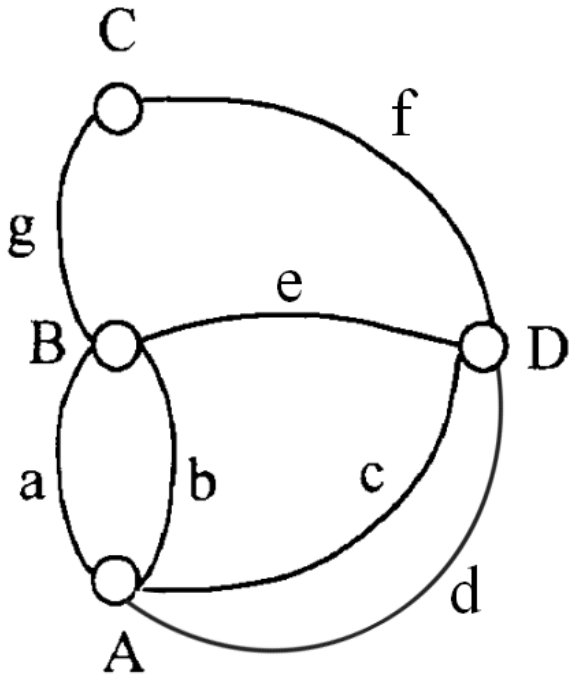
Graph Algorithms

Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

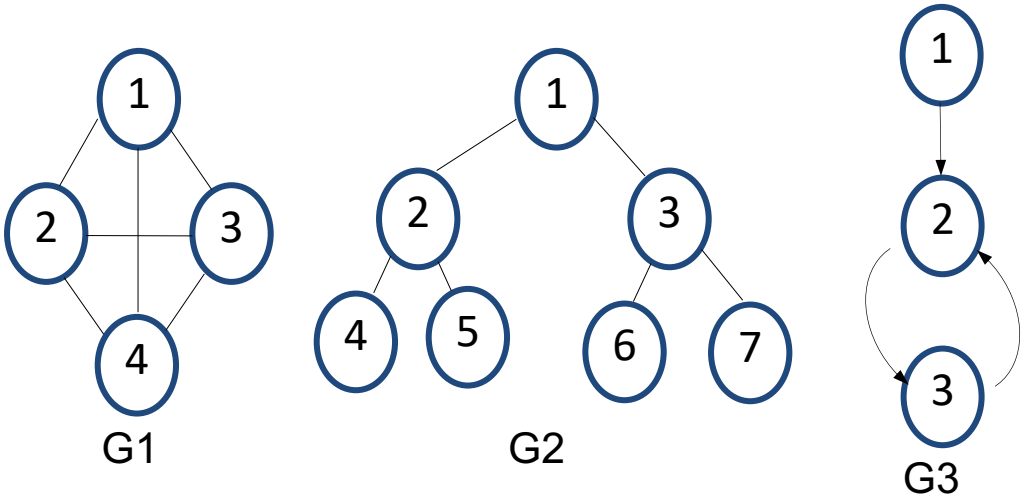
Definition of Graph



- **Vertex (V)**
  - A, D, B, D
- **Edge (E)**
  - BC, CD,…
- **degree (deg)**
  - The branch of a vertex
- **Path (P)**
  - A sequence of connected vertices, e.g. ADCB
- **Cycle (C)**
  - A sequence of connected vertices with same end points

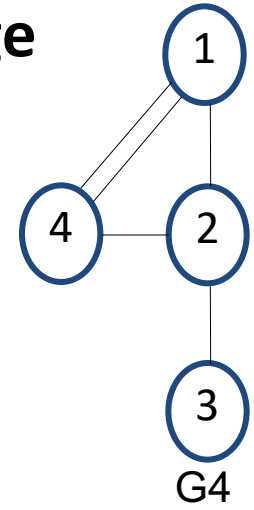
Definition of Graph

- **Undirected Graph – G1, G2**
- **Directed Graph – G3**
  - Indegree and outdegree



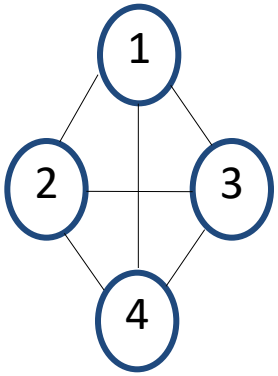
Definition of Graph

- **Multiple Edge**



G4

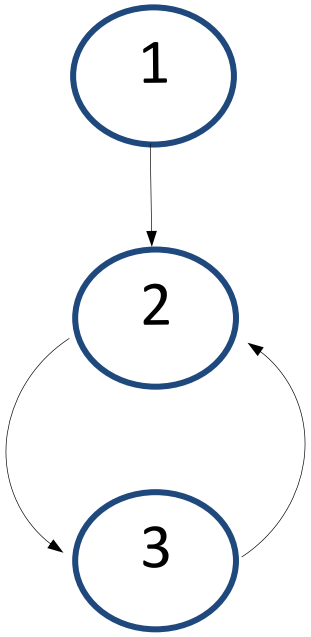
- **Complete Graph**



G1

Definition of Graph

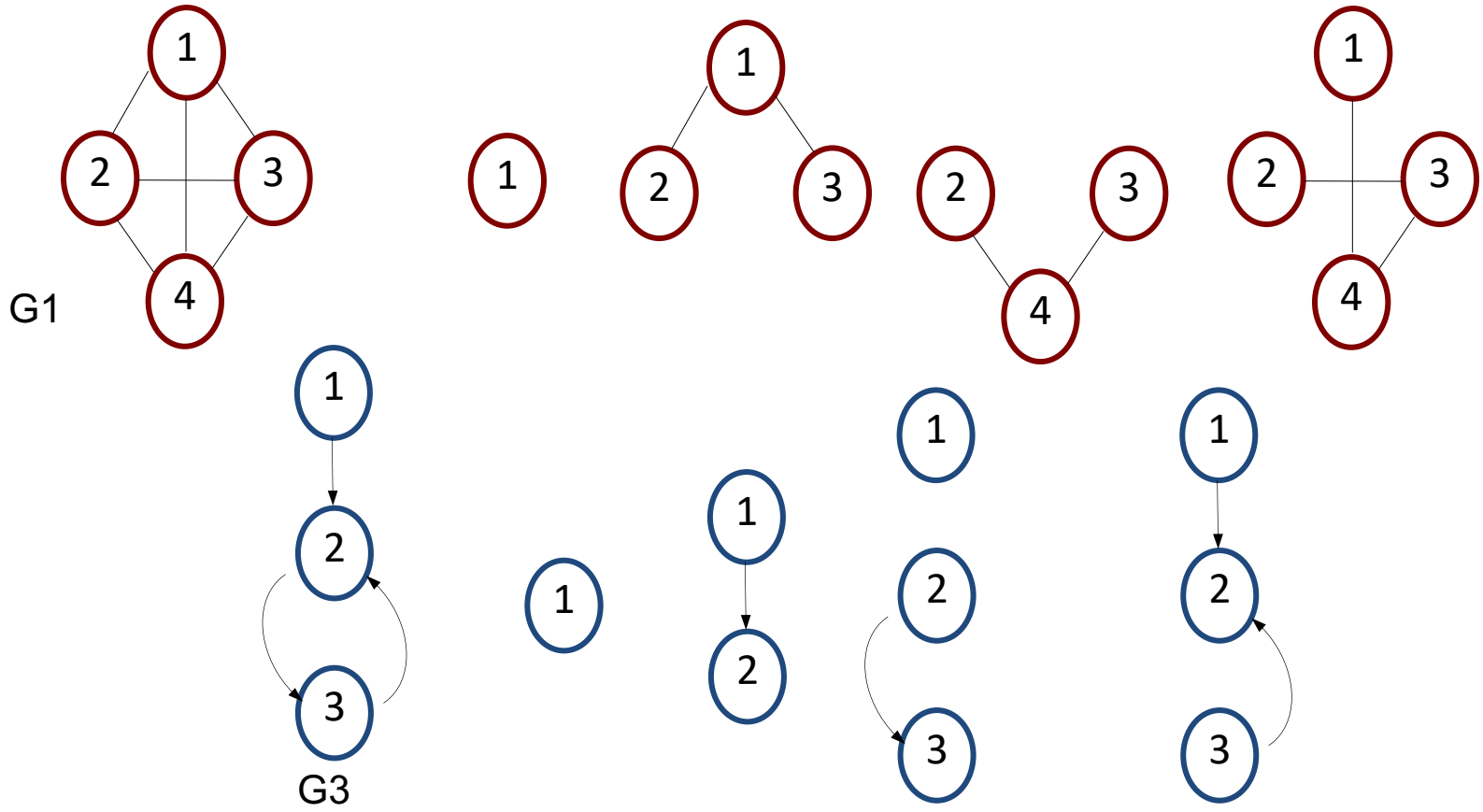
- **Adjacent**
  - for vertex
- **Incident**
  - for edge



G3

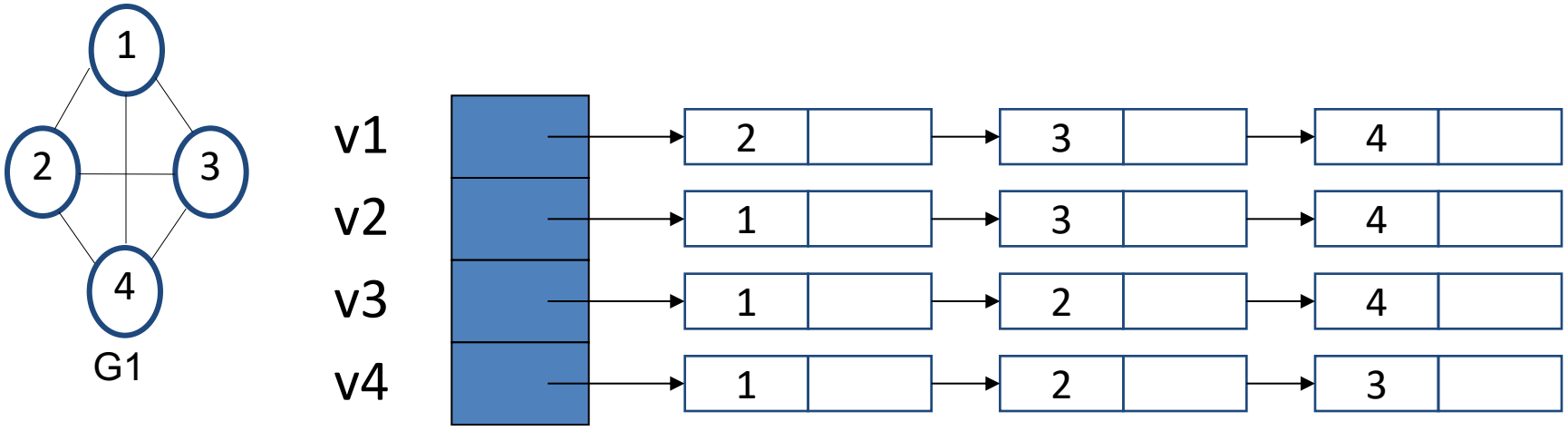
Definition of Graph

- **Subgraph**



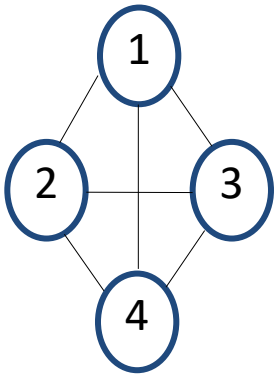
Graph Data Structure

- **Adjacency list**
  - std::vector<std::vector<int>>



Graph Data Structure

- **Adjacency Matrix**



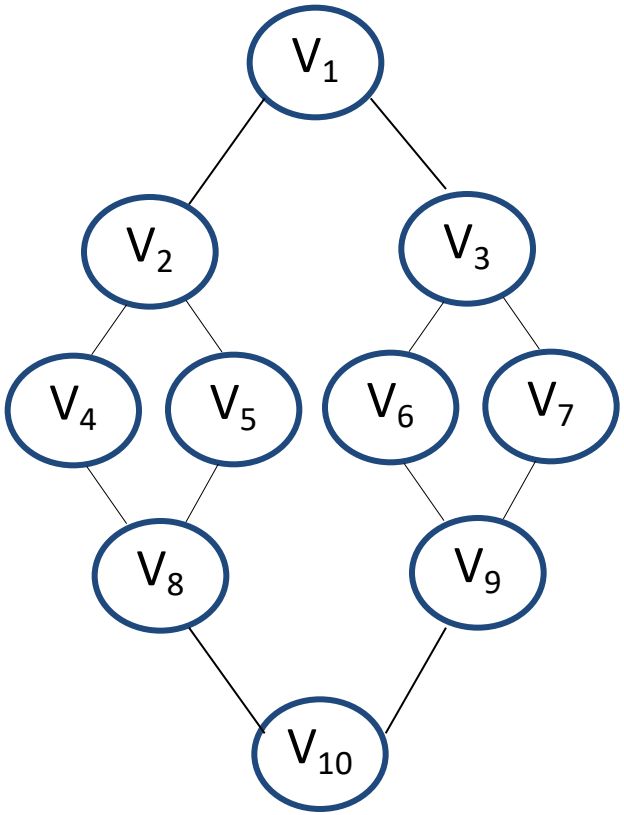
G1

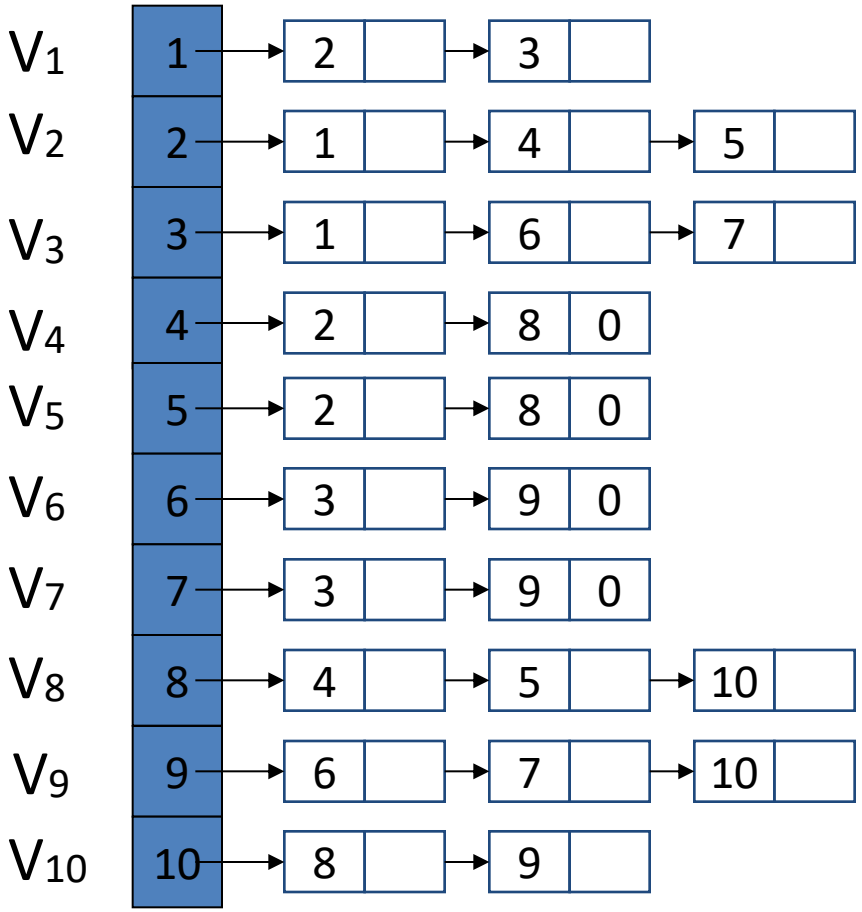
	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Graph Traversal

- **Depth First Search (DFS)**
  - Stack
  - The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- **Breadth First Search (BFS)**
  - Queue
  - The algorithm starts at the root node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

