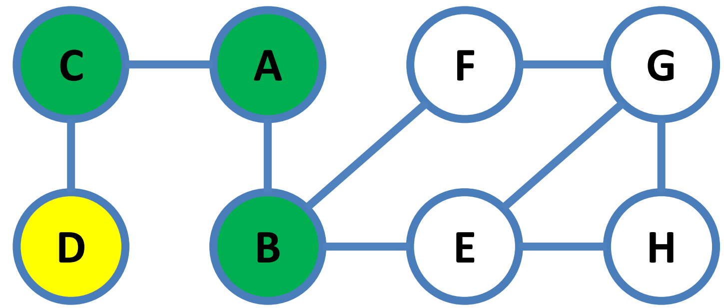
MONASH University

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



Discovered		**B,1**	**D,2**							
Visited	A,0	C,1								

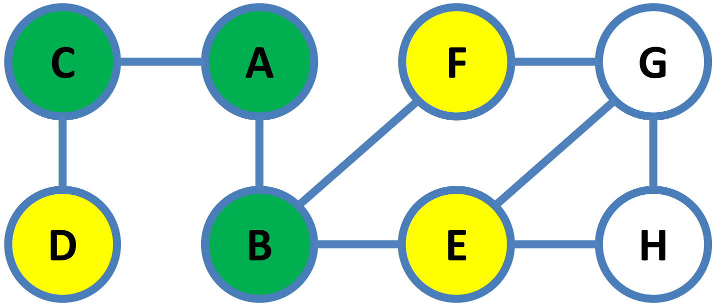
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



Discovered			**D,2**							
Visited	A,0	C,1	B,1							

Graph

Shortest distance with BFS



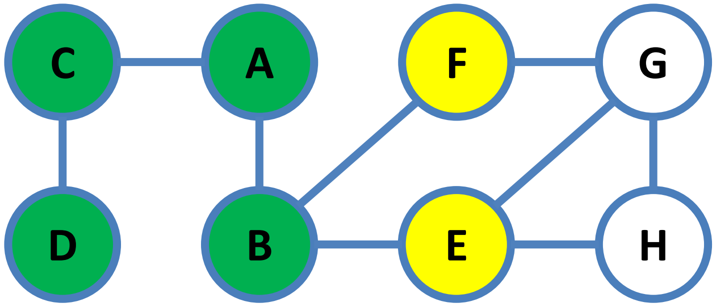
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

Discovered			**D,2**	**F,2**	**E,2**					
Visited	A,0	C,1	B,1							

Graph

Shortest distance with BFS

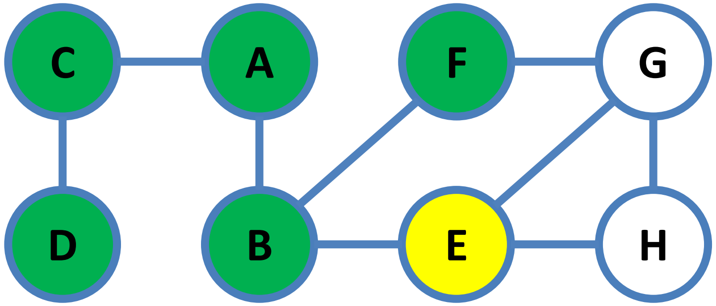
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



Discovered				**F,2**	**E,2**					
Visited	A,0	C,1	B,1	D,2						

Shortest distance with BFS

- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

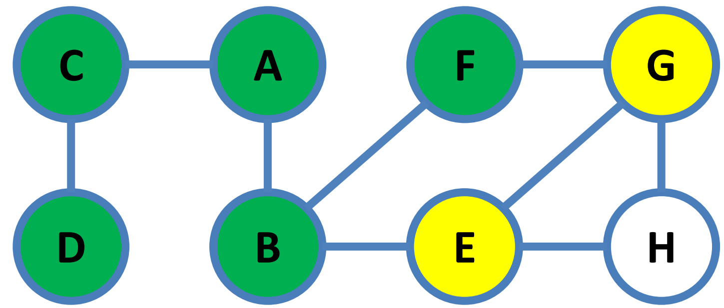


Discovered					E,2					
Visited	A,0	C,1	B,1	D,2	F,2					

Graph

Shortest distance with BFS

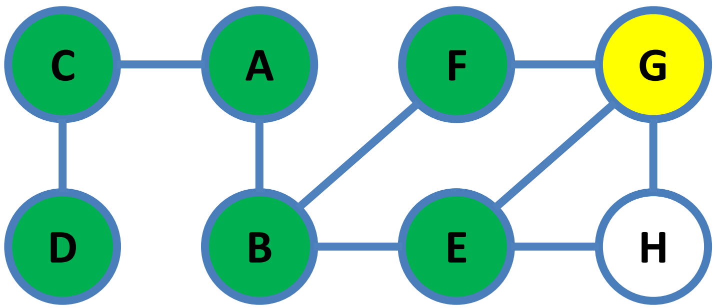
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



Discovered					**E,2**	**G,3**				
Visited	A,0	C,1	B,1	D,2	F,2					

Graph

Shortest distance with BFS

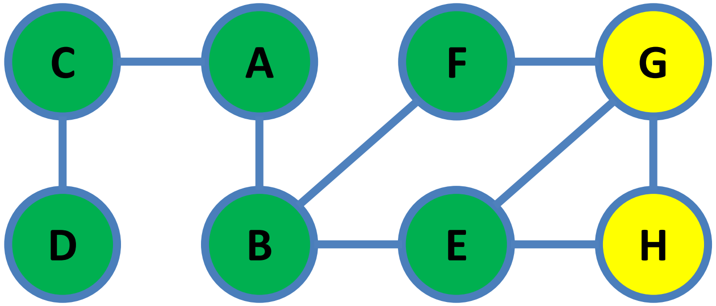


- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

Discovered						**G,3**				
Visited	A,0	C,1	B,1	D,2	F,2	E,2				

Graph
Shortest distance with BFS

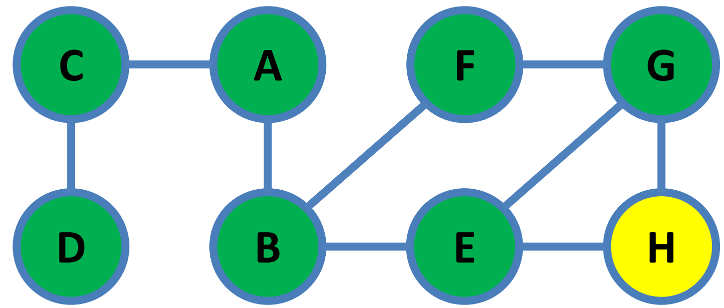
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



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

Graph
Shortest distance with BFS

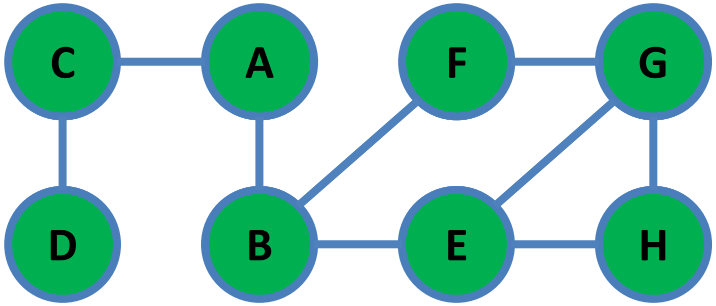
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1



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

Graph

Shortest distance with BFS



- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

        if v = E (v = target destination)
        break (to stop early)

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

Monash University

Questions?

MONASH University

- How would you modify the following to find the path?
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

- How would you modify the following to find the path?
- How would you implement it?
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1
      - v.previous = u # enable backtracking

Questions?

distance calculated from source

- What if the graph is weighted?

