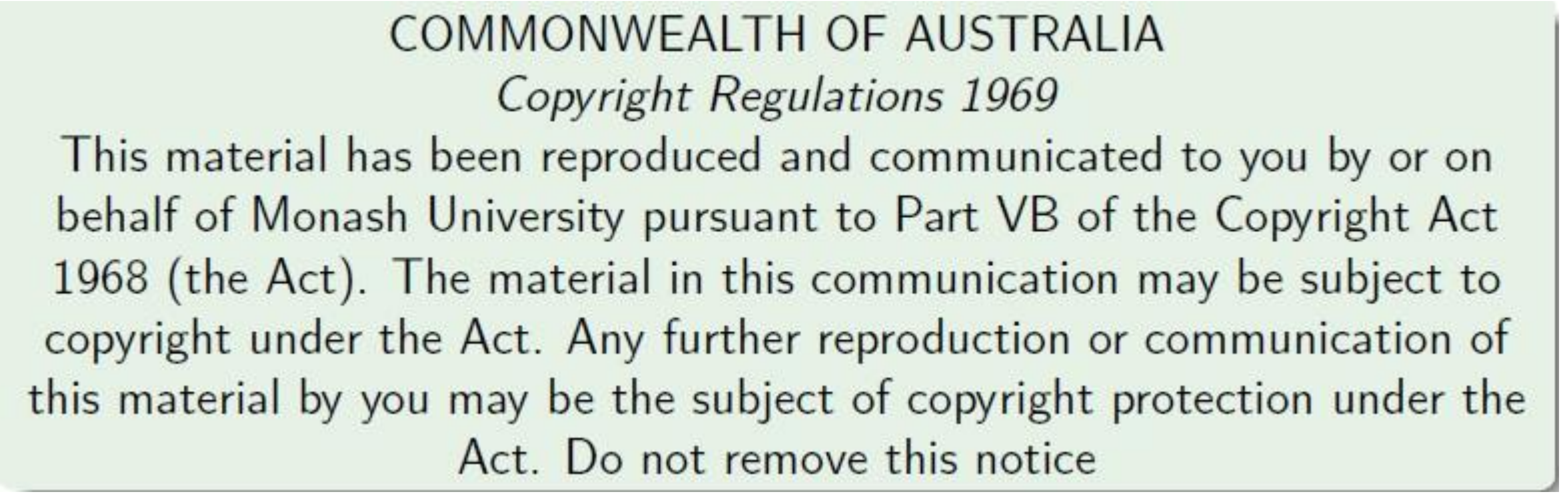
Faculty of Information Technology, Monash University



FIT2004: Algorithms and Data Structures

Week 8: Introduction to Graphs and Shortest Path Algorithms

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

Recommended reading

- Unit notes: Chapters 12&13
- Cormen et al. Introduction to Algorithms.
  - Section 22.1 Representation of graphs
  - Section 22.2 Breadth-First Search
  - Section 22.3 Depth-First Search
  - Section 24.3 Dijkstra's algorithm
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/
- http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Graph/Directed/

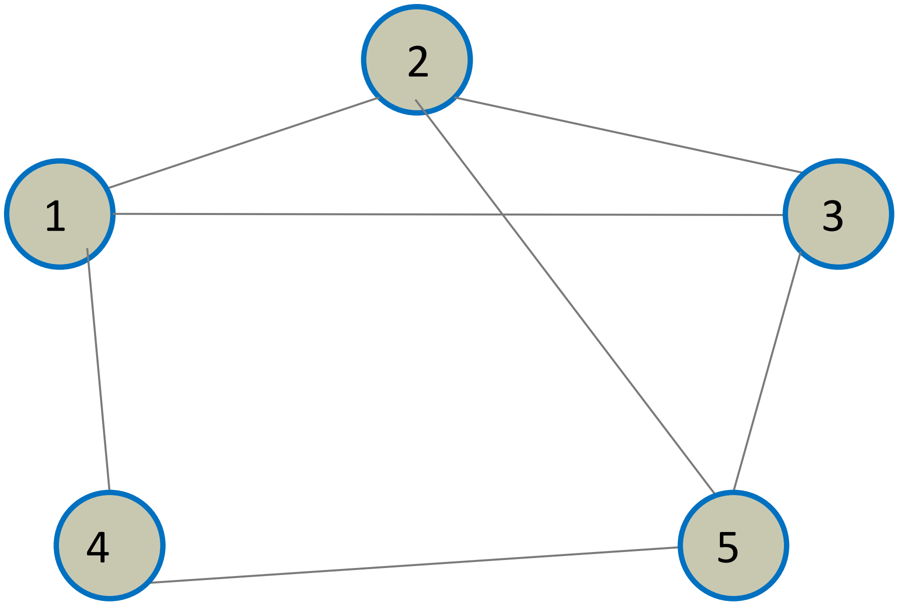
Outline

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

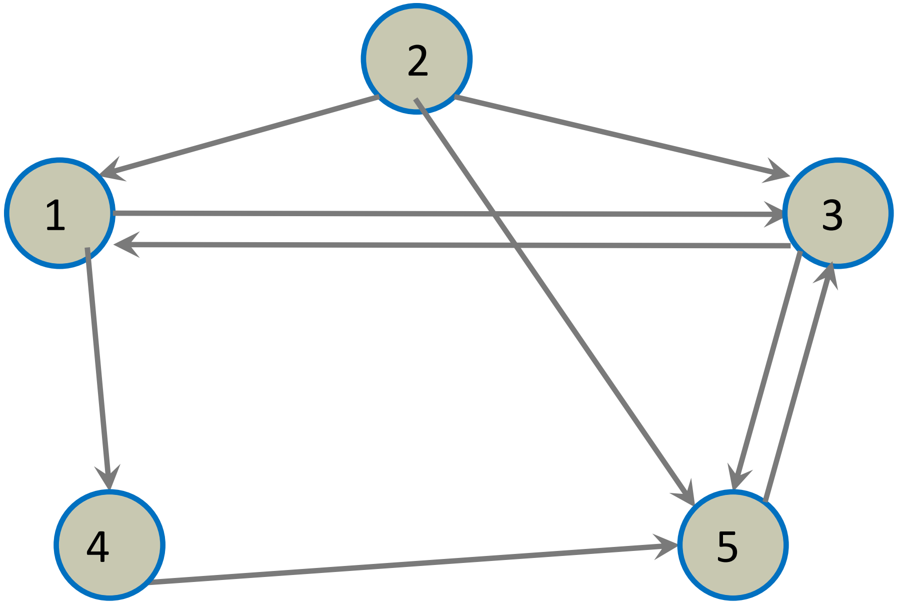
Graphs

- A graph is simply a way of encoding pairwise relationships among a set of objects.
- Each object is called a vertex or node.
- Each edge of the graph “connects” two nodes.

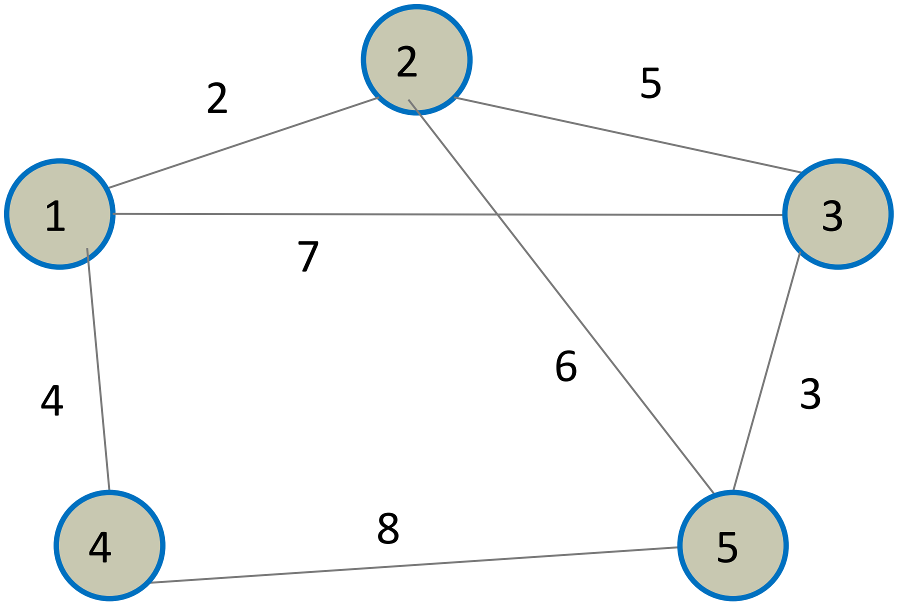
Undirected Graph - Example



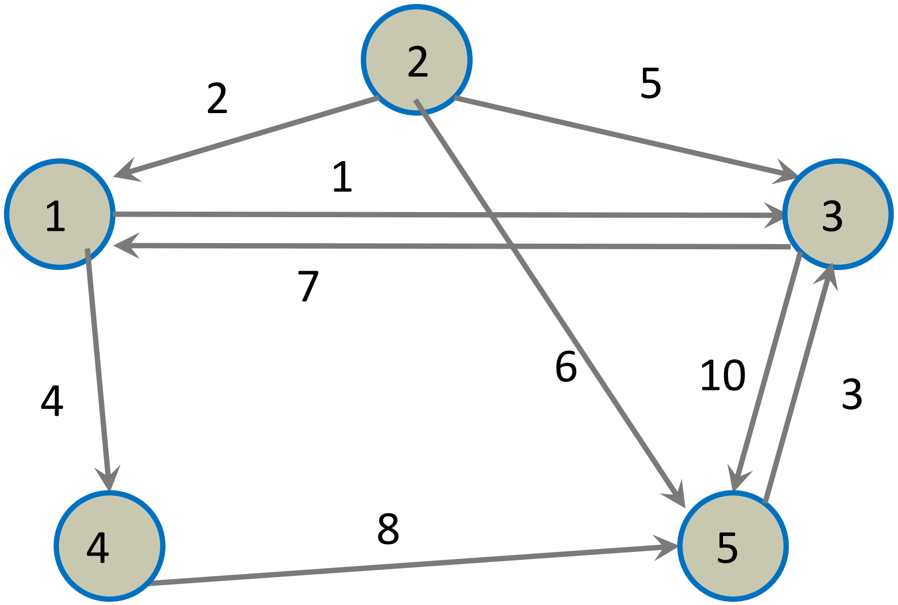
Directed Graph - Example



Undirected Weighted Graph - Example



Directed Weighted Graph - Example



Uses of Graphs

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

Graphs – Formal notations

- A graph G = (V, E) is defined using a set of vertices V and a set of edges E.
- An edge e is represented as e = (u, v) where u and v are two vertices
- For undirected graphs, (u, v) = (v, u) because there is no sense of direction. For a directed graph, (u, v) represents an edge **from** u **to** v and (u, v) ≠ (v, u).
- We will slightly abuse notation and use V (instead of |V|) for the number of vertices and E (instead of |E|) for the number of edges when what is meant is clear from the context.

Graphs – Formal notations

- A weighted graph is represented as G = (V, E) and each edge (u, v) has an associated weight w.

- A graph is called a **simple graph** if it does not have loops AND does not contain multiple edges between same pair of vertices.

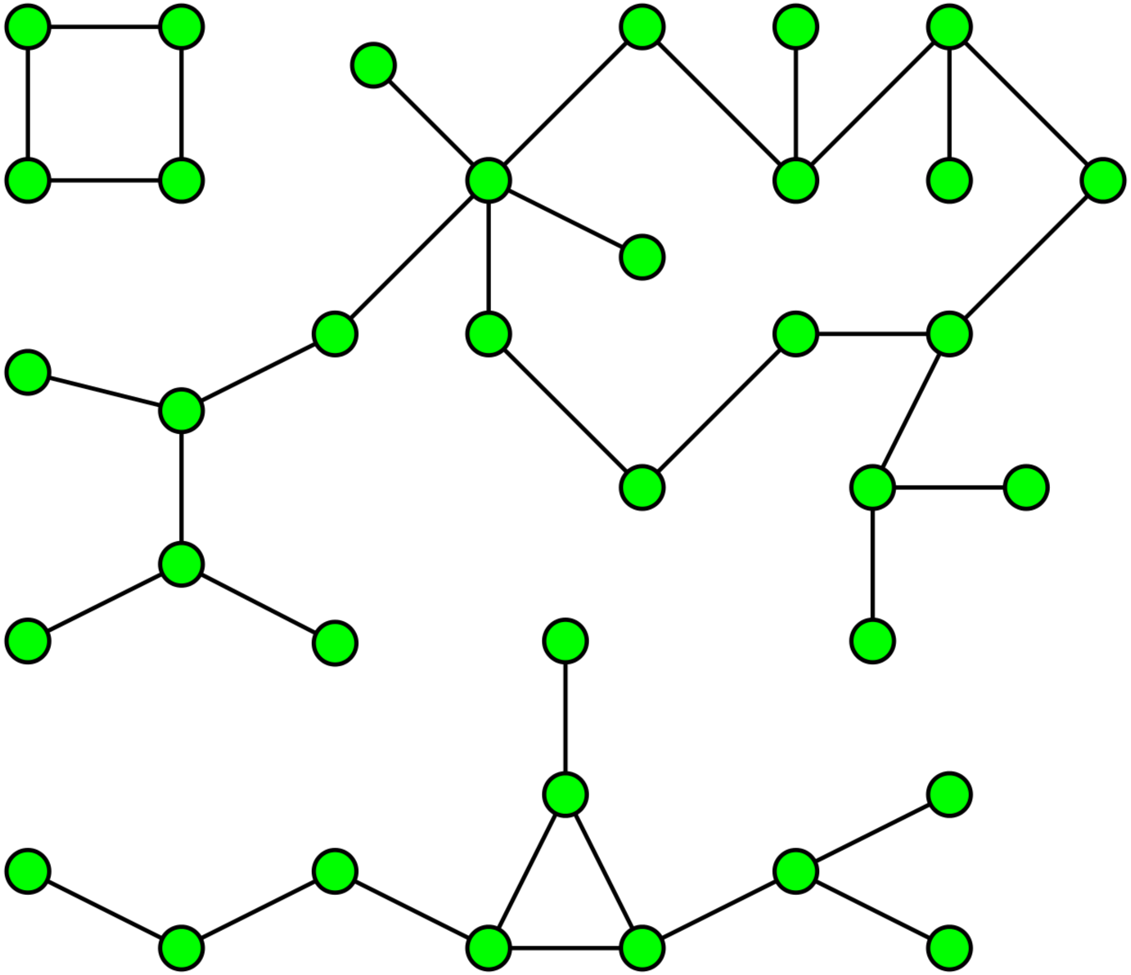
- **In this unit, we focus on simple graphs with a finite number of vertices.**

Graphs – Connected Components

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

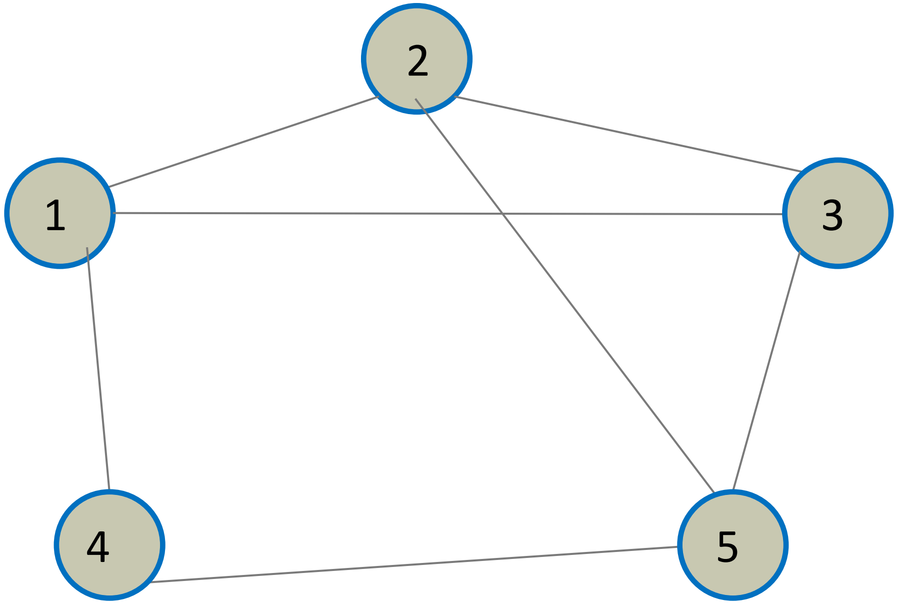
Graphs – Connected Components

- An undirected graph with 3 connected components:



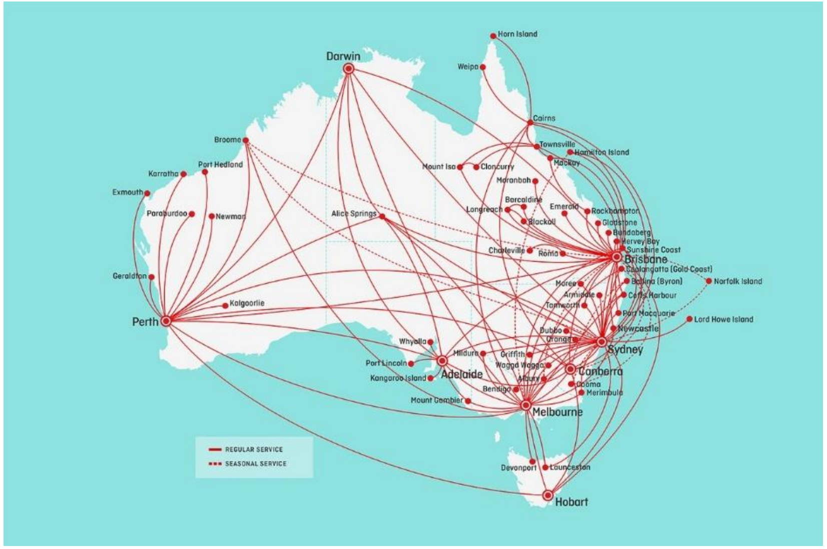
Graphs – Connected Components

- An undirected graph is **connected** if all vertices are part of a single connected component.
- In other words, for any pair of vertices u and v, there is a path between them.



Graphs – Connected Components

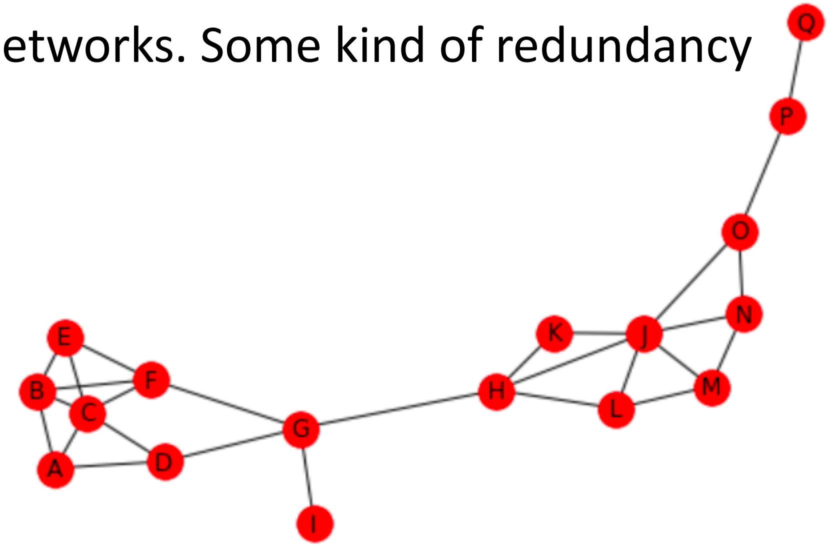
- Why is that an important concept in practice?
  - Example: Airlines normally want their air routes to form a connected graph.



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

Graphs – Connected Components

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



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

Some Graph Properties

Let G be a graph.

- The minimum number of edges in a connected undirected graph
  - ???
- The maximum number of edges in an undirected graph
  - ???

Quiz time!
https://flux.qa - YTJMAZ

Some Graph Properties

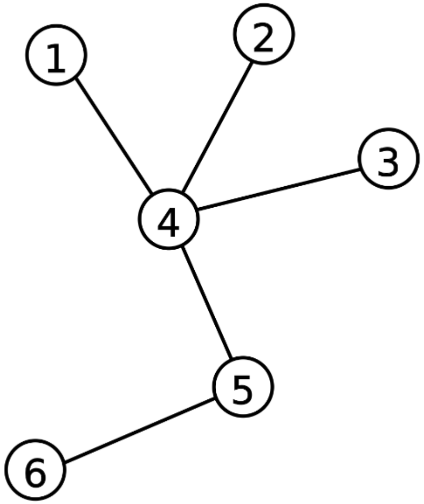
Let G be a graph.

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

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

Tree

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



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

Graphs – Connected Components

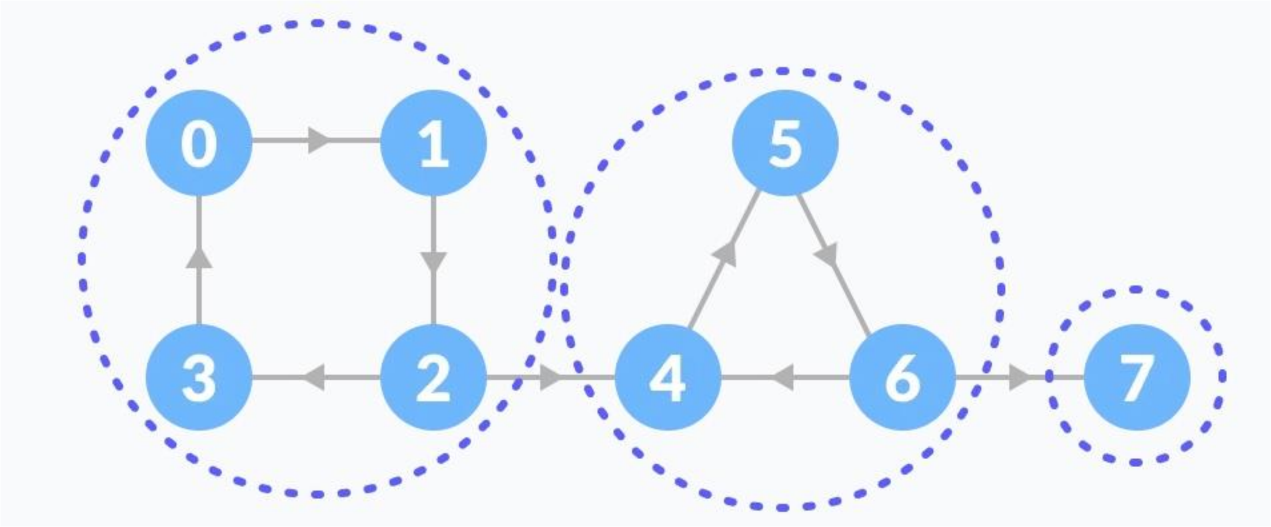
- In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric (i.e., possibly there could be a path from u to v, but no path from v to u).

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

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

Graphs – Connected Components

A directed graph with 3 strongly-connected components:



Graphs – Connected Components

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

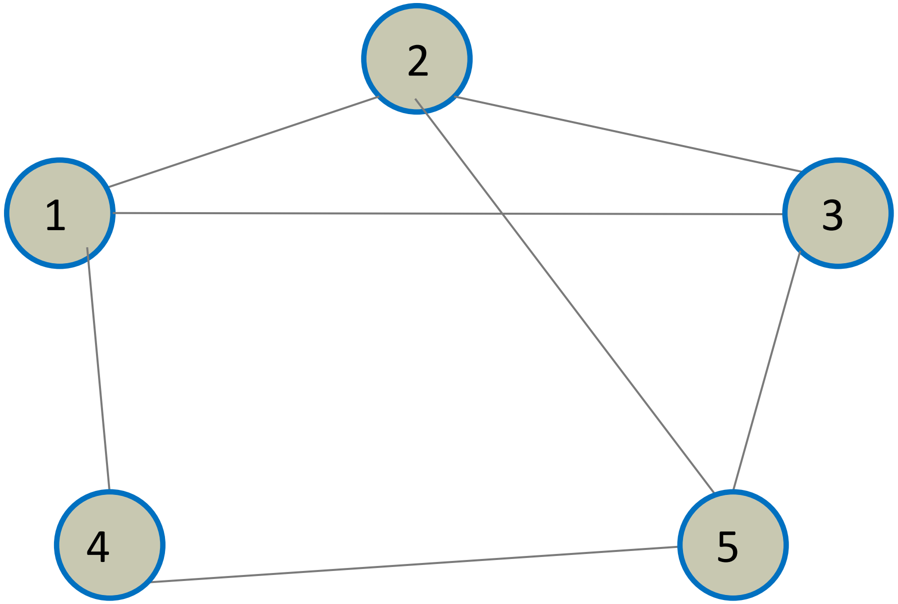
- I.e., the graph only has one strongly-connected component.

Representing Graphs

Adjacency Matrix (Undirected Graph):

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

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

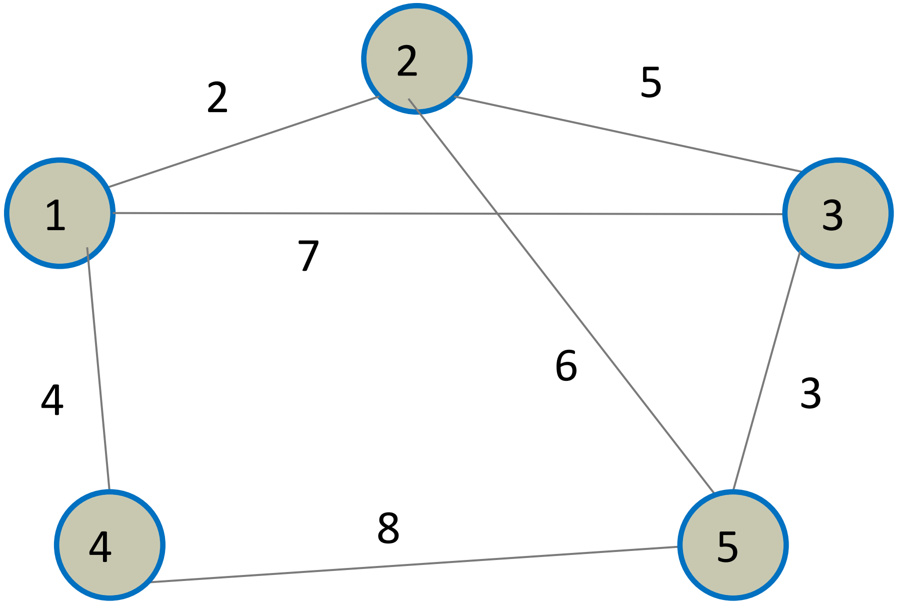


Representing Graphs

Adjacency Matrix (Undirected Weighted Graph):

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

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



Representing Graphs

Adjacency Matrix (Directed Weighted Graph):

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

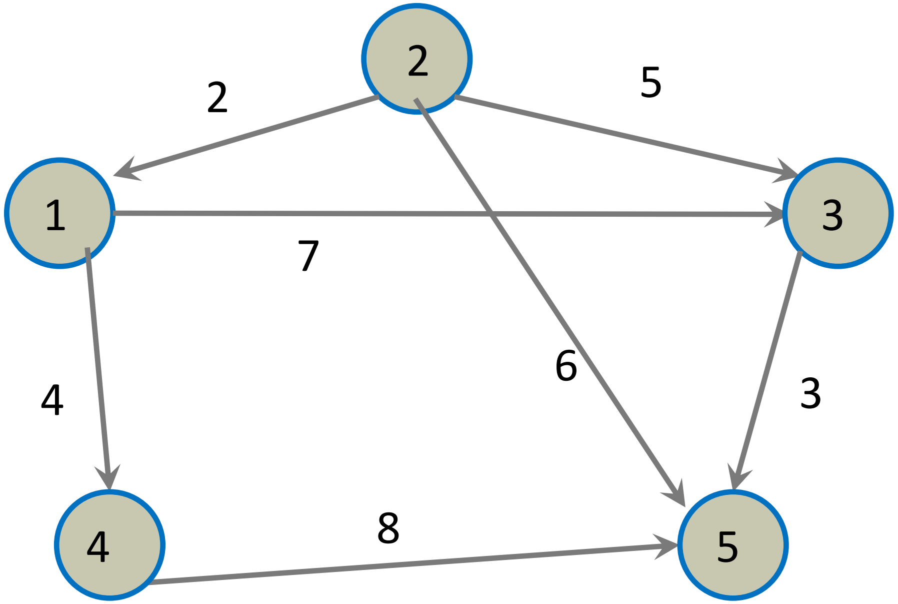
Space Complexity: O(V^2) regardless of the number of edges

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

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

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

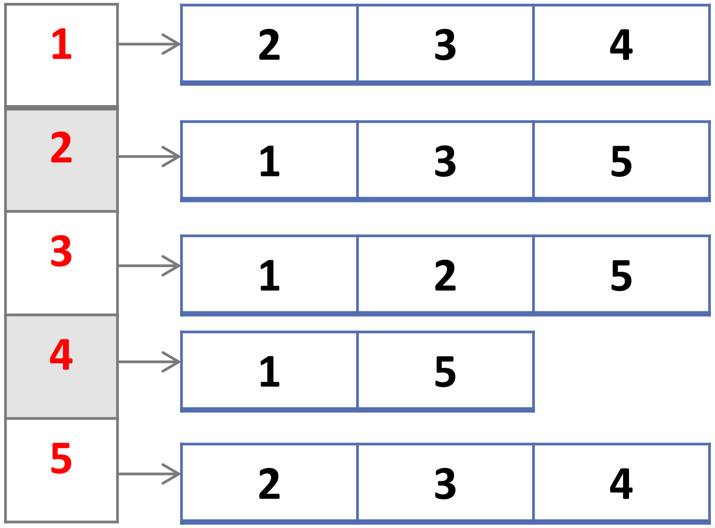
	1	2	3	4	5
1			7	4	
2	2		5		6
3					3
4					8
5					



Representing Graphs

Adjacency List (Undirected Graph):

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



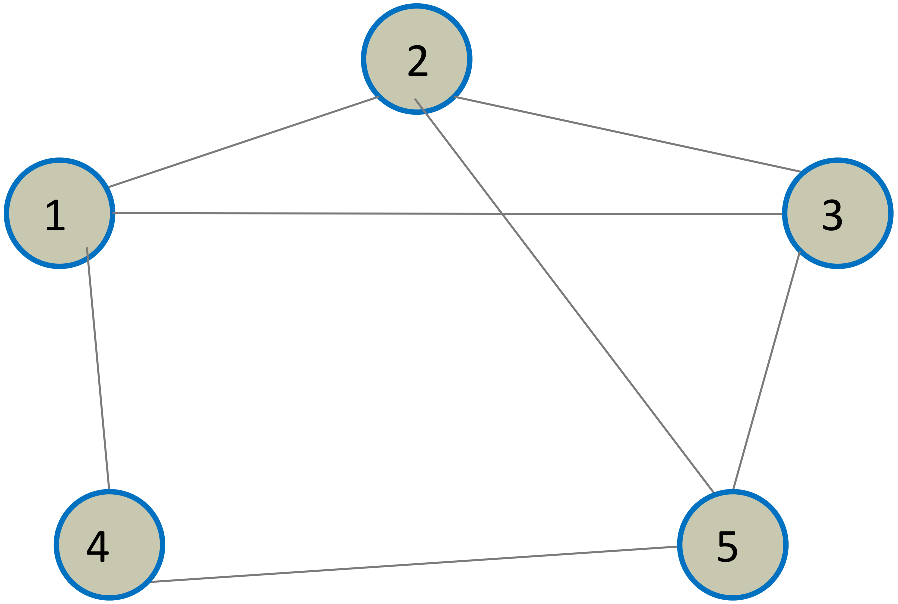
1 → 2 3 4

2 → 1 3 5

3 → 1 2 5

4 → 1 5

5 → 2 3 4



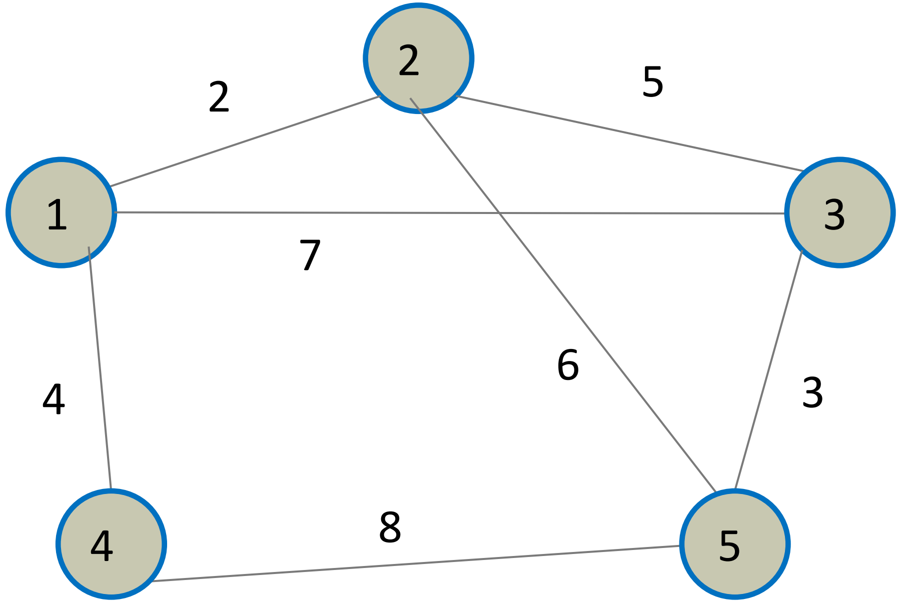
Representing Graphs

Adjacency List (Undirected Weighted Graph):

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

The numbers in parenthesis correspond to the weights.