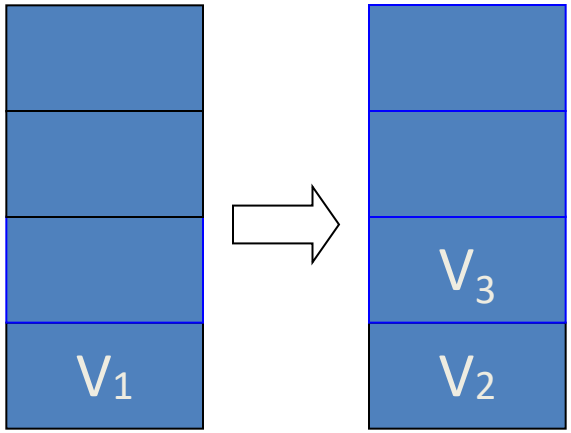
DFS

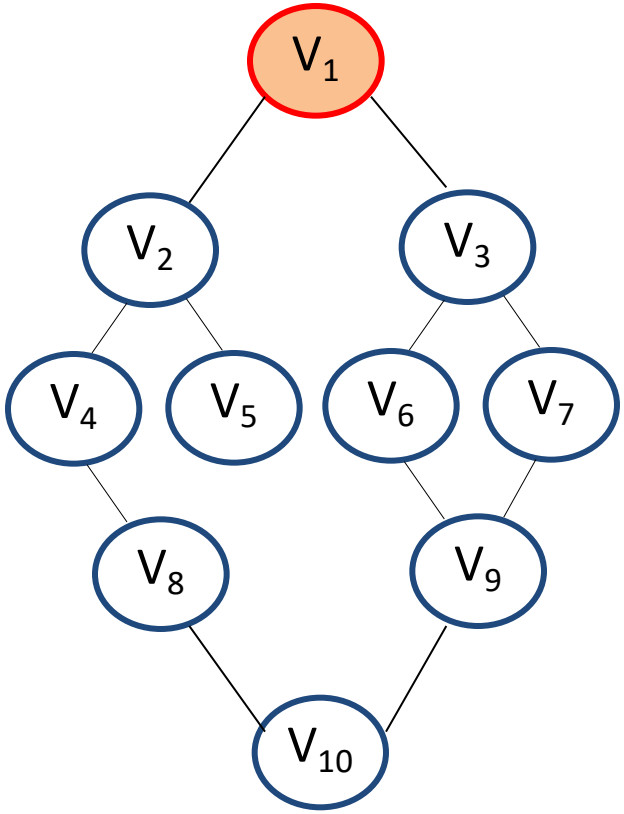

V1
V2
V3
V4
V5
V6
V7
V8
V9
V10


V1
1
2
3
V2
2
1
4
5
V3
3
1
6
7
V4
4
2
8
0
V5
5
2
8
0
V6
6
3
9
0
V7
7
3
9
0
V8
8
4
5
10
V9
9
6
7
10
V10
10
8
9

DFS

- Start from v1

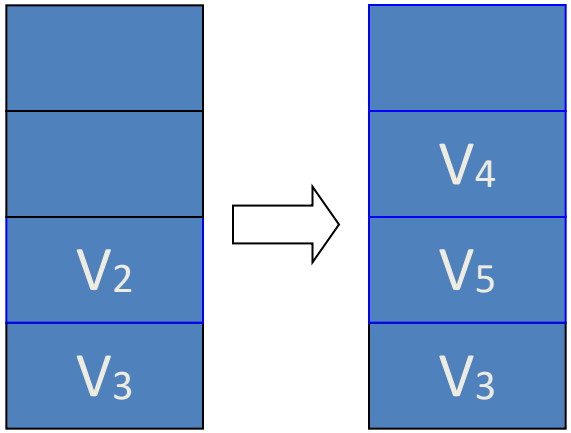


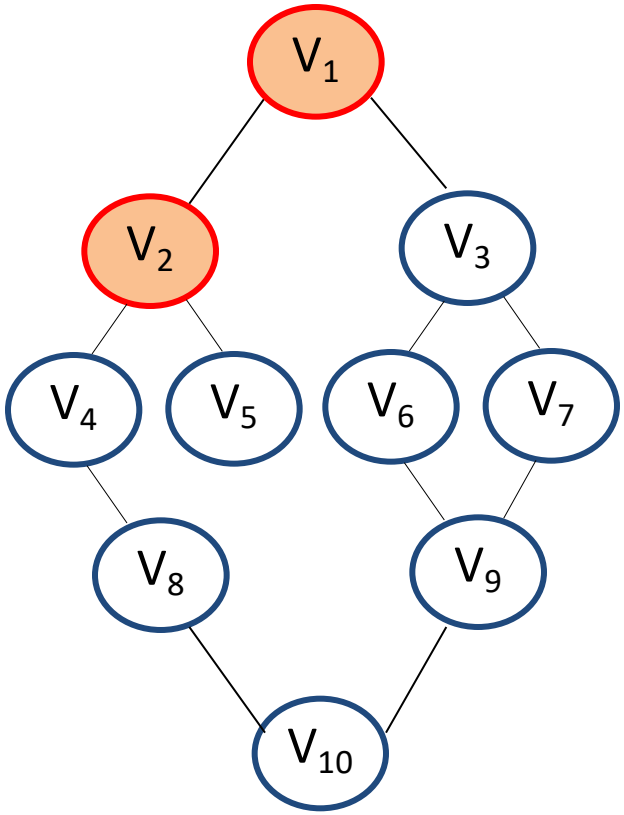


output：V_1

DFS

- Pop a node from the stack and insert its neighbors

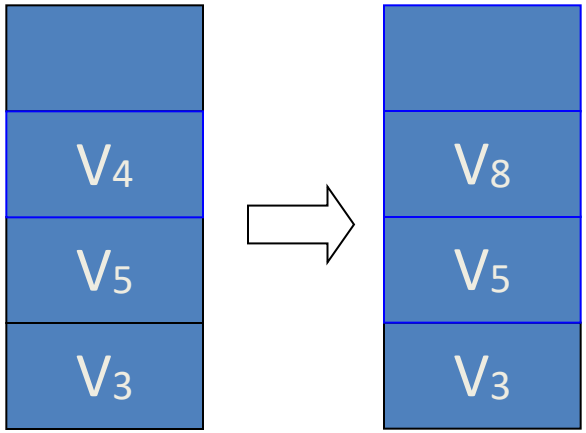


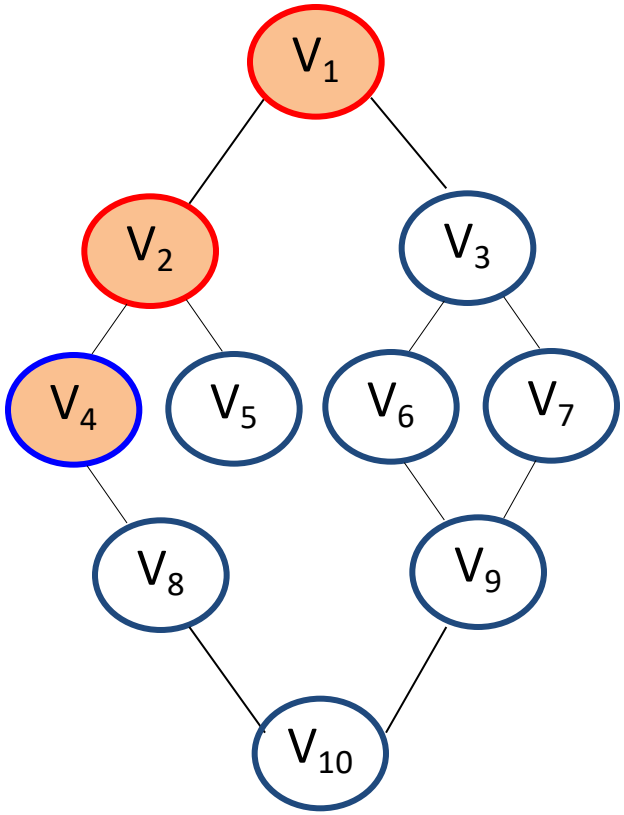


output：V_1 V_2

DFS

- Pop v4 from the stack and insert v8

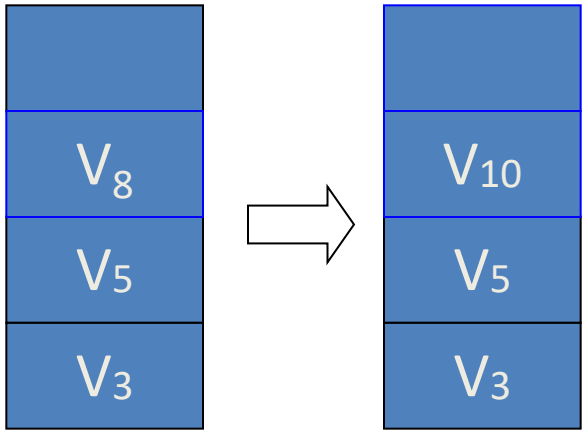


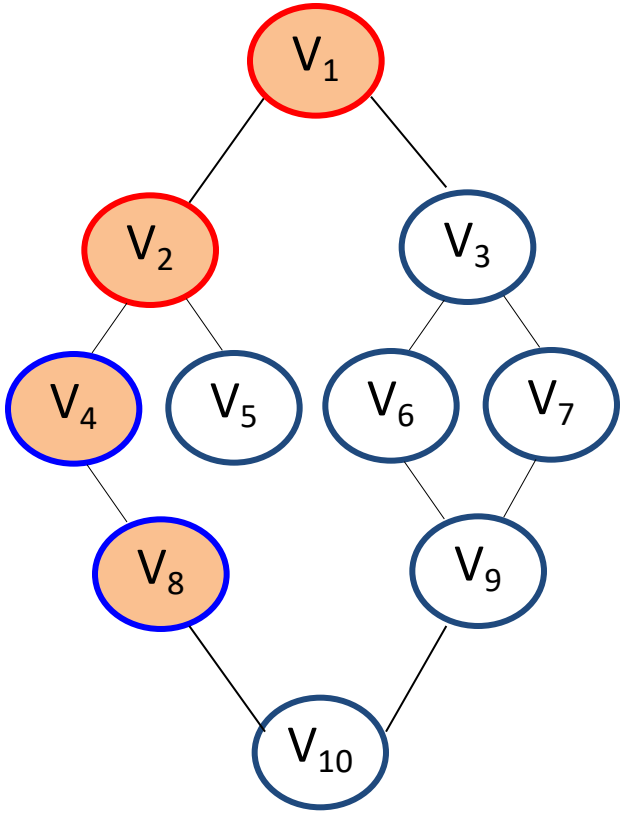


output：V_1 V_2 V_4

DFS

- Pop v10 from the stack and insert v9

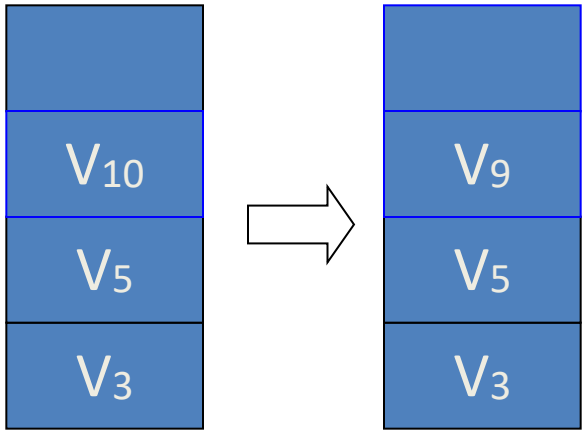


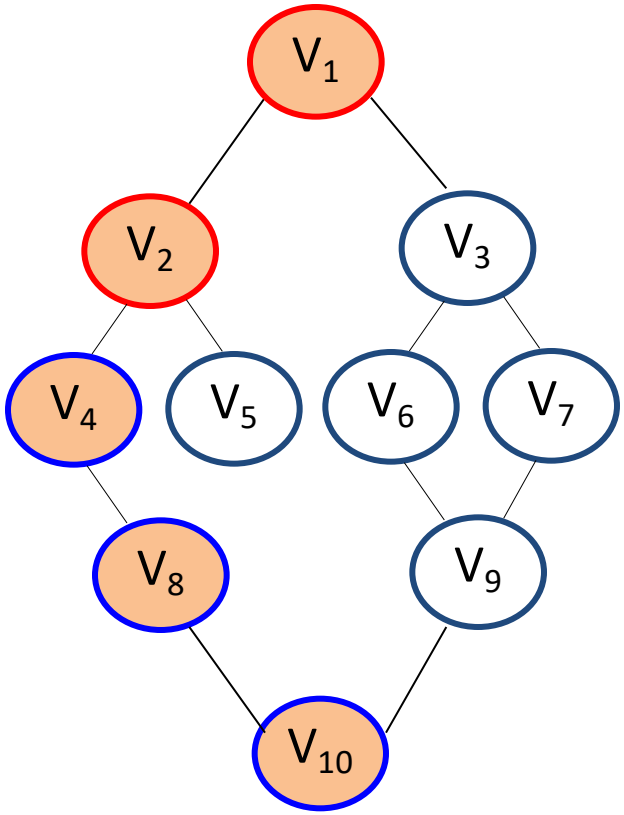


output：V_1 V_2 V_4 V_8

DFS

- Pop v8 from the stack and insert v10

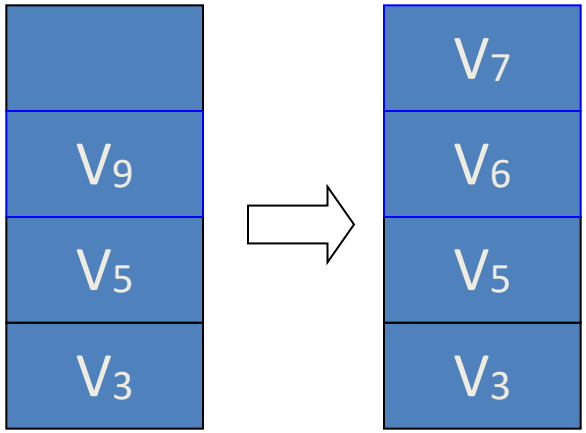


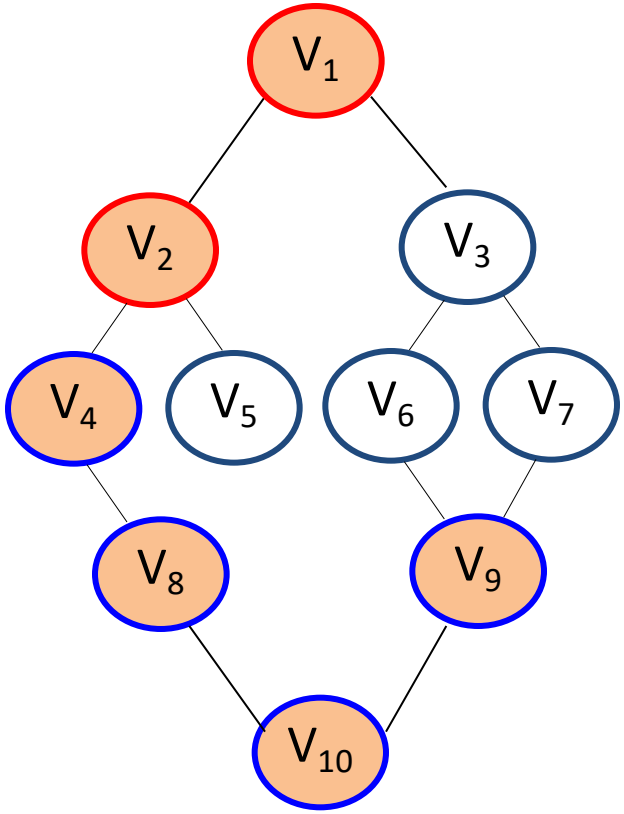


output：V_1 V_2 V_4 V_8 V_{10}

DFS

- Pop v10 from the stack and insert v9

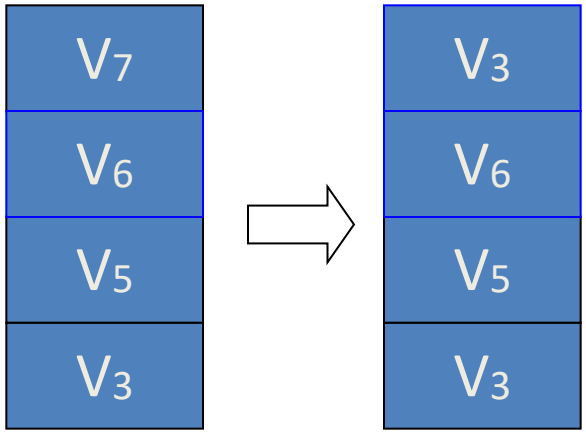


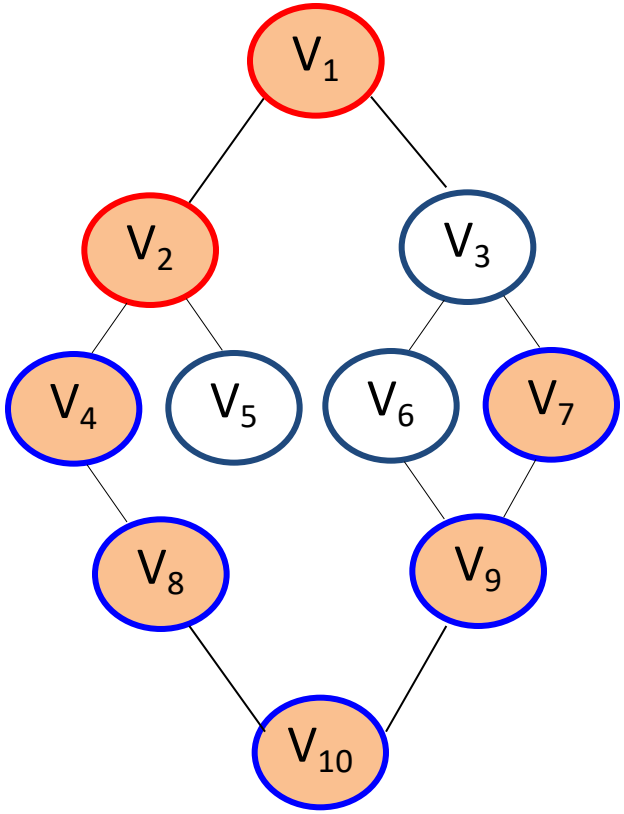


output：V_1 V_2 V_4 V_8 V_{10} V_9

DFS

- Pop v7 from the stack and insert v3

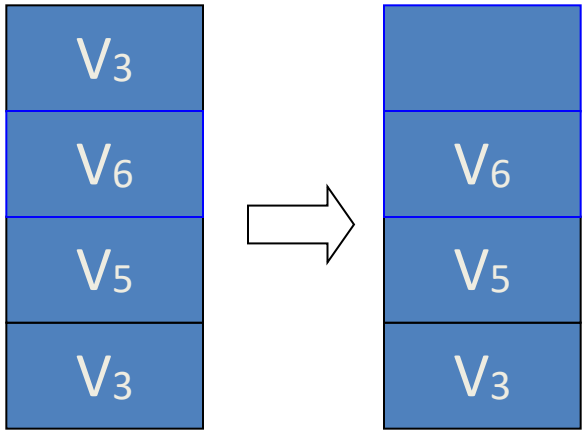


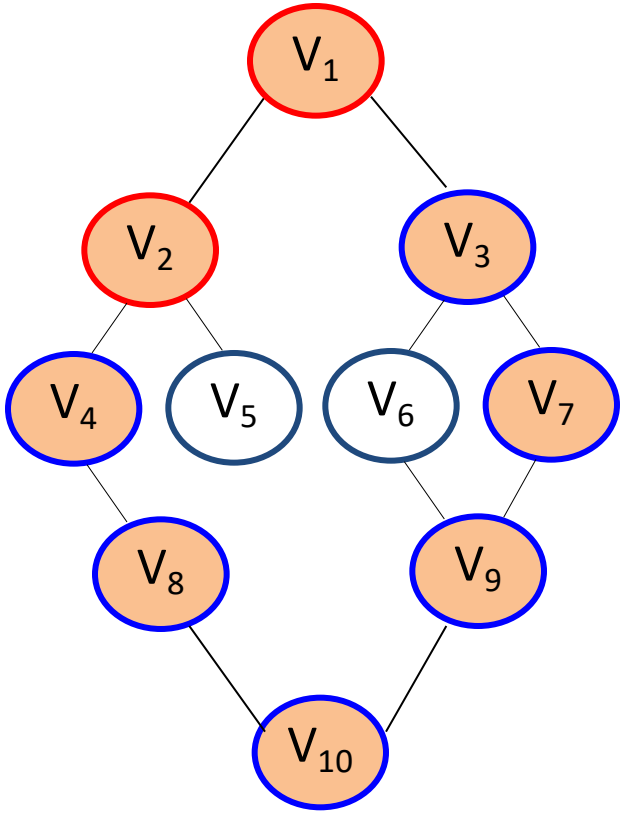


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7

DFS

- Pop v3 from the stack

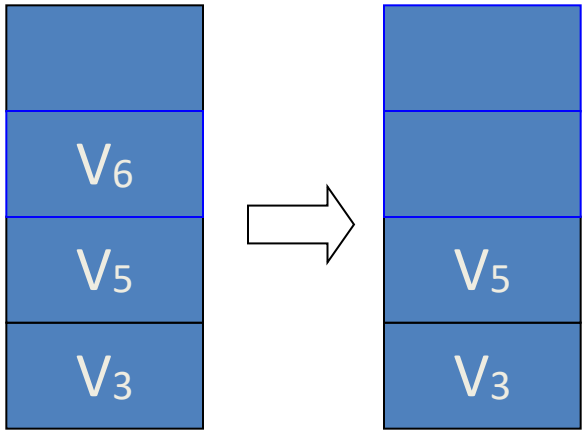


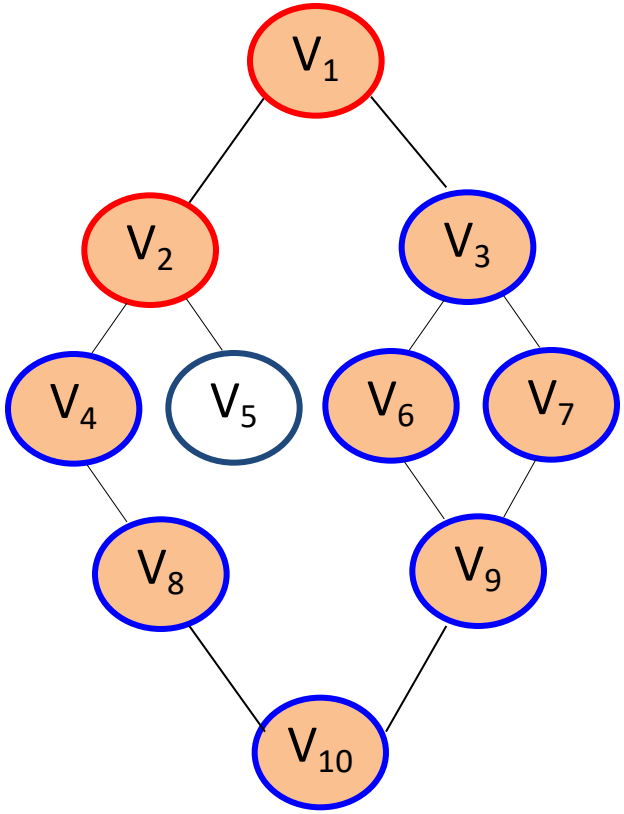


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3

DFS

- Pop v6 from the stack

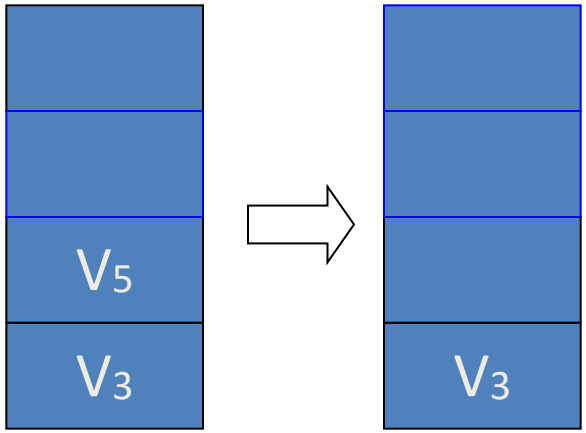


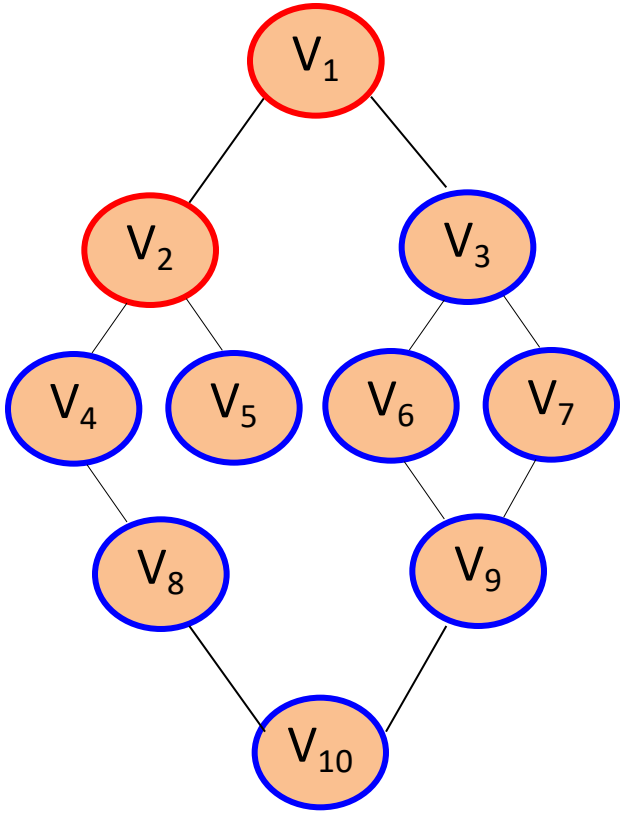


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6

DFS

- Pop v5 from the stack

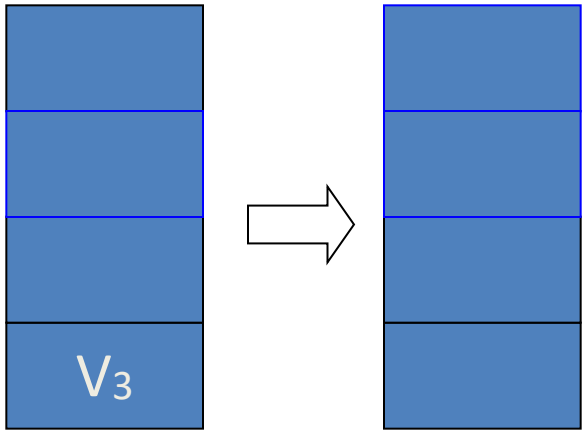


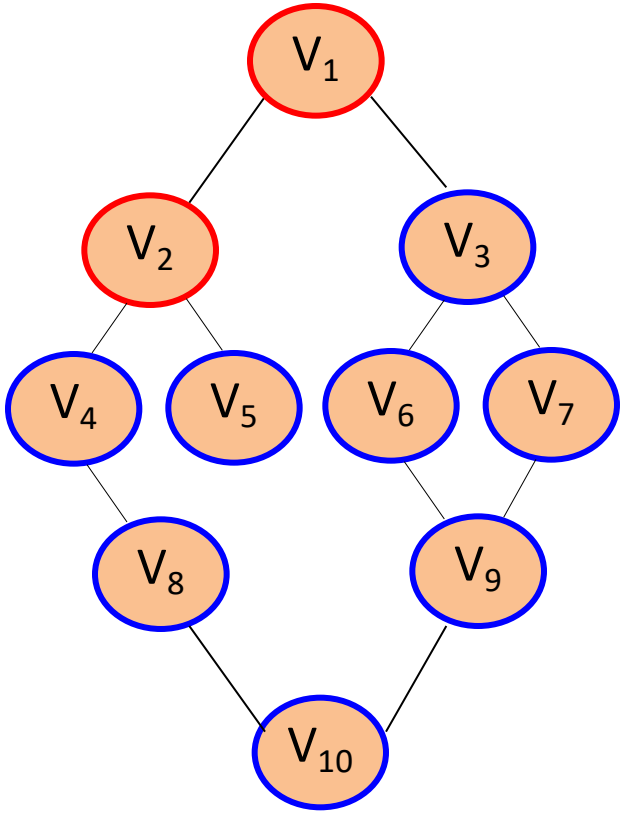


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6 V_5

DFS

- **Pop v3 from the stack**
  - No output for v3 as we did it before!





output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6 V_5

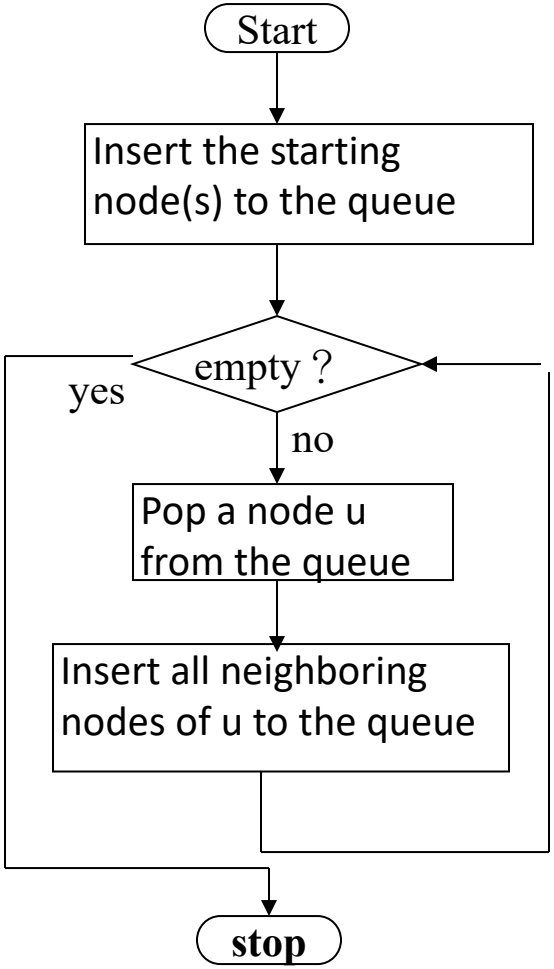
DFS Implementation in Stack

```
// Initially mark all vertices as not visited
vector<bool> visited(V, false);

// Create a stack for DFS
stack<int> stack;

// Push the current source node.
stack.push(s);

while (!stack.empty())
{
    // Pop a vertex from stack and print it
    s = stack.top();
    stack.pop();

    // Stack may contain same vertex twice. We print the popped item only if it is not visited.
    if (!visited[s])   {
        cout << s << " ";
        visited[s] = true;
    }

    // Get all nonvisited adjacent vertices of the popped vertex s
    for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
        if (!visited[*i])  stack.push(*i);
}
```

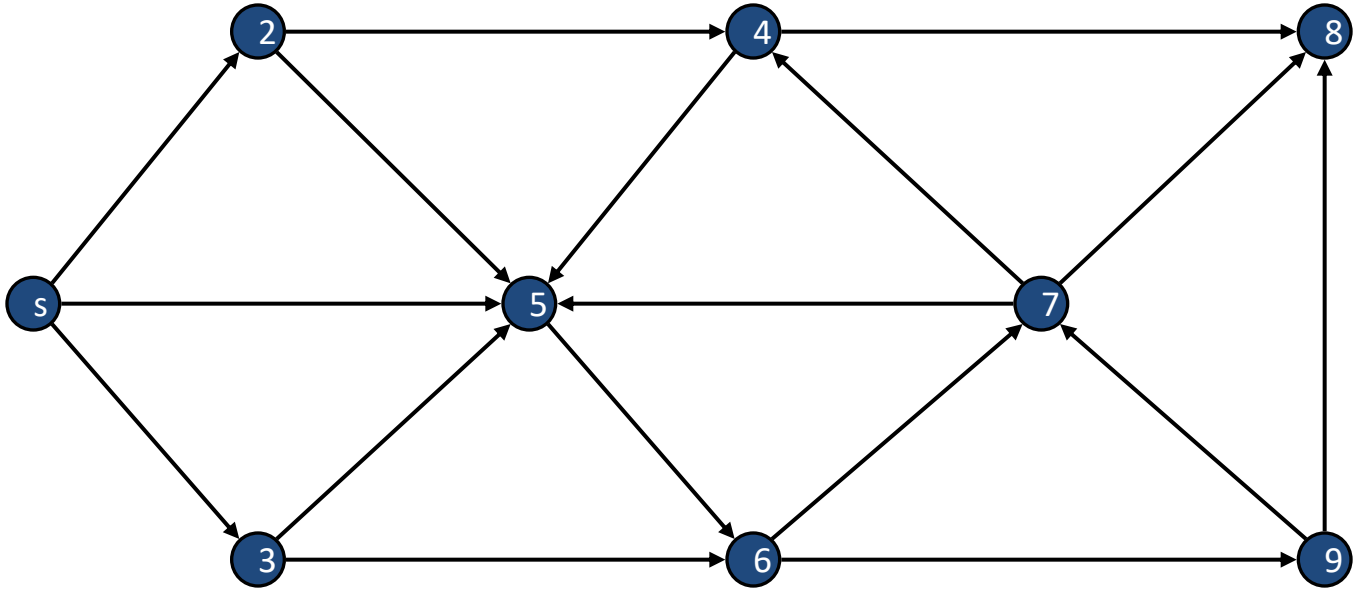
How can we write DFS recursively?

```
procedure DFS(G, v) is
    label v as discovered
    for all directed edges from v to w that are in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G, w)
```

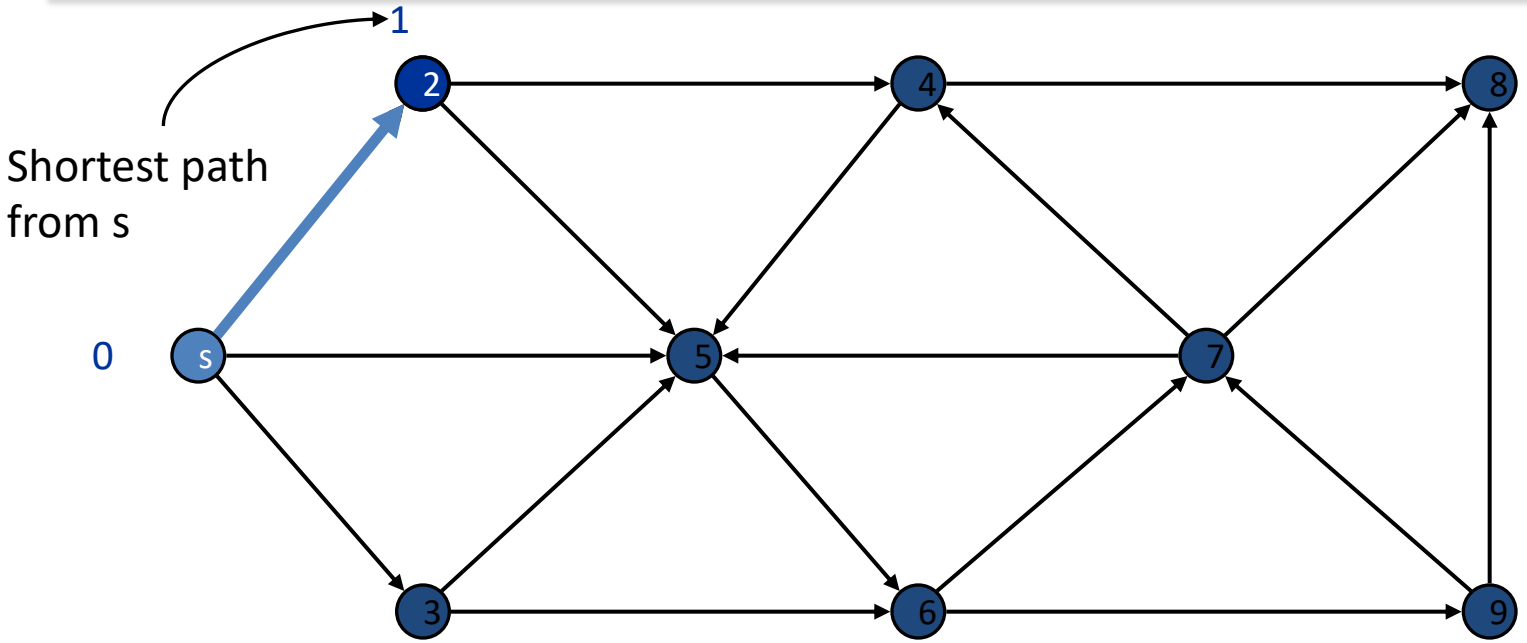
BFS


Start
Insert the starting node(s) to the queue
empty ?
yes
no
Pop a node u from the queue
Insert all neighboring nodes of u to the queue
stop

