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.

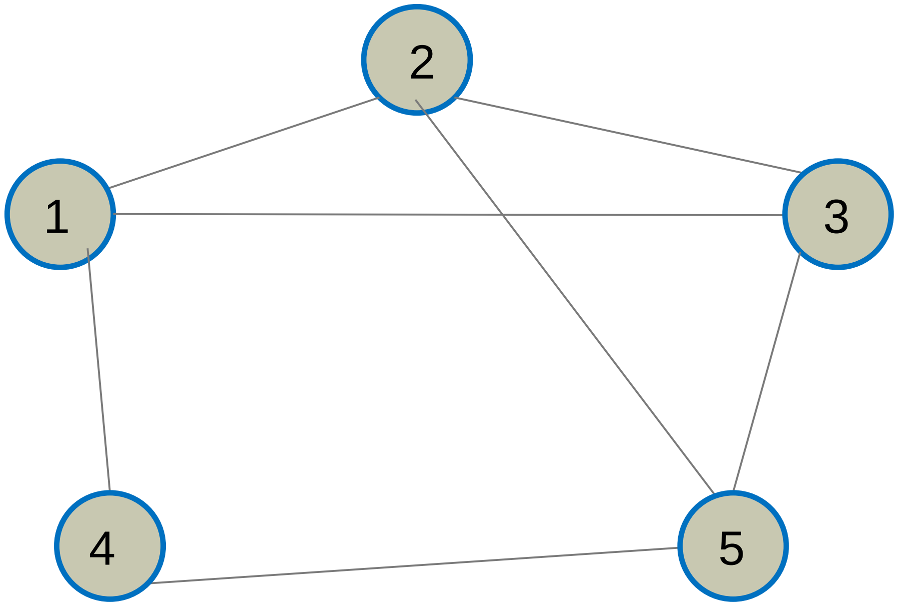
Recommended reading

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

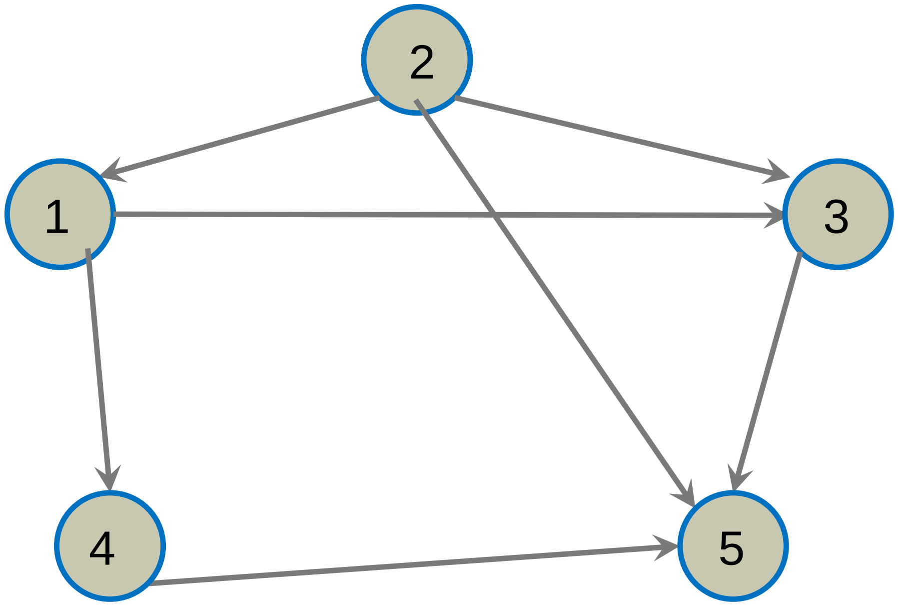
Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. 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)

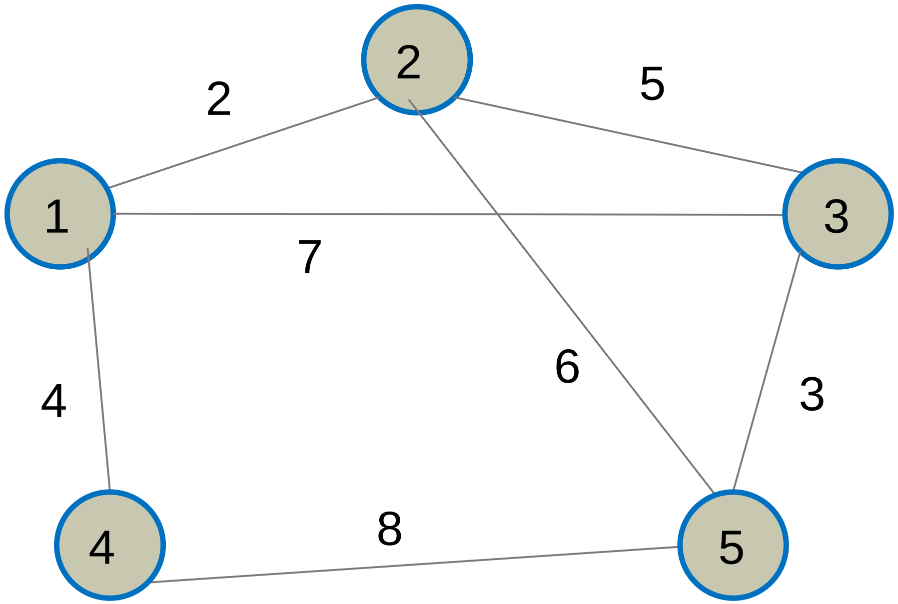
Undirected Graph - Example



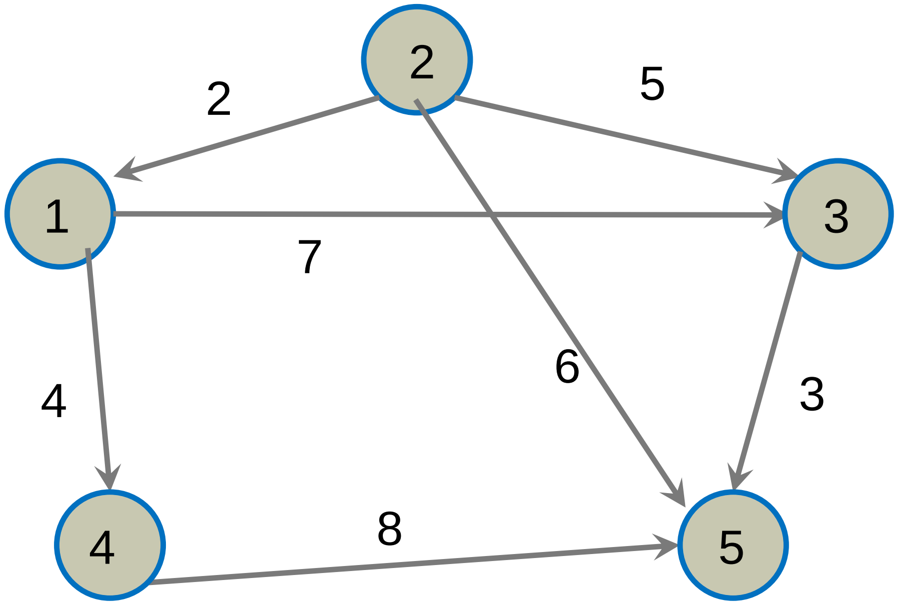
Directed Graph - Example



Undirected Weighted Graph - Example



Directed Weighted Graph - Example



Graphs – Formal notations

- A graph $G = (V, E)$ is defined using a set of vertices V and a set of edges E.
- An edge e is represented as $e = (u, v)$ where u and v are two vertices
- For undirected graphs, $(u, v) = (v, u)$ because there is no sense of direction.
- For a directed graph, (u, v) represents an edge **from** u **to** v and $(u, v) \neq (v, u)$.

Graphs – Formal notations

- A weighted graph is represented as G = (V, E, W) where W represents weights for the edges and each edge e is represented as (u, v, w) where w is the weight for the edge (u, v).

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

- In this unit, we focus on simple graphs.

Some Graph Properties

Let G be a graph.

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

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

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

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

Some Graph Properties

Let G be a graph.

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

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

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

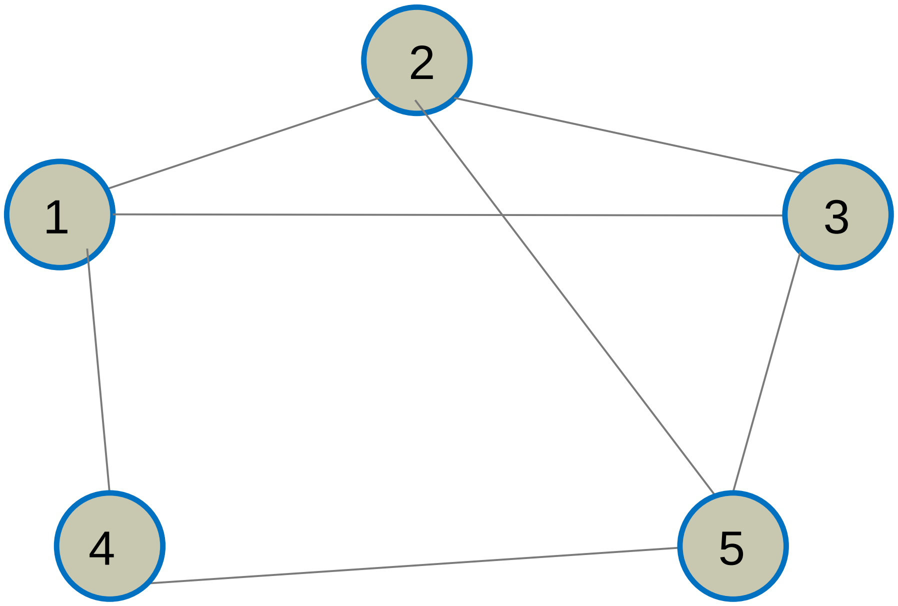
- A graph is called **sparse** if $E \ll V^2$ (<< means significantly smaller than)
- A graph is called **dense** if $E \approx V^2$

Representing Graphs

Adjacency Matrix:

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

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

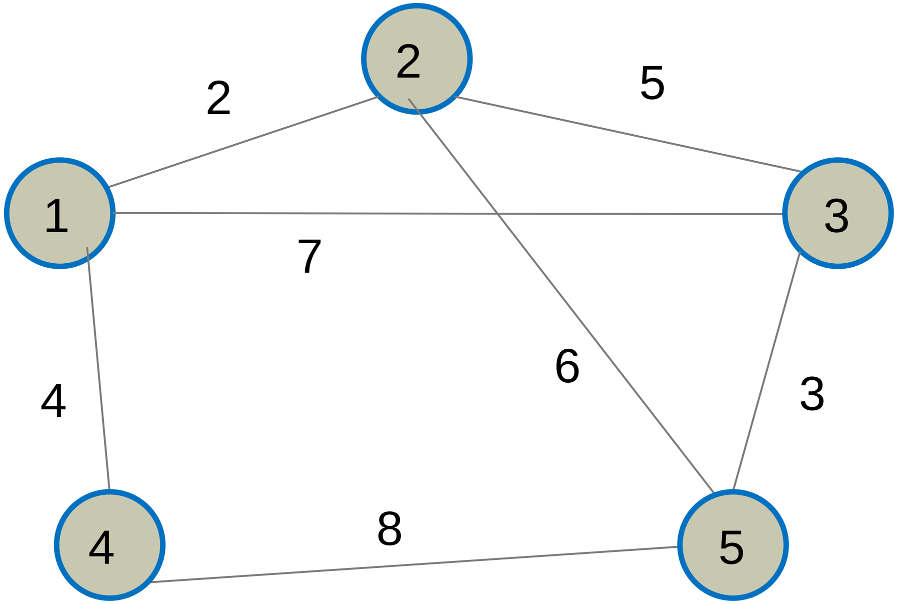


Representing Graphs

Adjacency Matrix:

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

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



Representing Graphs

Adjacency Matrix:

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

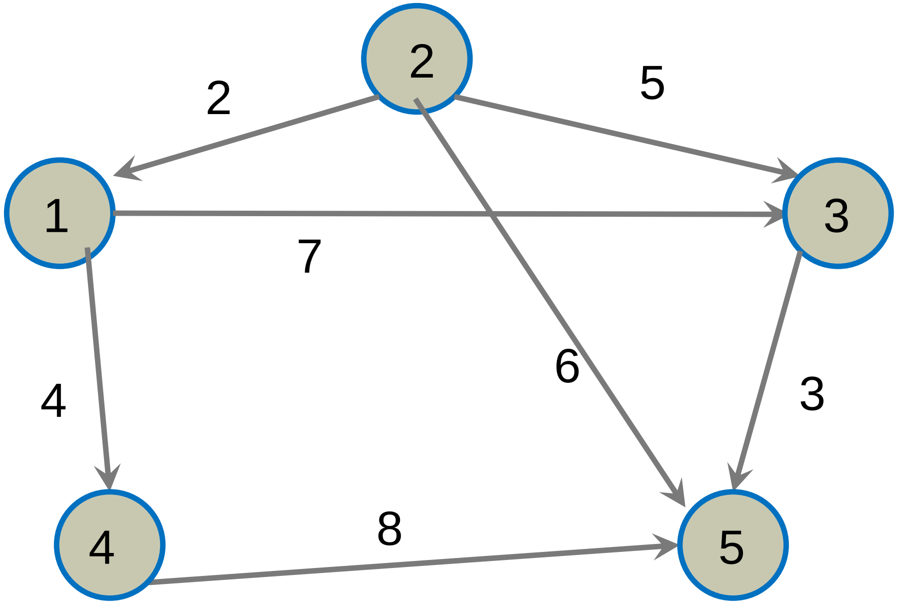
Space Complexity: $O(V^2)$ regardless of the number of edges

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

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

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

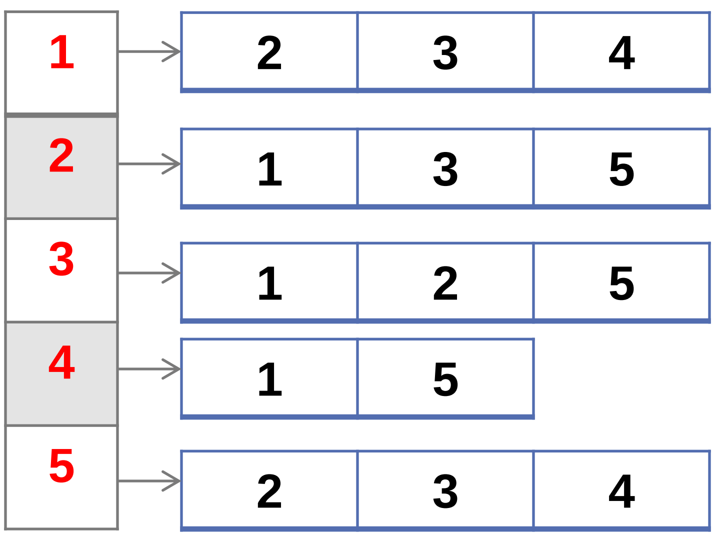
	1	**2**	**3**	**4**	**5**
1			7	4	
2	2		5		6
3					3
4					8
5					

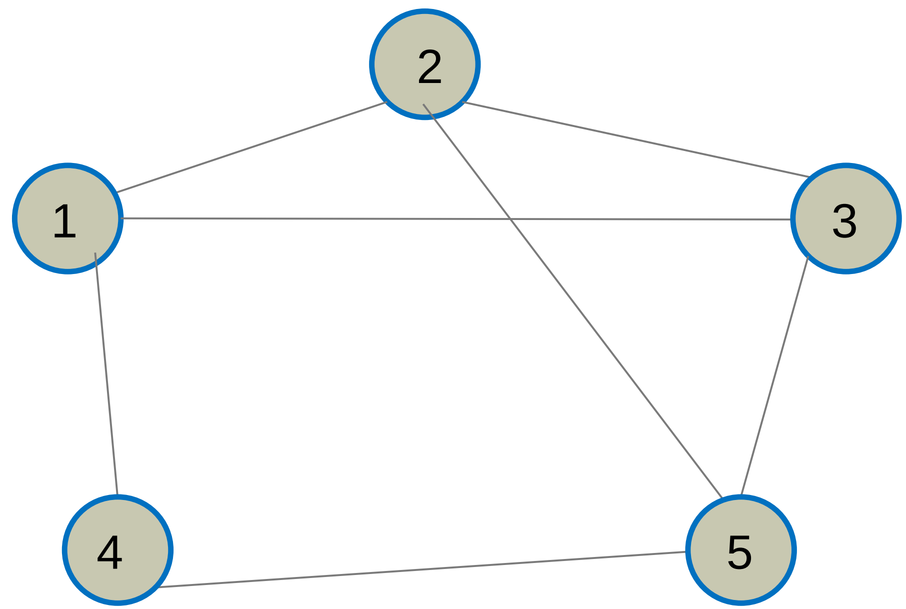


Representing Graphs

Adjacency List:

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





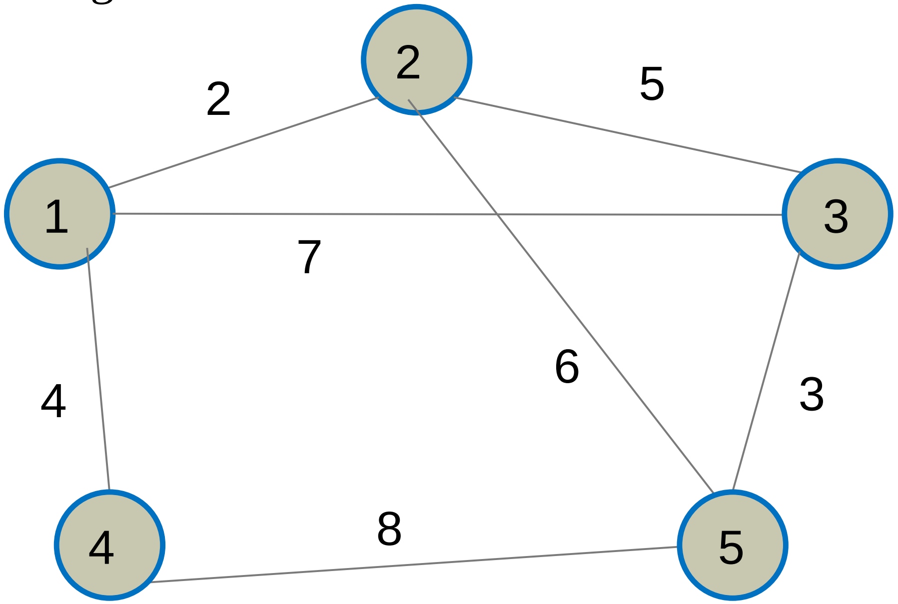
Representing Graphs

Adjacency List:

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

The numbers in parenthesis correspond to the weights.

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:

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

Space Complexity:

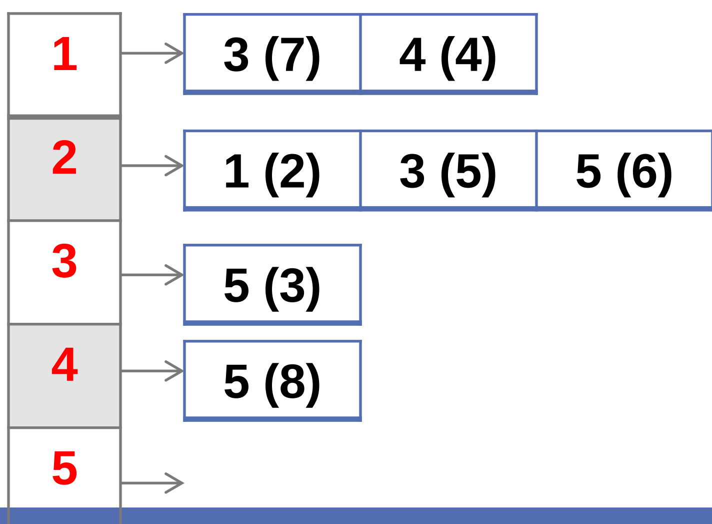
- O(V + E)

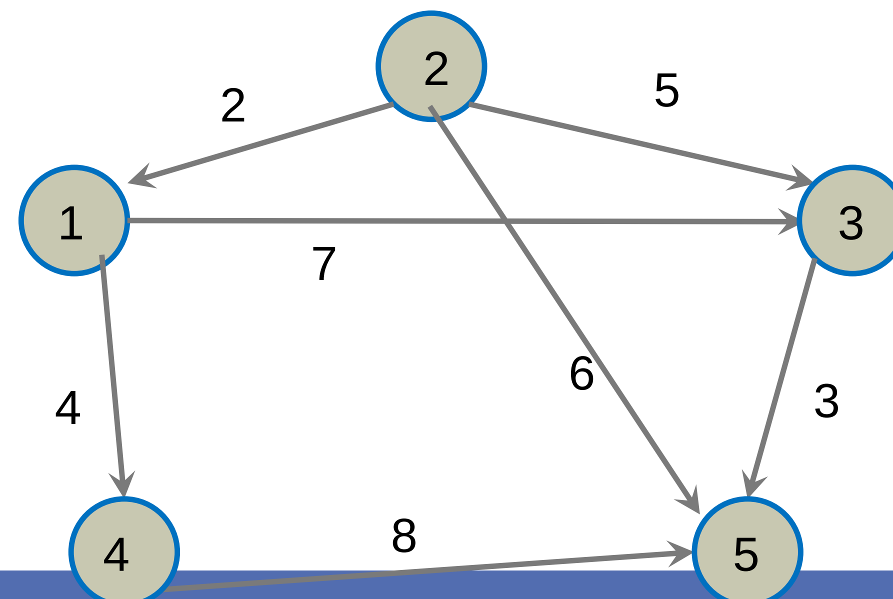
Time complexity of checking if a particular edge exists:

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

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

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





Outline

1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. 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) the nodes of a graph starting from a source vertex.

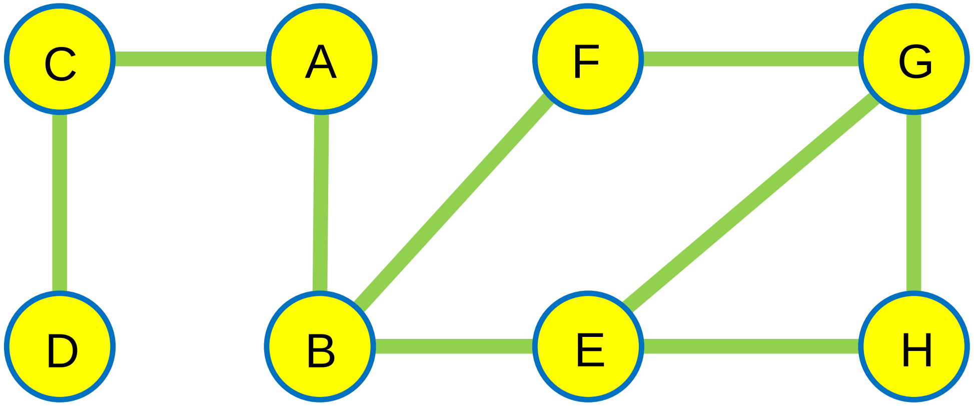
We will look into two algorithms:

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

Both of them visit all the vertices of the graph exactly once

The order in which they visit vertices is different

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

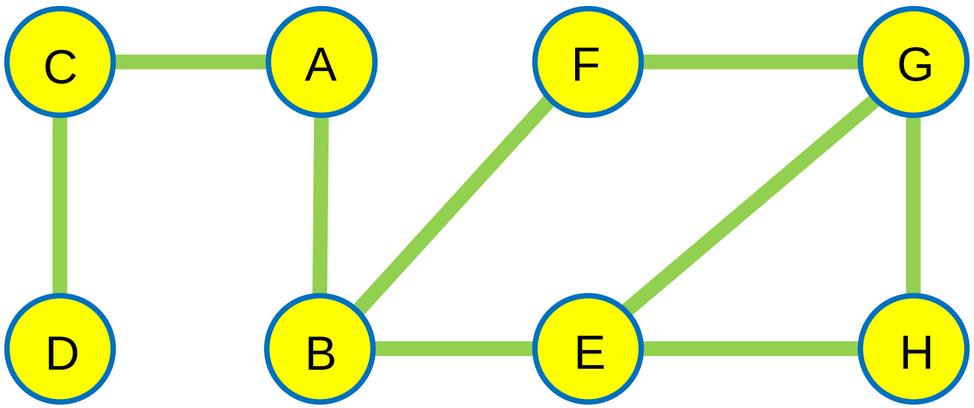


Graph Traversal - BFS

- 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 - RFIBMB

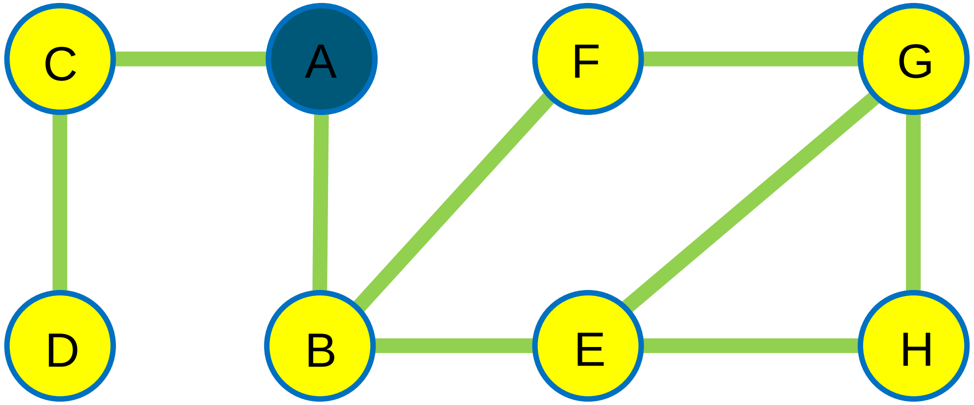


Graph Traversal - BFS

- 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!

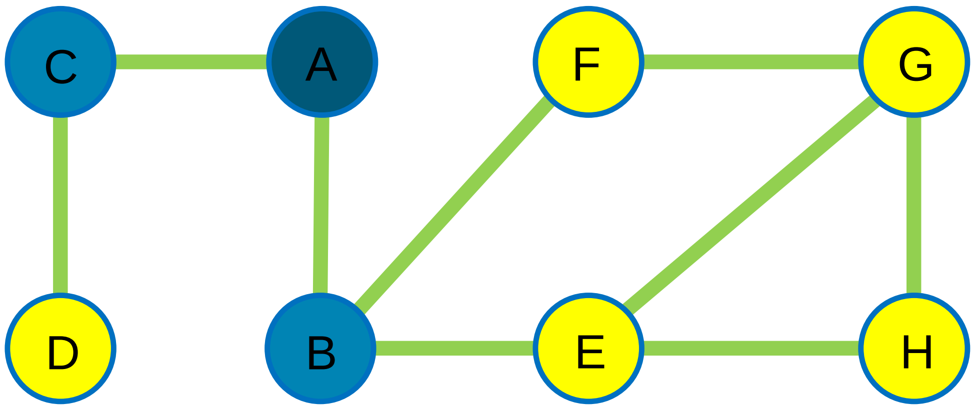


Graph Traversal - BFS

- 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!

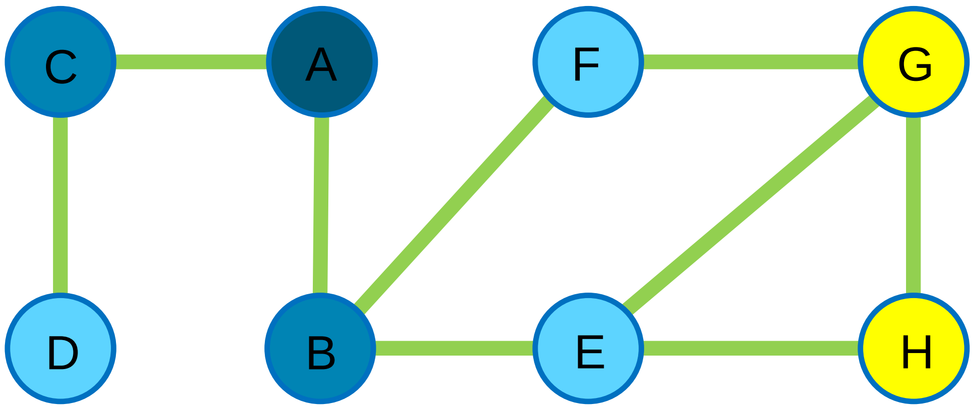


Graph Traversal - BFS

- 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!

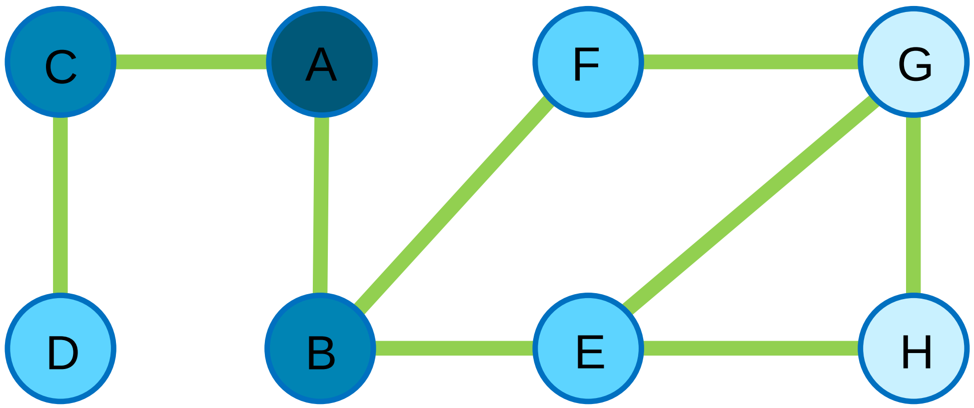


Graph Traversal - BFS

- 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!

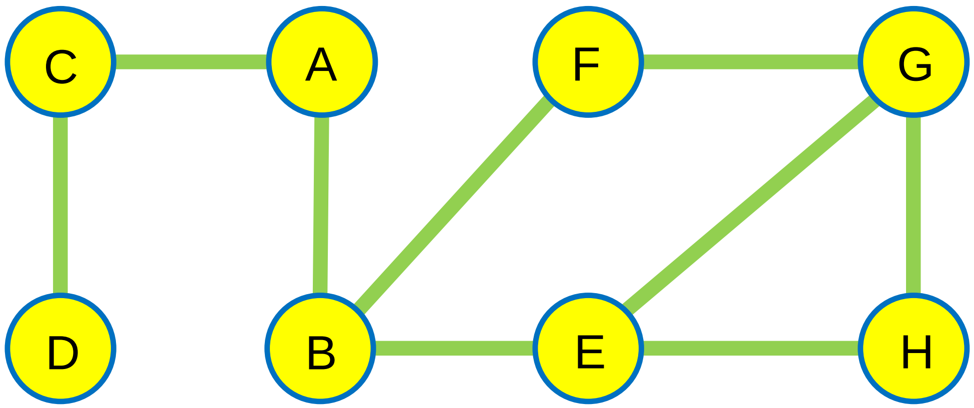


Graph Traversal - DFS

- Depth First Search (DFS)
  - Traverses the graph as deeply as possible before backtracking and traversing other nodes
  - In the 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?

