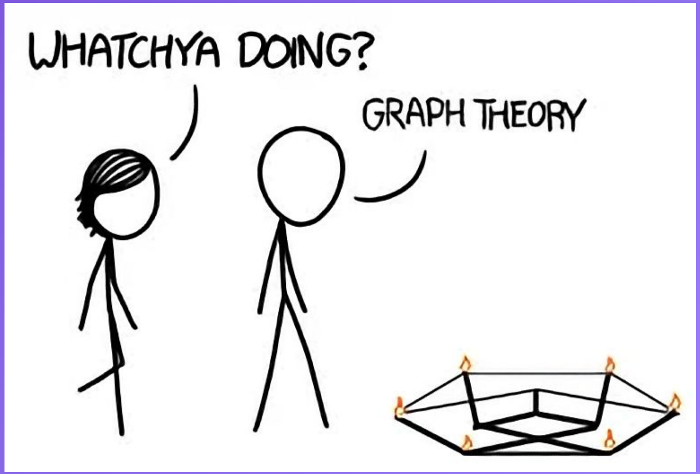
Graphs

CSE 332 – Section 6

Slides by James Richie Sulaeman

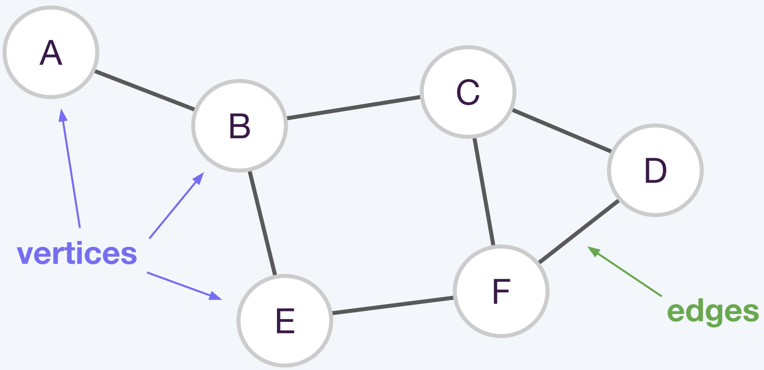


Graphs

Graphs

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



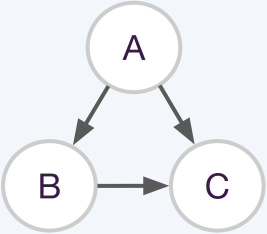
example of a undirected, unweighted, cyclic graph

Graph Terminology

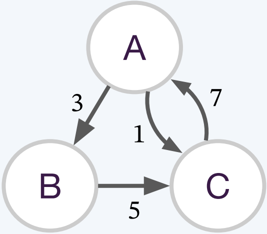
- Degree of vertex V
  - Number of edges connected to vertex V
  - In-degree: number of edges going into vertex V
  - Out-degree: number of edges going out of vertex V
- Weight of edge e
  - Numerical value/cost associated with traversing edge e
- Path
  - A sequence of adjacent vertices connected by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

Graph Terminology

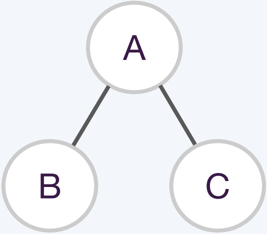
- Directed vs. undirected graphs
  - Edges can have direction (i.e. bidirectional vs. unidirectional)
- Weighted vs. unweighted graphs
  - Edges can have weights/costs (e.g. how many minutes to go from vertex A to B)
- Cyclic vs. acyclic graphs
  - Graph contains a cycle



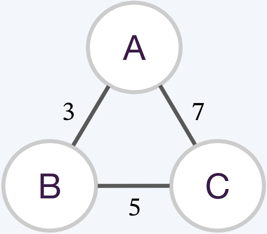
directed, unweighted, acyclic graph



directed, weighted, cyclic graph



undirected, unweighted, acyclic graph



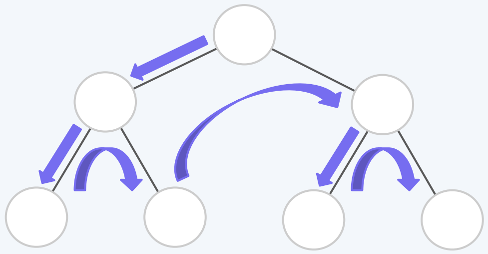
undirected, weighted, cyclic graph

Graph Traversals

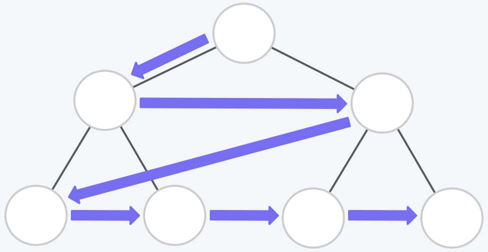
Graph Traversals

How do we iterate through a graph?

- Depth First Search (DFS)
  - Explores the graph by going as deep as possible
  - Can be implemented recursively/by using a stack
  - $\mathcal{O}(|V| + |E|)$ runtime
- Breadth First Search (BFS)
  - Explores the graph level by level
  - Implemented using a queue
  - Finds the shortest path in an unweighted, acyclic graph
  - $\mathcal{O}(|V| + |E|)$ runtime



Depth First Search (DFS)

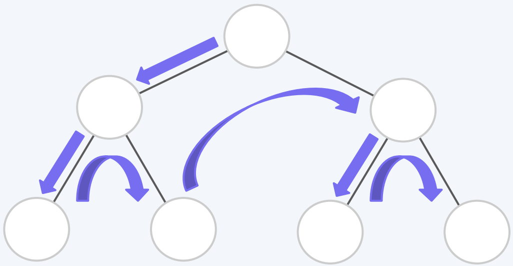


Breadth First Search (BFS)

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 by going as deep as possible
- Implemented recursively/by using a stack
- $\mathcal{O}(|V| + |E|)$ runtime

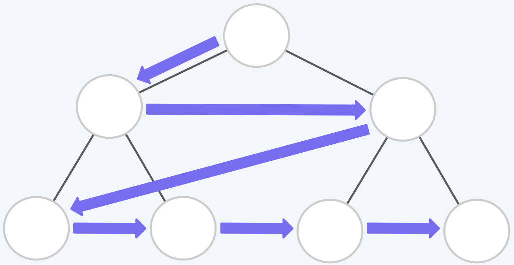


Depth First Search (DFS)