BFS



BFS



Undiscovered

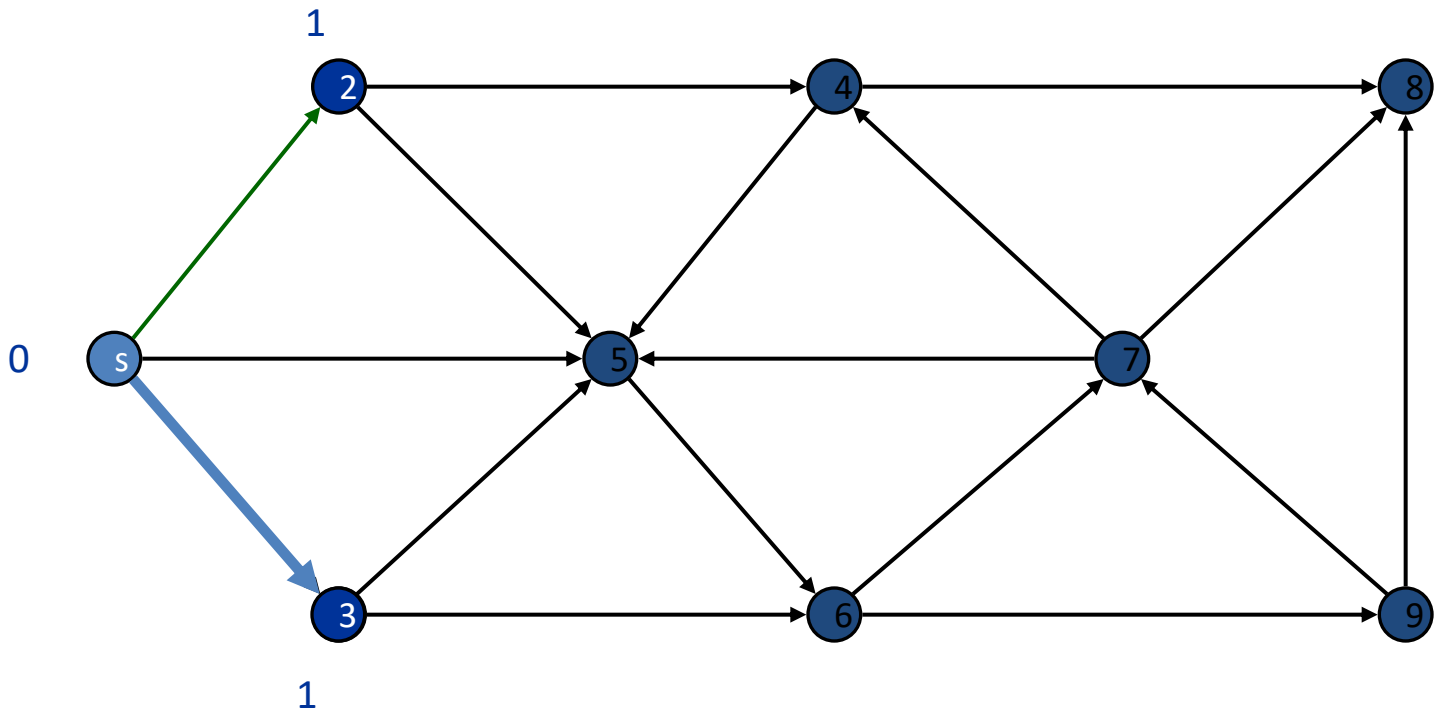
Discovered

Top of queue

Finished

Queue: s

BFS



Undiscovered

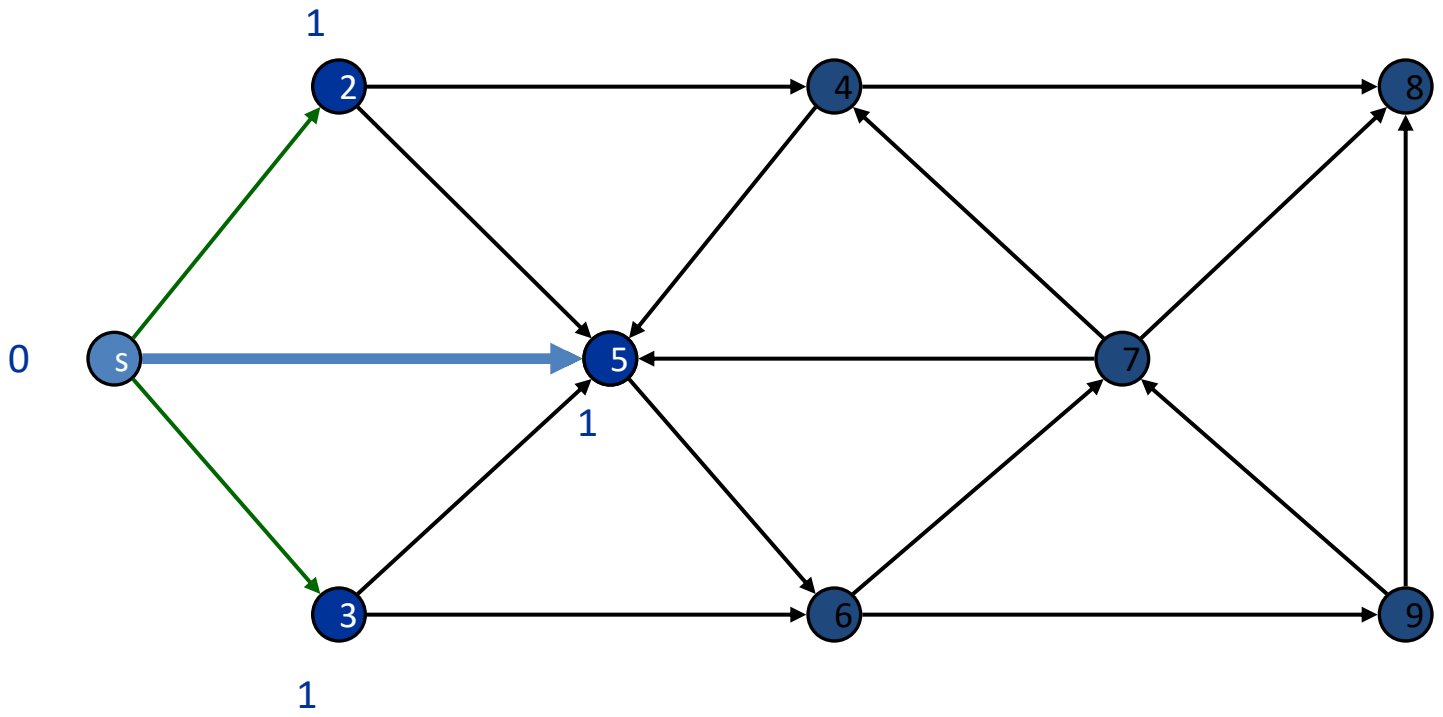
Discovered

Top of queue

Finished

Queue: s 2

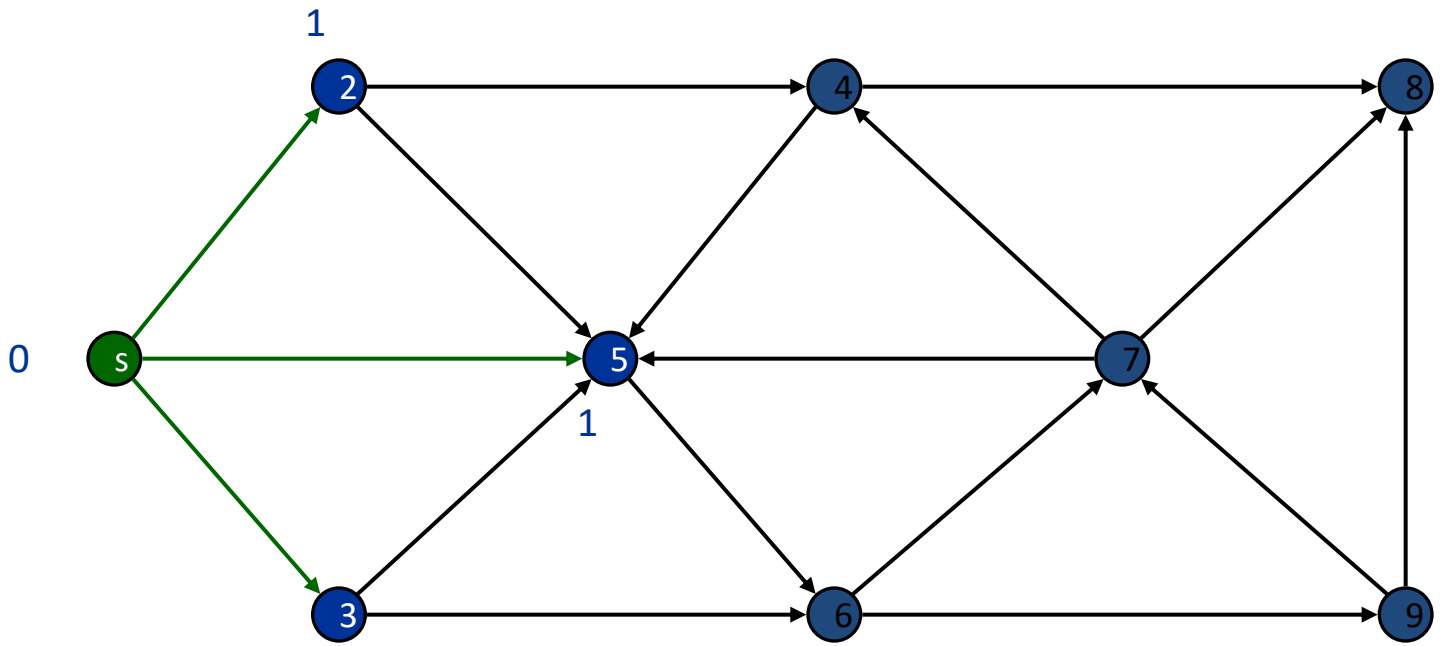
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: s 2 3

BFS



Undiscovered

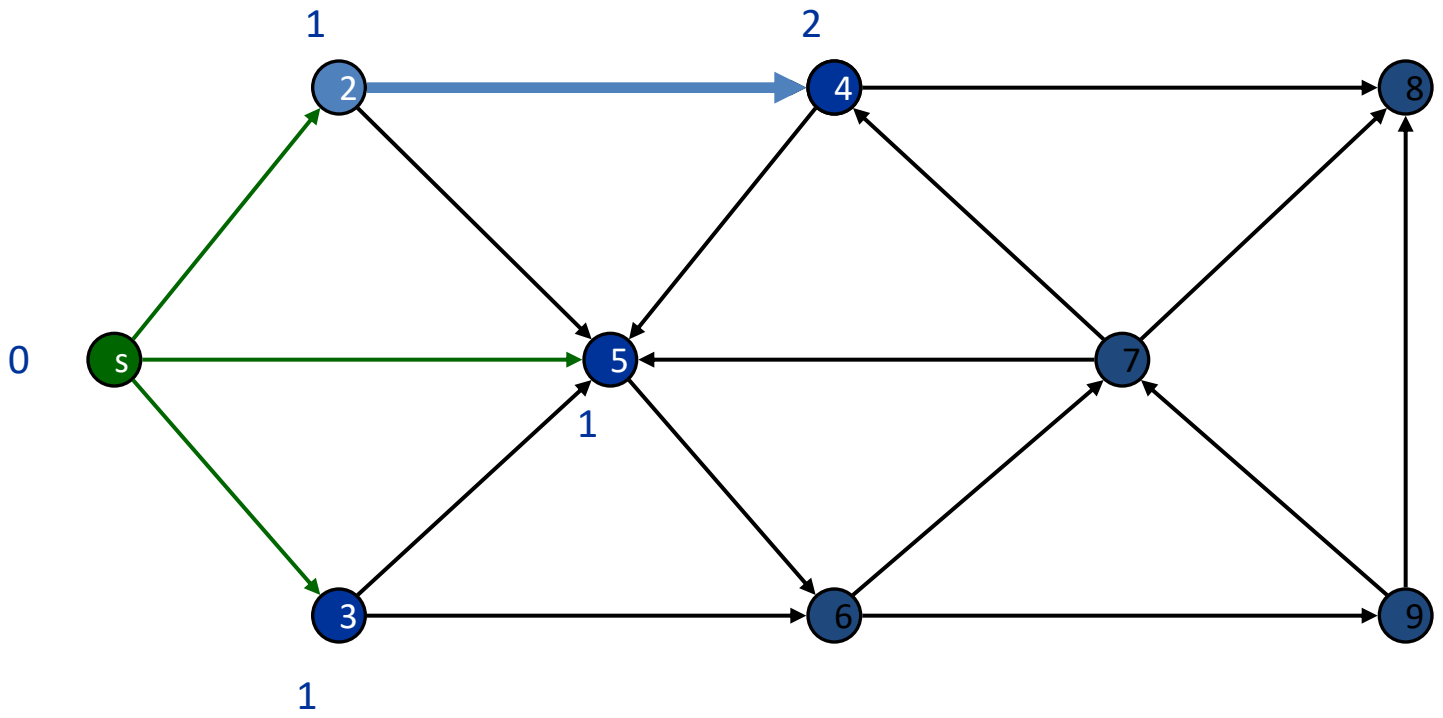
Discovered

Top of queue

Finished

Queue: 2 3 5

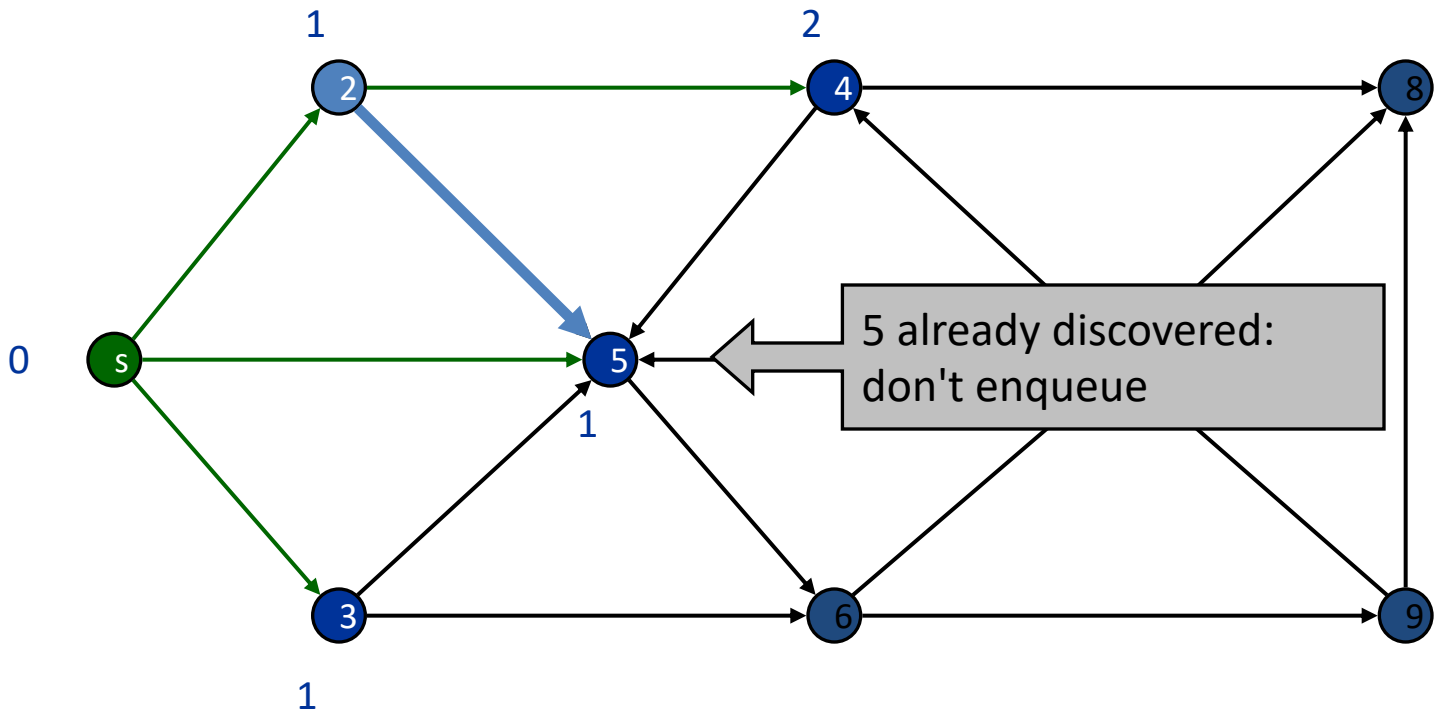
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5

BFS



Undiscovered

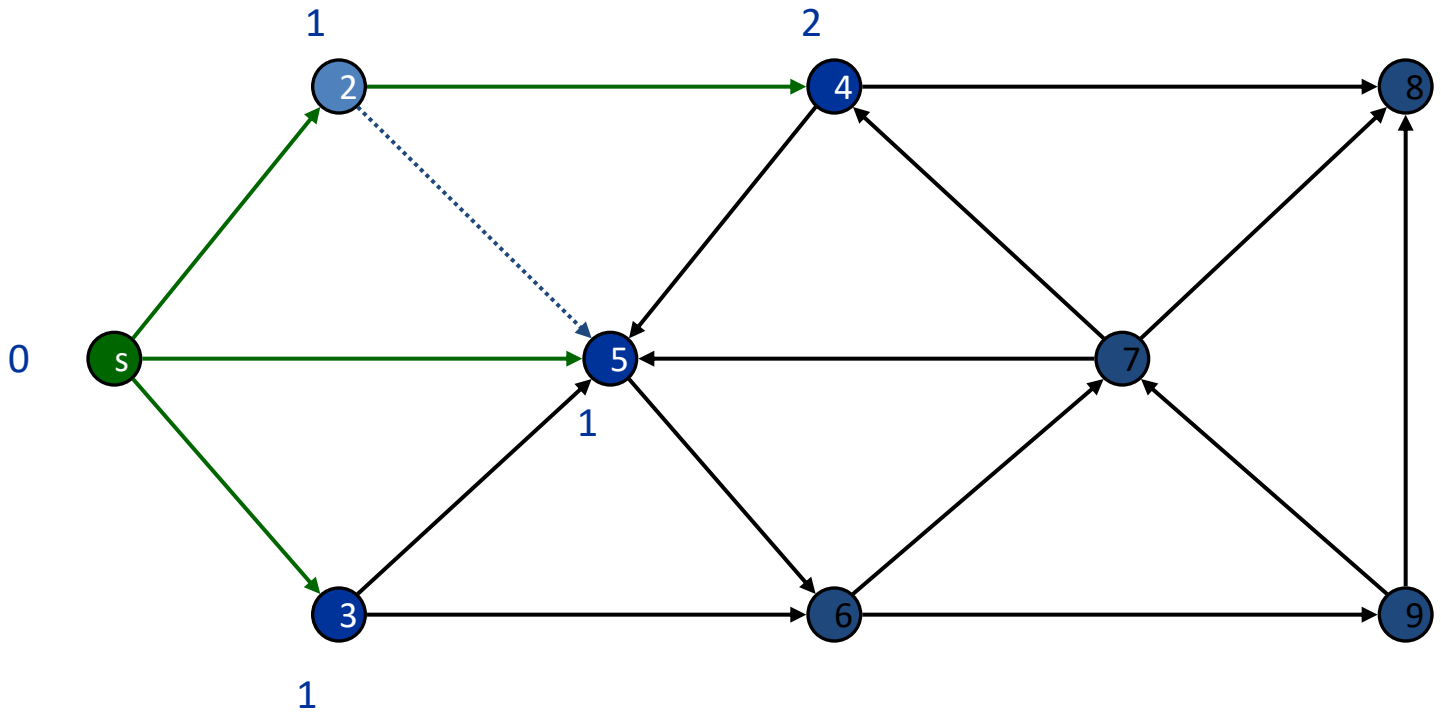
Discovered

Top of queue

Finished

Queue: 2 3 5 4

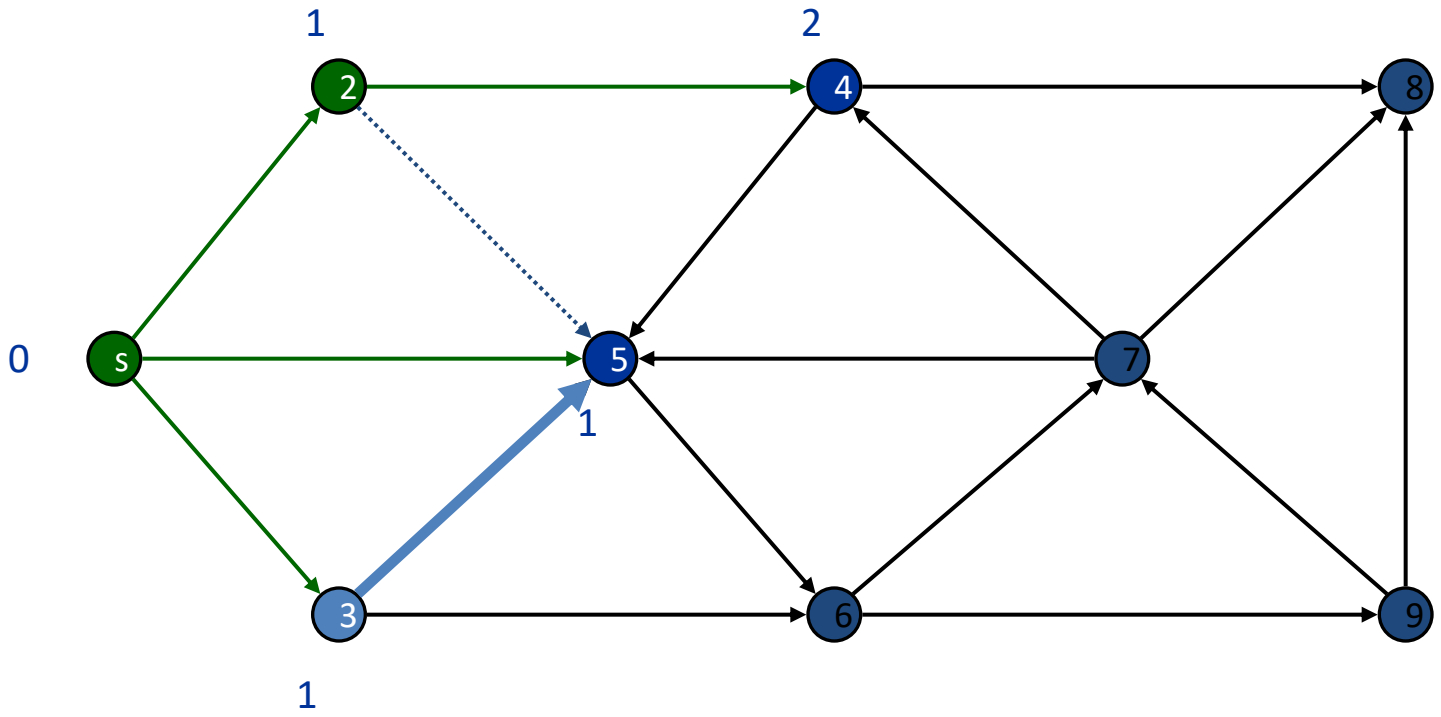
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 2 3 5 4

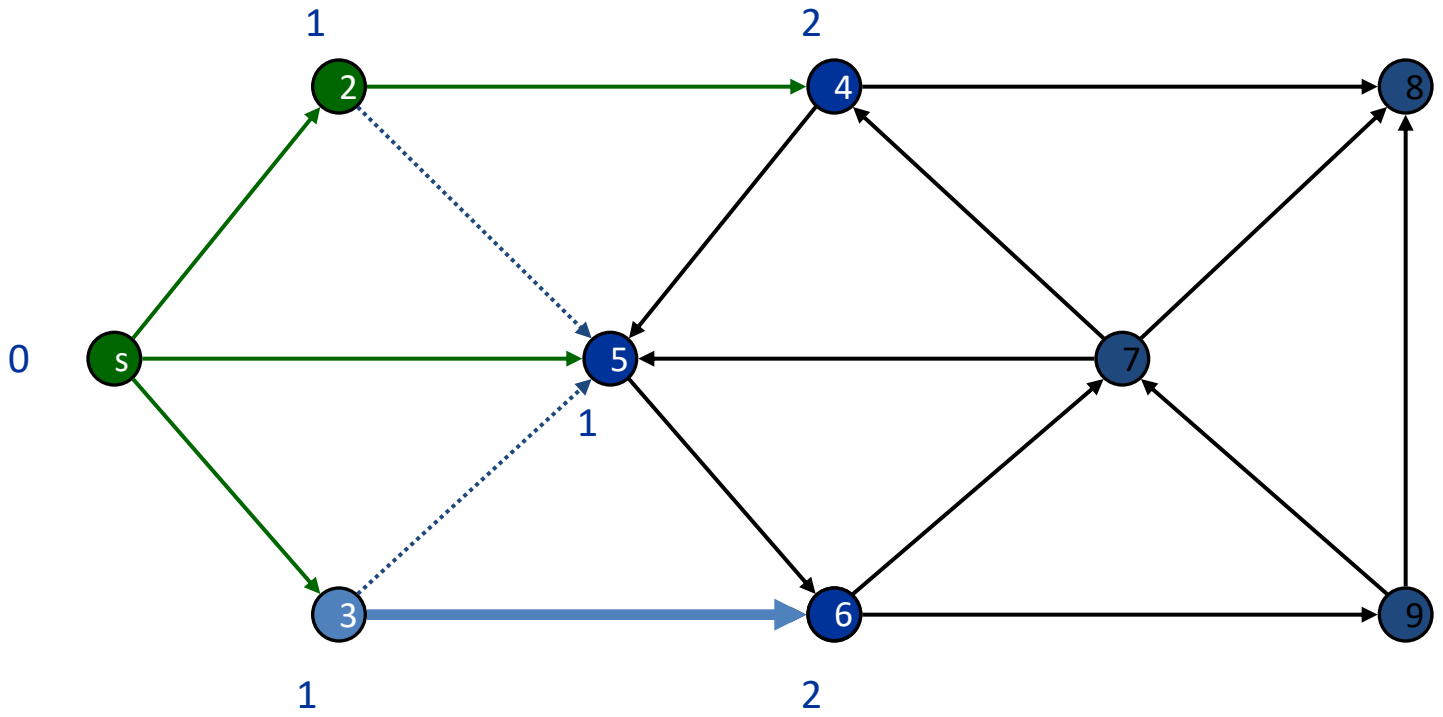
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4