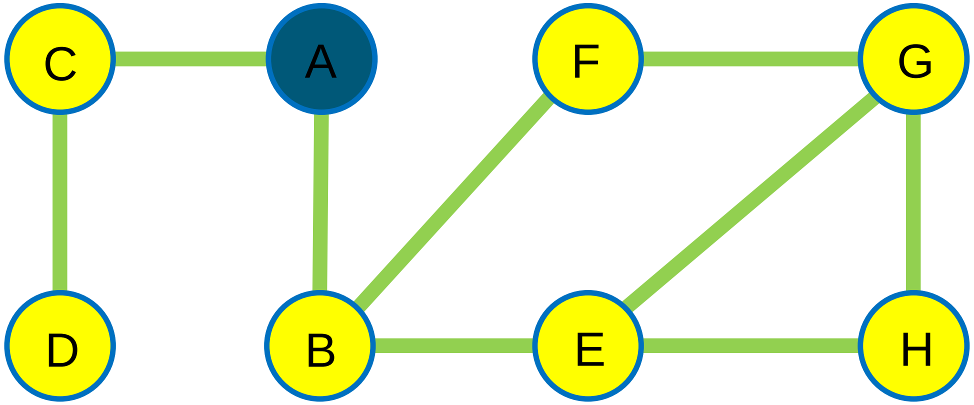
Quiz time!
https://flux.qa - RFIBMB



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

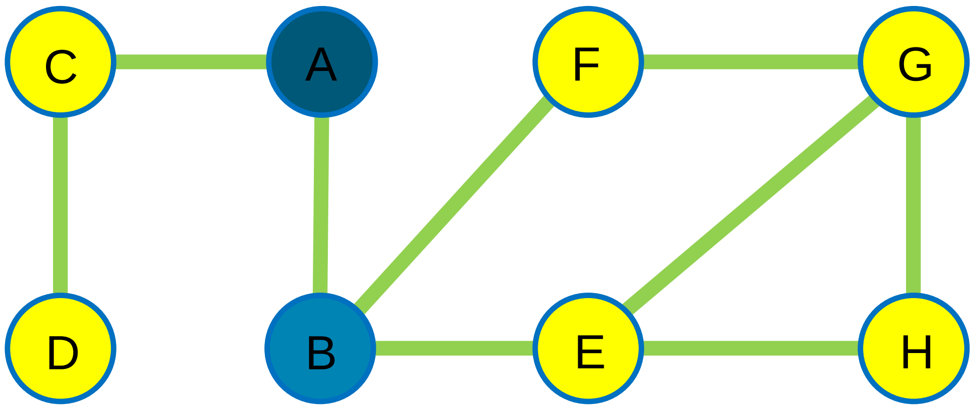
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

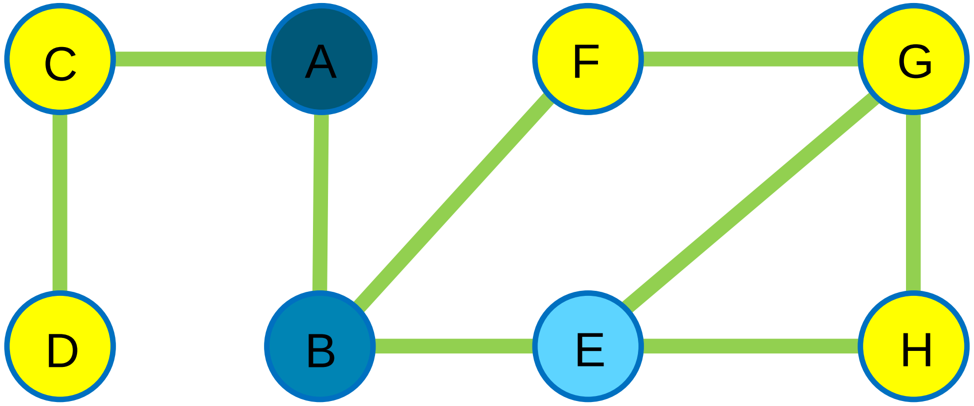
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

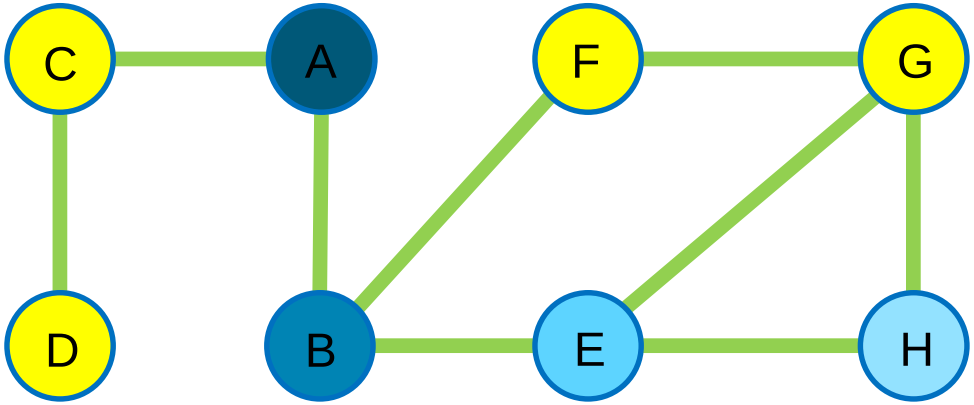
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

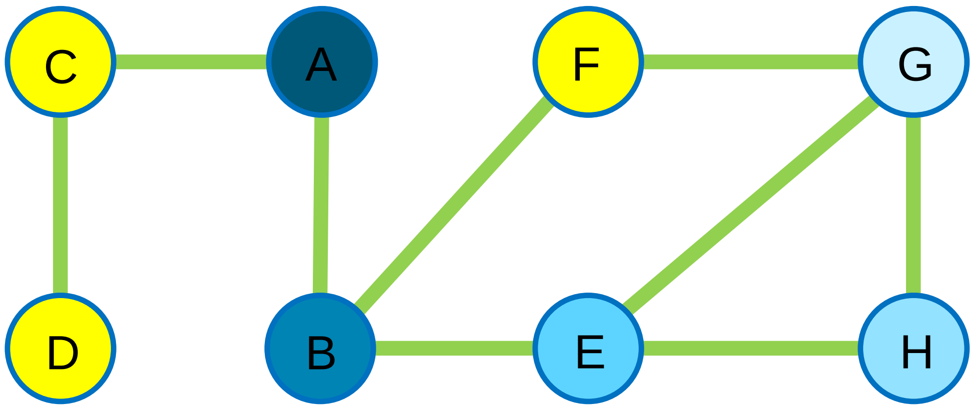
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

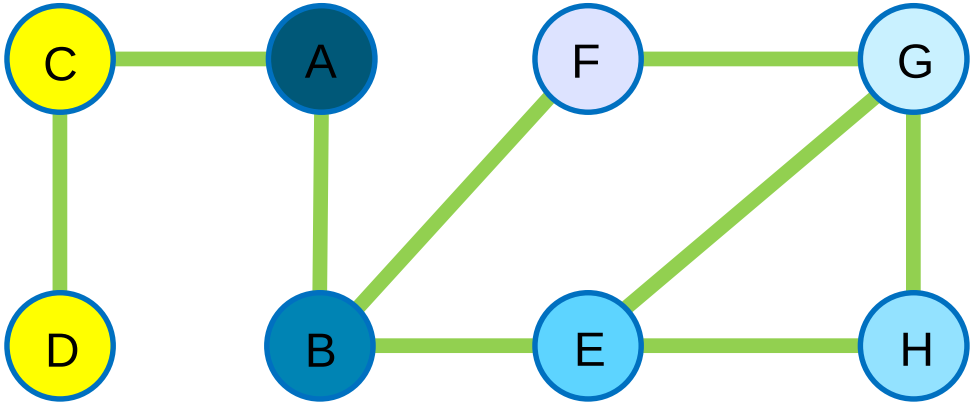
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

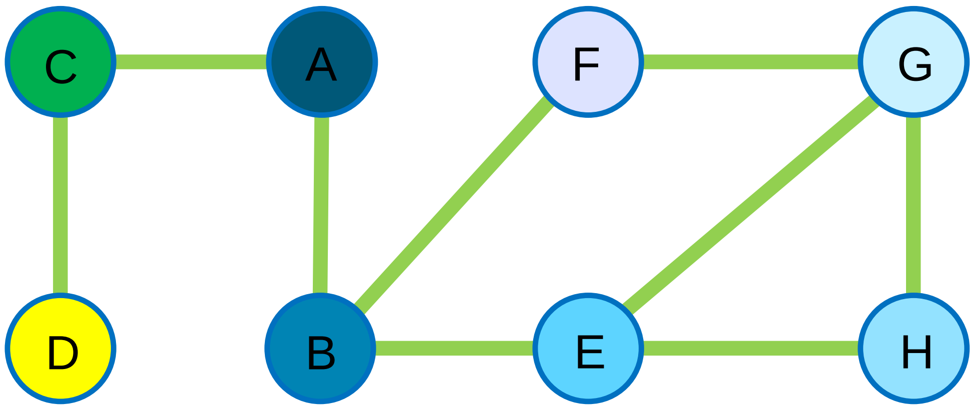
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

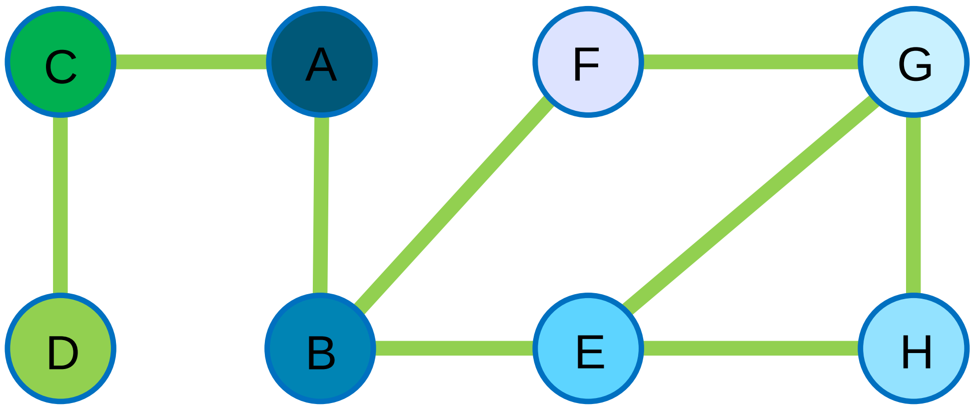
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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

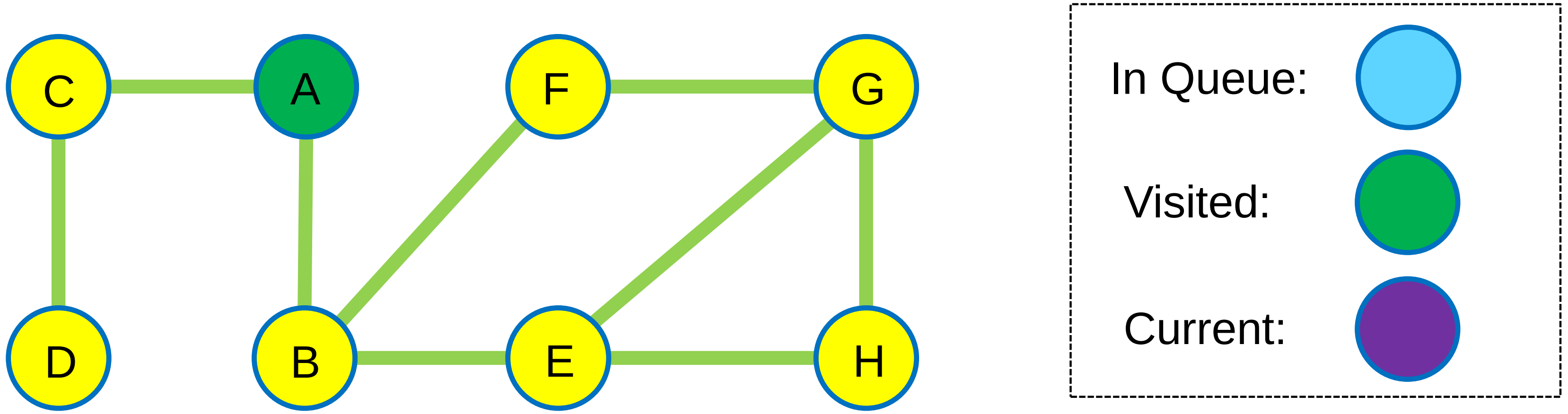
Is A, B, E, H, F, G, C, D as possible DFS order?
No!



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)



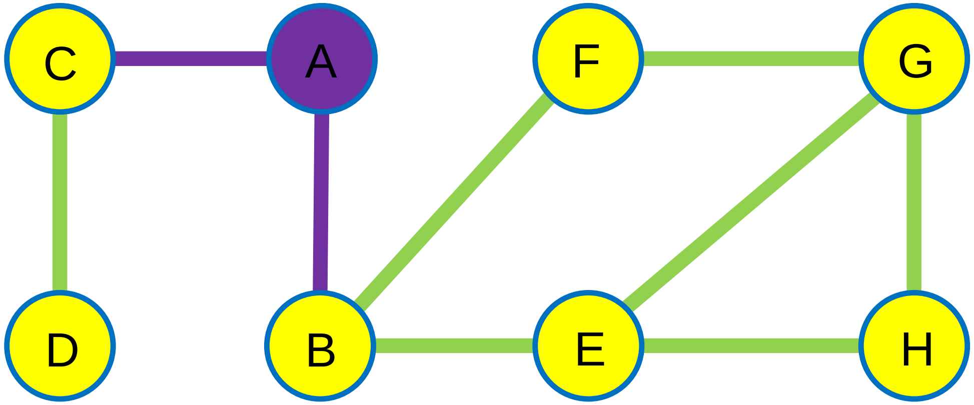
Current:

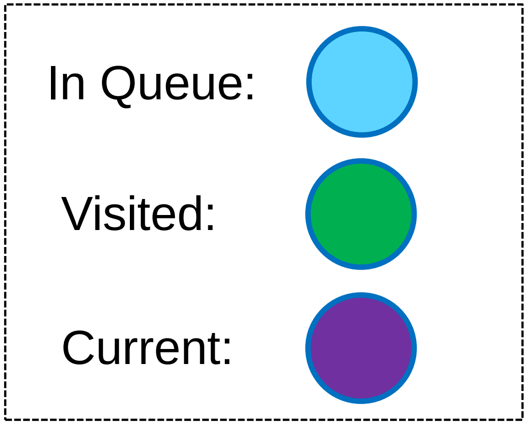
Queue: A

Visited: A

Note! The visualisation of **discovered** and **visited** does not correspond to implementation. See complexity discussion for optimised implementation

Breadth First Search (BFS)



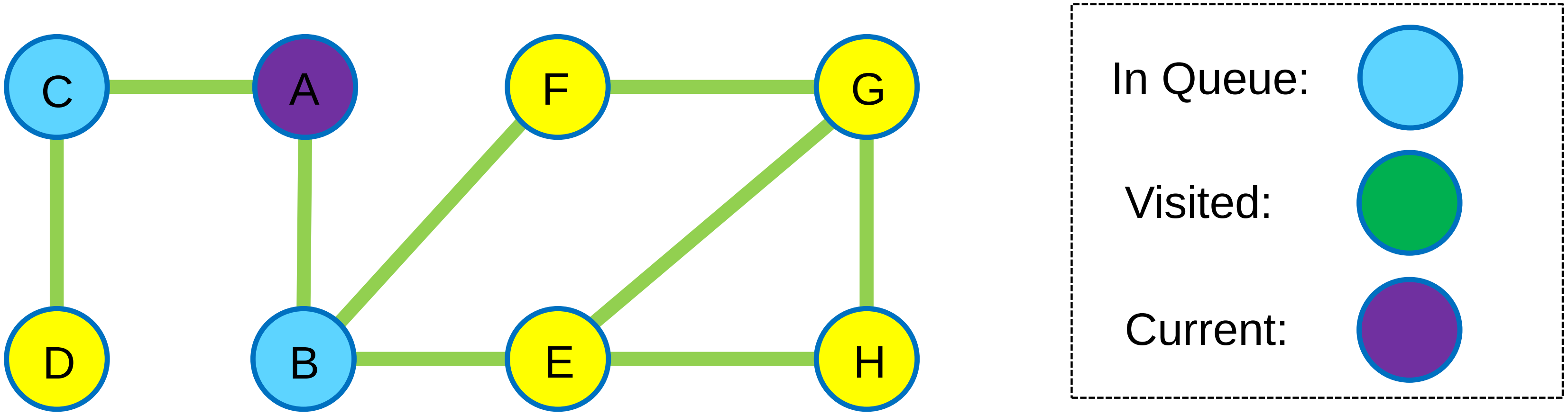


Current: A

Queue:

Visited: A

Breadth First Search (BFS)

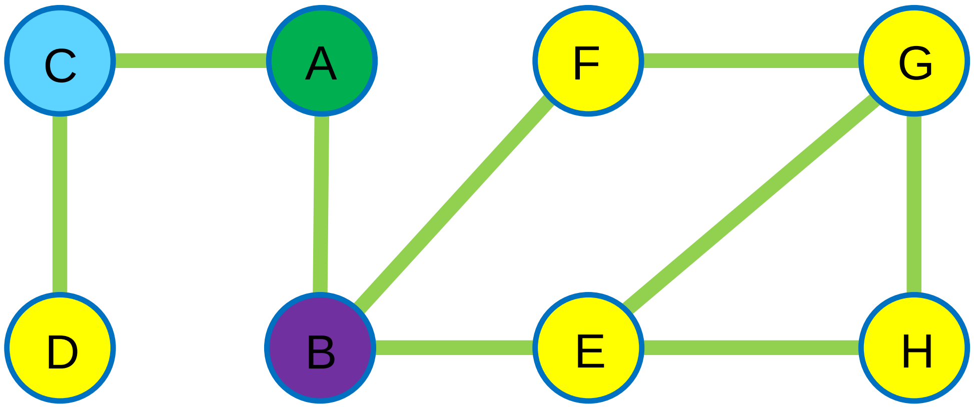


Current: A

Queue: B | C

Visited: A

Breadth First Search (BFS)

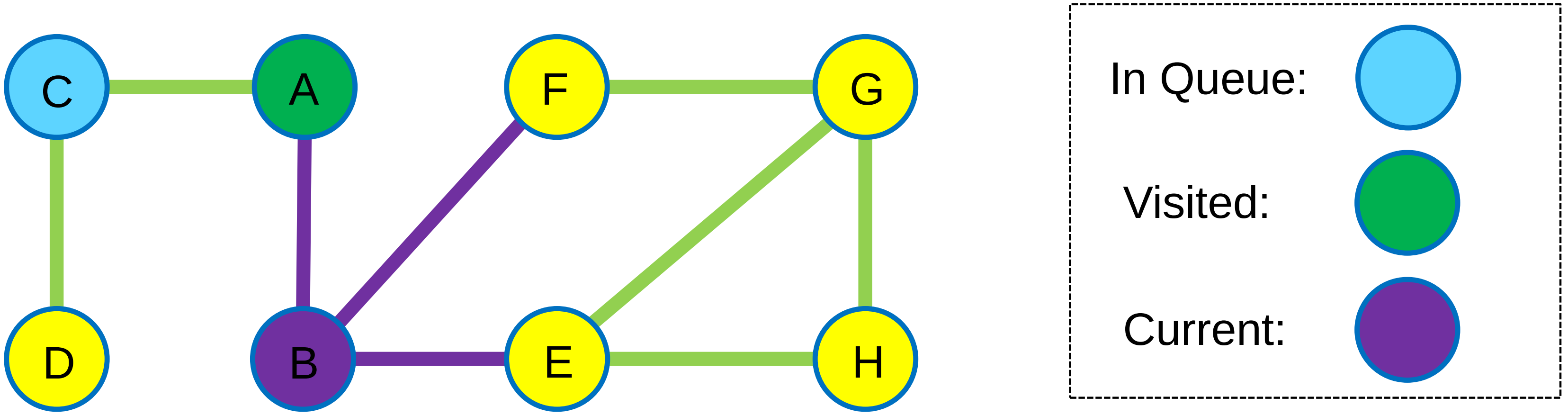


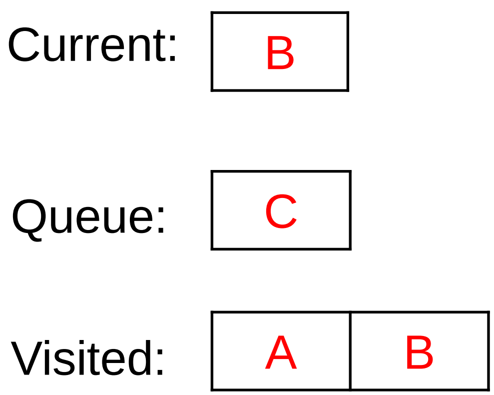
Current: B

Queue: C

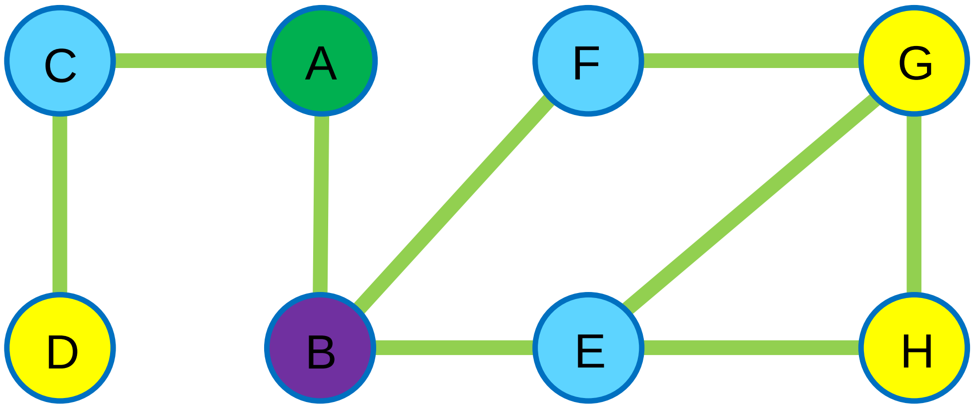
Visited: A B

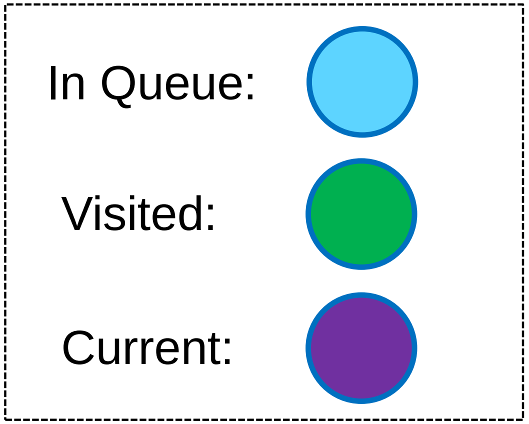
Breadth First Search (BFS)





Breadth First Search (BFS)



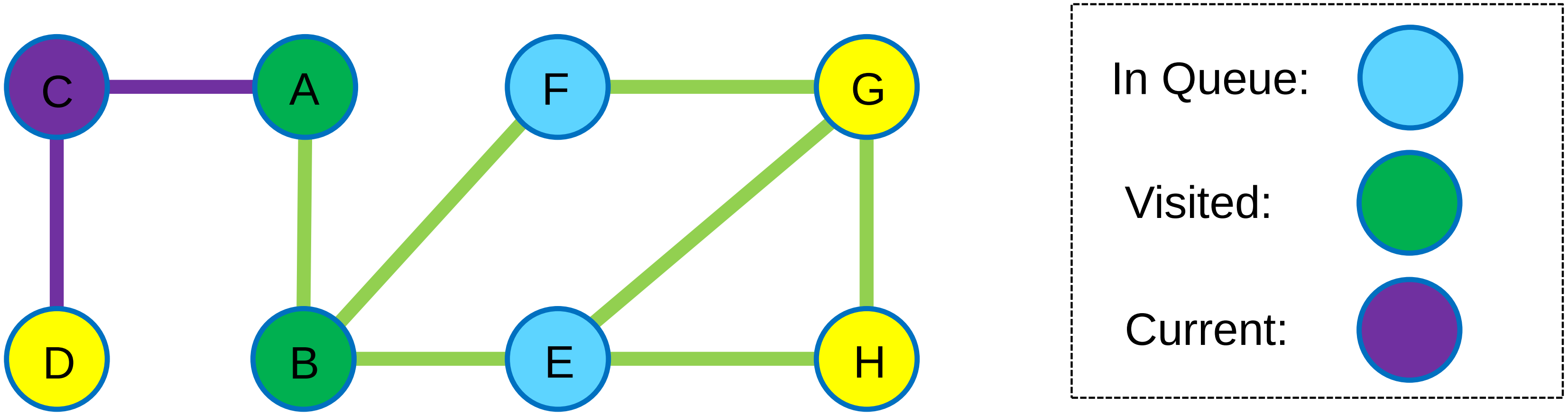


Current: B

Queue: C | E | F

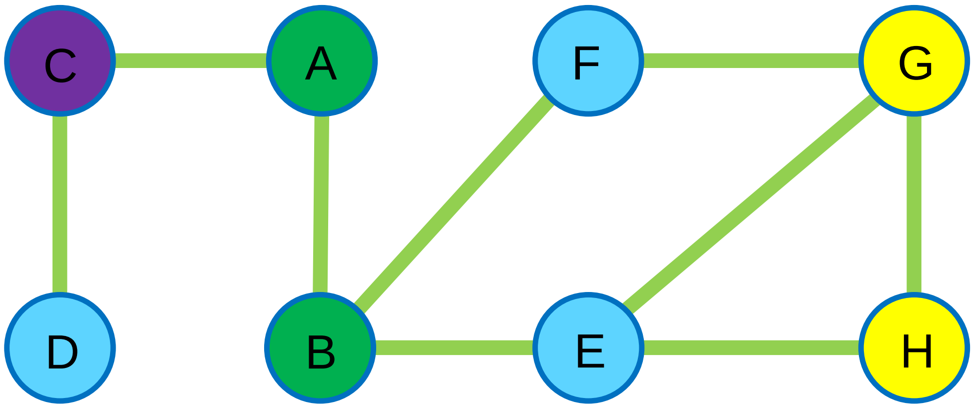
Visited: A | B

Breadth First Search (BFS)



Current:	C		
Queue:	E	F	
Visited:	A	B	C

Breadth First Search (BFS)

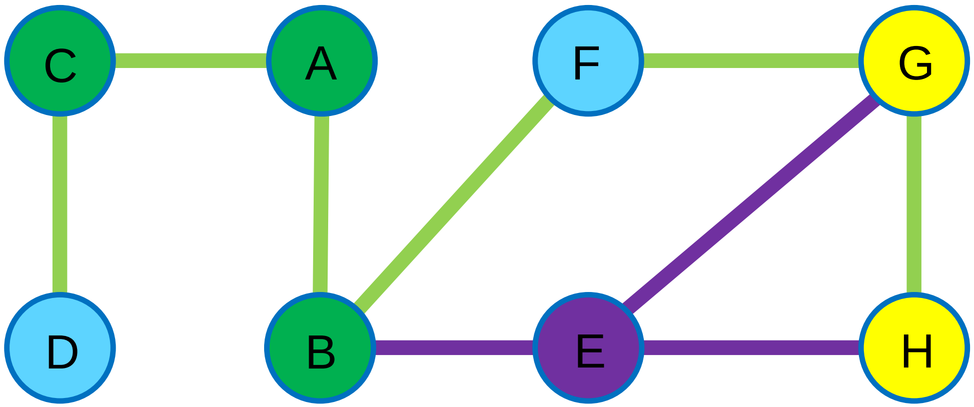


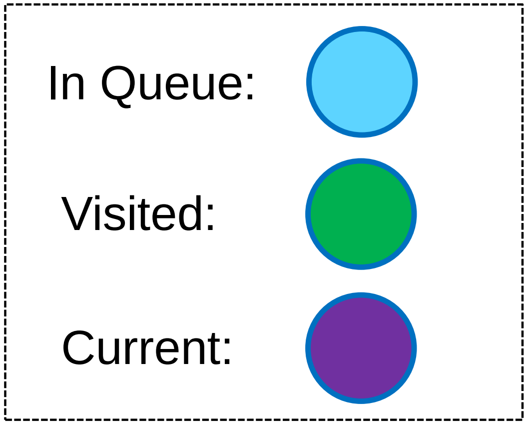
Current: C

Queue: E F D

Visited: A B C

Breadth First Search (BFS)





Current:

E

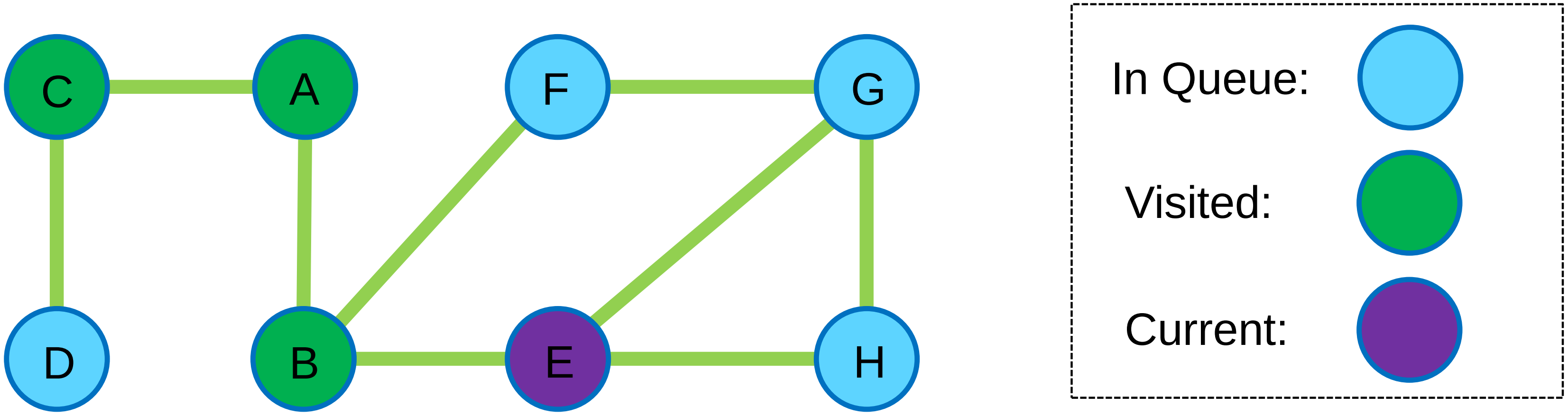
Queue:

F	D

Visited:

A	B	C	E

Breadth First Search (BFS)



Current: E

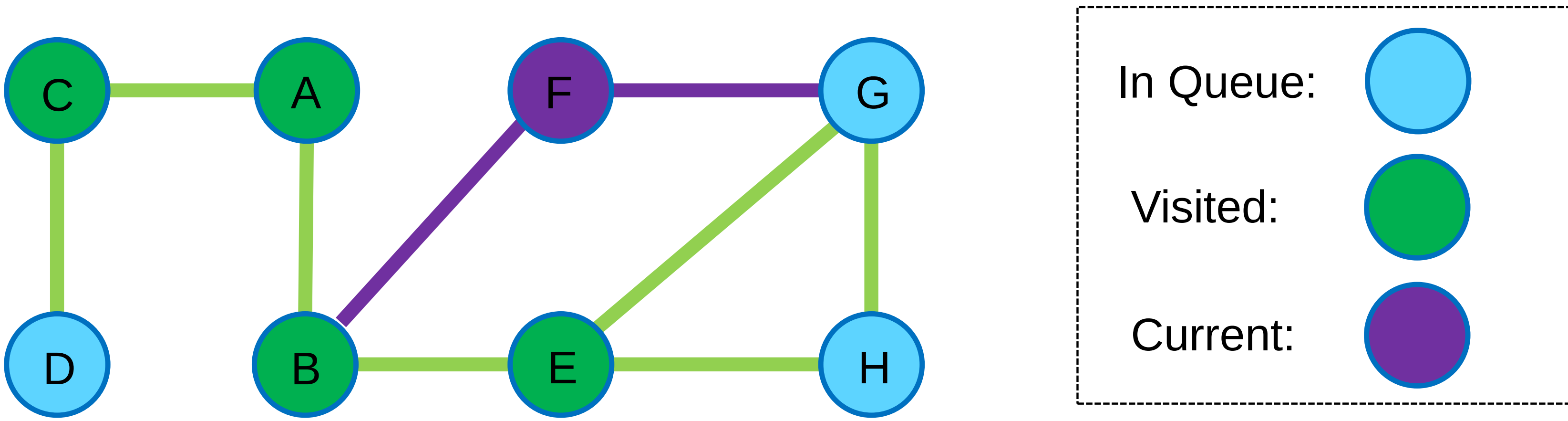
Queue:

F	D	G	H

Visited:

A	B	C	E

Breadth First Search (BFS)

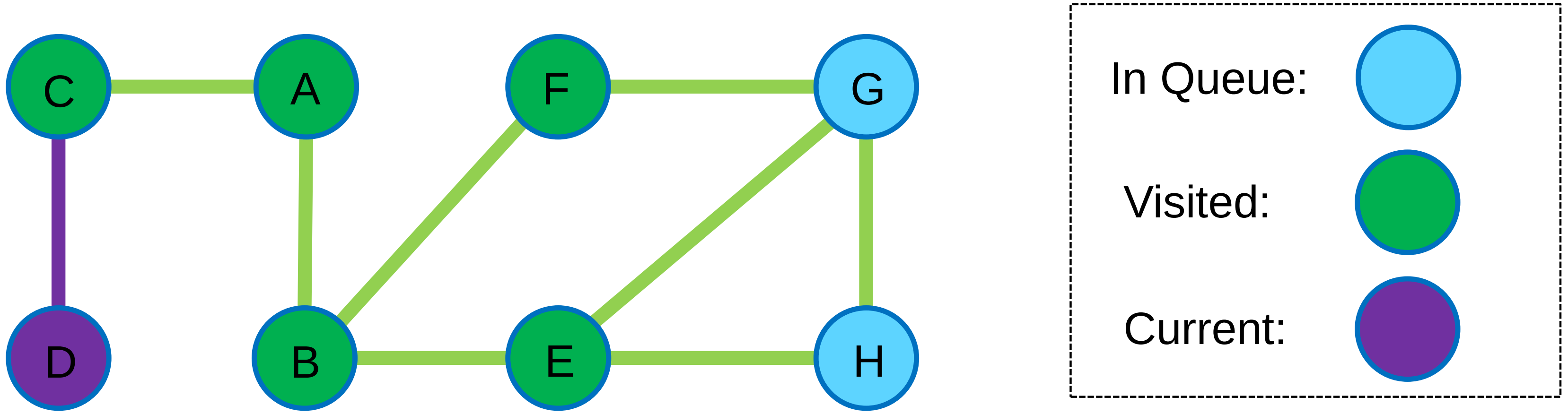


Current: F

Queue: D | G | H

Visited: A | B | C | E | F

Breadth First Search (BFS)

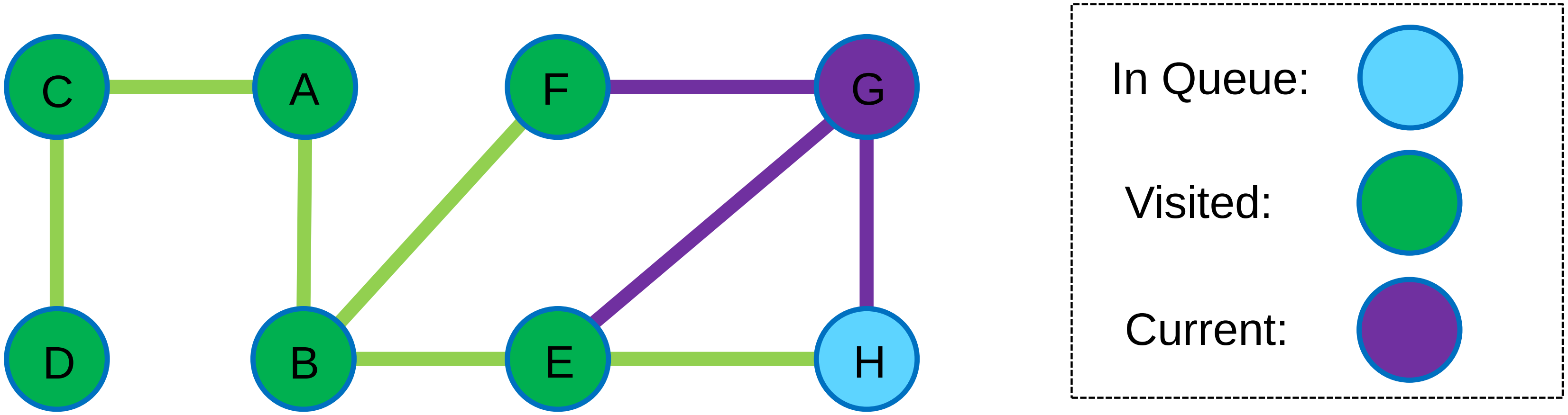


Current: D

Queue: G H

Visited: A B C E F D

Breadth First Search (BFS)



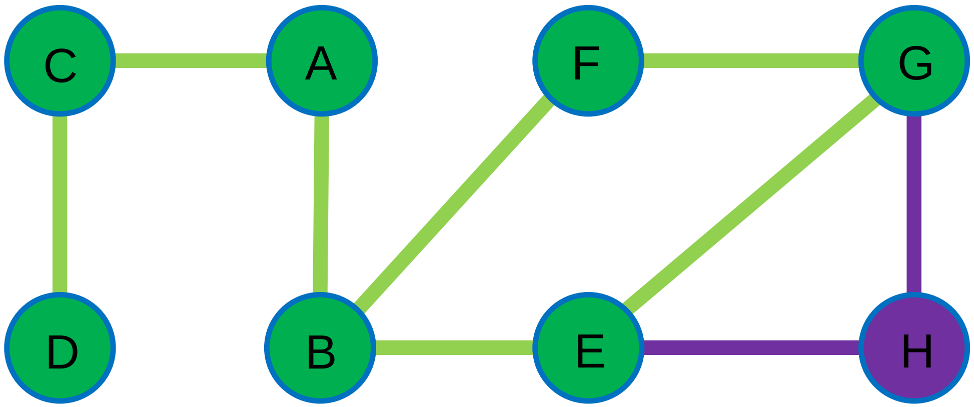
Current: G

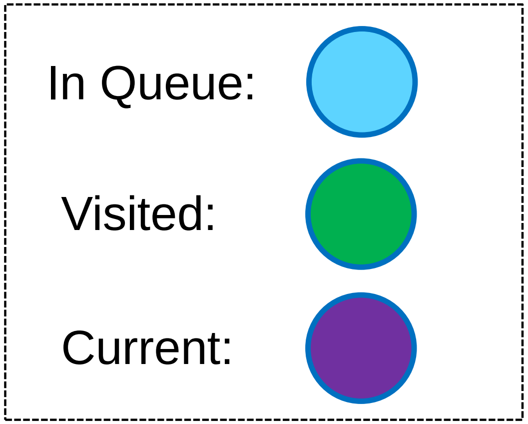
Queue: H

Visited:

A	B	C	E	F	D	G

Breadth First Search (BFS)





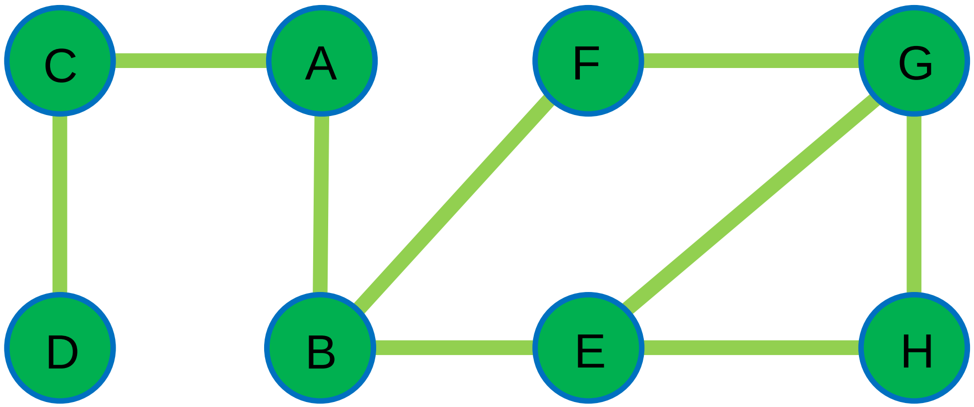
Current: H

Queue:

Visited:

A	B	C	E	F	D	G	H

Breadth First Search (BFS)



Current:

Queue:

Visited:

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 - RFIBMB

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 it 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

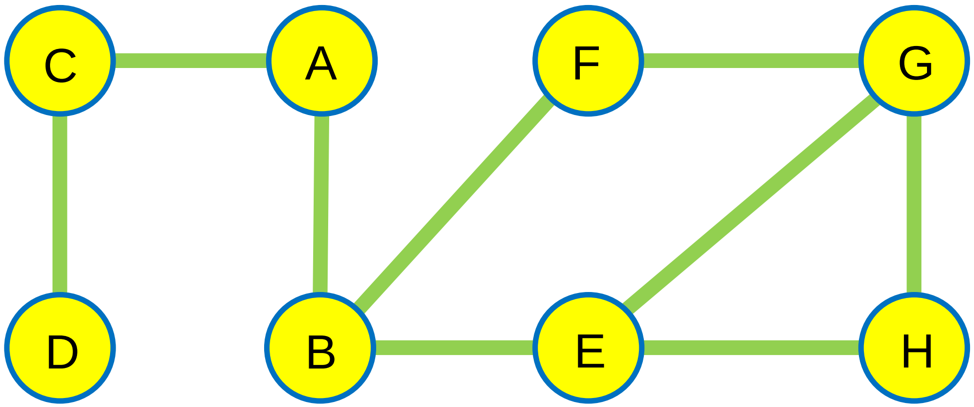
1. Introduction to Graphs
2. Graph Traversal Algorithms
   A. The idea
   B. Breadth First Search (BFS)
   C. Depth First Search (DFS)
   D. Applications
3. Shortest Path Problem
   A. Breadth First Search (for unweighted graphs)
   B. Dijkstra’s algorithm (for weighted graphs with non-negative weights)

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

- BFS should also use a bit list, the example above was just for intuition!

Depth First Search (DFS)



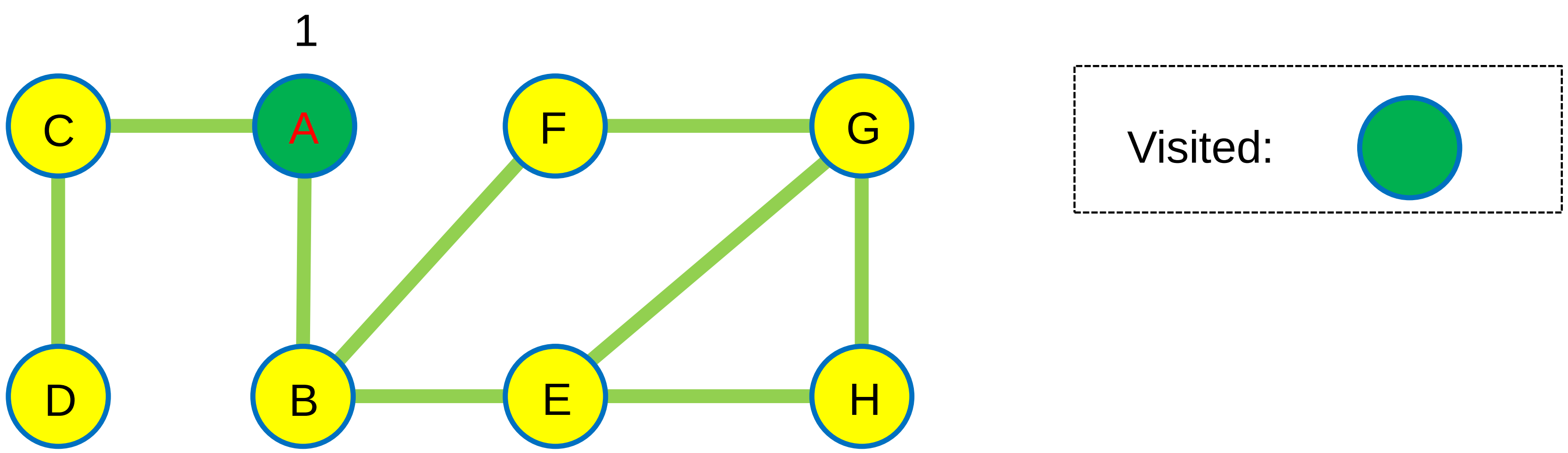
Current: **A**

Visited is indexed by vertex ID. Normally, the IDs are integers from from 0 to V-1 (to allows 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)

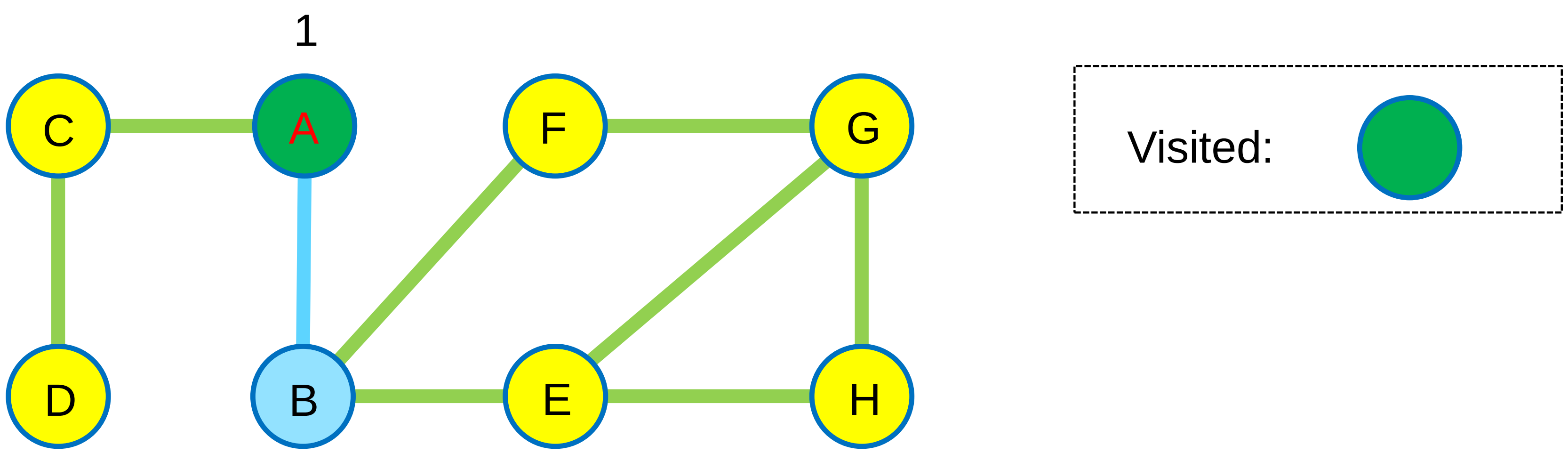


Current: **A**

Visited:

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

Depth First Search (DFS)

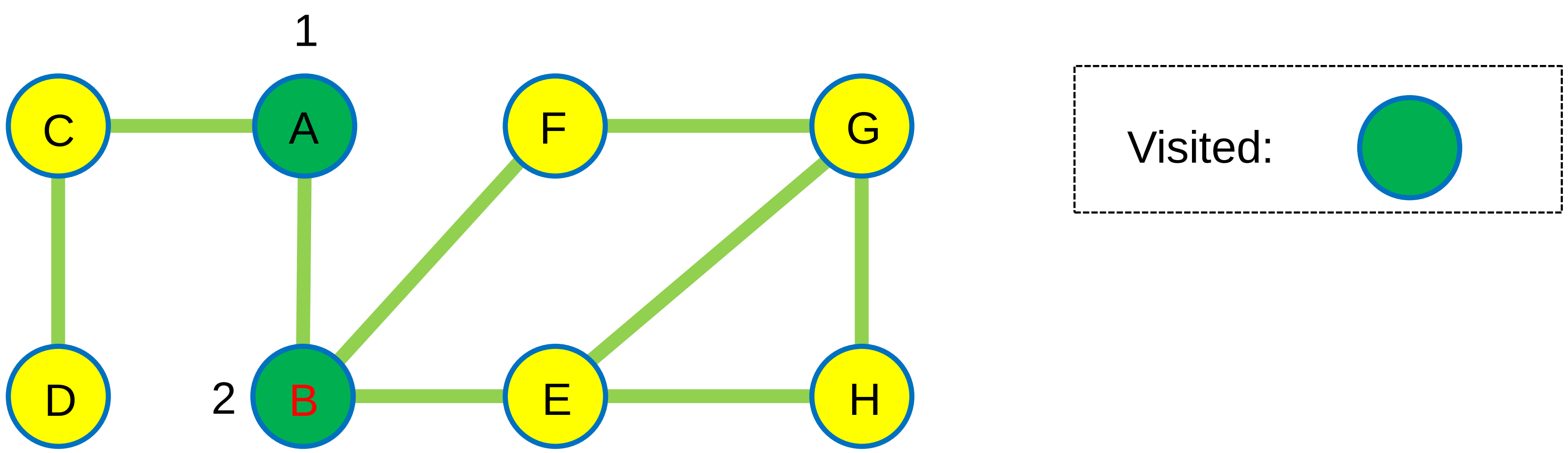


Current: **A**

Visited:

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

Depth First Search (DFS)

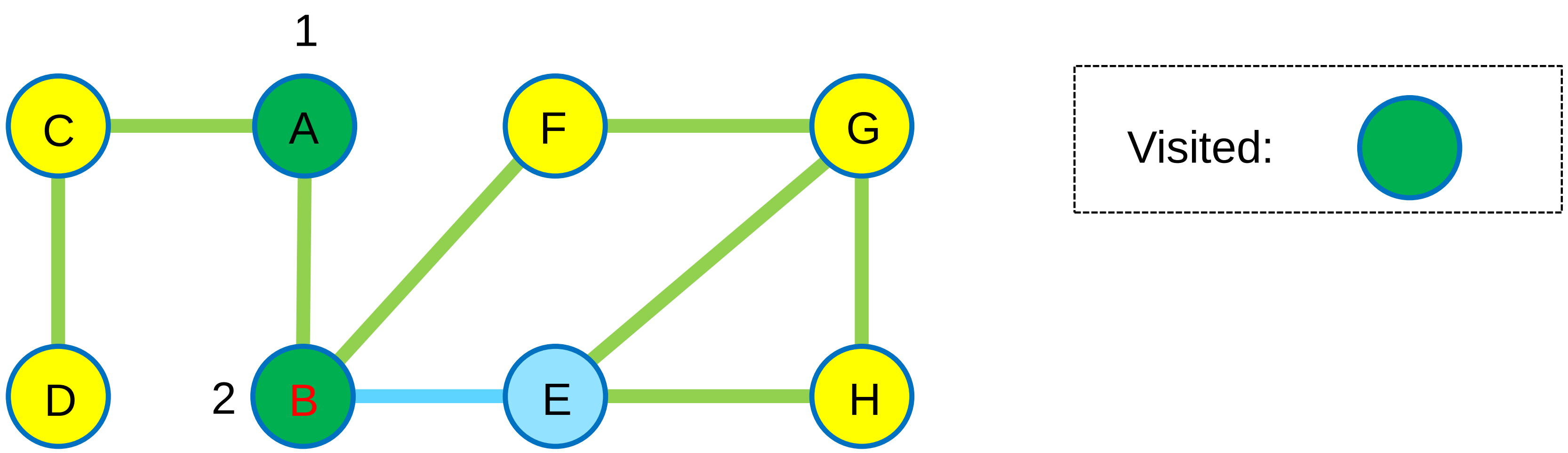


Current: **B**

Visited:

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

Depth First Search (DFS)

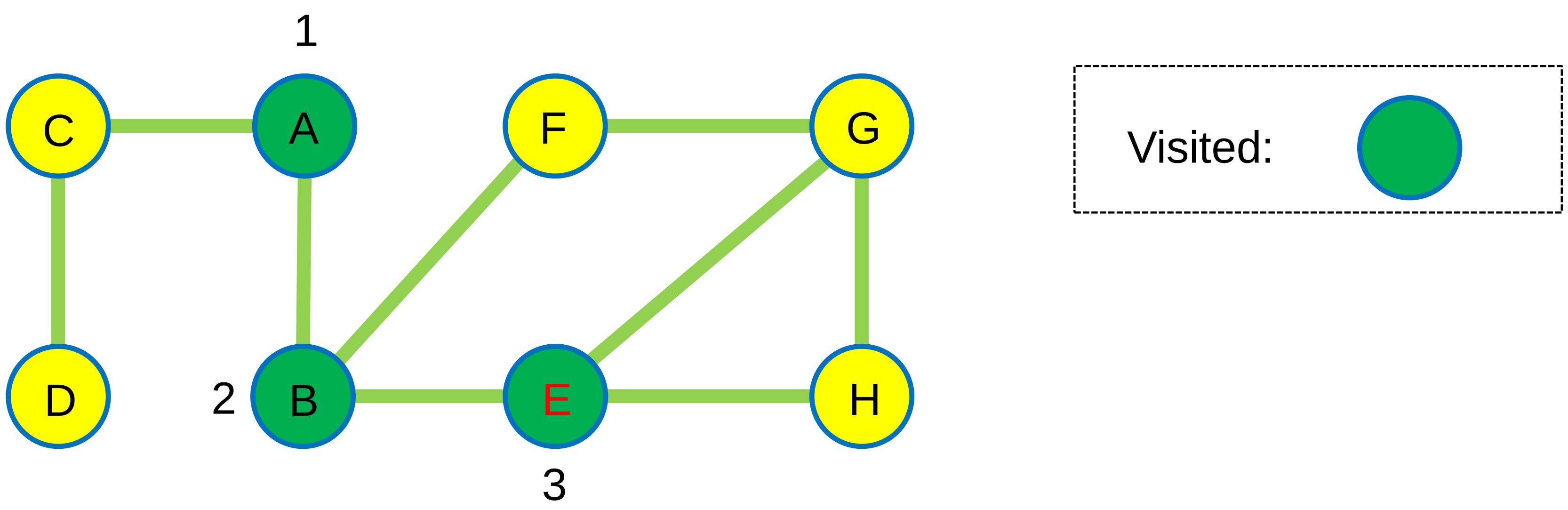


Current: **B**

Visited:

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

Depth First Search (DFS)

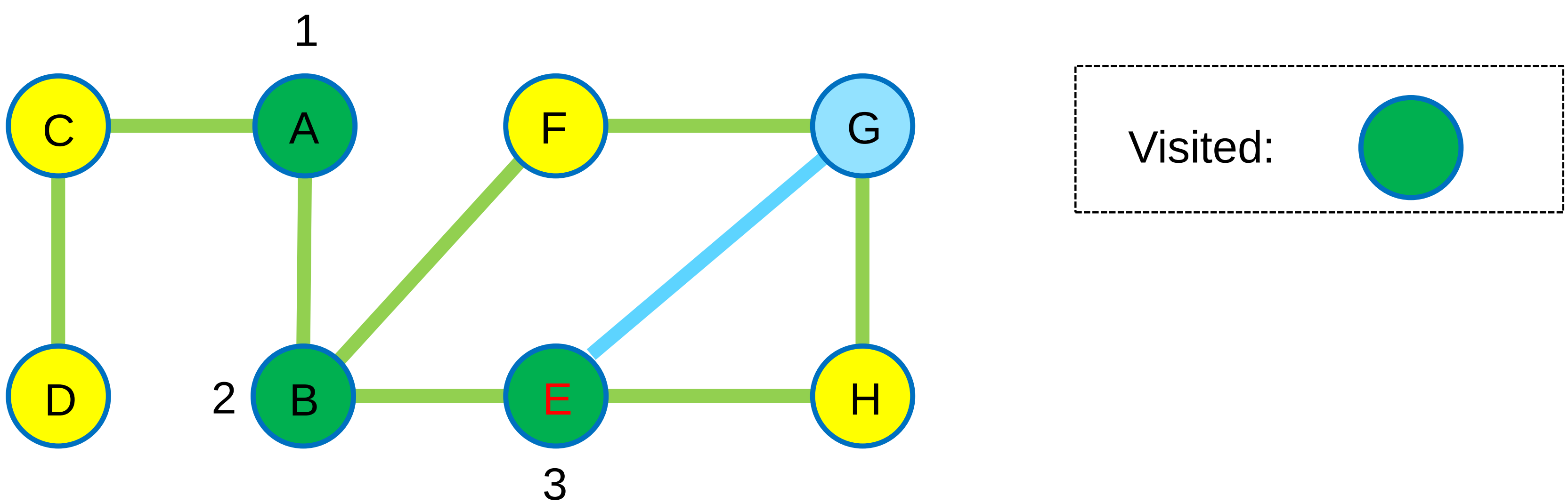


Current: **E**

Visited:

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

Depth First Search (DFS)

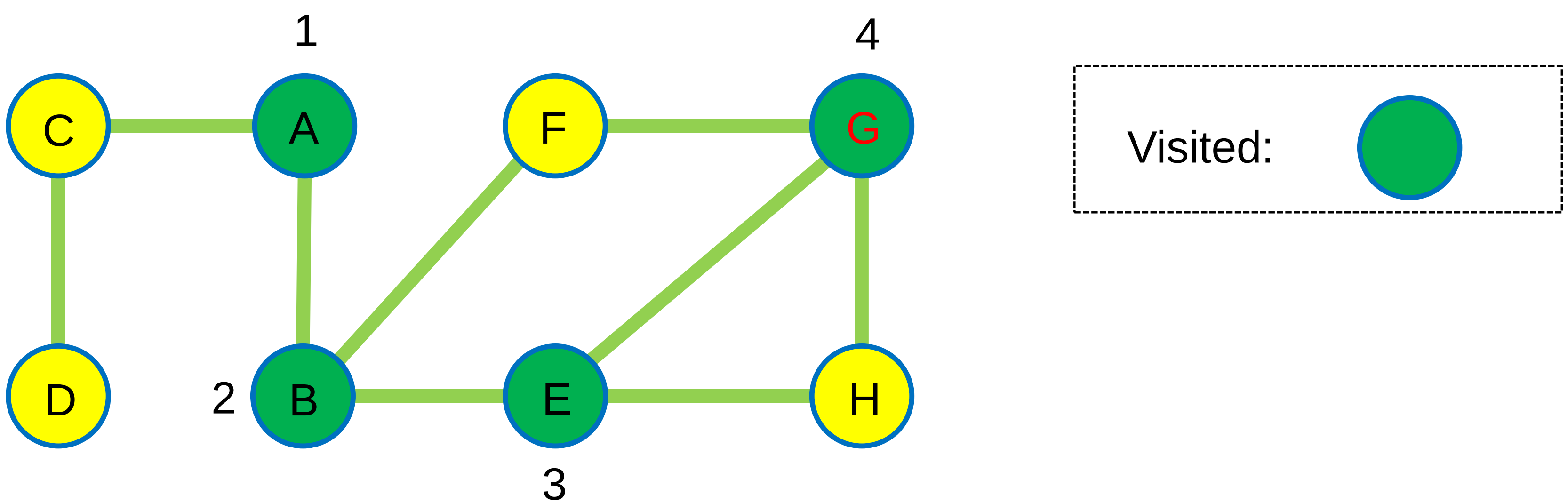


Current: **E**

Visited:

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

Depth First Search (DFS)

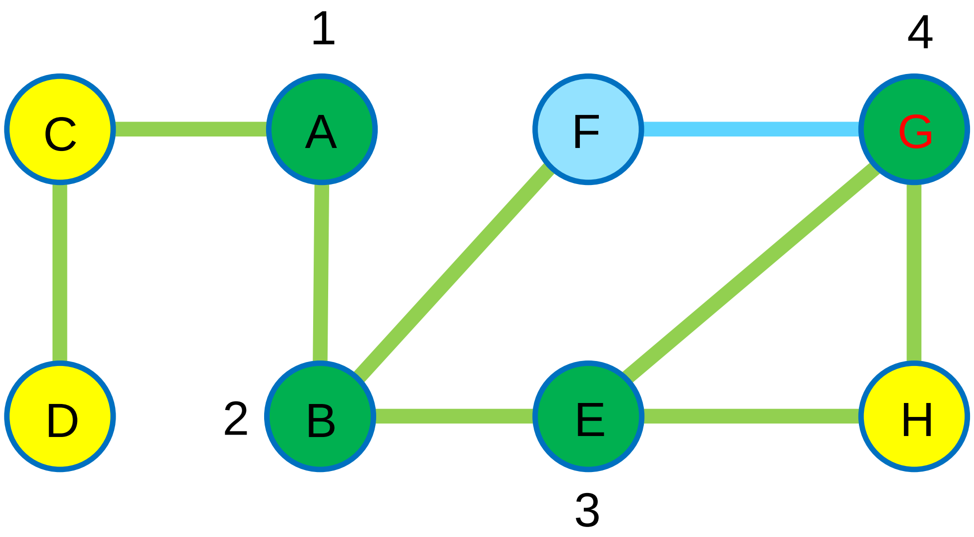


Current: **G**

Visited:

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

Depth First Search (DFS)

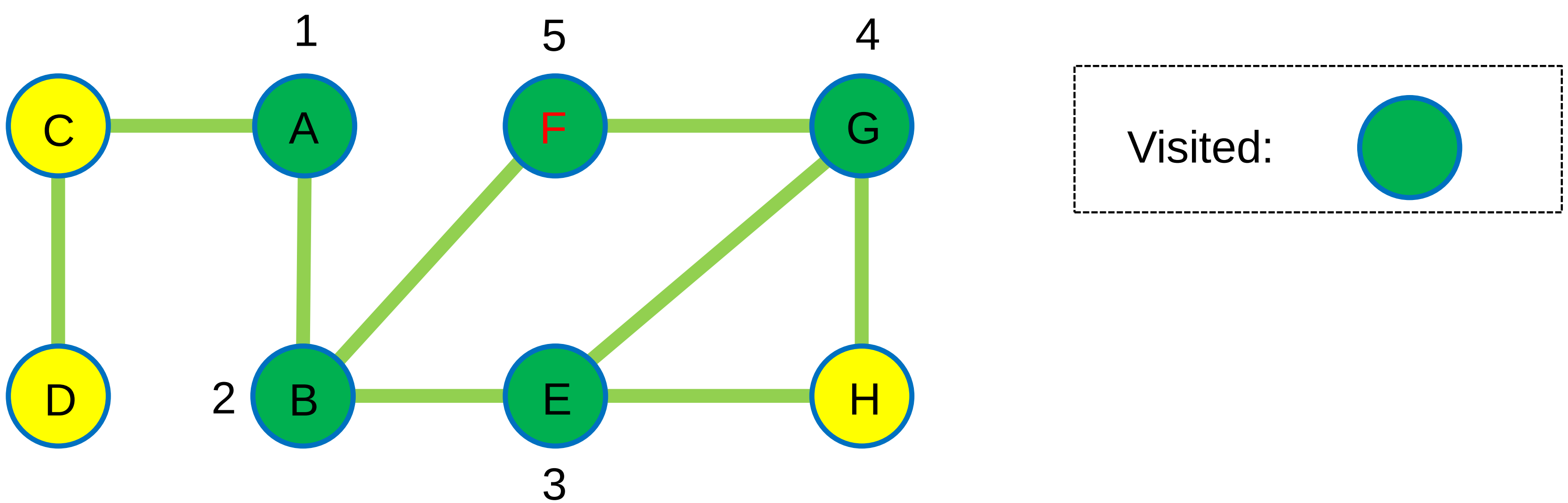


Visited:

Current: **G**

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

Depth First Search (DFS)

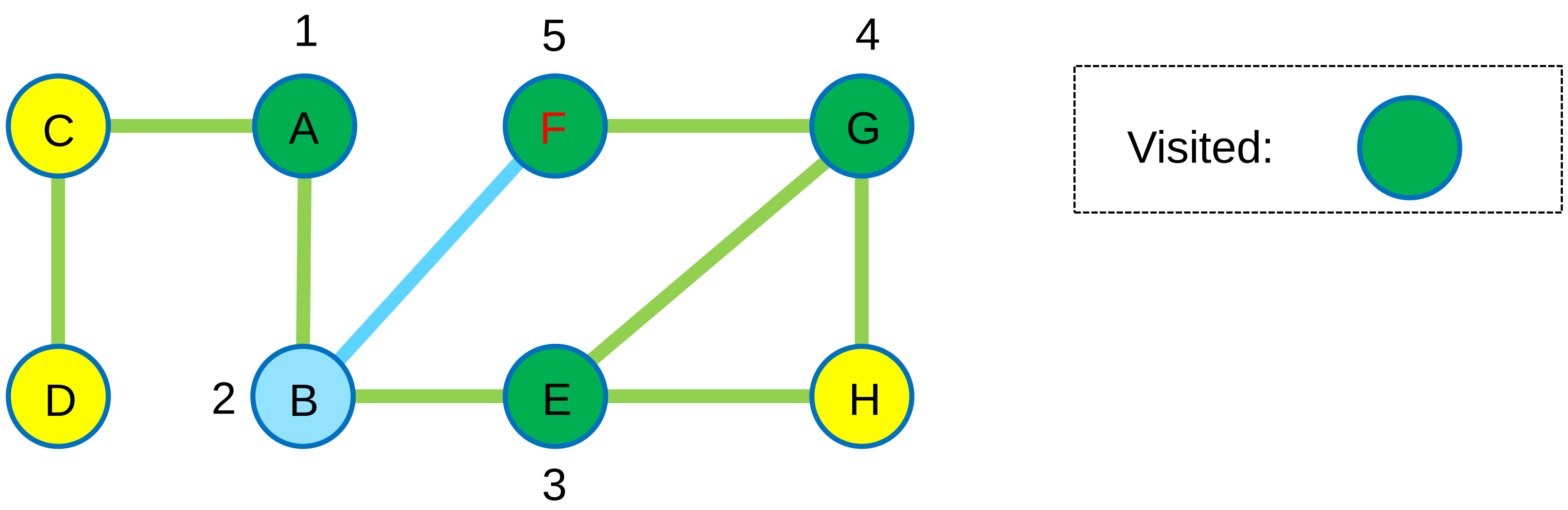


Current: **F**

Visited:

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

Depth First Search (DFS)

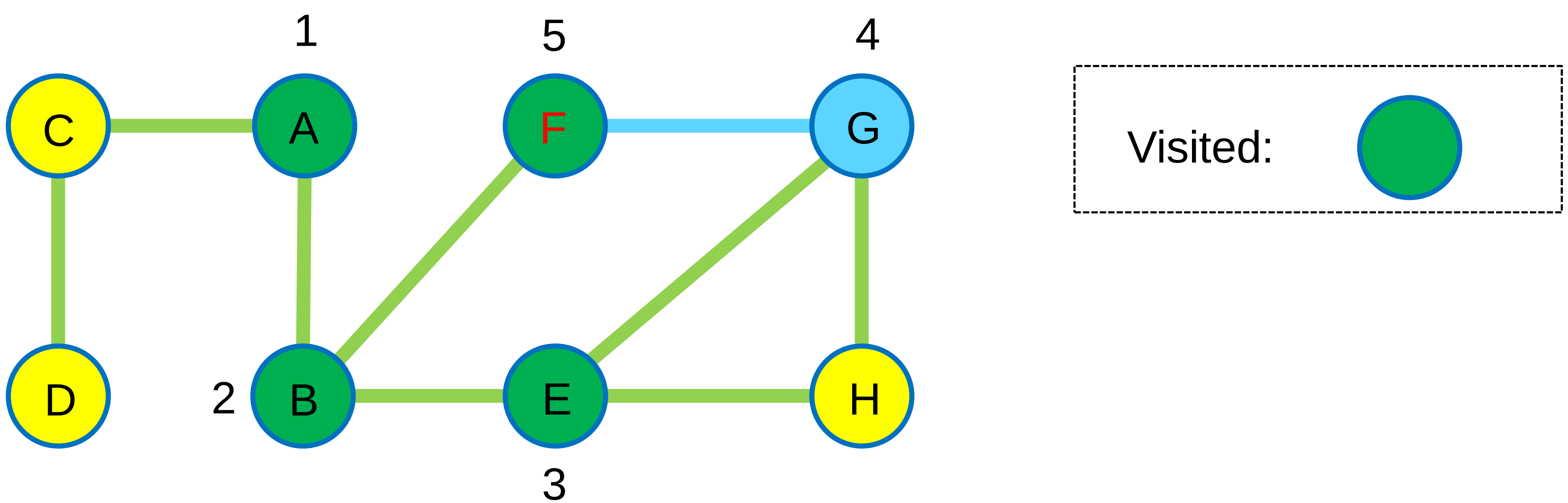


Current: **F**

Visited:

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

Depth First Search (DFS)

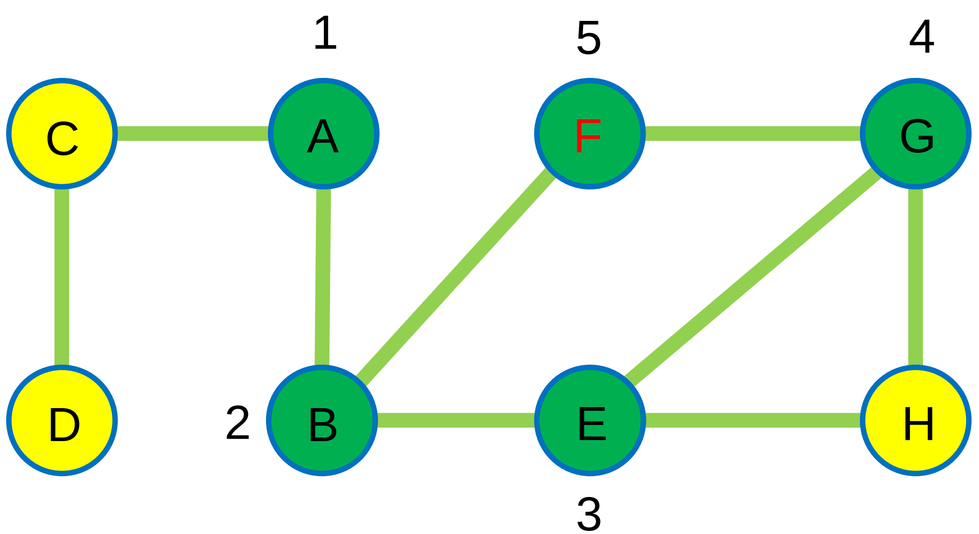


Current: **F**

Visited:

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

Depth First Search (DFS)



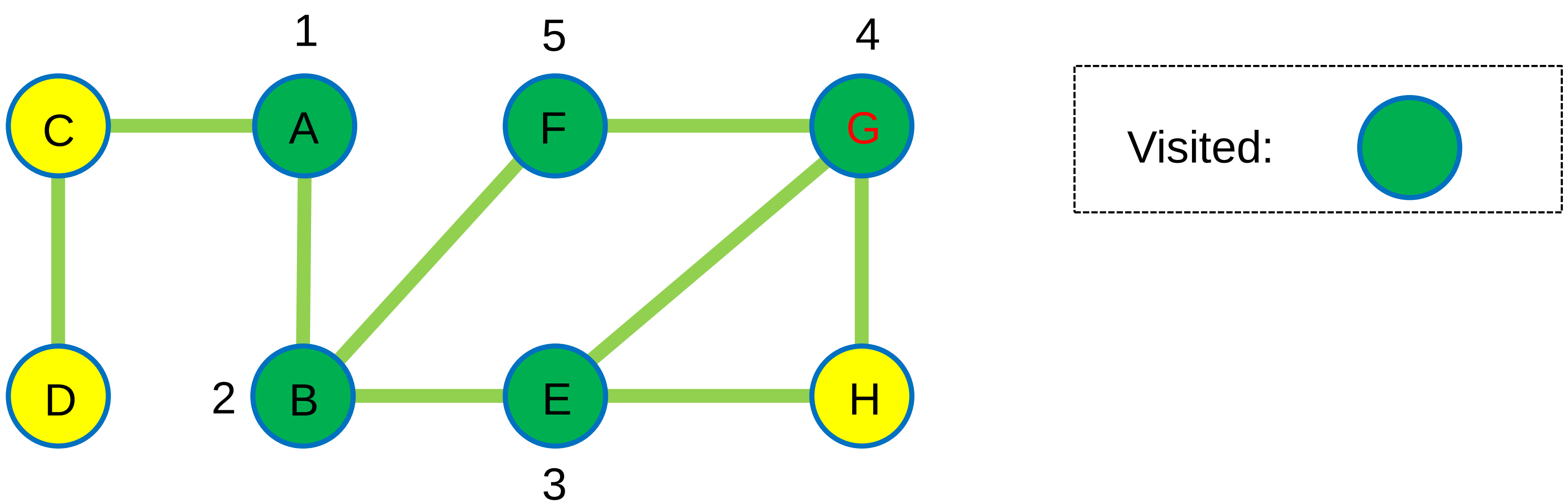
- We have checked all neighbours of F
- It is a dead end
- Go back to the last active node (G)

Current: **F**

Visited:

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

Depth First Search (DFS)

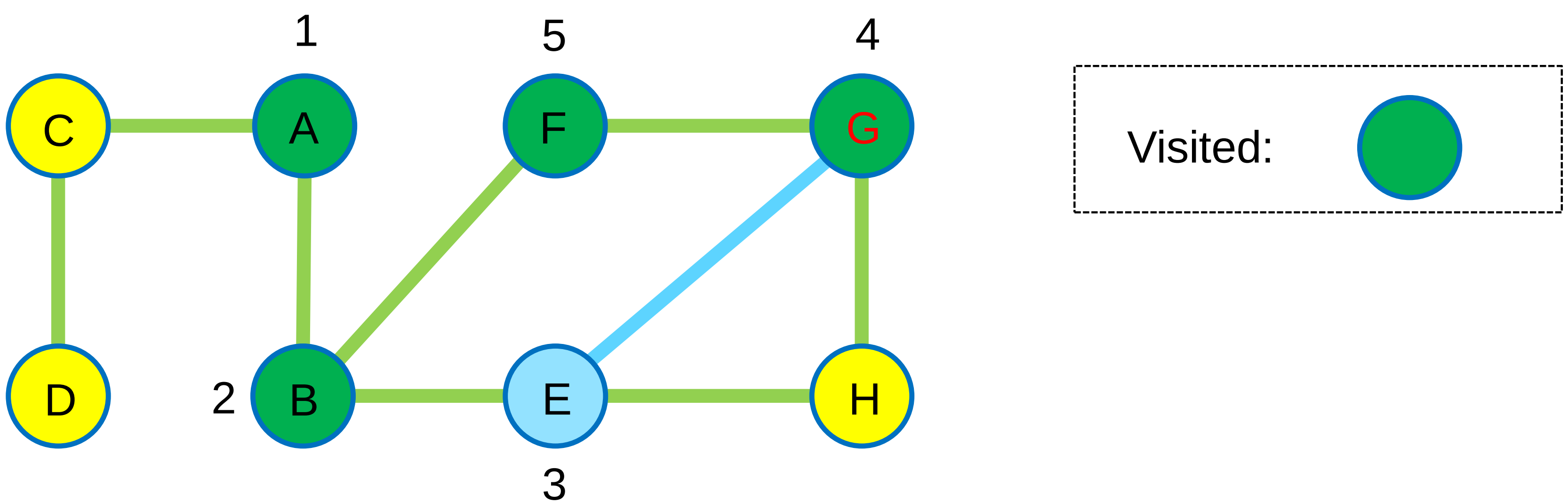


Current: **G**

Visited:

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

Depth First Search (DFS)

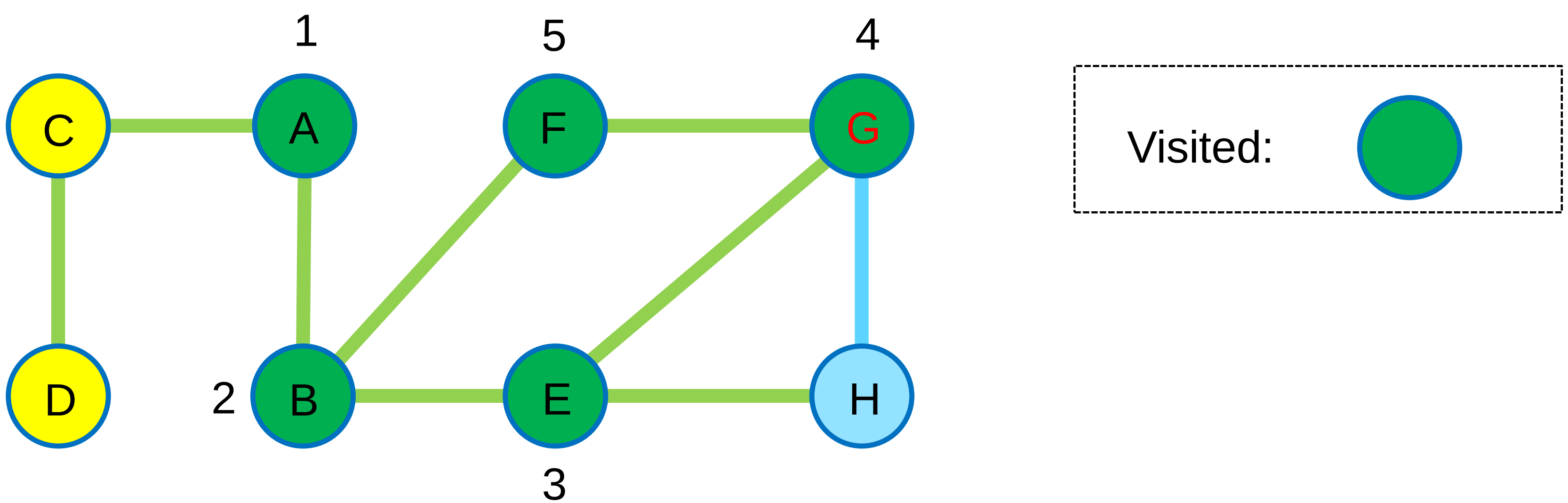


Current: **G**

Visited:

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

Depth First Search (DFS)

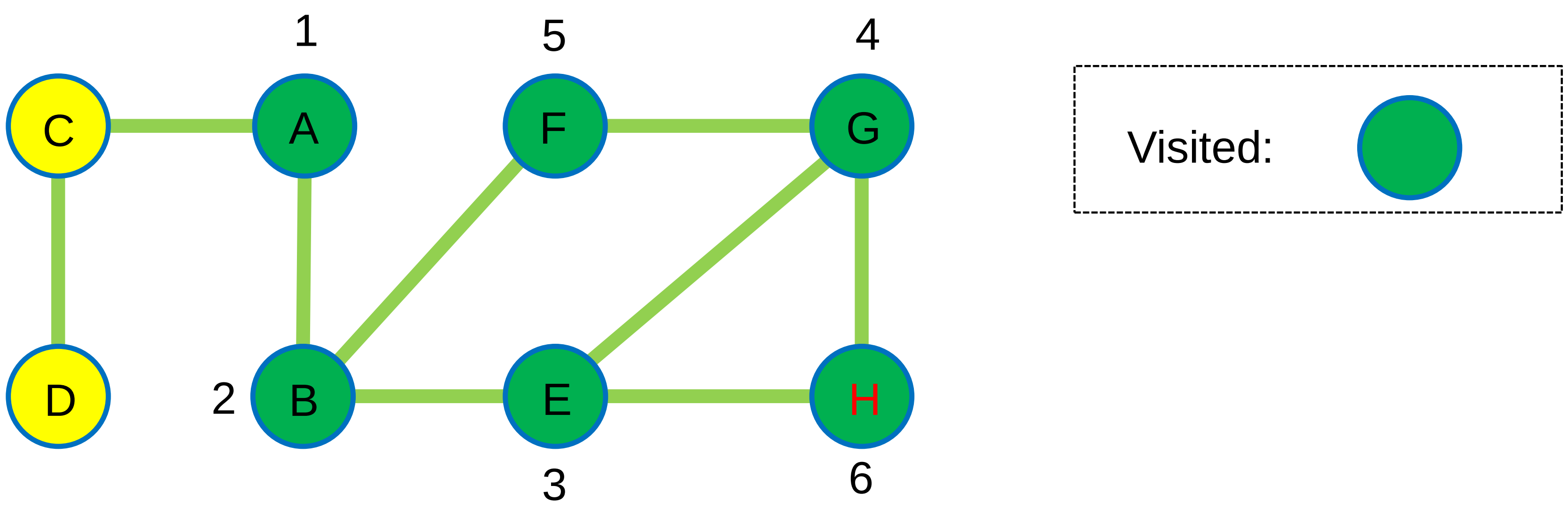


Current: **G**

Visited:

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

Depth First Search (DFS)

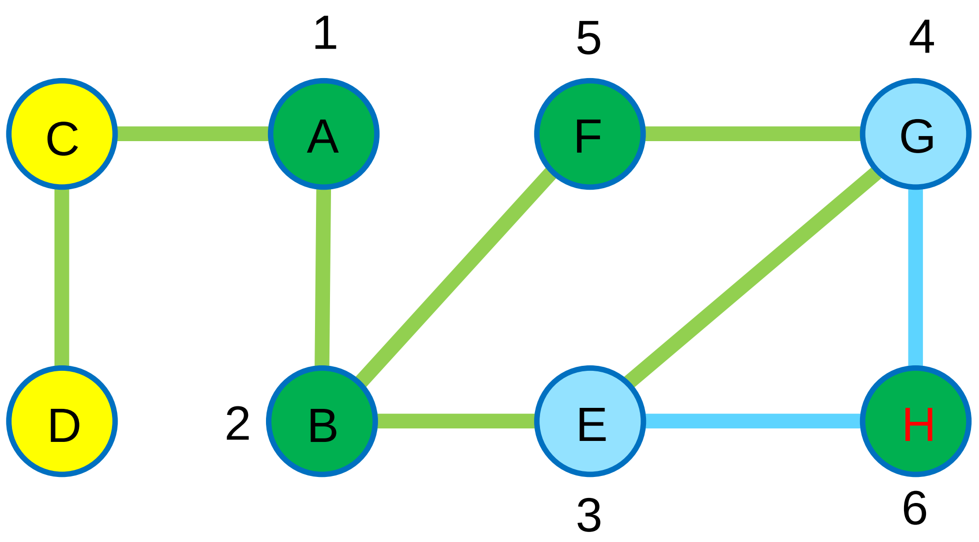


Current: **H**

Visited:

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

Depth First Search (DFS)



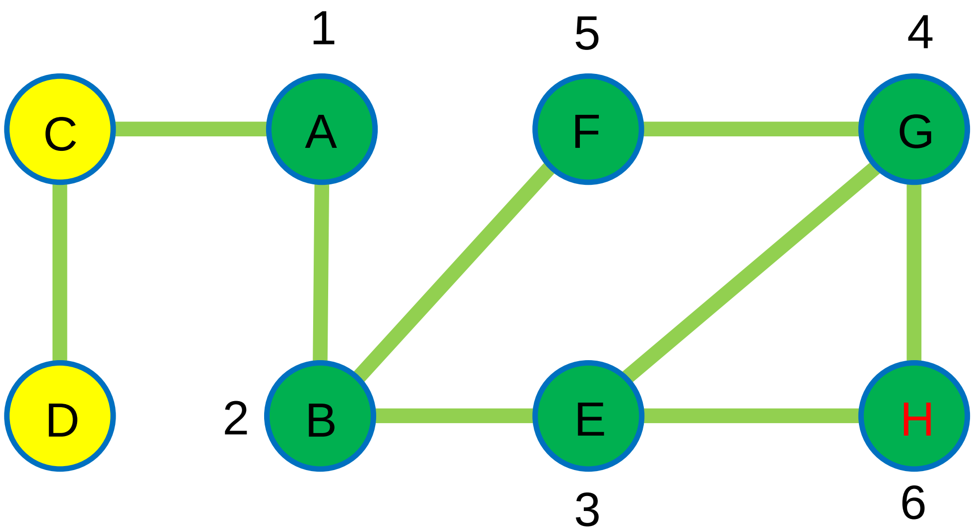
Speeding up visualisation…

Current: **H**

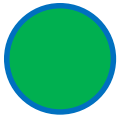
Visited:

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

Depth First Search (DFS)



Visited:



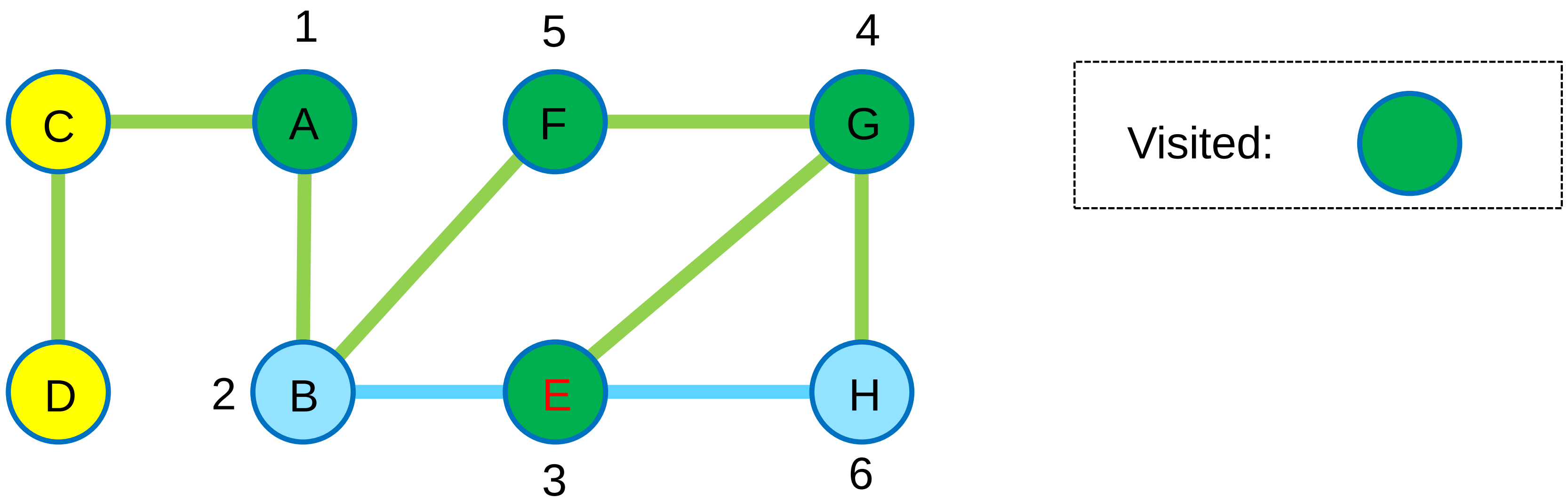
- H is a dead end
- Going back to G, it is also a dead end
- Go back to E

Current: **H**

Visited:

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

Depth First Search (DFS)

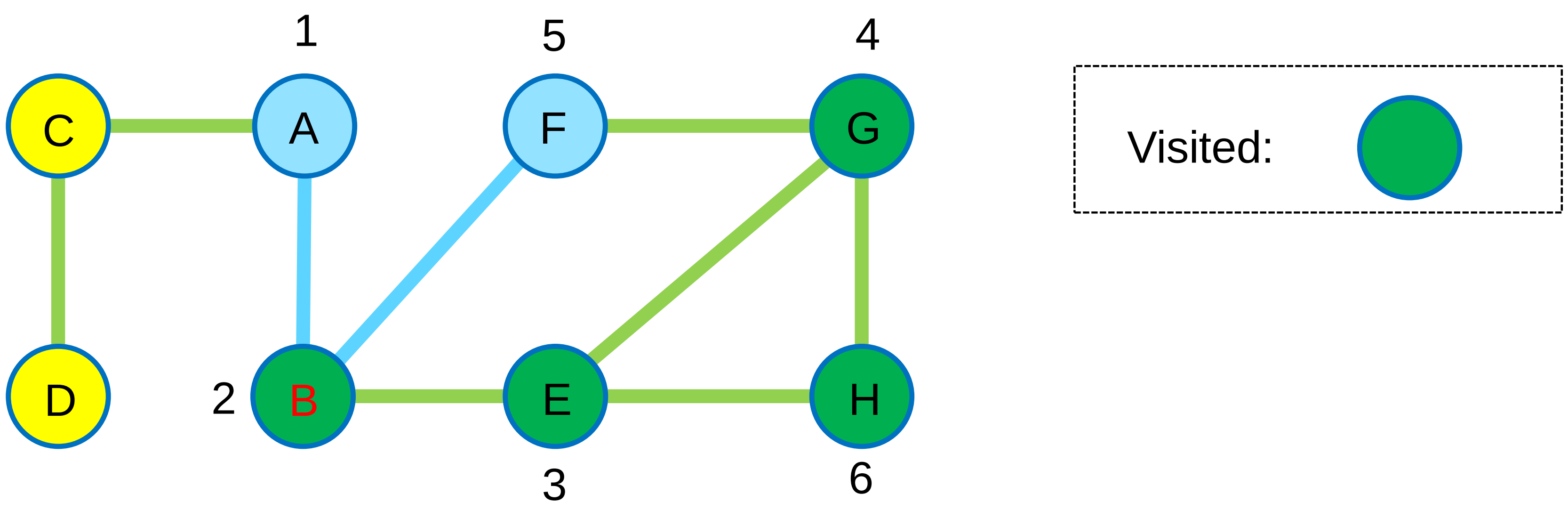


Current: **E**

Visited:

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

Depth First Search (DFS)

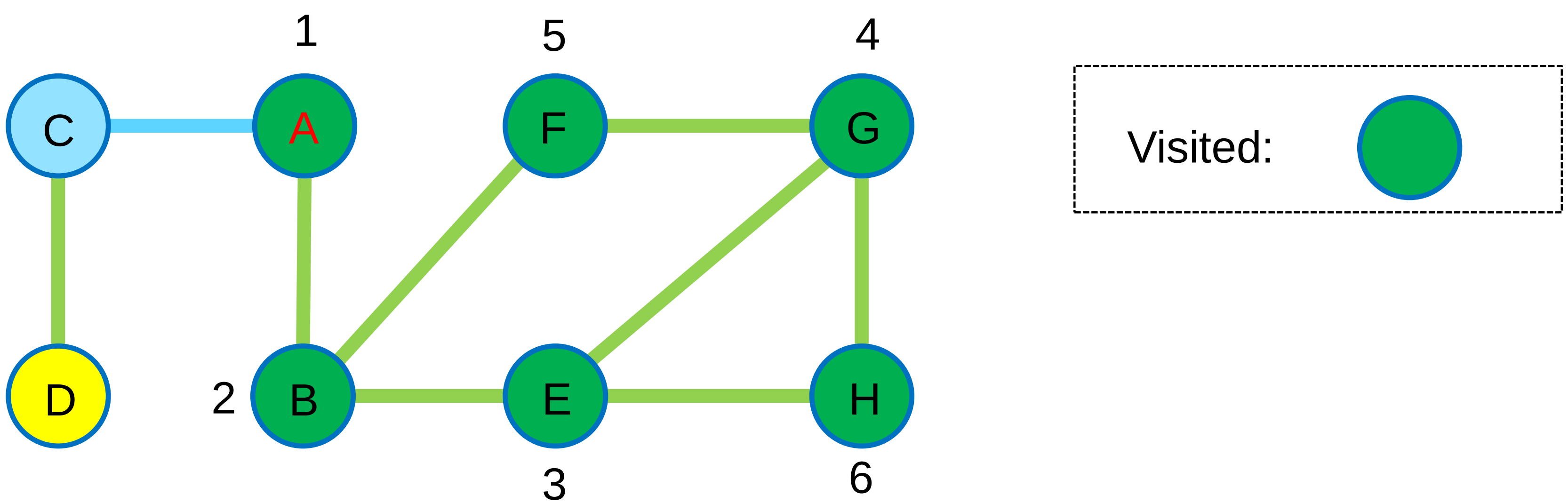


Current: **B**

Visited:

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

Depth First Search (DFS)

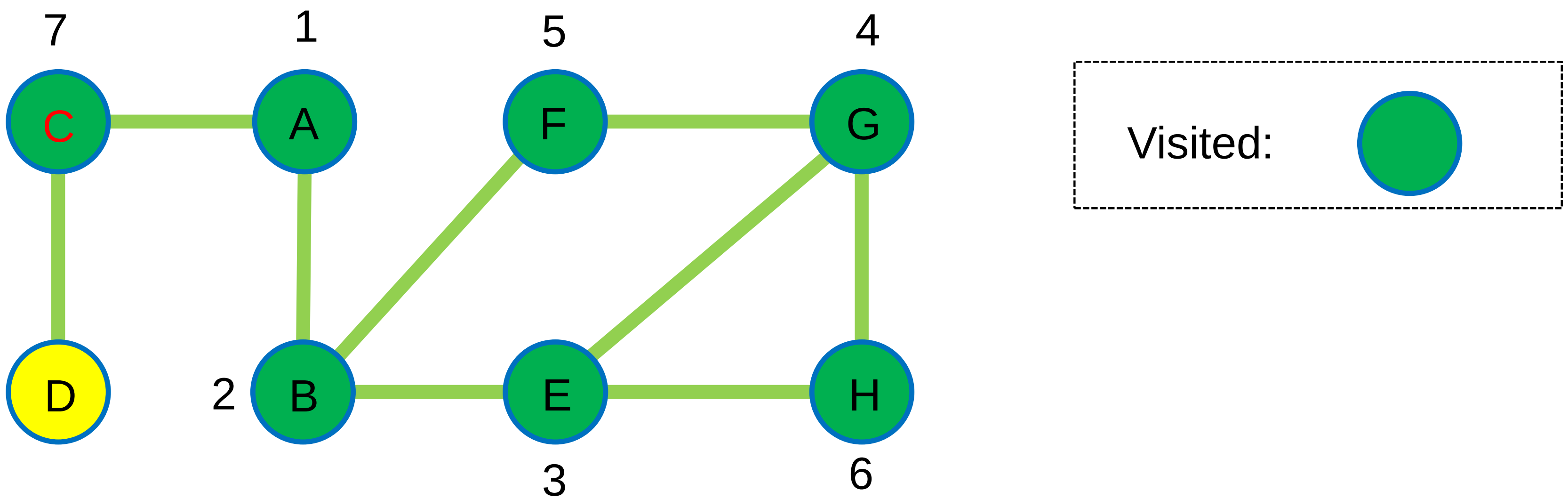


Current: **A**

Visited:

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

Depth First Search (DFS)

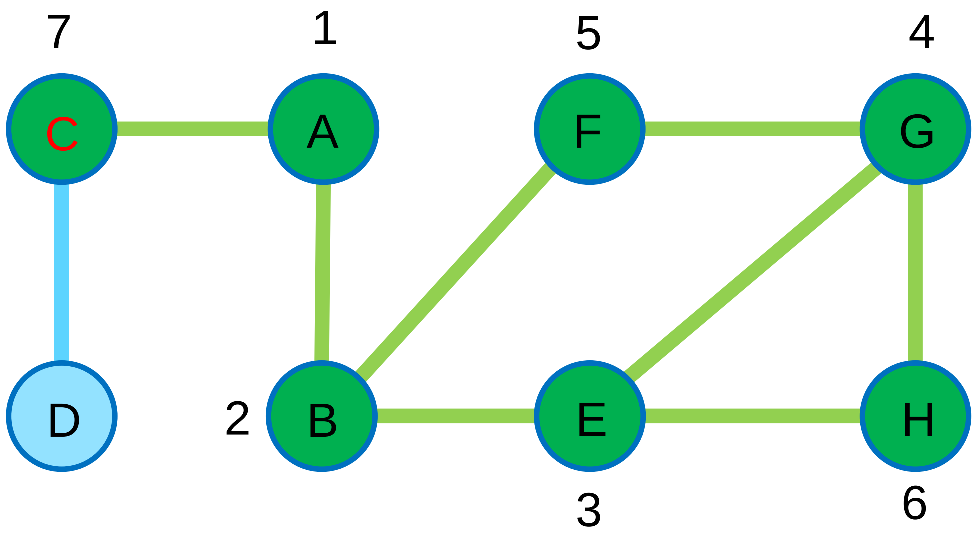


Current: **C**

Visited:

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

Depth First Search (DFS)

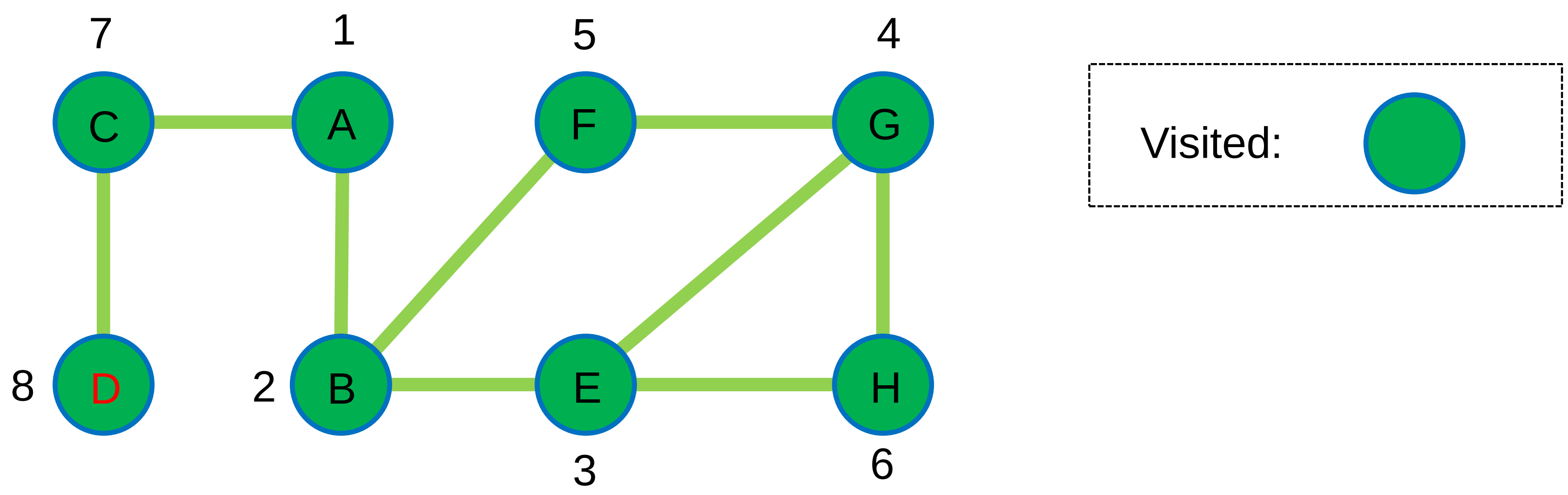


Current: **C**

Visited:

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

Depth First Search (DFS)



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)

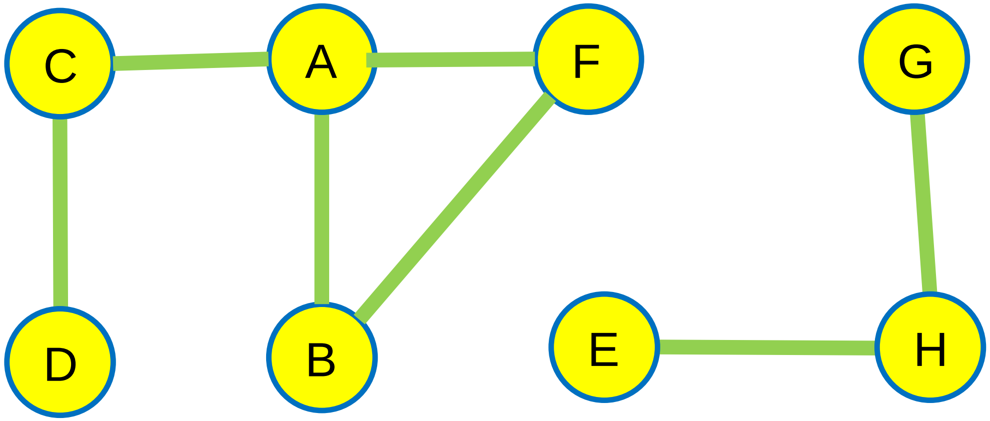
Applications of DFS and BFS

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
- Finding cycles
- Topological sort (week 11)
- 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 sources single target:

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

Single source all targets:

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

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

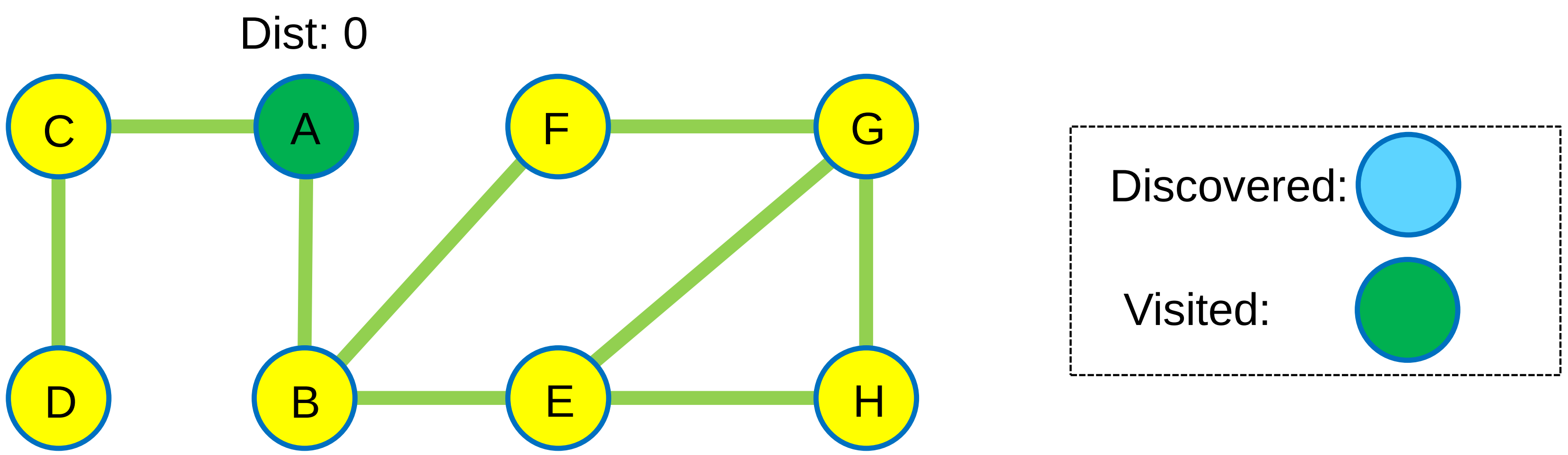
Shortest Path Algorithms

- Breadth First Search – (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)



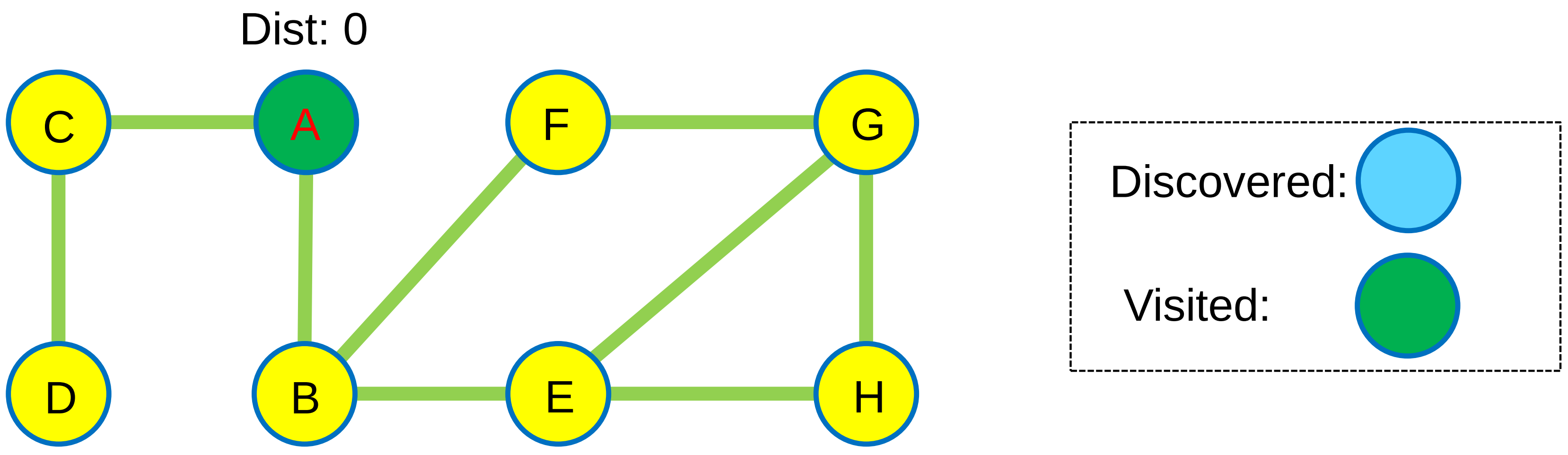
u (current):

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)



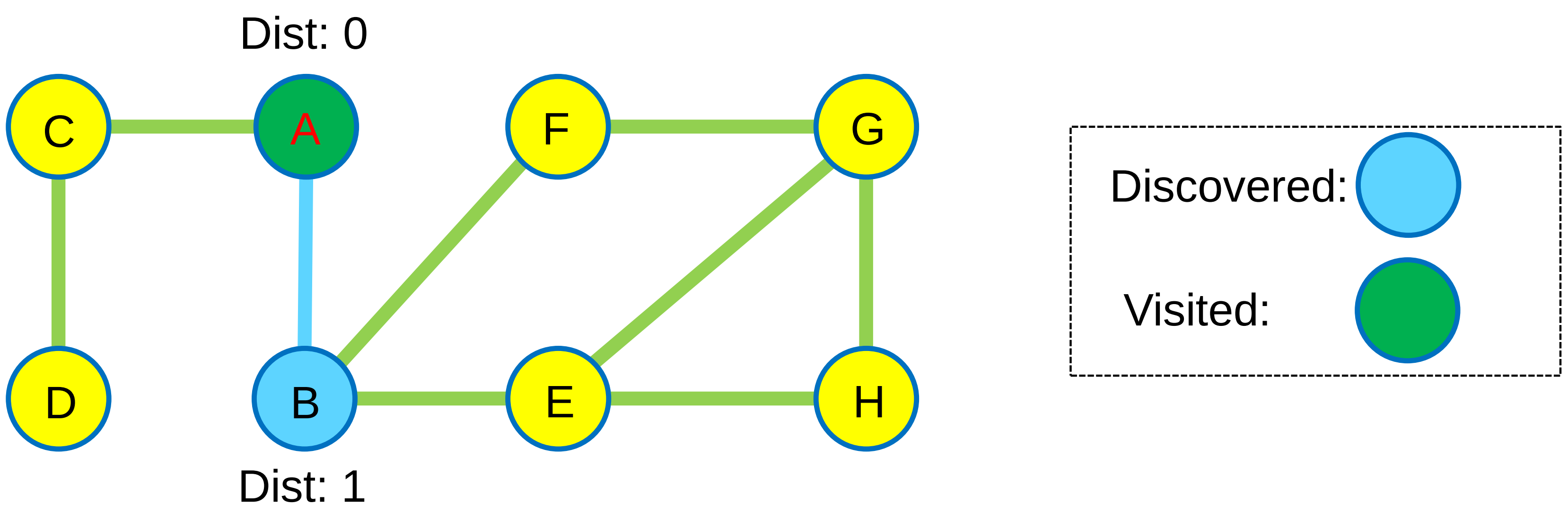
u (current): A

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)



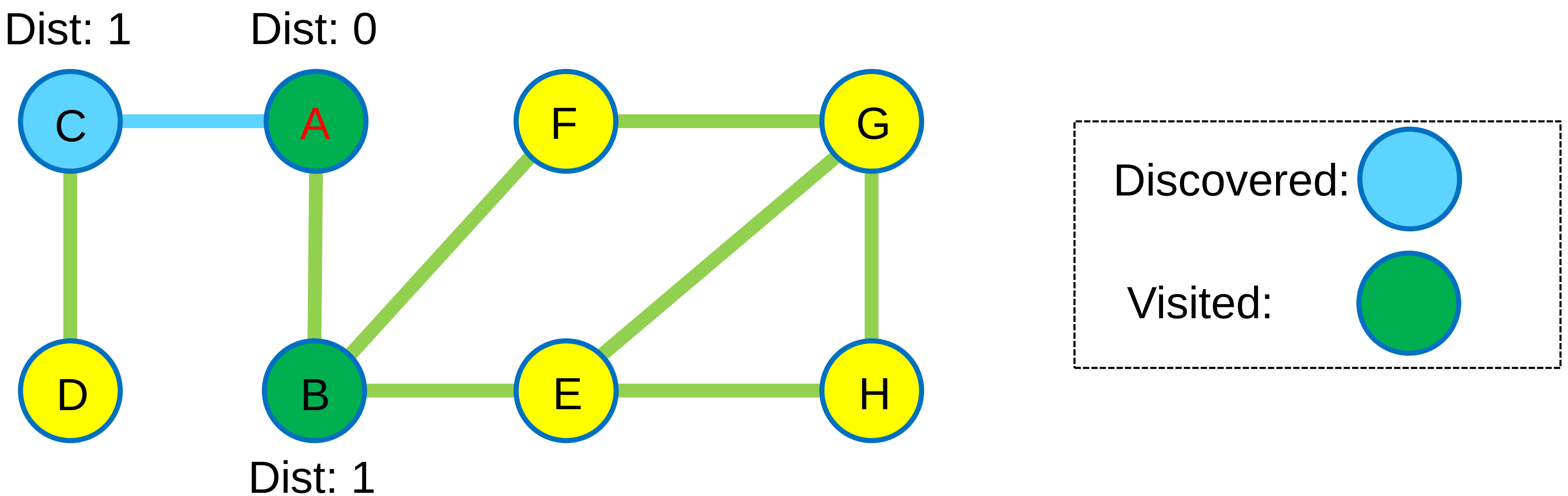
u (current): A

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)



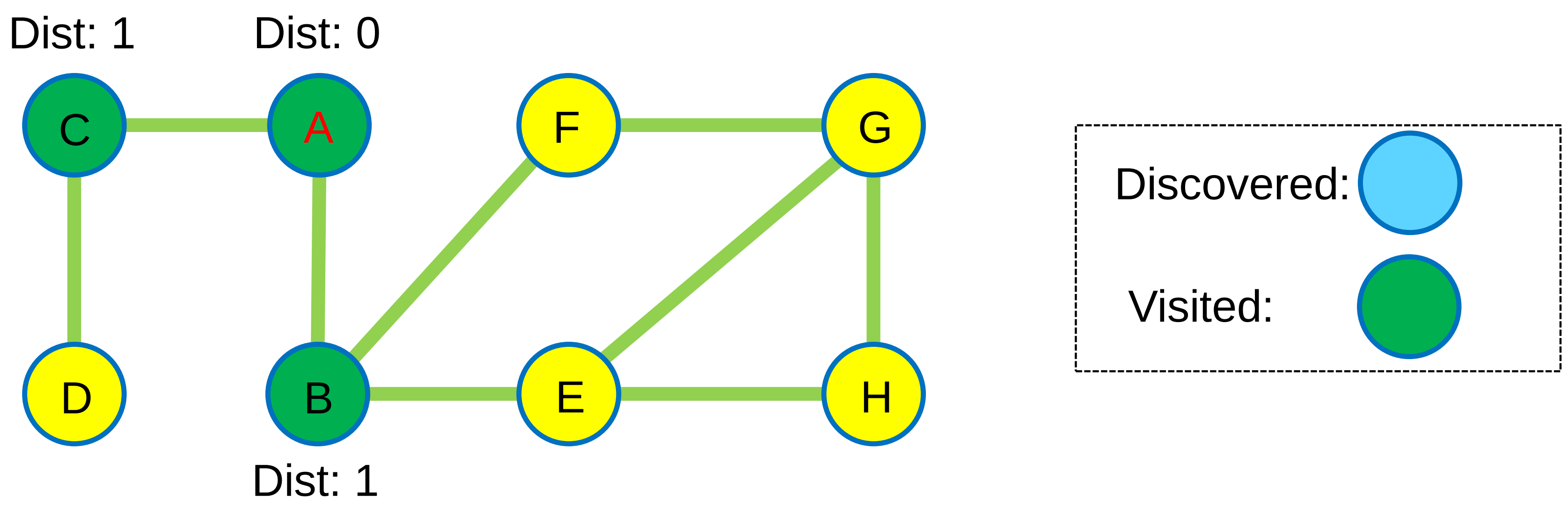
u (current): A

Queue: B

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)



u (current): A

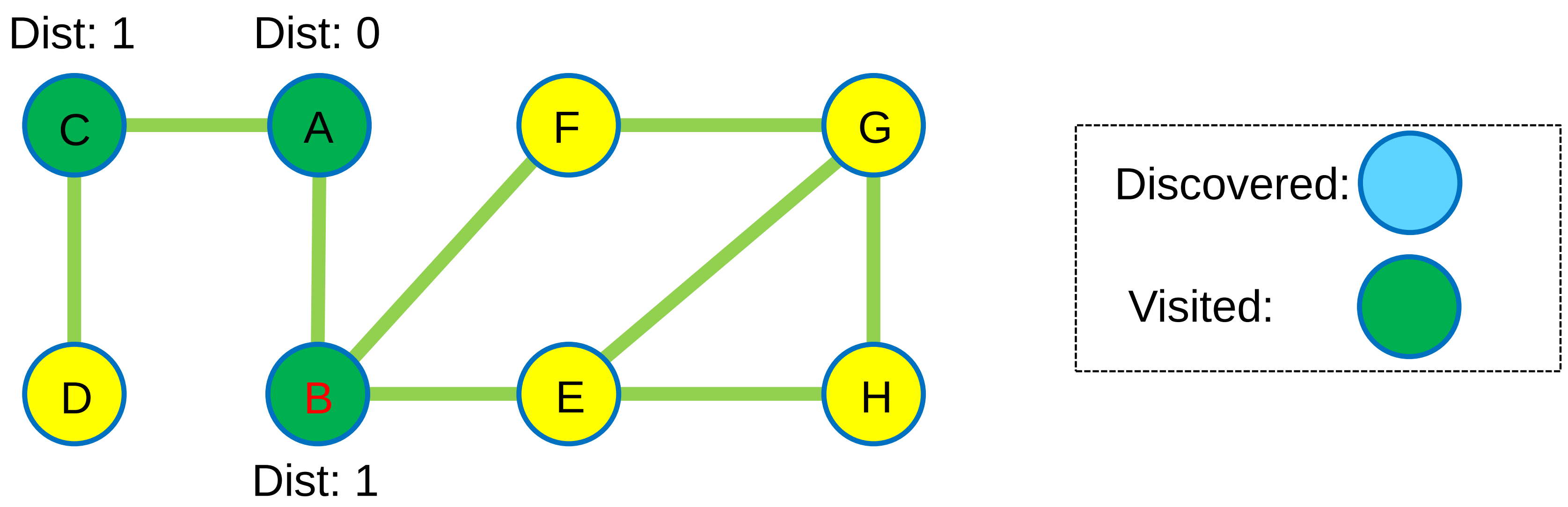
Queue:

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)



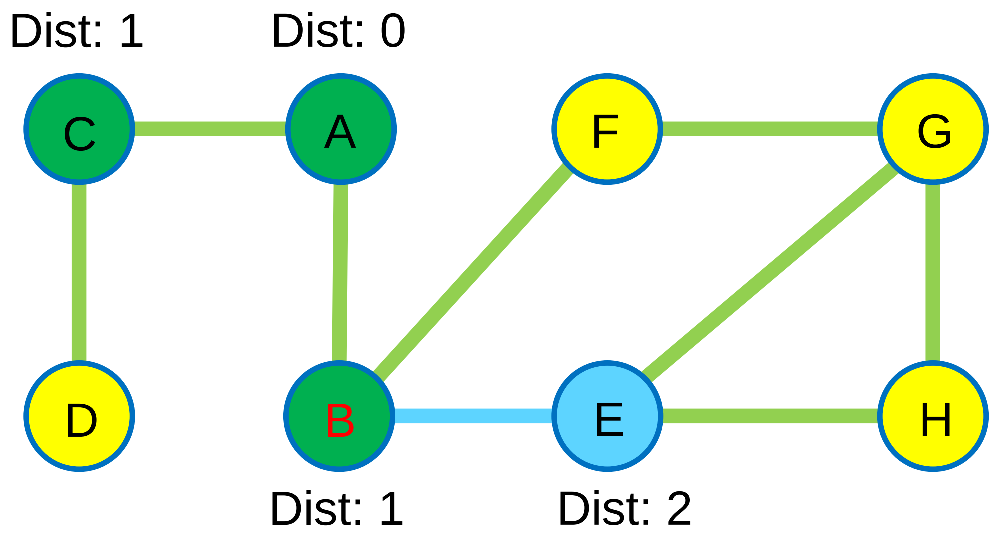
u (current): B

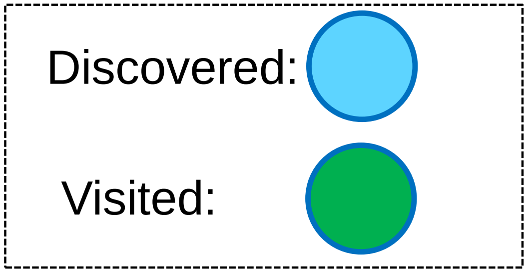
Queue: C

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)





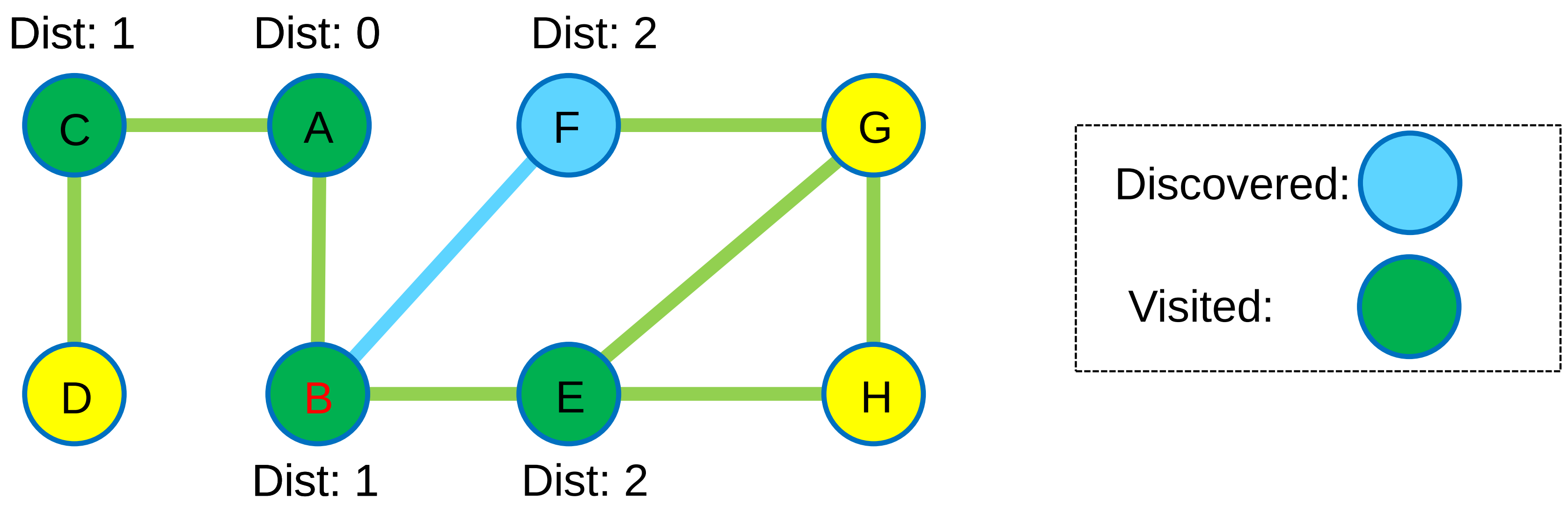
u (current): B

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)