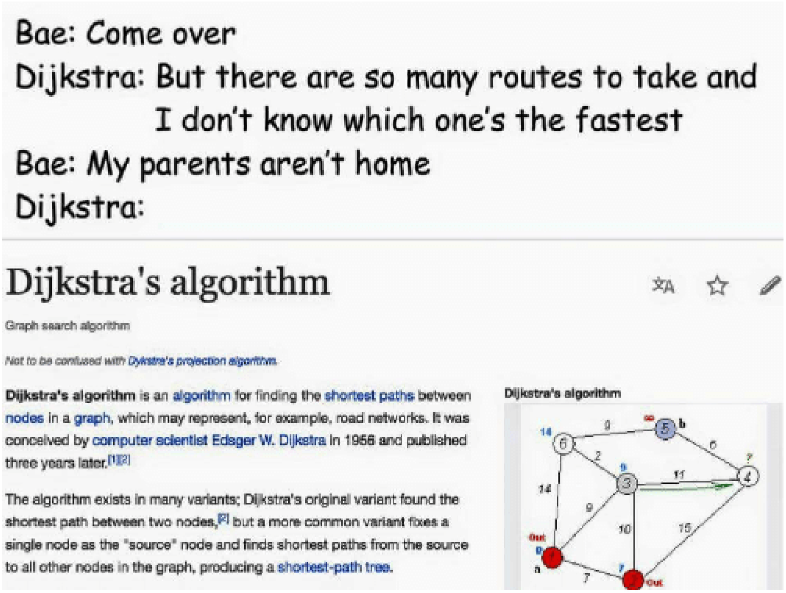
- What if the graph is weighted?
  – BFS is not able to do it anymore

- What if the graph is weighted?
  - BFS is not able to do it anymore since need Vertex object to store additional info self.weight
  - Sooo, we call in Dijkstra

- What if the graph is weighted?
  – BFS is not able to do it anymore
  – Sooo, we call in Dijkstra (the left one)

- What if the graph is weighted?
  - BFS is not able to do it anymore
  - Sooo, we call in Dijkstra (the left one)

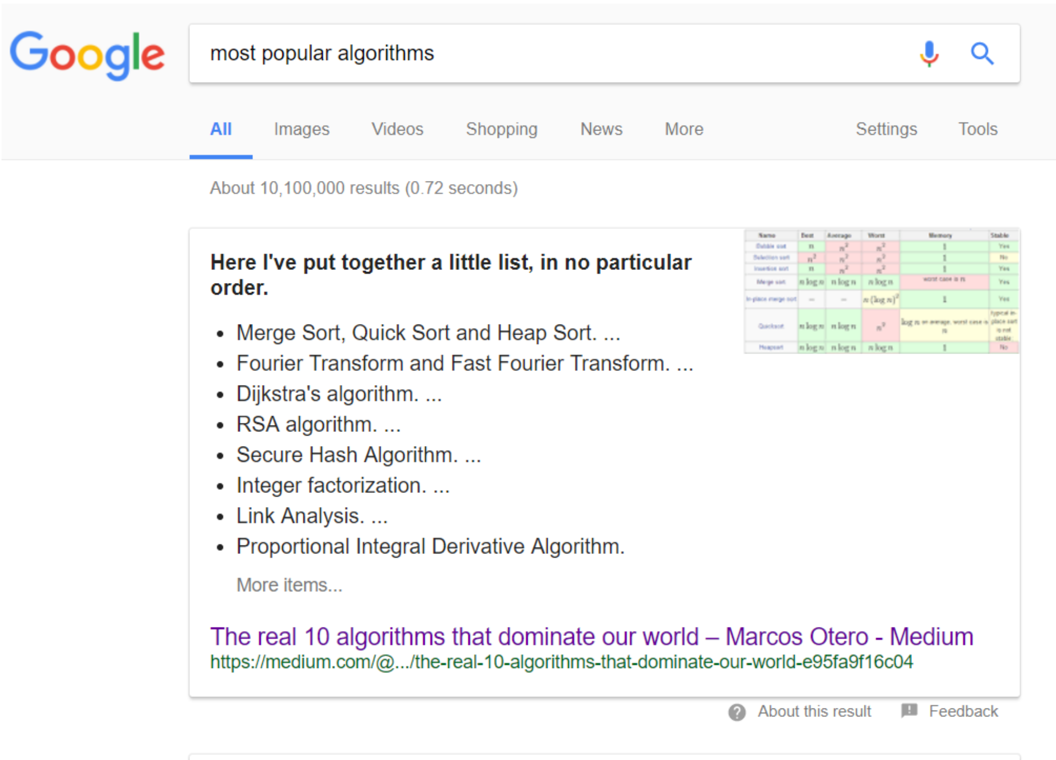
- So Dijkstra came up with the shortest distance algorithm
  - Recall we can backtrack (previous) to get the path

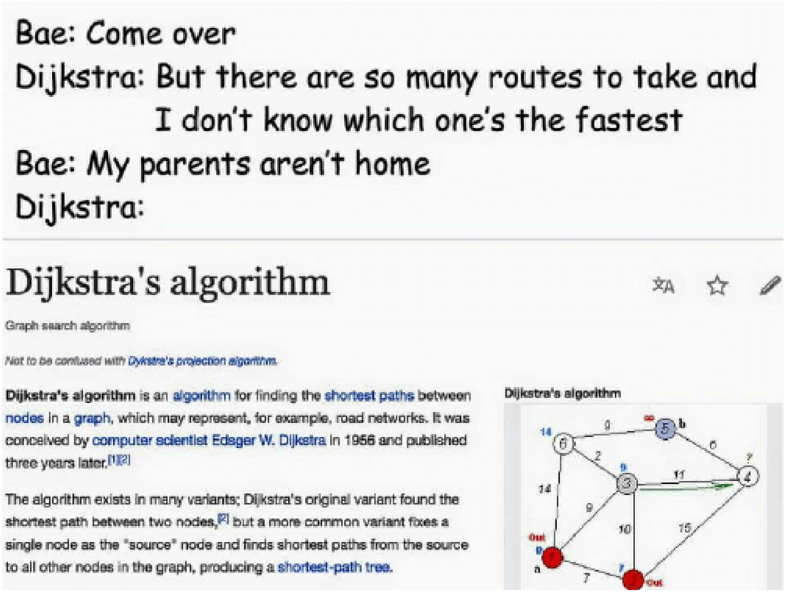


How Dijkstra came up with his algorithm

Graph

Shortest path with Dijkstra





How Dijkstra came up with his algorithm

- It is a combination of 2 algorithms

- It is a combination of 2 algorithms
  - Dynamic programming
  - Greedy

change queue in BFS into Heap (re-arrange itself with priority)
to become Dijkstra

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy

MONASH University

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy
    If I am at A, I can reach B and C. B is the closest, so I go to B and this is the shortest from A to B. I do not need to check if A to C then C to B (A->C->B) is the shortest anymore.

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy
    If I am at A, I can reach B and C. B is the closest, so I go to B and this is the shortest from A to B. I do not need to check if A to C then C to B (A->C->B) is the shortest anymore.

    but GREED IS NOT GOOD… when will this fail?

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy
    If I am at A, I can reach B and C. B is the closest, so I go to B and this is the shortest from A to B. I do not need to check if A to C then C to B (A->C->B) is the shortest anymore.

    but GREED IS NOT GOOD… when will this fail? When C to B is negative!

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy
    If I am at A, I can reach B and C. B is the closest, so I go to B and this is the shortest from A to B. I do not need to check if A to C then C to B (A->C->B) is the shortest anymore.

    but GREED IS NOT GOOD… when will this fail? When C to B is negative!
  - Thus, Dijkstra doesn’t work for negative edges

- It is a combination of 2 algorithms
  - Dynamic programming
    The minimum distance from A to C can be the minimum of A to B (which we know) and minimum of B to C (which we know as well).
  - Greedy
    If I am at A, I can reach B and C. B is the closest, so I go to B and this is the shortest from A to B. I do not need to check if A to C then C to B (A->C->B) is the shortest anymore. since no checking, don't know the C ot B combine with A to C can have shorter distance than A to B directly

    but GREED IS NOT GOOD… when will this fail? When C to B is negative!