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, acyclic graph
- $\mathcal{O}(|V| + |E|)$ runtime

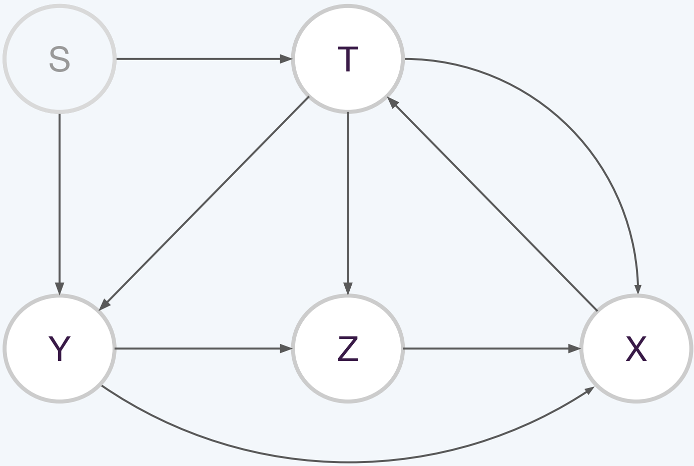


Breadth First Search (BFS)

Problem 0

```
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 0



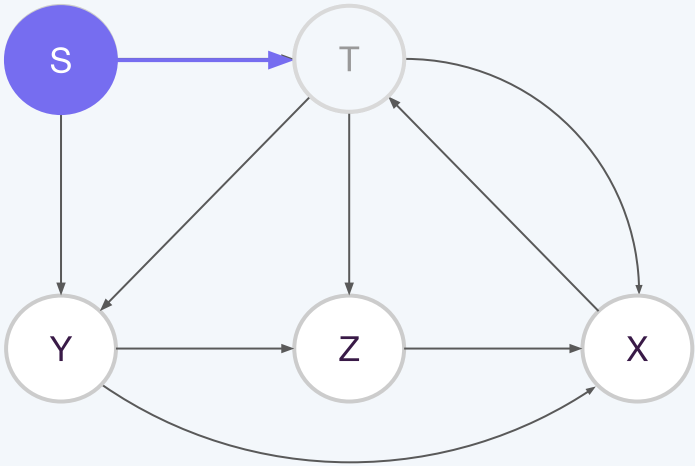
StackFrame (curr):

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 0



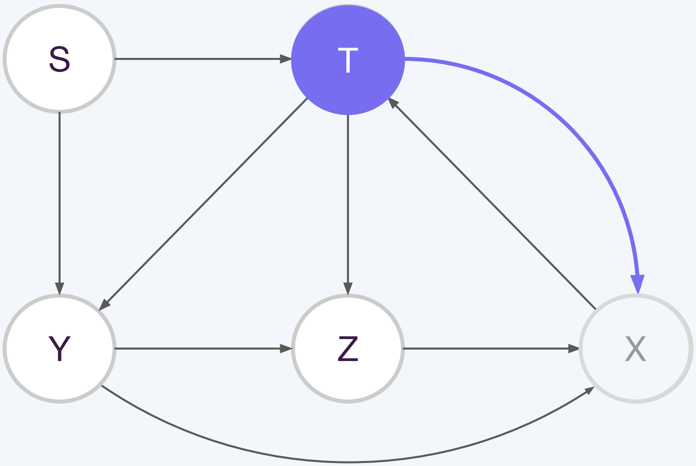
StackFrame (curr):

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 0



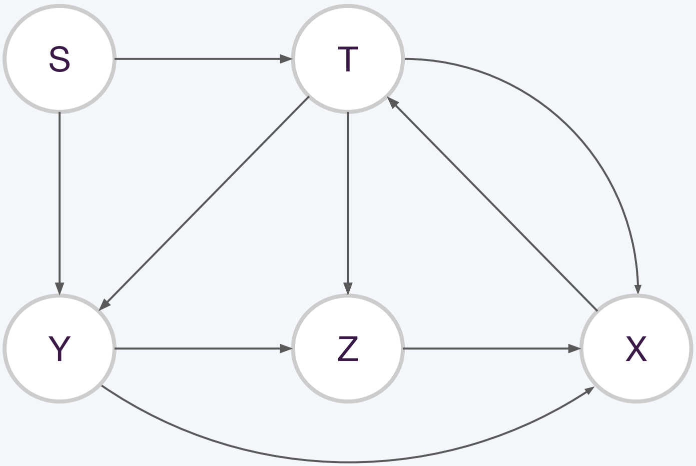
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	No
Y	No	No
Z	No	No

StackFrame (curr):

S	T	X

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

Problem 0



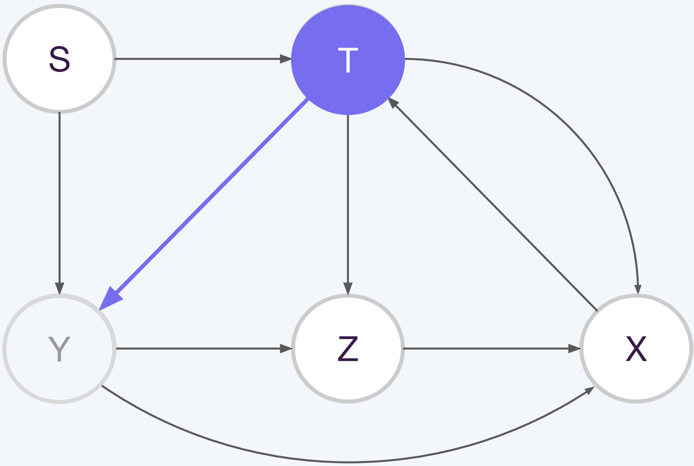
StackFrame (curr):

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 0



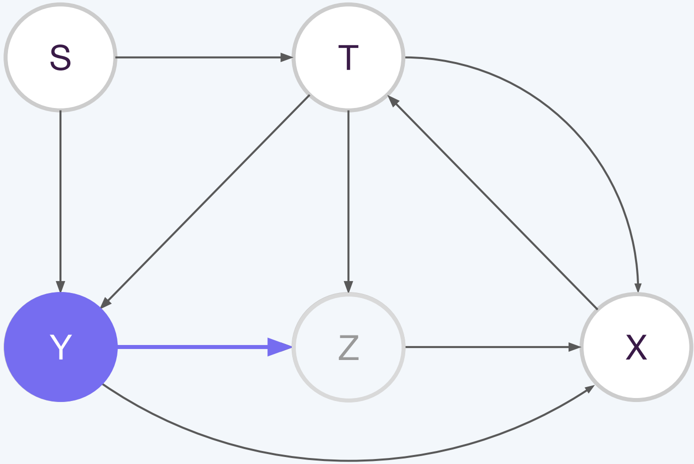
StackFrame (curr):

S	T	X
		Y

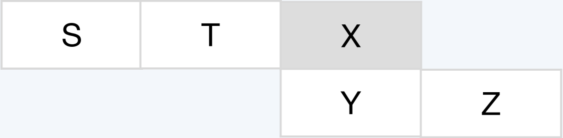
Vertex	Visited?	Done?
S	Yes	No
T	Yes	No
X	Yes	Yes
Y	Yes	No
Z	No	No

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

