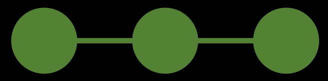
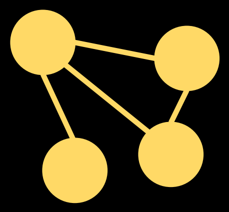
Graphs

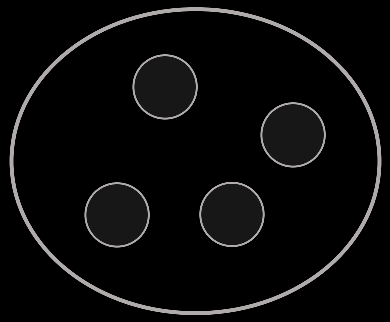
Data Structures & Algorithm

Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		

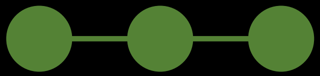


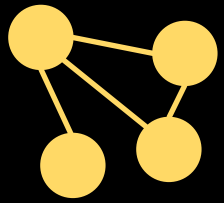


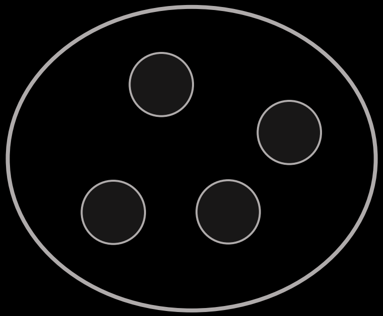


Categories of Data Structures

Linear Ordered	Non-linear Ordered	Not Ordered
Lists	Trees	Sets
Stacks	Graphs	Tables/Maps
Queues		





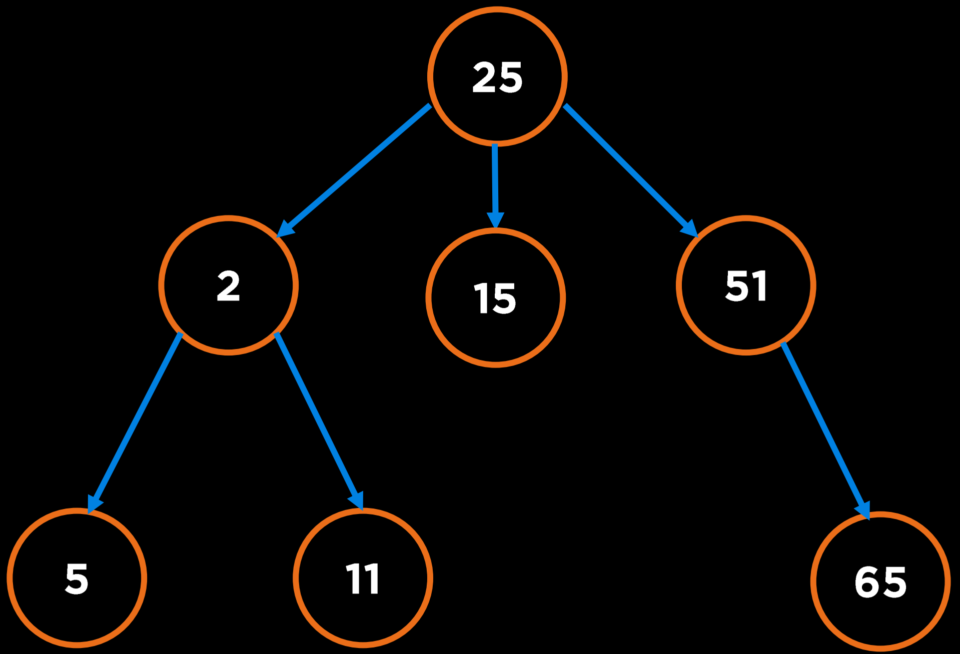


Agenda

- Graphs
  - Terminology
  - Types
  - Use cases
- Graph Implementations
  - Edge List
  - Adjacency Matrix
  - Adjacency List

