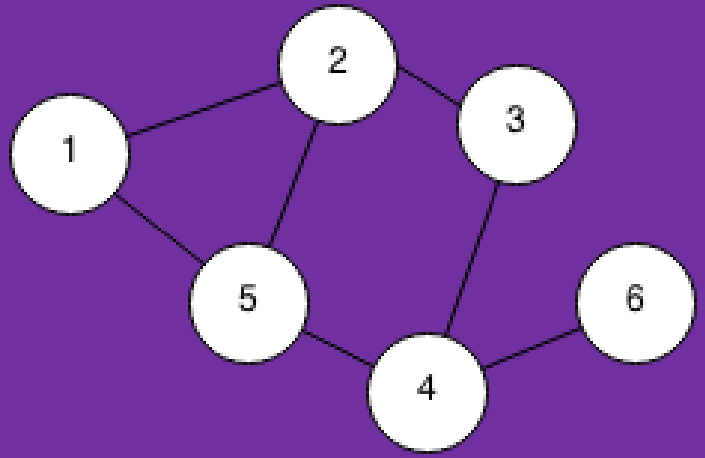
CSE 332 Section 6

Graphs



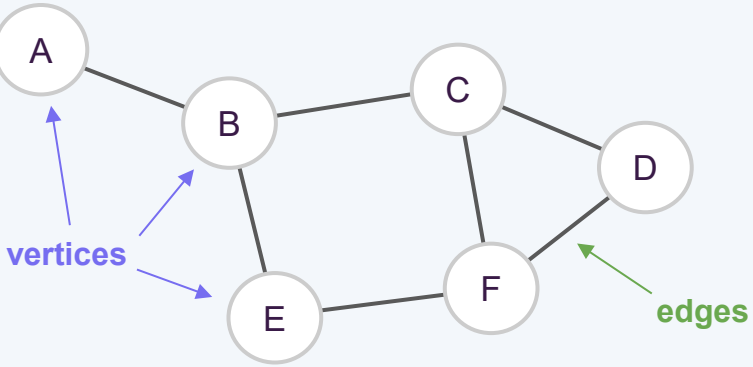
Original slides by James Richie Sulaeman
Revised by Rubee Zhao & Vicky Ye

Graphs

Graphs: Vertices and Edges

A graph is a set of **vertices** connected by **edges**

- A vertex is also known as a node
- An edge is represented as a pair of vertices



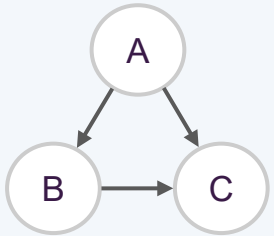
example of a undirected, unweighted, cyclic graph

Graph Terminology

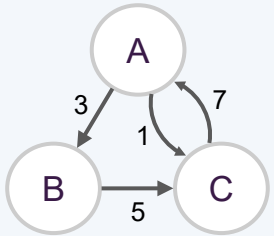
- Degree of vertex *V*
  - Number of edges connected to vertex *V*
  - In-degree: number of edges going into vertex *V*
  - Out-degree: number of edges going out of vertex *V*
- Weight of edge *e*
  - Numerical value/cost associated with traversing edge *e*
- Path
  - A sequence of adjacent vertices connected by edges
- Cycle
  - A path that begins and ends at the same vertex

Graph Terminology (continued)

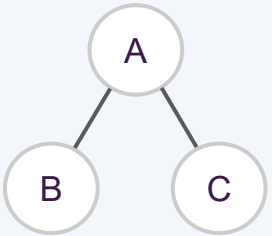
- Directed vs. undirected graphs
  - Edges can have direction (i.e. bidirectional vs. unidirectional)
- Weighted vs. unweighted graphs
  - Edges can have weights/costs (e.g. how many minutes to go from vertex *A* to *B*)
- Cyclic vs. acyclic graphs
  - Graph contains a cycle



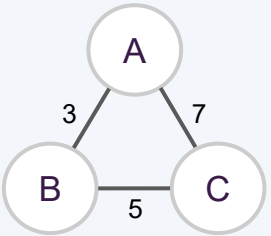
directed, unweighted, acyclic graph



directed, weighted, cyclic graph



undirected, unweighted, acyclic graph



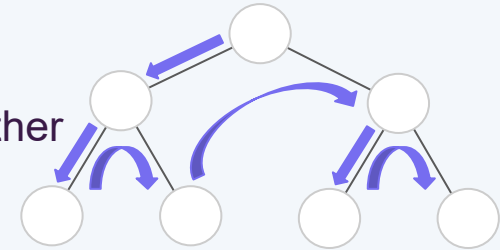
undirected, weighted, cyclic graph

Graph Traversal

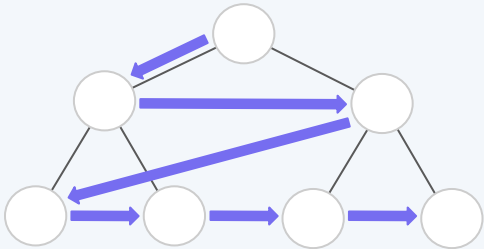
Graph Traversals

How do we iterate through a graph?

- Depth First Search (DFS)
  - Visit all nodes reachable by a neighbor before another
  - Can be implemented recursively/by using a stack (not recommend)
  - $O(|V| + |E|)$ runtime
- Breadth First Search (BFS)
  - Explores the graph level by level
  - Implemented using a queue
  - Finds the shortest path in an unweighted graph
  - $O(|V| + |E|)$ runtime



Depth First Search (DFS)



Breadth First Search (BFS)

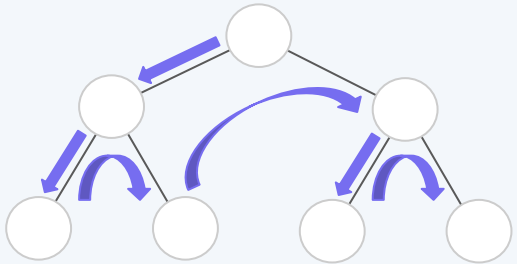
Depth First Search

```
DFS(Graph g, Vertex curr):
  mark curr as visited

  for (v : neighbors(current)):
    if (!v marked "visited")
      DFS(g, v)

  mark curr as "done";
```

- Explores the graph: visiting all nodes reachable by one neighbor before selecting another. (as deep as possible)
- Implemented recursively/by using a stack (not recommend)
- $O(|V| + |E|)$ runtime



Depth First Search (DFS)