BFS



Undiscovered

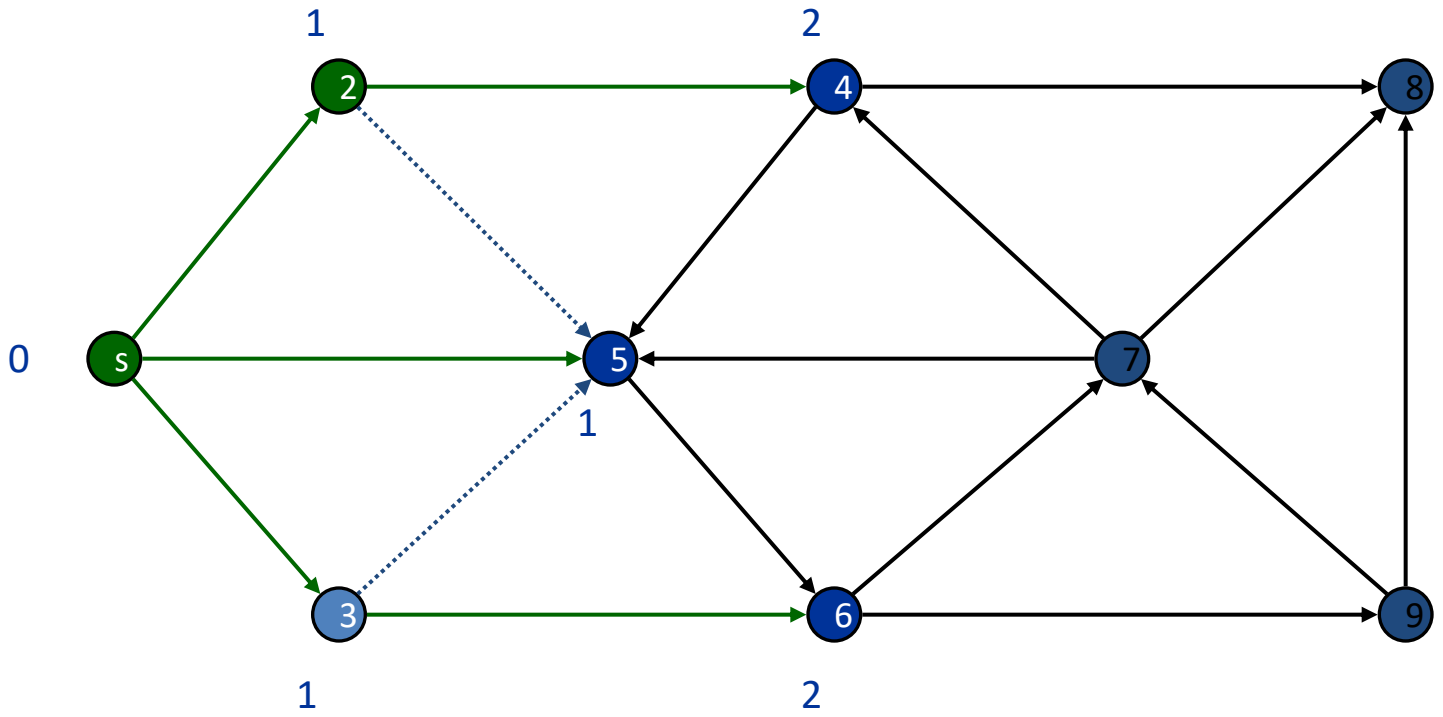
Discovered

Top of queue

Finished

Queue: 3 5 4

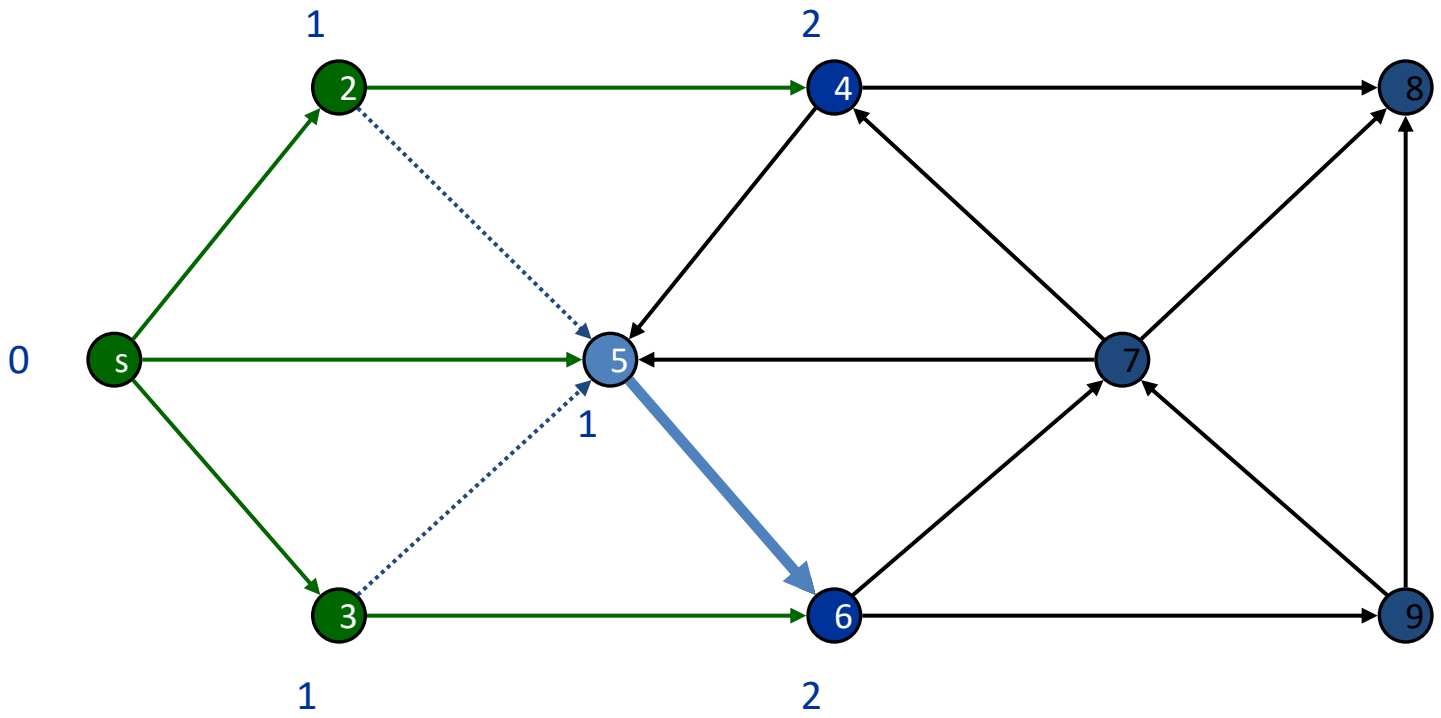
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 3 5 4 6

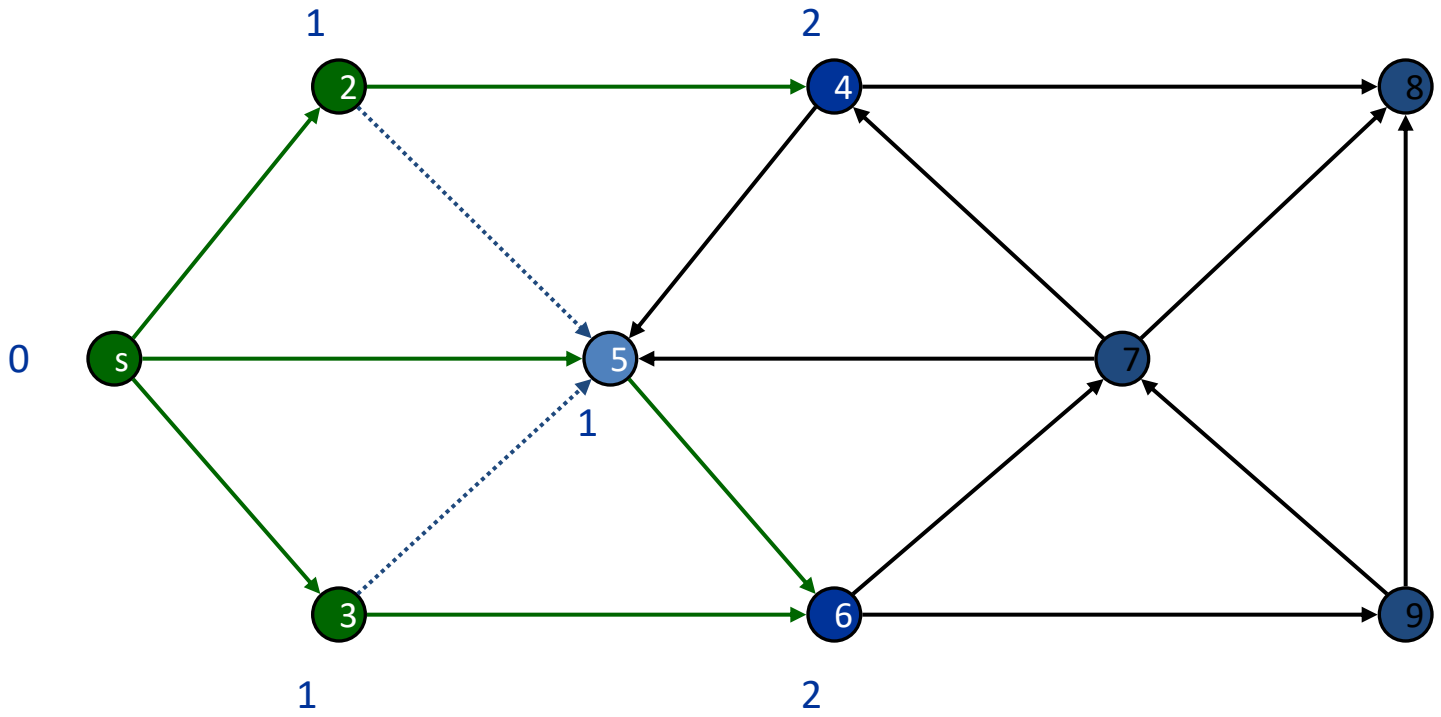
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

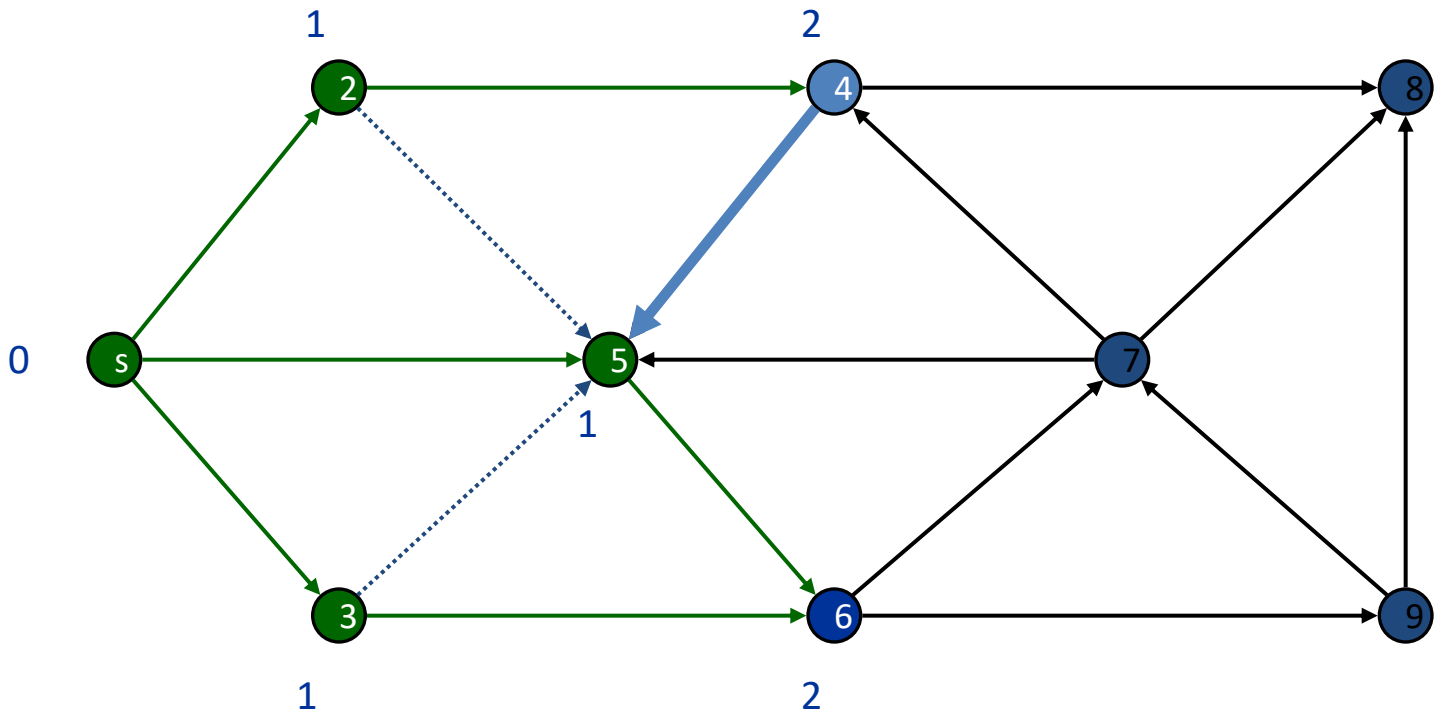
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 5 4 6

BFS



Undiscovered

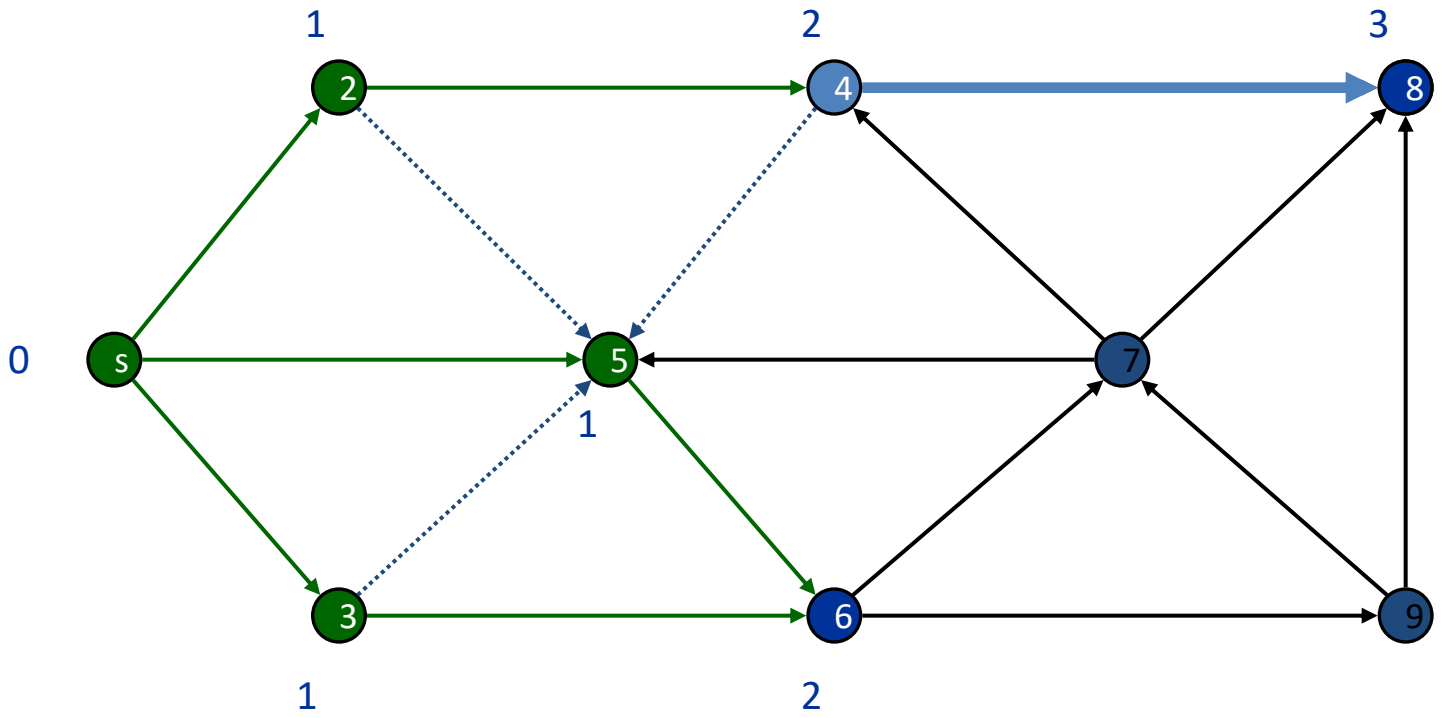
Discovered

Top of queue

Finished

Queue: 4 6

BFS



Undiscovered

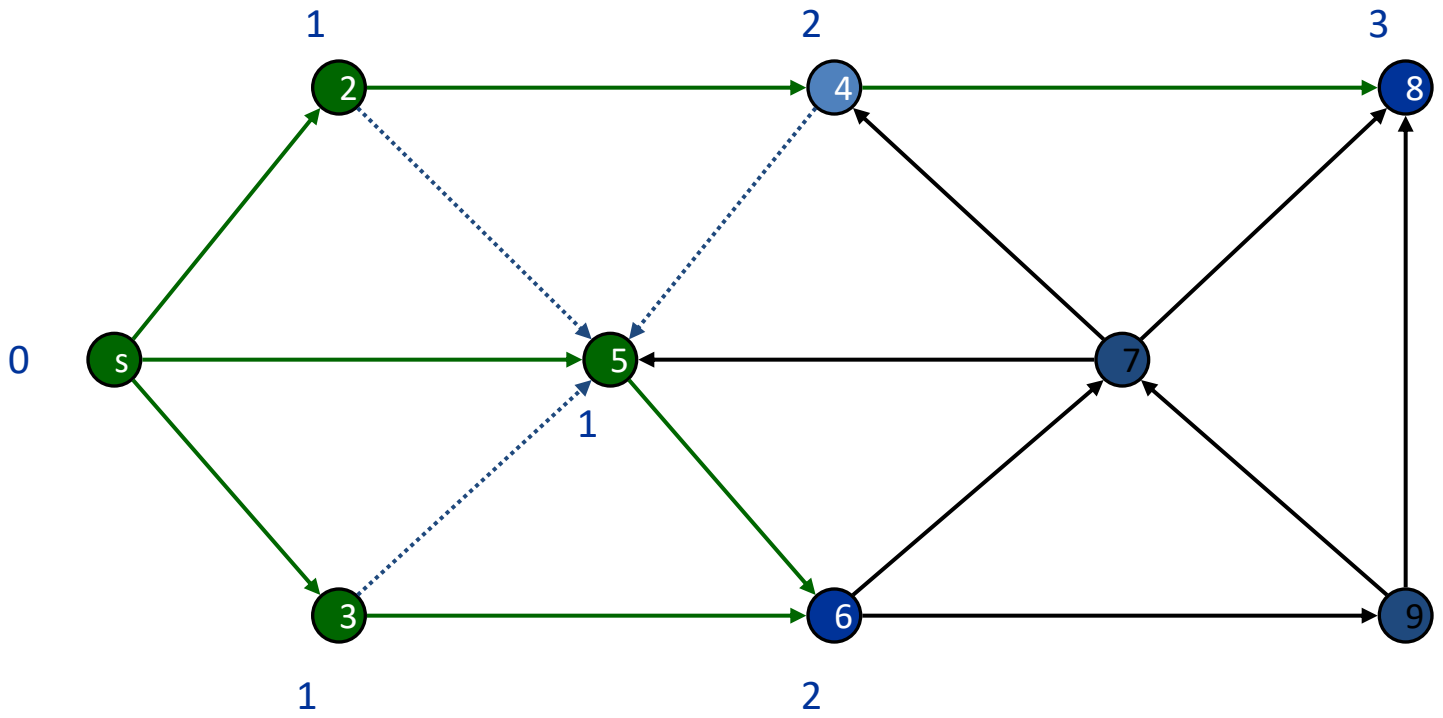
Discovered

Top of queue

Finished

Queue: 4 6

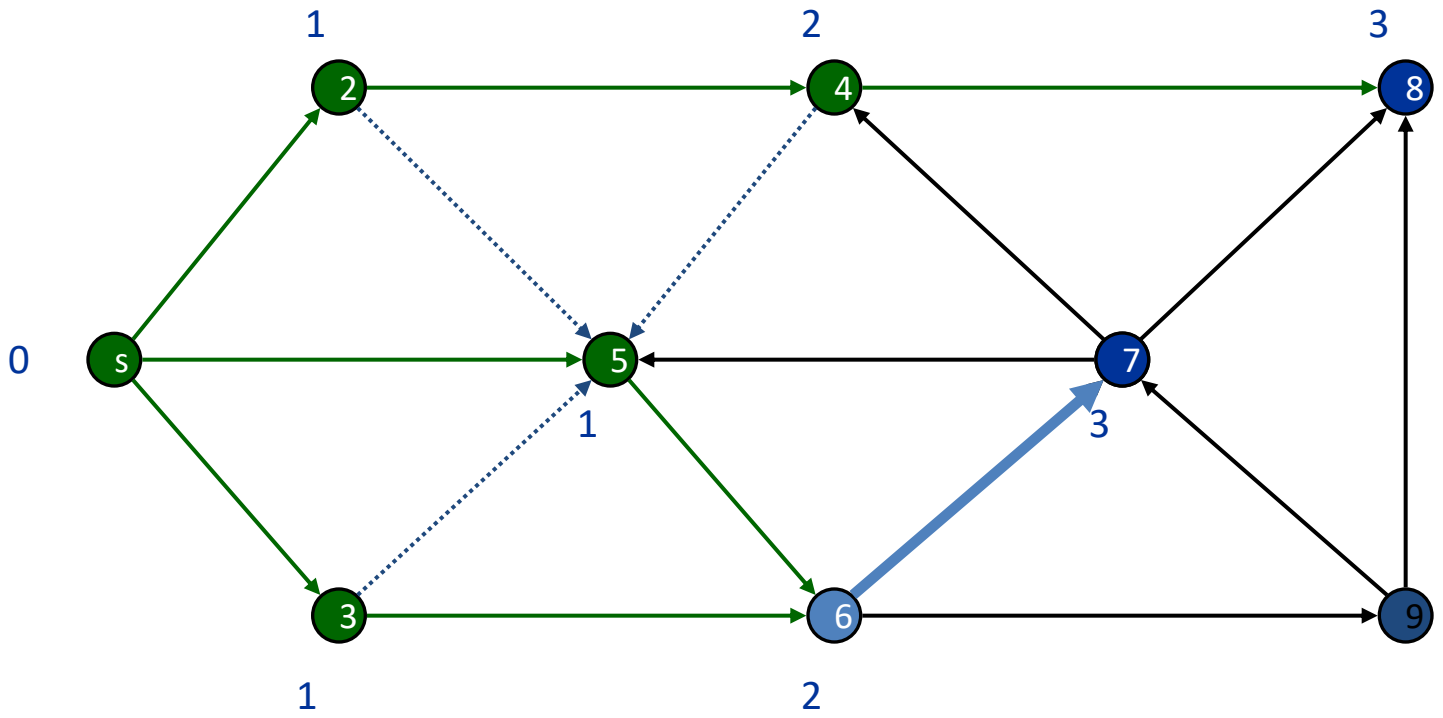
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 4 6 8