Problem 0



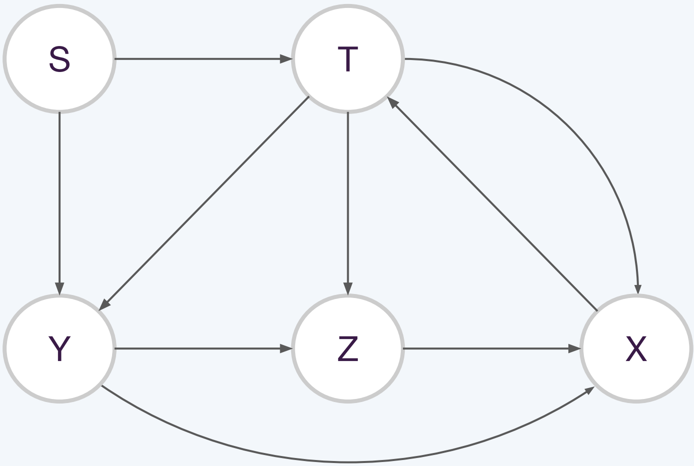
StackFrame (curr):



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

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

Problem 0



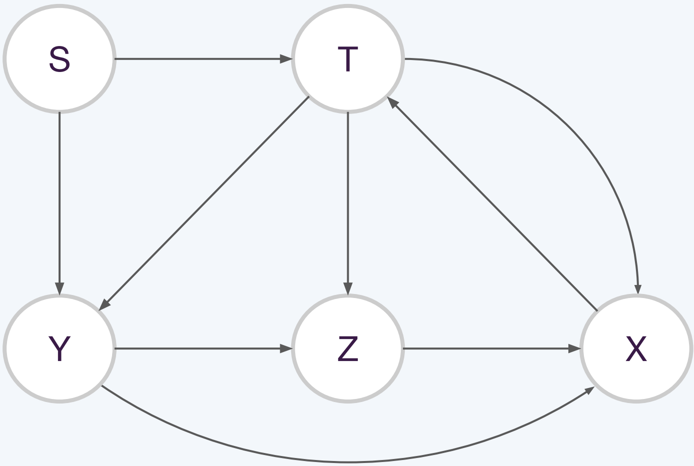
StackFrame (curr):

S	T	X	
		Y	Z

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

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

Problem 0

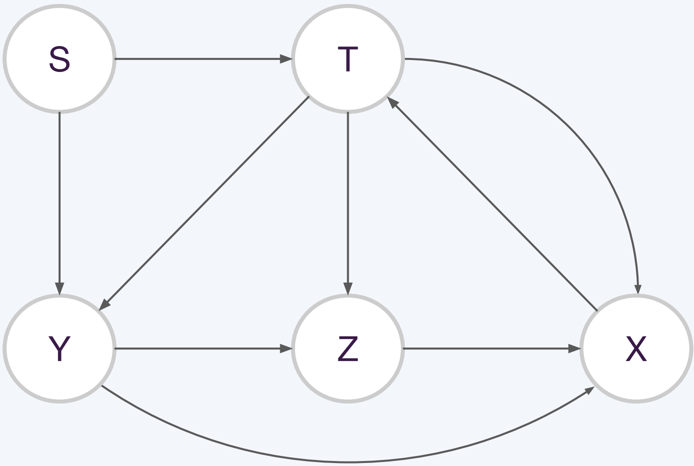


StackFrame (curr):

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

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

Problem 0



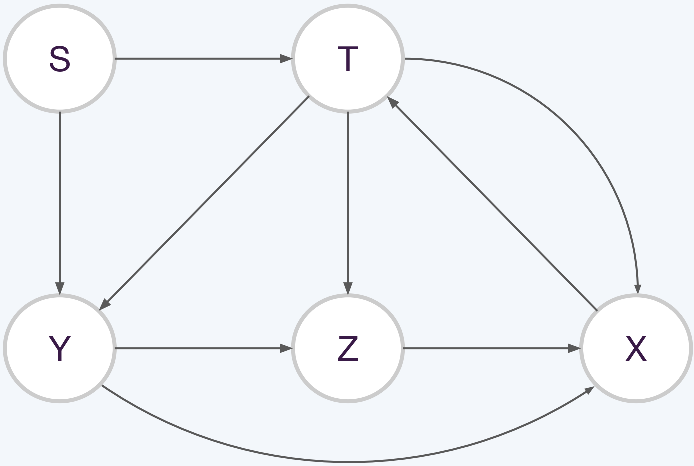
Vertex	Visited?	Done?
S	Yes	No
T	Yes	Yes
X	Yes	Yes
Y	Yes	Yes
Z	Yes	Yes

StackFrame (curr):

S	T	X	
		Y	Z

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

Problem 0



StackFrame (curr):

S	T	X	
		Y	Z

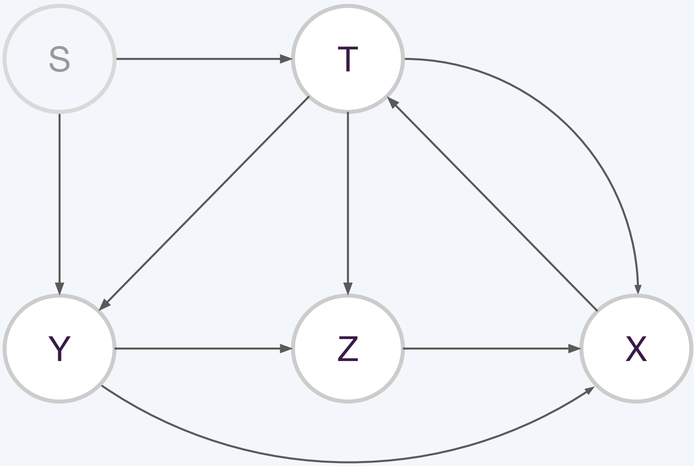
Vertex	Visited?	Done?
S	Yes	Yes
T	Yes	Yes
X	Yes	Yes
Y	Yes	Yes
Z	Yes	Yes

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

Problem 1