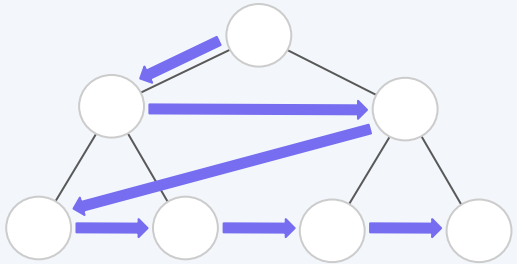
Breadth First Search

```
BFS(Vertex start):
  initialize queue q to hold start
  mark start as visited

  while q is not empty:
    vertex v = q.dequeue()

    for each neighbour u of v:
      if u is not visited:
        mark u as visited
        predecessor[u] = v
        add u to q
```

- Explores the graph level by level
- Implemented using a queue
- Finds the shortest path in an unweighted graph
- $O(|V| + |E|)$ runtime



Breadth First Search (BFS)

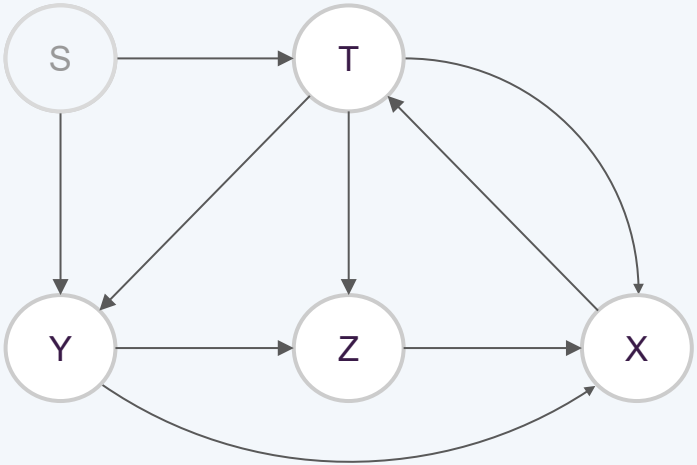
Problem 0

```
BFS(Vertex start):
  initialize queue q to hold start
  mark start as visited

  while q is not empty:
    vertex v = q.dequeue()

    for each neighbour u of v:
      if u is not visited:
        mark u as visited
        predecessor[u] = v
        add u to q
```

Problem 0 (1)



Queue:

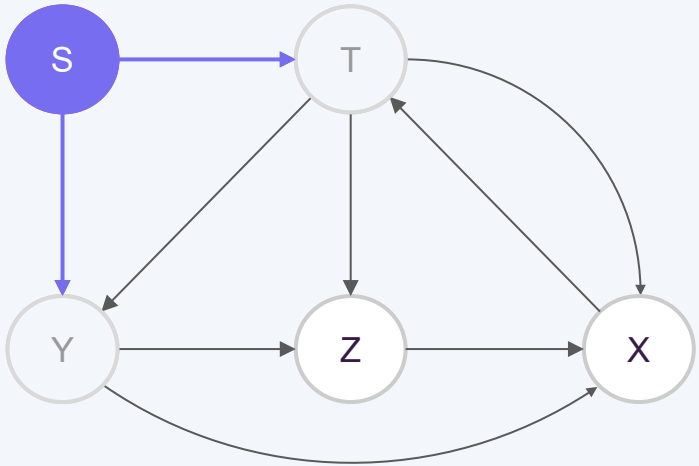
S				

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	–	No
X	–	No
Y	–	No
Z	–	No

- Initialize queue to hold starting vertex S
- Mark vertex S as visited

Problem 0 (2)



Queue:

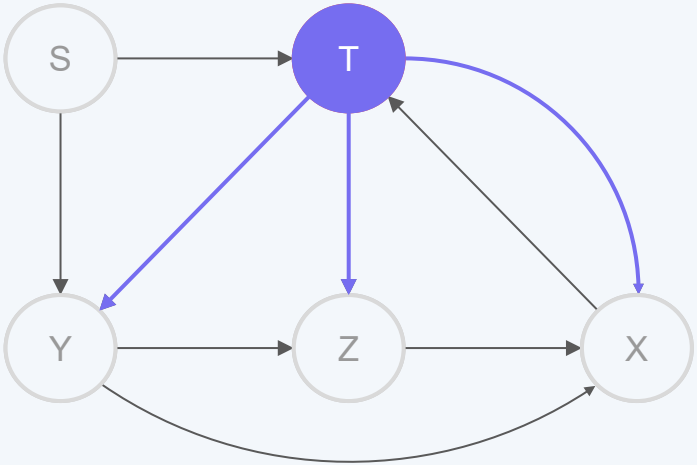
T	Y			

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	–	No
Y	S	Yes
Z	–	No

- Dequeue vertex S
- Add neighbors T, Y to the queue

Problem 0 (3)



Queue:

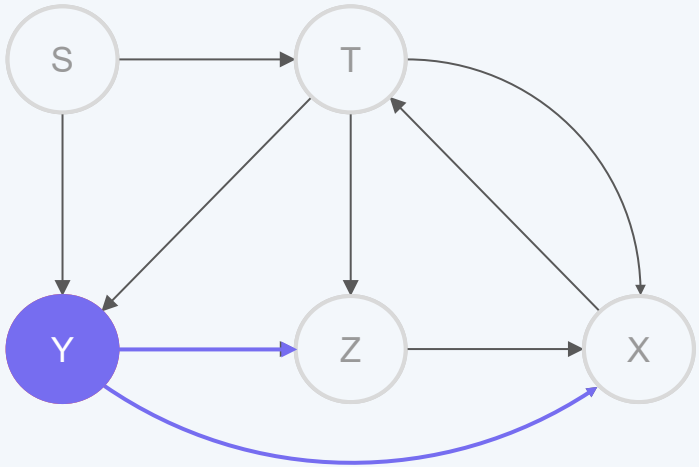
Y	X	Z		

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

- Dequeue vertex T
- Add neighbors X, Z to the queue (ignore Y since already visited)

Problem 0 (4)



Queue:

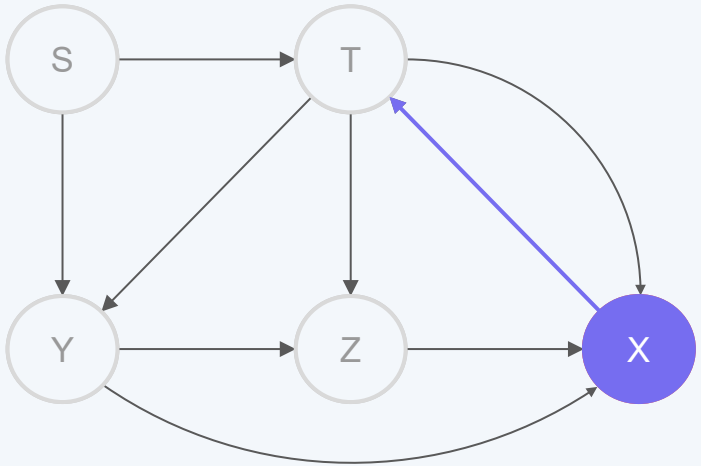
X	Z			

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

- Dequeue vertex Y
- Add neighbors to the queue
  (nothing happens since all already visited)

Problem 0 (5)



Queue:

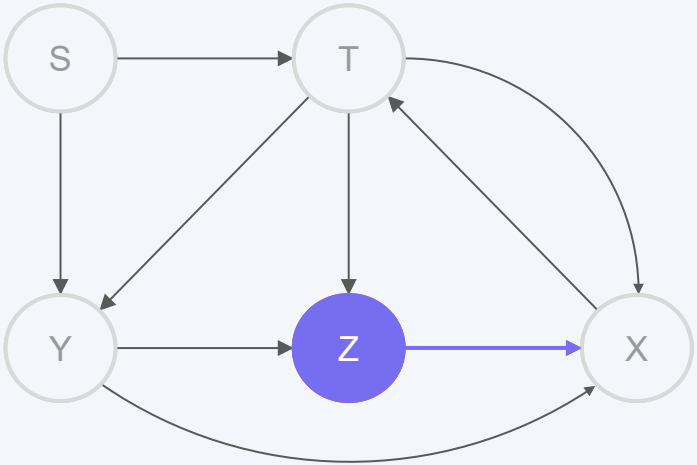
Z				

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

- Dequeue vertex X
- Add neighbors to the queue
  (nothing happens since all already visited)

Problem 0 (6)



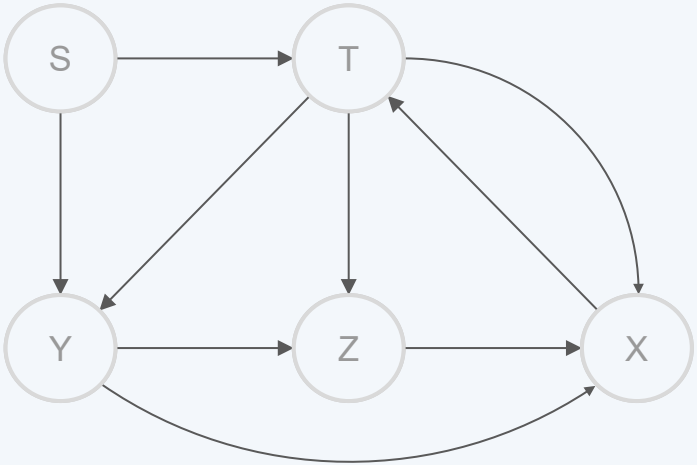
Queue:

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

- Dequeue vertex Z
- Add neighbors to the queue
  (nothing happens since all already visited)

Problem 0 (7)



Queue:

front back

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

- Queue is empty; we are done

BFS Table Interpretation

BFS Table Interpretation Example

How to check if a path exists from the start node to a target node?

- A path exists **if and only if** the target node has a predecessor in the table

How to find a path from the start node to a target node?

- Locate the target node in the table
- Backtrack through its predecessors until you reach the start node
- The sequence of predecessors form a path from the start to the target
- **Will be the shortest path by edge count (but not necessarily sum of edge costs)**

Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

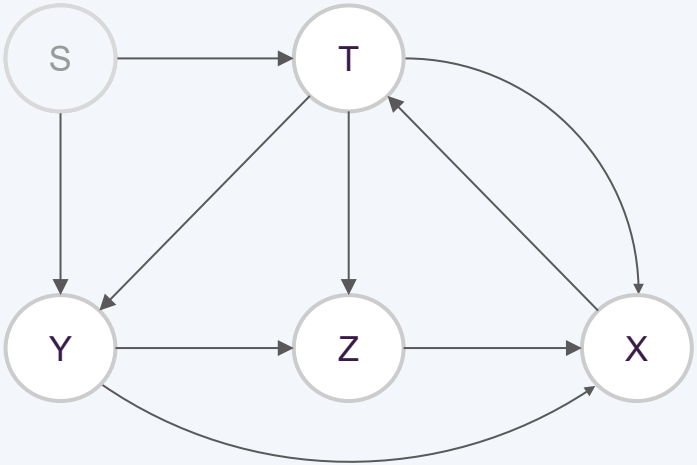
Problem 1

```
DFS(Graph g, Vertex curr):
  mark curr as visited

  for (v : neighbors(current)):
    if (!v marked “visited”)
      DFS(g, v)

  mark curr as “done”;
```

Problem 1 (1)



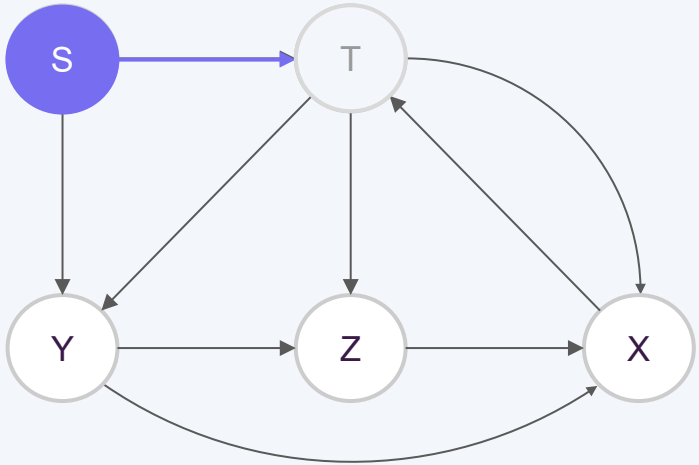
StackFrame (curr):

S

Vertex	Visited?	Done?
S	Yes	No
T	No	No
X	No	No
Y	No	No
Z	No	No

- Build stack frame of dfs(g, S)
- Mark vertex S as visited

Problem 1 (2)