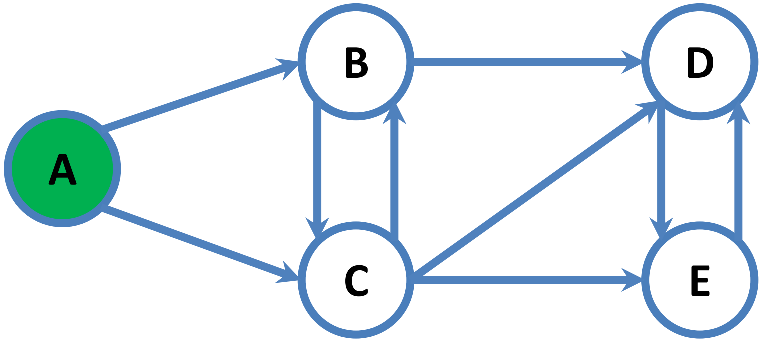
  - Thus, Dijkstra doesn't work for negative edges
    Note: might work at times when the negative edge isn't part of a cycle

Questions?

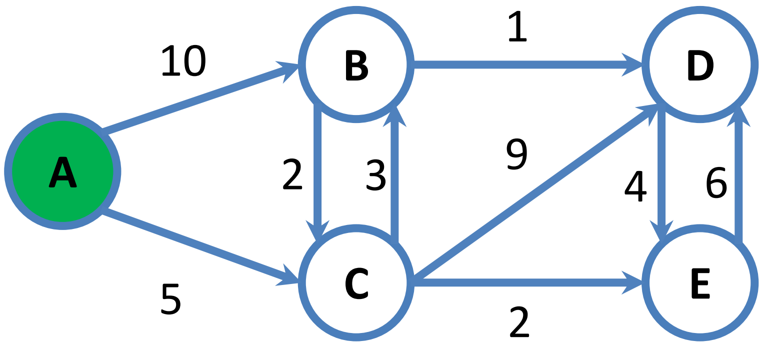
- So how does Dijkstra work?

- So how does Dijkstra work?
  - Consider the following directed graph

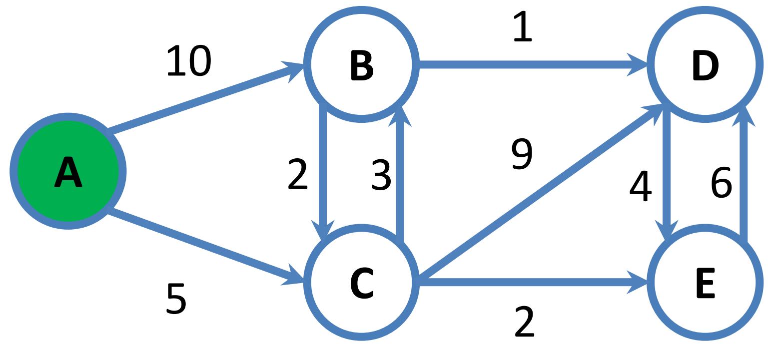
distance of visited vertex can not be changed



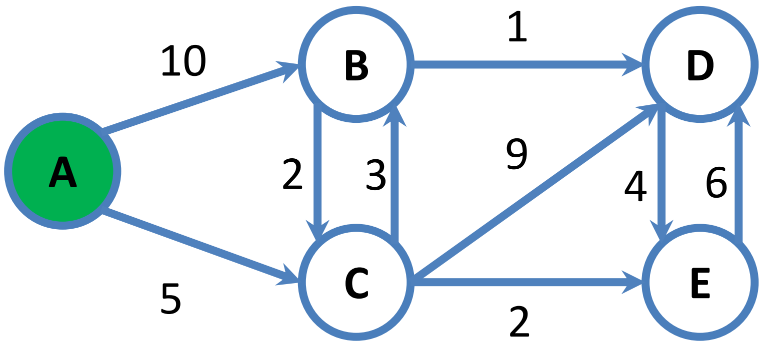
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted
  - So let us begin the algorithm…



- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted
  - So let us begin the algorithm…
  - We are at A (source), and



Graph
Shortest path with Dijkstra

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted
  - So let us begin the algorithm…
  - We are at A (source), and

Graph
Shortest path with Dijkstra

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted

  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted

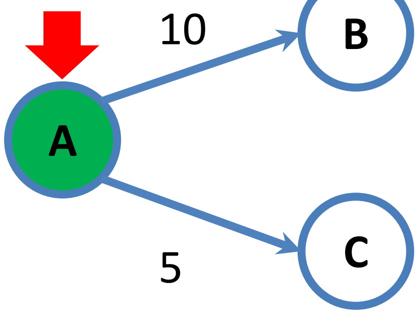
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)

Graph
Shortest path with Dijkstra

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted

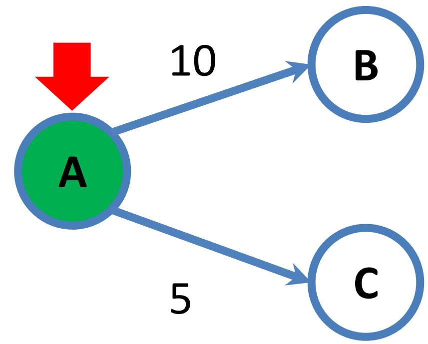
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0
    - B = infinity
    - C = infinity
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



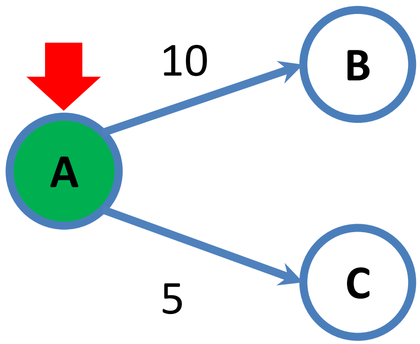
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A)
    - B = infinity
    - C = infinity
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



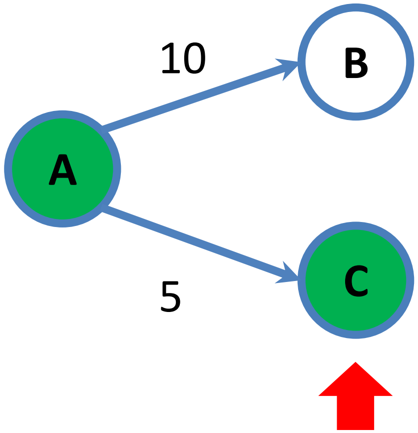
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10
    - C = 5
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



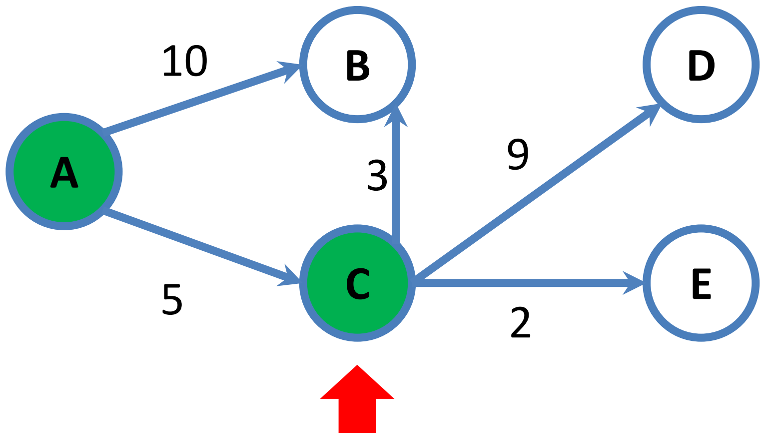
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10
    - C = 5
    - D = infinity
    - E = infinity
    - Closest is C, so we move to C

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10
    - C = 5
    - D = infinity
    - E = infinity
    - Closest is C, so we move to C