```
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 1



Queue:

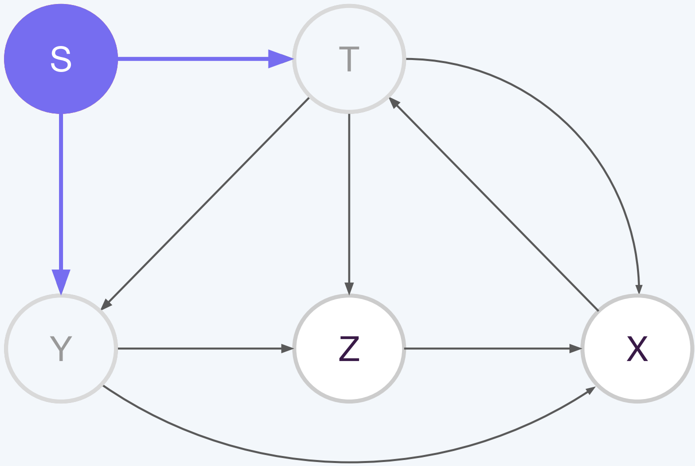
S				

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 1



Queue:

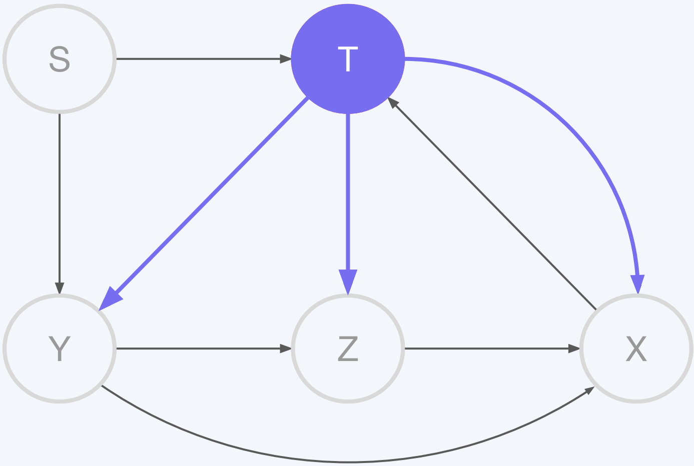
T	Y			

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 1

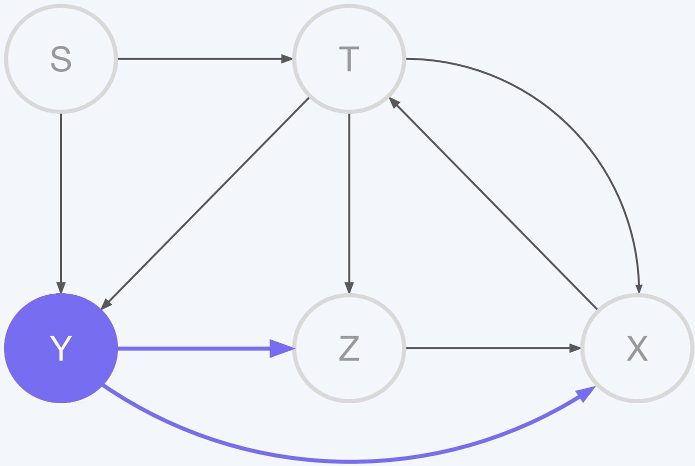


Queue:

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 1



Queue:

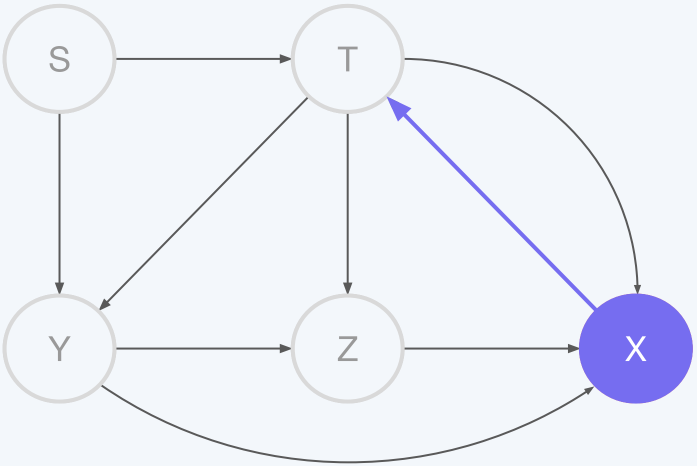
X	Z			

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 1



Queue:

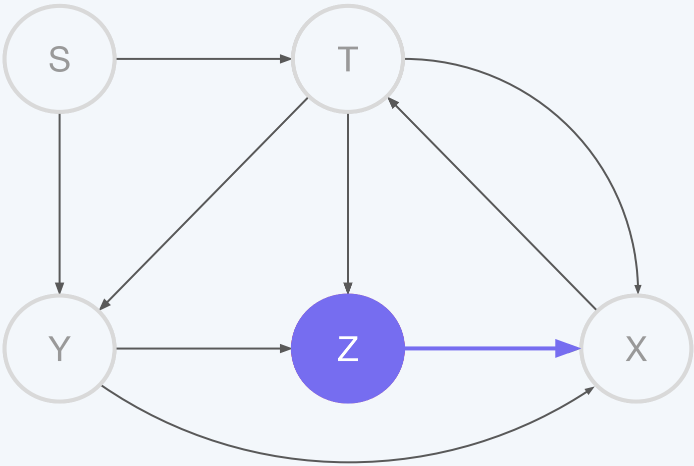
Z				

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 1



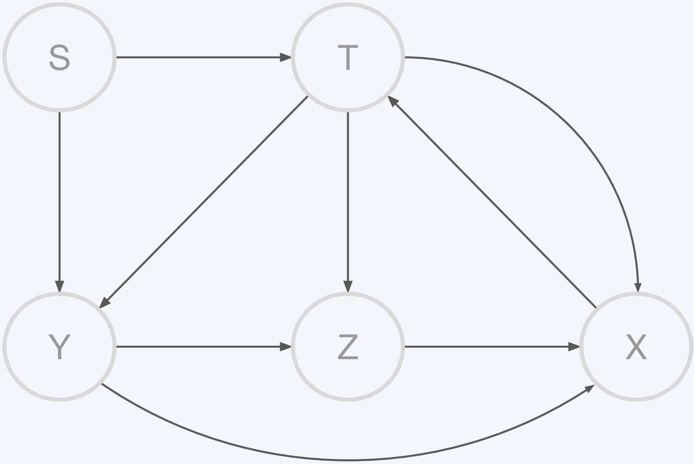
Queue:

front back

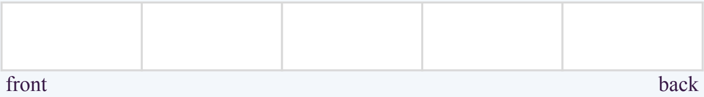
Vertex	Predecessor	Visited?
S	–	Yes
T	S	Yes
X	T	Yes
Y	S	Yes
Z	T	Yes

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

Problem 1



Queue:



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

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?