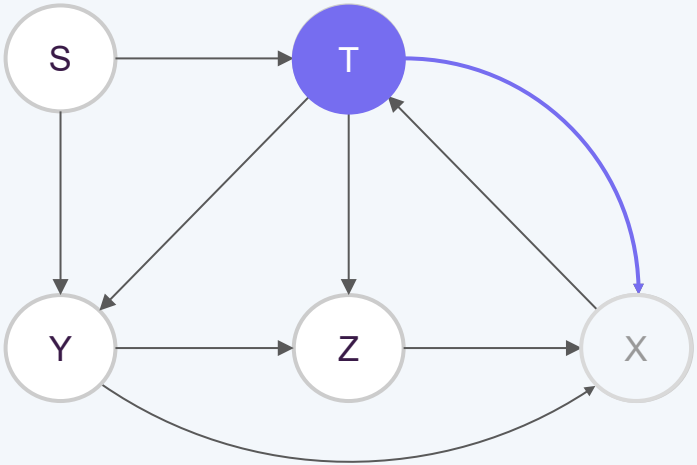
StackFrame (curr):

S	T

Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	No	No
Y	No	No
Z	No	No

- S Call dfs on unvisited neighbor T
- Mark vertex T as visited

Problem 1 (3)



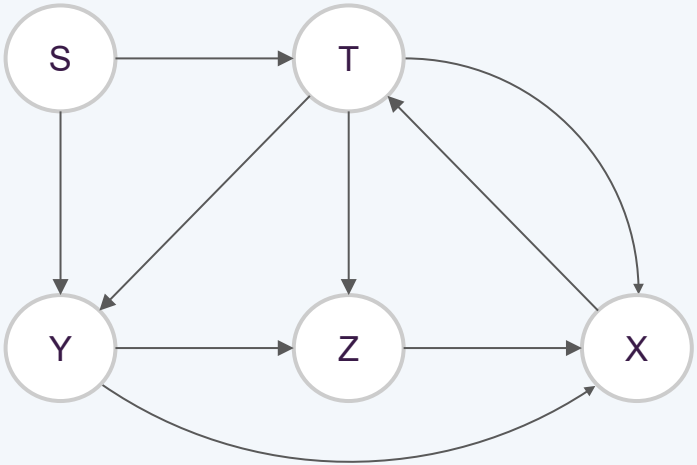
StackFrame (curr):

S	T	X

Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	No
Y	No	No
Z	No	No

- T Call dfs on unvisited neighbor X
- Mark vertex X as visited

Problem 1 (4)



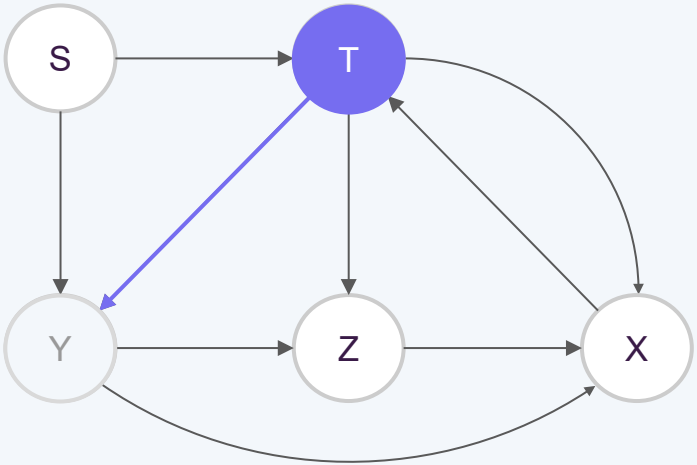
StackFrame (curr):

S	T	X

Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	No	No
Z	No	No

- No recursive call in X: all neighbors are visited
- Mark vertex X as done, exit X stack frame

Problem 1 (5)



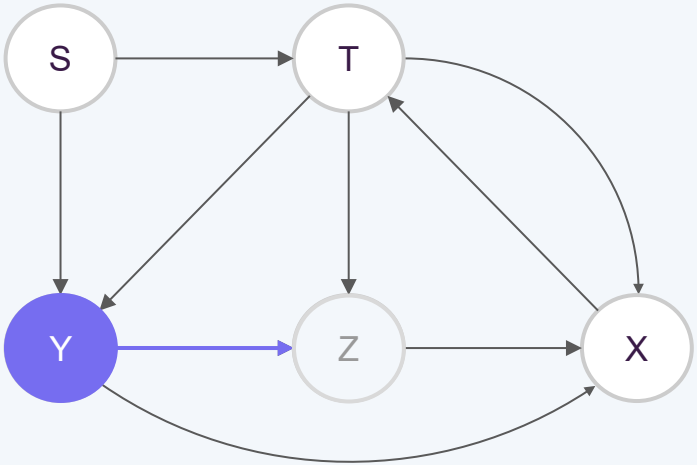
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	Yes	No
Z	No	No

StackFrame (curr):

S	T	X
		Y

- T Call dfs on unvisited neighbor Y
- Mark vertex Y as visited

Problem 1 (6)



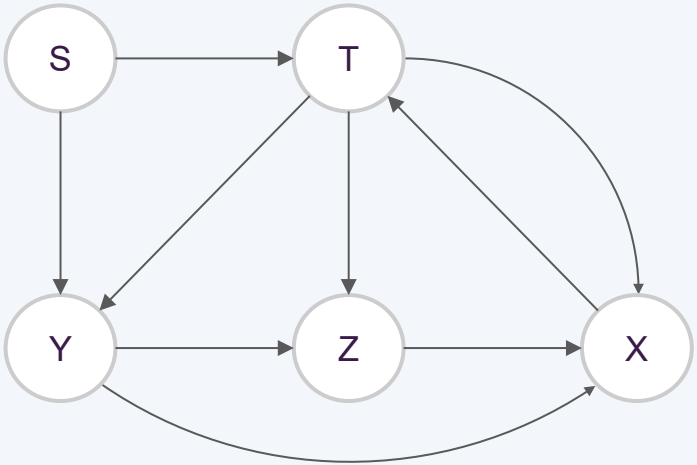
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	Yes	No
Z	Yes	No

StackFrame (curr):

S	T	X	
		Y	Z

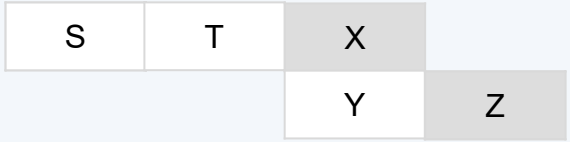
- Y Call dfs on unvisited neighbor Z
- Mark vertex Z as visited

Problem 1 (7)