Trees

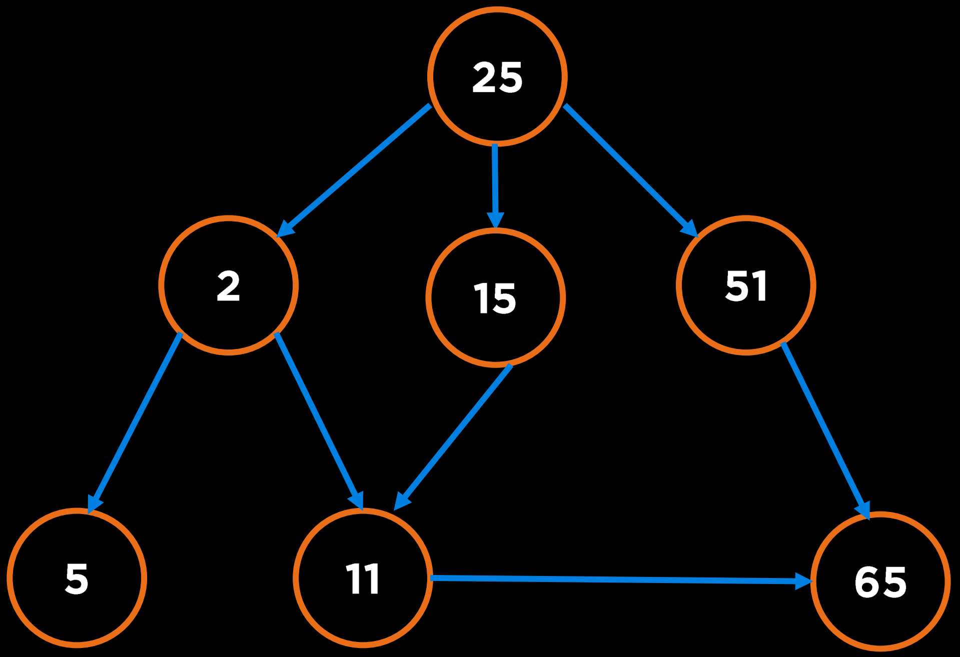
Hierarchical, Acyclic, and Exactly one path between two nodes



Graphs

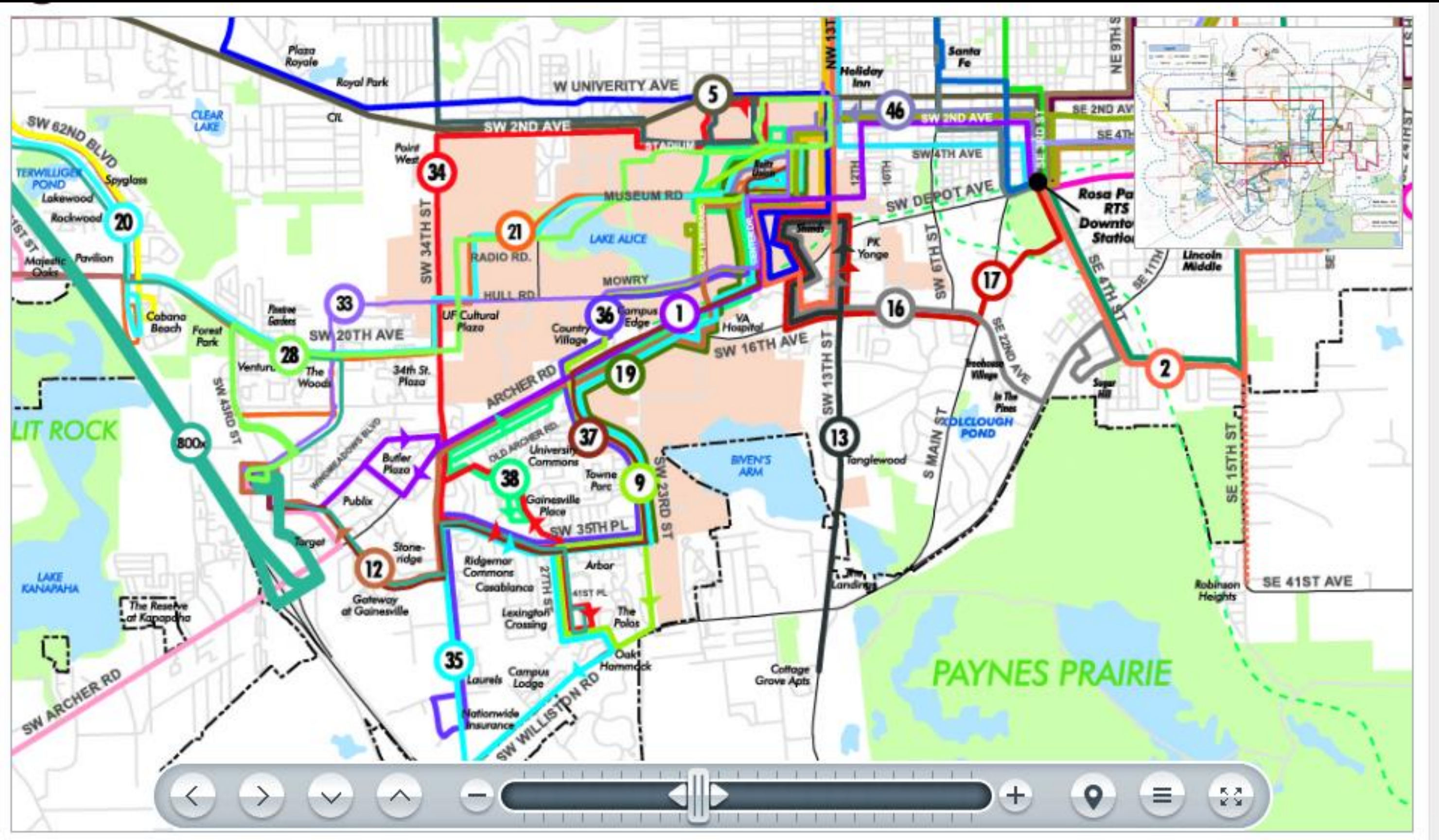
An ordered pair of a set of nodes and a set of edges.