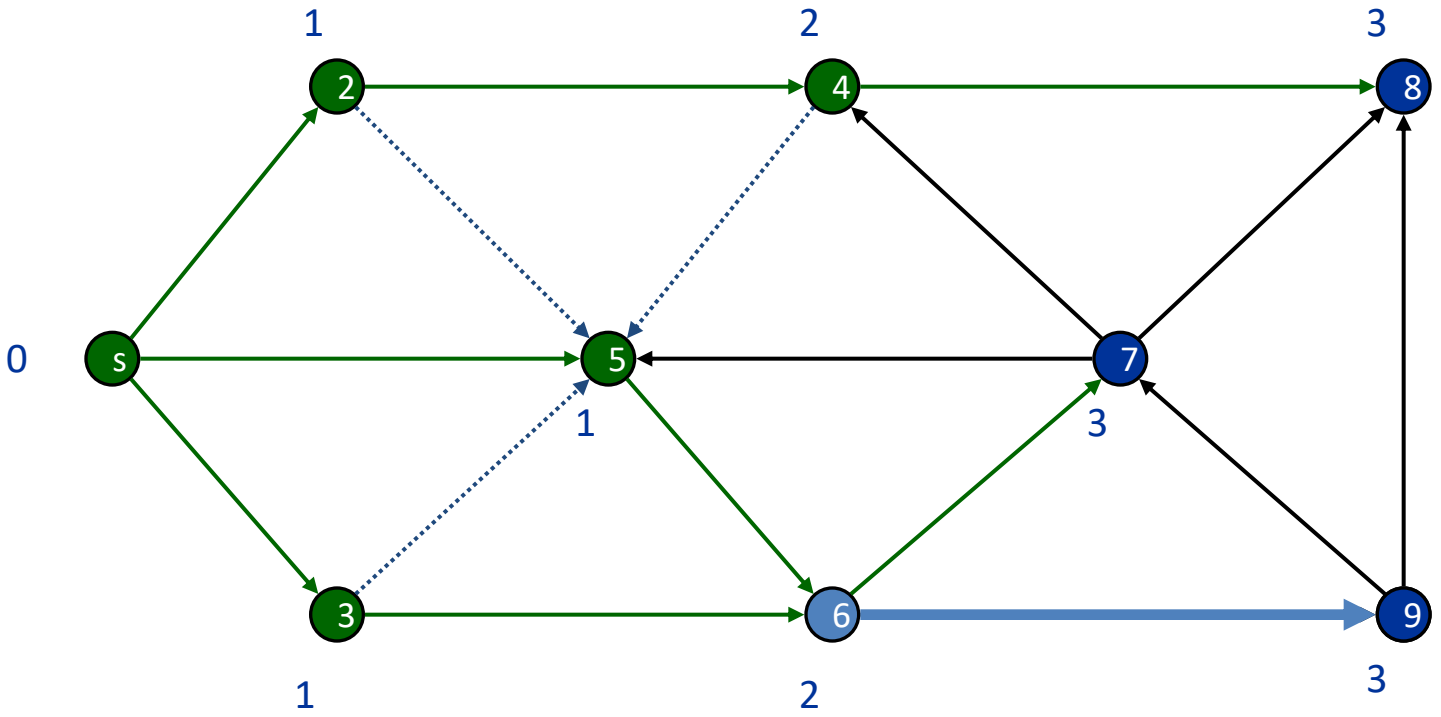
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8

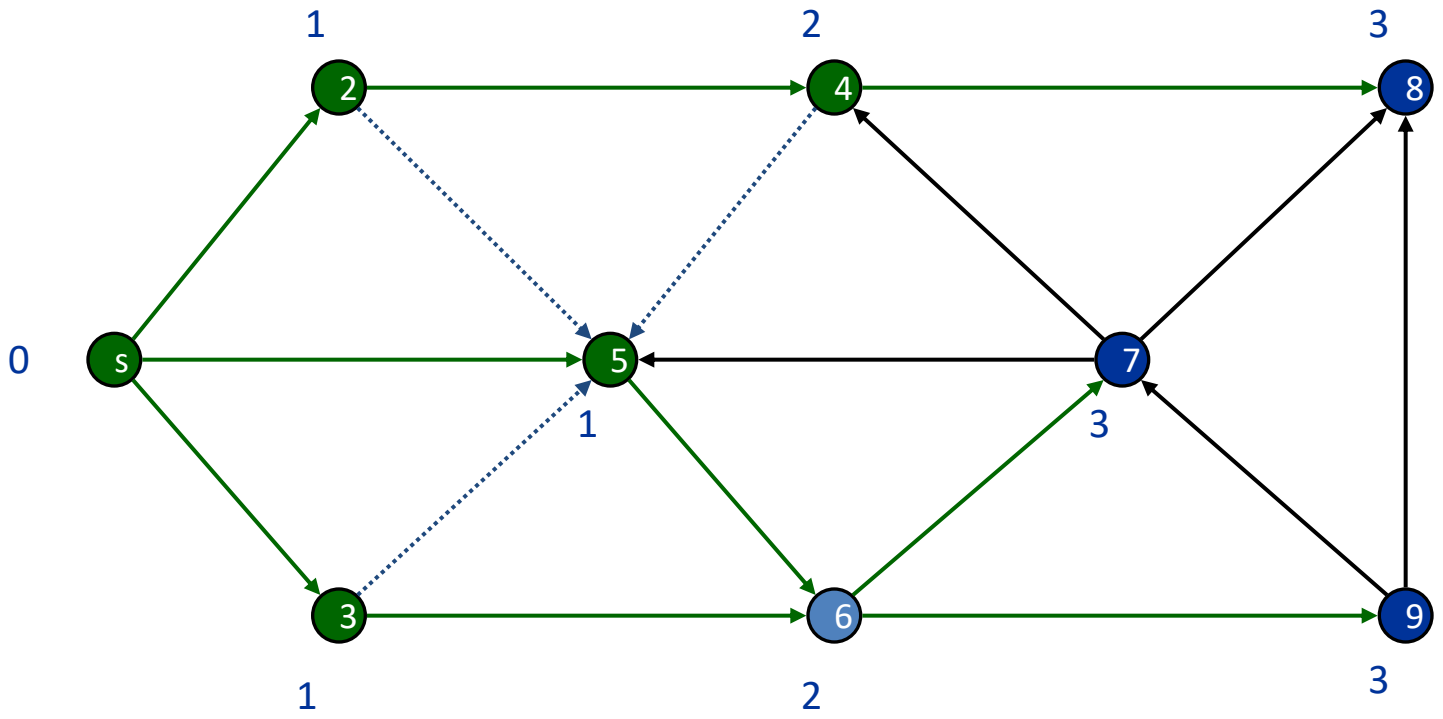
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7

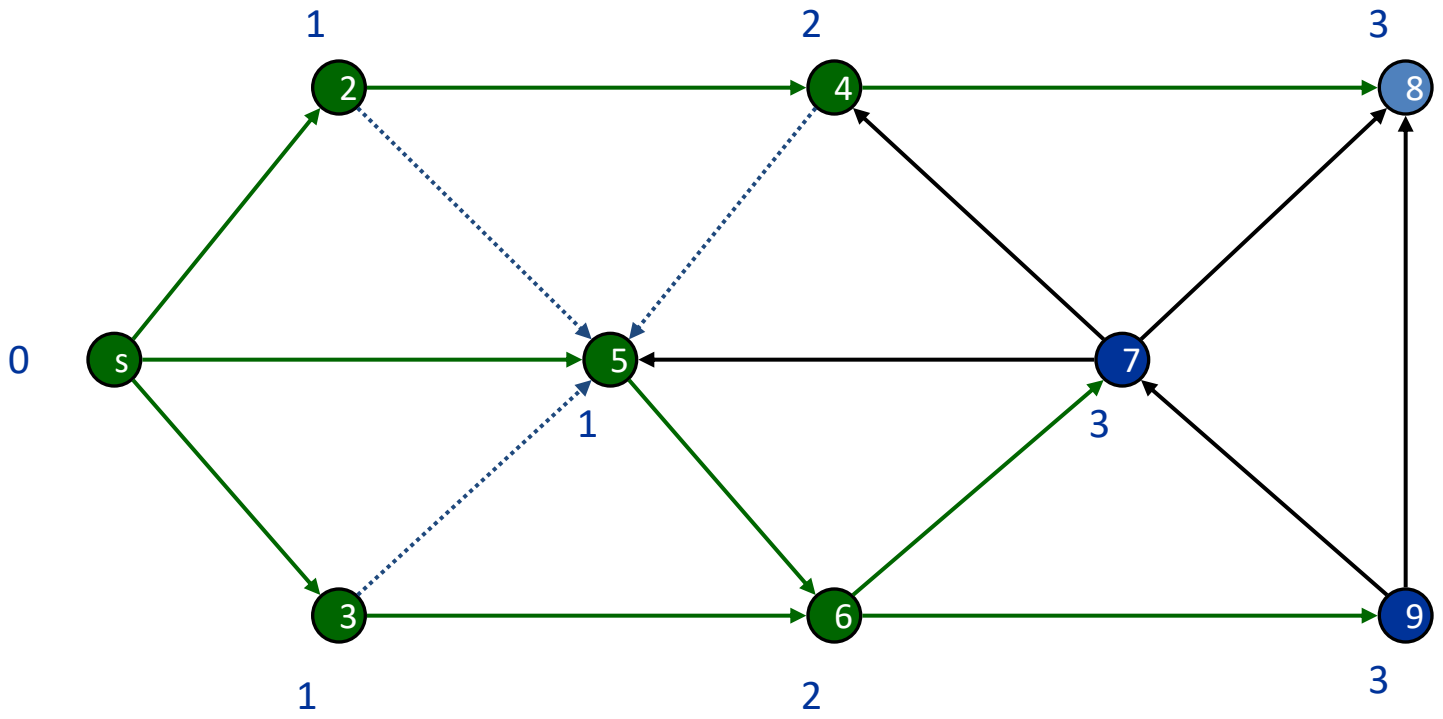
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 6 8 7 9

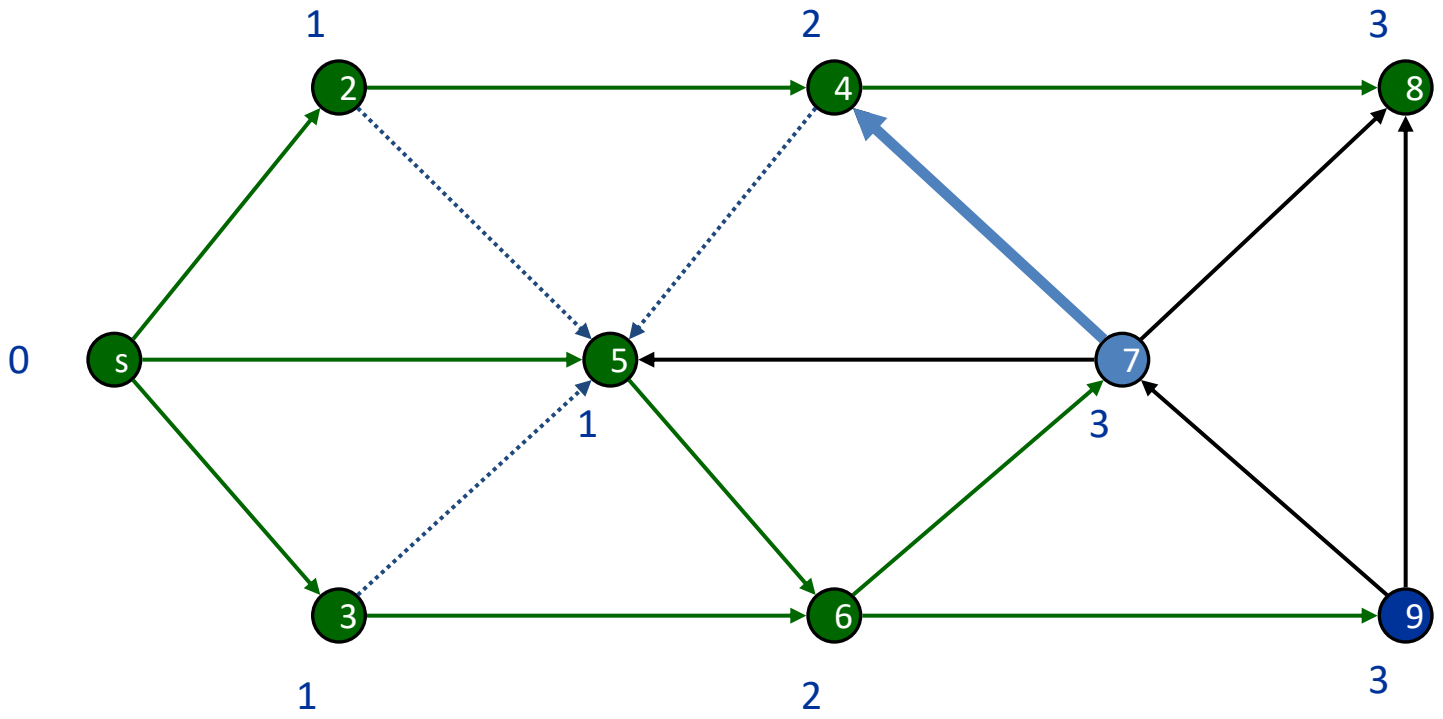
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 8 7 9

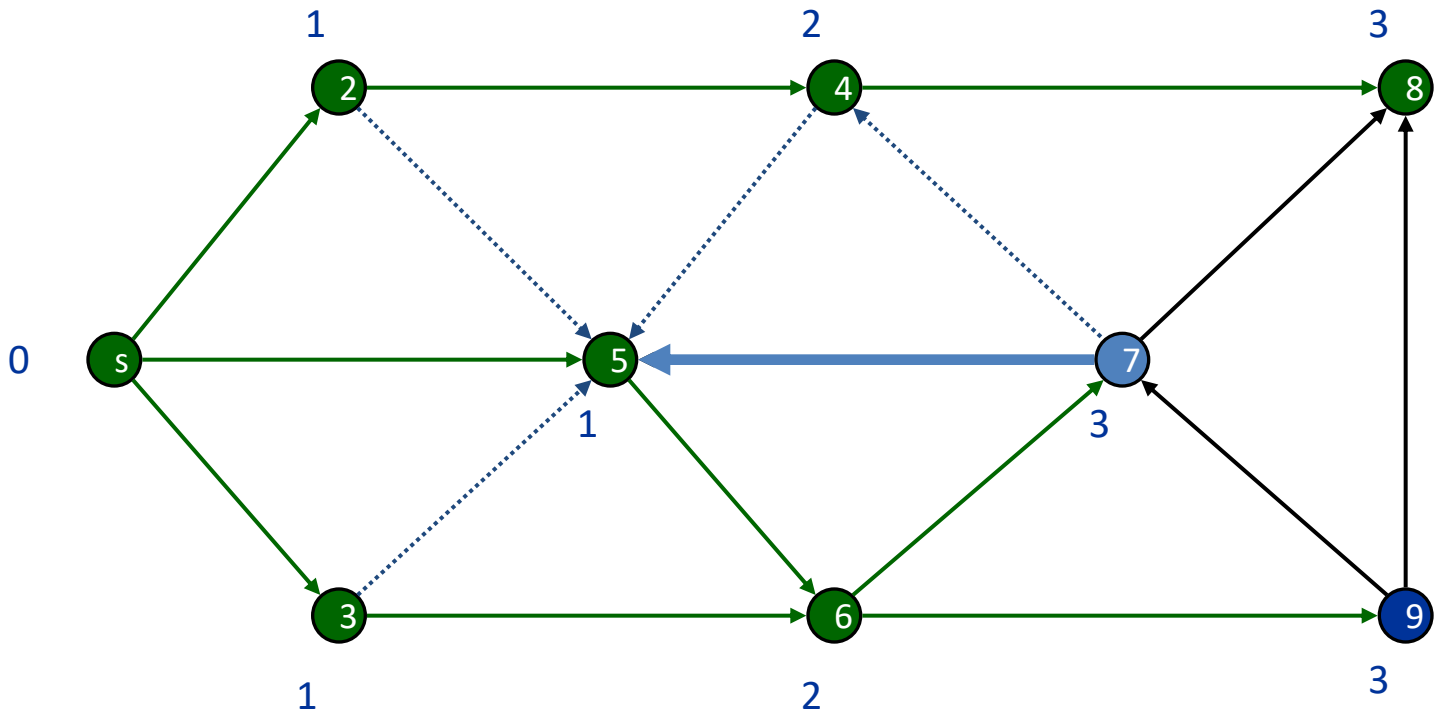
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

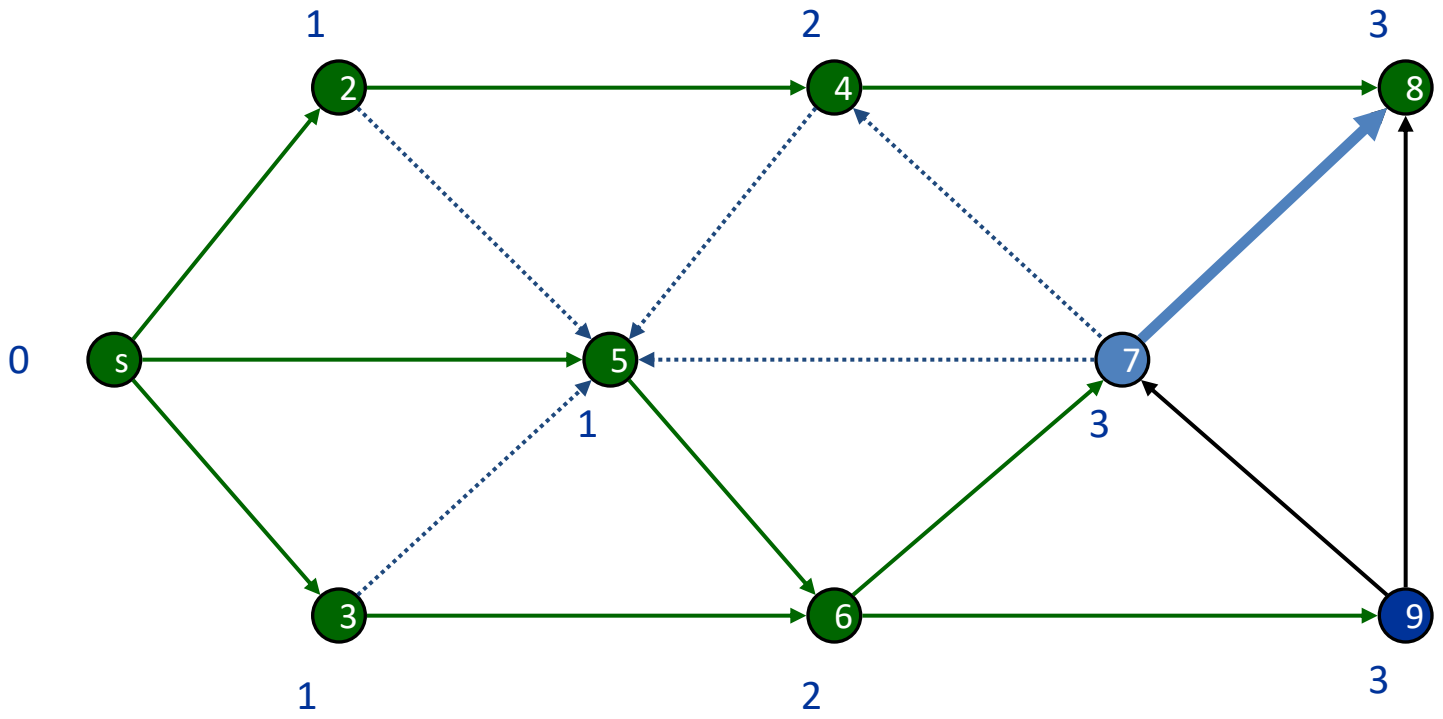
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

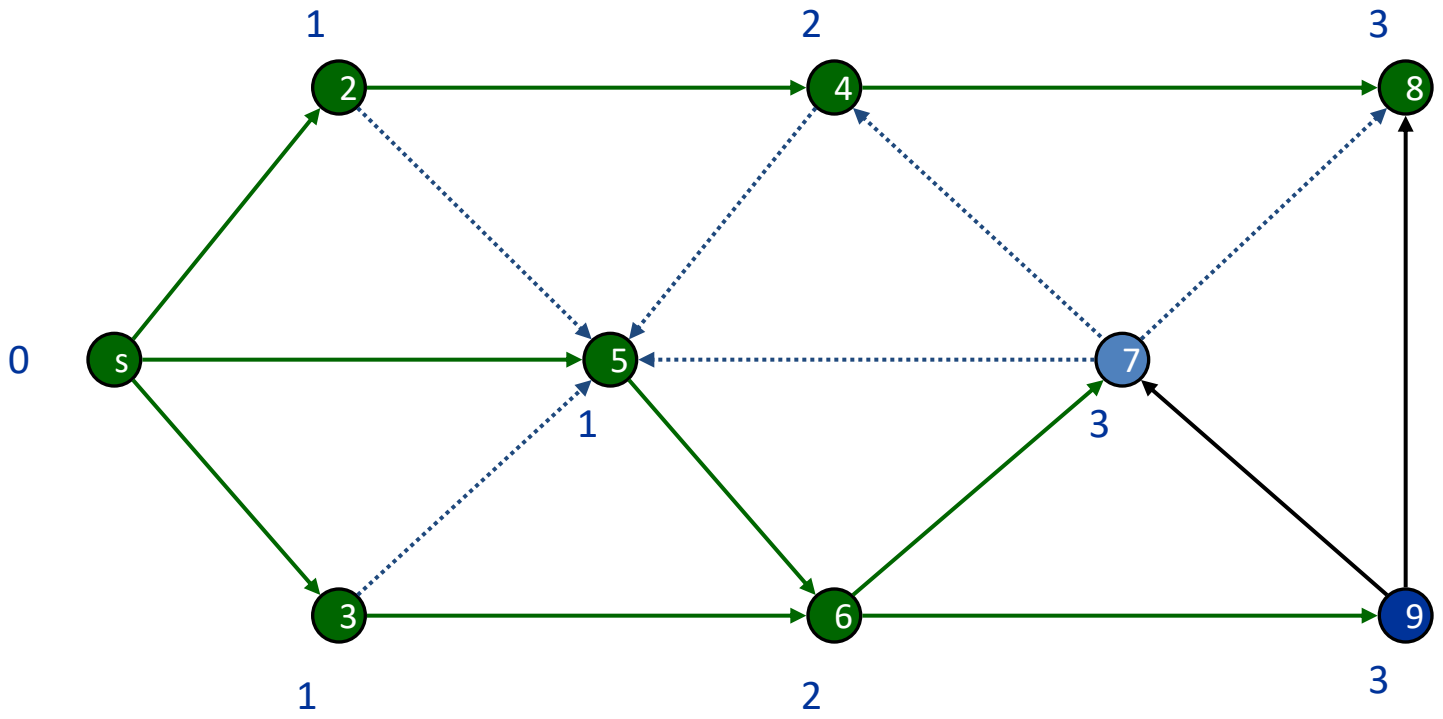
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

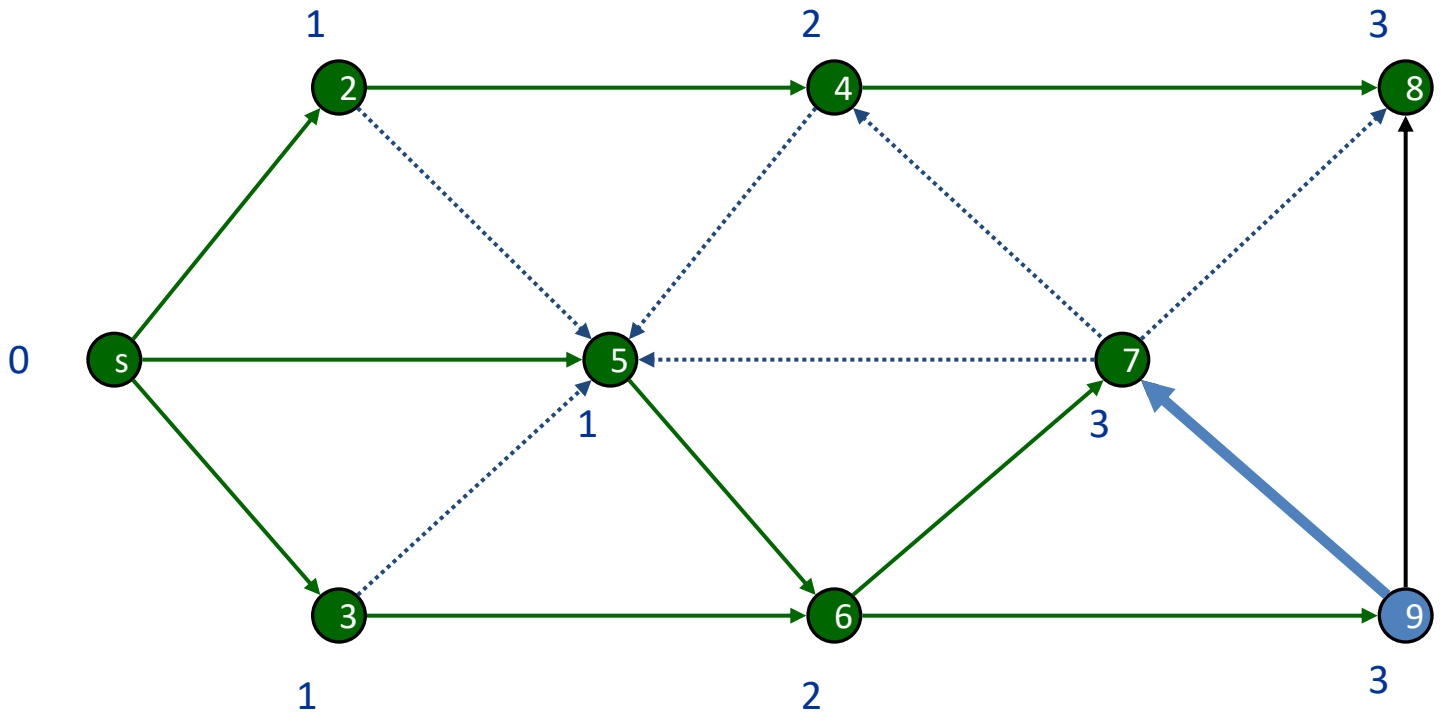
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 7 9