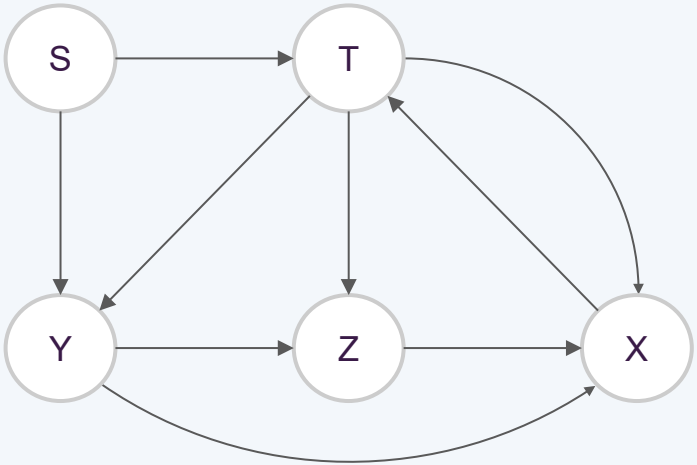
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	Yes	No
Z	Yes	Yes

StackFrame (curr):



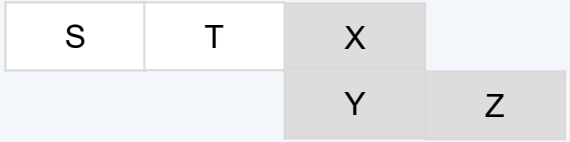
- No recursive call in Z: all neighbors are visited
- Mark vertex Z as done, exit Z stack frame

Problem 1 (8)



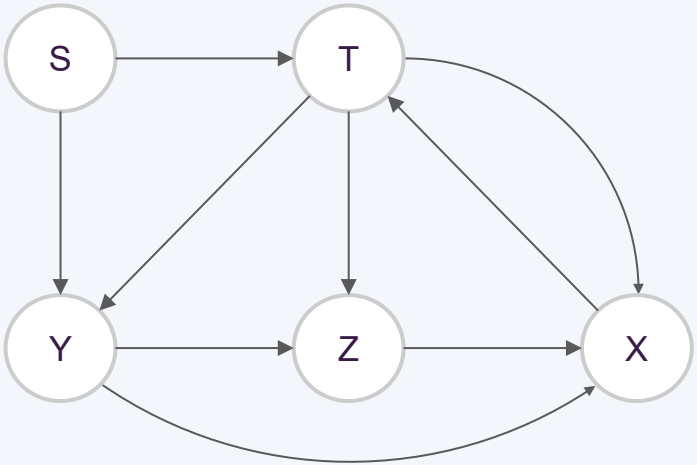
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	Yes	Yes
Z	Yes	Yes

StackFrame (curr):



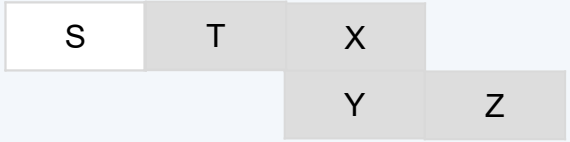
- Finish all recursive in Y: all neighbors are visited
- Mark vertex Y as done, exit Y stack frame

Problem 1 (9)



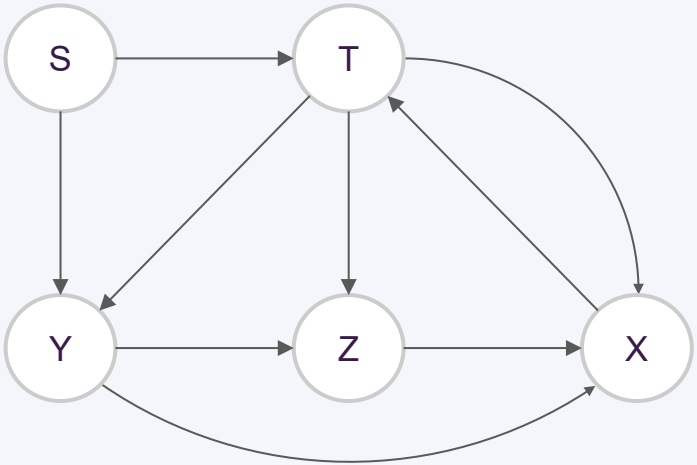
Vertex	Visited?	Done?
S	Yes	No
T	Yes	Yes
X	Yes	Yes
Y	Yes	Yes
Z	Yes	Yes

StackFrame (curr):



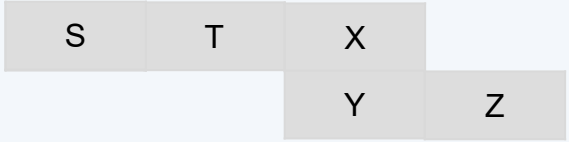
- Finish all recursive in T: all neighbors are visited
- Mark vertex T as done, exit T stack frame

Problem 1 (10)



Vertex	Visited?	Done?
S	Yes	Yes
T	Yes	Yes
X	Yes	Yes
Y	Yes	Yes
Z	Yes	Yes

StackFrame (curr):

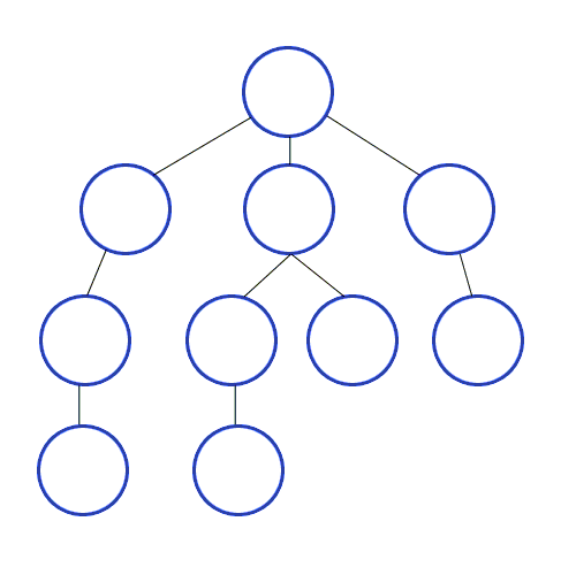


- Finish all recursive in S: all neighbors are visited
- Mark vertex S as done, exit S stack frame