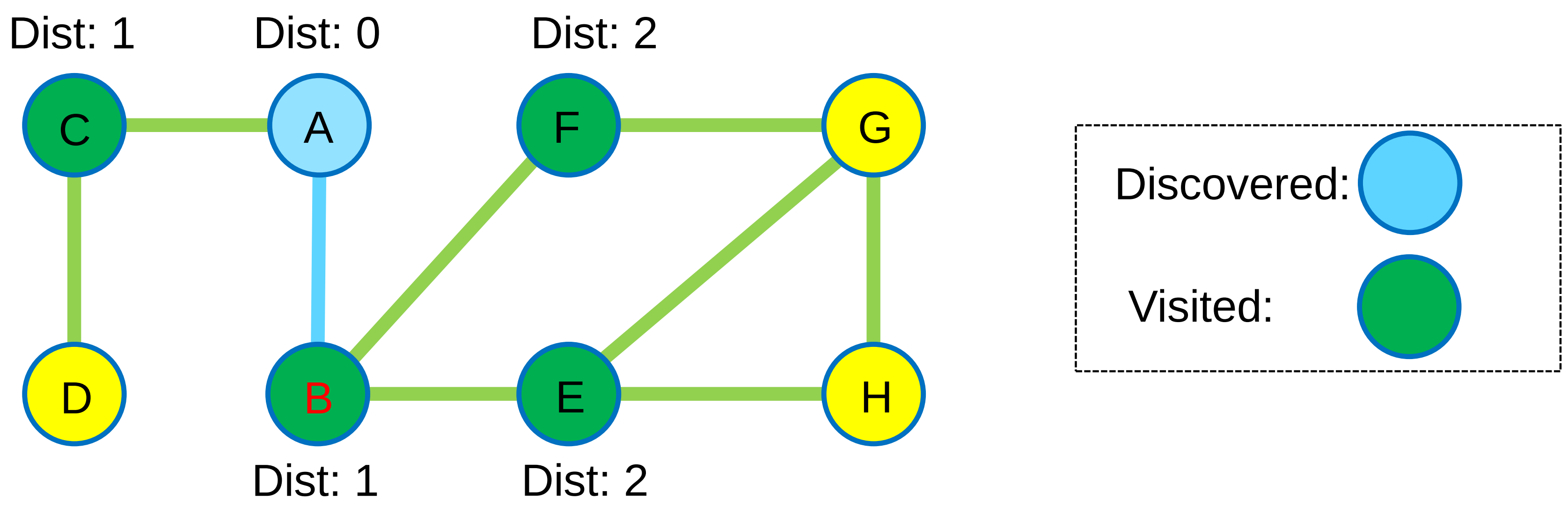
u (current): B

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)



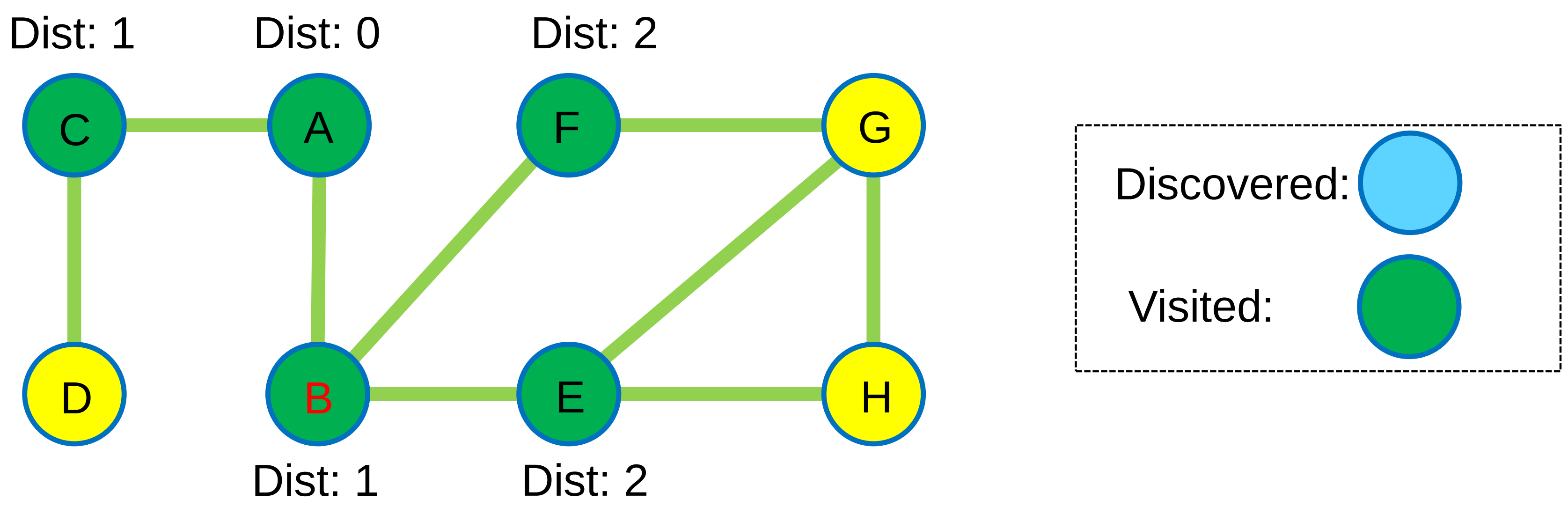
u (current): B

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)



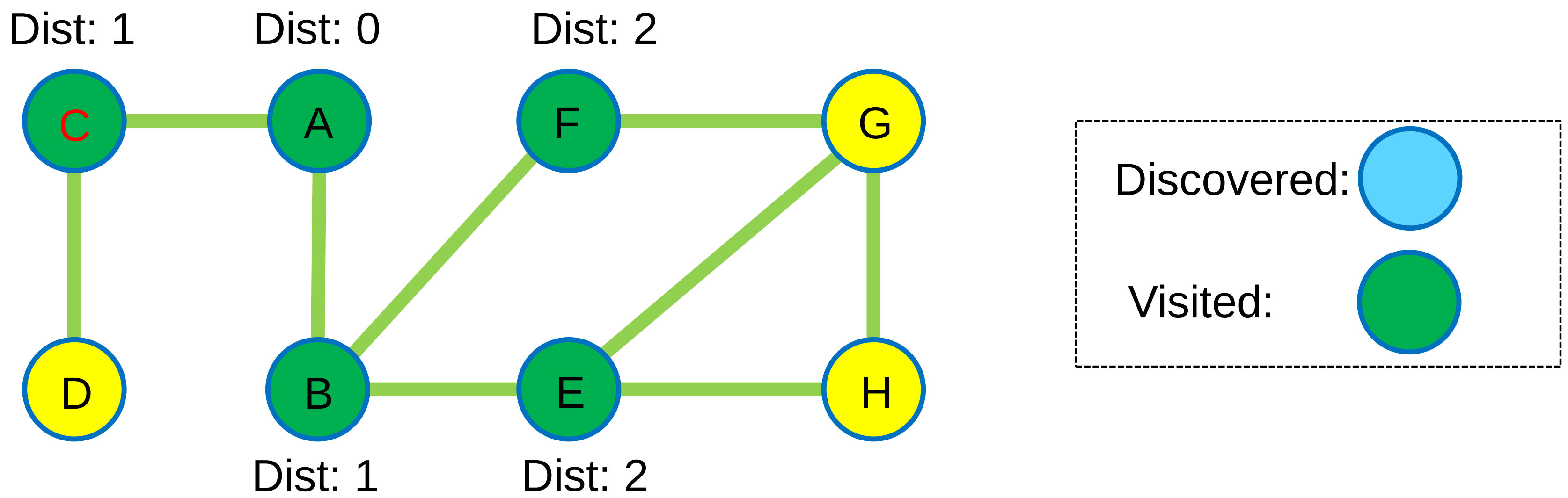
u (current): B

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)



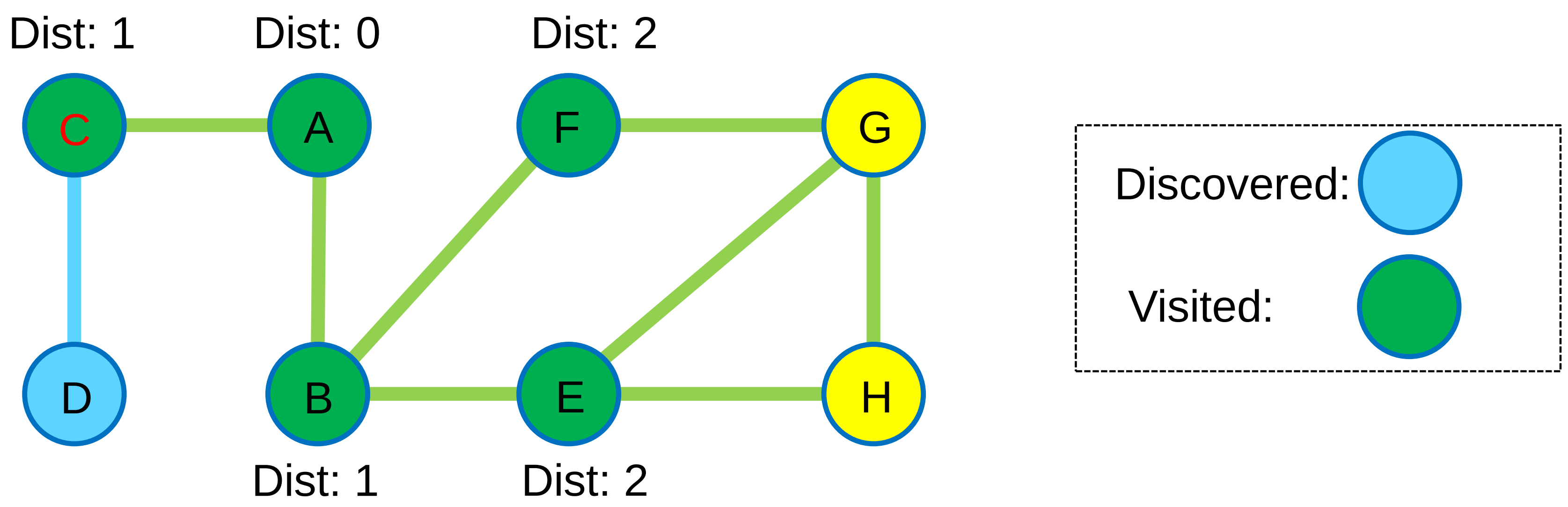
u (current): C

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)



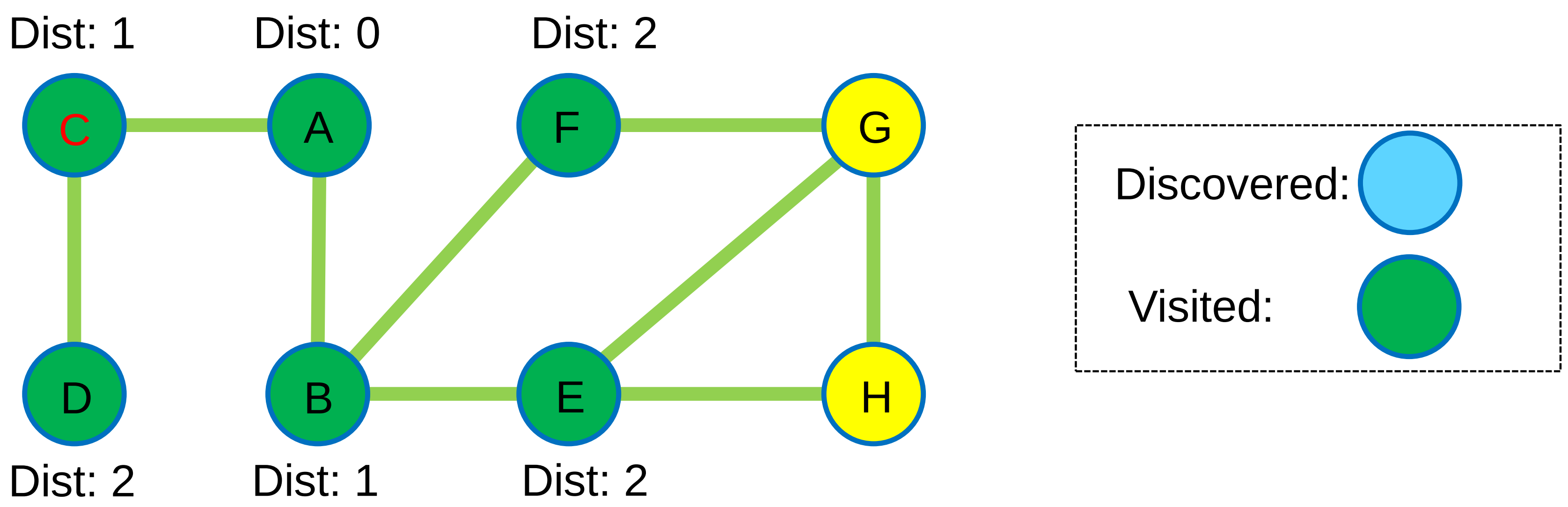
u (current): C

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)



u (current): C

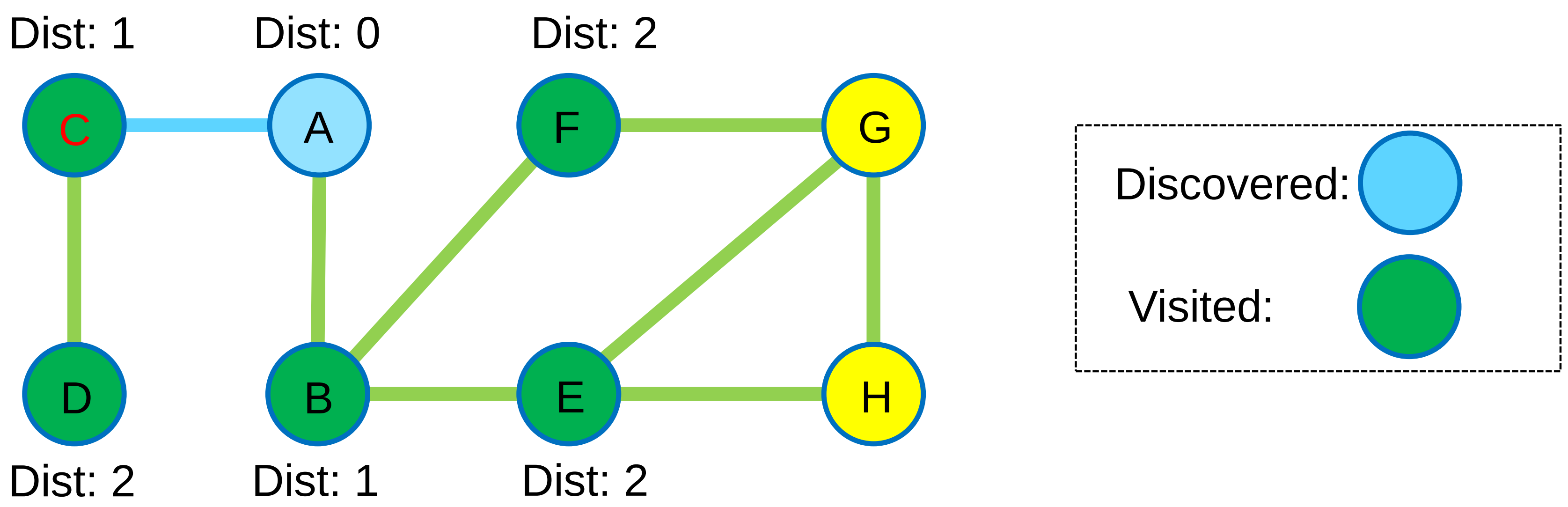
Queue:

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)



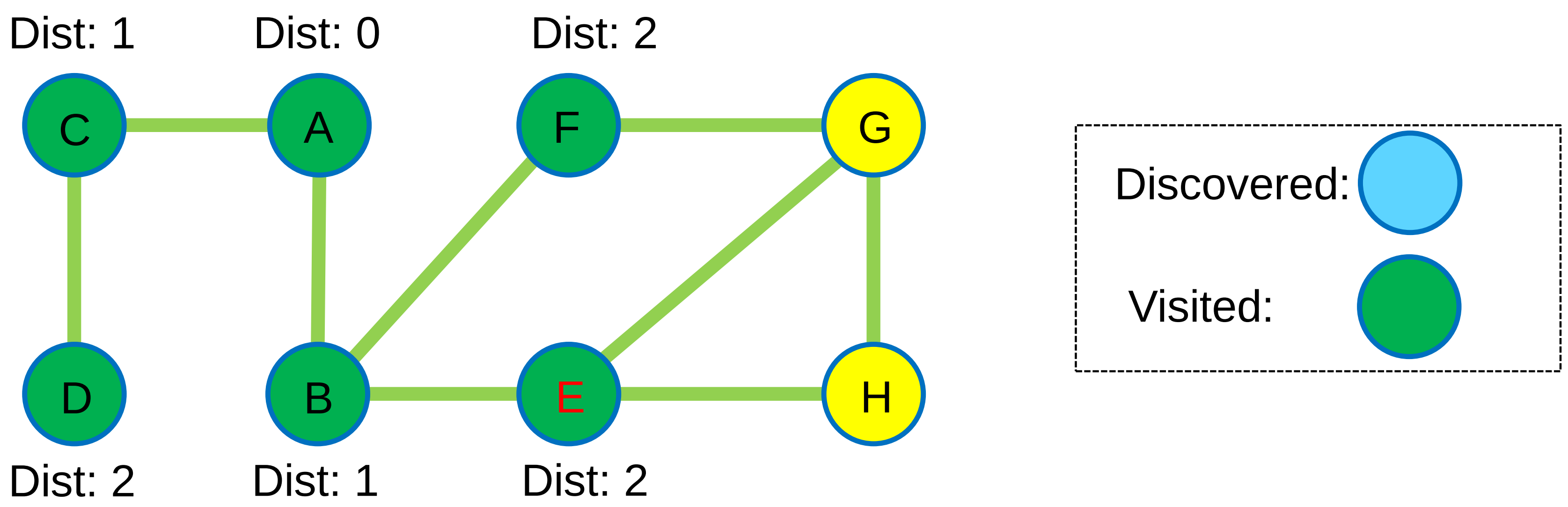
u (current): C

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)



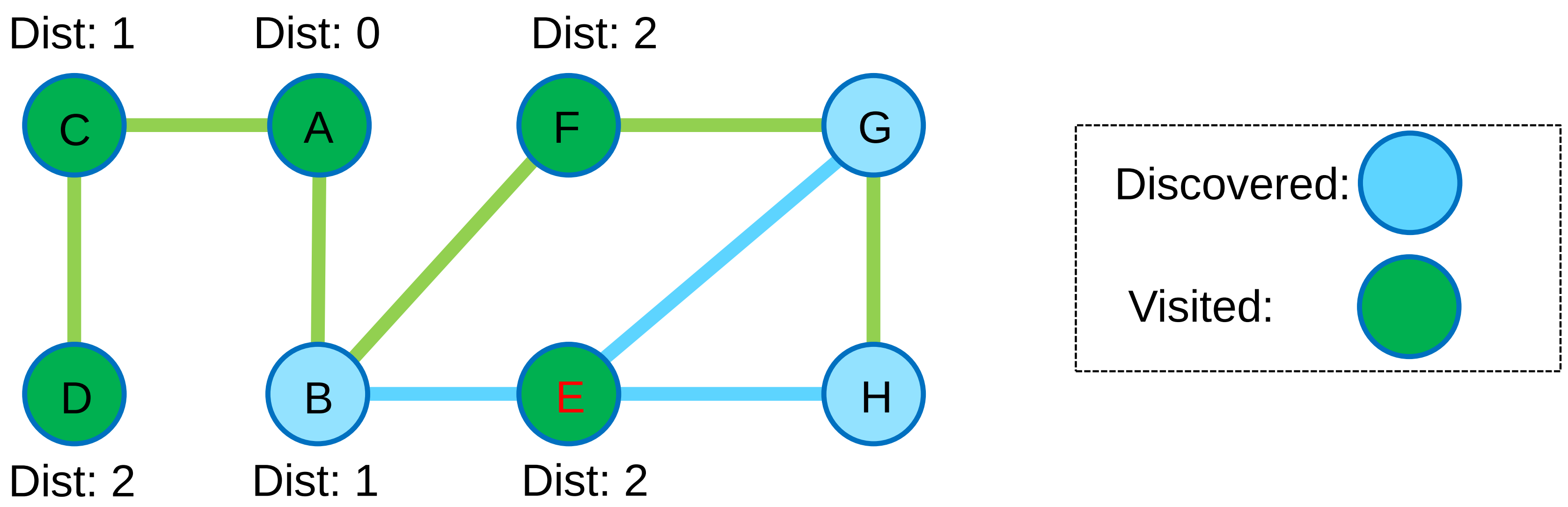
u (current): E

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)



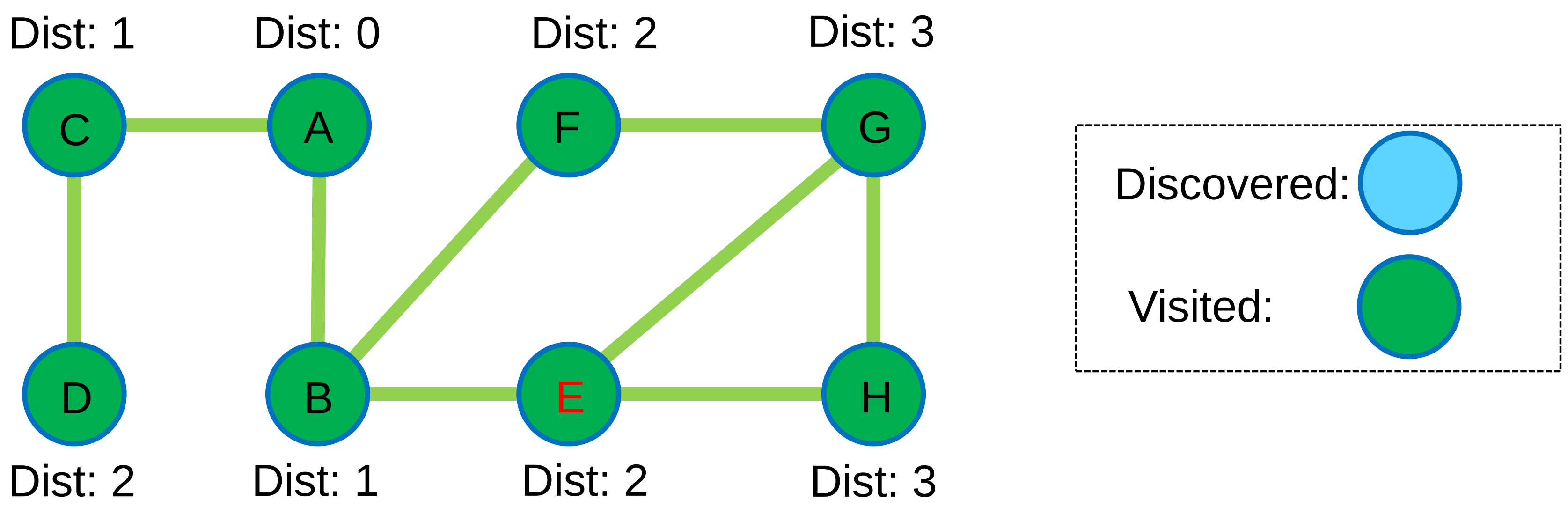
u (current): E

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)



u (current): E

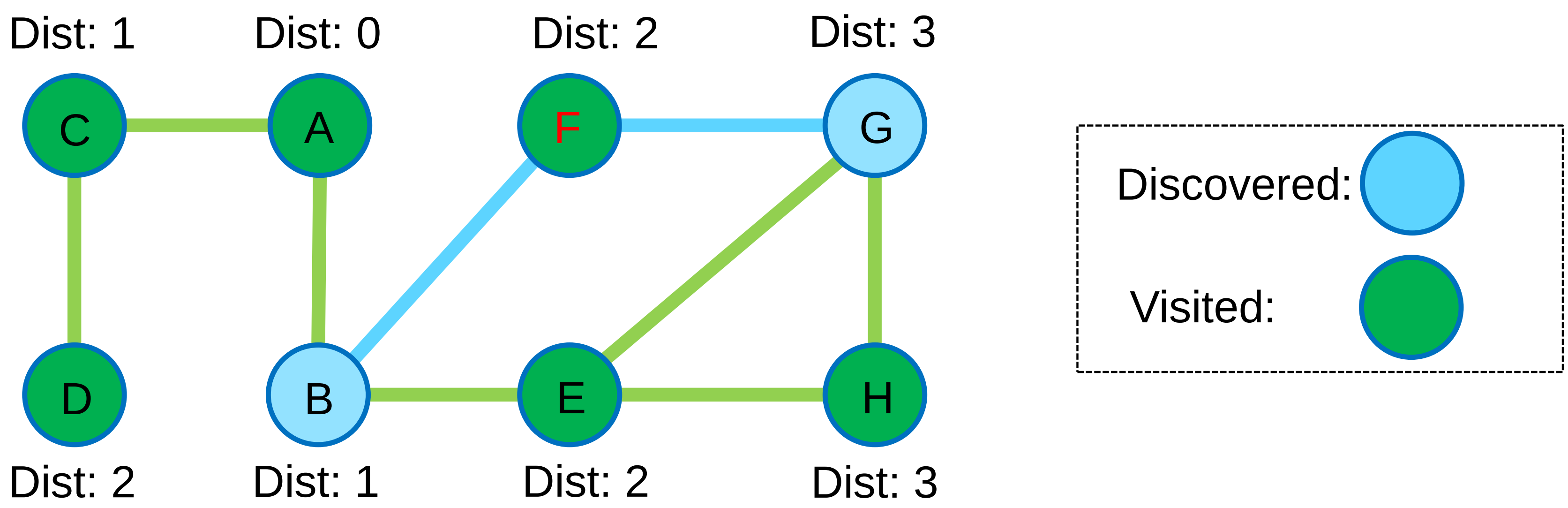
Queue:

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)



u (current): F

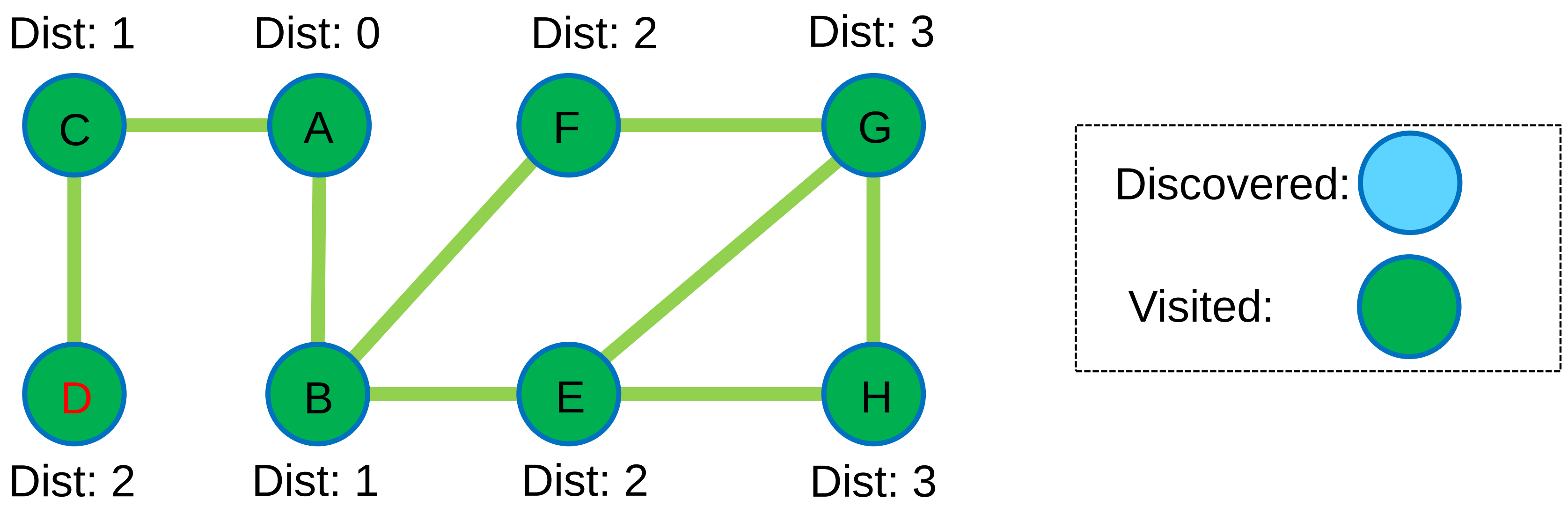
Queue:

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)



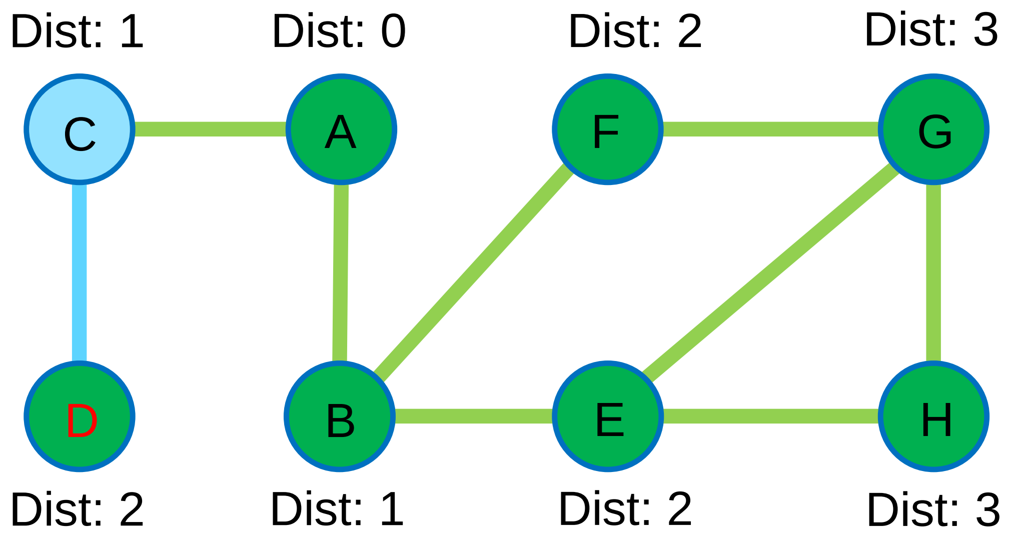
u (current): D

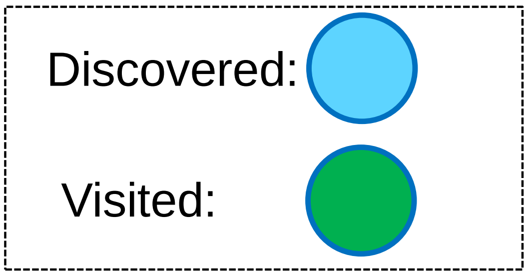
Queue: G | H

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)





u (current): D

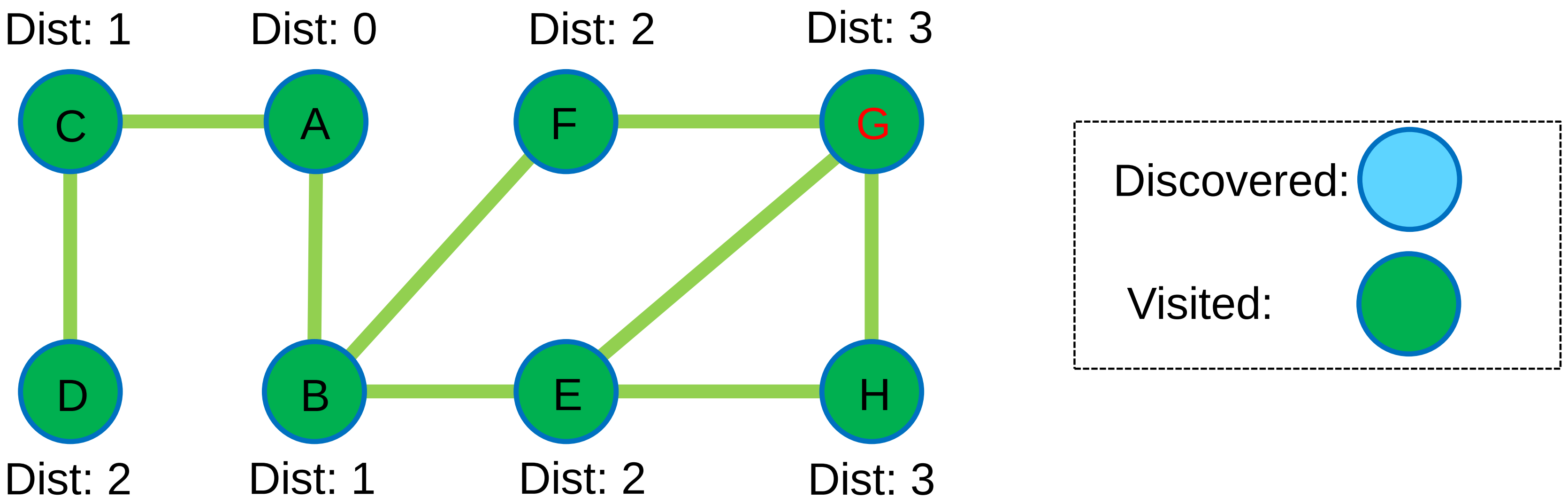
Queue:

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)



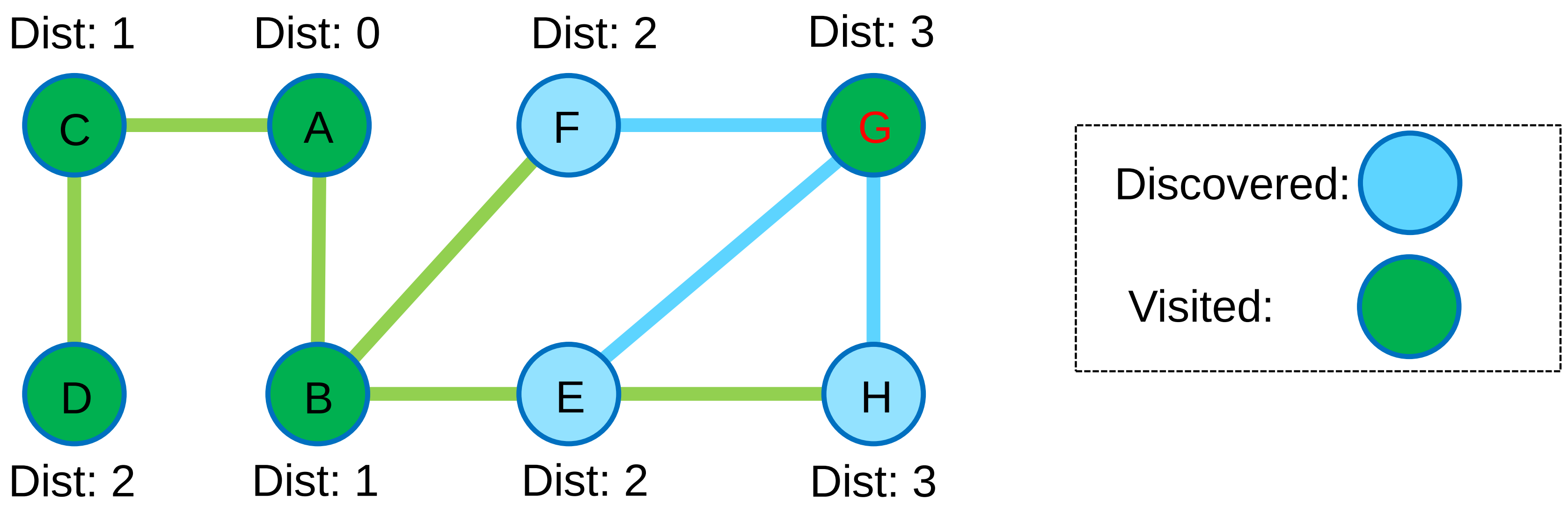
u (current): G

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)



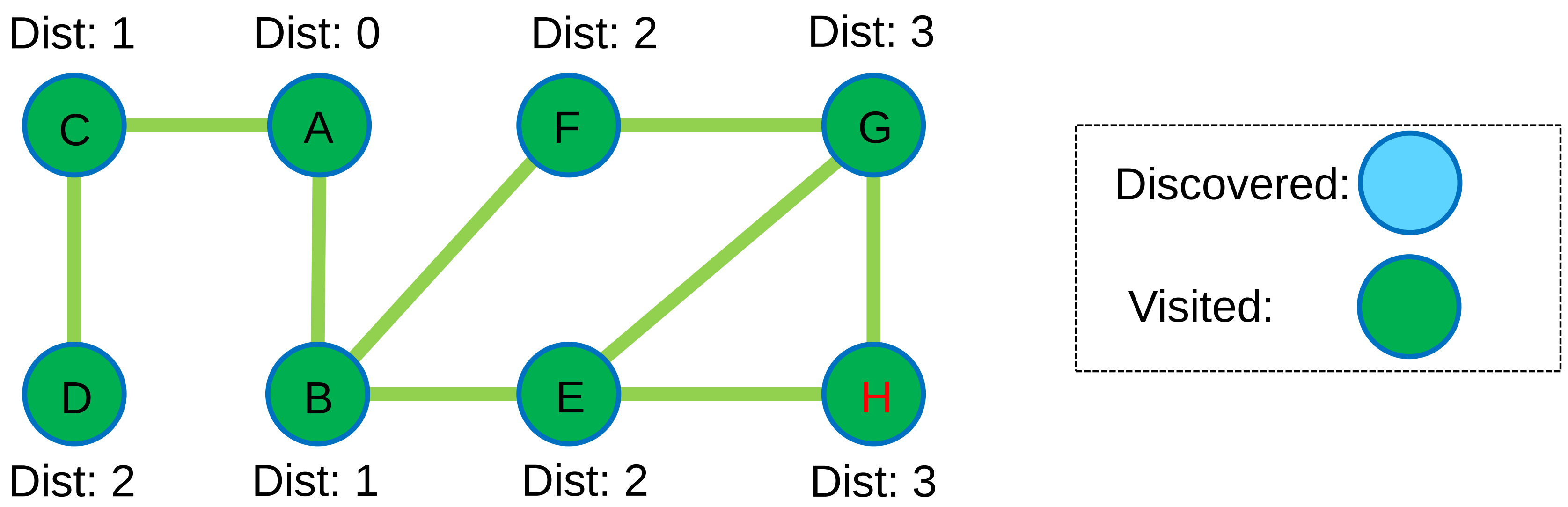
u (current): G

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)



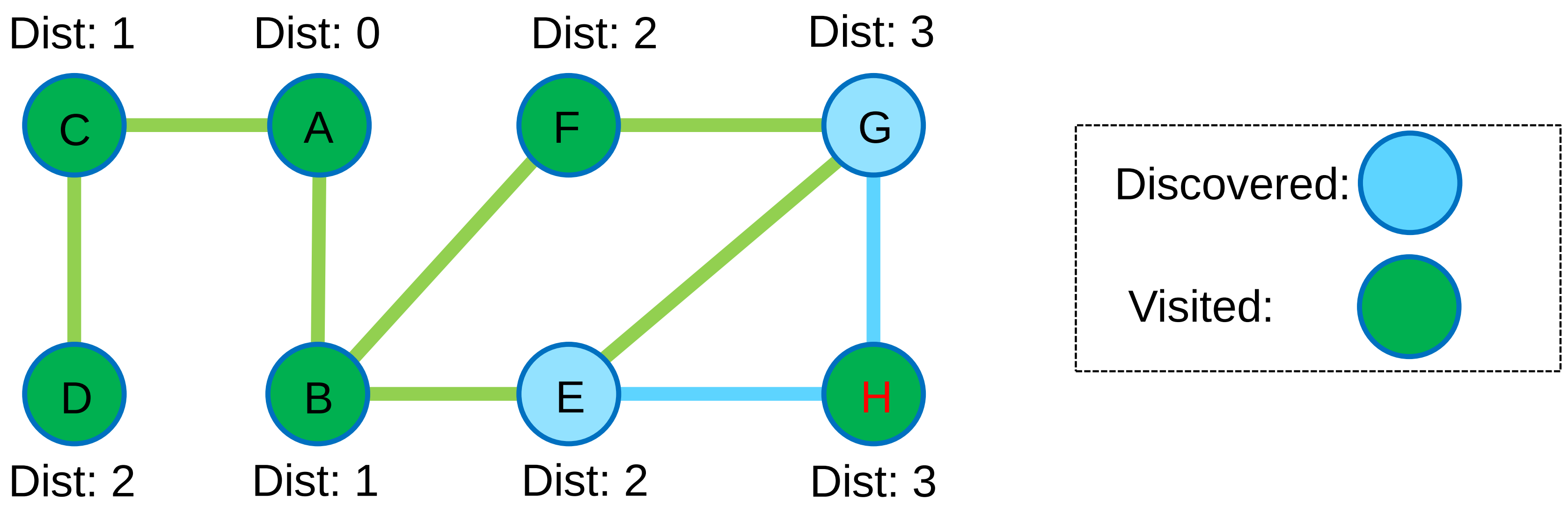
u (current): H

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)



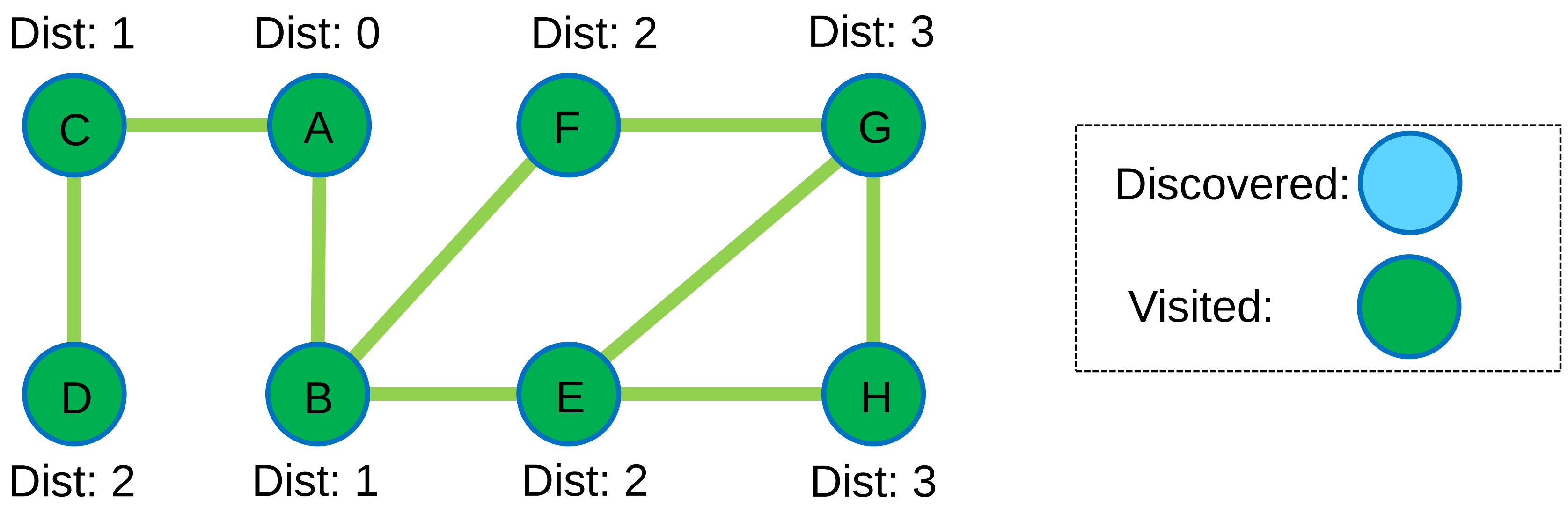
u (current): H

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)