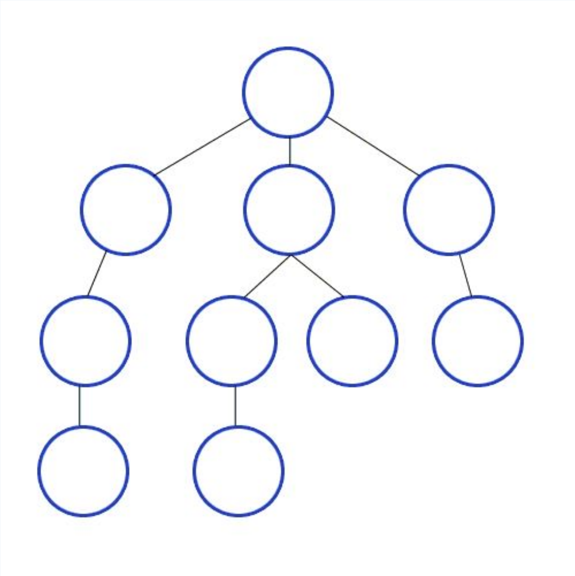
- Locate the target node in the table
- Backtrack through its predecessors until you reach the start node
- The sequence of predecessors form a path from the start to the target
- **Will be the shortest path by edge count (but not necessarily sum of edge costs)**

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

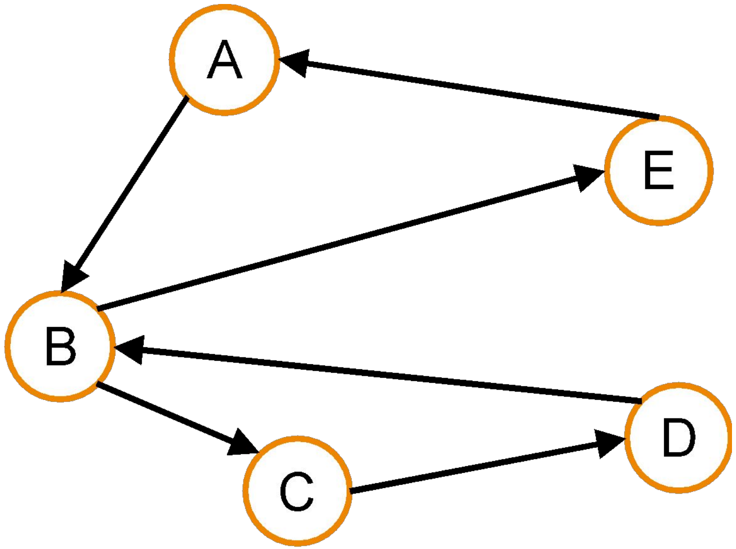
BFS/DFS Useful Properties

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 stack (earliest/shortest path)

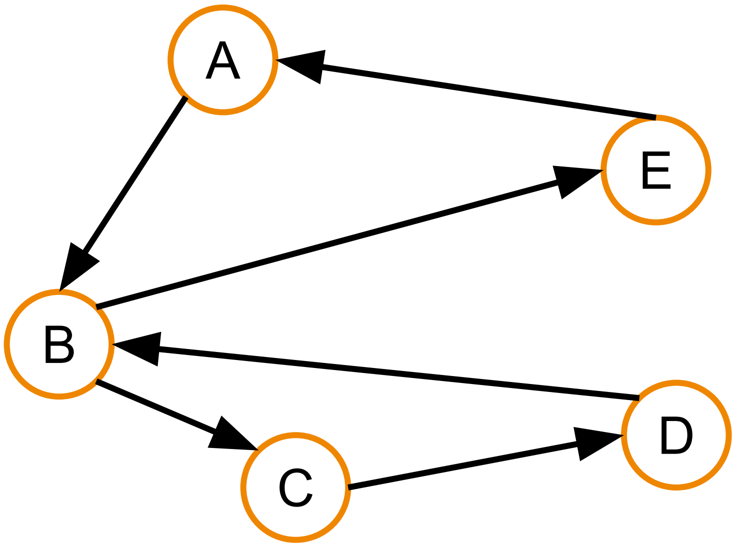


DFS – Detect Cycle



- 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 recursion stack.

DFS – Detect Cycle



Vertex: D
Mark as **done**

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

Vertex: B
Go to next unvisited

Vertex: A
Mark A as visited

Dijkstra's Algorithm
(Shortest Path)

Dijkstra's Algorithm

```
Dijkstra(Vertex source):
  for each vertex v:
    set v.cost = infinity
    mark v as unvisited

  set source.cost = 0

  while exist unvisited vertices:
    select unvisited vertex v with lowest cost
    mark v as visited

    for each edge (v, u) with weight w:
      if u is not visited:
        potentialBest = v.cost + w  // cost of best path to u through v
        currBest = u.cost           // cost of current best path to u

        if (potentialBest < currBest):
          u.cost = potentialBest
          u.pred = v
```

Dijkstra's algorithm finds the minimum-cost path from a source vertex to every other vertex in a **non-negatively weighted graph**