BFS/DFS Useful Properties

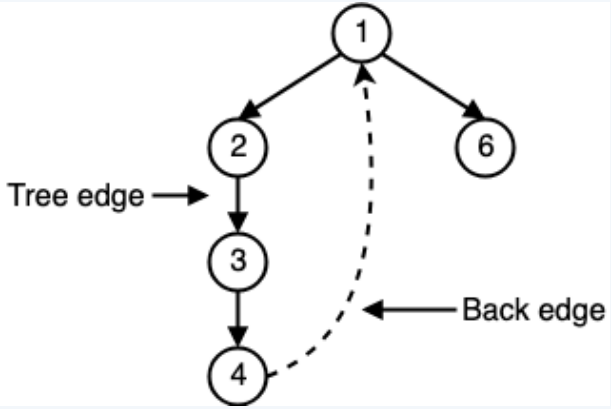
BFS - Shortest Path

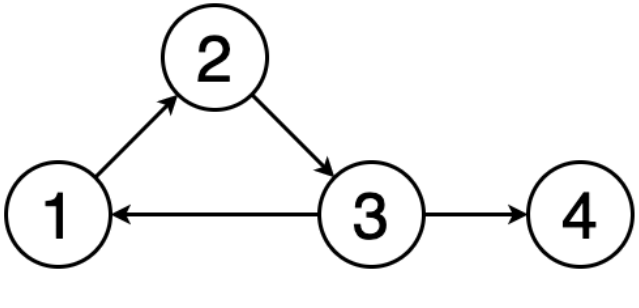
- BFS **always** returns the **shortest path from source to any other vertex by edge count**!
- Intuition:
  - Each step push neighbors that are one edge away, onto a queue.
  - **Because we use a queue,** we must process the vertices 1 edge away, before vertices farther away
  - Each vertex's predecessor in the table is the one which initially pushes it onto the queue (earliest/shortest path)



DFS - Cycle Detection

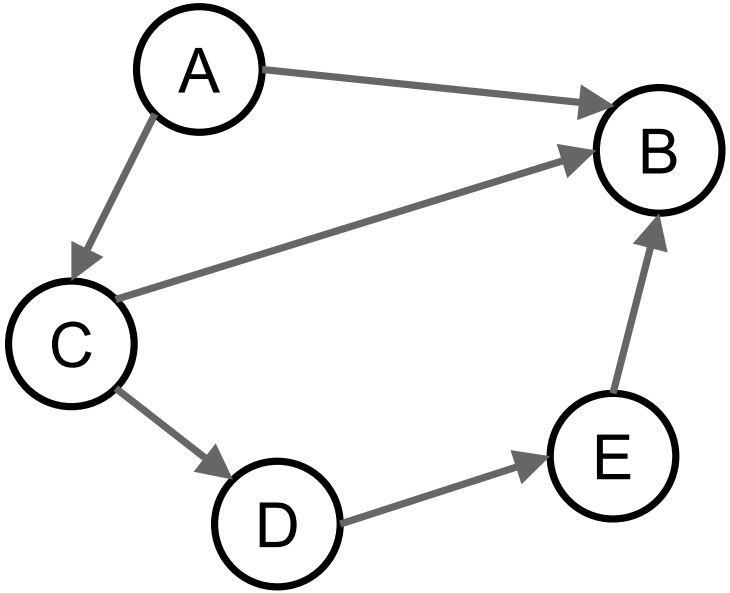
- DFS can be used to detect cycles in graphs
- Intuition:
  - Each step push neighbors that are one edge away from the current node onto a stack.
  - **Because we use recursion,** we process the vertices further away, before the vertices close to the source
  - If we ever see a vertex to a node currently on the stack, we know a back edge and thus a cycle exists





Detect Cycle Examples

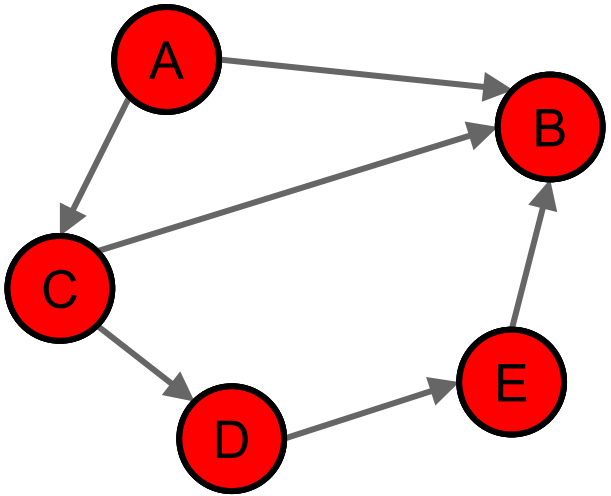
Start from A



- Use **DFS** to detect cycle by finding a back edge
- a **back edge** is an edge that connects a **node to one of its ancestors** in the current call stack.

No Cycle Example

Start from A



- Visited
- Done

Vertex: E
Mark E as done

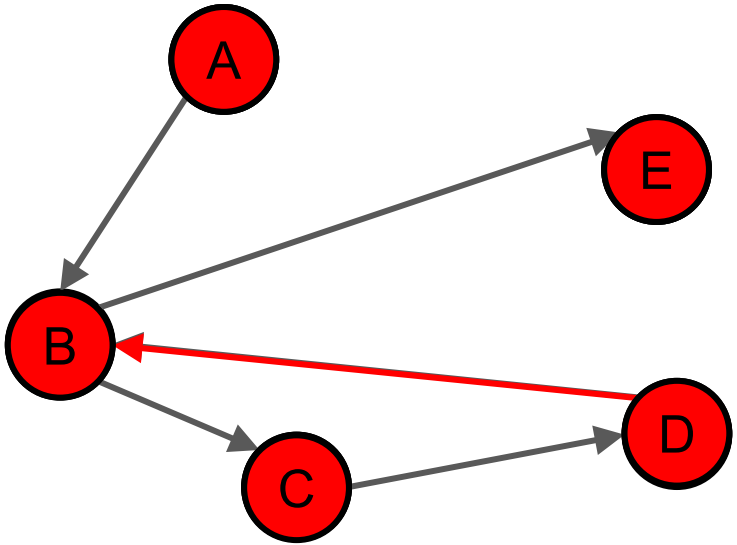
Vertex: D
Mark D as done

Vertex: C
Mark C as done

Vertex: A
Mark A as done

With Cycle Example

Start from A



Vertex: D
Mark as **done**
Edge points to a visited node, back edge detected!