$G = (V, E)$



Graphs

Example



Graph Terminology

Graph: Terminology

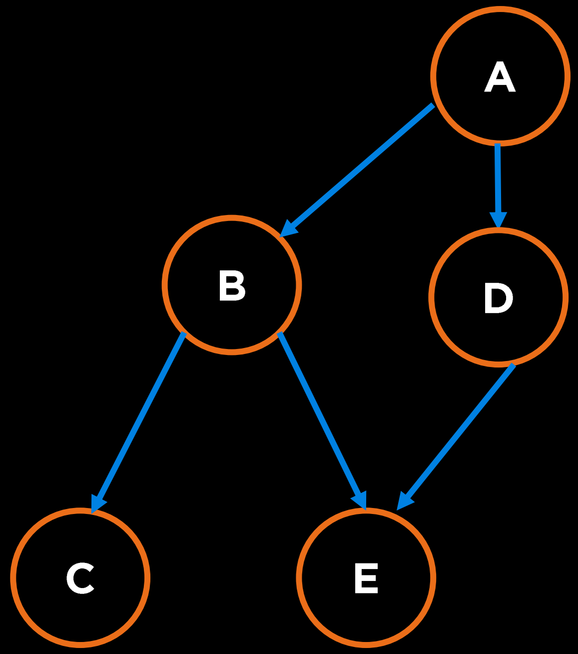
Vertex

Each node in a Graph is called a Vertex

V = {A, B, C, D, E}

|V| is the number of vertices in the graph

|V| = 5



Graph: Terminology

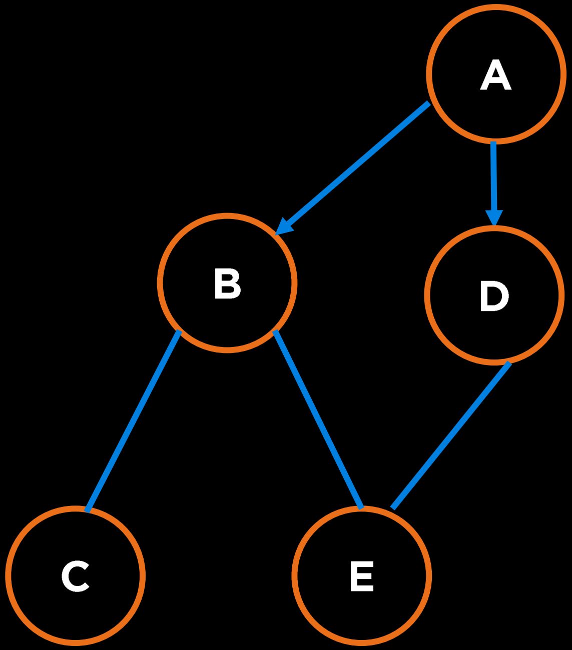
Edge

The connections between two nodes is called an edge.