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



Representing Graphs

Adjacency List (Directed Weighted Graph):

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

Space Complexity:

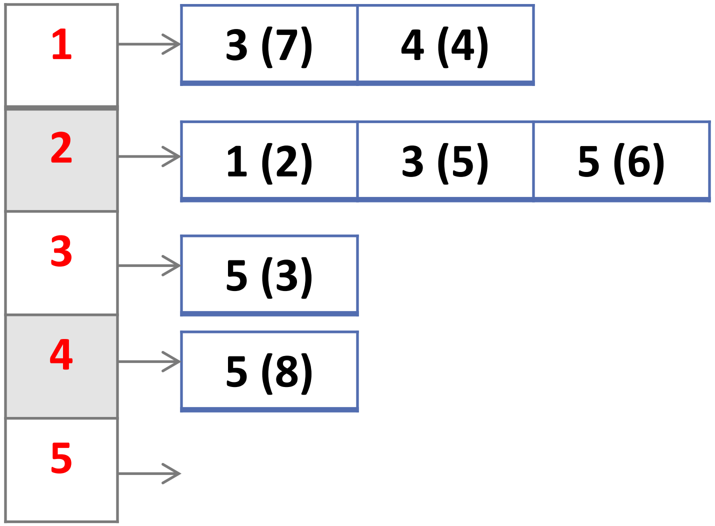
- O(V + E)

Time complexity of checking if a particular edge exists:

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

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

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



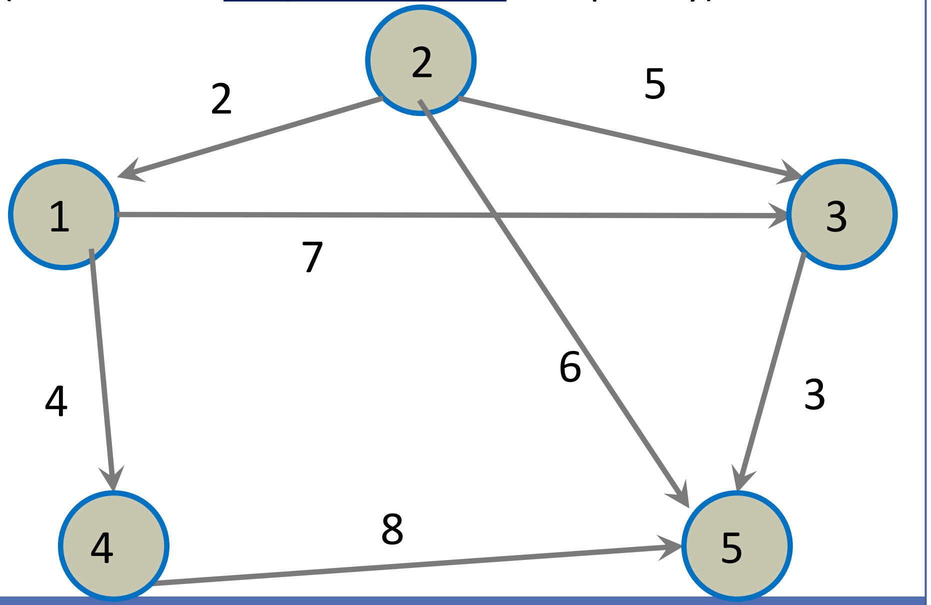
1 → 3 (7) | 4 (4)

2 → 1 (2) | 3 (5) | 5 (6)

3 → 5 (3)

4 → 5 (8)

5 →



Outline

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

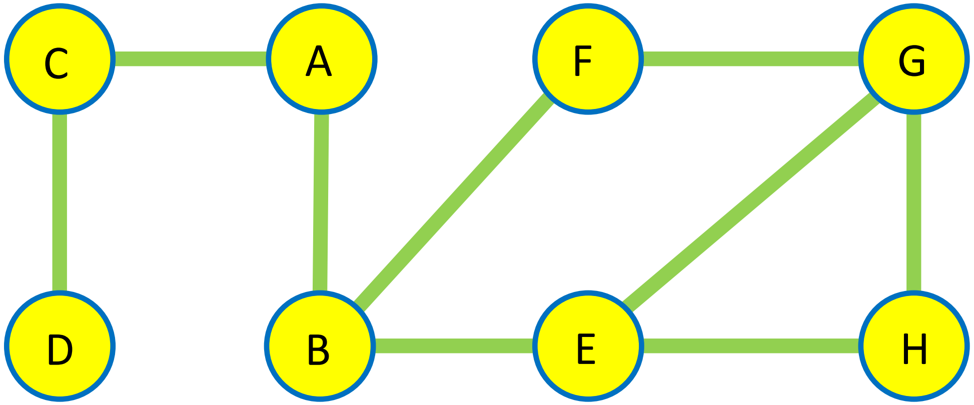
Graph Traversal

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

They are very important in the design of numerous algorithms.

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

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



Both of them visit the vertices exactly once.

They visit vertices in different orders.

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

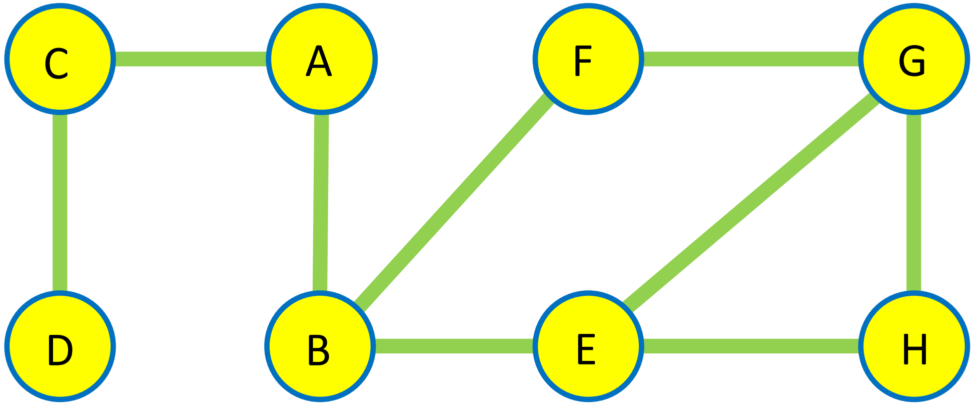
Each one has properties that makes it useful for certain kinds of graph problems.

Graph Traversal - BFS

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

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

Quiz time!
https://flux.qa - YTJMAZ

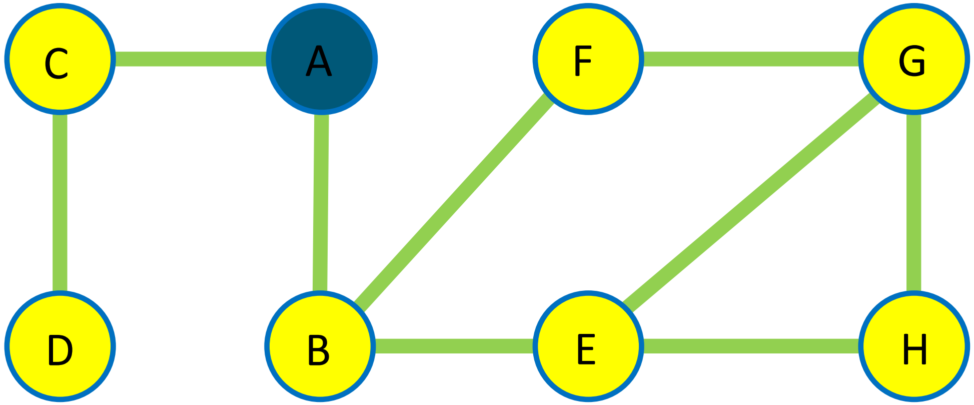


Graph Traversal - BFS

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

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

Yes!

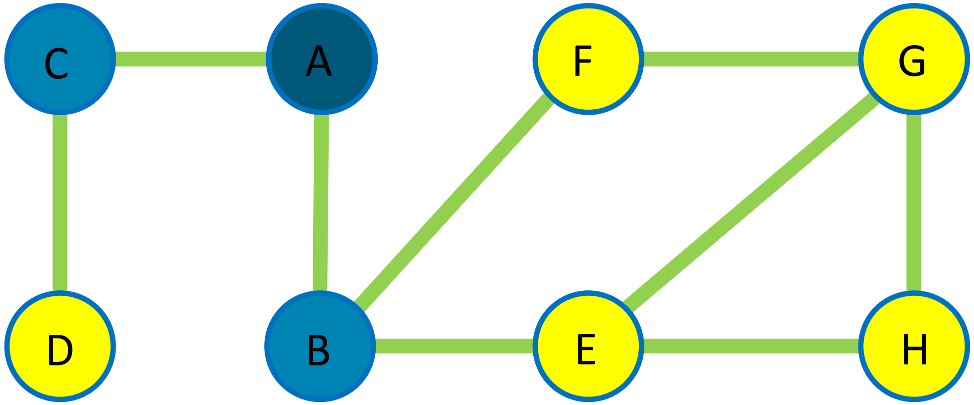


Graph Traversal - BFS

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

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

Yes!

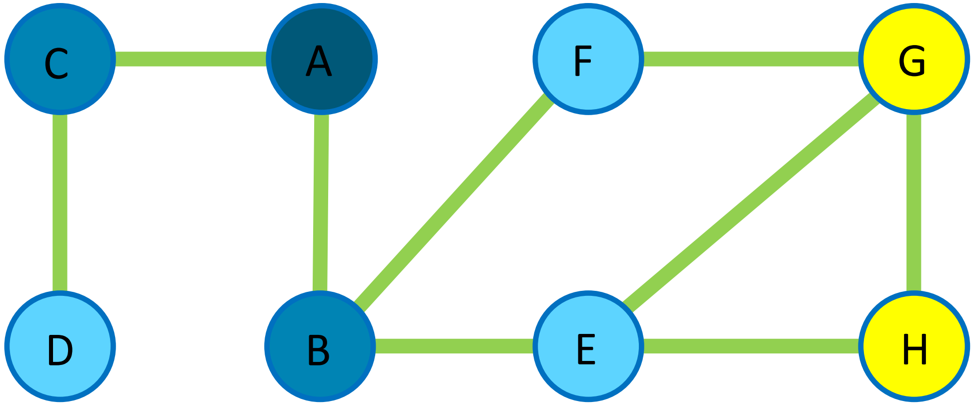


Graph Traversal - BFS

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

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

Yes!

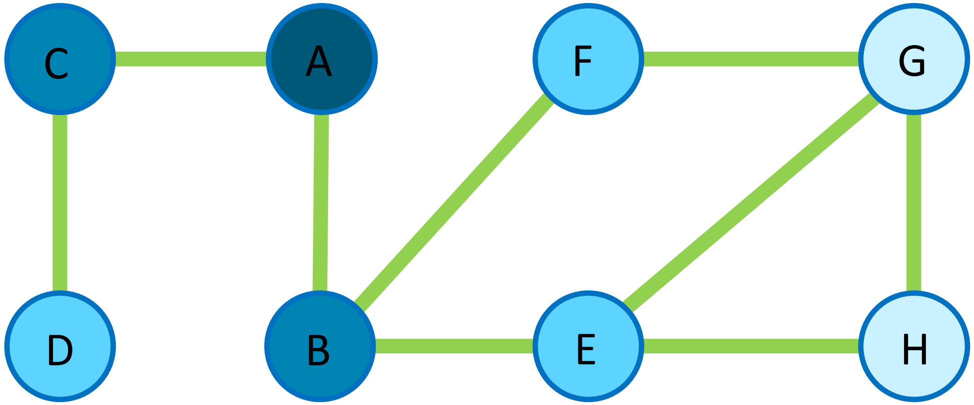


Graph Traversal - BFS

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

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

Yes!

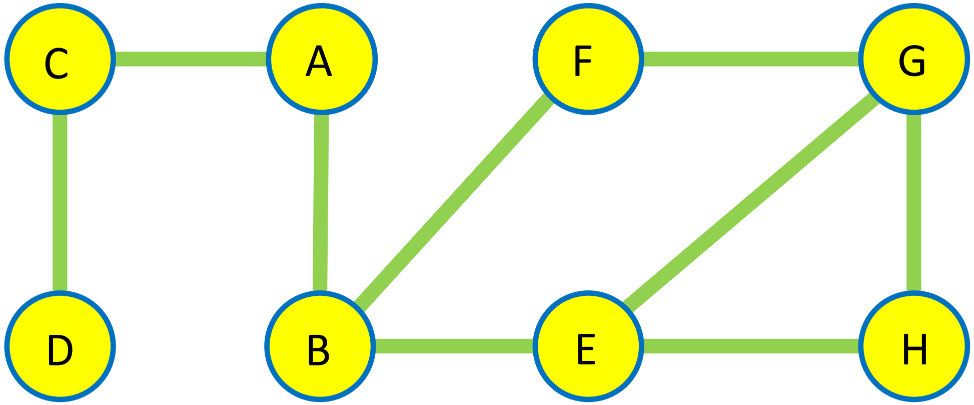


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D a possible DFS order?

Quiz time!
https://flux.qa - YTJMAZ

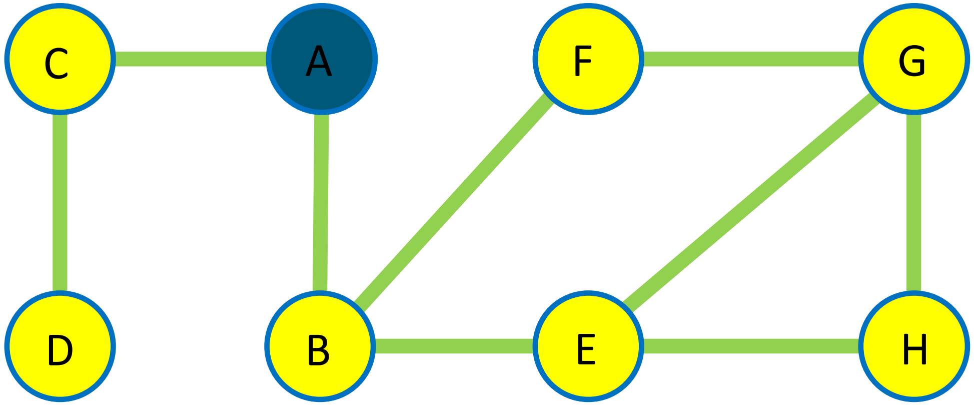


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

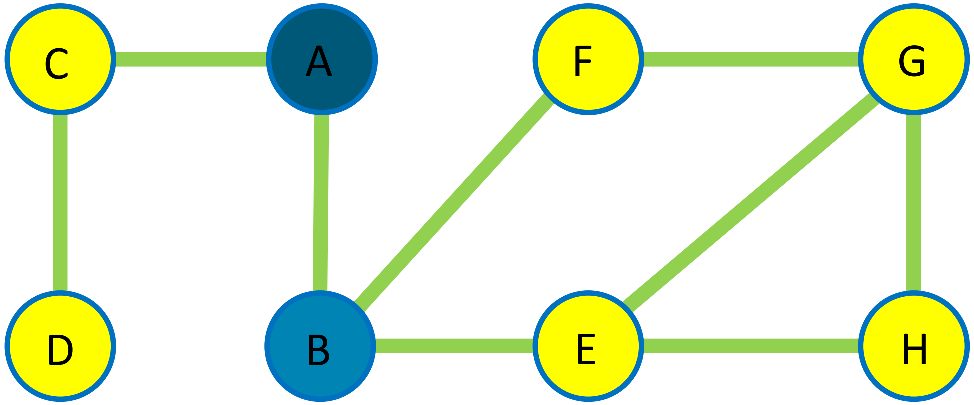
No!



Graph Traversal - DFS

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

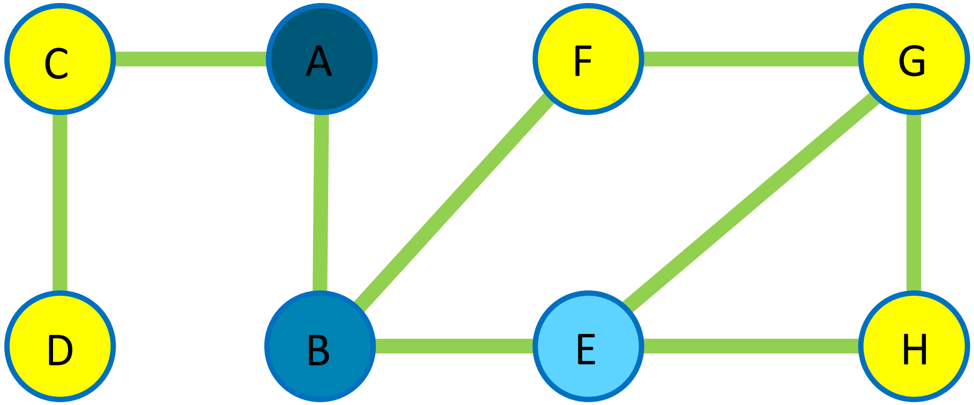
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



Graph Traversal - DFS

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

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

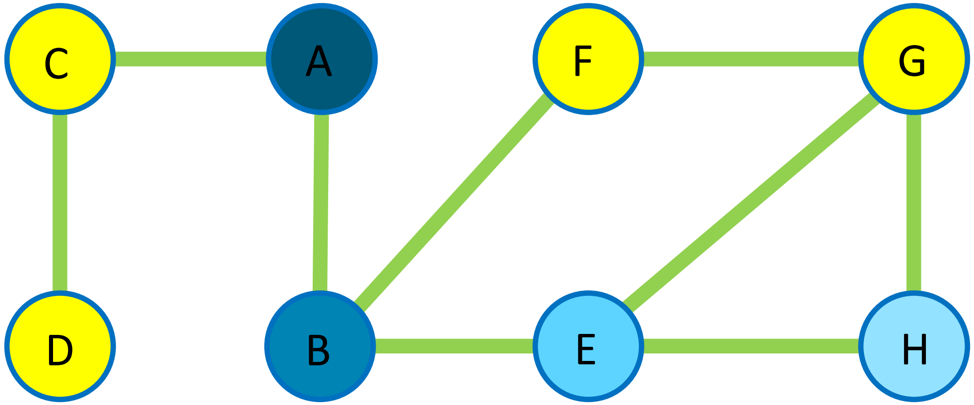


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

No!

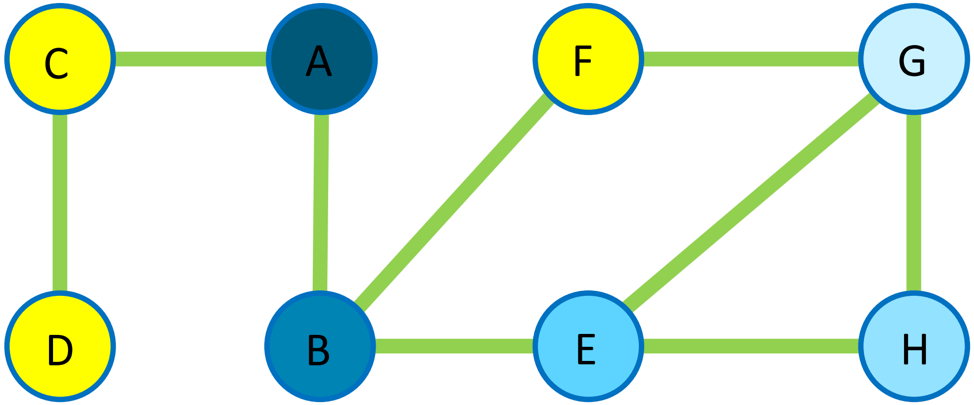


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

No!

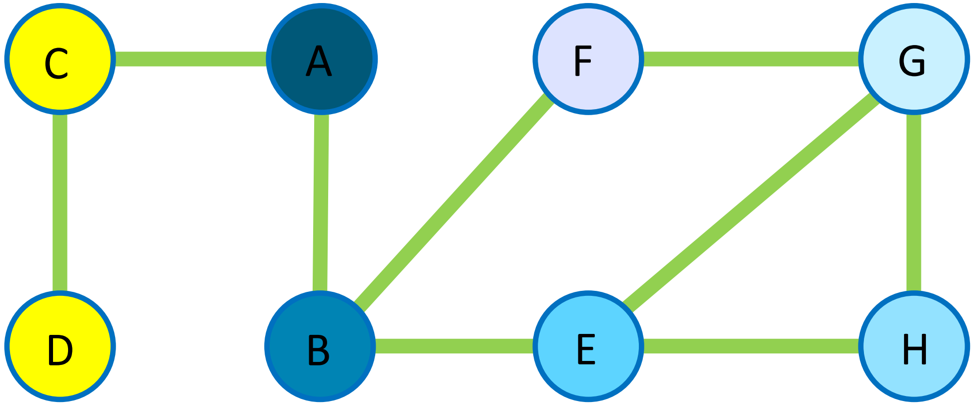


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

No!

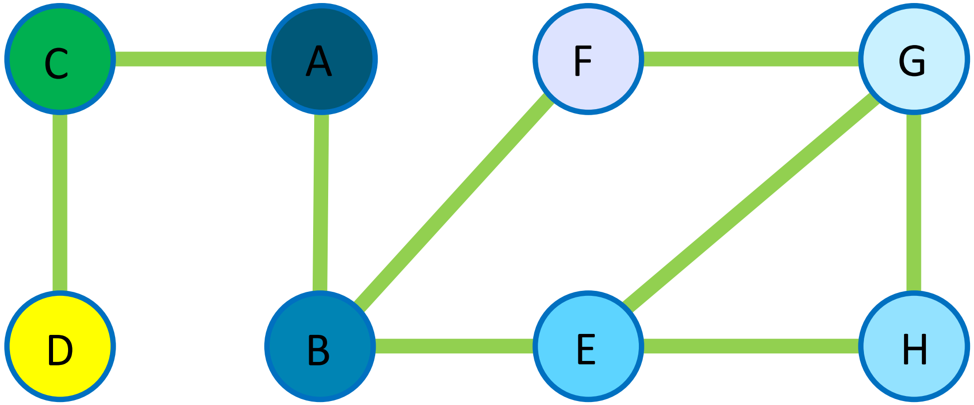


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

No!

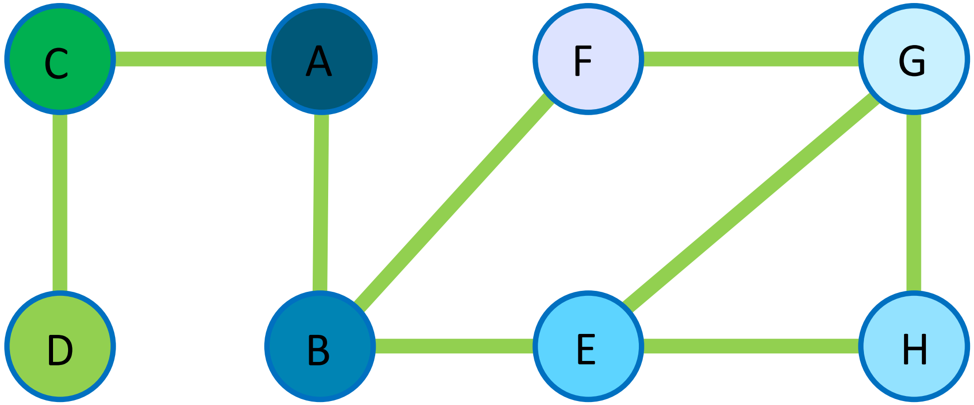


Graph Traversal - DFS

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

Is A, B, E, H, F, G, C, D as possible DFS order?

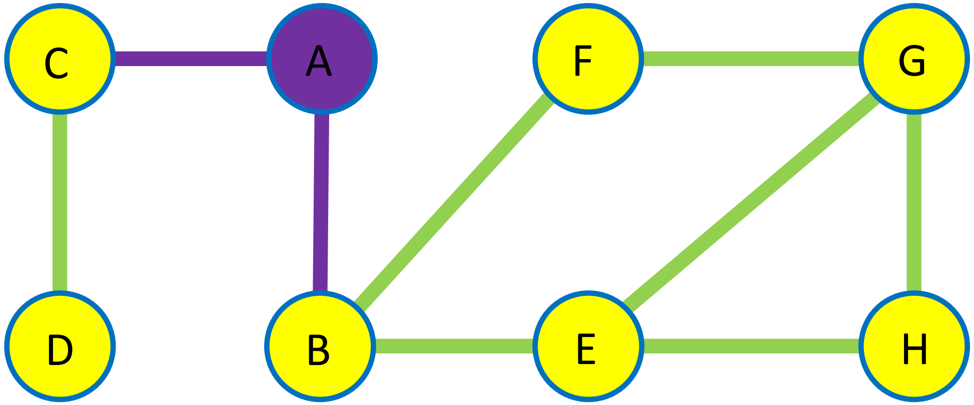
No!