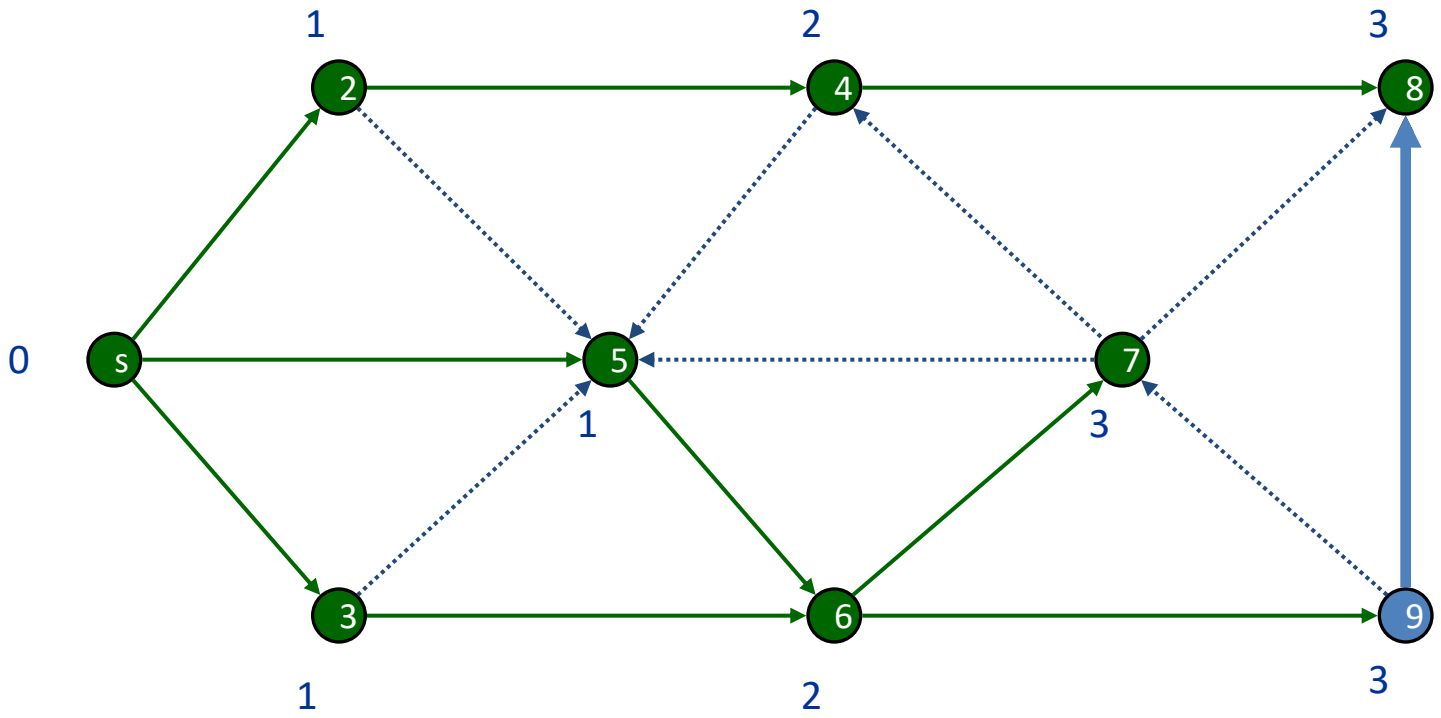
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

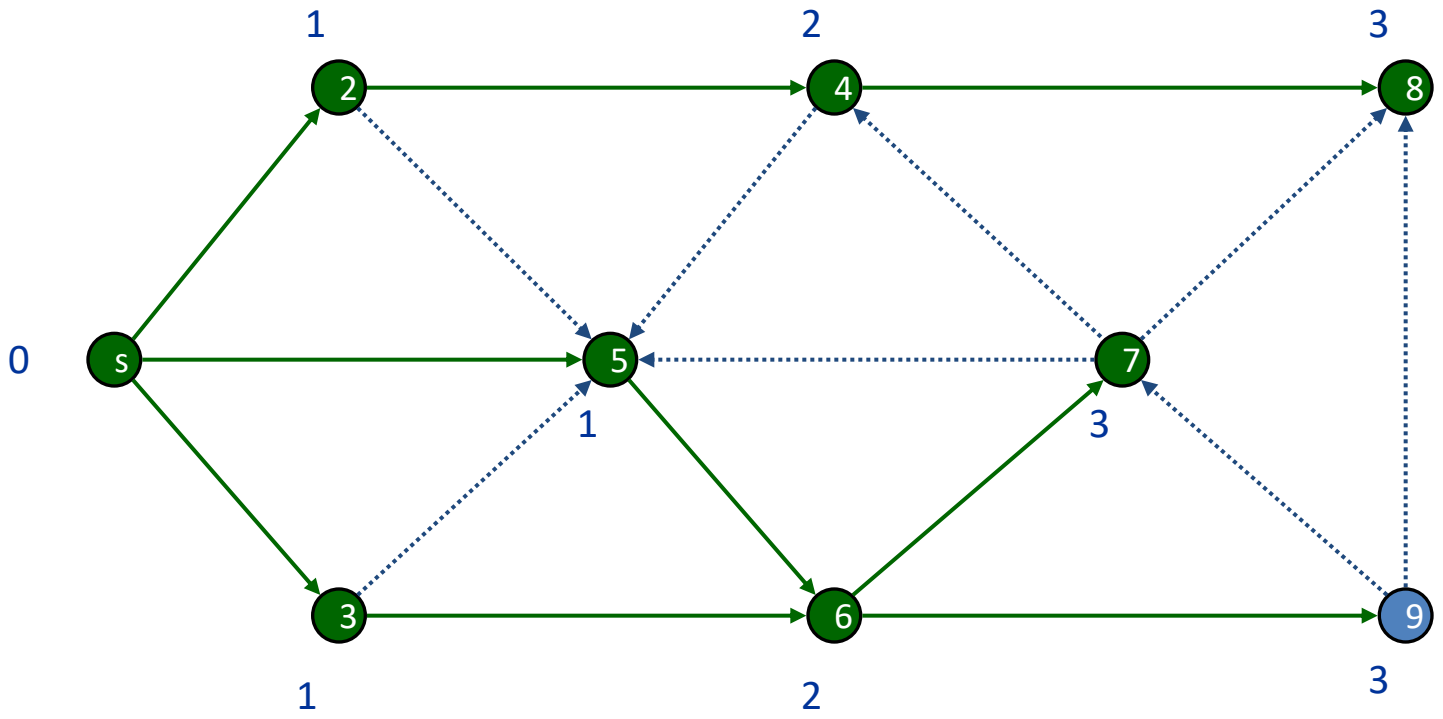
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

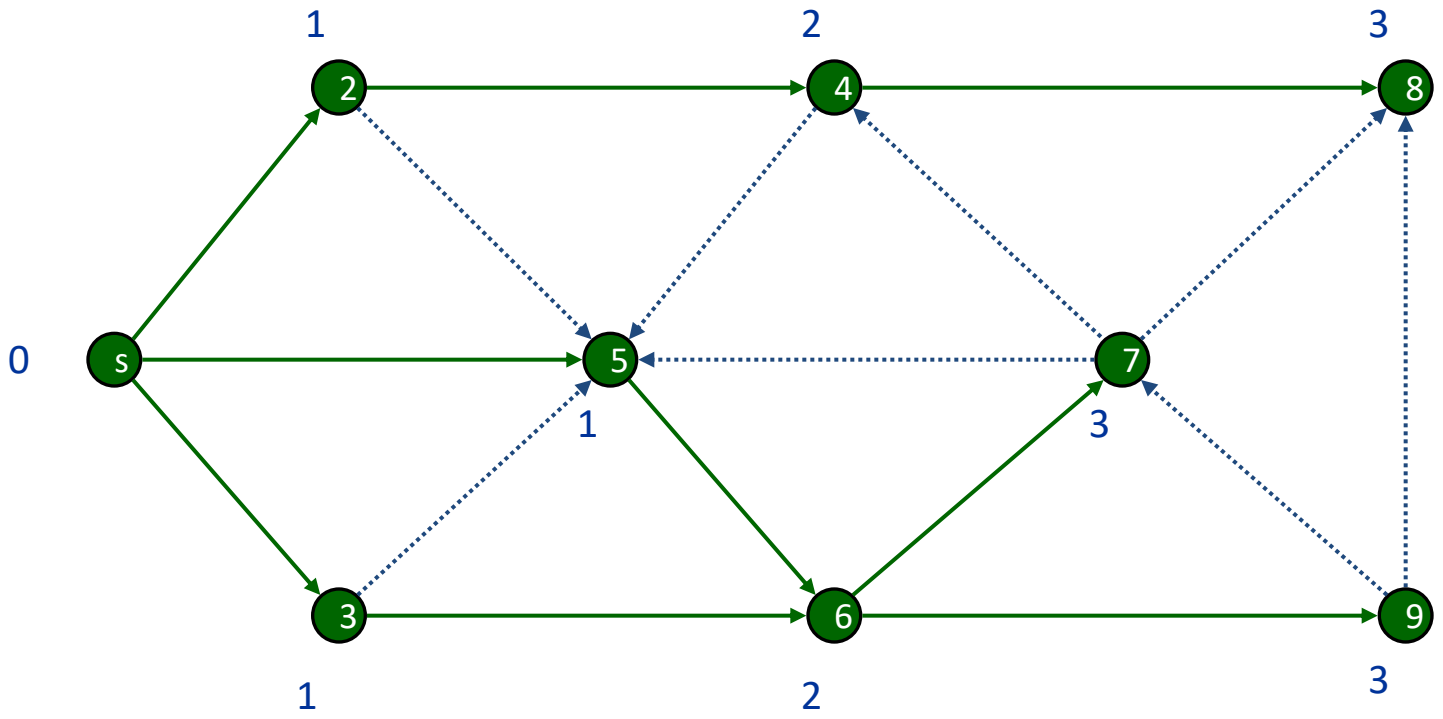
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue: 9

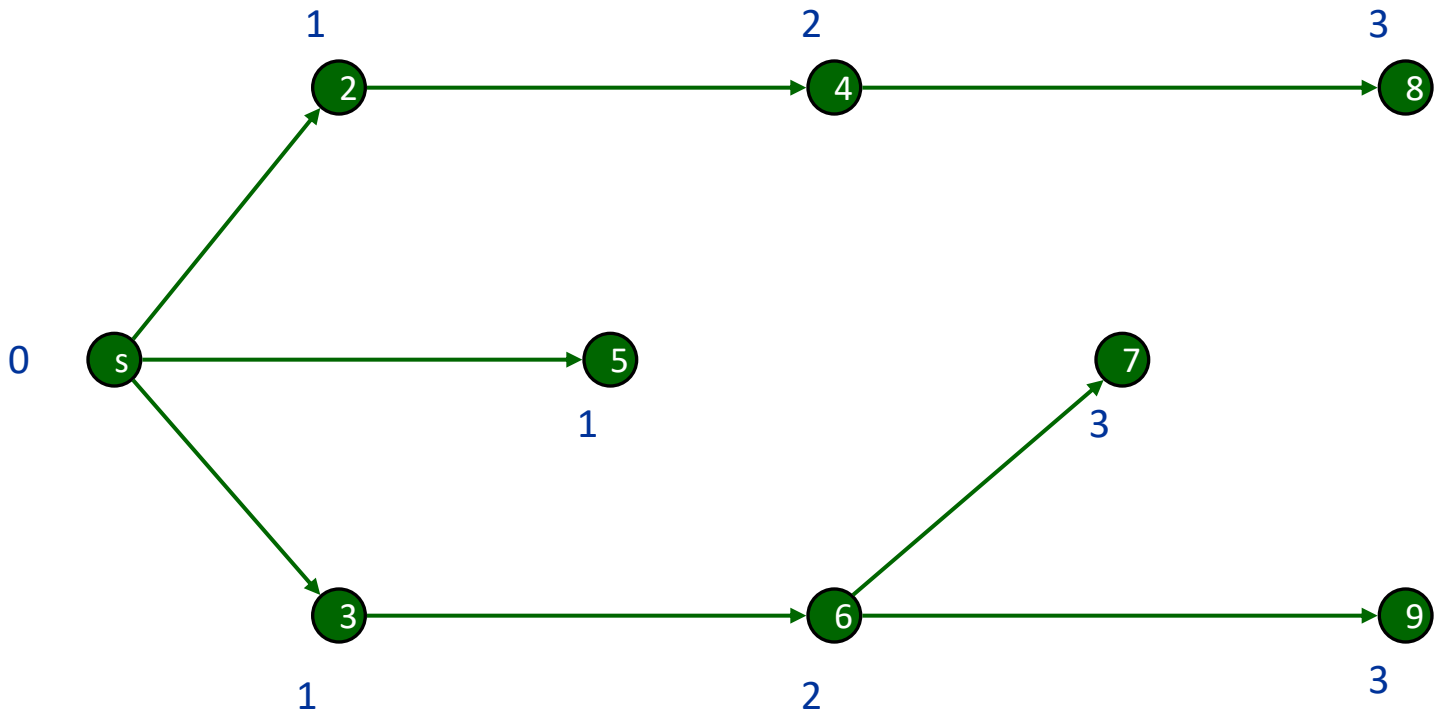
BFS



Undiscovered
Discovered
Top of queue
Finished

Queue:

BFS



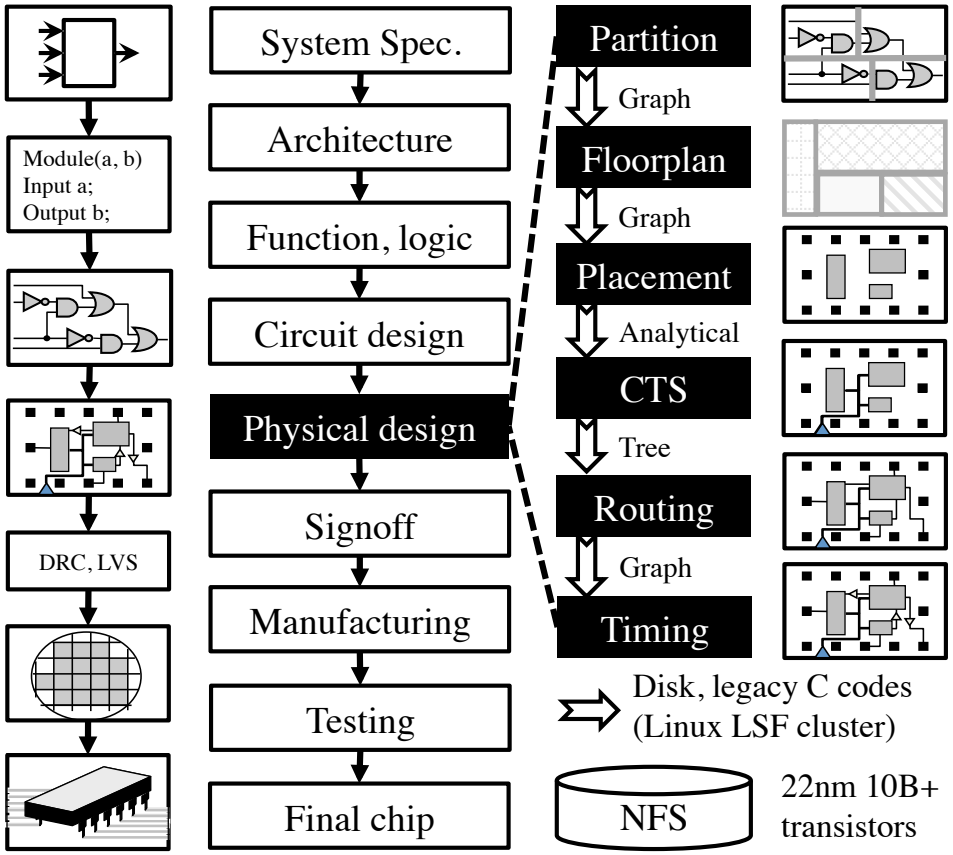
Level Graph

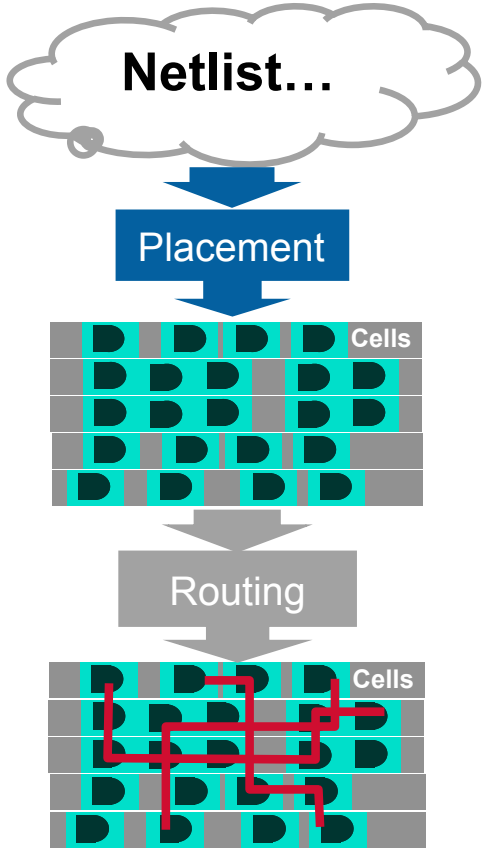
BFS Implementation

```
1   procedure BFS(G, start_v) is
2       let Q be a queue
3       label start_v as discovered
4       Q.enqueue(start_v)
5       while Q is not empty do
6           v := Q.dequeue()
7           if v is the goal then
8               return v
9           for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as discovered then
11                  label w as discovered
12                  w.parent := v
13                  Q.enqueue(w)
```

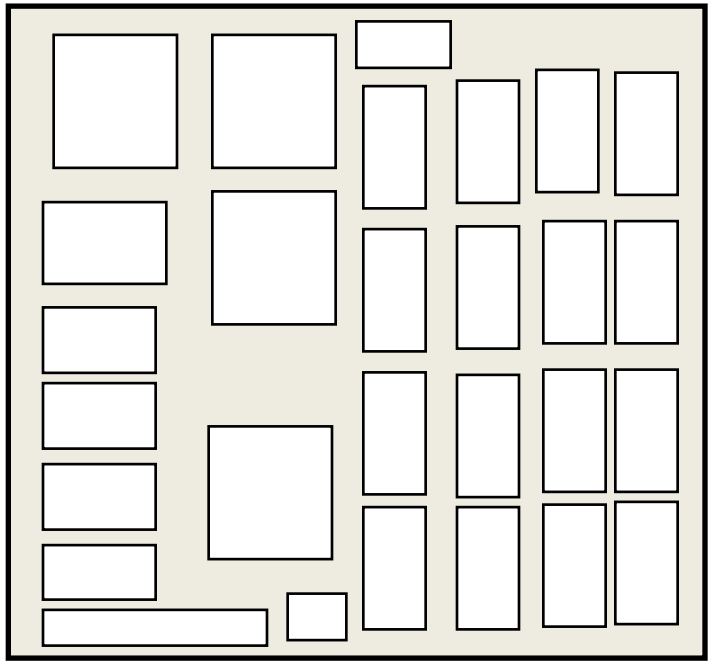
Computer Design Automation Flow

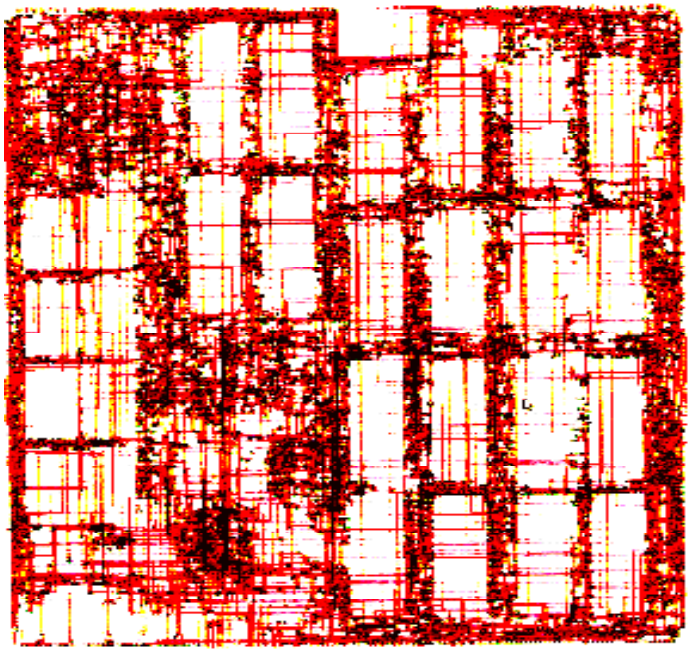
- **Physical synthesis (placement, routing, etc)**





The Problem





Thousands of macro blocks
Millions of gates
Millions of wires

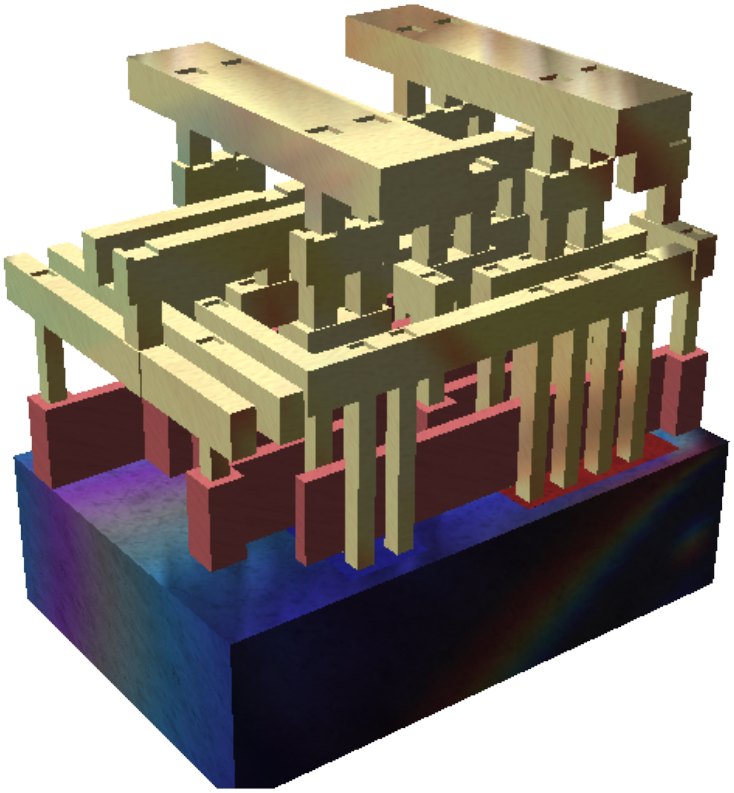
Kilometers of wire.

Basic Routing Problems

- **Scale**
  - Big chips have an enormous number (**millions**) of wires
  - Not every wire gets to take an "easy" path to connect its pins
  - **Must** connect them all--can't afford to tweak many wires manually
- **Geometric complexity**
  - It used to be that the representation of the layout was a simple "grid"
  - No longer true: at nanoscale, **geometry rules are complex** – makes routing hard
- **Electrical complexity**
  - It's not enough to make sure you connect all the wires
  - You also must ensure that the **delays** thru the wires are not too big
  - And that wire-to-wire **interactions** (crosstalk) don't mess up behavior

Physical Assumption

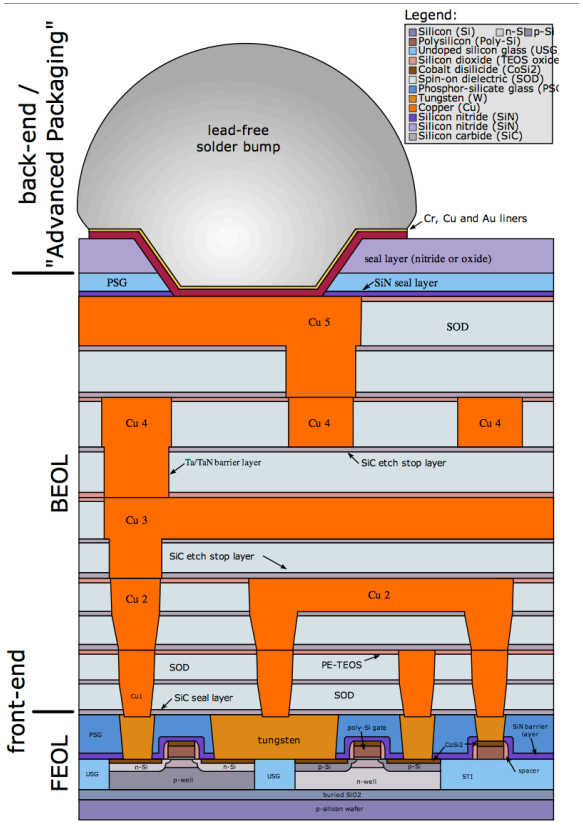
- **Many layers of wiring are available for routing**
  - Made of metal (today, copper)
  - We can connect wires in different layers with **vias**
- **A simplified view**
  - Standard cells are using metal on layers 1,2
  - Routing wires on layers 3-8
  - Upper layers (9, 10) reserved for power and clock distribution



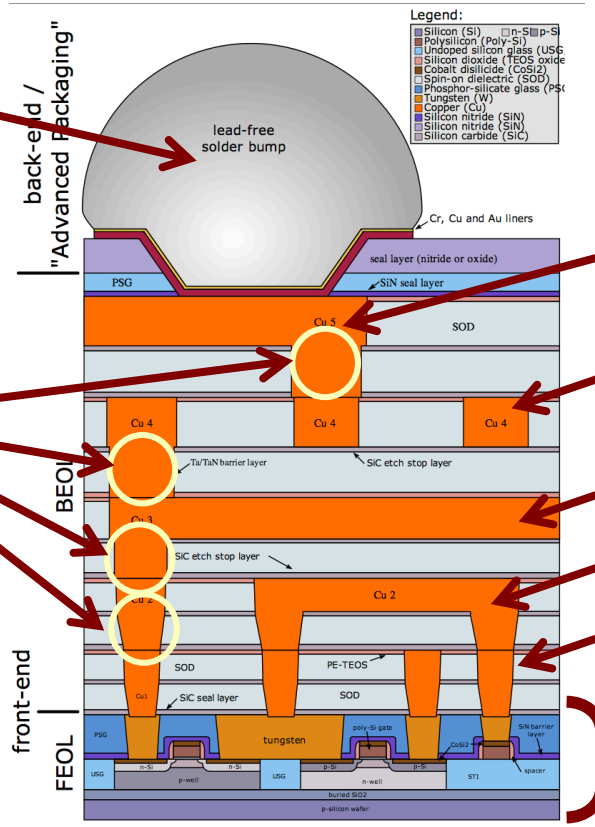
Typical Example (5 Layers, Circa 2000)

- **Some terminology**
  - **FEOL**: Front end of line. Chip fabrication steps that make transistors on Si wafer
  - **BEOL**: Back end of line. Fabrication steps that make layers of wires on top of transistors

- **This picture**
  - Cross sectional view of FEOL and BEOL structures in 5-layers of wiring…
  - … along with packaging connection ("bump") which is how chip connects to pins on package



Typical Example (5 Layers, Circa 2000)



Package interconnect, (like a via), but from chip to board

Vias, connect Metal layers k and k+1

Layer 5 metal

Layer 4 metal

Thicker layers (less resistance) for clock, power

Layer 3 metal

Layer 2 metal

Layer 1 metal

Wiring for logic

Transistors to make gate-level logic

Placement vs Routing

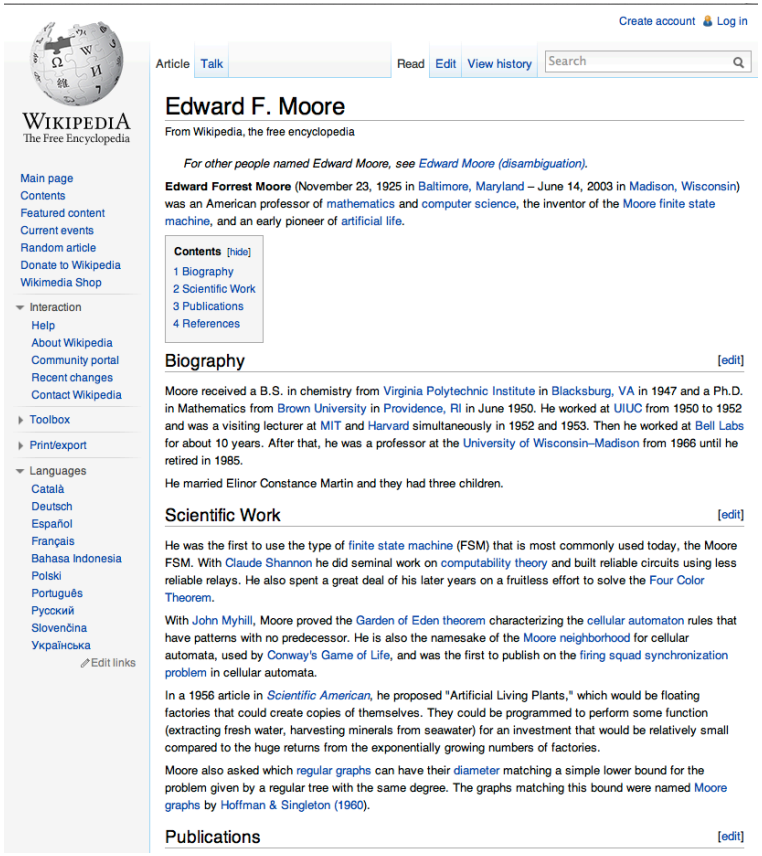
- **There are lots of different placement algorithms**
  - Iterative methods. Used mainly for high-level floorplanning, not gate layout
  - Many analytical methods based on solving/optimizing large systems of equations
    - Quadratic wirelength, exponential model, etc.