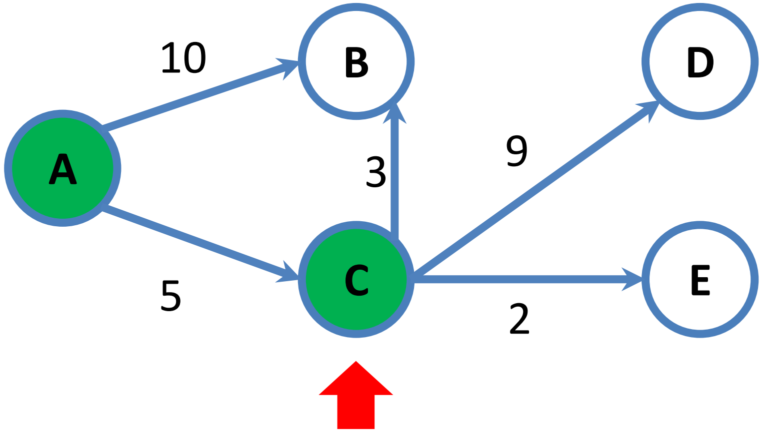
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10
    - C = 5, from here, we can see B, D and E
    - D = infinity
    - E = infinity

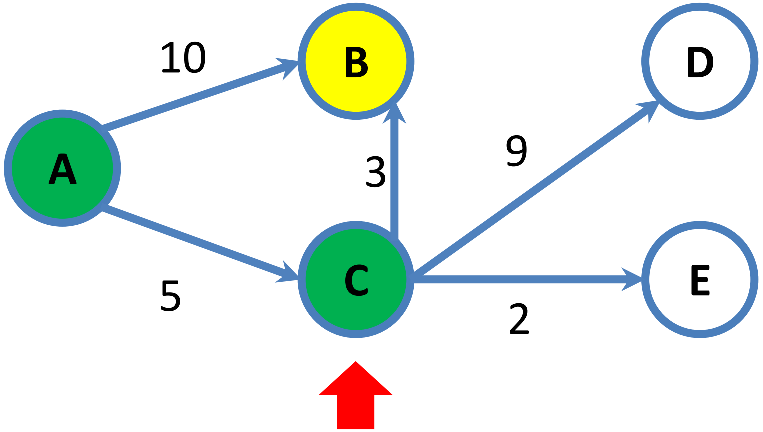
Graph
Shortest path with Dijkstra

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



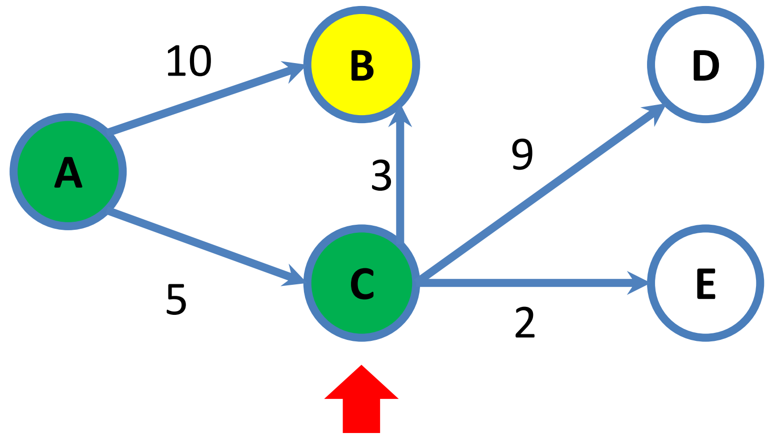
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



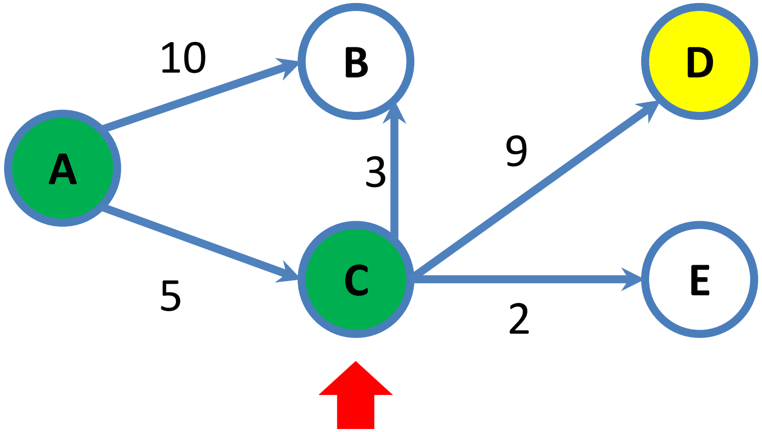
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 10 (A->B) vs 8 (A->C->B)
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



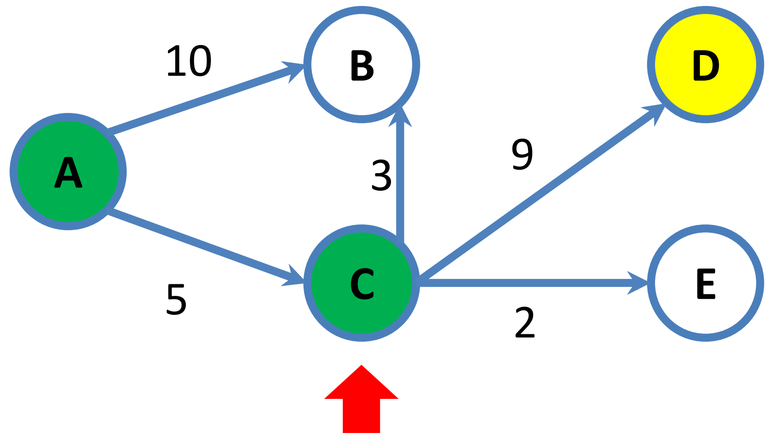
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = infinity
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



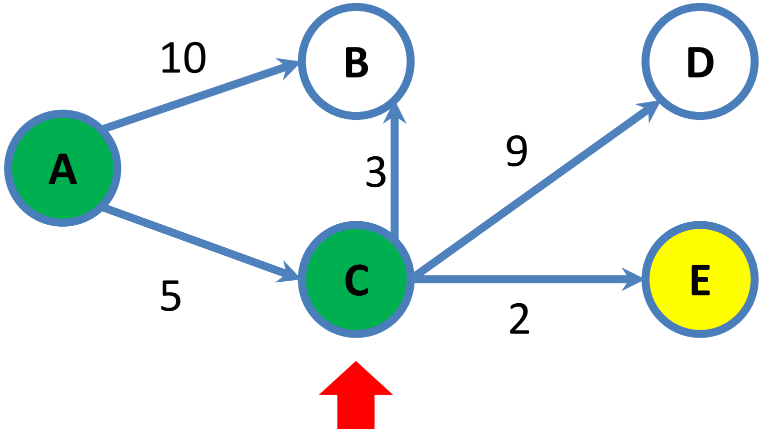
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9?
    - E = infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



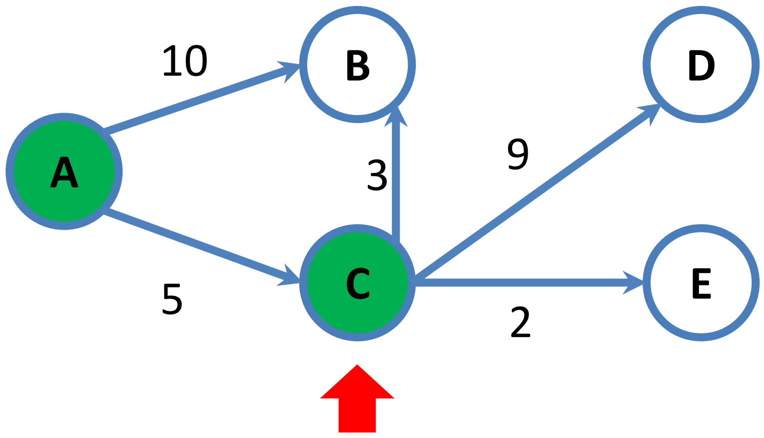
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14 because distance is from A
    - E = infinity comparing with infinity