u (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] = ∞
    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] = ∞ then
                dist[v] = dist[u] + 1
                pred[v] = u
                queue.push(v)
```

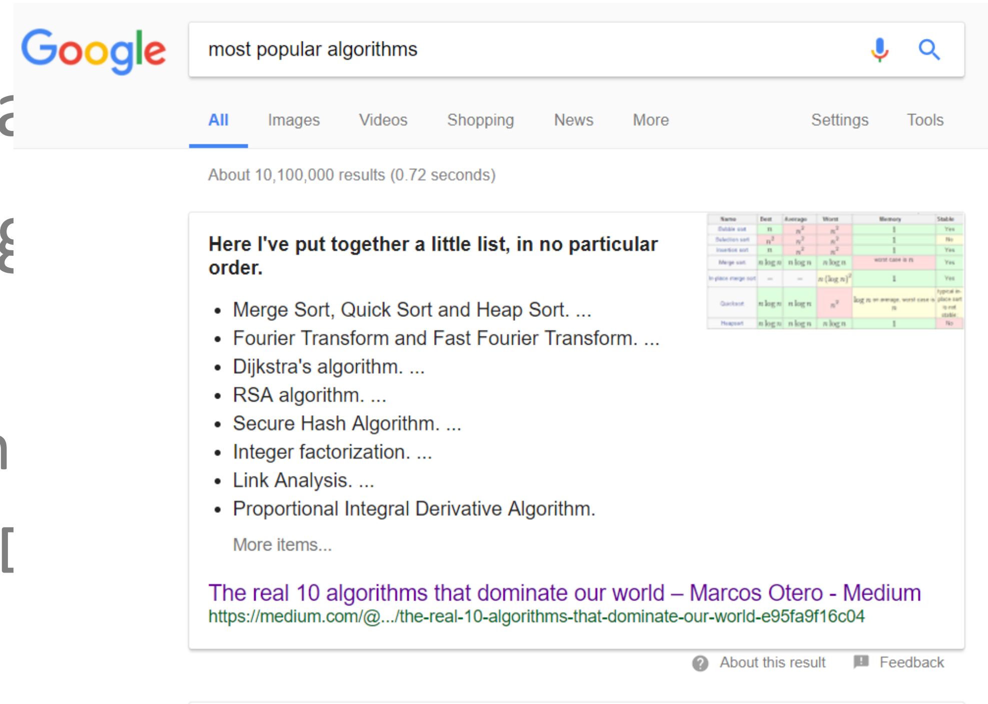
Note that distances are stored in an O(1) lookup structure

Complexity is the same as regular BFS, O(V+E)

Distances are set by lookup at the distance of the current vertex and adding 1

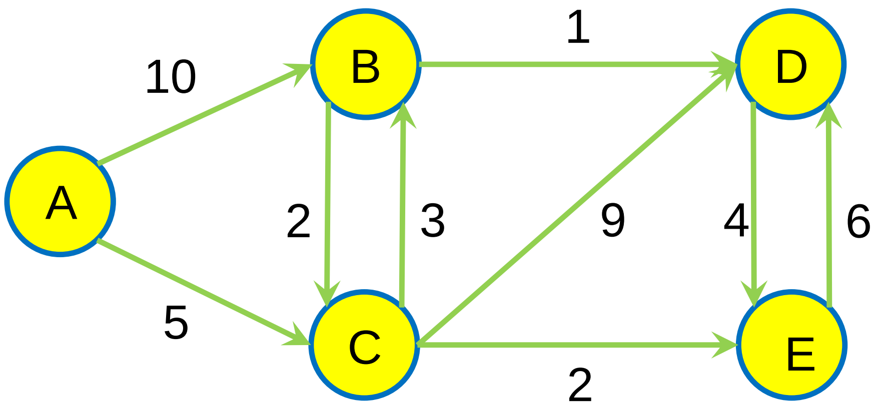
Outline

1. Introduction to Gra
2. Graph Traversal Alg
   - A. The idea
   - B. Breadth First Search
   - C. Depth First Search (I
   - D. Applications

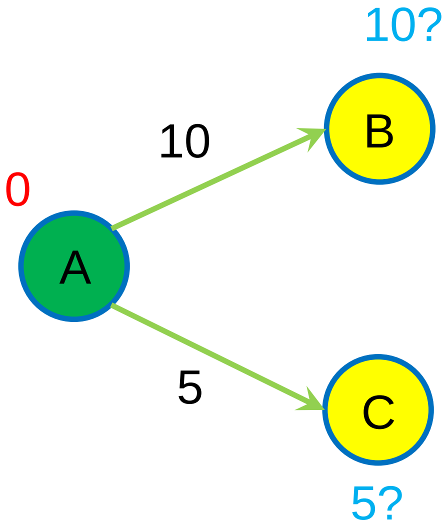


3. Shortest Path Problem
   - A. Breadth First Search (for unweighted graphs)
   - B. Dijkstra's algorithm (for weighted graphs with non-negative weights)

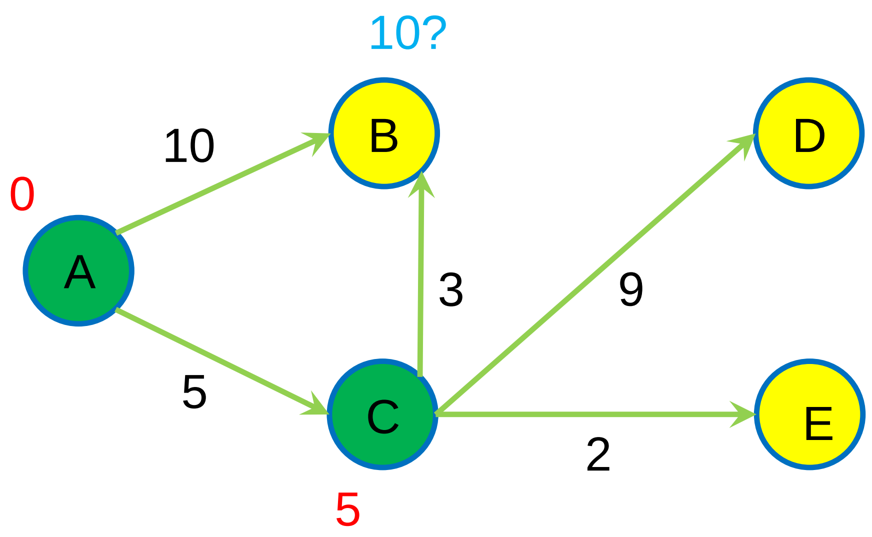
Dijkstra's Algorithm



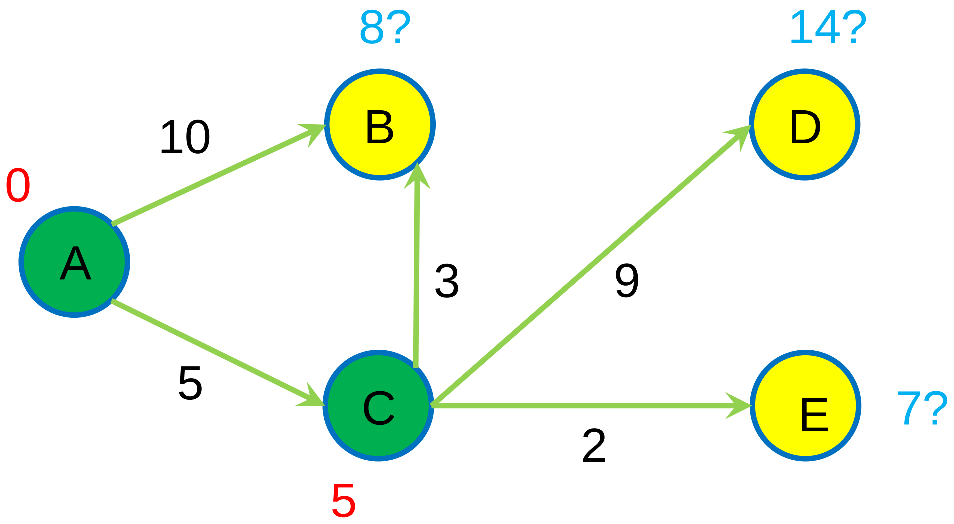
Dijkstra’s Algorithm



Dijkstra's Algorithm

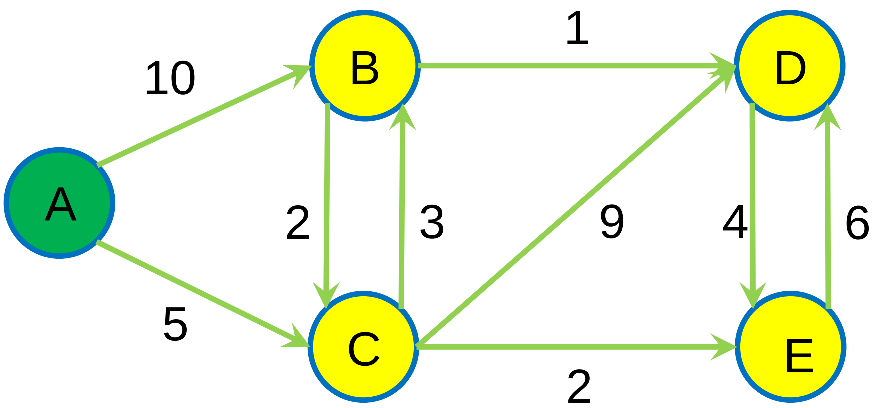


Dijkstra's Algorithm



Dijkstra's Algorithm

u:



Q:

A	B	C	D	E
0	Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

Dist:

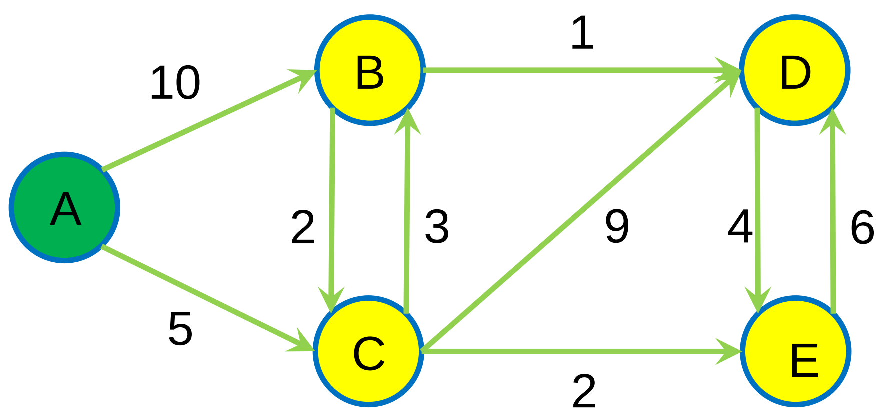
A	B	C	D	E
0	Inf	Inf	Inf	inf

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



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

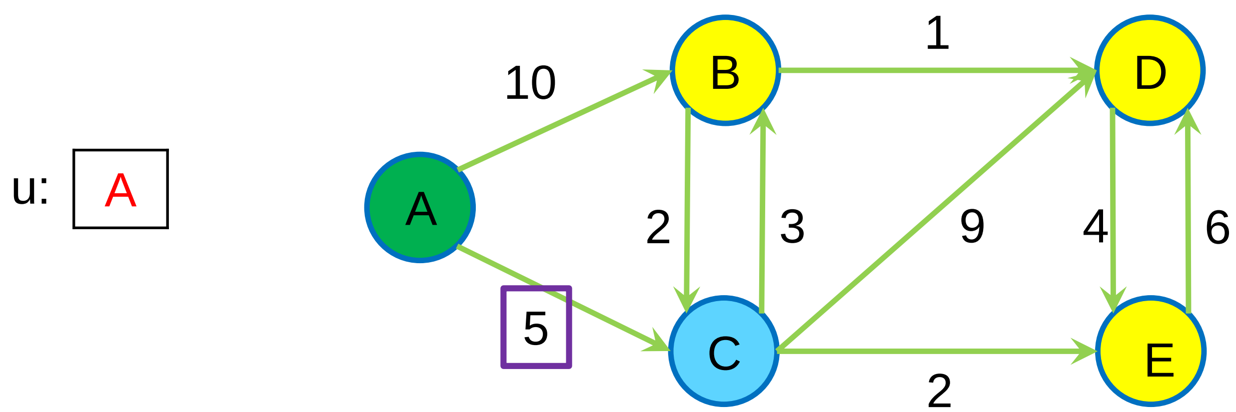
Dist:

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



Q:

B	C	D	E
Inf	Inf	Inf	Inf

Pred:

A	B	C	D	E
-	-	-	-	-

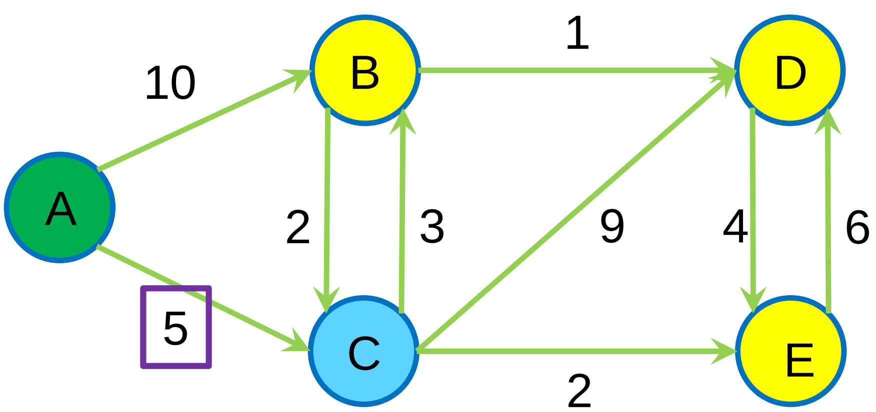
Dist:

A	B	C	D	E
0	Inf	Inf	Inf	inf

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

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

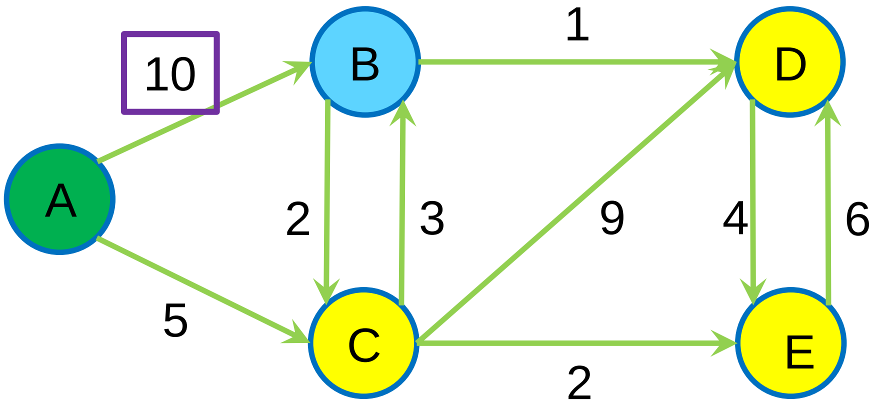
For each neighbour v of u, relax along that edge

- 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:

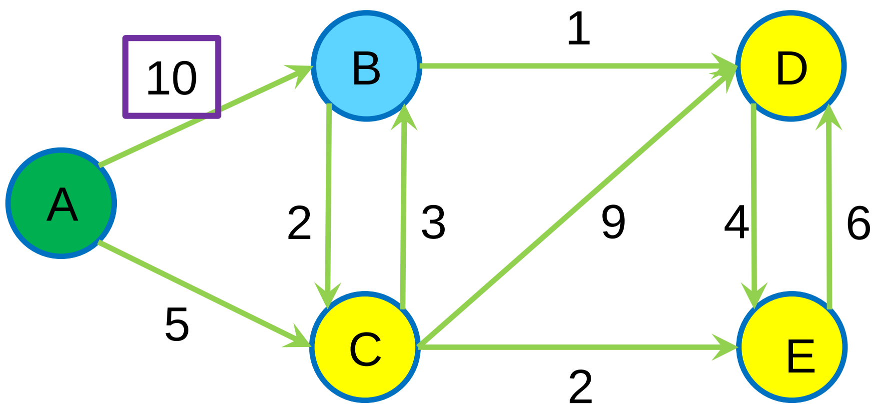
A	B	C	D	E
-	-	A	-	-

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:

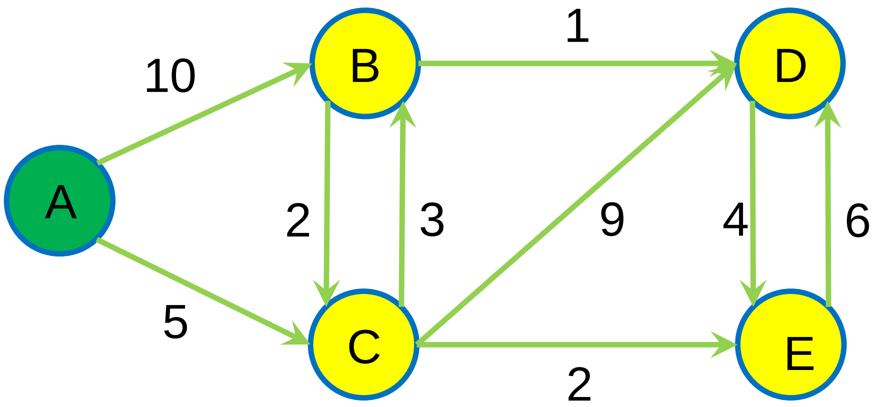
A	B	C	D	E
0	10	5	Inf	inf

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	-	-

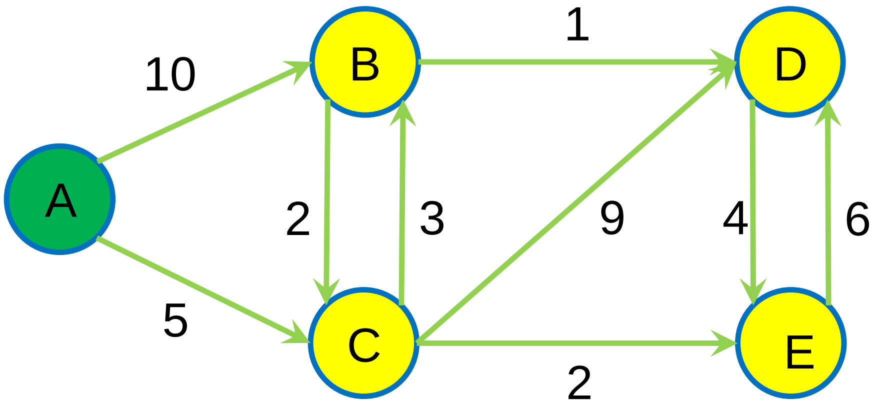
Dist:

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:

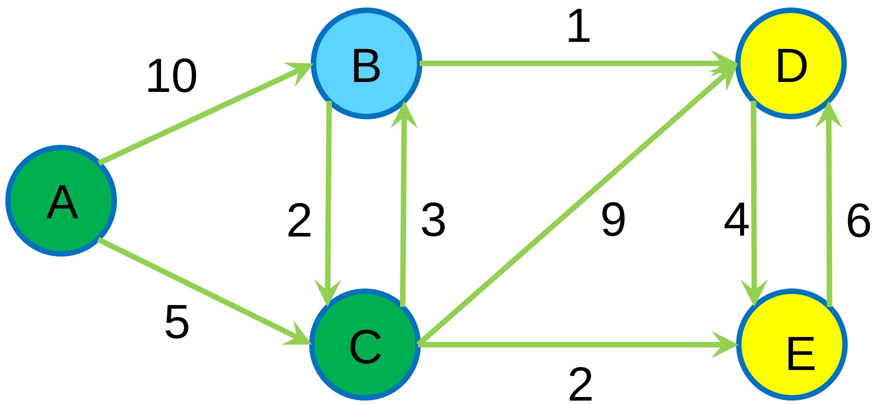
A	B	C	D	E
-	A	A	-	-

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

Q:

B	D	E
10	Inf	Inf

Pred:

A	B	C	D	E
-	A	A	-	-

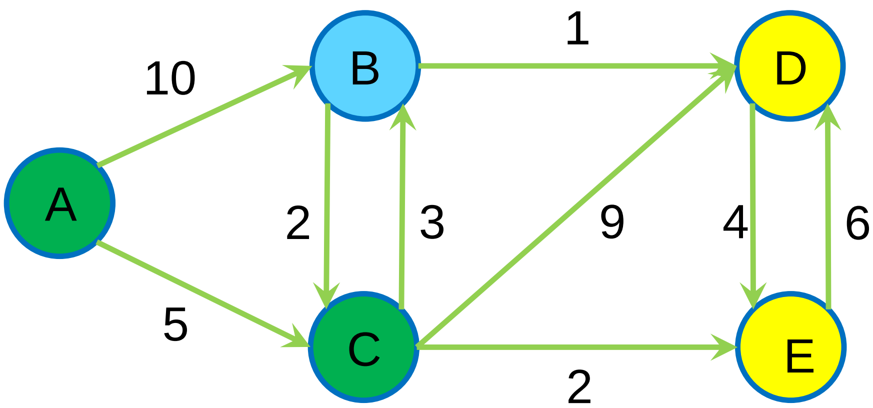
Dist:

A	B	C	D	E
0	10	5	Inf	inf

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

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

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:

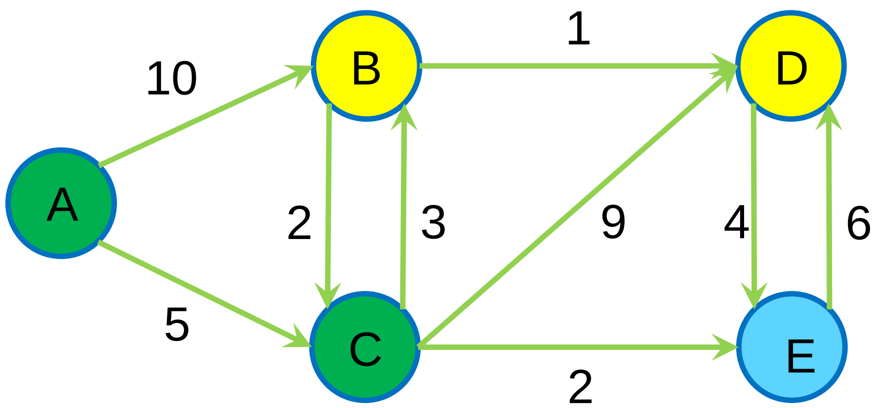
A	B	C	D	E
-	C	A	-	-

Dist:

A	B	C	D	E
0	8	5	Inf	inf

Dijkstra's Algorithm

u: C



Q:

B	D	E
8	Inf	Inf

Pred:

A	B	C	D	E
-	C	A	-	-

Dist:

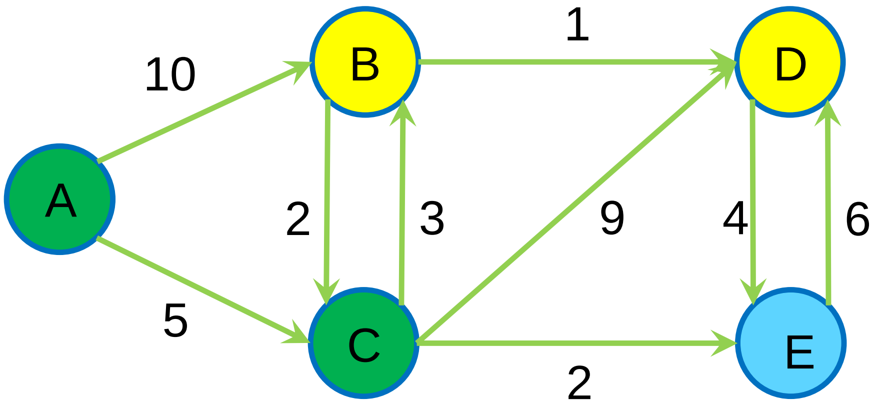
A	B	C	D	E
0	8	5	Inf	inf

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

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

Dijkstra's Algorithm

u: C



Q:

E	B	D
7	8	inf

Pred:

A	B	C	D	E
-	C	A	-	C

Dist:

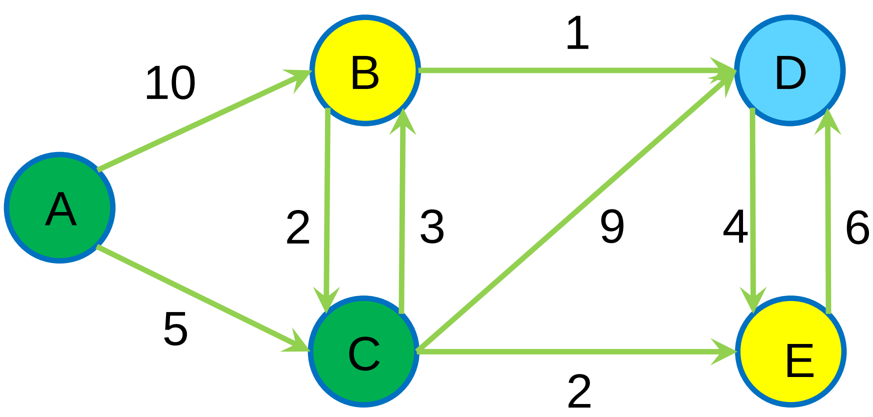
A	B	C	D	E
0	8	5	Inf	7

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

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

Dijkstra's Algorithm

u: C



Q:

E	B	D
7	8	14

Pred:

A	B	C	D	E
-	C	A	C	C

Dist:

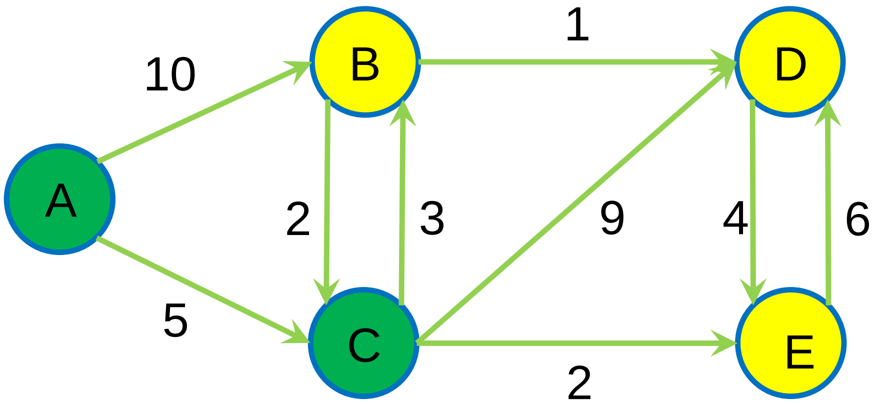
A	B	C	D	E
0	8	5	14	7

- 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

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

Pred:

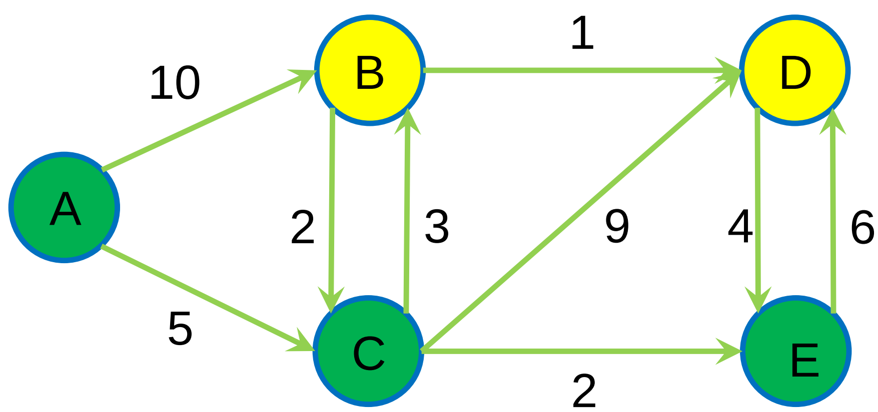
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra's Algorithm

u: E



Q:

B	D
8	14

Pred:

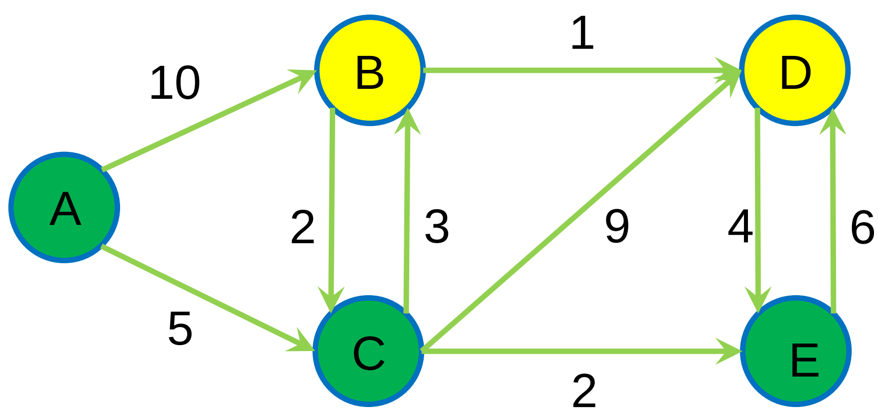
A	B	C	D	E
-	C	A	C	C

Dist:

A	B	C	D	E
0	8	5	14	7

Dijkstra's Algorithm

u: E



- Relax from E

Q:

B	D
8	13

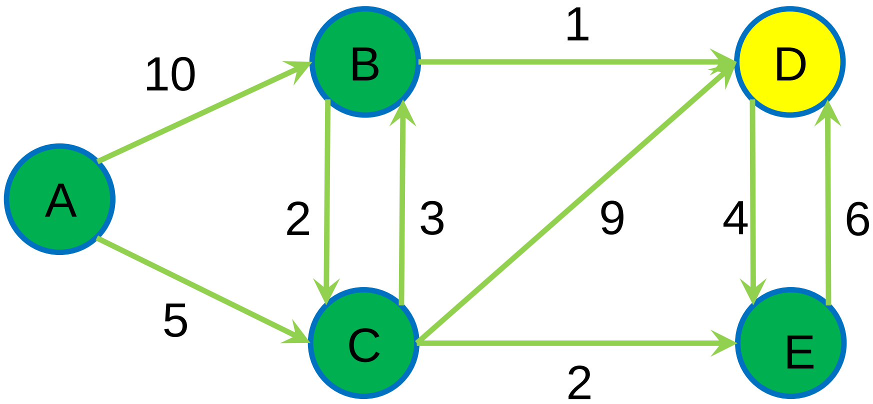
Pred:

A	B	C	D	E
-	C	A	E	C

Dist:

A	B	C	D	E
0	8	5	13	7

Dijkstra's Algorithm



u: B

Q:

D
13

- Done with E
- Pop B

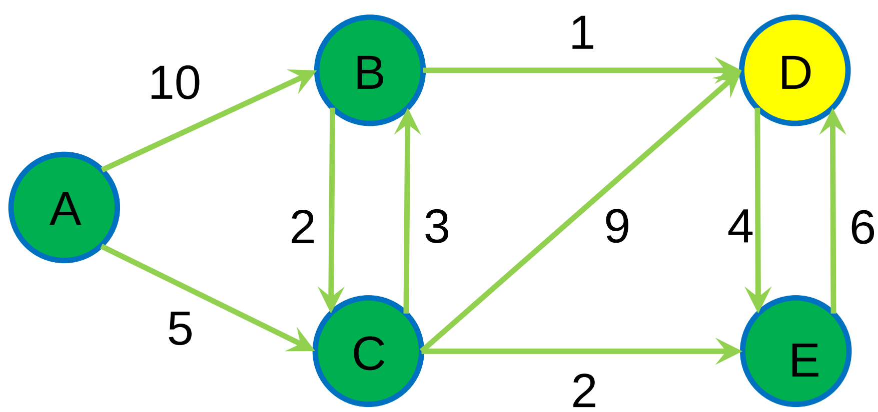
Pred:

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

- Relax from B

Q:

D
9

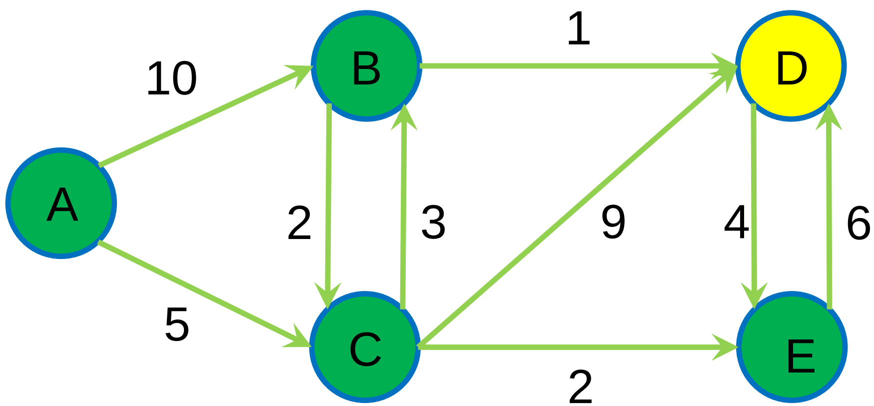
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra's Algorithm



u: D

Q:

- Done with B
- Pop D
- No neighbours to relax
- Done!

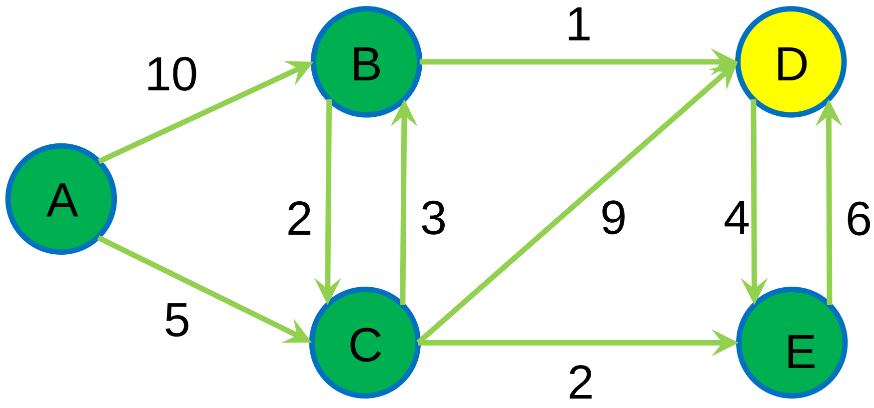
Pred:

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:

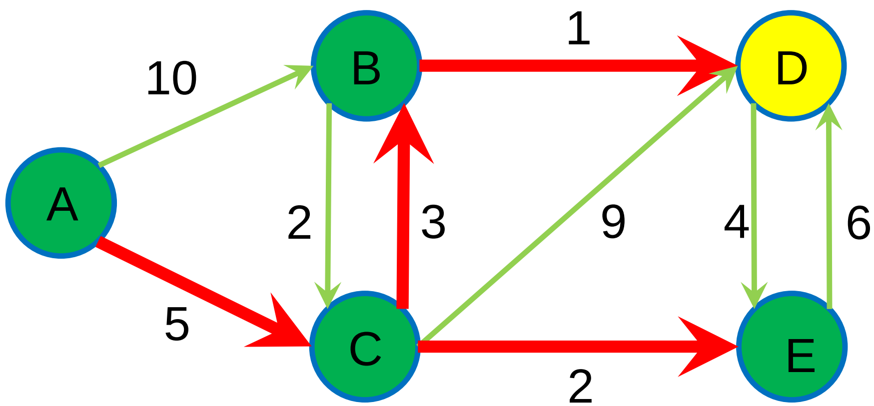
Pred:

A	B	C	D	E
-	C	A	B	C

Dist:

A	B	C	D	E
0	8	5	9	7

Dijkstra's Algorithm



u: D

Q:

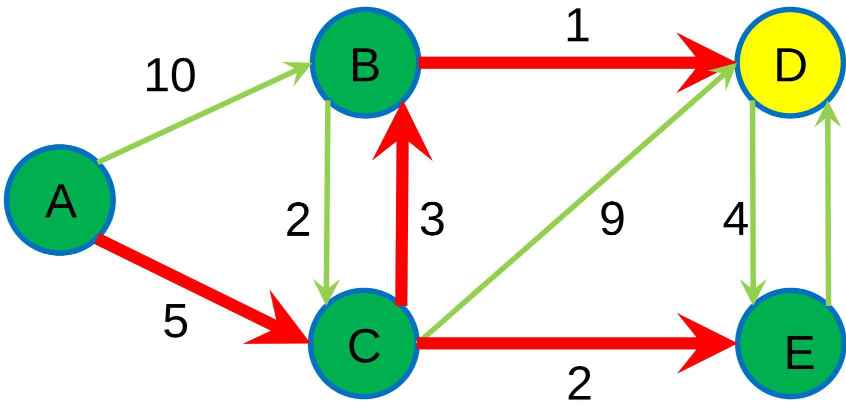
Pred:

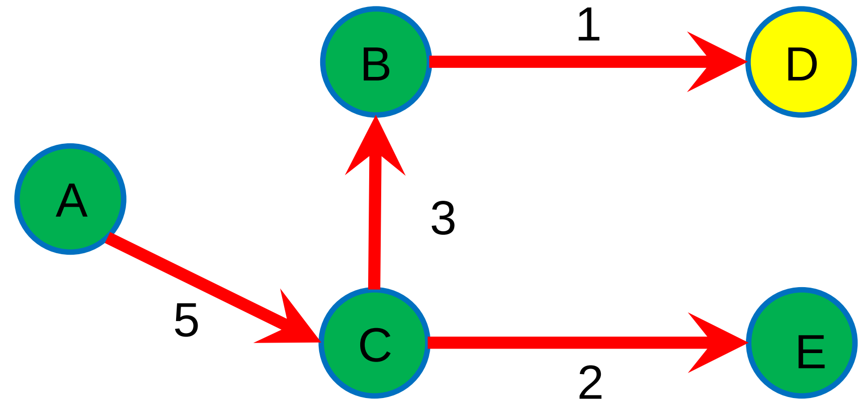
A	B	C	D	E
-	C	A	B	C

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 - RFIBMB

Dijkstra's Algorithm

Algorithm 61 Dijkstra's algorithm

```
function DIJKSTRA(G = (V, E), s)
    dist[1..n] = ∞
    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:

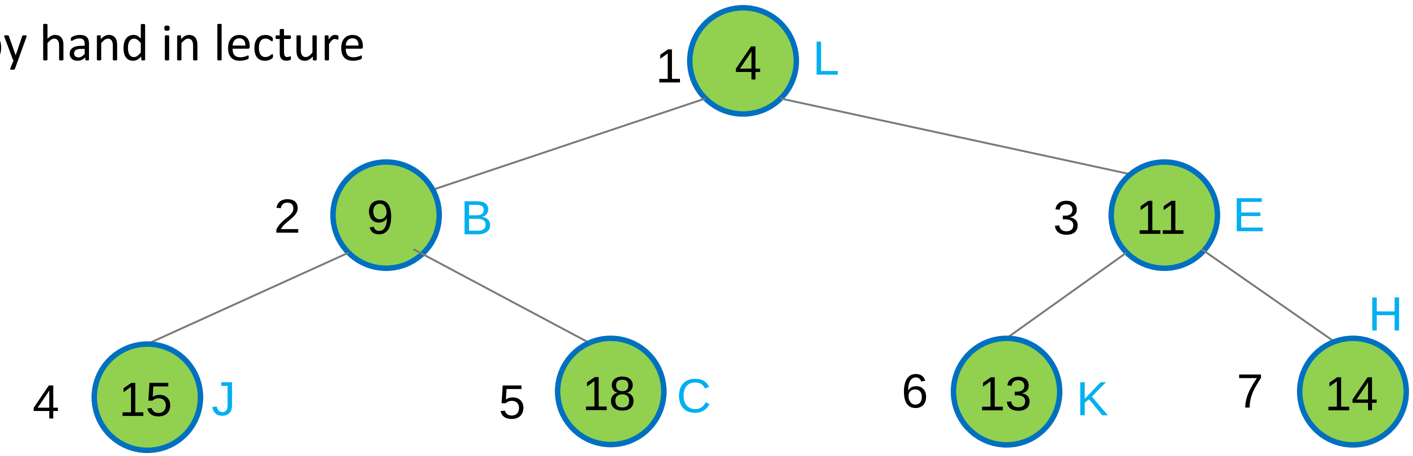
- Create an array called Vertices.
- Vertices[i] will record the location of i-th vertex in the min-heap

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

Dijkstra's Algorithm using min-heap

Example worked by hand in lecture



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

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

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

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 priortiy 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)

- Minimum value of E is V-1 since graph is connected
- E dominates V

- Total cost O(ElogV)

Time Complexity of Dijkstra's Algorithm

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

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

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

Proof of Correctness

Claim: For every vertex v 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

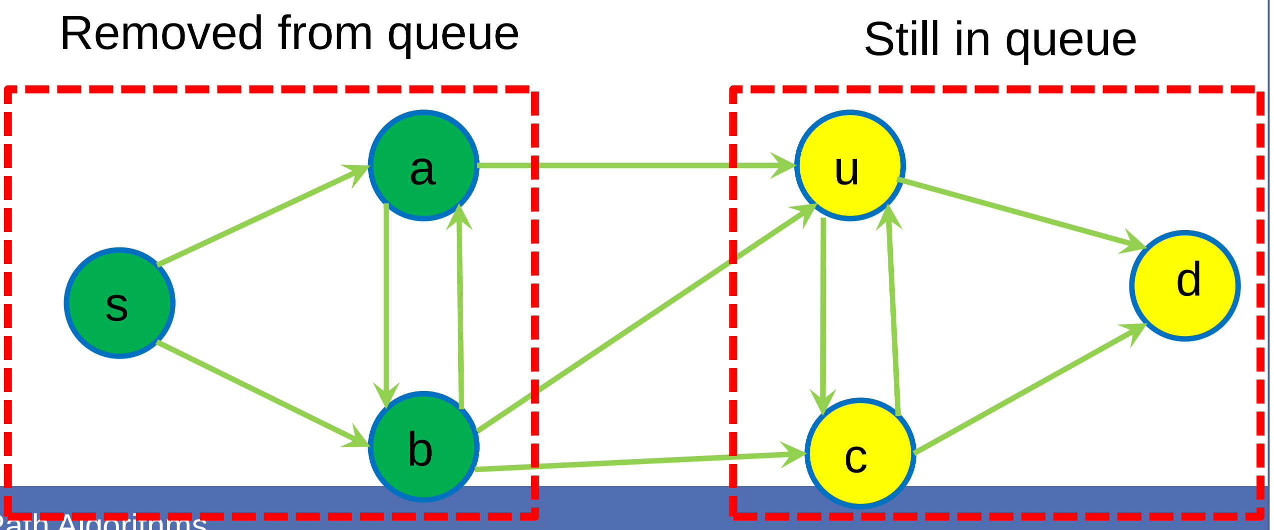
- Dist[s] is initialised to 0, which is the shortest distance from s to s (since there are no negative weights)

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



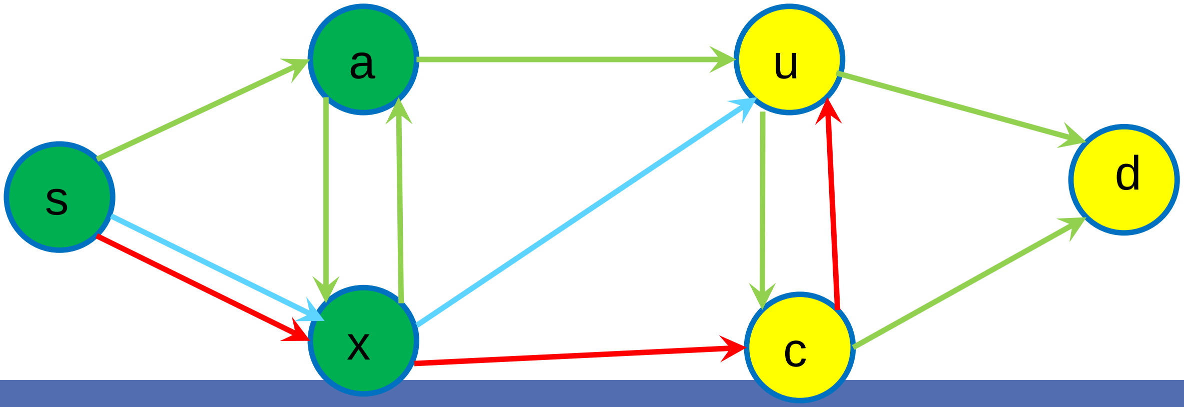
Proof of Correctness

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)

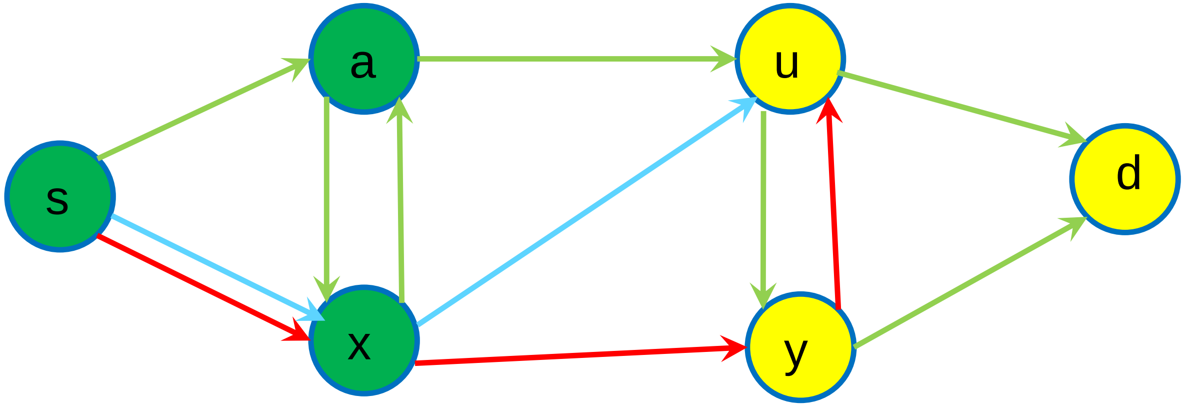


Proof of Correctness

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⇝y) <= len(P) < dist[u]

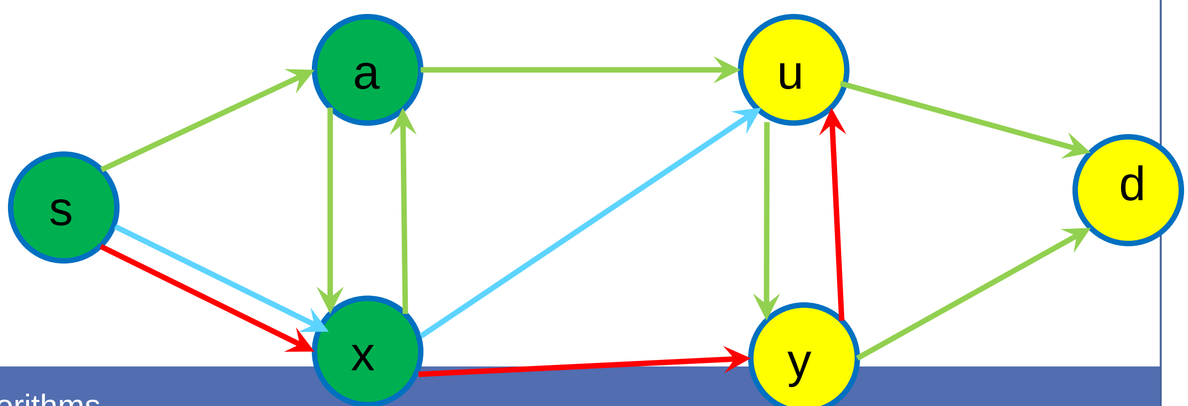


Proof of Correctness

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]???

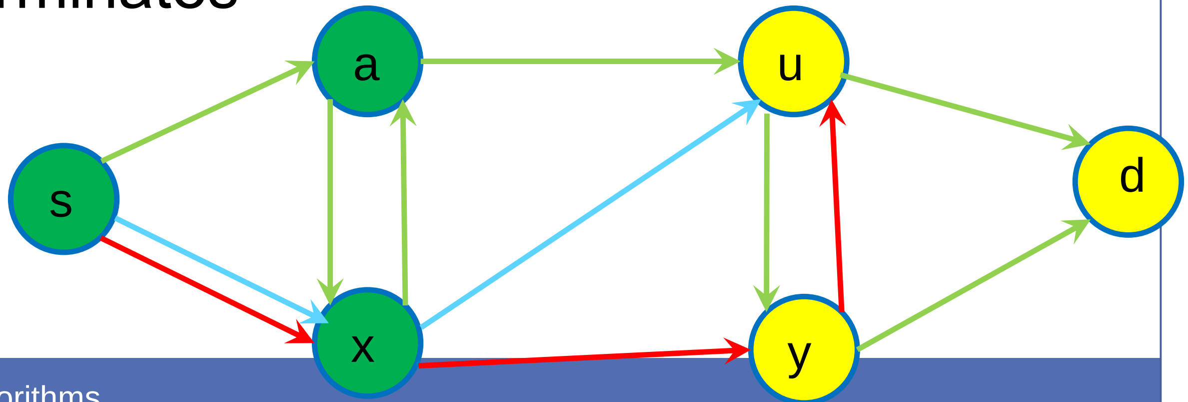


Proof of Correctness

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, $s \rightsquigarrow u$ with len(P) < dist[u]"
- So there is no such path, so dist[u] is correct
- So by induction, the distance of every vertex is correct when Dijkstra's algorithm terminates



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