Outline

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

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

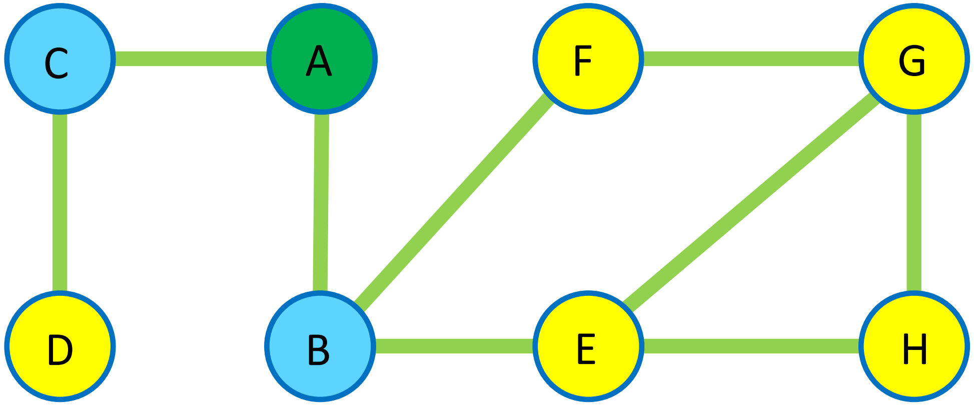
Current: 🟣

Current: A

Queue:

Finished:

Breadth-First Search (BFS)



In Queue: ●

Finished: ●

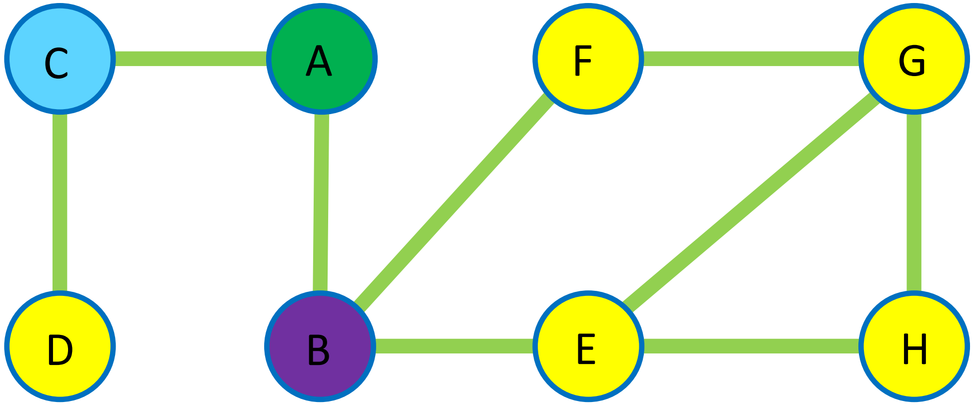
Current: ●

Current:

Queue: B | C

Finished: A

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

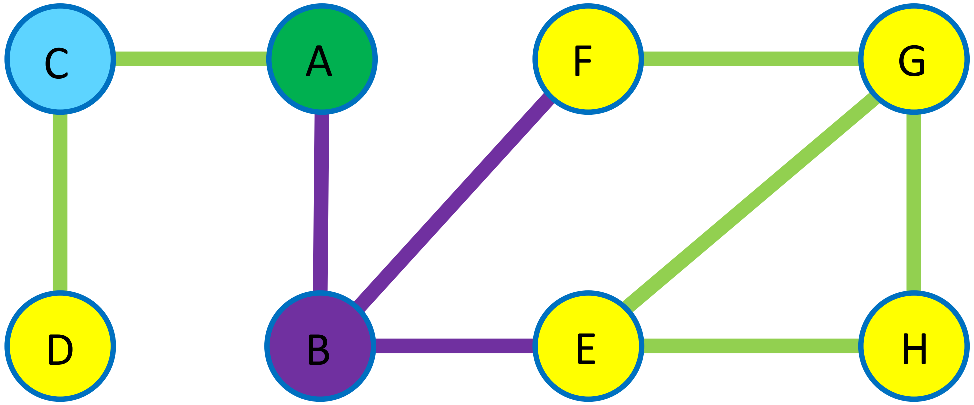
Current: 🟣

Current: B

Queue: C

Finished: A

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

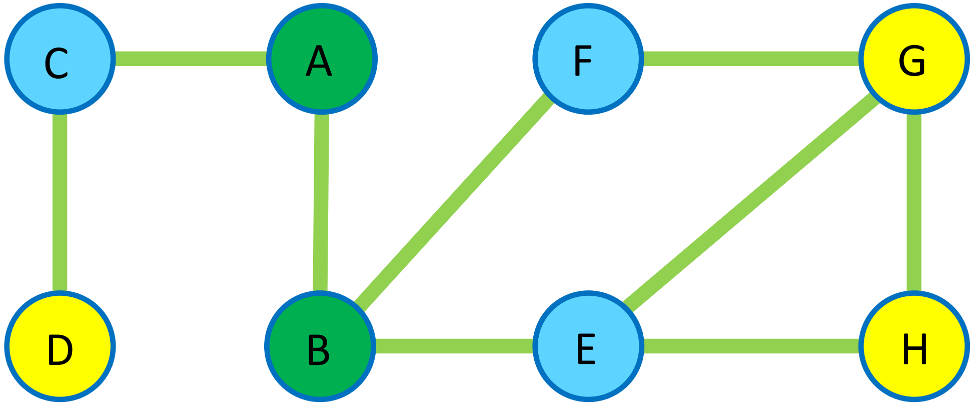
Current: 🟣

Current: B

Queue: C

Finished: A

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

Current: 🟣

Current:

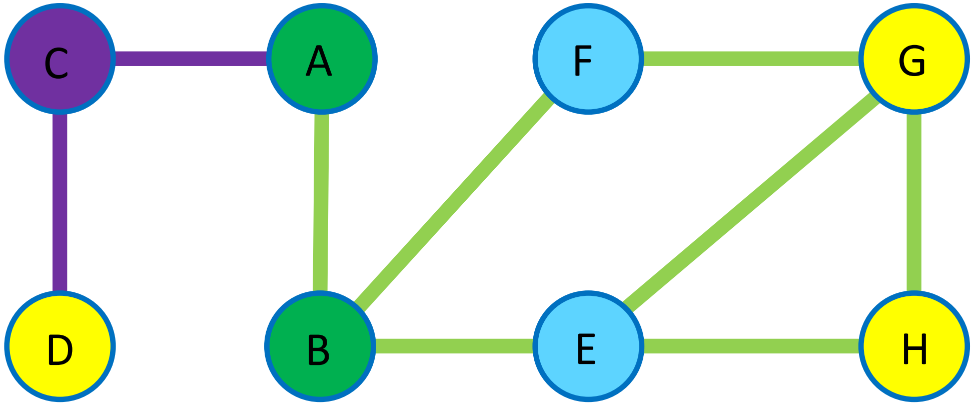
Queue:

C	E	F

Finished:

A	B

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

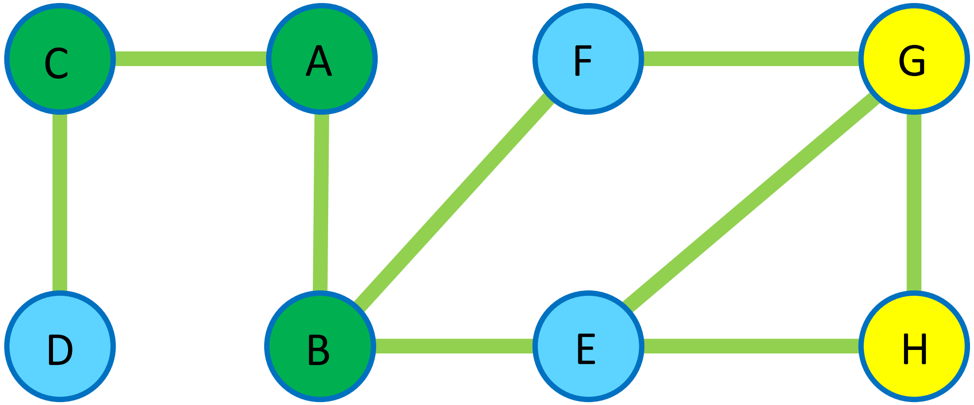
Current: 🟣

Current: C

Queue: E F

Finished: A B

Breadth-First Search (BFS)



In Queue: 🔵

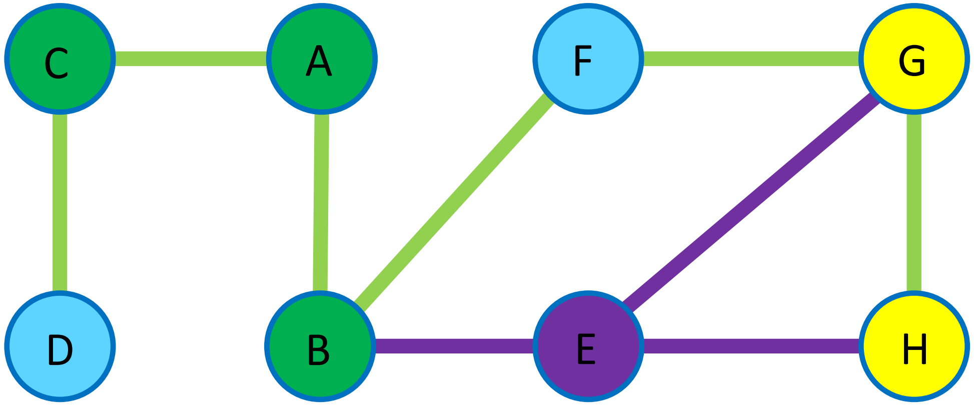
Finished: 🟢

Current: 🟣

Current:

Queue:			
Queue:	E	F	D
Finished:	A	B	C

Breadth-First Search (BFS)



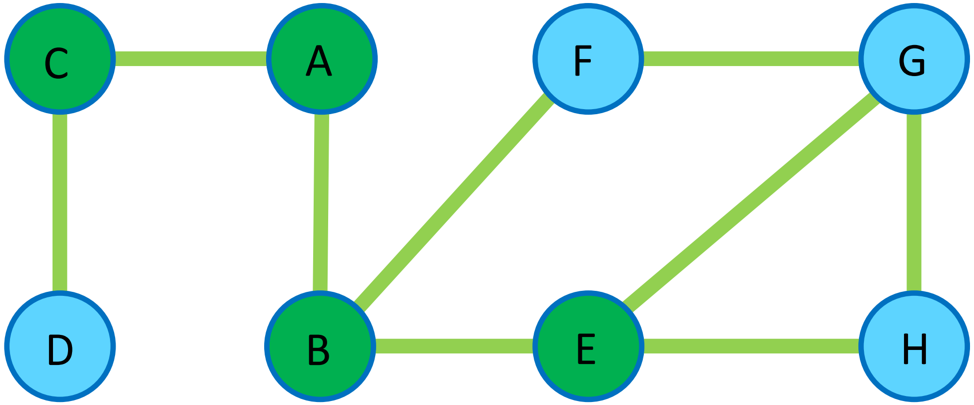
In Queue: 🔵

Finished: 🟢

Current: 🟣

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

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

Current: 🟣

Current:

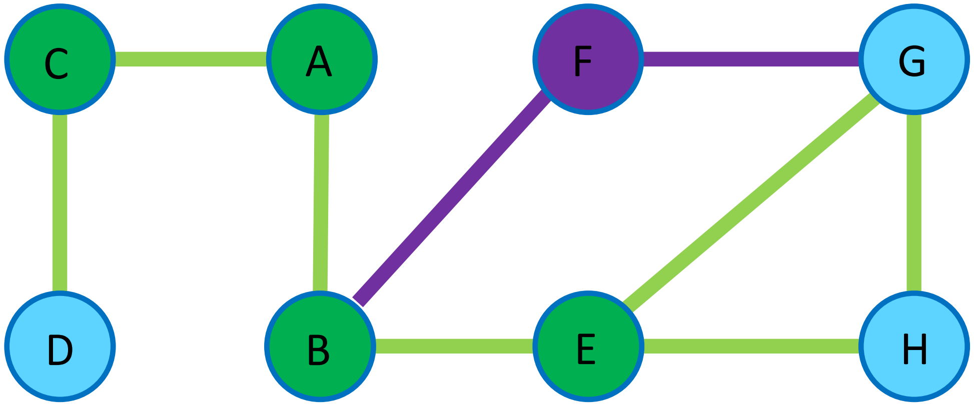
Queue:

F	D	G	H

Finished:

A	B	C	E

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

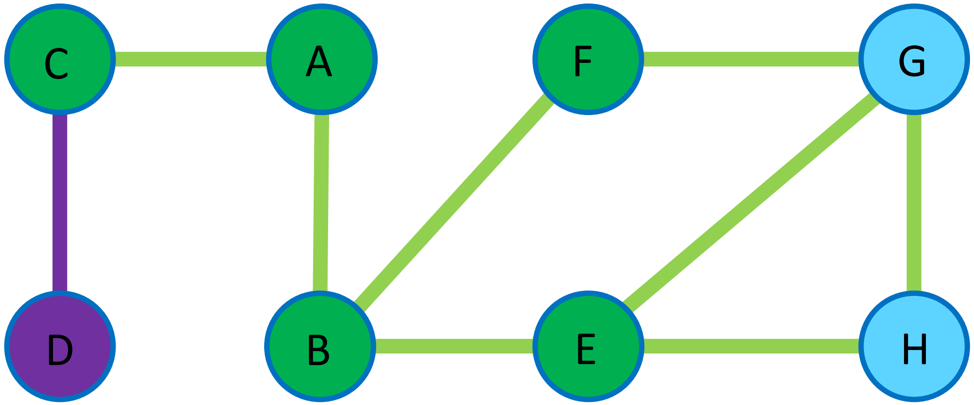
Current: 🟣

Current: F

Queue: D G H

Finished: A B C E

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

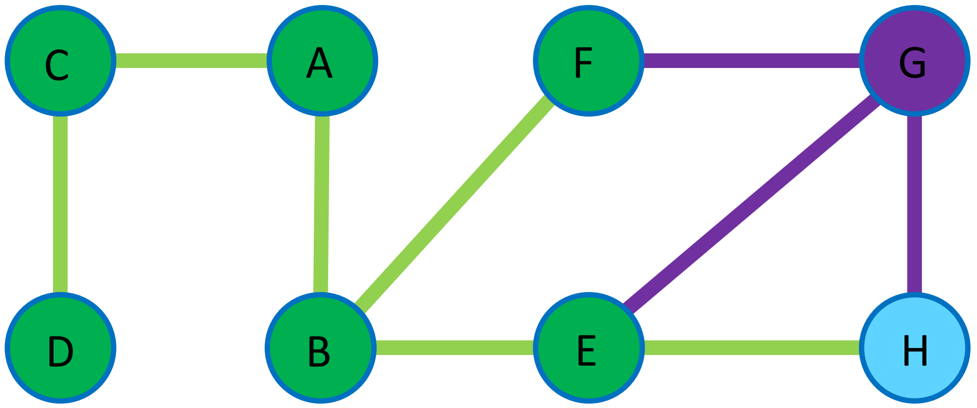
Current: 🟣

Current: D

Queue: G | H

Finished: A | B | C | E | F

Breadth-First Search (BFS)



In Queue: ●

Finished: ●

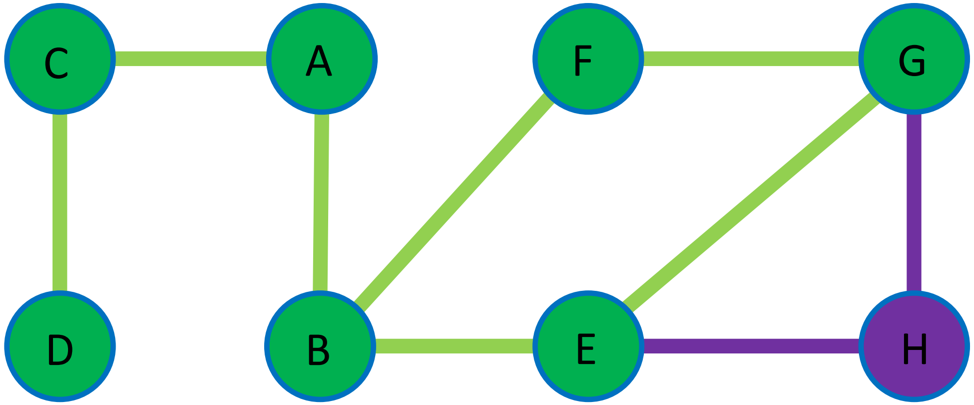
Current: ●

Current: G

Queue: H

Finished: A | B | C | E | F | D

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

Current: 🟣

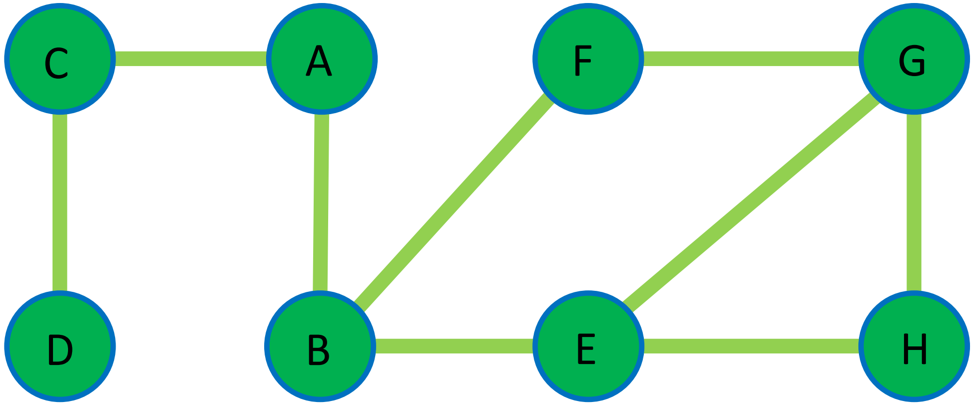
Current: H

Queue:

Finished:

A	B	C	E	F	D	G

Breadth-First Search (BFS)



In Queue: 🔵

Finished: 🟢

Current: 🟣

Current:

Queue:

Finished:

A	B	C	E	F	D	G	H

Breadth-First Search (BFS)

- Initialize some data structure “**queue**” and some data structure “**visited**”, both empty of vertices
- Put an initial vertex in **queue**
- Mark the initial vertex as visited
- While **queue** is not empty
  - Get the first vertex, **u**, from **queue**
  - For each edge (**u**,**v**)
    - If **v** is not **visited**
      - Add **v** to visited
      - add **v** at the end of **queue**

Quiz time!
https://flux.qa - YTJMAZ

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices → $O(?)$
- Put an initial vertex in **queue** → $O(1)$
- Mark the initial vertex as visited → $O(?)$
- While **queue** is not empty → $O(1)$
    - Get the first vertex, **u**, from **queue** → $O(V) total$
    - For each edge (**u**,**v**) → $O(E)\ total$
        - If **v** is not **visited** → $O(?)$
            - Add **v** to visited → $O(?)$
            - Add **v** at the end of **queue** → $O(V) total$

Assuming adjacency list representation.

Time Complexity:

- We look at every edge twice
- For each edge, we do a lookup on visited (with some complexity)
- We insert vertices to visited at most O(V) times
- O(V*insert to visited + E*lookup on visited)

Breadth-First Search (BFS)

- Initialize some data structure “**queue**” and some data structure “**visited**”, both empty of vertices → $O(?)$
- Put an initial vertex in **queue** → $O(1)$
- Mark the initial vertex as visited → $O(?)$
- While **queue** is not empty → $O(1)$
    - Get the first vertex, **u**, from **queue** → $O(V) total$
    - For each edge (**u**,**v**) → $O(E)$ $total$
        - If **v** is not **visited** → $O(?)$
            - Add **v** to visited → $O(?)$
            - Add **v** at the end of **queue** → $O(V) total$

Assuming adjacency list representation.

Time Complexity:

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

Breadth-First Search (BFS)

- Initialize some data structure "**queue**" and some data structure "**visited**", both empty of vertices
- Put an initial vertex in **queue**
- Mark the initial vertex as visited
- While **queue** is not empty
  - Get the first vertex, **u**, from **queue**
  - For each edge (**u**,**v**)
    - If **v** is not **visited**
      - Add **v** to visited
      - add **v** at the end of **queue**

Assuming adjacency list representation.

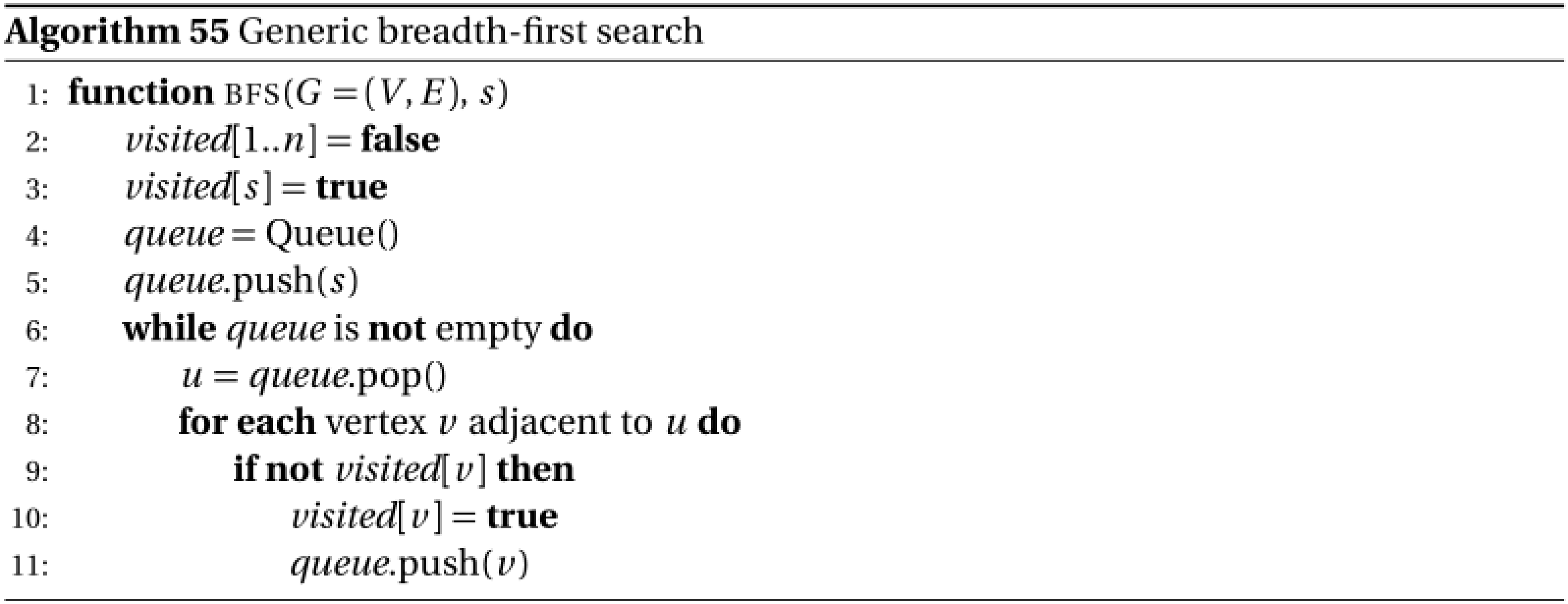
Time Complexity:

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

Space Complexity:

- O(V+E)

Breadth-First Search (BFS)



Algorithm 55 Generic breadth-first search

```
function BFS($G = (V, E)$, $s$)
    $visited[1..n]$ = false
    $visited[s]$ = true
    $queue$ = Queue()
    $queue$.push($s$)
    while $queue$ is not empty do
        $u$ = $queue$.pop()
        for each vertex $v$ adjacent to $u$ do
            if not $visited[v]$ then
                $visited[v]$ = true
                $queue$.push($v$)
```

Assuming adjacency list representation.

Time Complexity:

- O(V+E)

Space Complexity:

- O(V+E)

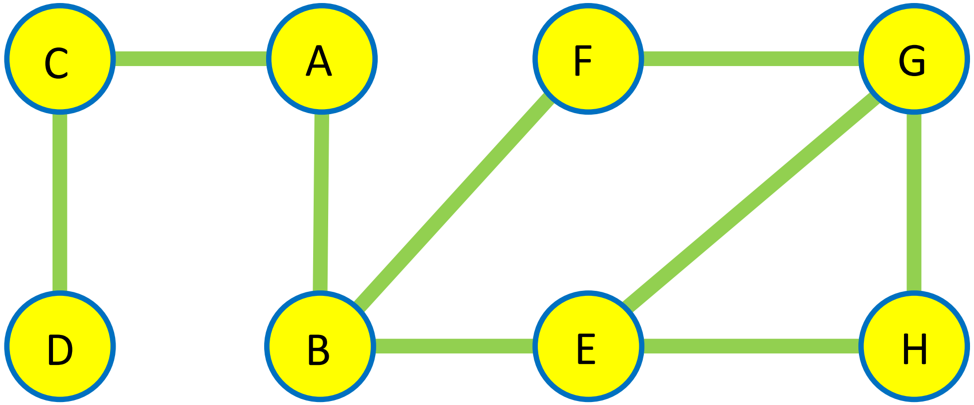
Outline

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

Example with good data structures

- Now that you have the idea of BFS
- And we have seen how to use a bit list to make it faster
- We will do the DFS example with a bit list

Depth-First Search (DFS)



Visited: 🟢

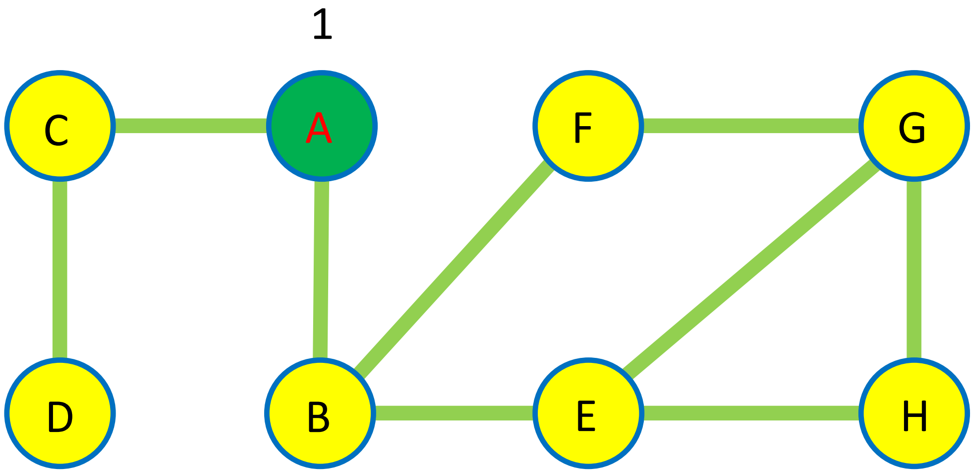
Current: A

Visited is indexed by vertex ID. Normally, the IDs are integers from from 0 to V-1 (to allow O(1) lookup), but letters are used here for ease of understanding

Visited:

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

Depth-First Search (DFS)

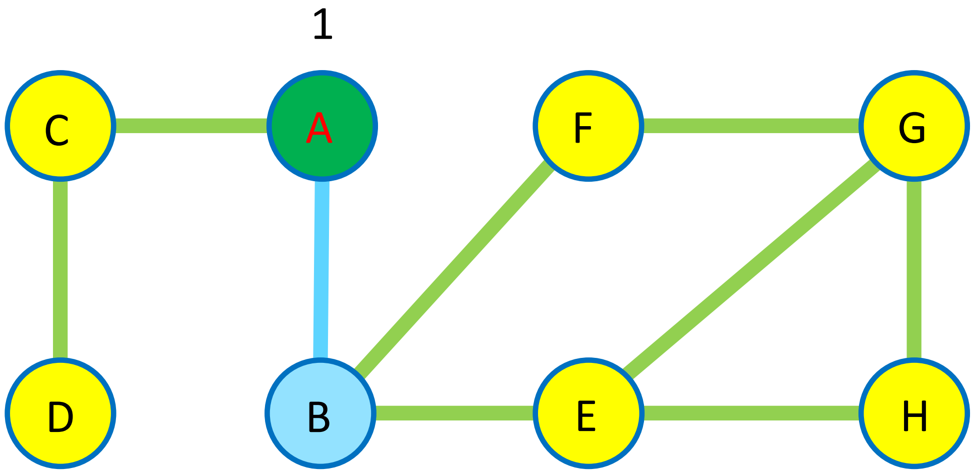


Visited: ●

Current: **A**

Visited:	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
	1	0	0	0	0	0	0	0

Depth-First Search (DFS)

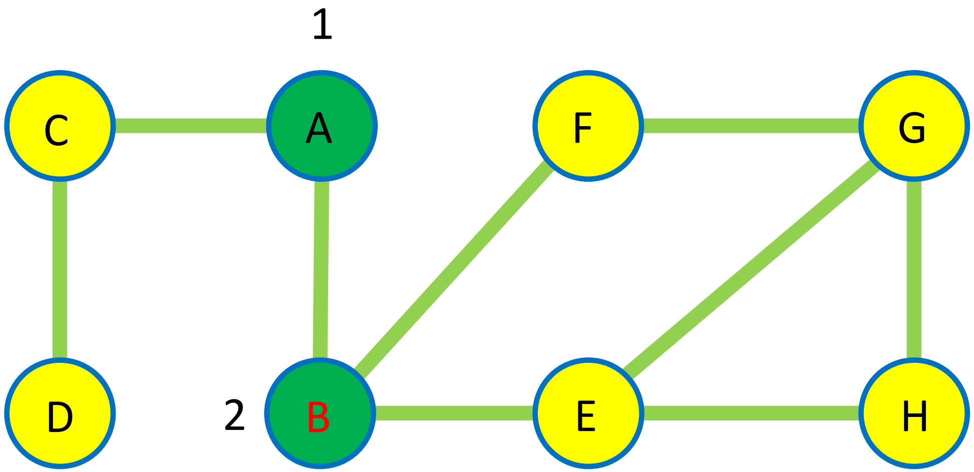


Visited: ●

Current: A

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

Depth-First Search (DFS)



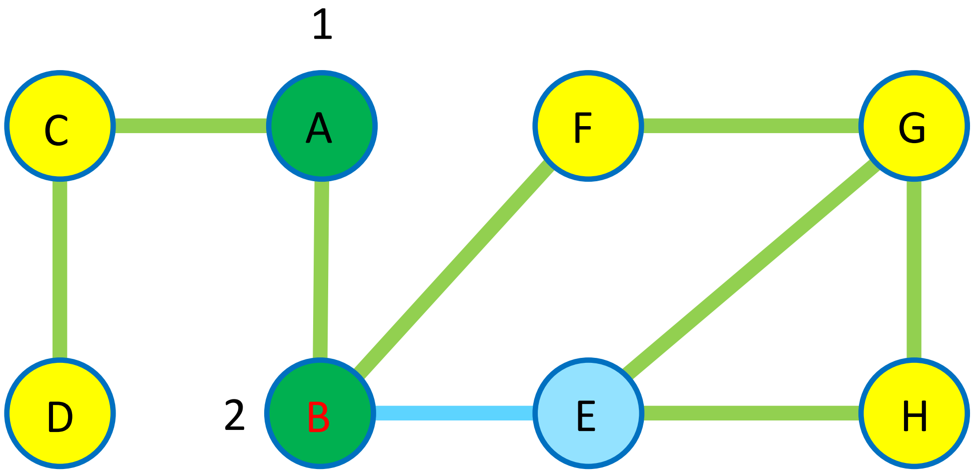
Visited: 🟩

Current: **B**

Visited:

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

Depth-First Search (DFS)

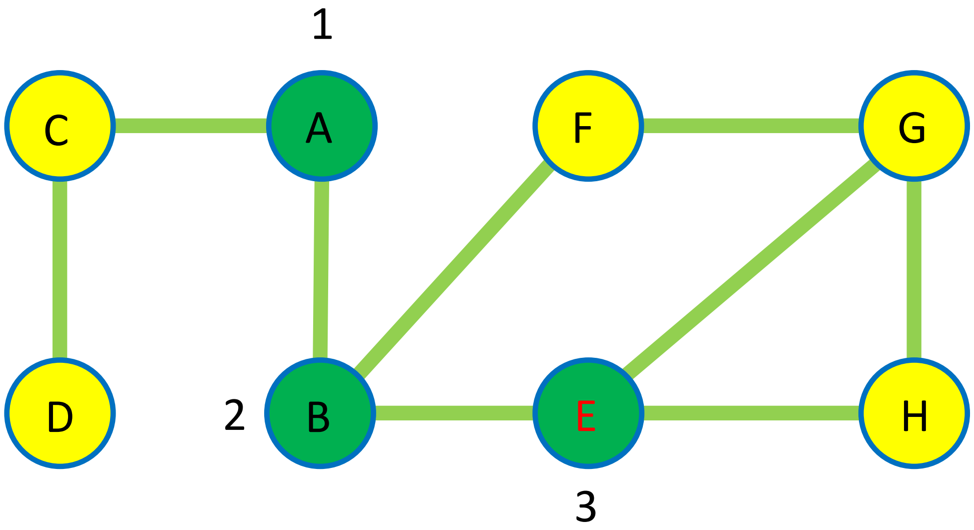


Current: **B**

Visited:

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

Depth-First Search (DFS)



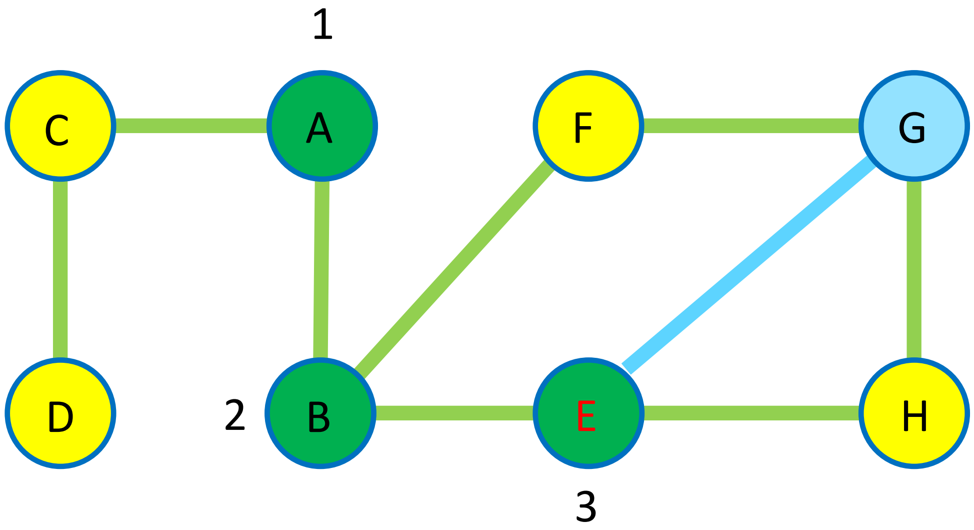
Visited: ●

Current: **E**

Visited:

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

Depth-First Search (DFS)



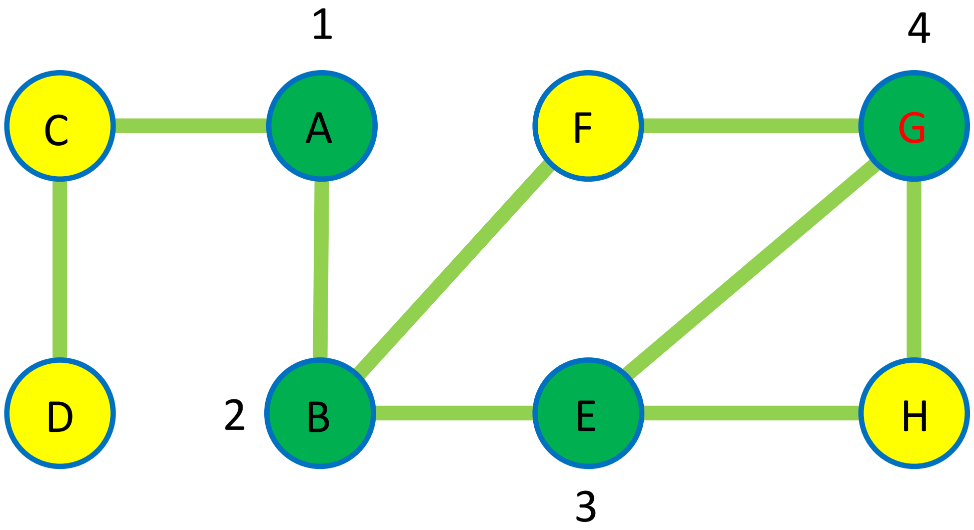
Visited: ●

Current: E

Visited:

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

Depth-First Search (DFS)



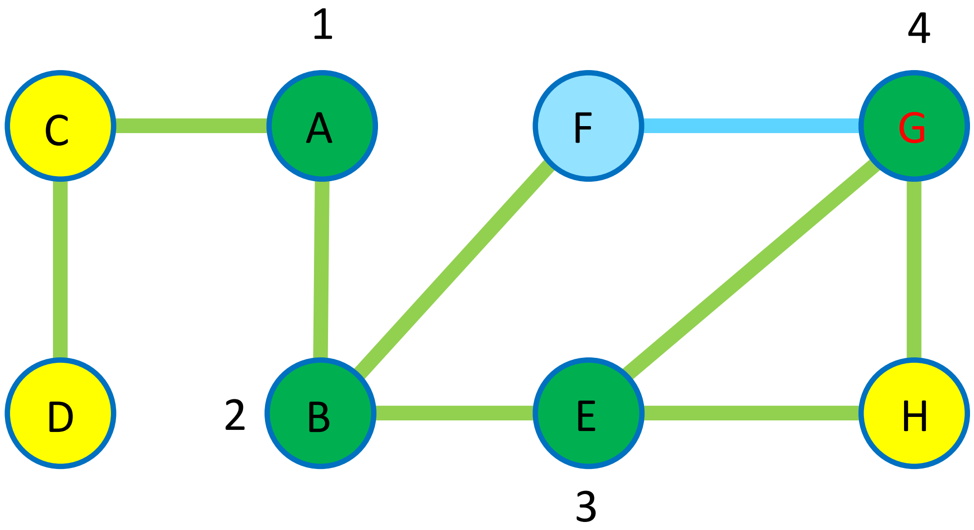
Visited: ●

Current: **G**

Visited:

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

Depth-First Search (DFS)



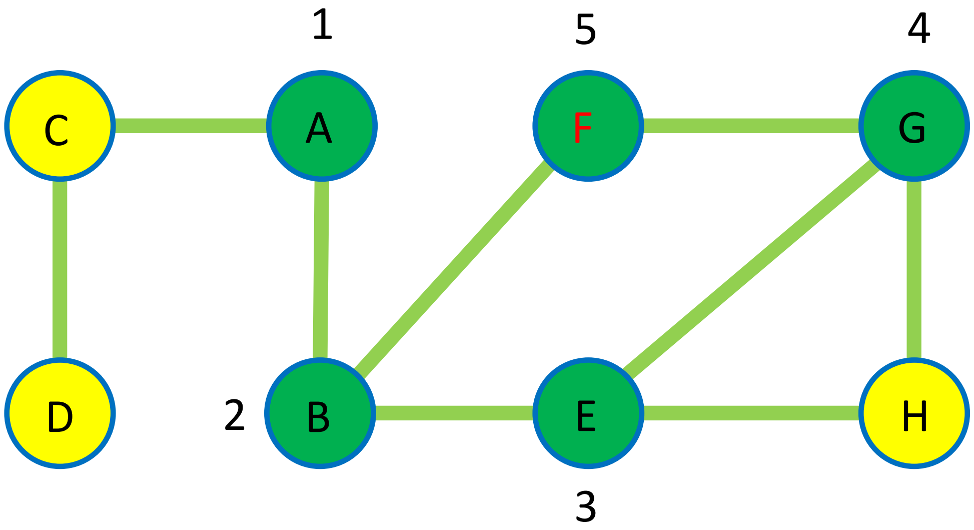
Visited: ●

Current: **G**

Visited:

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

Depth-First Search (DFS)



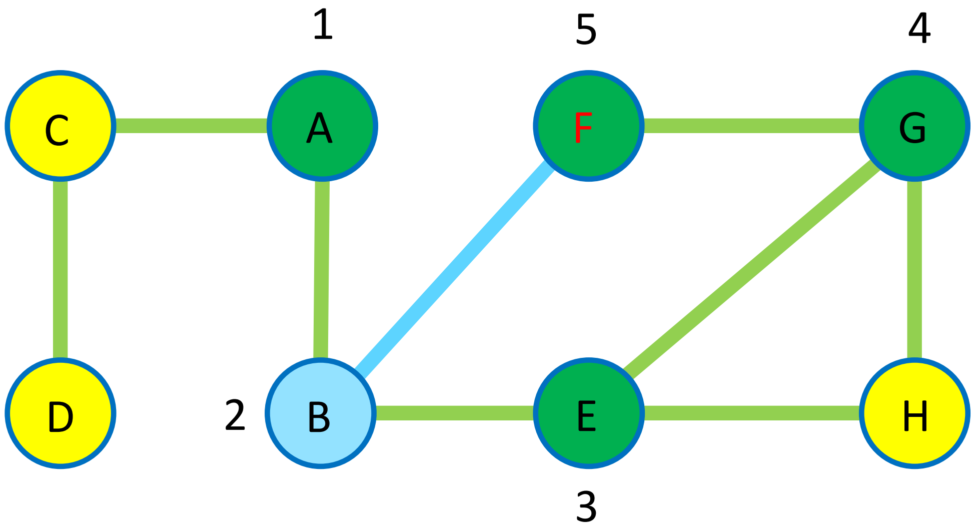
Visited: ●

Current: **F**

Visited:

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

Depth-First Search (DFS)



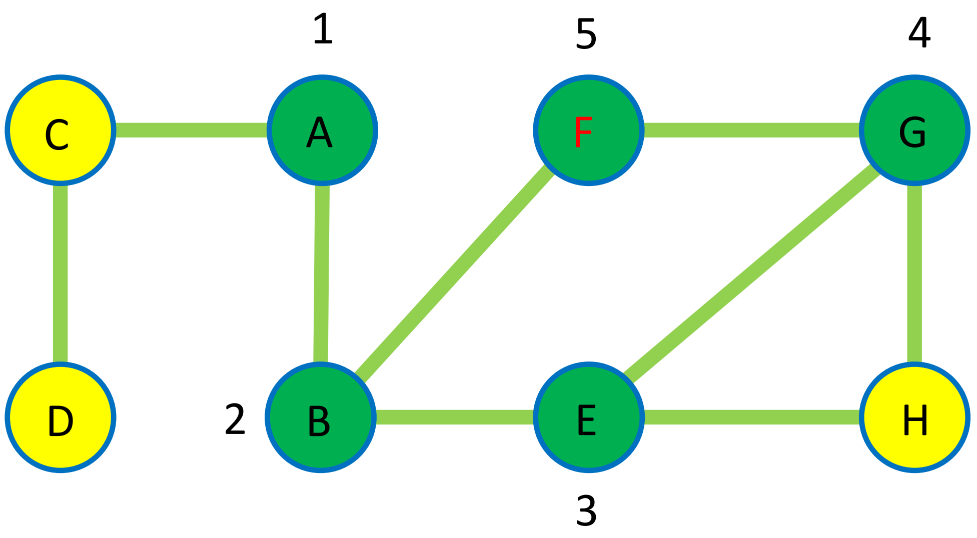
Visited: 🟢

Current: **F**

Visited:

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

Depth-First Search (DFS)



Visited: ●

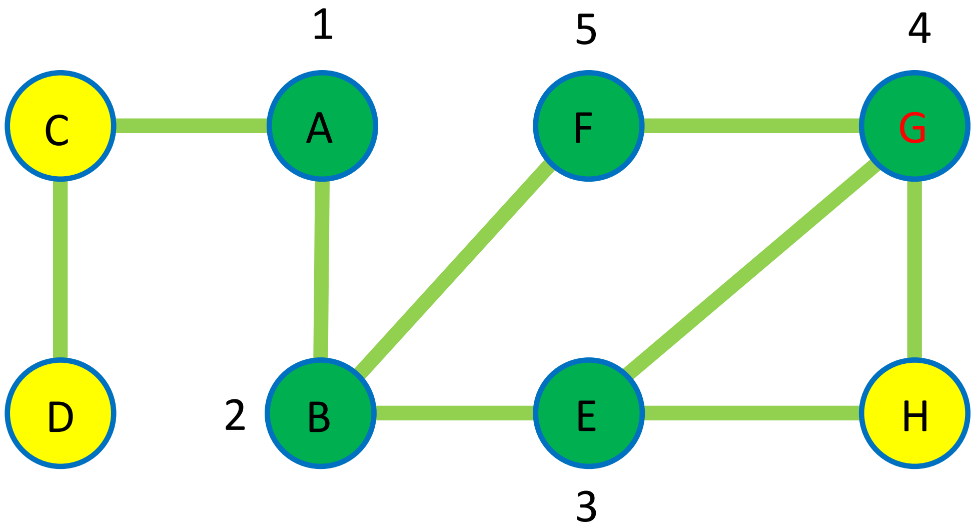
- F is a dead end
- Go back to the last active node (G)

Current: **F**

Visited:

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

Depth-First Search (DFS)



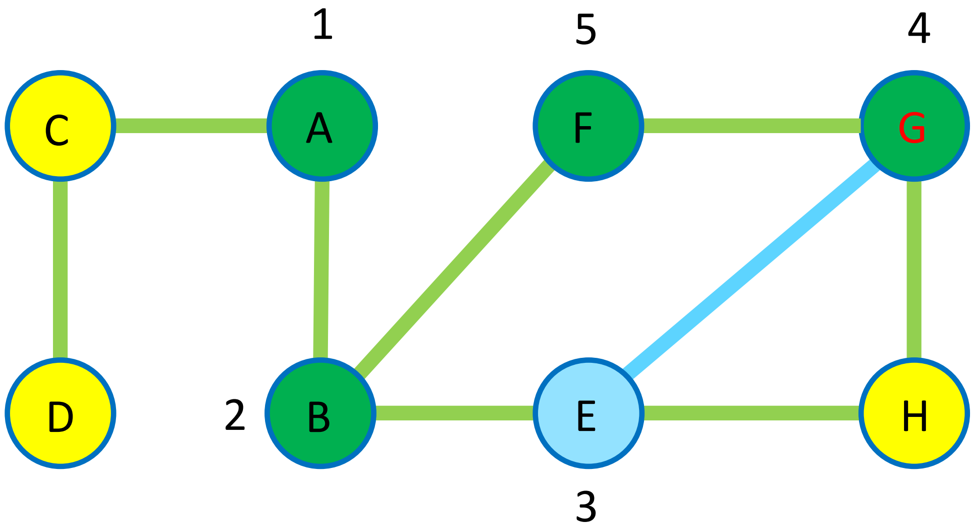
Visited: ●

Current: G

Visited:

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

Depth-First Search (DFS)



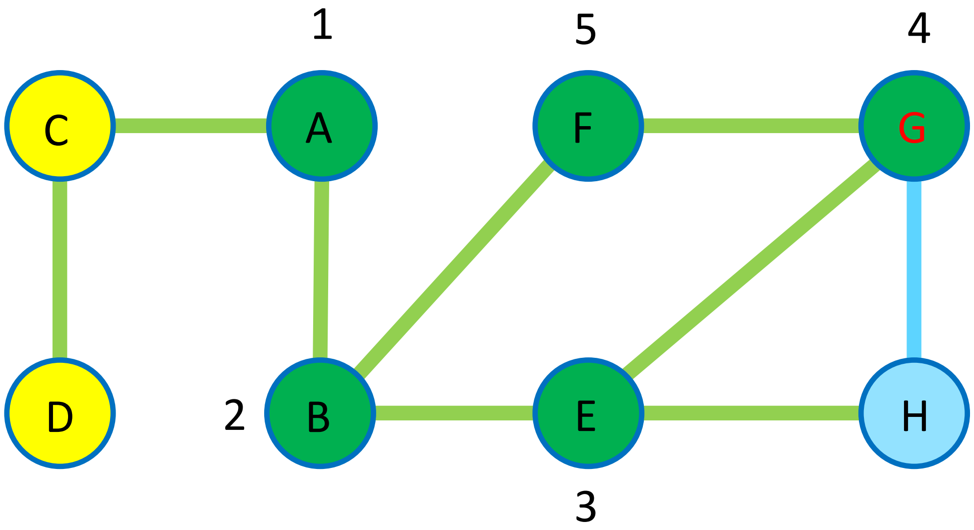
Visited: ●

Current: G

Visited:

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

Depth-First Search (DFS)



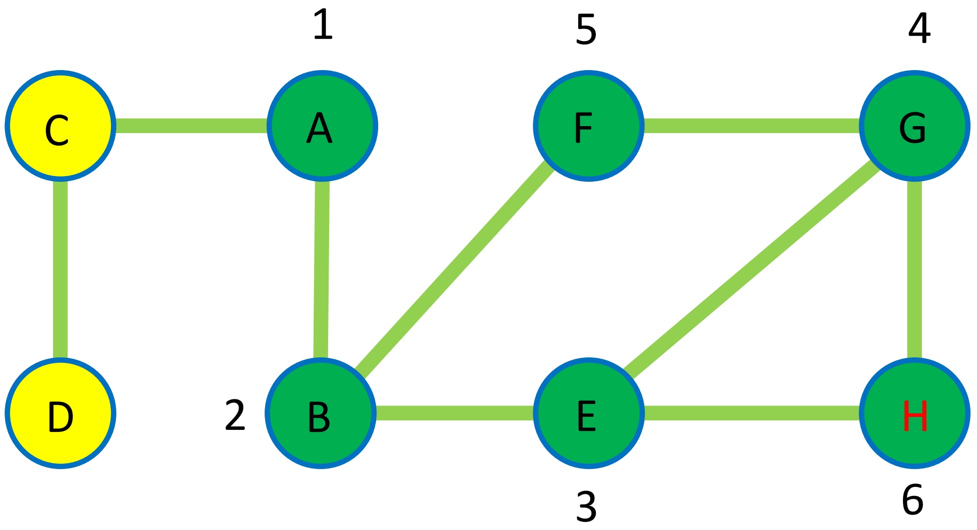
Visited: ●

Current: **G**

Visited:

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

Depth-First Search (DFS)



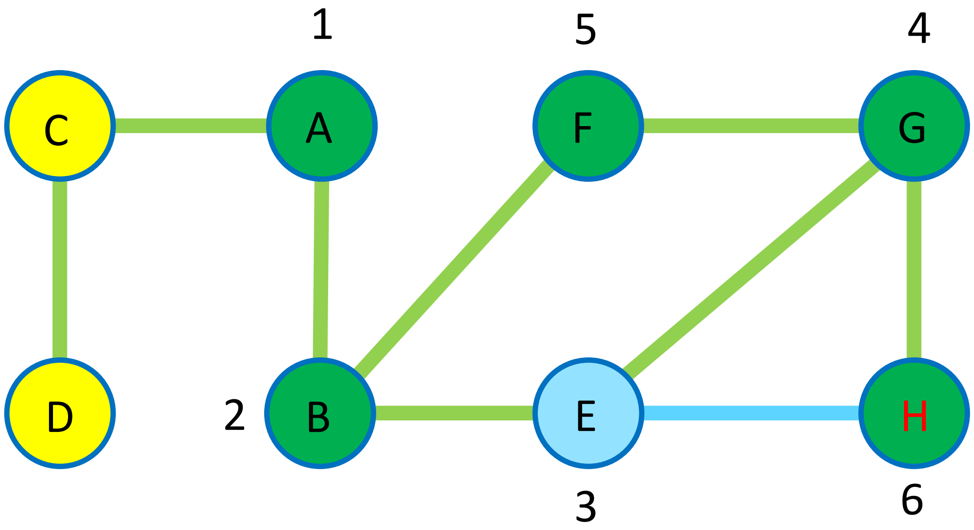
Visited: ●

Current: **H**

Visited:

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

Depth-First Search (DFS)



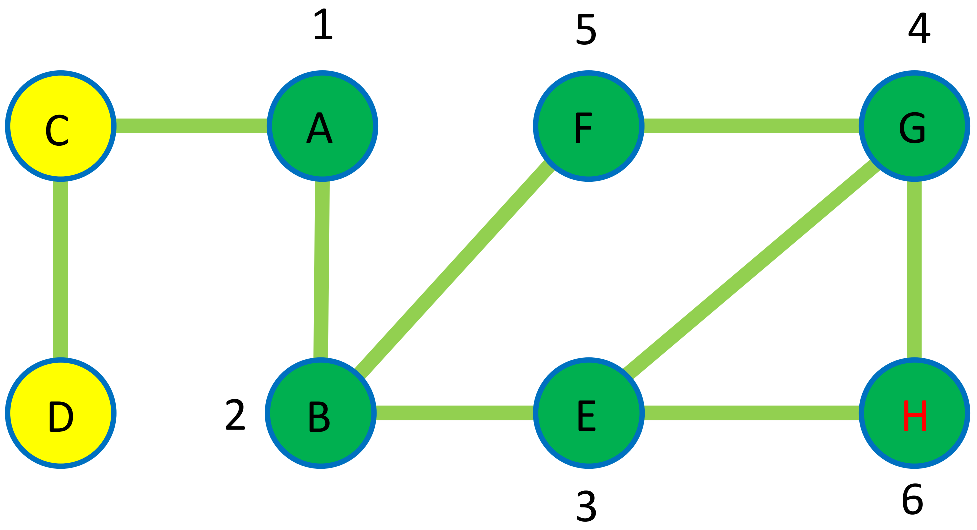
Visited: 🟢

Current: **H**

Visited:

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

Depth-First Search (DFS)



Visited: 🟢

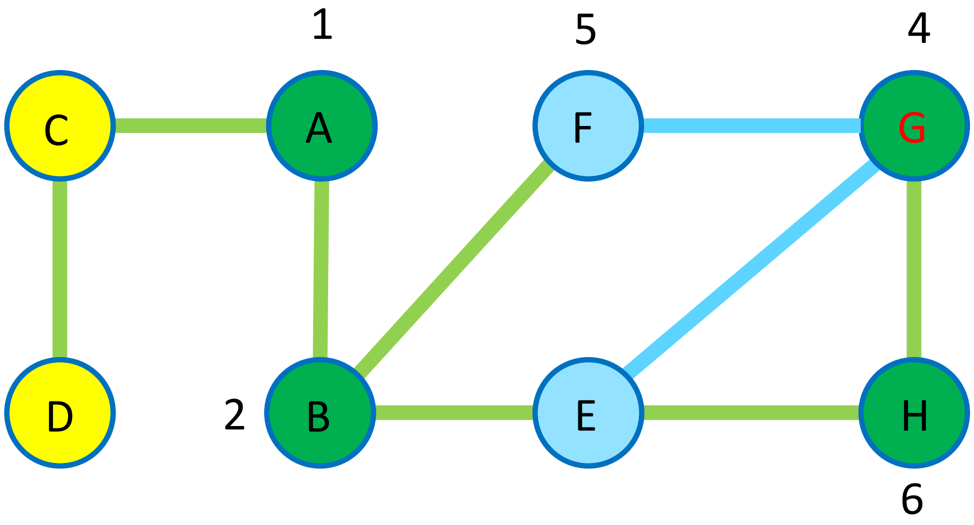
- H is a dead end
- Speeding up visualisation…

Current: H

Visited:

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

Depth-First Search (DFS)



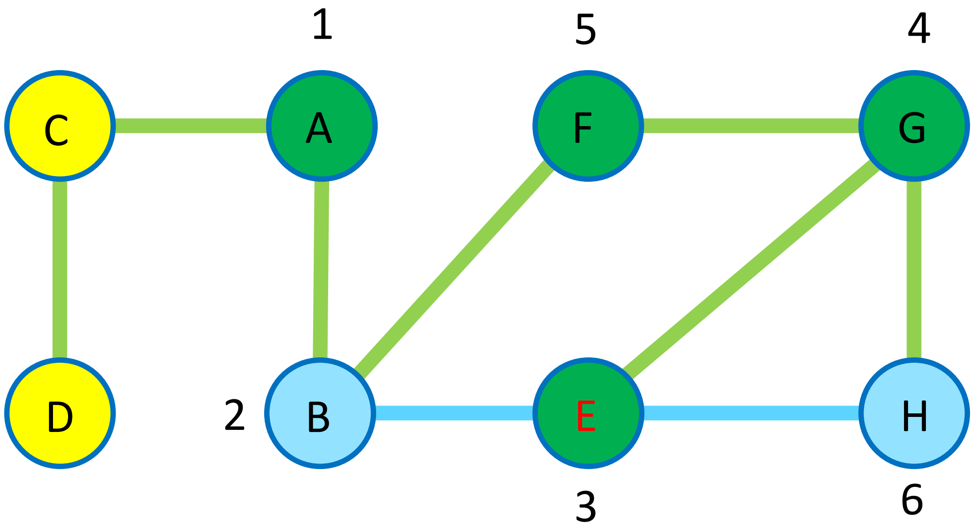
Visited: ●

Current: G

Visited:

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

Depth-First Search (DFS)



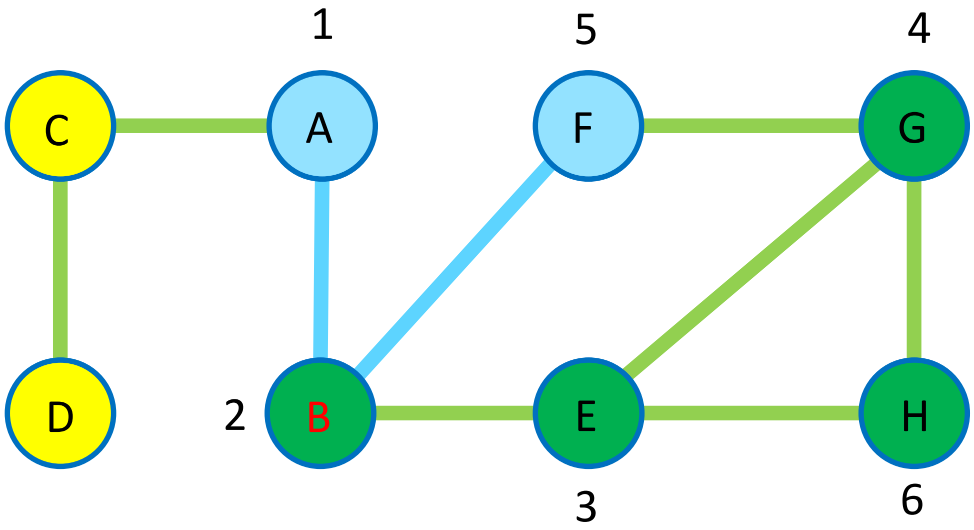
Visited: ●

Current: E

Visited:

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

Depth-First Search (DFS)



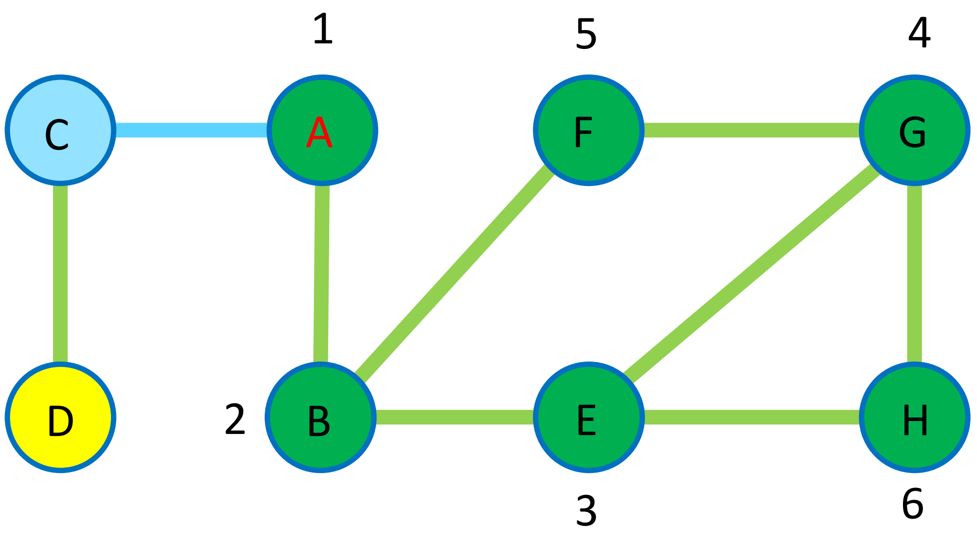
Visited: ●

Current: **B**

Visited:

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

Depth-First Search (DFS)