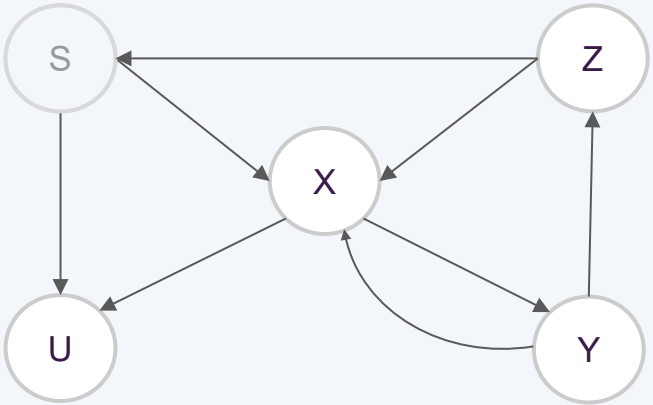
Vertex: E
Mark E as done

Vertex: B
Mark B as done

Vertex: A
Mark A as done

Problem 2: Detecting Cycles

Problem 2 (0)



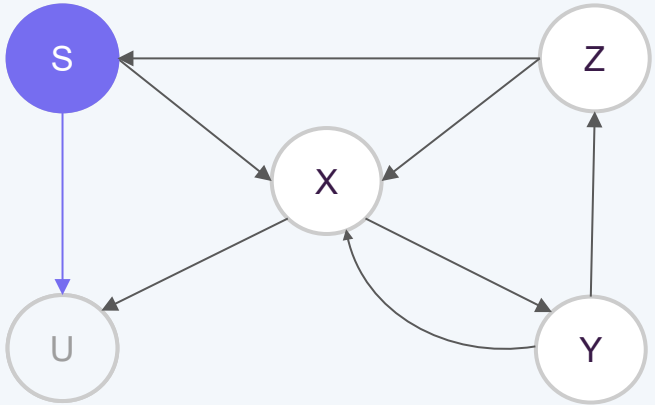
StackFrame (curr):

S

Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	No	No	None
X	No	No	None
Y	No	No	None
Z	No	No	None

- Build stack frame of dfs(g, S)
- Mark vertex S as visited

Problem 2 (1)



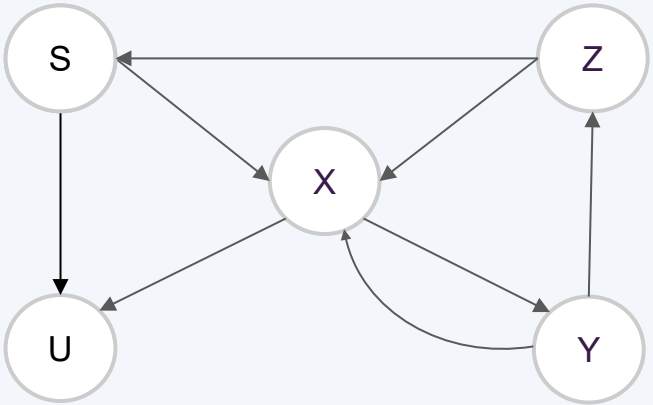
StackFrame (curr):

S	U

Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	No	None
X	No	No	None
Y	No	No	None
Z	No	No	None

- S Call dfs on unvisited neighbor U
- Mark vertex U as visited

Problem 2 (2)



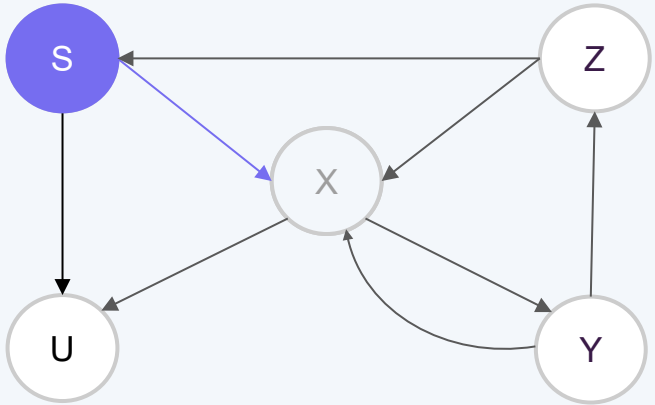
StackFrame (curr):

S	U

Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	No	No	None
Y	No	No	None
Z	No	No	None

- No recursive call in U: all neighbors are visited
- Mark vertex U as done

Problem 2 (3)



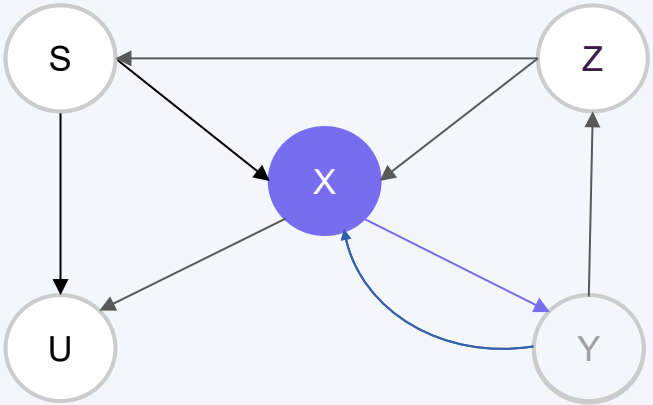
StackFrame (curr):

S	U
	X

Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	No	None
Y	No	No	None
Z	No	No	None

- S Call dfs on unvisited neighbor X
- Mark vertex X as visited

Problem 2 (4)



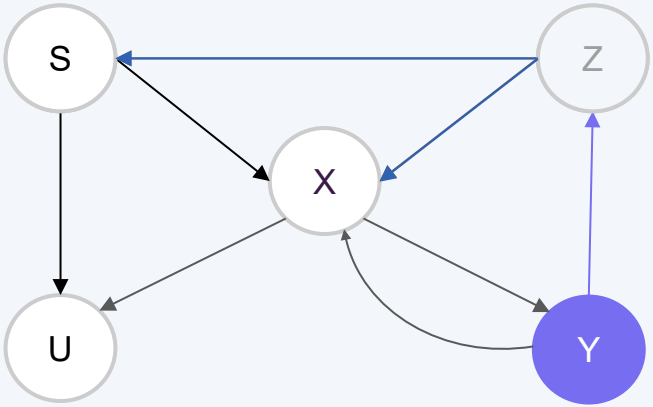
StackFrame (curr):

S	U	
	X	Y

Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	No	None
Y	Yes	No	(X,Y)
Z	No	No	None

- X Call dfs on unvisited neighbor Y
- Mark vertex Y as visited
- Check neighbors of Y: back edge to X detected

Problem 2 (5)