- **There are not quite so many routing algorithms**
  - There are lots of routing data structures – to represent the geometry efficiently
  - But there is **one very, very big idea** at core of most real routers…

Big Idea "Maze Routing"

- **From one famous early paper:**
  - E.F. Moore. "**The shortest path through a maze**"
  - International Symposium on the Theory of Switching Proceedings, pp 285--292, Cambridge, MA, Apr. 1959. Harvard University Press

- **Yes – it's that Moore of "Moore finite state machines" fame**
  - Given a maze (or a graph), find a shortest path from entrance to exit



Create account Log in

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop
Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact Wikipedia
Toolbox
Print/export
Languages
Català
Deutsch
Español
Français
Bahasa Indonesia
Polski
Português
Русский
Slovenčina
Українська
Edit links

Edward F. Moore

From Wikipedia, the free encyclopedia

For other people named Edward Moore, see Edward Moore (disambiguation).

Edward Forrest Moore (November 23, 1925 in Baltimore, Maryland – June 14, 2003 in Madison, Wisconsin) was an American professor of mathematics and computer science, the inventor of the Moore finite state machine, and an early pioneer of artificial life.

Contents [hide]
1 Biography
2 Scientific Work
3 Publications
4 References

Biography [edit]

Moore received a B.S. in chemistry from Virginia Polytechnic Institute in Blacksburg, VA in 1947 and a Ph.D. in Mathematics from Brown University in Providence, RI in June 1950. He worked at UIUC from 1950 to 1952 and was a visiting lecturer at MIT and Harvard simultaneously in 1952 and 1953. Then he worked at Bell Labs for about 10 years. After that, he was a professor at the University of Wisconsin–Madison from 1966 until he retired in 1985.

He married Elinor Constance Martin and they had three children.

Scientific Work [edit]

He was the first to use the type of finite state machine (FSM) that is most commonly used today, the Moore FSM. With Claude Shannon he did seminal work on computability theory and built reliable circuits using less reliable relays. He also spent a great deal of his later years on a fruitless effort to solve the Four Color Theorem.

With John Myhill, Moore proved the Garden of Eden theorem characterizing the cellular automaton rules that have patterns with no predecessor. He is also the namesake of the Moore neighborhood for cellular automata, used by Conway's Game of Life, and was the first to publish on the firing squad synchronization problem in cellular automata.

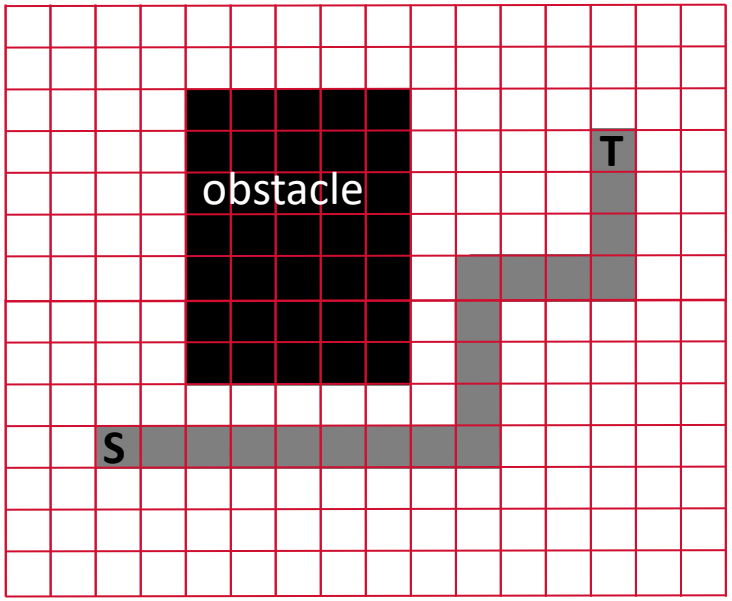
In a 1956 article in *Scientific American*, he proposed "Artificial Living Plants," which would be floating factories that could create copies of themselves. They could be programmed to perform some function (extracting fresh water, harvesting minerals from seawater) for an investment that would be relatively small compared to the huge returns from the exponentially growing numbers of factories.

Moore also asked which regular graphs can have their diameter matching a simple lower bound for the problem given by a regular tree with the same degree. The graphs matching this bound were named Moore graphs by Hoffman & Singleton (1960).

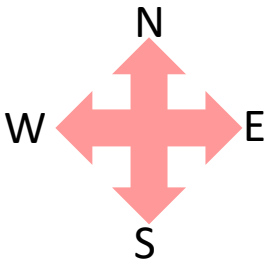
Publications [edit]

How to Get from "Mazes" to "Wires"

- **Make a big geometric assumption: Gridded routing**
  - The layout surface is a grid of regular squares
  - A legal wire path = a **set of connected grid cells**, through unobstructed cells in grid

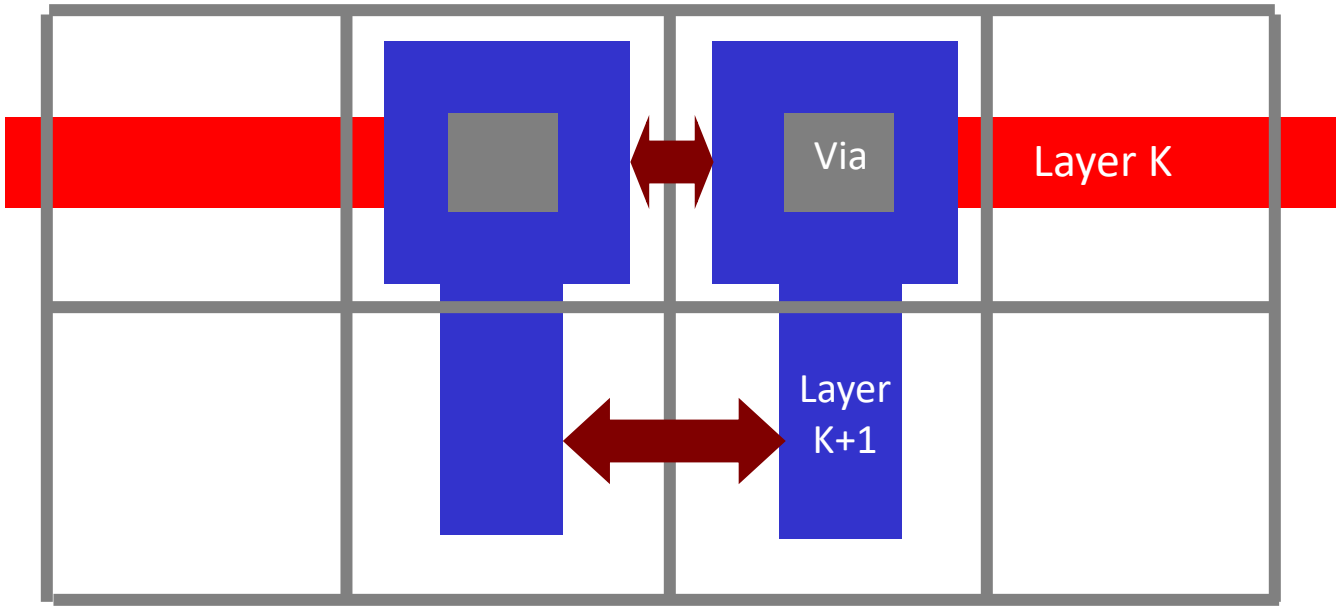


For us: wires are also **strictly horizontal and vertical.** No diagonal (eg, 45°) angles. A path goes east/west, or north/south, in this grid.



Grid Assumption

- **A critical assumption – implies many constraints on wires**
  - All wires are the same size (width)
  - All pins we want to connect are also "on grid" ie, center of grid cell
  - Wires and their vias fit in the grid, without any geometry rule (eg, spacing) violations



Maze Routers History

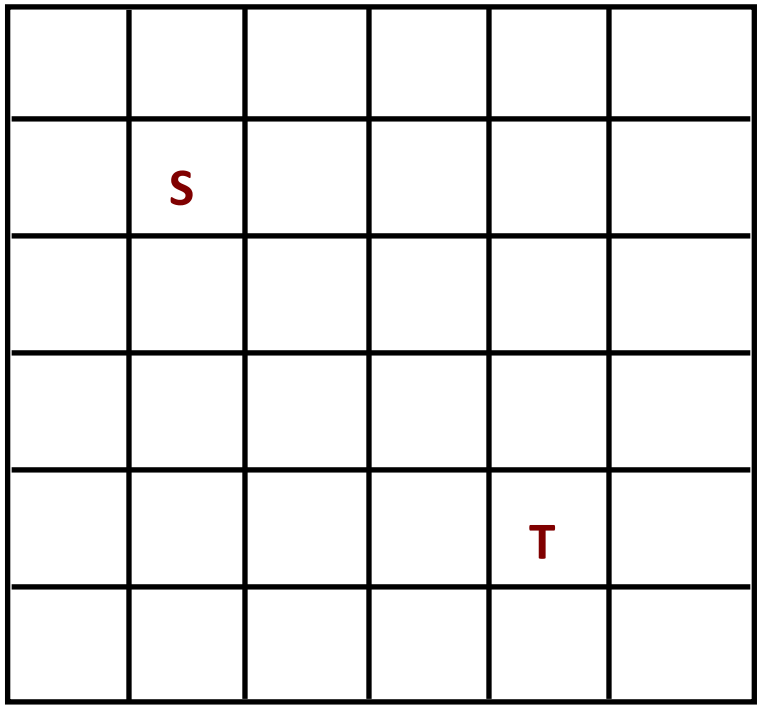
- **1959 – Basic Idea**
  - E.F. Moore. The shortest path through a maze. In International Symposium on the Theory of Switching Proceedings, Apr. 1959. Harvard University Press
- **1961 – Applied to electronics (board wiring)**
  - Lee, C. Y., "An algorithm for path connections and its applications", IRE Trans. on Electronic Computers, pp. 346-365, Sept. 1961
  - Chester Lee of Bell Labs invents the algorithm; gets famous for "Lee routers"
- **1974**
  - Rubin, F., "The Lee path connection algorithm", IEEE Trans. on Computers, vol. c-23, no. 9, pp. 907-914, Sept. 1974.
  - Frank Rubin applies some ideas from recent AI results to make it go much faster (some interesting ideas in the AI can be applied to this and vice versa)
- **1983**
  - Hightower, D., "The Lee router revisited", ICCAD, pp. 136-139, 1983.
  - Dave Hightower uses fact that "modern" computers have big virtual memory, builds great router
- **ACM ISPD 2007-2008 Global Routing Contests**
- **ACM ISPD 2018-2019 Initial Detailed Routing Contest**
- **… a lot of works still going on**

Maze Routing Strategy

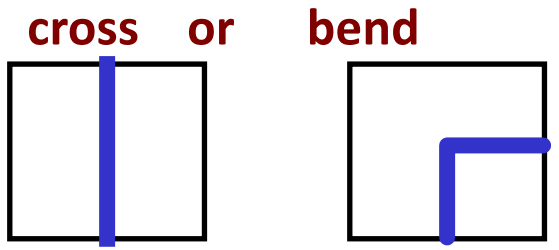
- **Strategy**
  - **One net at a time**: completely wire **one net**, then move onto next net
  - **Optimize net path:** find the **best** wiring path

- **Problems**
  - Early nets wired may block path of later nets
  - Optimal choice for one net may block later nets
  - We are just going to ignore this one for the moment…

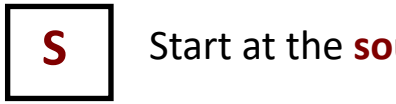
Maze Router: Basic Idea for 2-Pin Nets



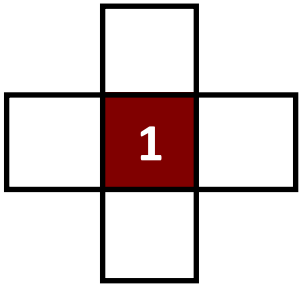
- **Given:**
  - **Grid** - each square cell represents where **one** wire can cross
  - A ***source*** and ***target***
- **Problem:**
  - Find shortest path connecting **source** cell (S) and **target** cell (T)
  - When using cells, a wire can:



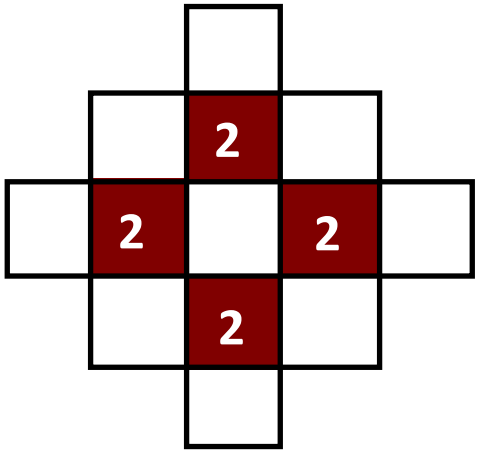
Maze Routing: Expansion



Start at the **source**



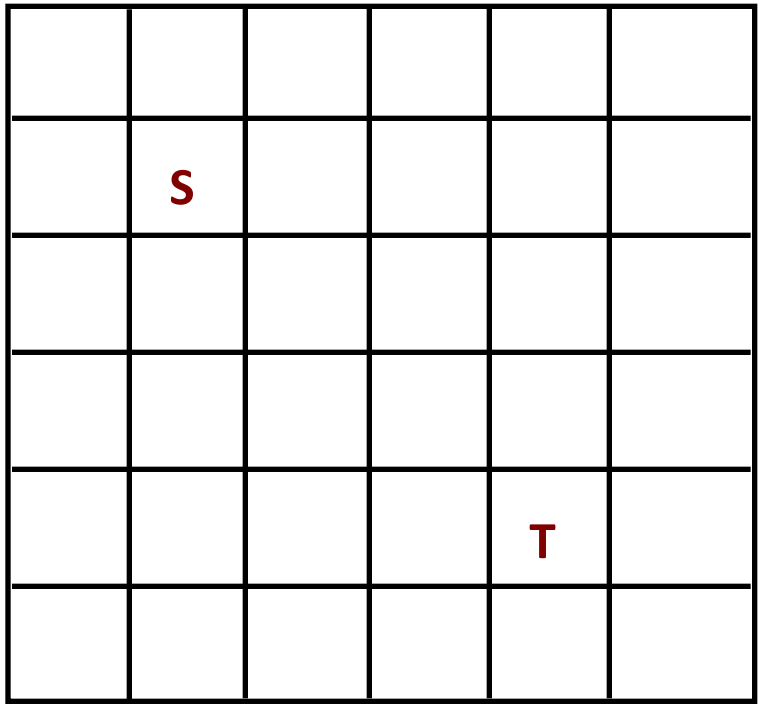
Find all new cells that are reachable at **pathlength 1**, ie, all paths that are just 1 unit in total length (just 1 cell) - mark all with this the pathlength



Using the **pathlength 1** cells, find all new cells which are reachable at **pathlength 2**

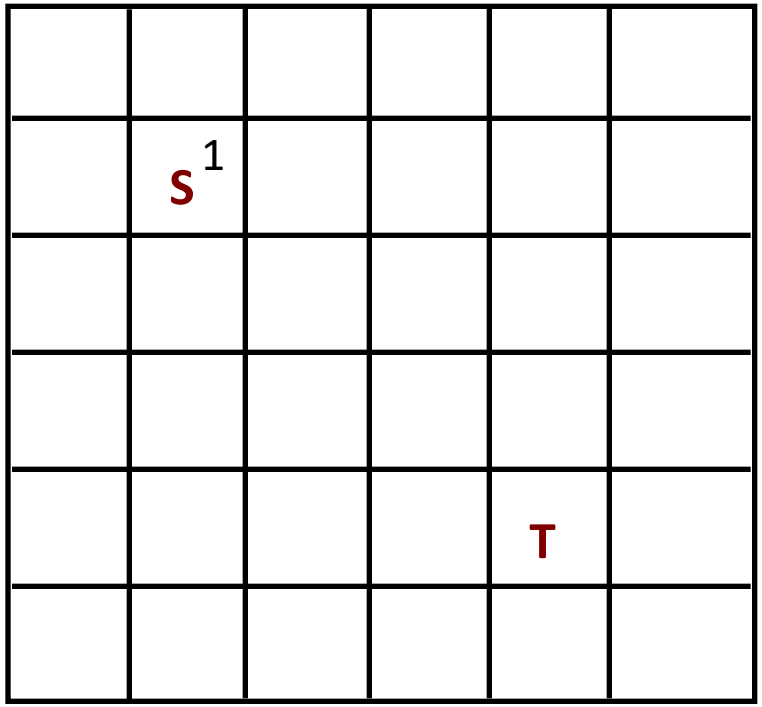
Repeat until the target is found.

Maze Router Step 1: Expand



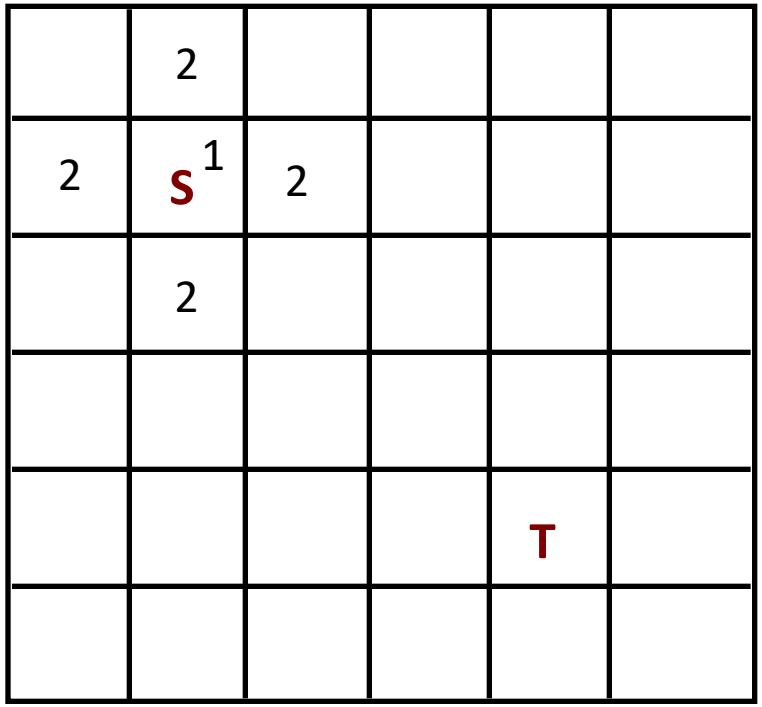
- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand



- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand



- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand

3	2	3			
2	S 1	2	3		
3	2	3			
	3				
				T	

- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand

3	2	3	4		
2	S 1	2	3	4	
3	2	3	4		
4	3	4			
	4			T	

- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand

3	2	3	4	5	
2	S 1	2	3	4	5
3	2	3	4	5	
4	3	4	5		
5	4	5		T	
	5				

- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand

3	2	3	4	5	6
2	S 1	2	3	4	5
3	2	3	4	5	6
4	3	4	5	6	
5	4	5	6	T	
6	5	6			

- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 1: Expand

3	2	3	4	5	6
2	S 1	2	3	4	5
3	2	3	4	5	6
4	3	4	5	6	7
5	4	5	6	T 7	
6	5	6	7		

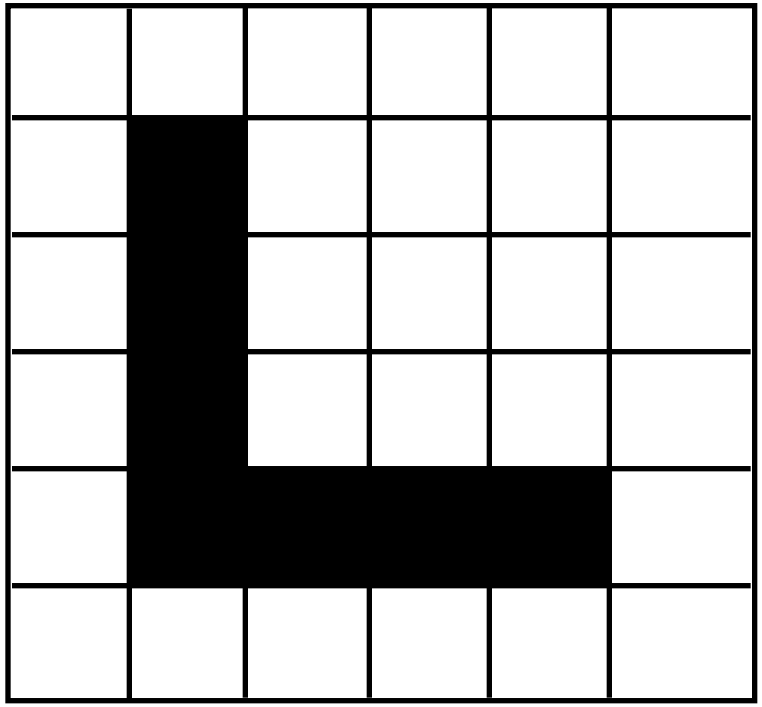
- **Strategy**
  - Expand **one cell at a time** until all the shortest paths from **S** to **T** are found.
  - Expansion creates a **wavefront** of paths that search broadly out from source cell until target is hit

Maze Router Step 2: Backtrace

3	2	3	4	5	6
2	S 1	2	3	4	5
3	2	3	4	5	6
4	3	4	5	6	7
5	4	5	6	T 7	
6	5	6	7		

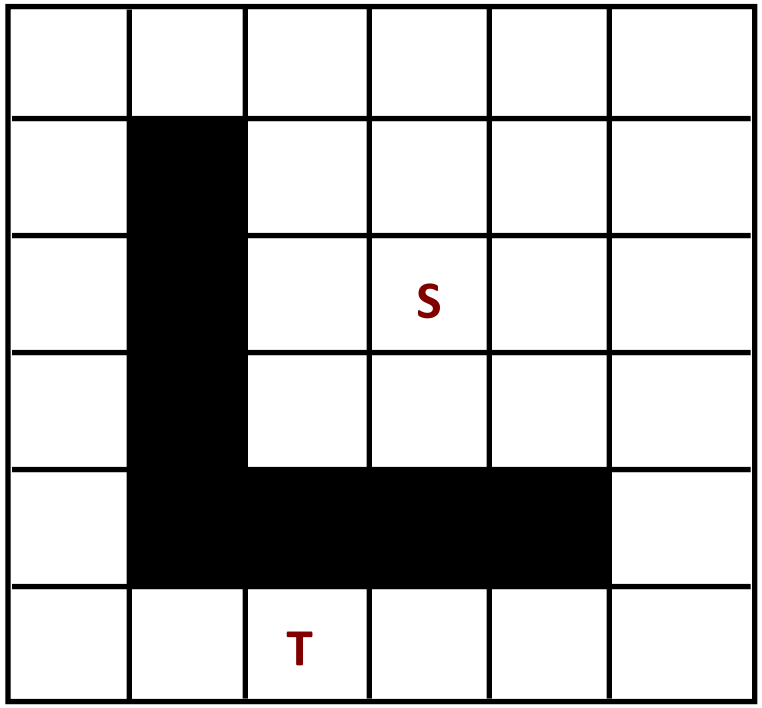
- **Now what? Backtrace**
  - Select a shortest-path (**any** shortest-path) from target back to source
  - Mark its cells so they cannot be used again – mark them as **obstacles** for later wires we want to route
  - Since there are many paths back, optimization information can be used to select the best one
  - Here, just follow the pathlengths in the cells **in descending order**

Maze Router Step 3: Clean-up



- **Now what? Clean-up**
  - Clean up the grid for the next net, leaving the **S** to **T** path as an **obstacle**
  - Now, ready to route the next net with the obstacles from the previously routed net in place in the grid

Maze Router: Obstacles



- **Also called "Blockages"**
  - Any cell you cannot use for a wire is a an **obstacle** or a **blockage**
  - There may be parts of the routing surface you just cannot use
  - But most importantly, you **label each newly routed net as a blockage**
  - Thus, all future nets must **route around** this blockage

Classical Maze Router

- **Summarizing three main steps:**

- **Expand**
  - Breadth-first-search to find all paths from source to target
- **Backtrace**
  - Walk shortest path back to the source and mark path cells as used
- **Clean-Up**
  - Erase all distance marks from other grid cells before next net is route.