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



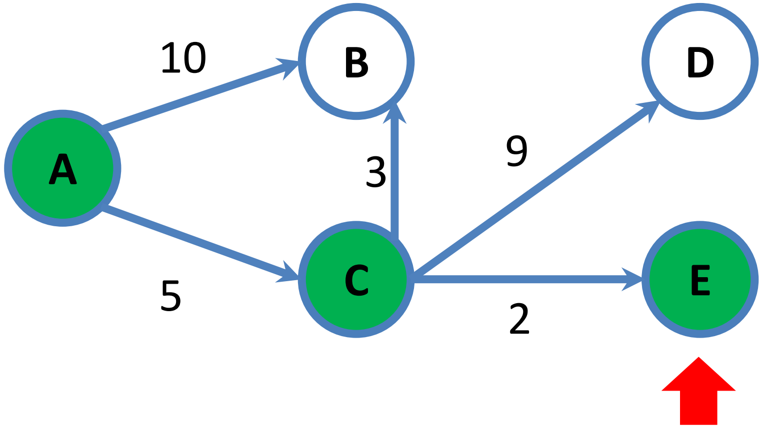
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14
    - E = 7

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



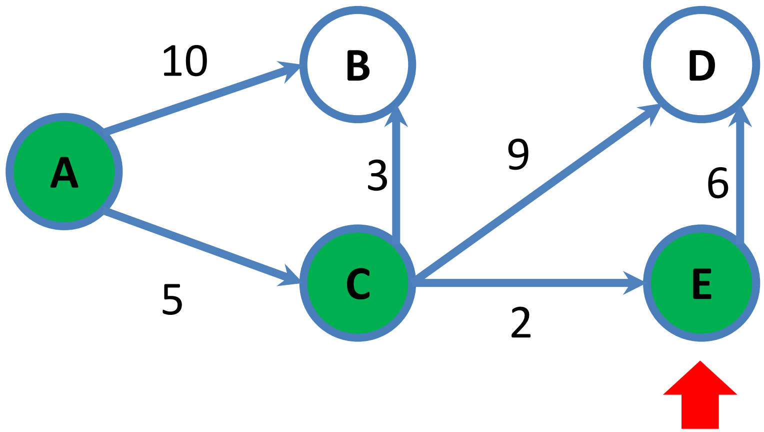
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14
    - E = 7
    - Closest is E, so we go E

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



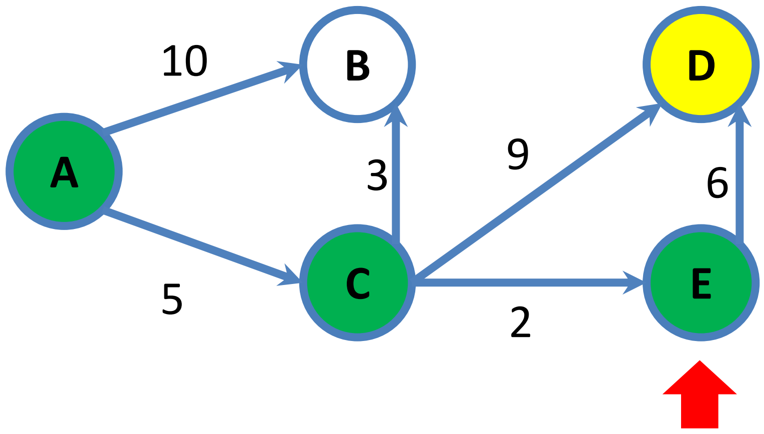
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14
    - E = 7
    - Closest is E, so we go E

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted

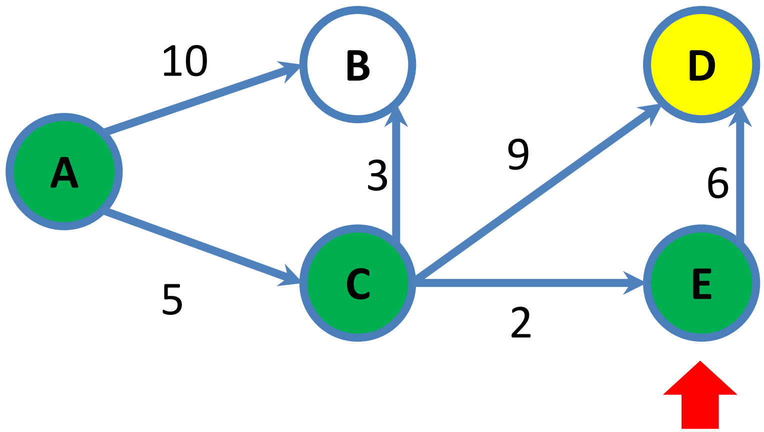


  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14
    - E = 7, from here, we can see D.

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted

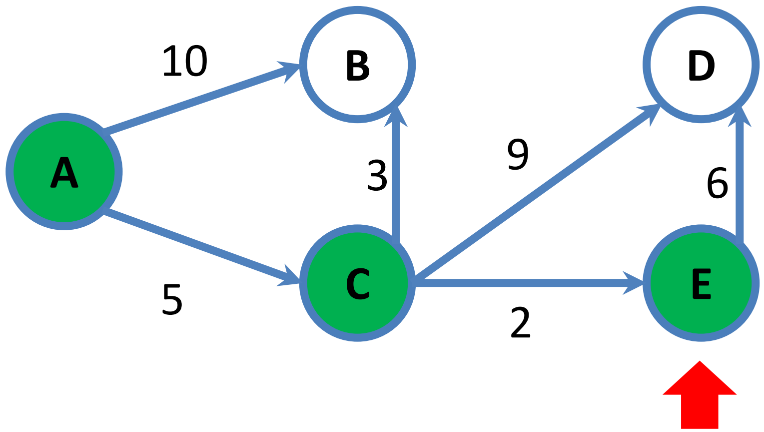


  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 14 vs 7+6 (A->E->D)
    - E = 7, from here, we can see D. Update the distance



- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance

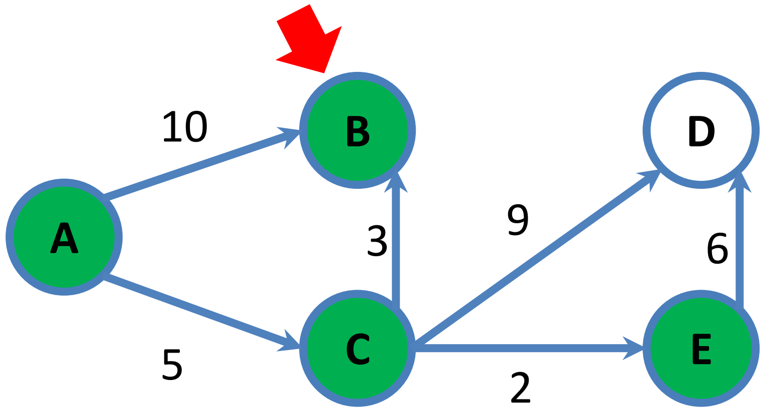
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance
    - Closest is B, so we go B