E = {(A,B), (A,D), {B,C}, {B,E}, {D,E}}

|E| is the number of edges in the graph

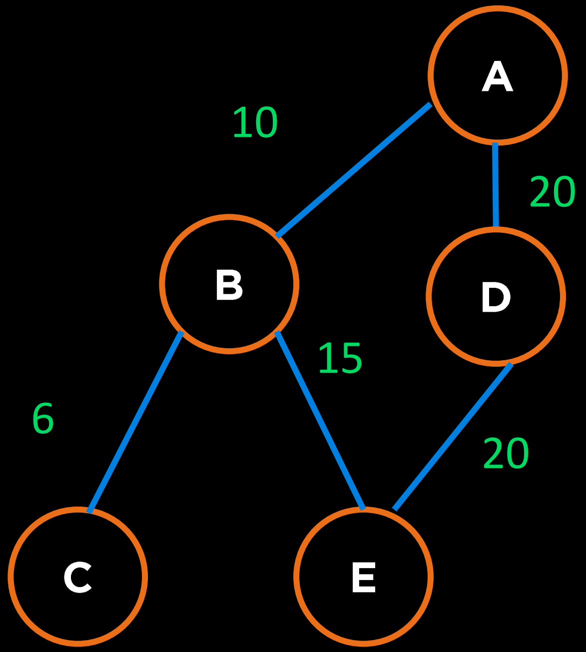
|E| = 5



Graph: Terminology

Weight

The edges in a graph may have associated values known as their weights. A weight is like a cost to travel from one vertex to the other over the edge.

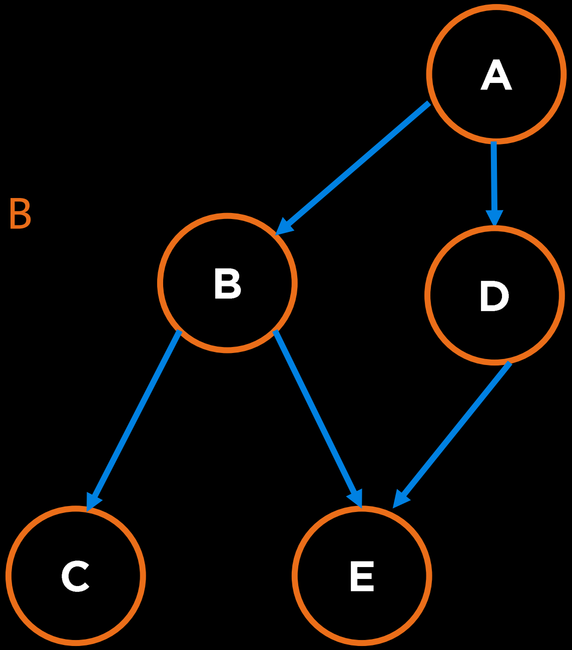


Graph: Terminology

Adjacent Vertices

A vertex is adjacent to another vertex if there is an edge to it from that other vertex.

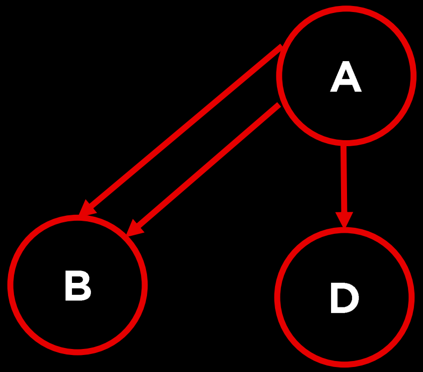
`B is adjacent to A but A is not adjacent to B`