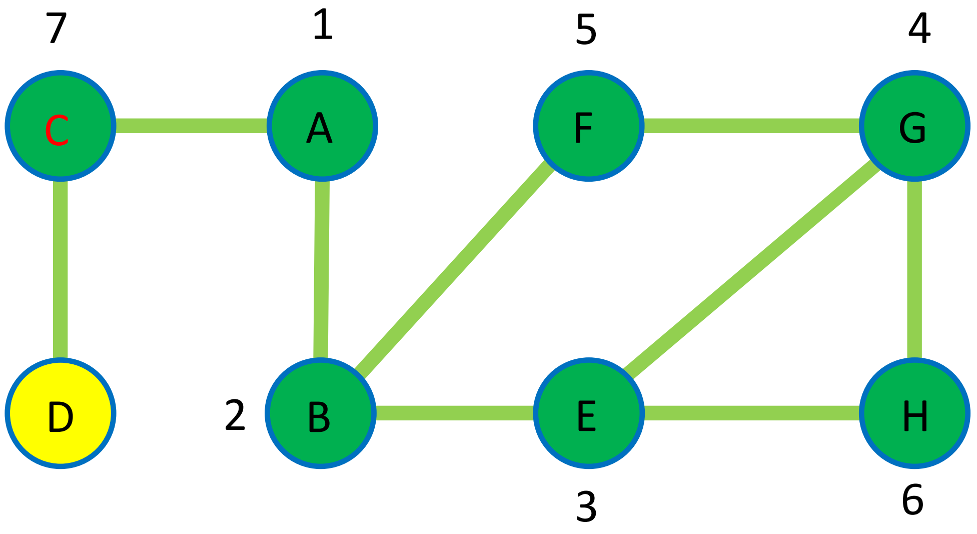
Visited: ●

Current: **A**

Visited:

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

Depth-First Search (DFS)



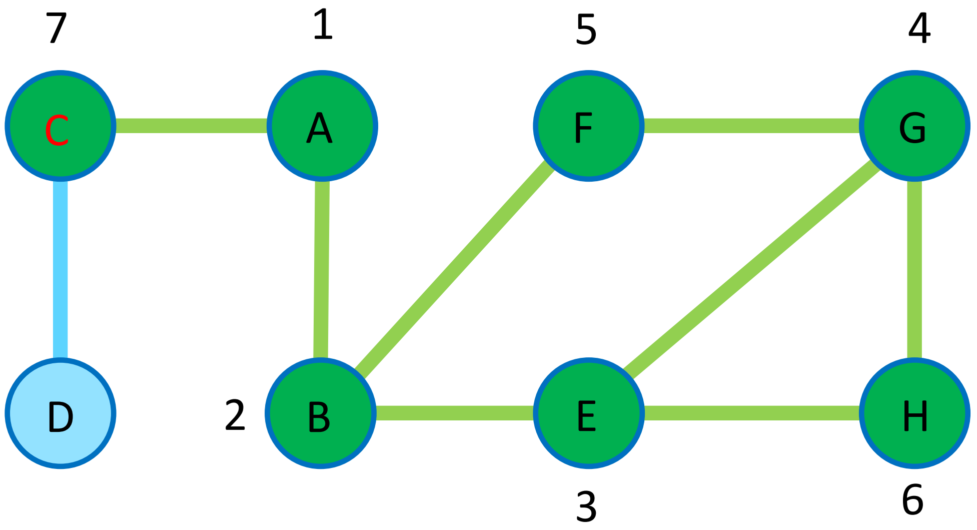
Visited: ●

Current: **C**

Visited:

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

Depth-First Search (DFS)



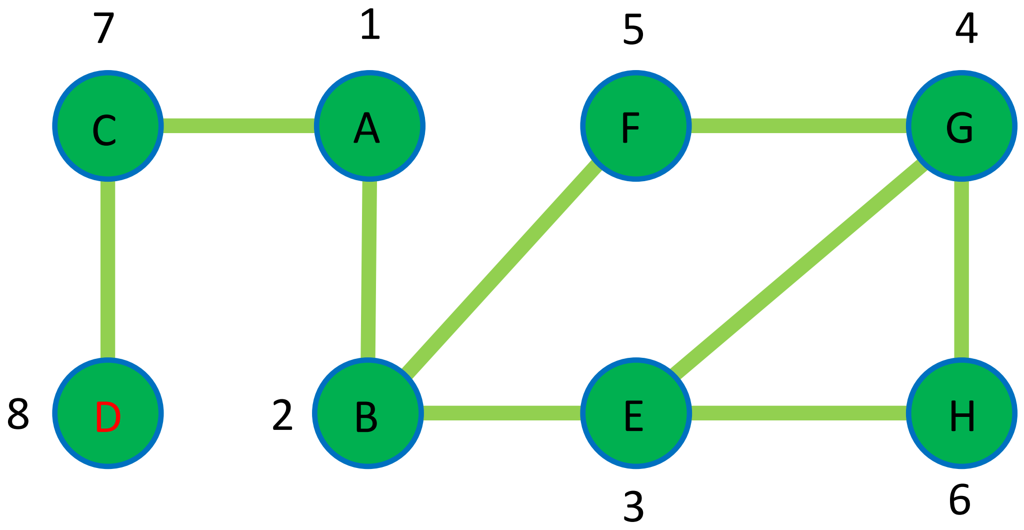
Visited: ●

Current: C

Visited:

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

Depth-First Search (DFS)



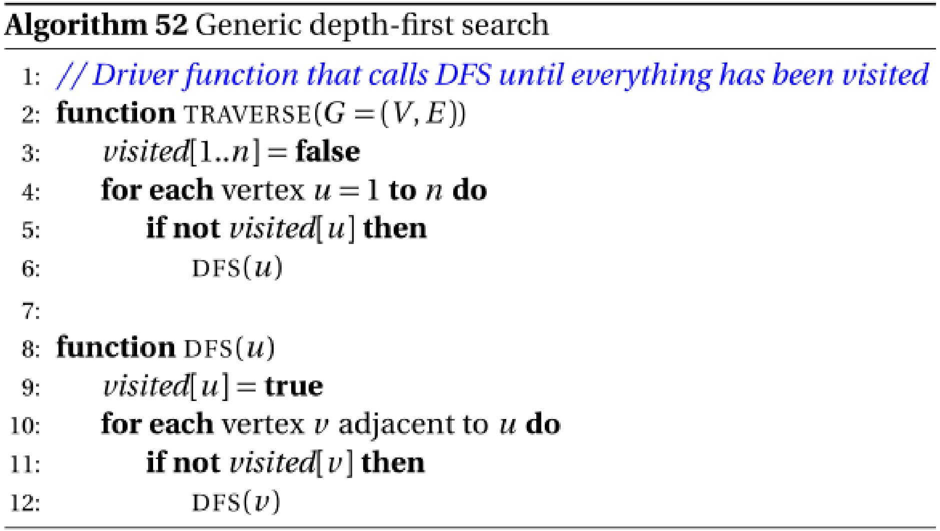
Visited: ●

Current: **D**

Visited:

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

Depth-First Search (DFS)



Algorithm 52 Generic depth-first search

```
// Driver function that calls DFS until everything has been visited
function TRAVERSE($G = (V, E)$)
    $visited[1..n]$ = false
    for each vertex $u = 1$ to $n$ do
        if not $visited[u]$ then
            DFS($u$)

function DFS($u$)
    $visited[u]$ = true
    for each vertex $v$ adjacent to $u$ do
        if not $visited[v]$ then
            DFS($v$)
```

Assuming adjacency list representation.

Time Complexity:

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

Space Complexity:

- O(V+E)

Outline

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

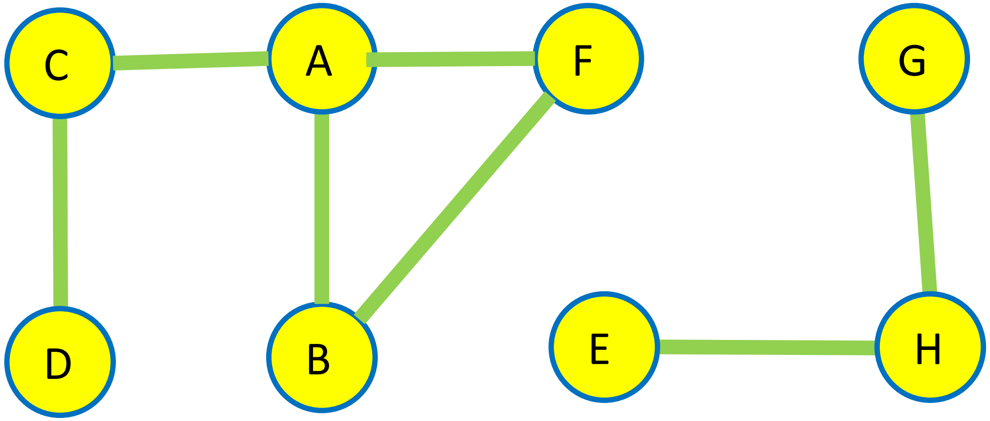
Applications of DFS and BFS

The algorithms we saw can also be applied on directed graphs.

BFS and DFS have a wide variety of applications:

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

More details are given in unit notes and tutorials.



Shortest Path Problem

Length of a path:

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

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

Shortest Path Problem

Single source, single target:

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

Single source, all targets:

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

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

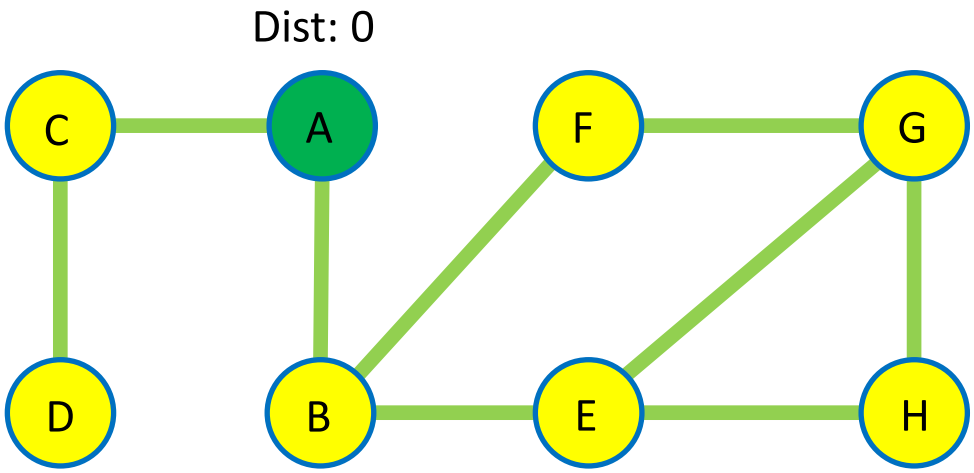
Shortest Path Algorithms

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

Outline

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

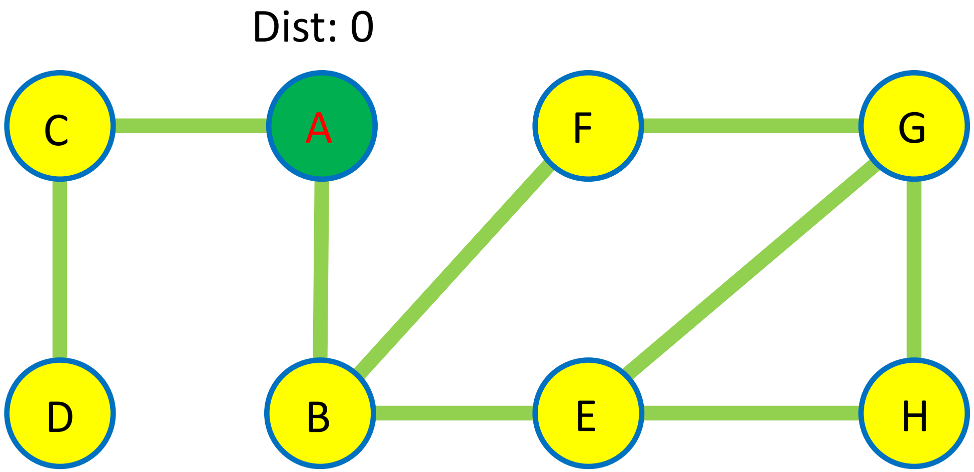
Current:

Queue: A

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🔵

Visited: 🟢

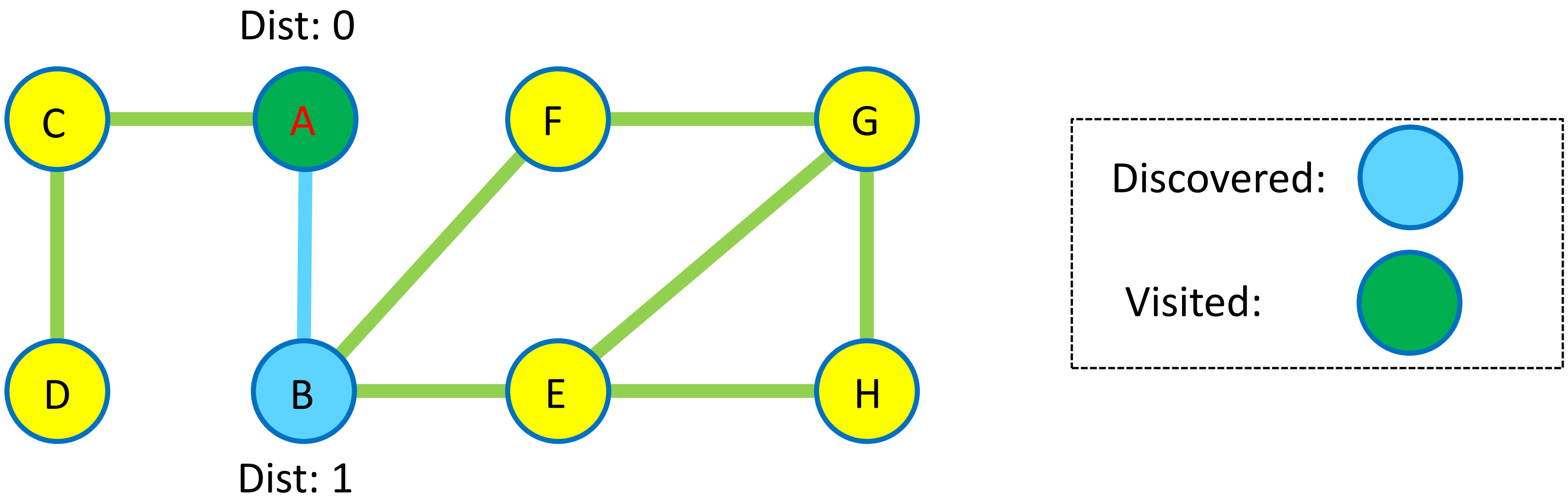
Current: A

Queue:

Visited:

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

BFS shortest path (now with good data structures)



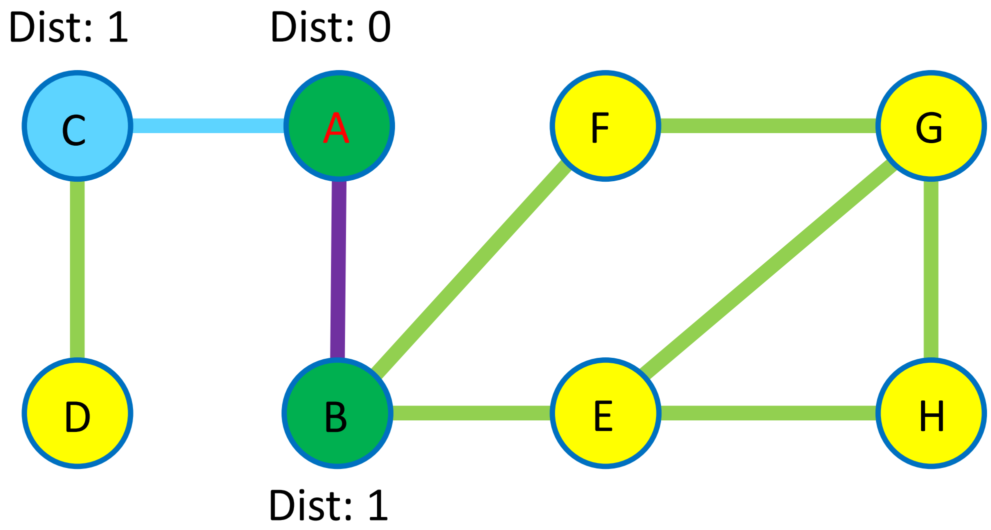
Current: A

Queue:

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

Current: A

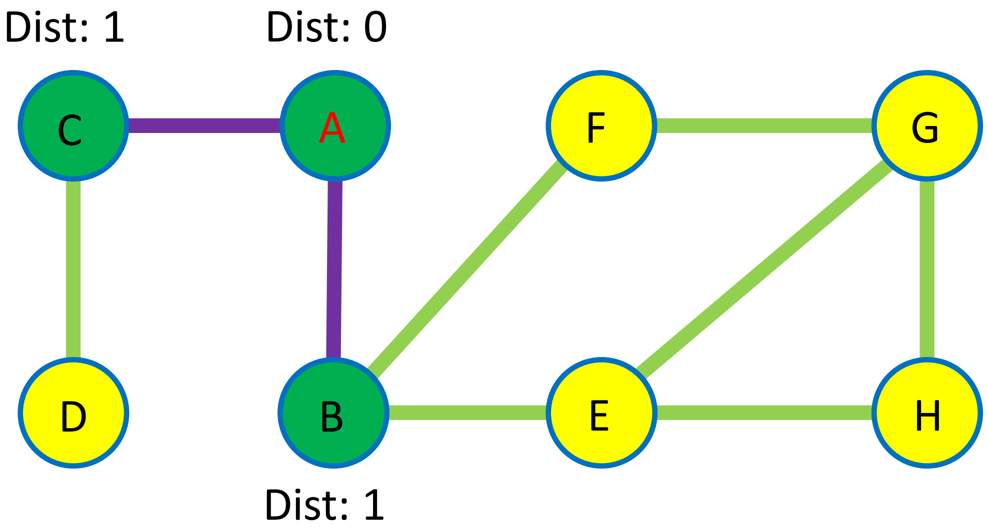
Queue: B

Always keep track of the predecessor nodes.

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🔵

Visited: 🟢

Current: A

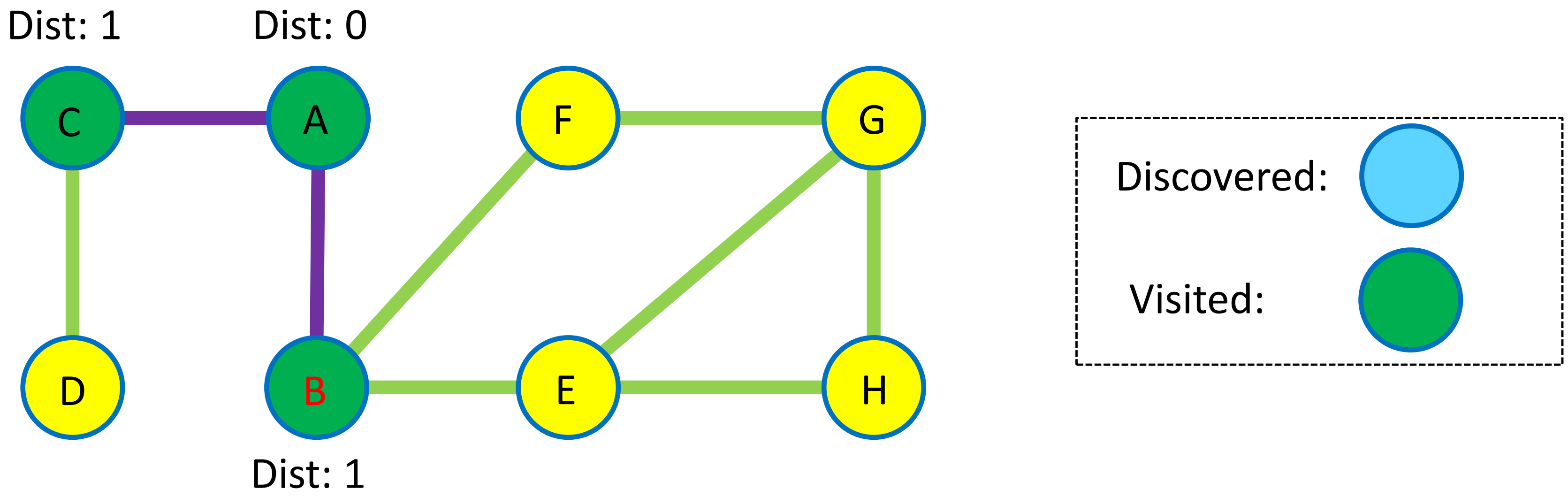
Queue:

B	C

Visited:

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

BFS shortest path (now with good data structures)



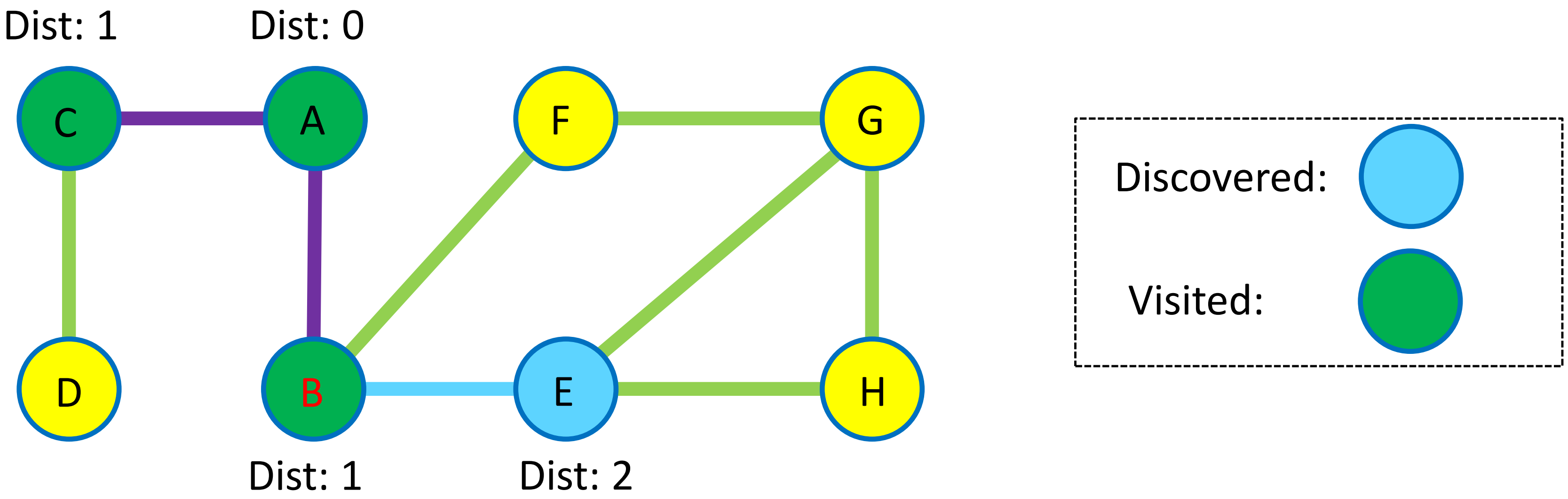
Current: B

Queue: C

Visited:

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

BFS shortest path (now with good data structures)



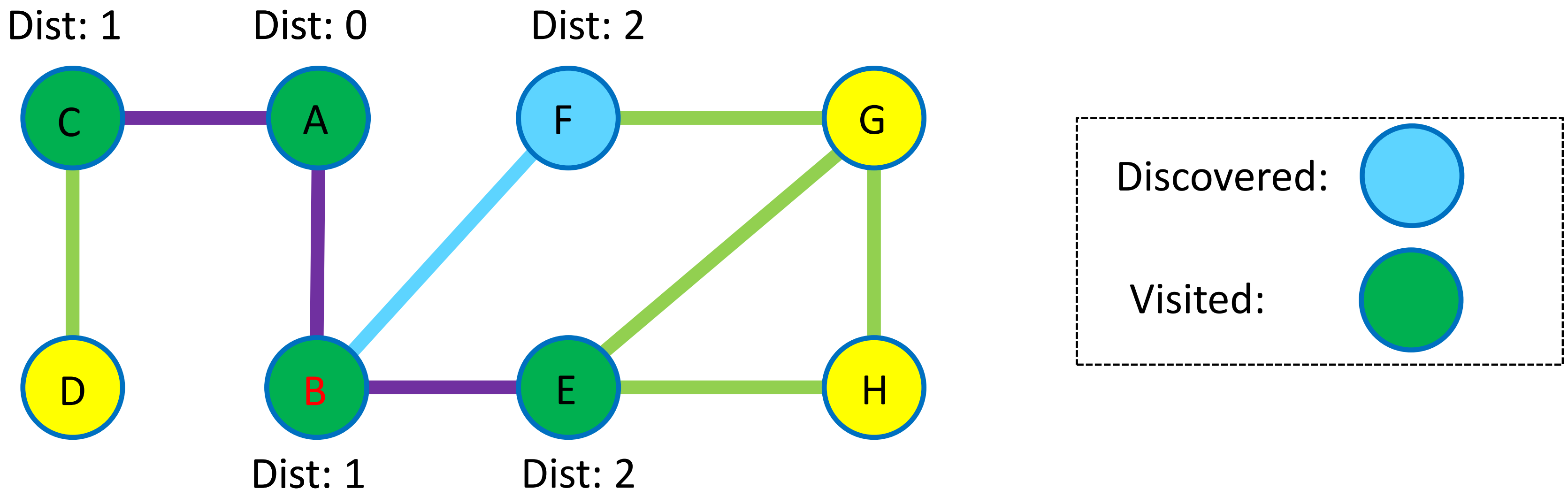
Current: B

Queue: C

Visited:

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

BFS shortest path (now with good data structures)



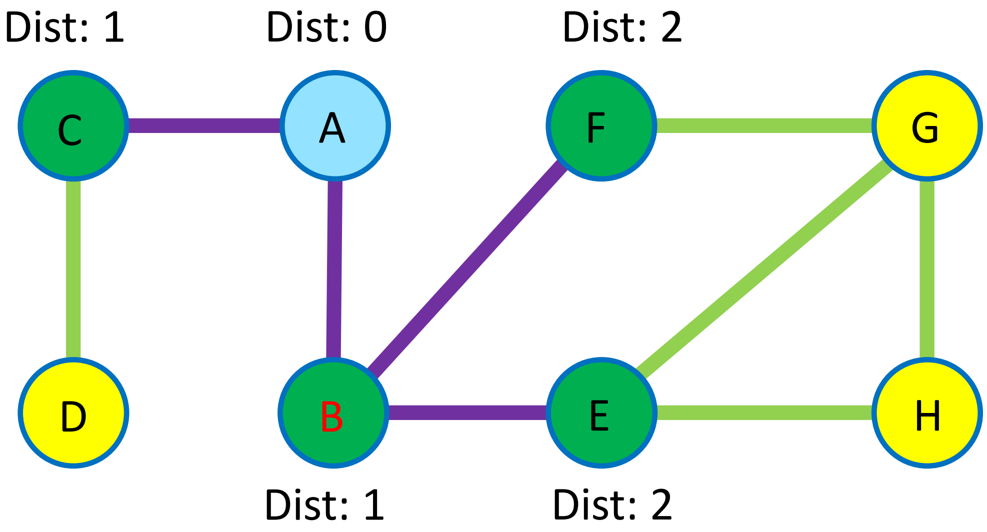
Current: B

Queue: C | E

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🔵

Visited: 🟢

Current: B

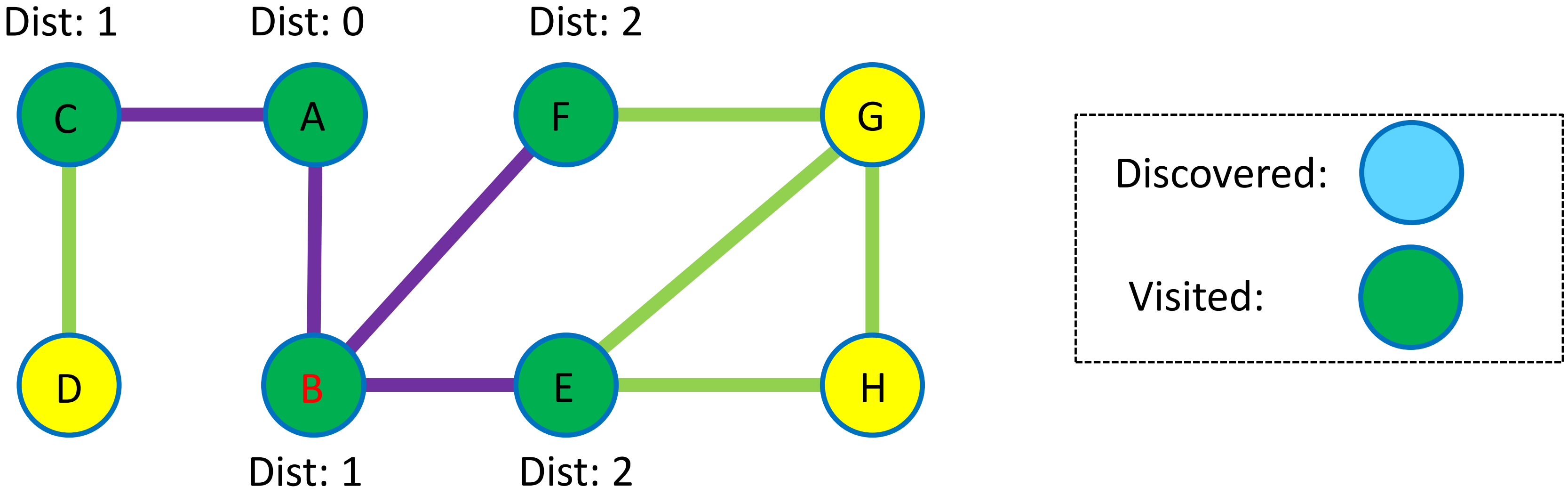
Queue:

C	E	F

Visited:

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

BFS shortest path (now with good data structures)