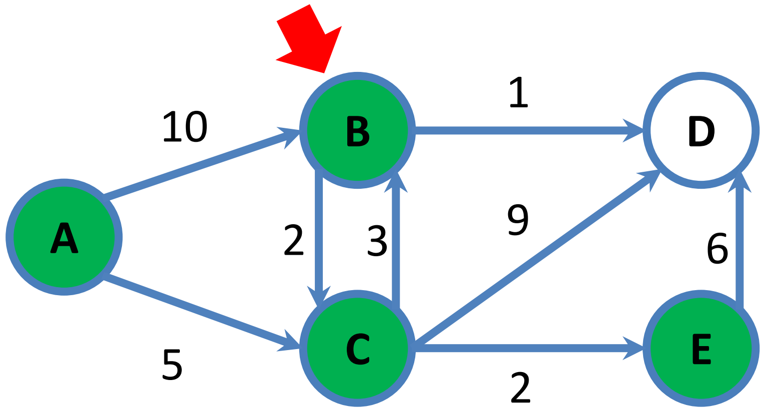
Shortest path with Dijkstra

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



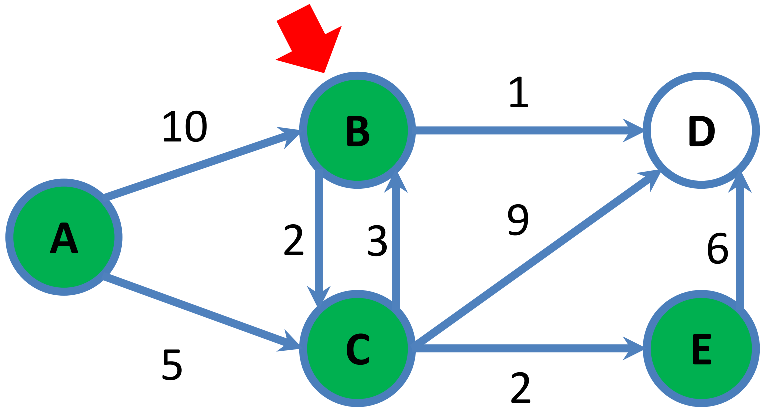
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance
    - Closest is B, so we go B

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



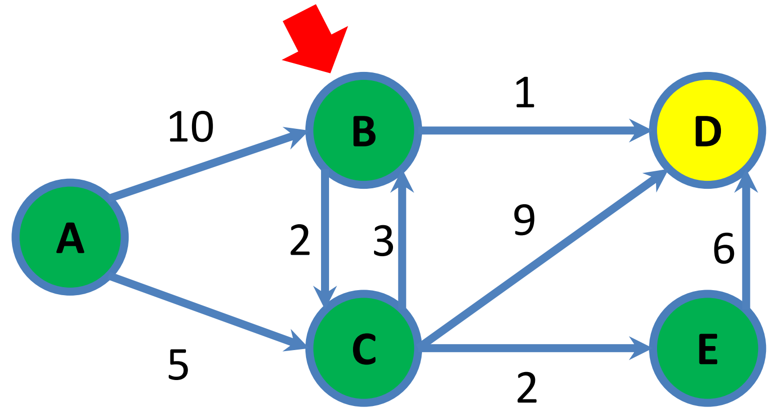
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for C?
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance

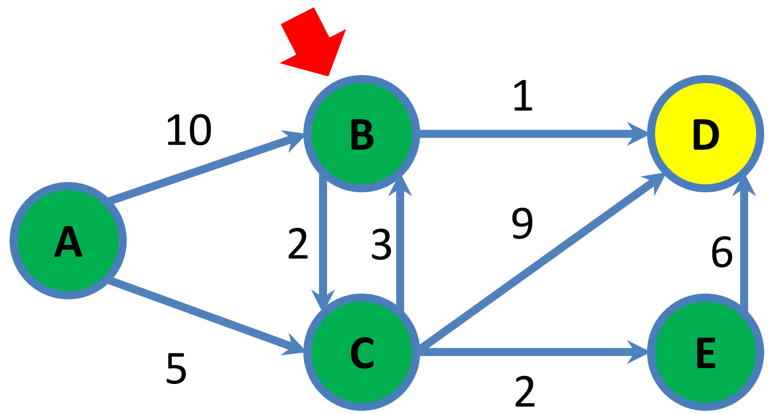
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 13
    - E = 7, from here, we can see D. Update the distance

Graph
Shortest path with Dijkstra

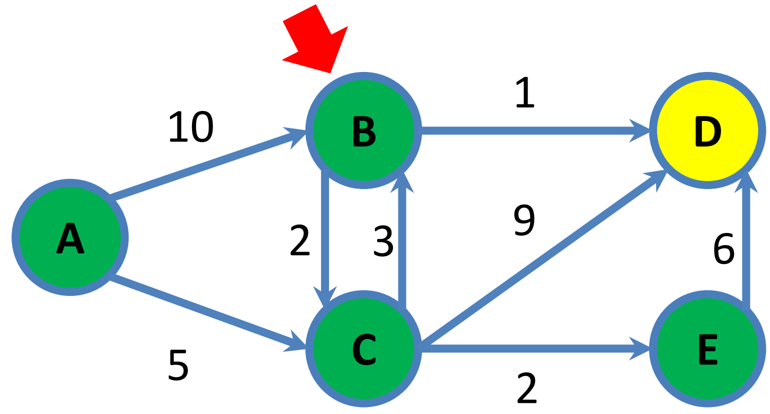
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9 (8+1 via A->B->D)
    - E = 7, from here, we can see D. Update the distance

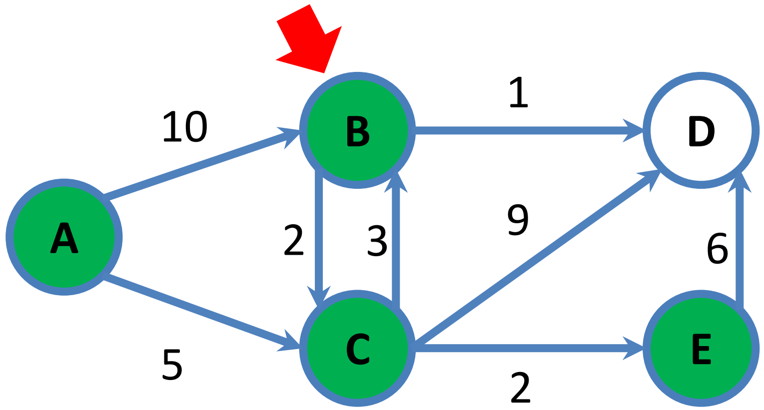
update distance from source A to the discovered vertex
in A, compare distance to B and C, choose the smallest distance among the discovered vertices to visit
the visited vertex can not change its distance
the visited vertex can not change its distance

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9
    - E = 7, from here, we can see D. Update the distance

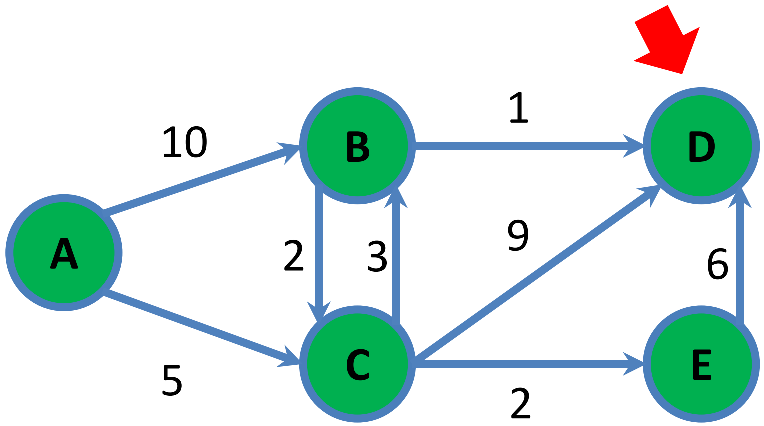
- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9
    - E = 7, from here, we can see D. Update the distance
    - Closest is D, so we move there