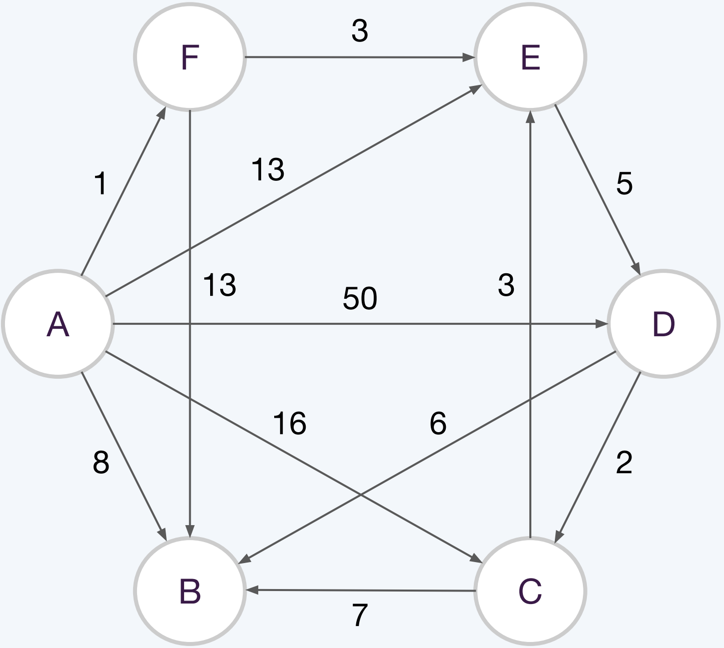
- $\mathcal{O}(|V| \log |V| + |E| \log |V|)$ runtime

Problem 2

```
Dijkstra(Vertex source):
  for each vertex v:
    set v.cost = infinity
    mark v as unvisited

  set source.cost = 0

  while exist unvisited vertices:
    select unvisited vertex v with lowest cost
    mark v as visited

    for each edge (v, u) with weight w:
      if u is not visited:
        potentialBest = v.cost + w
        currBest = u.cost

        if (potentialBest < currBest):
          u.cost = potentialBest
          u.pred = v
```

Problem 2

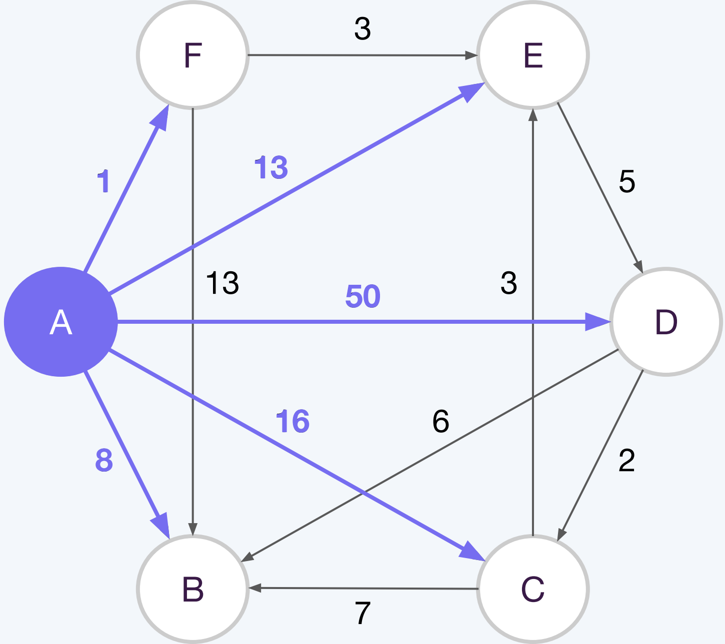
- Initialize each vertex as unvisited with cost ∞
- Set cost of source vertex A to 0



Vertex	Visited?	Cost	Predecessor
A	No	0	–
B	No	∞	–
C	No	∞	–
D	No	∞	–
E	No	∞	–
F	No	∞	–

Problem 2

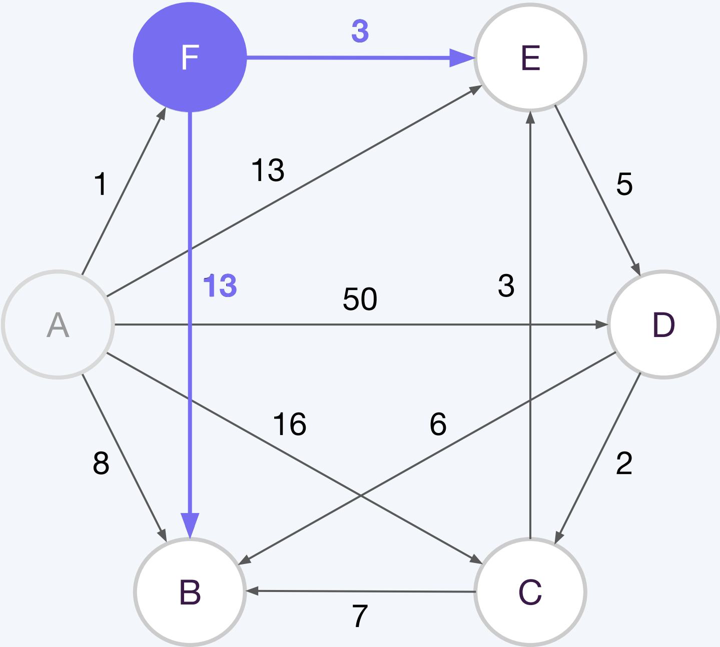
- Select unvisited vertex with lowest cost (A)
- Mark A as visited
- Process each outgoing edge



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	No	~~∞~~ 8	A
C	No	~~∞~~ 16	A
D	No	~~∞~~ 50	A
E	No	~~∞~~ 13	A
F	No	~~∞~~ 1	A

Problem 2

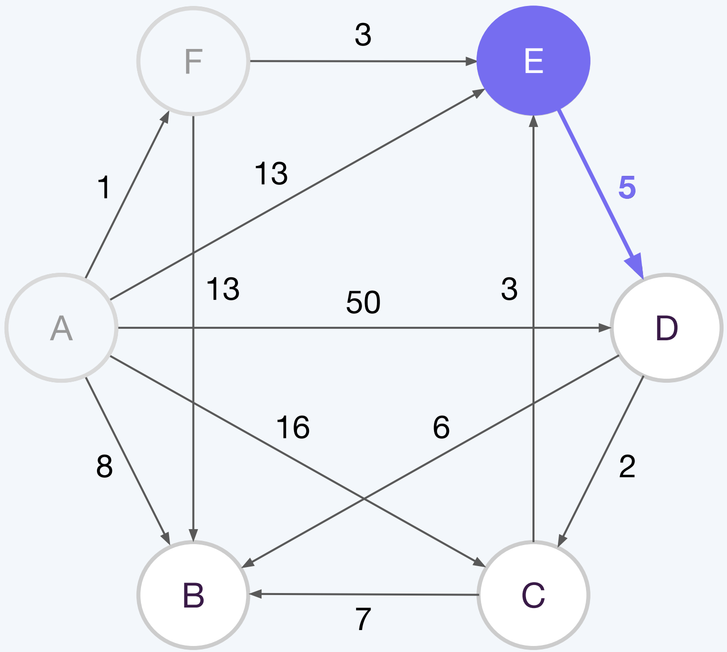
- Select unvisited vertex with lowest cost (F)
- Mark F as visited
- Process each outgoing edge