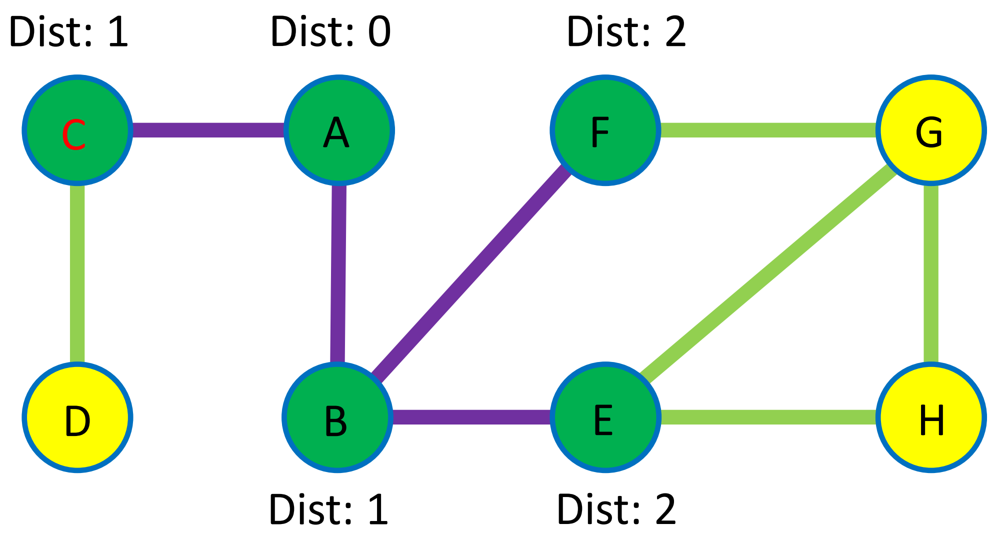
Current: B

Queue: C | E | F

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

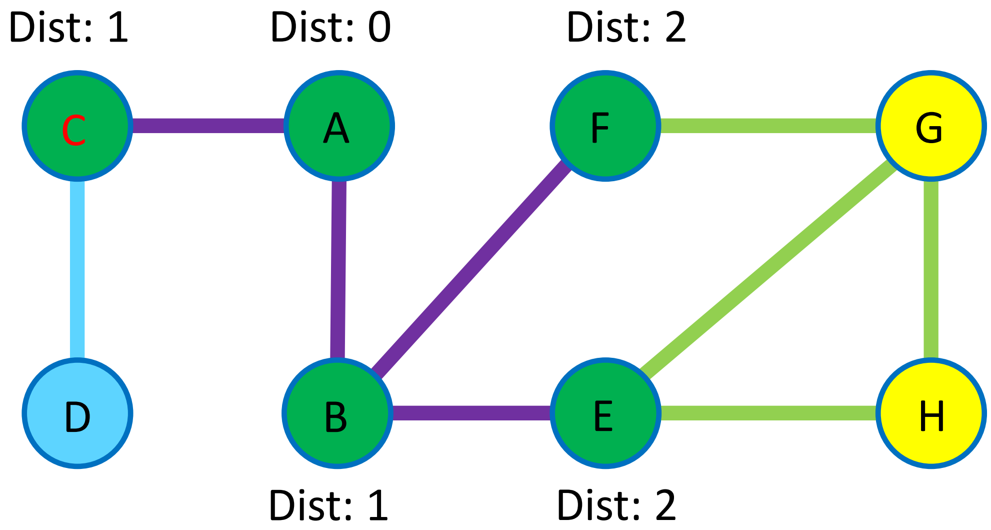
Current: C

Queue: E | F

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

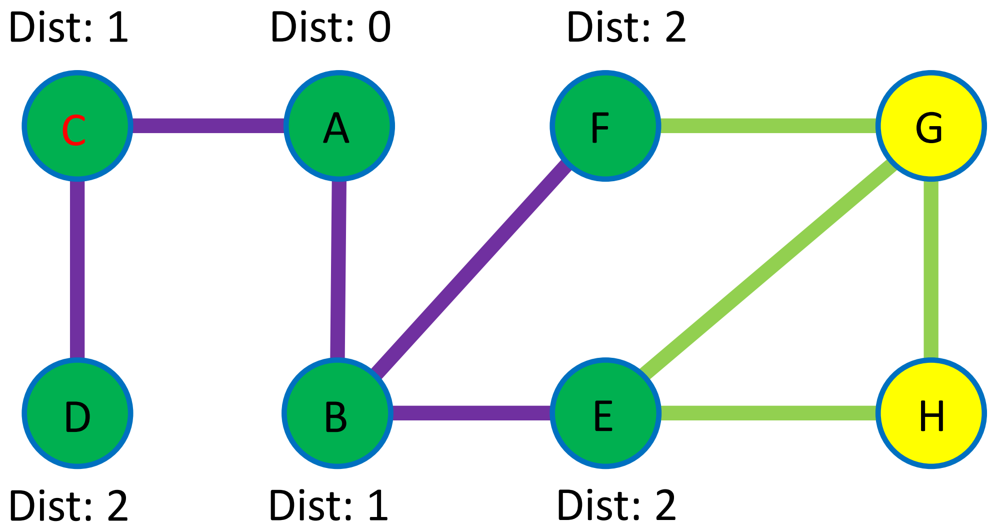
Current: C

Queue: E | F

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

Current:

C

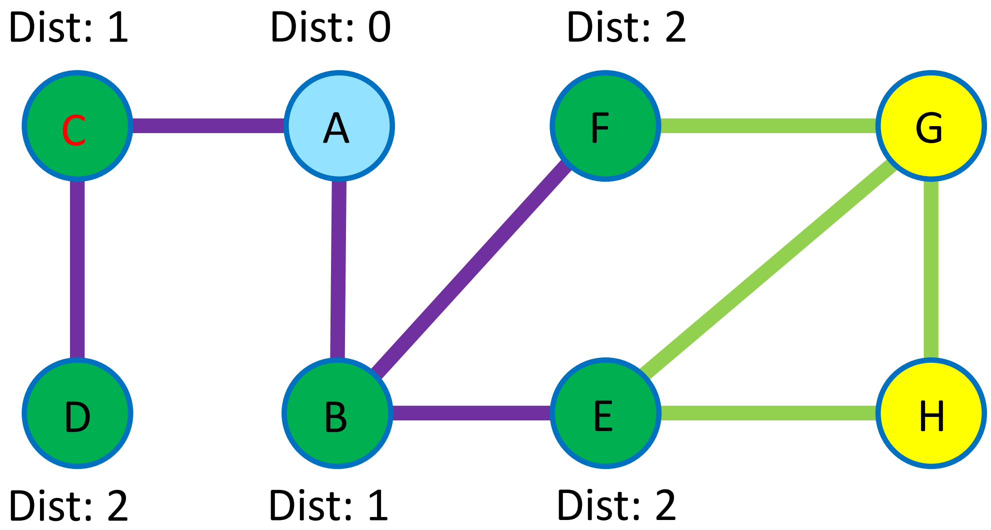
Queue:

E	F	D

Visited:

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

BFS shortest path (now with good data structures)



Discovered: ●

Visited: ●

Current: C

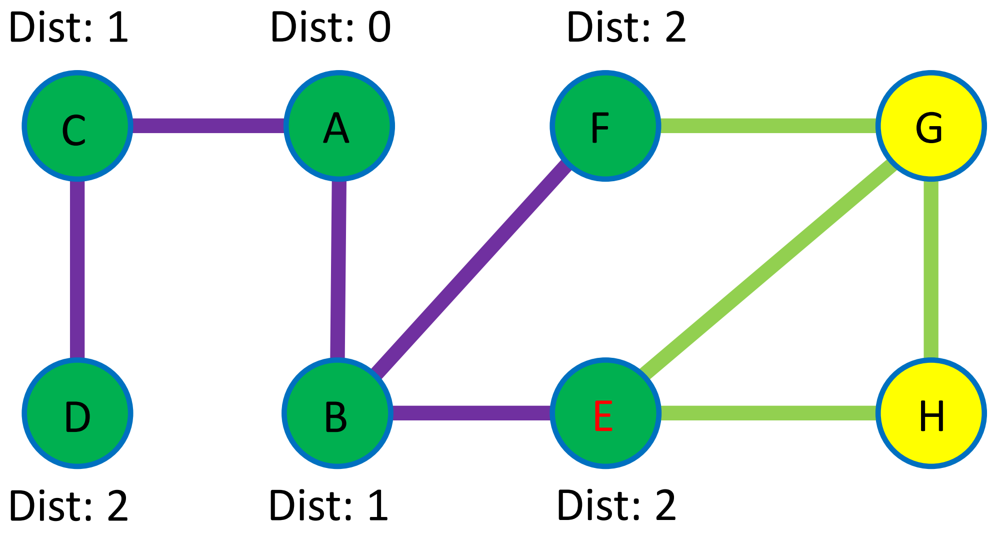
Queue:

E	F	D

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

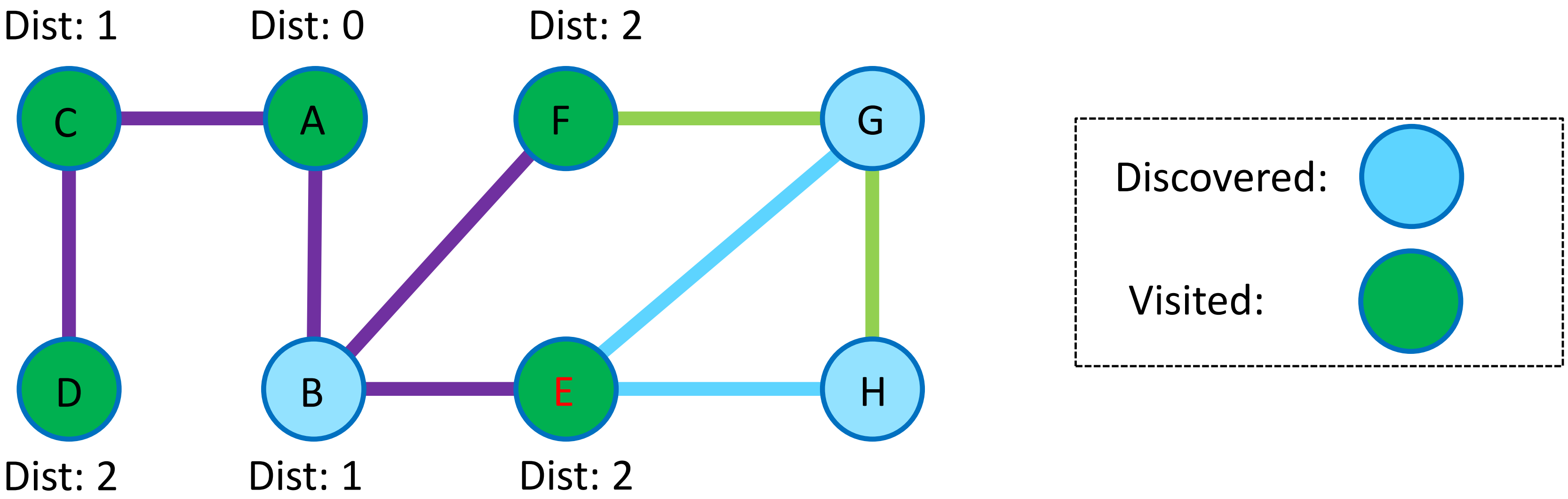
Current: E

Queue: F | D

Visited:

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

BFS shortest path (now with good data structures)



Current: E

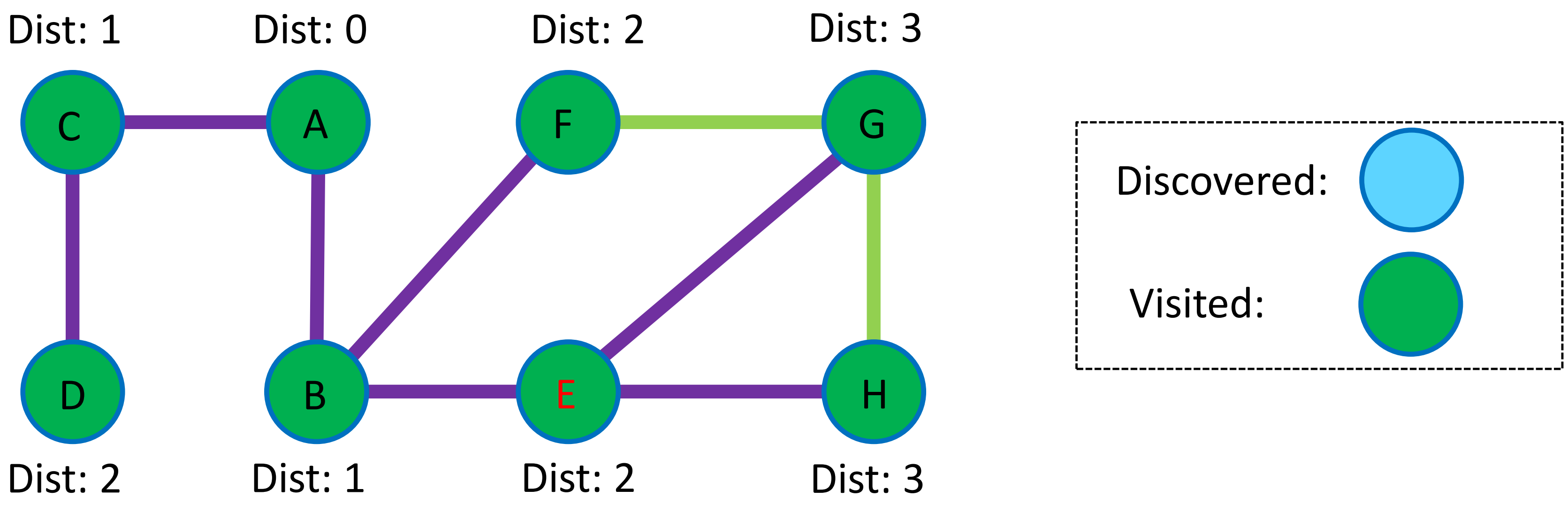
Speeding up visualisation…

Queue: F D

Visited:

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

BFS shortest path (now with good data structures)



Current: E

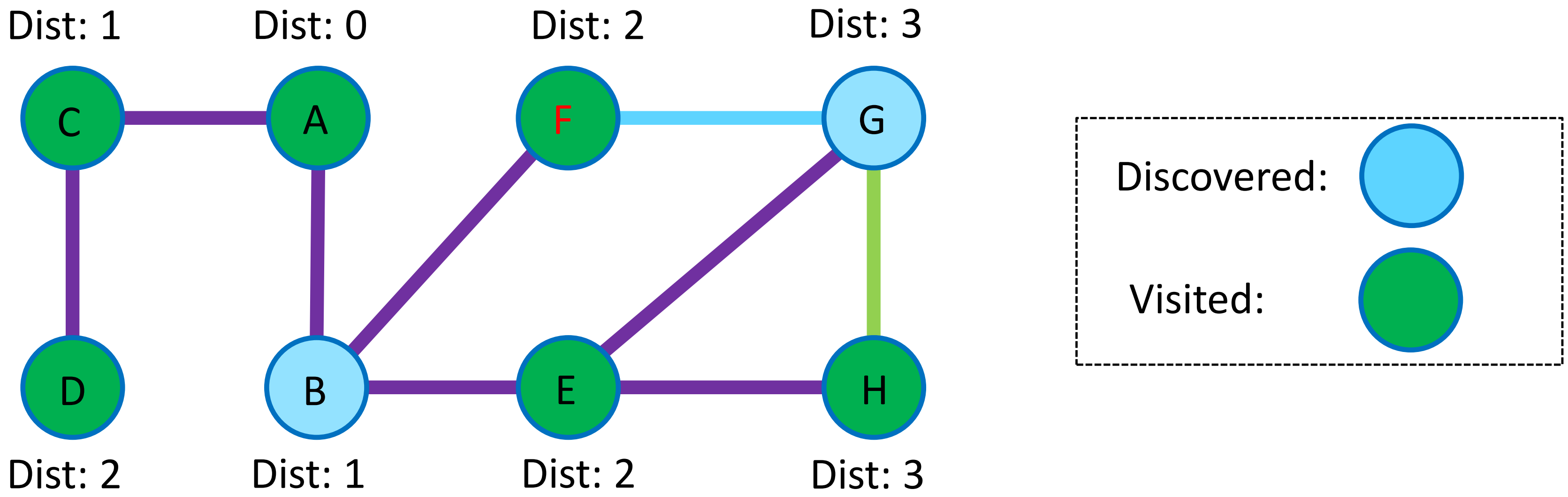
Queue:

F	D	G	H

Visited:

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

BFS shortest path (now with good data structures)



Current: F

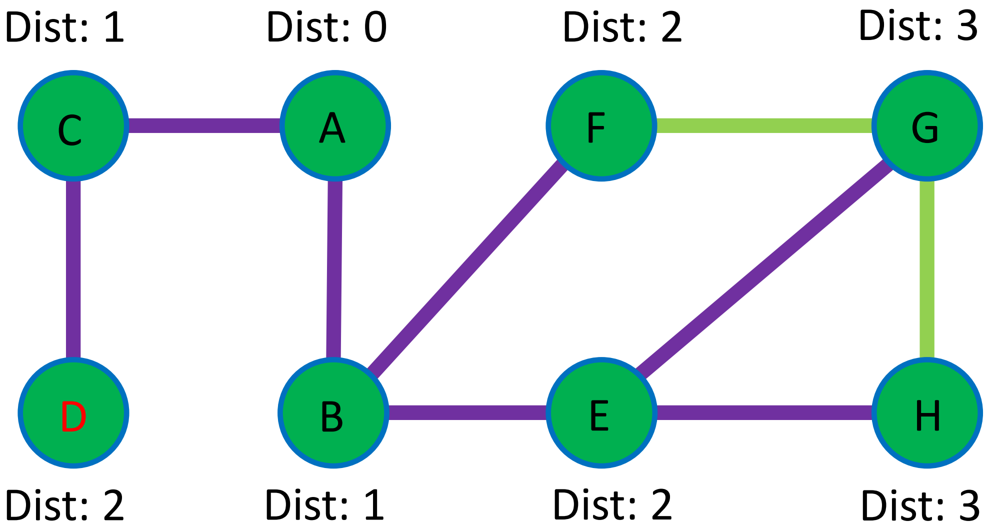
Queue:

D	G	H

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🔵

Visited: 🟢

Current: D

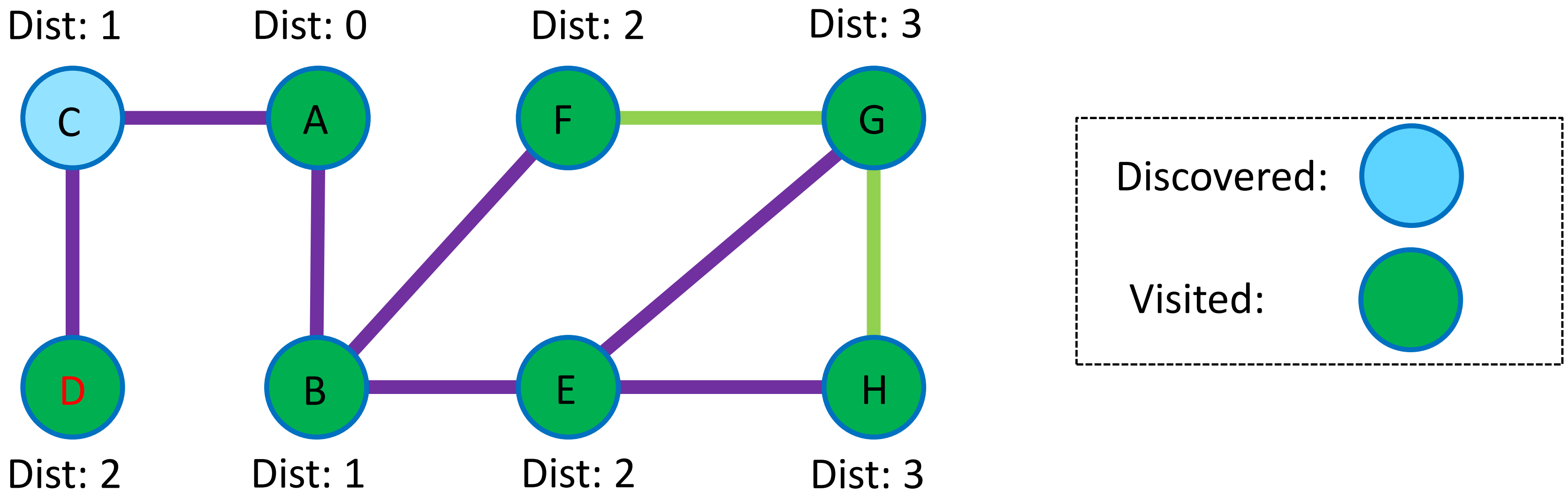
Queue:

G	H

Visited:

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

BFS shortest path (now with good data structures)



Current: D

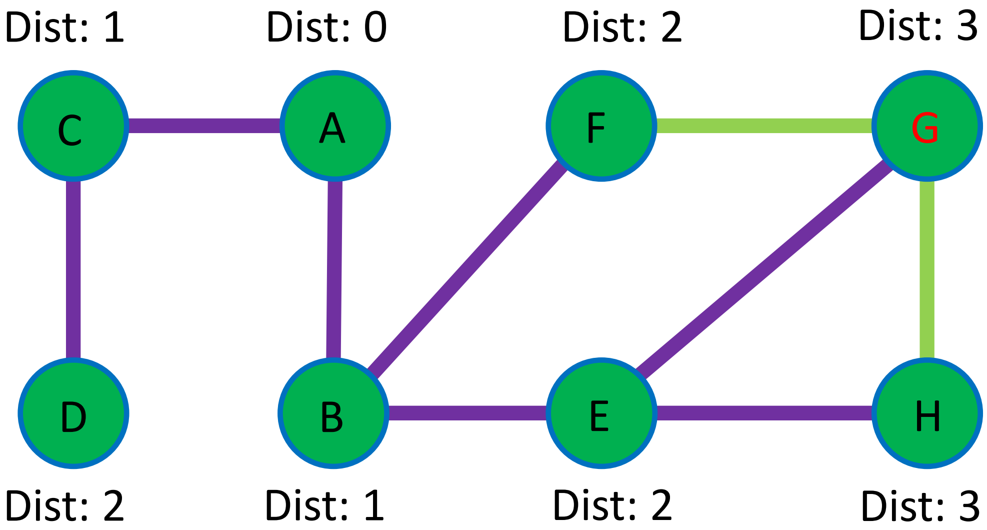
Queue:

G	H

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🔵

Visited: 🟢

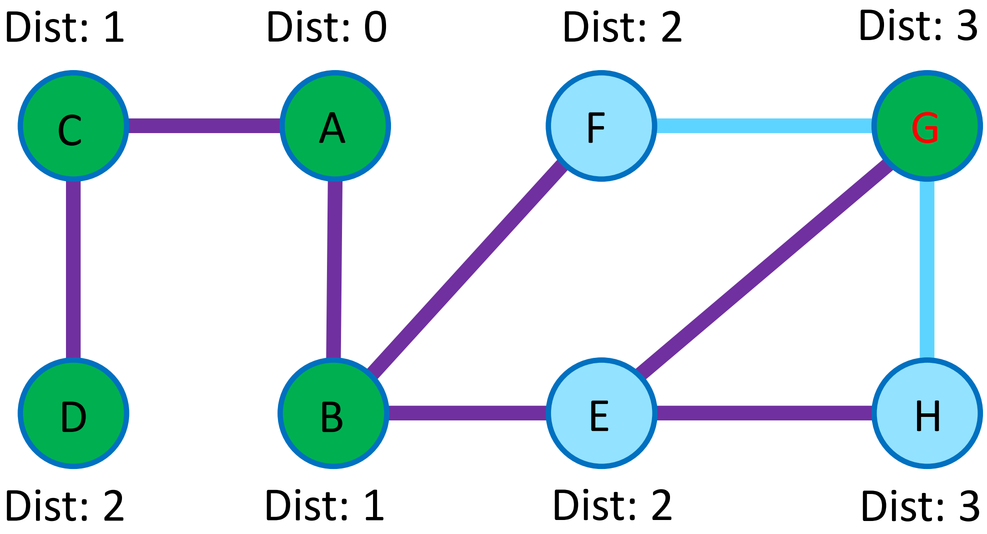
Current: G

Queue: H

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

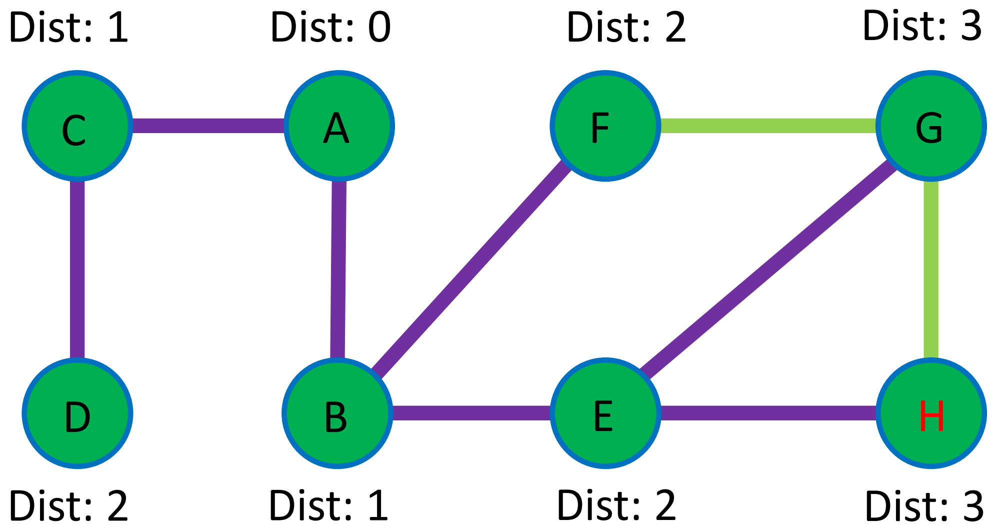
Current: G

Queue: H

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

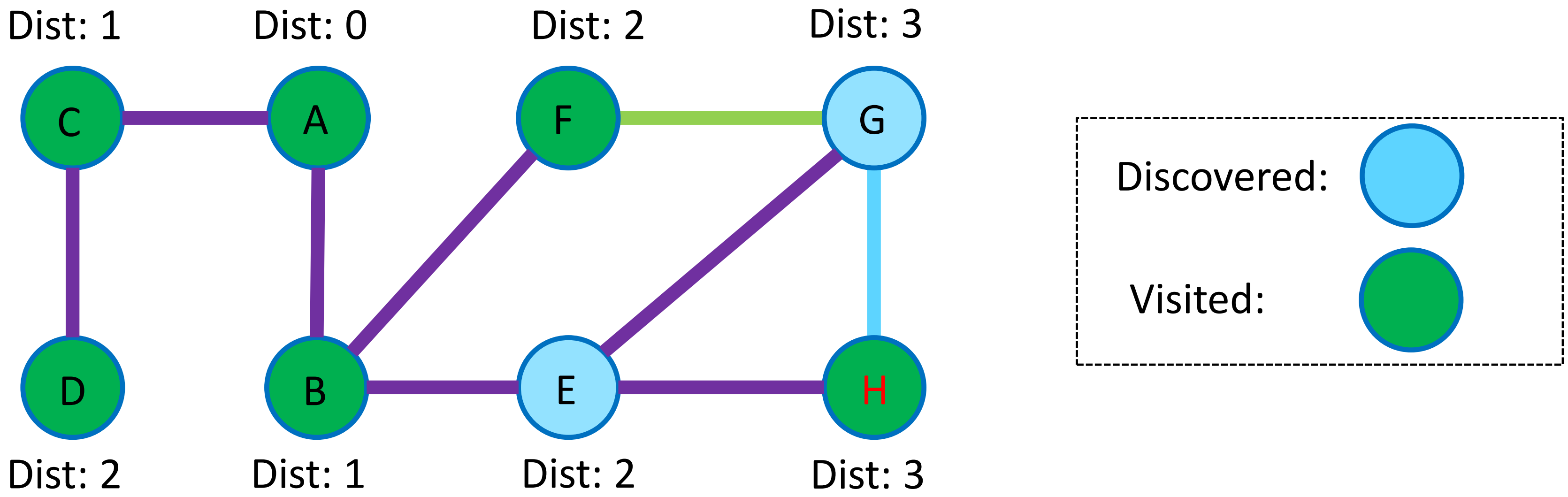
Current: H

Queue:

Visited:

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

BFS shortest path (now with good data structures)



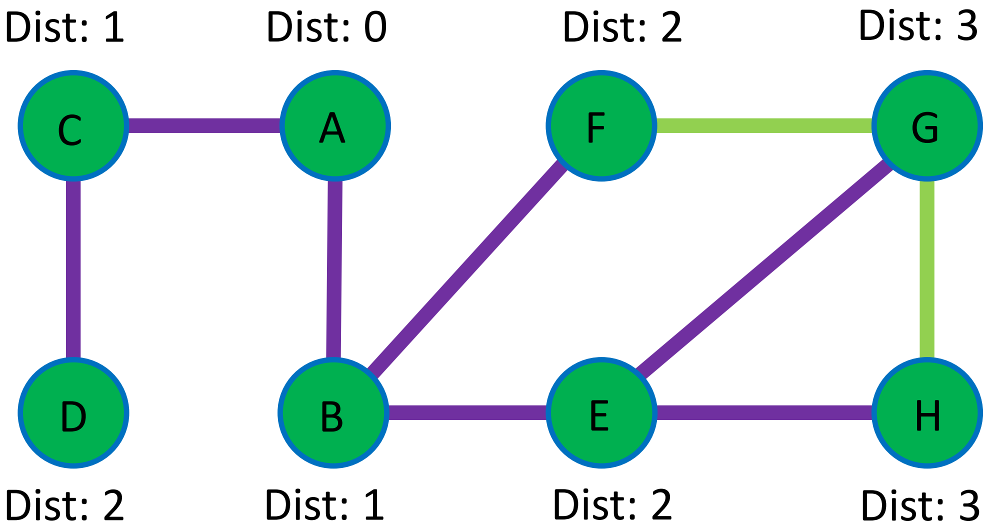
Current: H

Queue:

Visited:

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

BFS shortest path (now with good data structures)



Discovered: 🟦

Visited: 🟩

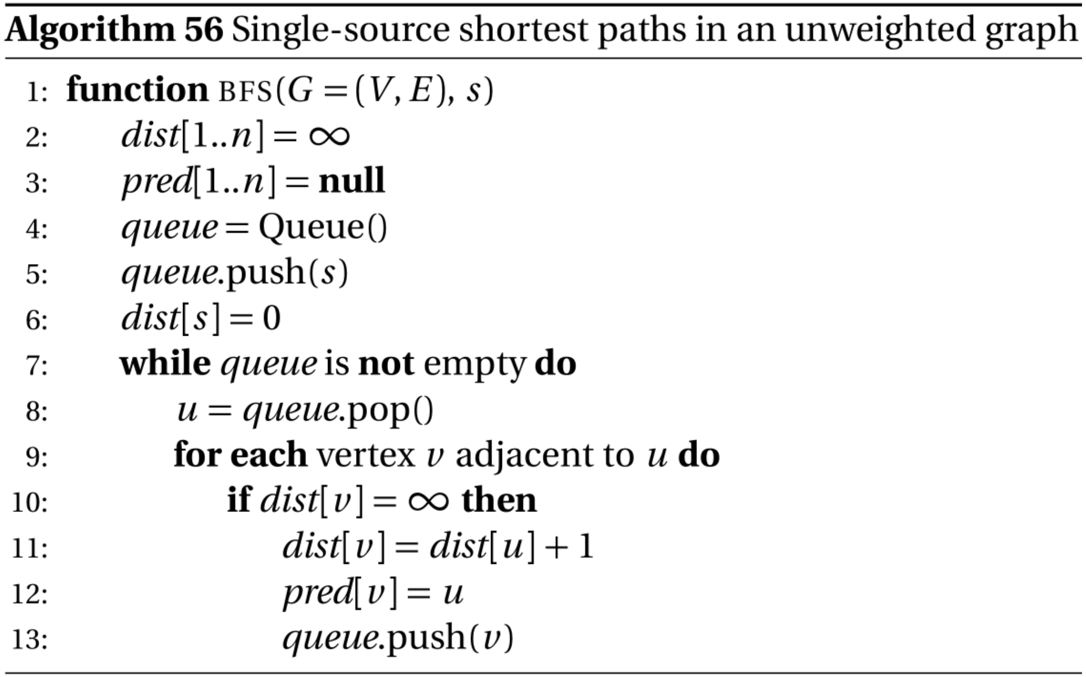
Current:

Queue:

Visited:

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

Unweighted shortest paths



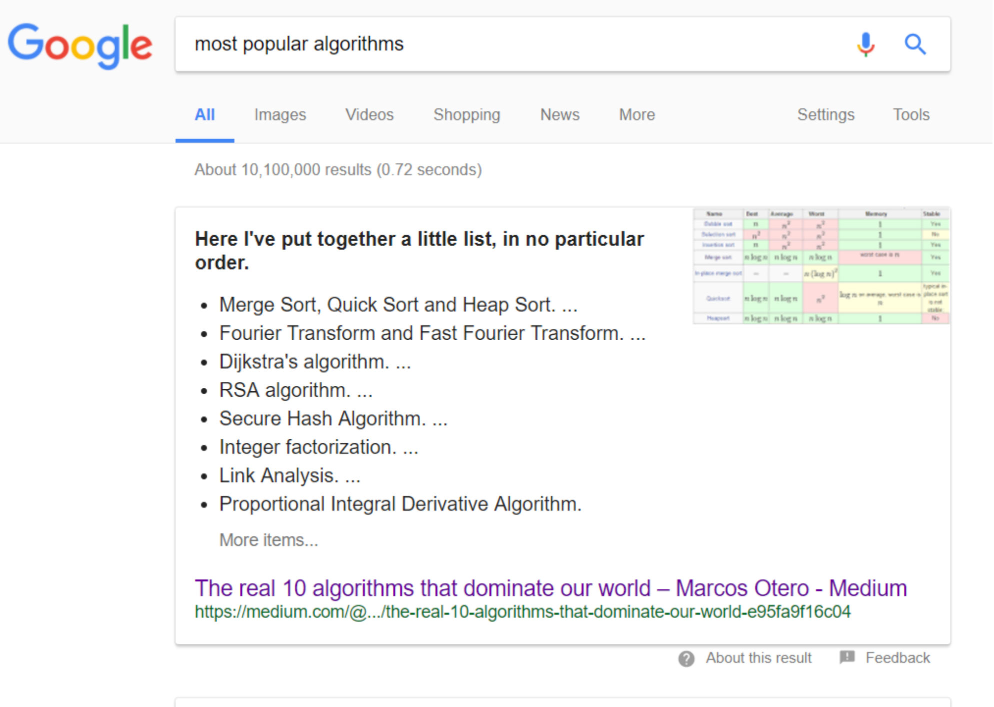
Algorithm 56 Single-source shortest paths in an unweighted graph

```
function BFS($G = (V, E)$, $s$)
    $dist[1..n] = \infty$
    $pred[1..n] =$ null
    $queue =$ Queue()
    $queue$.push($s$)
    $dist[s] = 0$
    while $queue$ is not empty do
        $u = queue$.pop()
        for each vertex $v$ adjacent to $u$ do
            if $dist[v] = \infty$ then
                $dist[v] = dist[u] + 1$
                $pred[v] = u$
                $queue$.push($v$)
```

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

Outline

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



most popular algorithms

About 10,100,000 results (0.72 seconds)

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

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

More items...

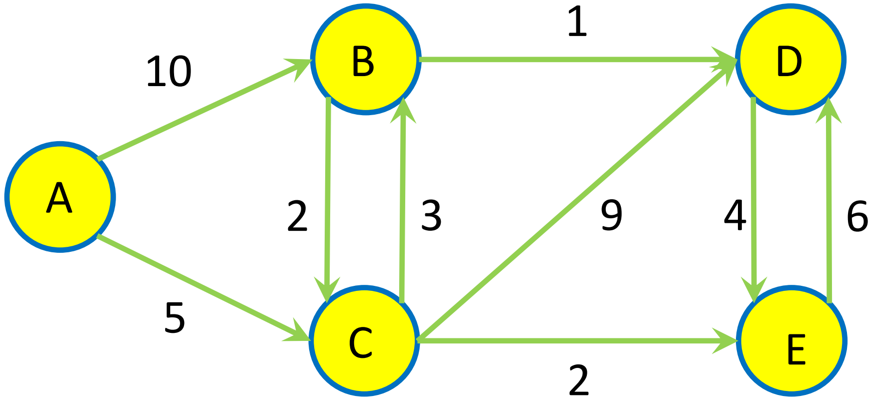
Name	Best	Average	Worst	Memory	Stable
Bubble sort	n	n^2	n^2	1	Yes
Selection sort	n^2	n^2	n^2	1	No
Insertion sort	n	n^2	n^2	1	Yes
Merge sort	$n \log n$	$n \log n$	$n \log n$	worst case is n	Yes
In-place merge sort	—	—	$n (\log n)^2$	1	Yes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$ on average, worst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

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

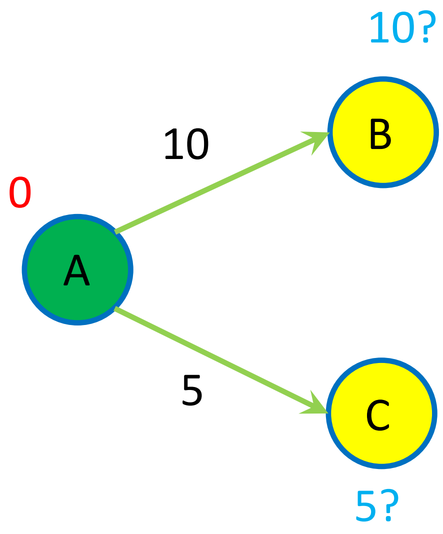
Dijkstra’s Algorithm

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

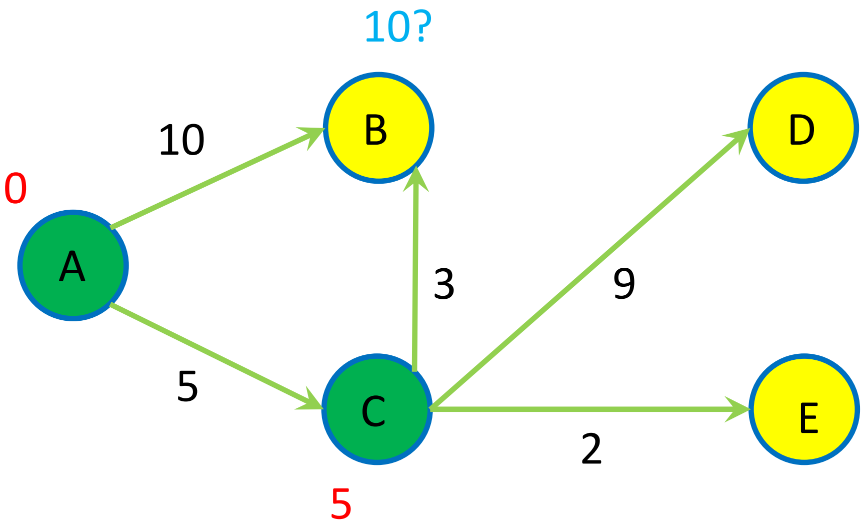
Dijkstra’s Algorithm



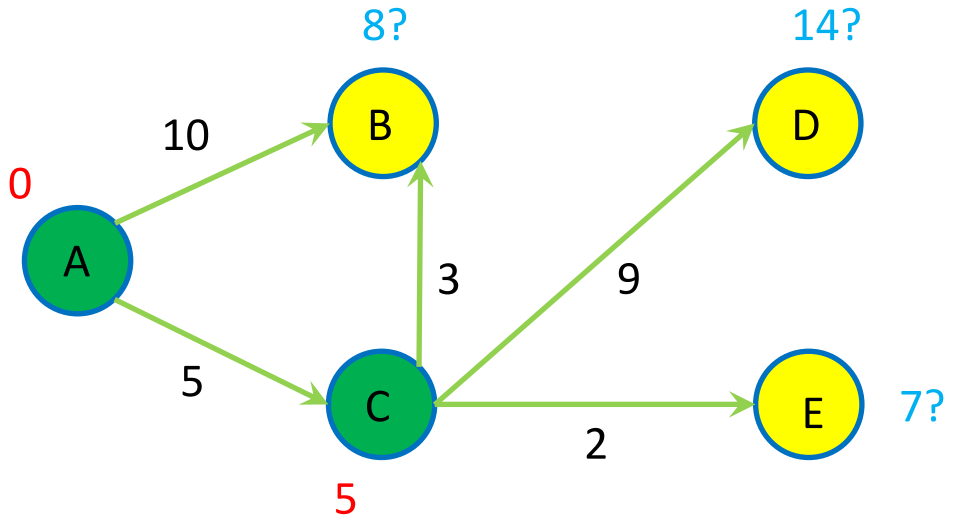
Dijkstra's Algorithm



Dijkstra's Algorithm

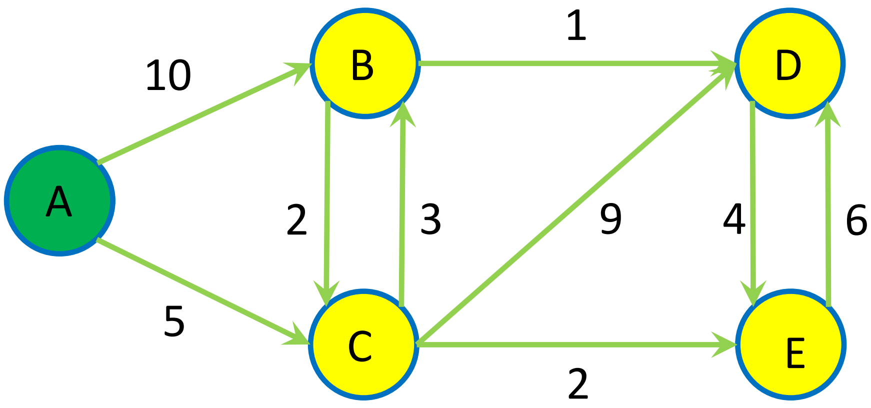


Dijkstra's Algorithm



Dijkstra’s Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

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

Pred and Dist are the usual ID-indexed arrays

Pred:

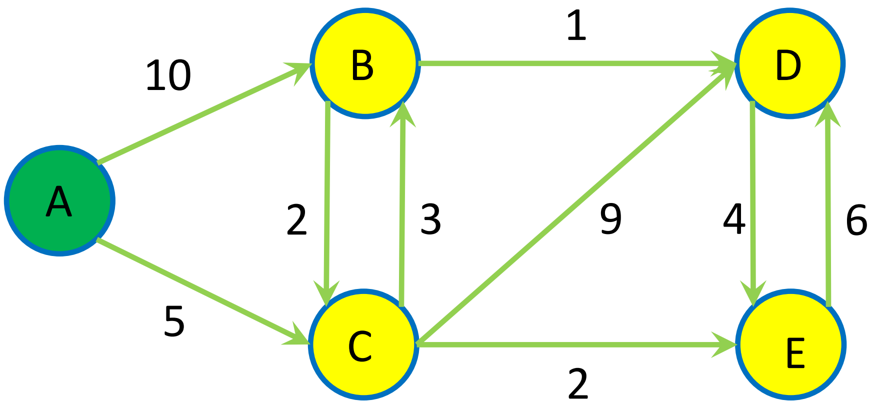
A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Dijkstra's Algorithm

u: A



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

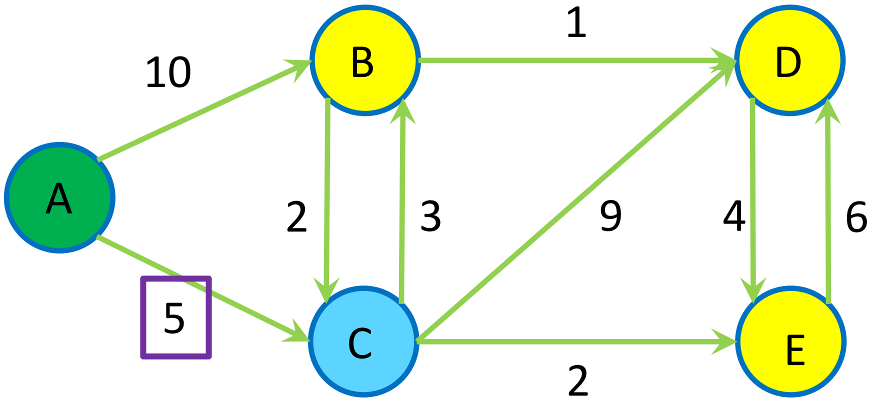
A	B	C	D	E
0	Inf	Inf	Inf	Inf

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

Pred and Dist are the usual ID-indexed arrays

Dijkstra's Algorithm

u: A



For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

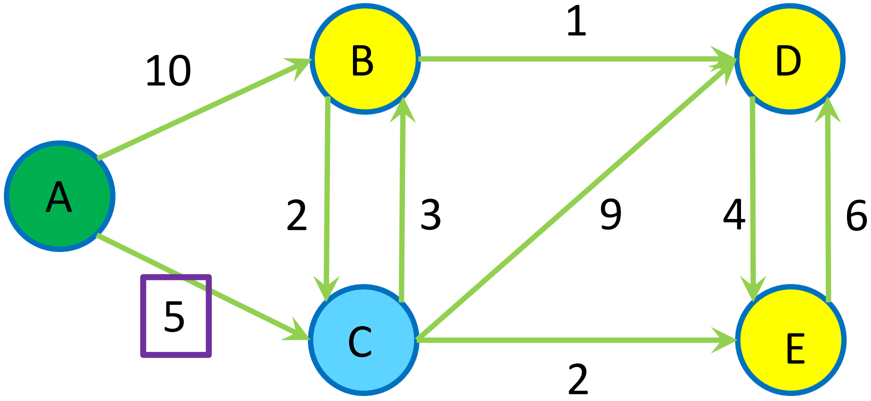
A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Dijkstra’s Algorithm

u: A



For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

- 0 + 5 < Inf

Q:

B	C	D	E
Inf	Inf	Inf	Inf

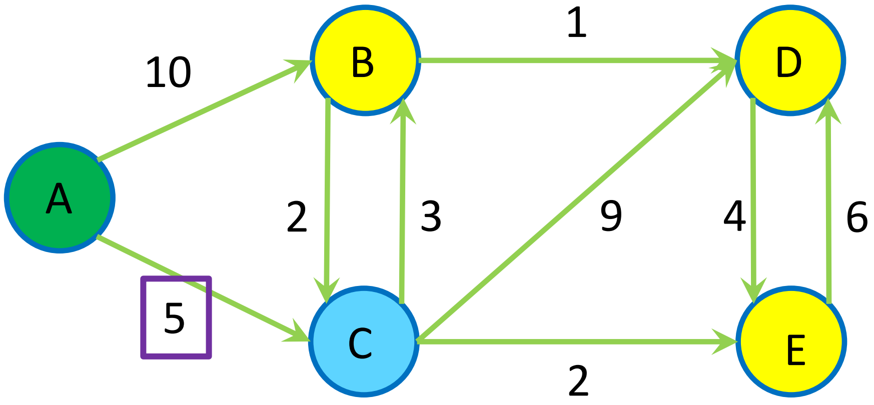
Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Dijkstra's Algorithm



u: A

Q:

C	B	D	E
5	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	A	-	-

Dist:

A	B	C	D	E
0	Inf	5	Inf	Inf

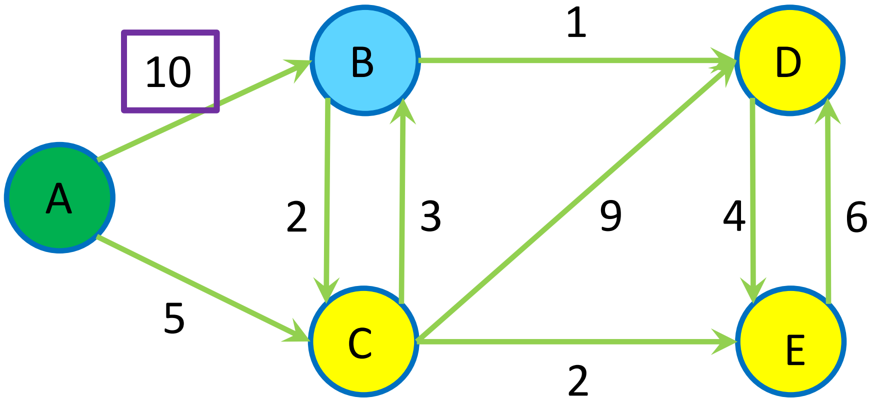
For each neighbour v of u, relax along that edge

- If dist[u] + w(u,v) < dist[v]
- Update dist[v]
- Set pred[v] = u

- 0 + 5 < Inf
- Set dist[C] = 5
- Set pred[C] = A
- Note that this changes the order of Q

Dijkstra's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Doing the same for B

- 0 + 10 < Inf
- Dist[B] = 10

Pred:

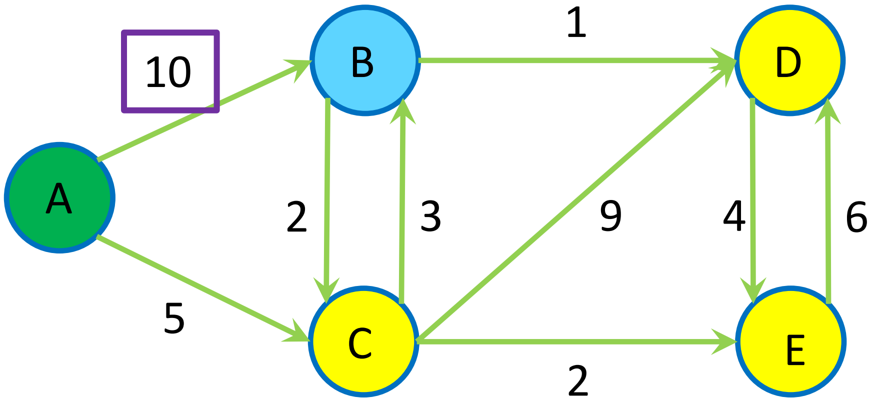
A	B	C	D	E
-	-	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

Dijkstra’s Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

Dist:

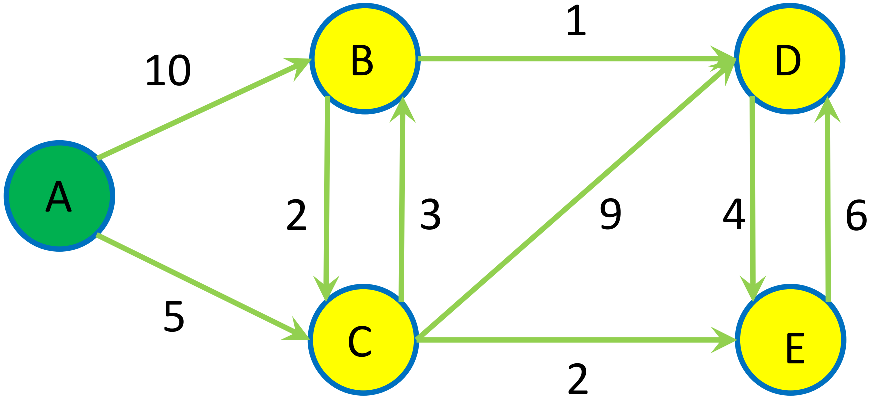
A	B	C	D	E
0	10	5	Inf	Inf

Doing the same for B

- 0 + 10 < Inf
- Dist[B] = 10
- Pred[B] = A

Dijkstra's Algorithm

u: A



Q:

C	B	D	E
5	10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

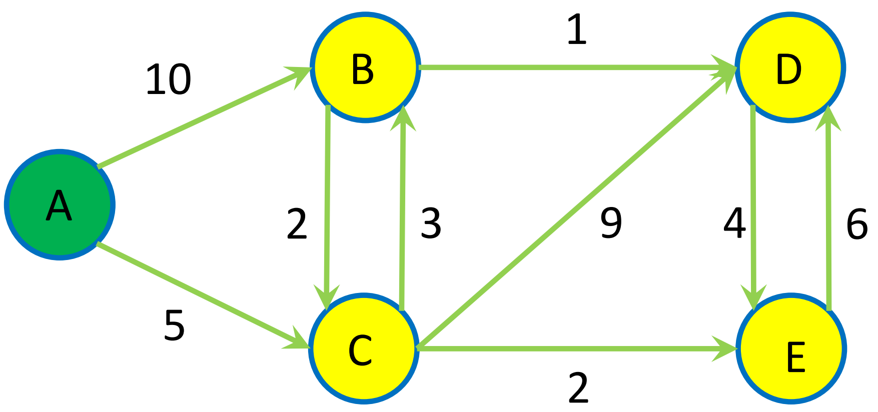
Dist:

A	B	C	D	E
0	10	5	Inf	Inf

- Finished with A, so pop from Q
- Notice that this will always be the vertex with the smallest dist

Dijkstra's Algorithm

u: C



Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

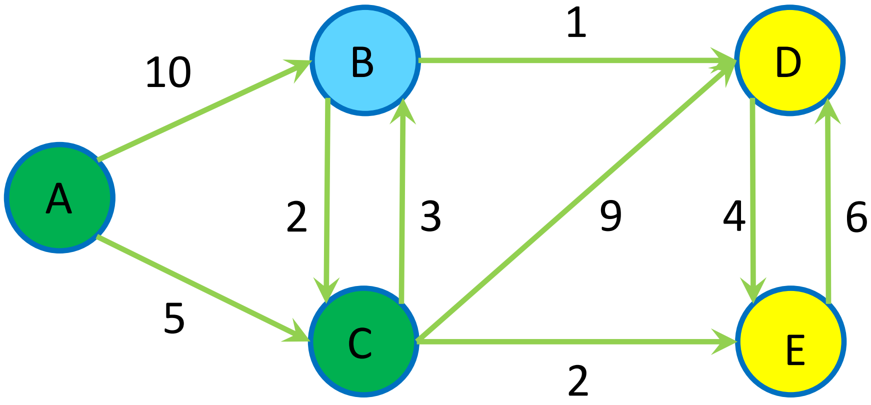
Dist:

A	B	C	D	E
0	10	5	Inf	Inf

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

Dijkstra's Algorithm

u: C



- Relax B from C
- Dist[C] + w(C, B) = 5 + 3 < 10

Q:

B	D	E
10	Inf	Inf

Pred:

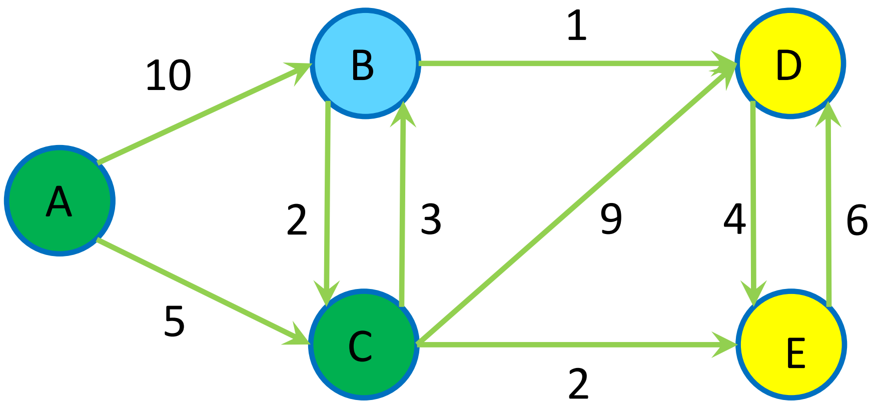
A	B	C	D	E
-	A	A	-	-

Dist:

A	B	C	D	E
0	10	5	Inf	Inf

Dijkstra's Algorithm

u: C



Q:

B	D	E
8	Inf	Inf

- Relax B from C
- Dist[C] + w(C, B) = 5 + 3 < 10

- Dist[B] = 8
- Pred[B] = C

Pred:

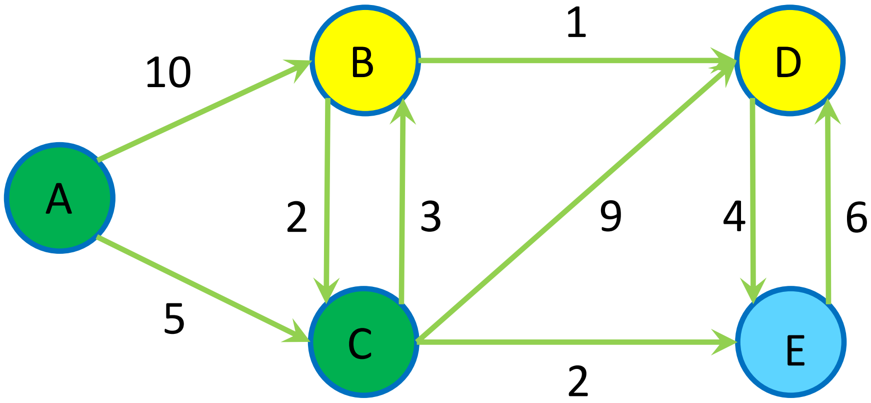
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	8	5	Inf	Inf

Dijkstra’s Algorithm

u: C



- Relax E from C
- Dist[C] + w(C, E) = 5 + 2 < Inf

Q:

B	D	E
8	Inf	Inf

Pred:

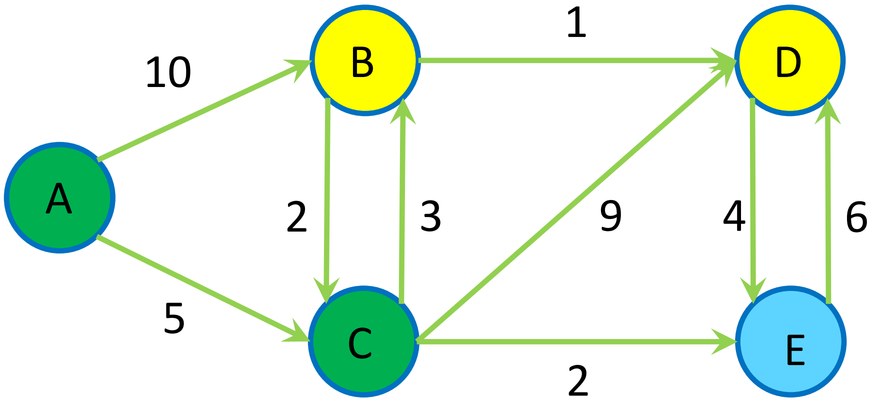
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	8	5	Inf	Inf

Dijkstra’s Algorithm

u: C



Q:

E	B	D
7	8	Inf

- Relax E from C
- Dist[C] + w(C, E) = 5 + 2 < Inf

- Dist[E] = 7
- Pred[E] = C

Pred:

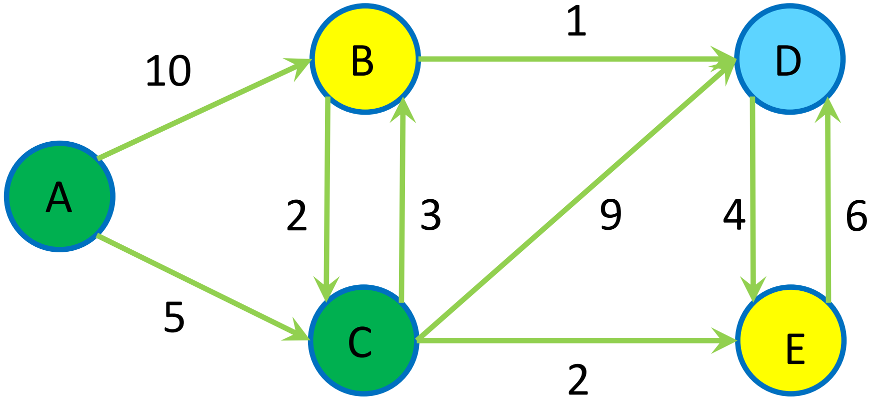
A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	8	5	Inf	7