Some Algorithmic Concerns

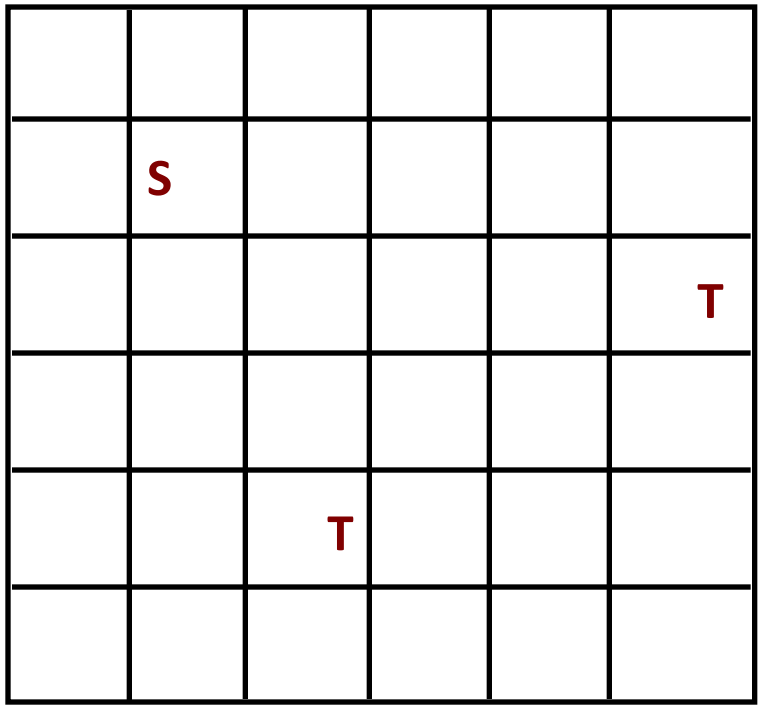
- **Storage**
  - Do we need a really big grid to represent a big routing problem?
  - What information is required in each cell of this grid?
- **Complexity**
  - Do we really have to search the whole grid each time we add a wire?
- **Technology**
  - Just 1 wiring layer? How do we do 2 layers? 3? 4? 6? 8? 10?
  - How do we deal with vias (connecting different routing layers?)
- **2 issues here**
  - **Applications** of basic algorithm versus **implementation** issues

How do we Deal Multi-point Nets

- **Multi-point Nets**
  - **One** source → **Many** targets
  - You get this with any net that represents **fanout** (ie, almost all real nets)

- **Simple strategy**
  - **Start**: Pick **one** point as source, label **all the others** as targets
  - **First**: Use maze route algorithm to find path from source to **nearest** target
    - **Note**: You don't know which one this is up front, routing will find it
  - **Next**: Re-label all cells on found path as sources, then rerun maze router
    using all sources simultaneously
  - **Repeat**: For each remaining unconnected target point

Multi-point Nets



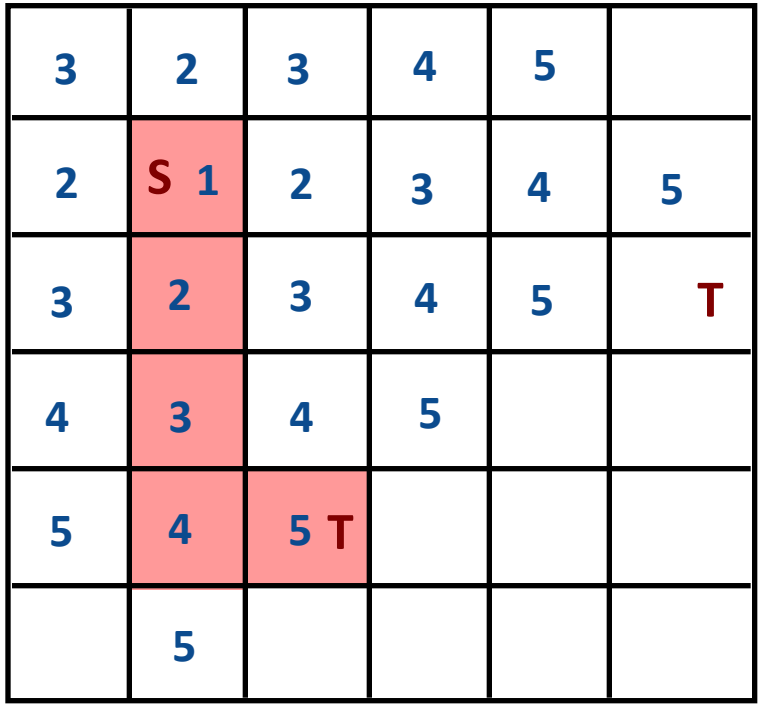
- **Given:**
  - A source and **many targets**
- **Problem:**
  - Find a **short** path connecting source and targets

Multi-point Nets

3	2	3	4	5	
2	S 1	2	3	4	5
3	2	3	4	5	T
4	3	4	5		
5	4	5 T			
	5				

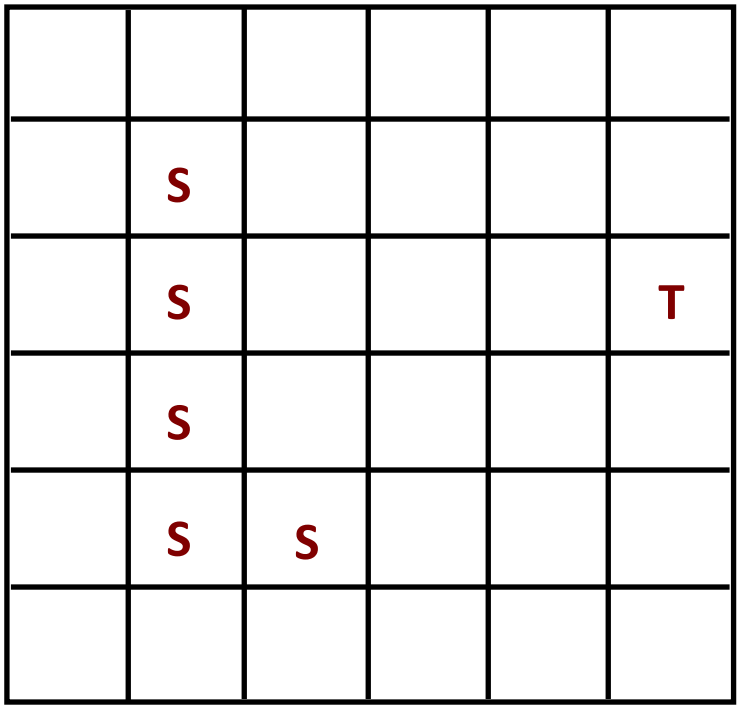
- **First segment of path...**
  - Run maze route to find the closest target
  - Start at source, go till we find **any target**

Multi-point Nets



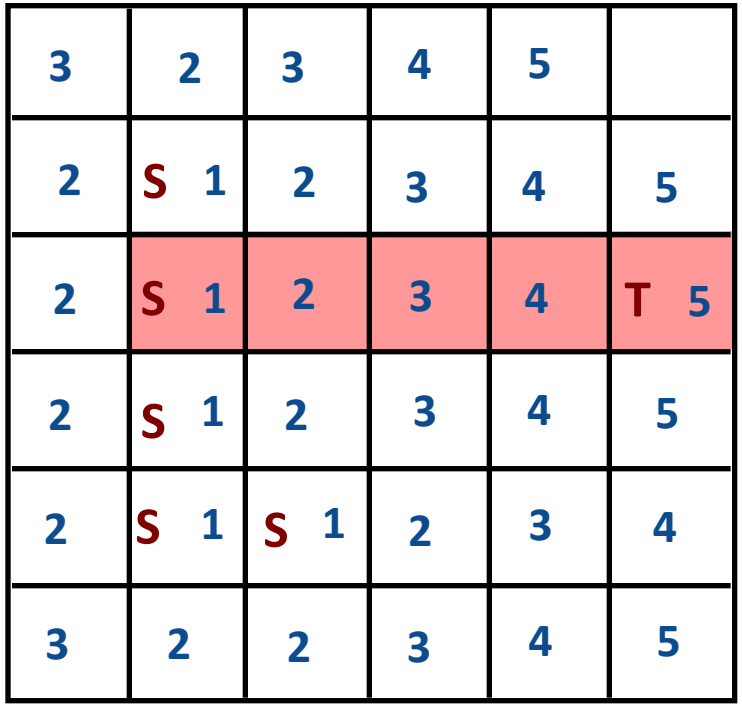
- **Second...**
  - **Backtrace and re-label the whole route as Sources for the next pass**
- **Note – this is different**
  - We don't relabel the path as blockage (yet), as we did before
  - We label it as source, so we can find paths from any point on this segment, to the rest of the targets

Multi-point Nets



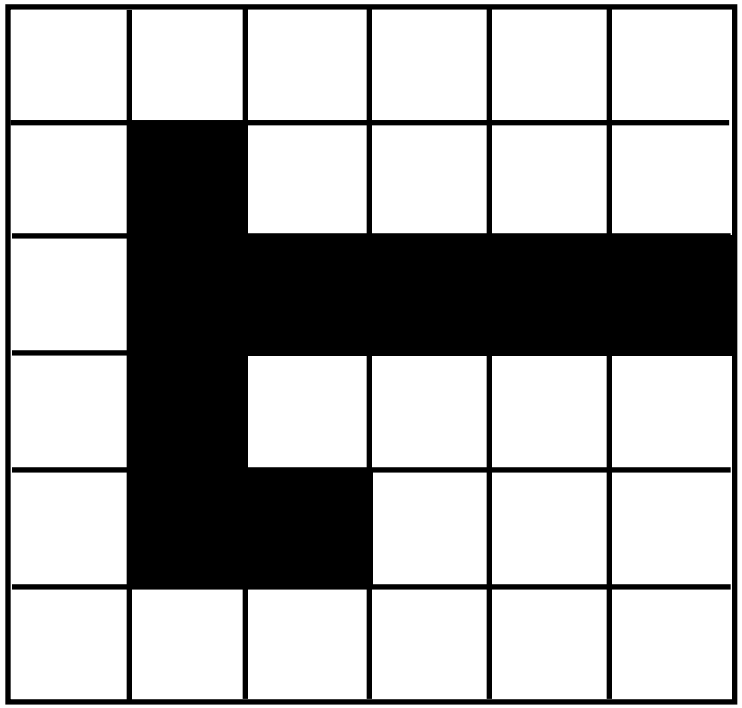
- **Second...**
  - We will expand this entire set of source cells to find next segment of the net
  - Idea is we will look for paths of length 1 away from this whole set of sources, then length 2, 3, etc.
  - Go till hit another target

Multi-point Nets



- **Trick**
  - Expand from **all these sources** to find the shortest path from the existing route to the next target
- **Next: Backtrace as before**
  - Follow pathlengths in decreasing order from target, to **some** source cell

Multipoint Nets



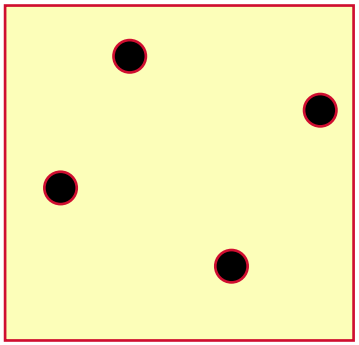
- **Finally**
  - Do usual cleanup
  - Mark all of the segment cells as used and clean-up the grid
  - Now, have embedded a multipoint net, and rendered it an obstacle for future nets

Is this Strategy Optimal?

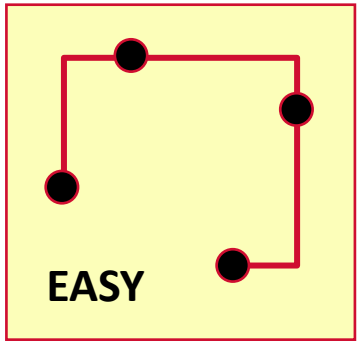
- **Does this method give us guaranteed shortest multi-point net?**
- **No!**
  - Maybe surprising, but this is just a good heuristic
  - The optimal path has a name: called a **Steiner Tree**

- **How hard is to get the optimal Steiner Tree?**
  - NP-hard! (ie, exponentially hard)
  - Yet another example of why CAD is full of tough, important problems to solve

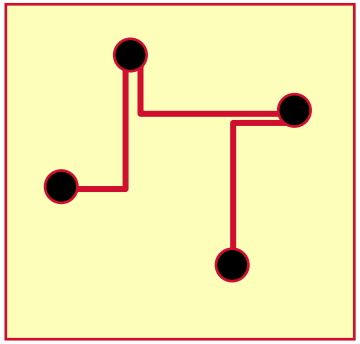
Steiner Tree Construction



Pins to connect

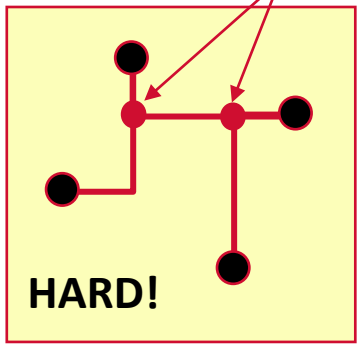


Route it so we guarantee each 2-point path is shortest; this is **Minimum Spanning Tree**



Redraw it--different orientations of 2-point paths

2 so-called "Steiner-points"



Now we can see the better (shorter) Steiner tree

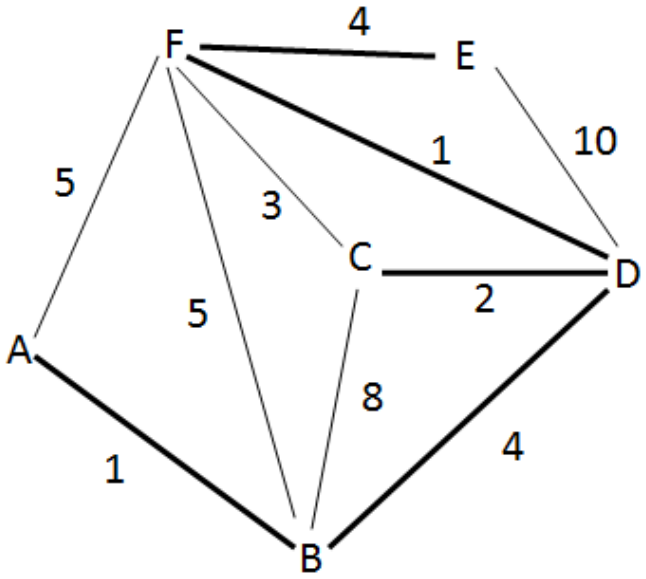
Real Routers …

- **Are often iterative**
  - **Problem:** How do you know where the wire wants to go? You don't know till you know where the other wires want to go?
  - **Solution:** So, route all the wires, leave some info on the routing surface, or even previous wires, about congestion, overlaps, etc. Then iteratively ripup (remove) and reroute the wires, using cost info from geometric footprint of previous wires.

- **Are always hierarchical**
  - **Problem:** measured in terms of metal pitch, big chip has an intractable number of locations to search for each wire
  - In 2008, the M1 pitch was about 120nm. A simple gridded router for 1cm x 1cm chip = 83K x 83K cells = ~7 billion grid cells (ugh)
  - This is *waaaay* too many places to look for each path…

Minimum Spanning Tree

- **A fast alternative to Steiner tree routing**
  - Not optimal, but quite good in practice
- **Input: graph G with weights on the edges**
- **Output: connected sub graph G' of G that includes all the vertices of G, of minimum total weight**



Exhaustive Search

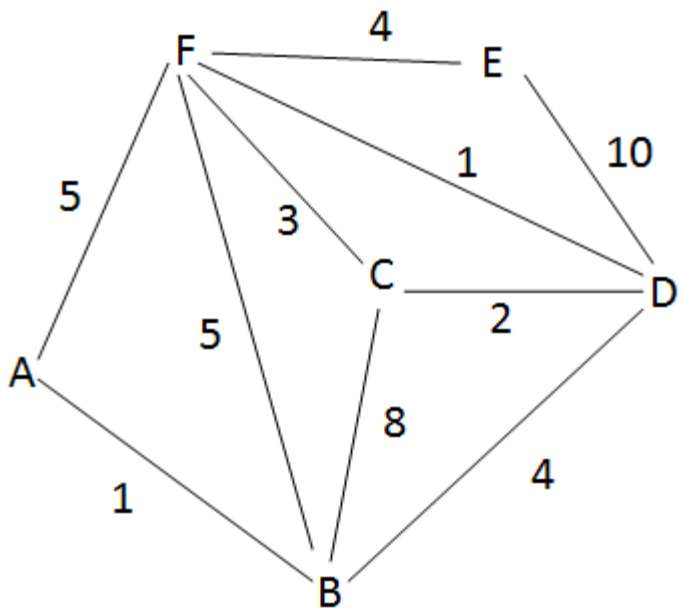
- List all connected sub-graphs of G
- Return sub-graph with least weight
- Number of vertices, N, and number of edges, M
- $O(m^{n-1})$

Greedy Algorithm

- **Start with a graph containing just the vertices, G'={V,Ø}**
- **Add edges with the least weight until the graph is connected but no cycles are created**
- **To do this:**
  - Order the edges in increasing weight
  - While the graph is not connected, add an edge
  - End when all vertices are connected

Greedy Algorithm - Example

Initial Graph:



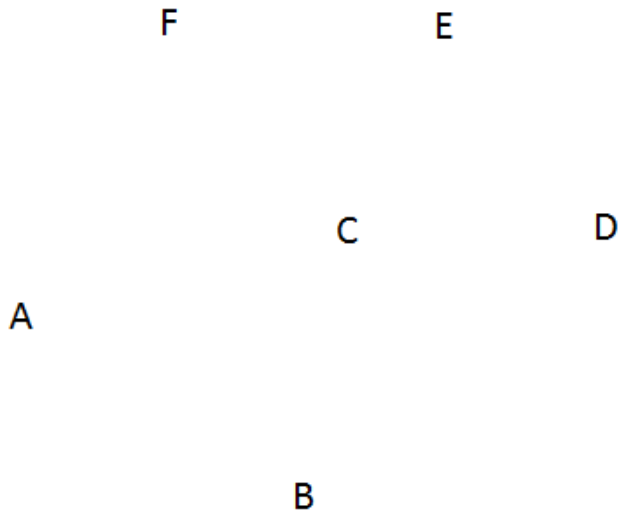
List of Edges:

Edge	Weight
AB	1
AF	5
BC	8
BD	4
BF	5
CD	2
CF	3
DE	10
DF	1
EF	4

Greedy Algorithm - Example

Start with G'={V,Ø} and sorted list of edges

Graph:



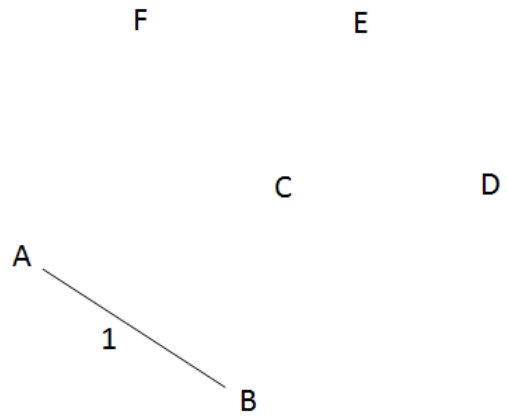
Sorted List of Edges:

Edge	Weight
AB	1
DF	1
CD	2
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created
Remove edge from list

Graph:



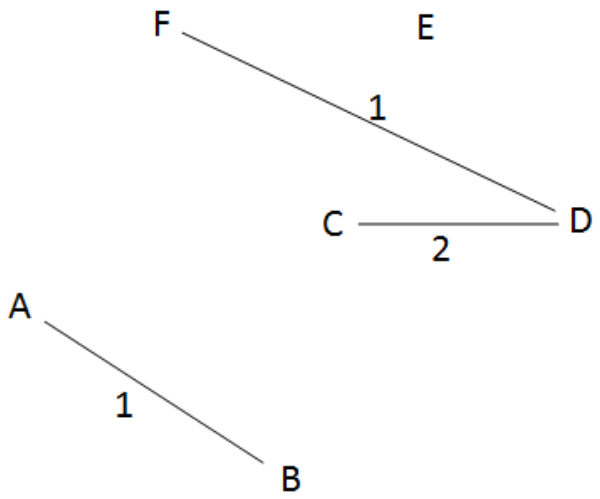
Sorted List of Edges:

Edge	Weight
~~AB~~	~~1~~
DF	1
CD	2
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created
Remove edge from list

Graph:

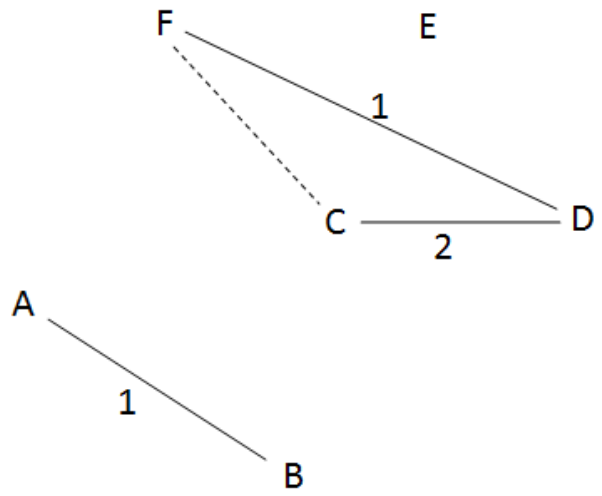


Sorted List of Edges:

Edge	Weight
~~AB~~	~~1~~
~~DF~~	~~1~~
~~CD~~	~~2~~
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

Greedy Algorithm - Example

Graph:



Sorted List of Edges:

Edge	Weight
CF	3
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

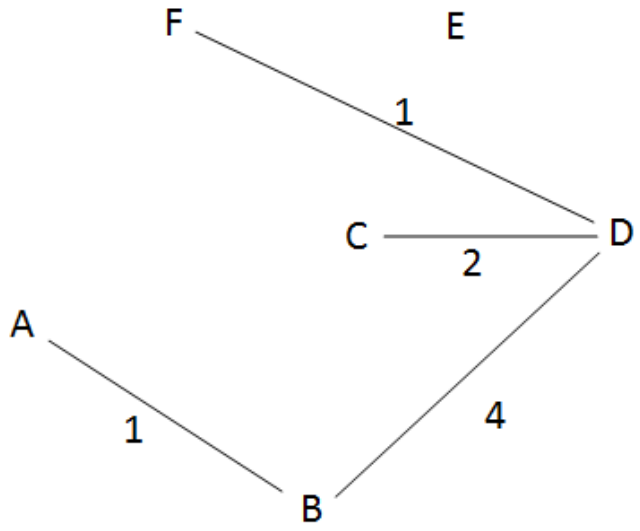
Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at C
- C-> D -> F -> C
- C gets visited twice => a cycle exists and CF should not be added to the graph

Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created
Remove edge from list
End when all vertices can be visited (graph is connected)

Graph:



Sorted List of Edges:

Edge	Weight
BD	4
EF	4
AF	5
BF	5
BC	8
DE	10

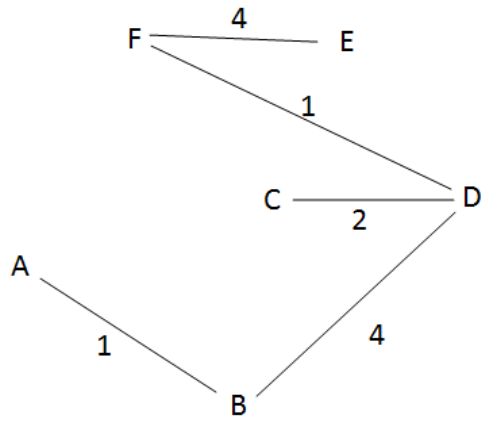
Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at B
- B ->D ->C -> F -> A
- If all vertices are visited by DFS, then the graph is connected and we are done

Greedy Algorithm - Example

Add edge from list to graph s.t. no cycles are created
Remove edge from list
End when all vertices can be visited (graph is connected)

Graph:



Sorted List of Edges:

Edge	Weight
EF	4
AF	5
BF	5
BC	8
DE	10

Check for cycles using Depth First Search starting at a vertex in the added edge:

- Start at E
- E -> F -> D -> B -> A ->C
- All vertices were visited and there were no cycles => we have found a minimum spanning tree with weight 12.

Greedy Algorithm- Psuedocode

- **Given G = (V, E)**
- **G' <- (V,Ø)**
- **While G' is not connected**
  - Add e Є E to G' s.t. G' is acyclic and e is minimum
  - Remove e from E

Summary

- **Graph algorithms**
  - DFS
  - BFS
- **Routing application in circuit designs**
- **Minimum spanning tree**
- **Final starts on 12/8 and ends at 23:59 PM 12/12**
  - Cover everything in the class