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	No	~~∞~~ 8	A
C	No	~~∞~~ 16	A
D	No	~~∞~~ 50	A
E	No	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 2

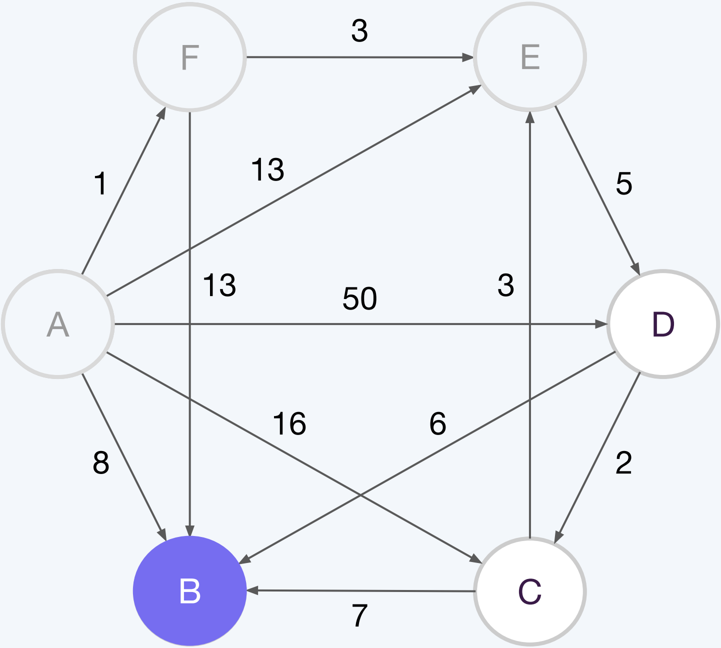
- Select unvisited vertex with lowest cost (E)
- Mark E as visited
- Process each outgoing edge



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	No	~~∞~~ 8	A
C	No	~~∞~~ 16	A
D	No	~~∞~~ ~~50~~ 9	~~A~~ E
E	Yes	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 2

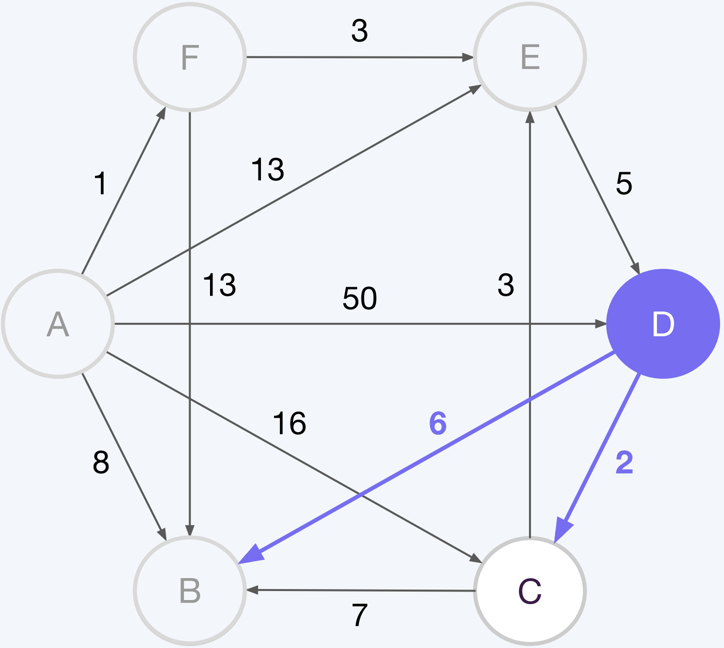
- Select unvisited vertex with lowest cost (B)
- Mark B as visited
- Process each outgoing edge
- No outgoing edges; continue



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	Yes	~~∞~~ 8	A
C	No	~~∞~~ 16	A
D	No	~~∞~~ ~~50~~ 9	~~A~~ E
E	Yes	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 2

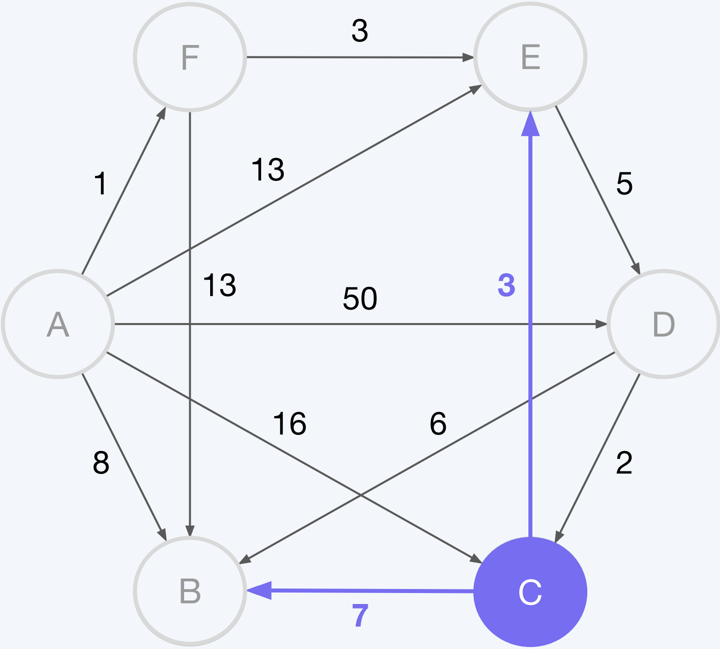
- Select unvisited vertex with lowest cost (D)
- Mark D as visited
- Process each outgoing edge
  (ignore D→B since B is already visited)



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	Yes	~~∞~~ 8	A
C	No	~~∞~~ ~~16~~ 11	~~A~~ D
D	Yes	~~∞~~ ~~50~~ 9	~~A~~ E
E	Yes	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 2