Dijkstra's Algorithm

u: C



- Relax D from C
- Dist[D] + w(C, D) = 5 + 9 < Inf

Q:

E	B	D
7	8	Inf

Pred:

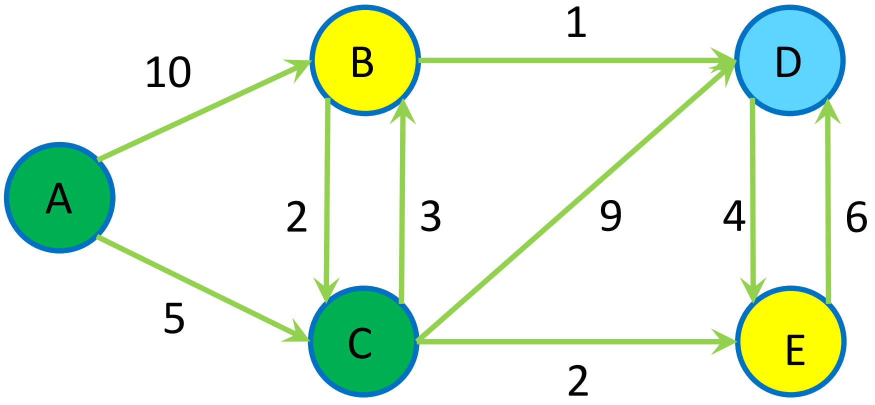
A	B	C	D	E
-	C	A	-	C

Dist:

A	B	C	D	E
0	8	5	Inf	7

Dijkstra's Algorithm

u: C



Q:

E	B	D
7	8	14

Pred:

A	B	C	D	E
-	C	A	C	C

Dist:

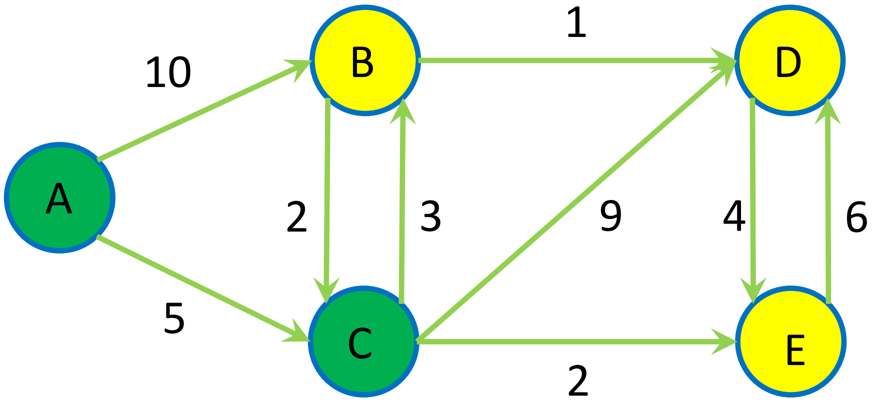
A	B	C	D	E
0	8	5	14	7

- Relax D from C
- Dist[D] + w(C, D) = 5 + 9 < Inf

- Dist[D] = 14
- Pred[D] = C

Dijkstra’s Algorithm

u: C



Q:

E	B	D
7	8	14

Pred:

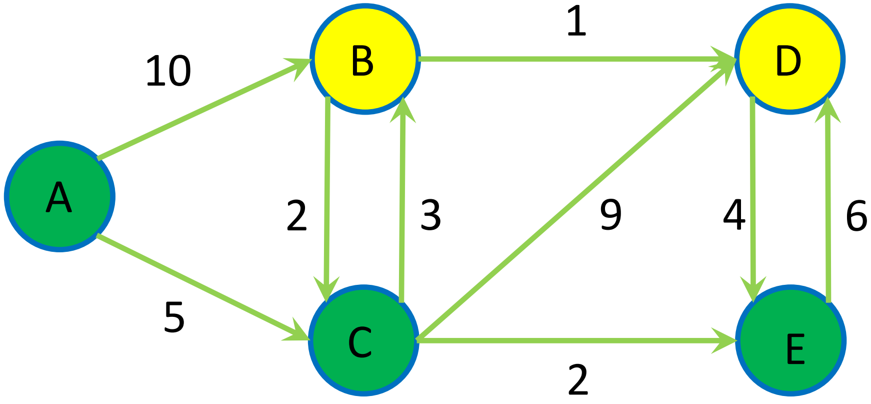
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

- Done with C
- Pop another vertex from Q and finalise it

Dijkstra’s Algorithm



u: E

Q:

B	D
8	14

Pred:

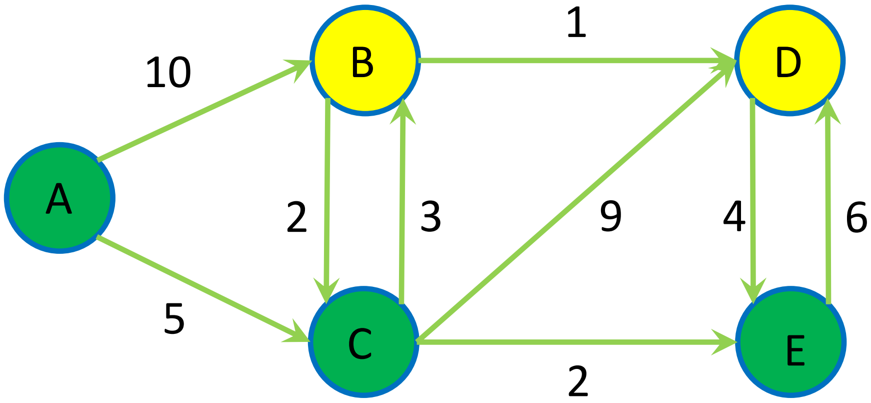
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra's Algorithm

u: E



Q:

B	D
8	14

- Relax from E
- Dist[E] + w(E, D) = 7 + 6 < 14 = Dist[D]

Pred:

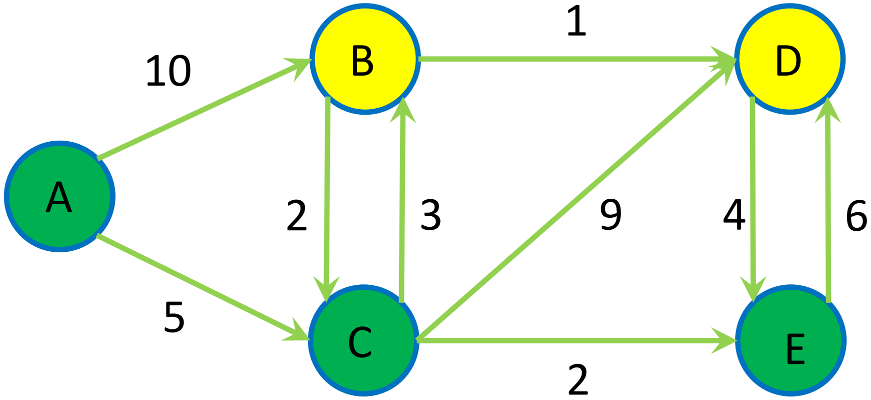
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra's Algorithm

u: E



Q:

B	D
8	13

Pred:

A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	8	5	13	7

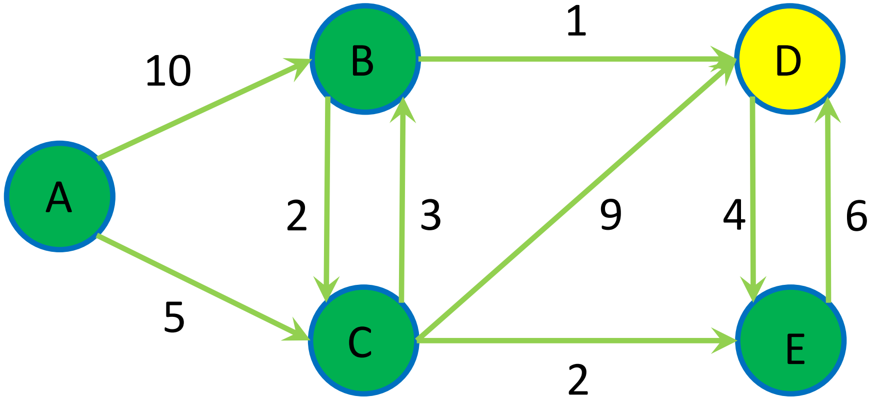
- Relax from E
- Dist[E] + w(E, D) = 7 + 6 < 14 = Dist[D]

- Dist[D] = 13
- Pred[D] = D

- Done with E

- Pop B

Dijkstra’s Algorithm



u: B

Q:

D
13

- Relax from B
- Dist[B] + w(B, D) = 8 + 1 = 9 < 13 = Dist[D]

Pred:

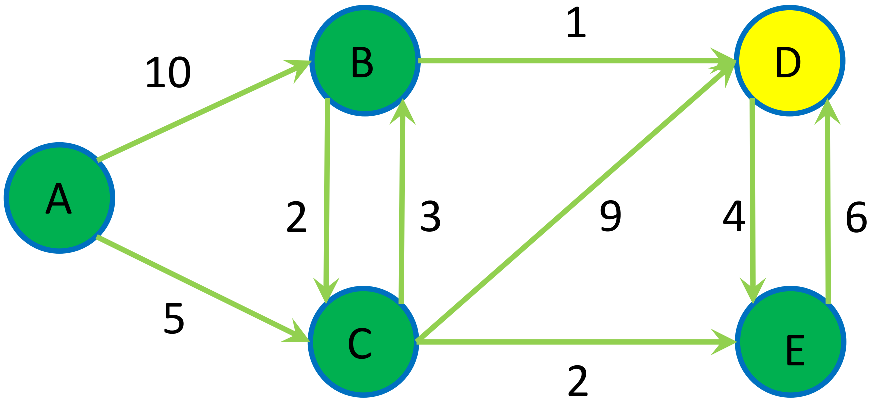
A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	8	5	13	7

Dijkstra's Algorithm

u: B



Q:

D
9

Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

- Relax from B
- Dist[B] + w(B, D) = 8 + 1 = 9 < 13 = Dist[D]

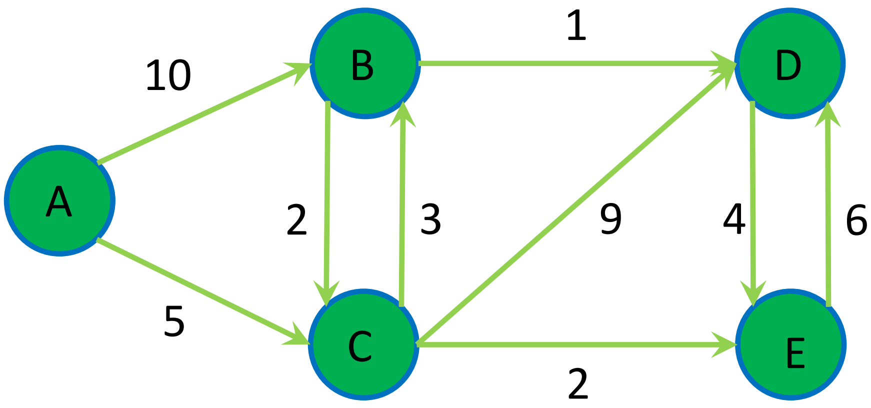
- Dist[D] = 9
- Pred[D] = B

- Done with B

- Pop D

Dijkstra's Algorithm

u: D



Q:

- No neighbours to relax
- Done!

Pred:

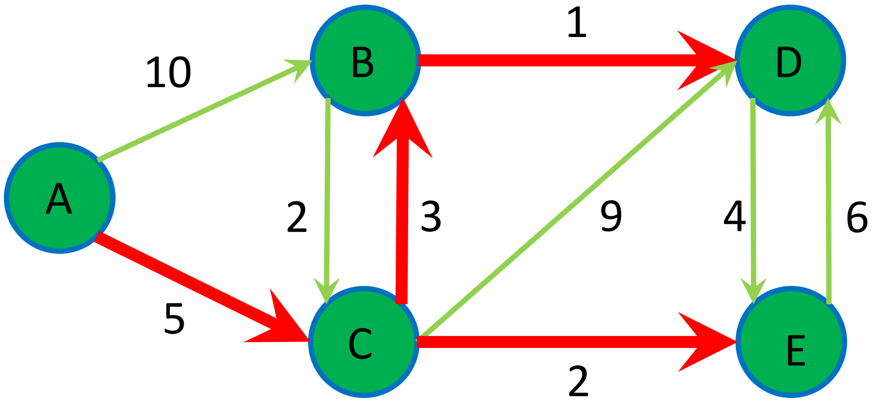
A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra’s Algorithm

u: D



Q:

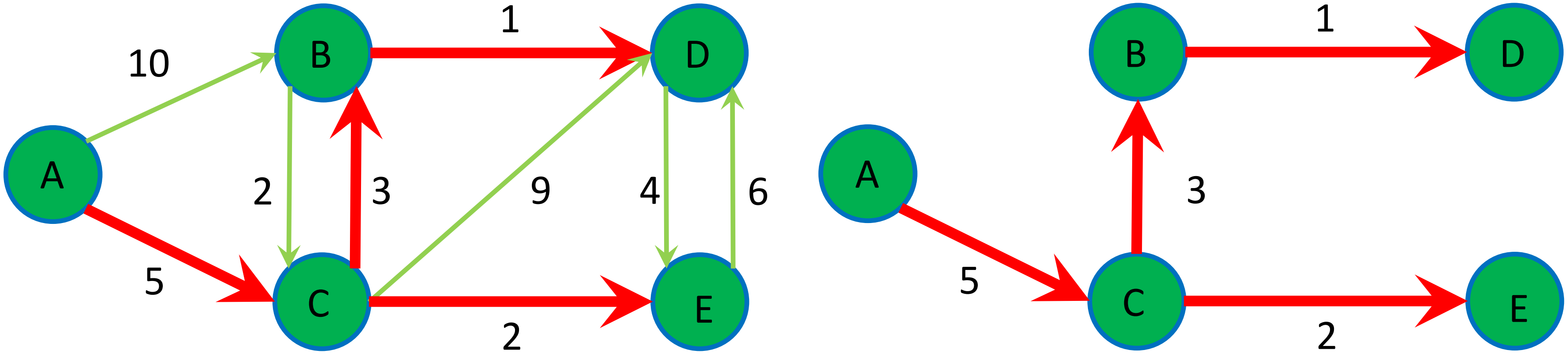
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra’s Algorithm



Q:

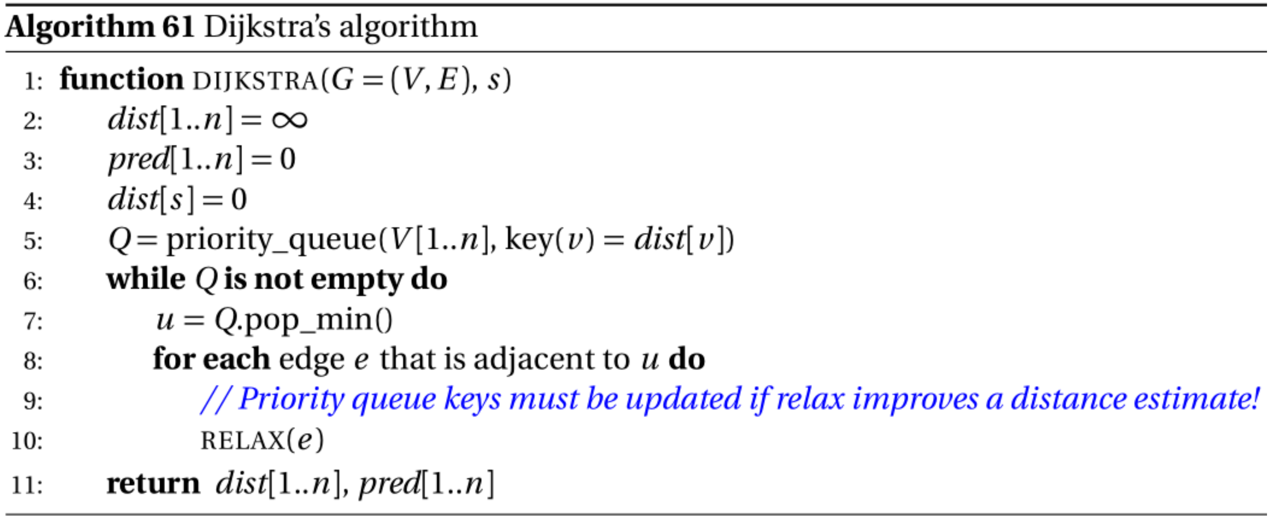
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra's Algorithm

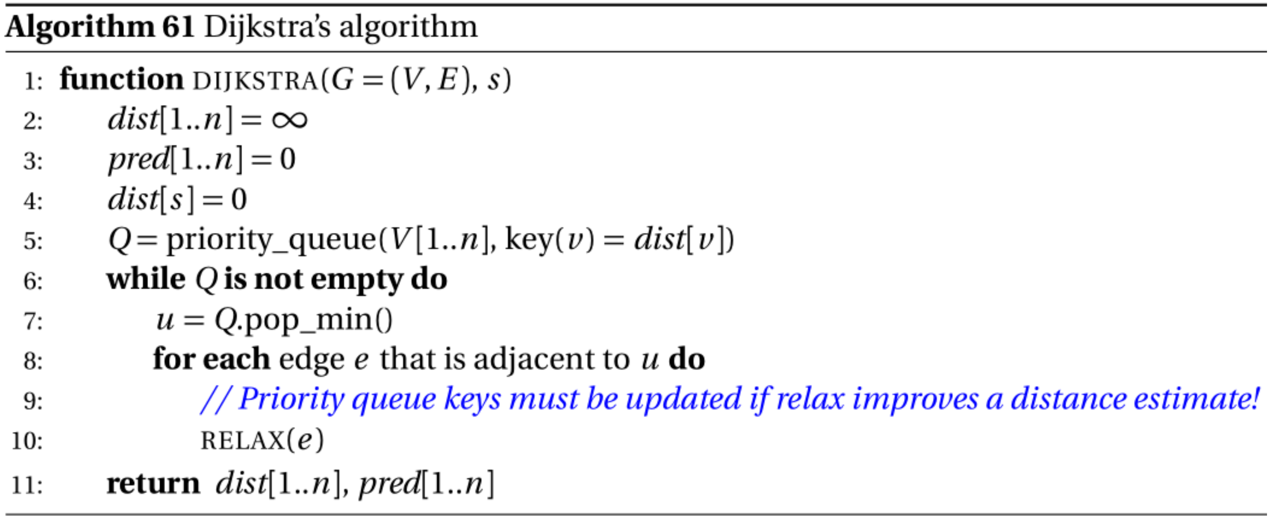


Algorithm 61 Dijkstra's algorithm

```
function DIJKSTRA($G = (V, E), s$)
    $dist[1..n] = \infty$
    $pred[1..n] = 0$
    $dist[s] = 0$
    $Q$ = priority_queue($V[1..n]$, key($v$) = $dist[v]$)
    while $Q$ is not empty do
        $u = Q$.pop_min()
        for each edge $e$ that is adjacent to $u$ do
            // Priority queue keys must be updated if relax improves a distance estimate!
            RELAX($e$)
    return $dist[1..n]$, $pred[1..n]$
```

Quiz time!
https://flux.qa - YTJMAZ

Dijkstra’s Algorithm



Algorithm 61 Dijkstra’s algorithm

```
function DIJKSTRA($G = (V, E)$, $s$)
    $dist[1..n] = \infty$
    $pred[1..n] = 0$
    $dist[s] = 0$
    $Q$ = priority_queue($V[1..n]$, key($v$) = $dist[v]$)
    while $Q$ is not empty do
        $u$ = $Q$.pop_min()
        for each edge $e$ that is adjacent to $u$ do
            // Priority queue keys must be updated if relax improves a distance estimate!
            RELAX($e$)
    return $dist[1..n]$, $pred[1..n]$
```

Time Complexity:

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

Dijkstra’s Algorithm using min-heap

Required additional structure:

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

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

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

Time Complexity of Dijkstra’s Algorithm

Time Complexity:

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

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

Time Complexity of Dijkstra's Algorithm

Time Complexity:

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

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

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

- Total cost O(ElogV)

Time Complexity of Dijkstra's Algorithm

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

Alternative approach that does not require the vertices array (with the same complexity) is described in the notes section 13.4, "Practical considerations: Implementing Dijkstra's algorithm"

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

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

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

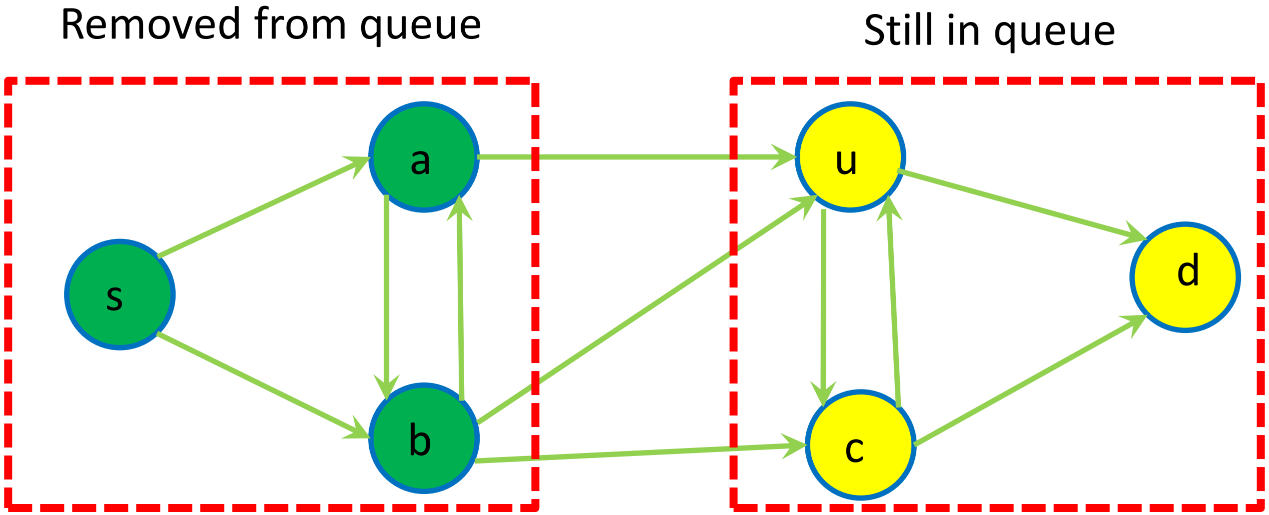
- dist[s] is initialised to 0, which is the shortest distance from s to s (since there are no negative weights)

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

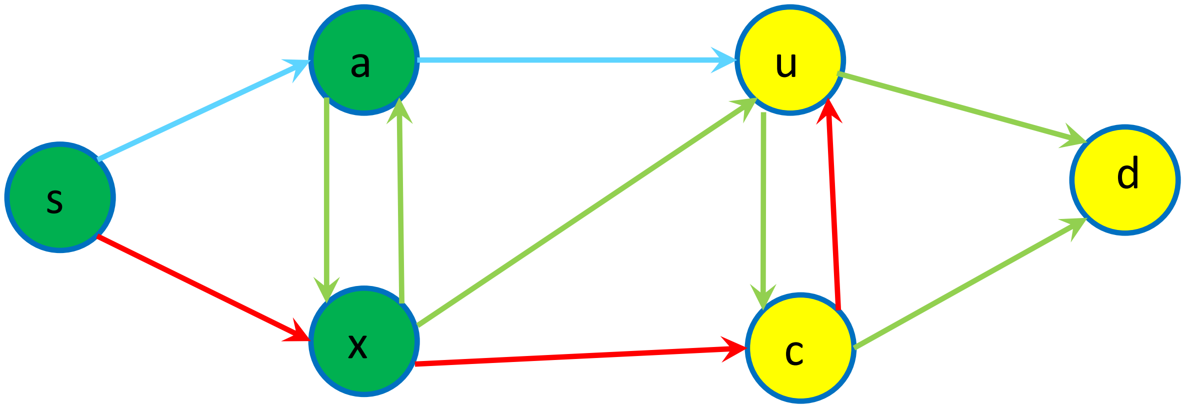
Claim: For every vertex v which has been removed from the queue, dist[v] is correct

Inductive Step:

- Suppose (for contradiction) there is a shortest path P, $s \rightsquigarrow u$ 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

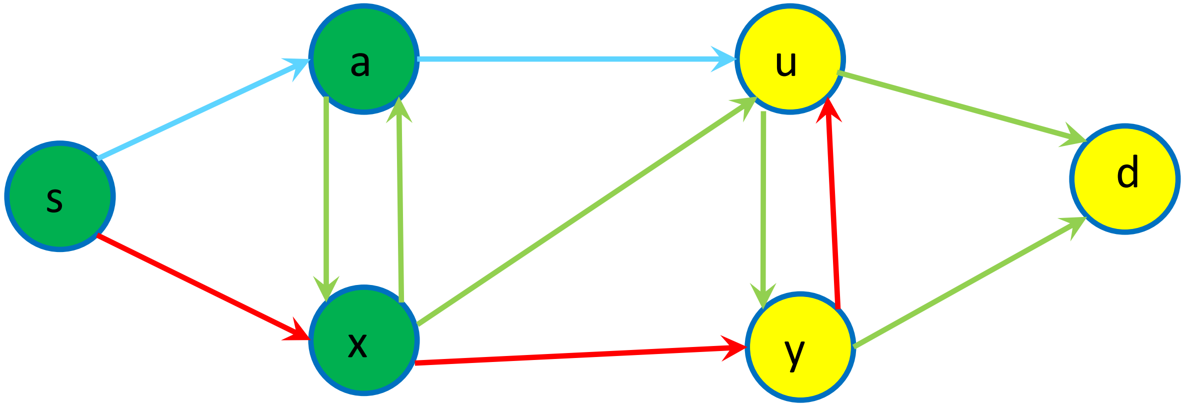
Claim: For every vertex v which has been removed from the queue, dist[v] is correct

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⇝y) <= len(P) < dist[u]

Current path

Assumed shorter path (P)

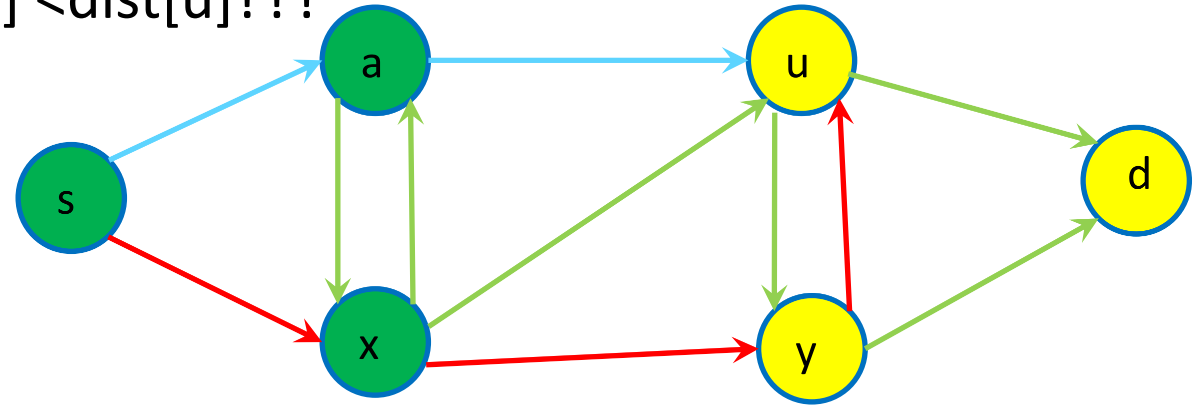


Proof of Correctness

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

Inductive Step:

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



Current path

Assumed shorter path (P)

Proof of Correctness

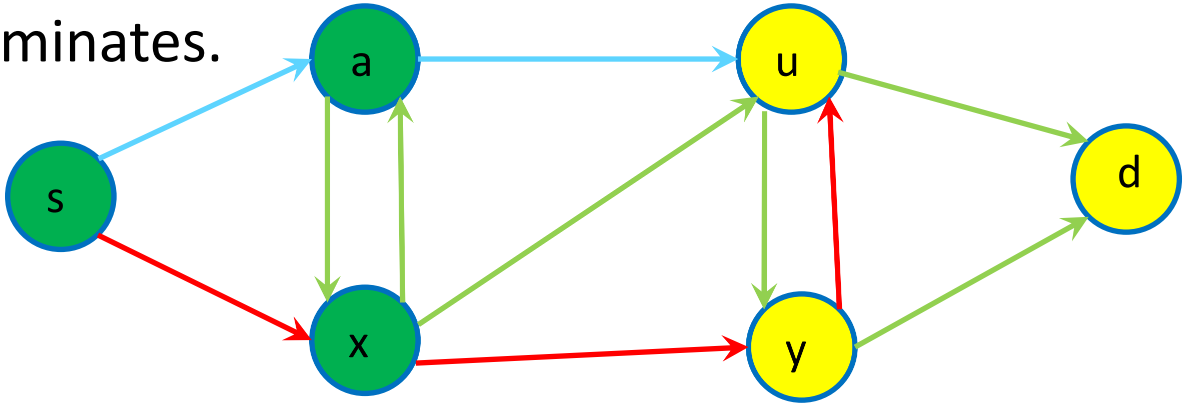
Claim: For every vertex v which has been removed from the queue, dist[v] is correct

Inductive Step:

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

Current path

Assumed shorter path (P)



Summary

Take home message

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

Things to do (this list is not exhaustive)

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

Coming Up Next

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