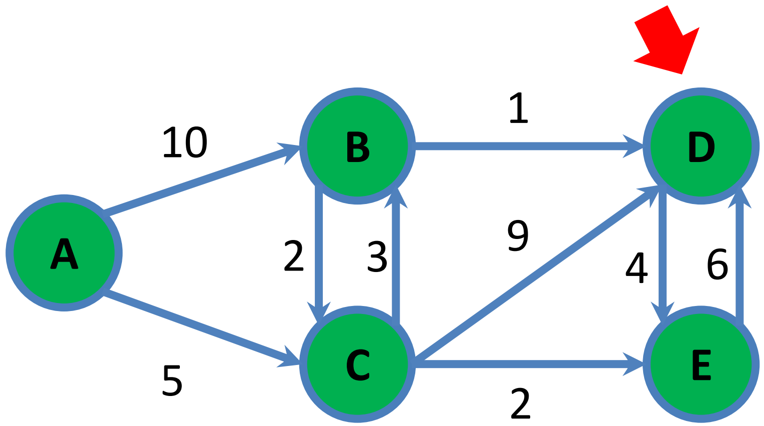
MONASH University

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



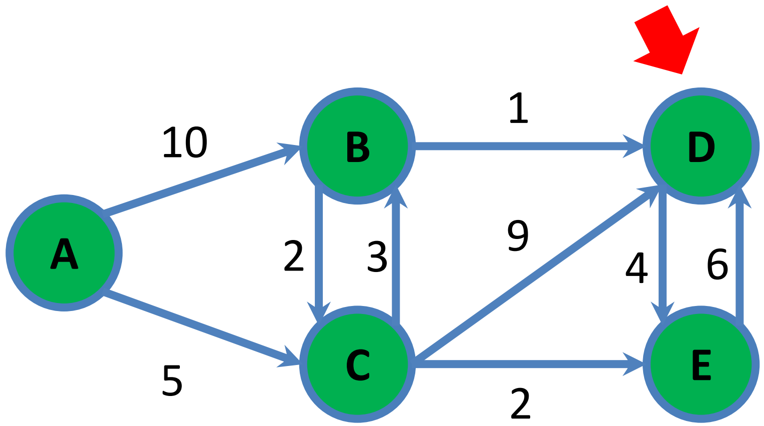
  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9
    - E = 7, from here, we can see D. Update the distance
    - Closest is D, so we move there

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9, from here, we can see E but E is already finalized
    - E = 7, from here, we can see D. Update the distance
    - Closest is D, so we move there

- So how does Dijkstra work?
  - Consider the following directed graph
    - Graph is weighted



  - So let us begin the algorithm…
  - So what happen is we will slowly wander to the closest point (from A)
    - A = 0, from here, we can see B and C (edges from A). Update distance
    - B = 8, from here, we can see C and D. Update distance for D.
    - C = 5, from here, we can see B, D and E. Update the distance
    - D = 9, from here, we can see E but E is already finalized
    - E = 7, from here, we can see D. Update the distance
    - And we are done!

Questions?

- Algorithm?

- Algorithm? Very similar to the BFS except…

- Algorithm? Very similar to the BFS except…
  - Priority queue instead of a normal queue
    - Serve the closest vertex (not finalized)

- Algorithm? Very similar to the BFS except…
  - Priority queue instead of a normal queue
    - Serve the closest vertex (not finalized)
  - Update the distance if the neighbour vertex is visited but not finalized
    - To the shorter one

- This is the BFS algorithm, we change to Dijkstra now
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

- This is the BFS algorithm, we change to Dijkstra now
  - Let say we begin from vertex A
  - Have a queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
      - v.distance = u.distance + 1

- Try to modify this as part of the in-class activity use MinHeap()

- This is the BFS algorithm, we change to Dijkstra now
  - Let say we begin from vertex A
  - Have a priority queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v,w> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
        - » Set v.distance = u.distance + w   v.previous = u, for the first path from the source to v has v.distance
      - If vertex v is discovered but not visited and v.distance > u.distance + w
        - » Update v.distance = u.distance + w   if new path found with lower distance from source to v, then v.previous = new u, to switch to second path that has lower v.distance

edge realisation

discovered = MinHeap()
discovered.append(source.distance, source)   (key, data)
discovered Heap always have lowest distance element at the beginning

- This is the BFS algorithm, we change to Dijkstra now
  - Let say we begin from vertex A
  - Have a priority queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v,w> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
        - » Set v.distance = u.distance + w
      - If vertex v is discovered but not visited and v.distance > u.distance + w
        - » Update v.distance = u.distance + w
  - We use a min-heap for our priority queue!

- This is the BFS algorithm, we change to Dijkstra now
  - Let say we begin from vertex A
    like Minheap, for discovered, to add in vertex with distance, Minheap woud prioritise the one with lowest distance
  - Have a priority queue for discovered
    - Put source (A) into it with a distance 0
  - While discovered is not empty
    - Serve from discovered, to visited
    - For each edge <u,v,w> where u is the served
      - If vertex v is not discovered or visited, add to discovered queue
        - » Set v.distance = u.distance + w
      - If vertex v is discovered but not visited and v.distance > u.distance + w
        - » Update v.distance = u.distance + w
  - We use a min-heap for our priority queue!
    - Note that we need a pointer to the nodes to update distance in O(1)

MONASH University

- Algorithm can be as follows (might differ):

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Questions?

- Complexity?

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Graph

Shortest path with Dijkstra

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Graph
Shortest path with Dijkstra

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

O(V)

discovered-queue

Graph

Shortest path with Dijkstra

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

if dense graph,maximum number of edges from each vertex
= V (total number of vertices) -1

upheap() method in MinHEAP

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

how many V, how many

edges of u only not entire graph

O(V)

bounded by V - 1
u can connect to any other vertices

Graph

Shortest path with Dijkstra

MONASH University

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

O(V)

Update: O(log V)

- Time Complexity?

MinHeap like Tree

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

O(V)

search the vertex by key in tree

Update: O(log V)

Graph

Shortest path with Dijkstra

- Time Complexity? O(V^2 log V)

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

O(V)

Update: O(log V)

Graph
Shortest path with Dijkstra

- Time Complexity? O(V^2 log V) = O(E log V)

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

Serve: O(log V)

O(V)

Update: O(log V)

Graph

Shortest path with Dijkstra

MONASH University

- Time Complexity? O(V^2 log V) = O(E log V)
  - Recall for dense graph, E ≈ V^2

O(V)

```
discover_queue = MinHeap()
discover_queue.append([source,0])

while discover_queue is not empty:
    u = discover_queue.serve()
    u.visited = True
    for each <u,v,w> in u.edges:
        if v.visited = True:
            pass
        else:
            if v.discovered = False:
                discover_queue.append([v, u.distance+w])
                v.discovered = True
            else:
                if v.distance > u.distance+w:
                    discover_queue.update(v, u.distance+w)
                    v.discovered = True
```

downHeap tree, so log(V)

Serve: O(log V)

O(V)

Update: O(log V)

V^2

- Time Complexity? $O(V^2 \log V) = O(E \log V)$
  - Recall for dense graph, $E \approx V^2$

- Note that with Fibonacci heap instead of your binary heap, we can reduce the complexity further to $O(E + V \log V)$

- Time Complexity? O(V^2 log V) = O(E log V)
  - Recall for dense graph, E ≈ V^2

- Note that with Fibonacci heap instead of your binary heap, we can reduce the complexity further to O(E + V log V) = O(V^2 + V log V) = O(V^2)
  - For dense graph

Questions?

- What if we have a single source
  - As usual
- But single target?

- What if we have a single source
  - As usual
- But single target?
- We can terminate after we have move the target vertex to the visited portion!

- What if we have a single source
  - As usual
- But single target?
- We can terminate after we have move the target vertex to the visited portion!
  - We would have the shortest distance

- What if we have a single source
  - As usual
- But single target?
- We can terminate after we have move the target vertex to the visited portion!
  - We would have the shortest distance
  - We can backtrack for the shortest path
    - Via vertex.previous attribute

Questions?

- Why does Dijkstra work?