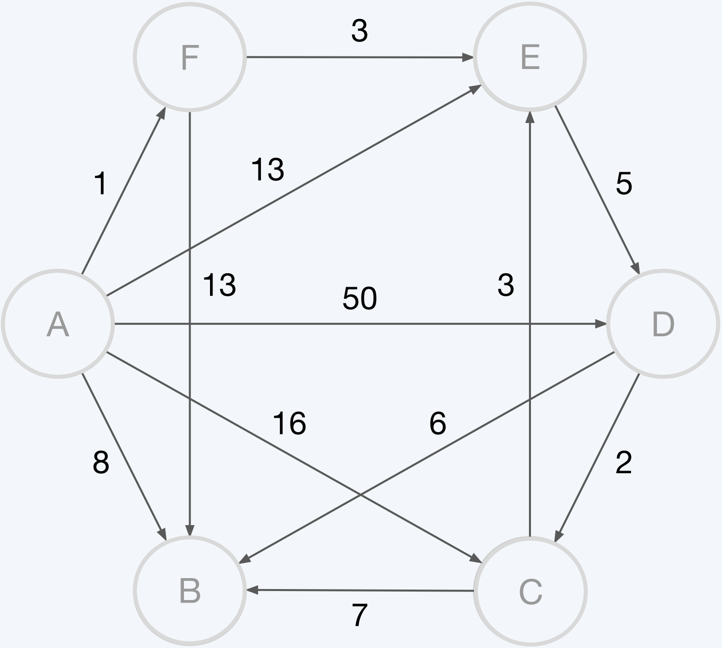
- Select unvisited vertex with lowest cost (C)
- Mark C as visited
- Process each outgoing edge
  (ignore C→B & C→E since B & E are already visited)
- No outgoing edges to unvisited nodes; continue



Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	Yes	~~∞~~ 8	A
C	Yes	~~∞~~ ~~16~~ 11	~~A~~ D
D	Yes	~~∞~~ ~~50~~ 9	~~A~~ E
E	Yes	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 2

- No more unvisited nodes; we are done



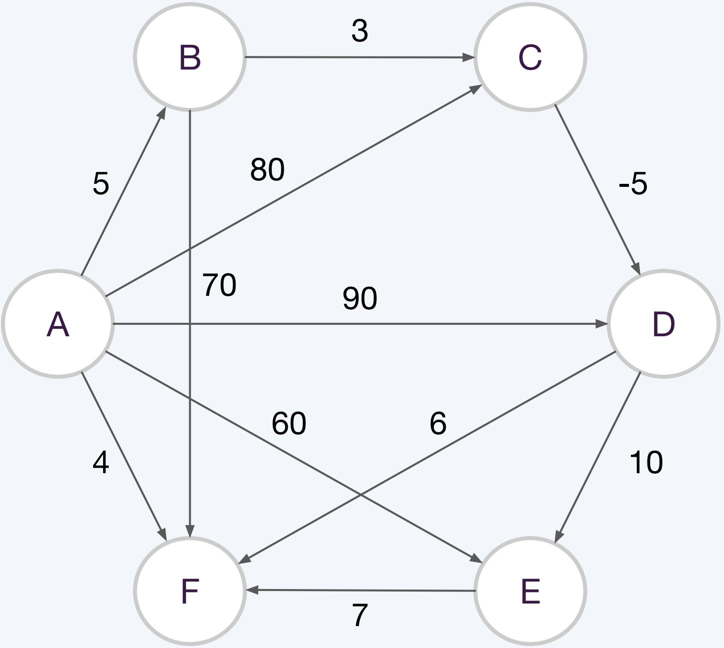
Vertex	Visited?	Cost	Predecessor
A	Yes	0	–
B	Yes	~~∞~~ 8	A
C	Yes	~~∞~~ ~~16~~ 11	~~A~~ D
D	Yes	~~∞~~ ~~50~~ 9	~~A~~ E
E	Yes	~~∞~~ ~~13~~ 4	~~A~~ F
F	Yes	~~∞~~ 1	A

Problem 3

```
Dijkstra(Vertex source):
  for each vertex v:
    set v.cost = infinity
    mark v as unvisited

  set source.cost = 0

  while exist unvisited vertices:
    select unvisited vertex v with lowest cost
    mark v as visited

    for each edge (v, u) with weight w:
      if u is not visited:
        potentialBest = v.cost + w
        currBest = u.cost

        if (potentialBest < currBest):
          u.cost = potentialBest
          u.pred = v
```

Problem 3

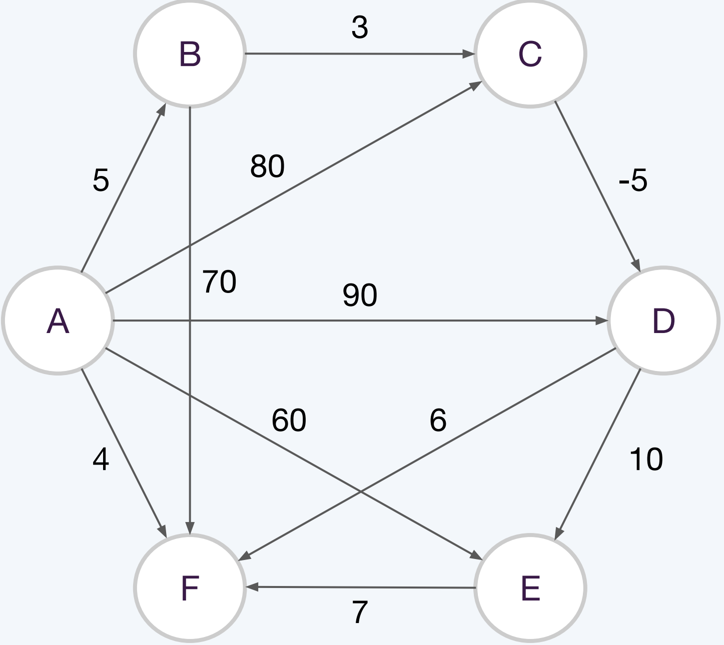
- Initialize each vertex as unvisited with cost ∞
- Set cost of source vertex A to 0



Vertex	Visited?	Cost	Predecessor
A	No	0	–
B	No	∞	–
C	No	∞	–
D	No	∞	–
E	No	∞	–
F	No	∞	–

Problem 3

- Initialize each vertex as unvisited with cost ∞
- Set cost of source vertex A to 0



Vertex	Visited?	Cost of Path	Pred
a	True	0	
b	True	~~∞~~ 05	~~a~~
c	True	~~∞~~ ~~80~~ 08	~~a~~ b
d	True	~~∞~~ ~~90~~ 03	~~a~~ c
e	True	~~∞~~ ~~60~~ 13	~~a~~ d
f	True	~~∞~~ 04	a

Order added to known set: a, f, b, c, d, e

Thank You!