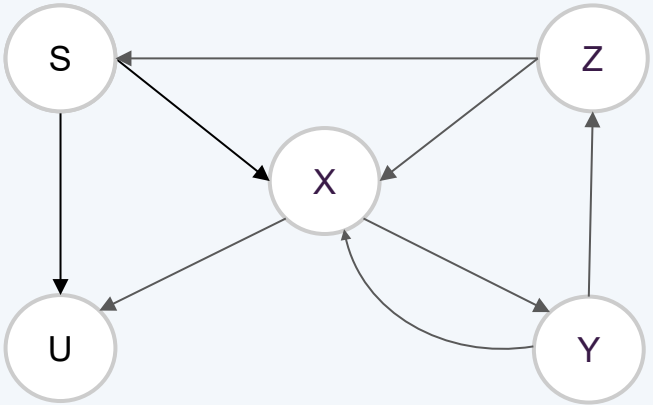
StackFrame (curr):

S	U		
	X	Y	Z

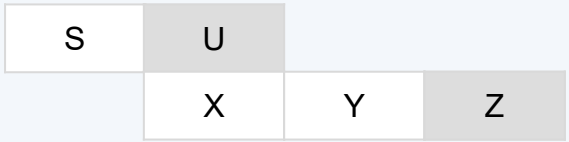
Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	No	None
Y	Yes	No	(X,Y)
Z	Yes	No	(S,X,Y,Z), (X,Y,Z)

- Y Call dfs on unvisited neighbor Z
- Mark vertex Z as visited
- Check neighbors of Z: back edge to S and X detected

Problem 2 (6)



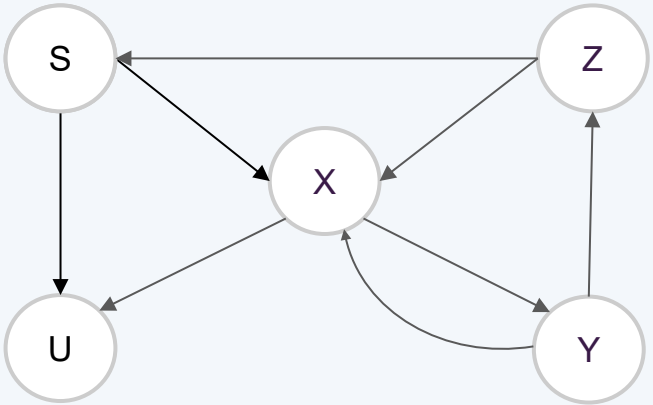
StackFrame (curr):



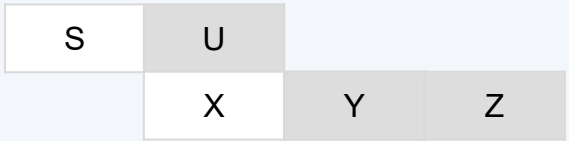
Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	No	None
Y	Yes	No	(X,Y)
Z	Yes	Yes	(S,X,Y,Z), (X,Y,Z)

- No recursive call in Z: all neighbors are visited
- Mark vertex Z as done

Problem 2 (7)



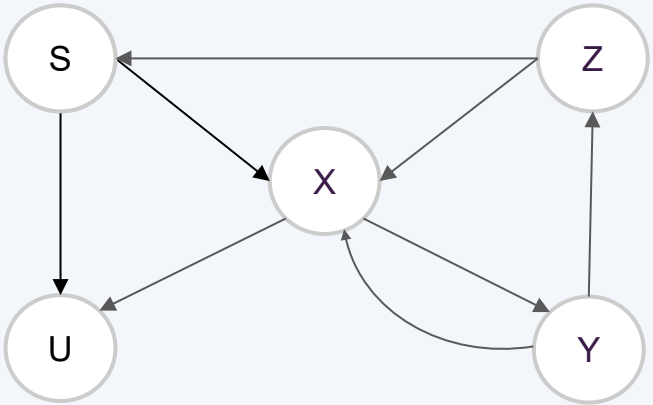
StackFrame (curr):



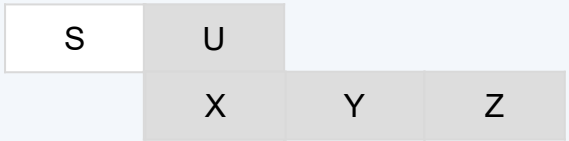
Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	No	None
Y	Yes	Yes	(X,Y)
Z	Yes	Yes	(S,X,Y,Z), (X,Y,Z)

- Finish all recursive in Y: all neighbors are visited
- Mark vertex Y as done

Problem 2 (8)



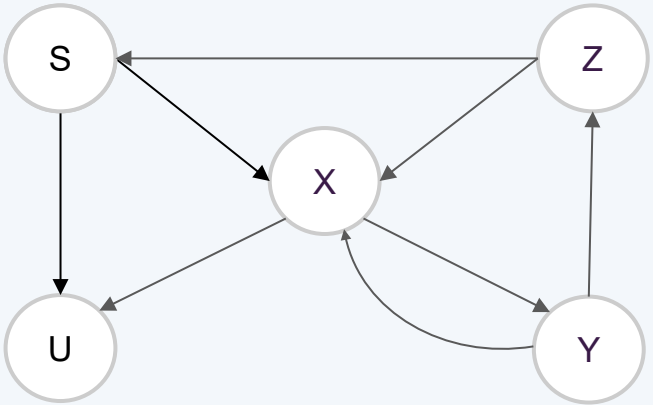
StackFrame (curr):



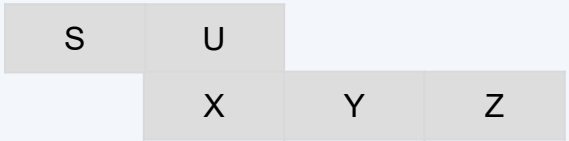
Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	No	None
U	Yes	Yes	None
X	Yes	Yes	None
Y	Yes	Yes	(X,Y)
Z	Yes	Yes	(S,X,Y,Z), (X,Y,Z)

- Finish all recursive in X: all neighbors are visited
- Mark vertex X as done

Problem 2 (9)



StackFrame (curr):



Vertex	Visited?	Done?	Cycles Detected at Vertex
S	Yes	Yes	None
U	Yes	Yes	None
X	Yes	Yes	None
Y	Yes	Yes	(X,Y)
Z	Yes	Yes	(S,X,Y,Z), (X,Y,Z)

- Finish all recursive in S: all neighbors are visited
- Mark vertex S as done

Thank You!