- Why does Dijkstra work?
  - Let us use Nathan’s slides

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

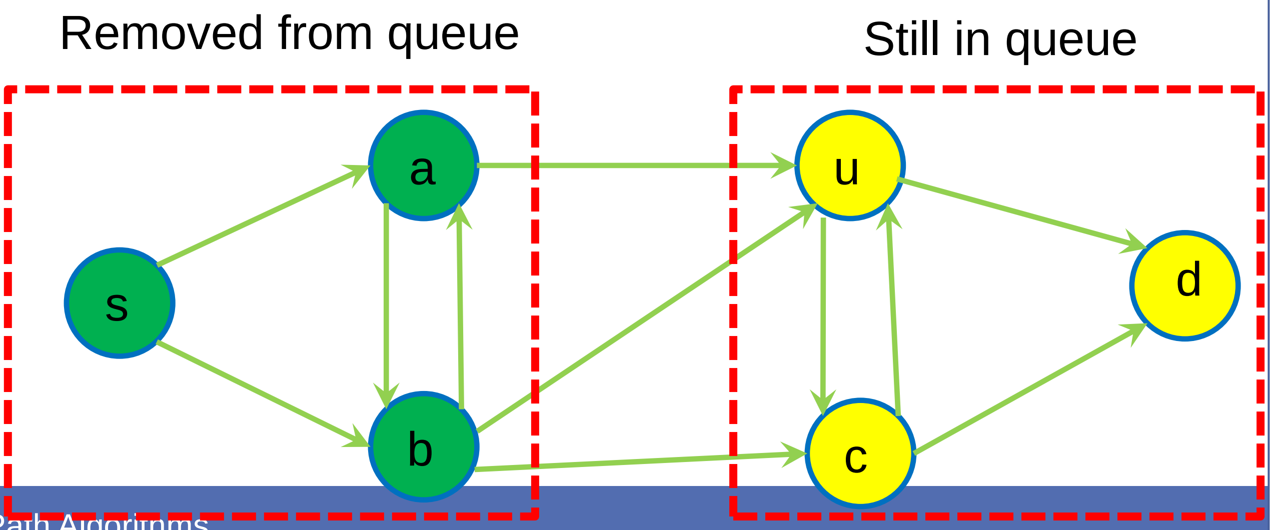
- Dist[s] is initialised to 0, which is the shortest distance from s to s (since there are no negative weights)

Proof of Correctness

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

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



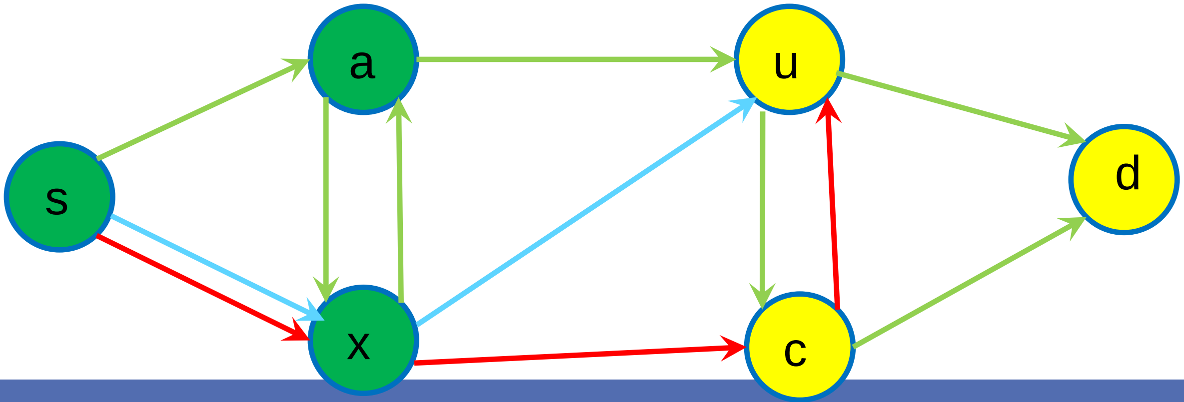
Proof of Correctness

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

Inductive Step:

- Suppose (for contradiction) there is a shorter 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)



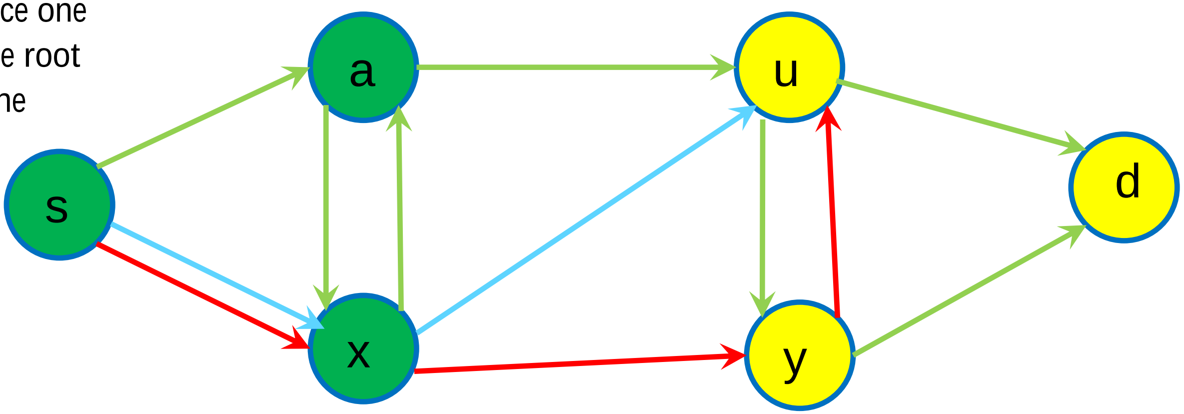
Proof of Correctness

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

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($s \rightsquigarrow y$) <= len(P) < dist[u]

MinHeap() only serve the minuum distance one
so, MinHeap() would serve the one on the root
and compare its children roots and put one
to be root, so O(log(V))

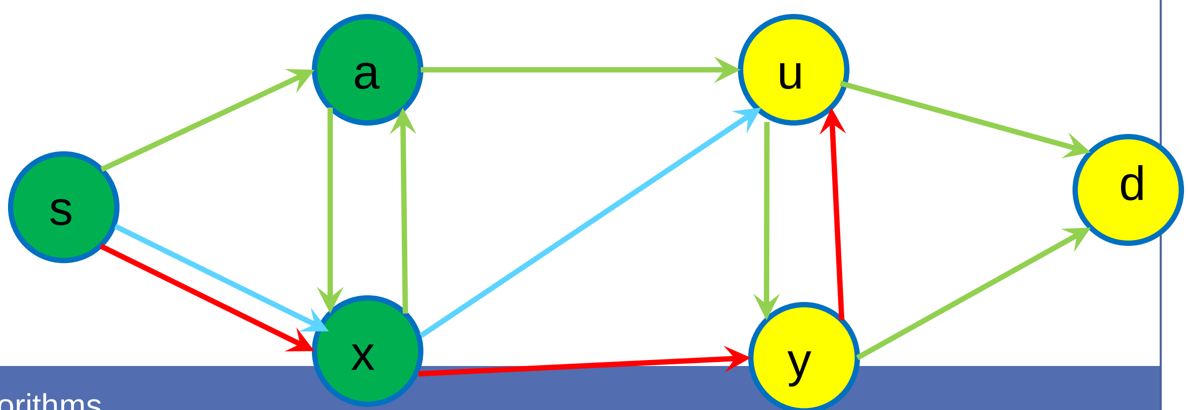


Proof of Correctness

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

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]???



- Why does Dijkstra work?
  - Let us use Nathan's slides
  - Or let me just explain it on the whiteboard…
  - Via proof by contradiction!

Questions?

Graph

Other shortest path?

- Bellman-Ford not work for negative distances
- Floyd-Warshall all pair

Other shortest path?

- Bellman-Ford
- Floyd-Warshall
  - With transitive closure

Other shortest path?

- Bellman-Ford
- Floyd-Warshall
  - With transitive closure

- We see it later in next lectures

Other shortest path?

- Bellman-Ford
  - Single source
  - Can know negative edges
- Floyd-Warshall
  - With transitive closure
  - Single or more sources
  - Can know negative edges

- We see it later in next lectures

Questions?

MONASH University

Thank You