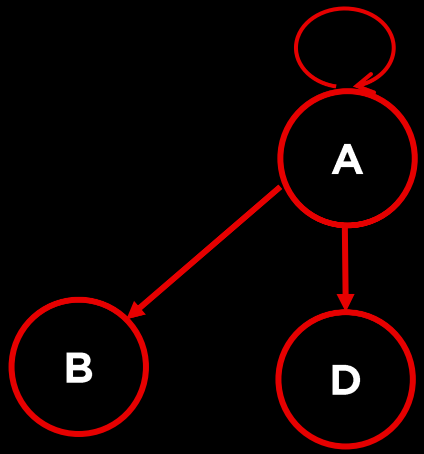


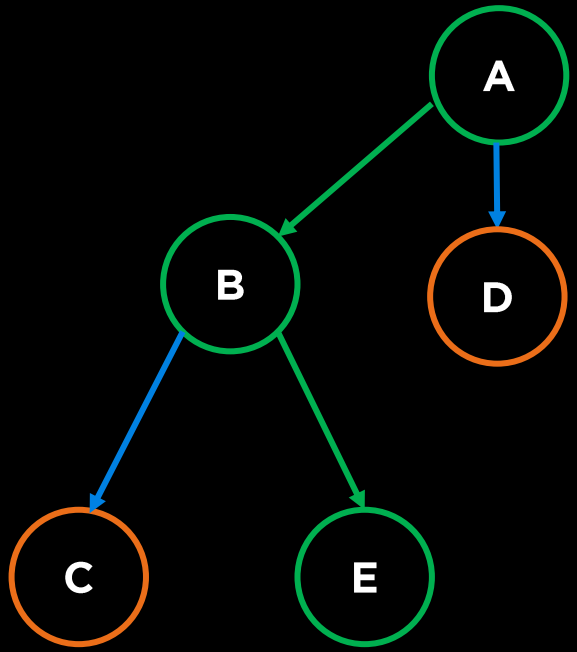
Graph: Terminology

Simple Graph

A simple graph is a graph with no edges that connect a vertex to itself, i.e. no “loops” and no two edges that connect the same vertices, i.e. no “parallel edges”.





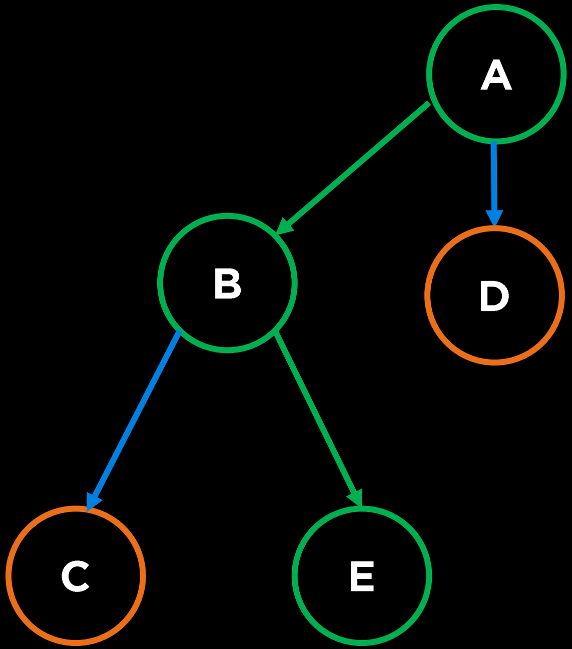


Graph: Terminology

Path

A path is a sequence of vertices in which each successive vertex is adjacent to its predecessor.

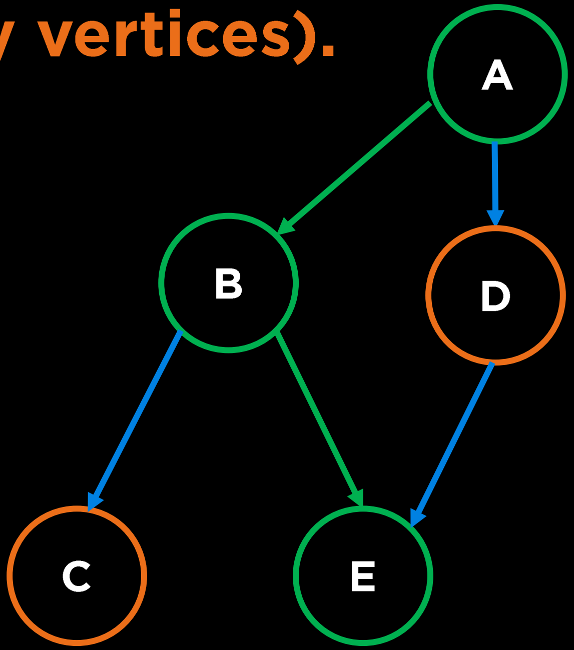
`Path from A to E: A, B, E`



Graph: Terminology

Simple Path

In a simple path, the vertices and edges are distinct except that the first and last vertex may be the same (no repeated intermediatory vertices).

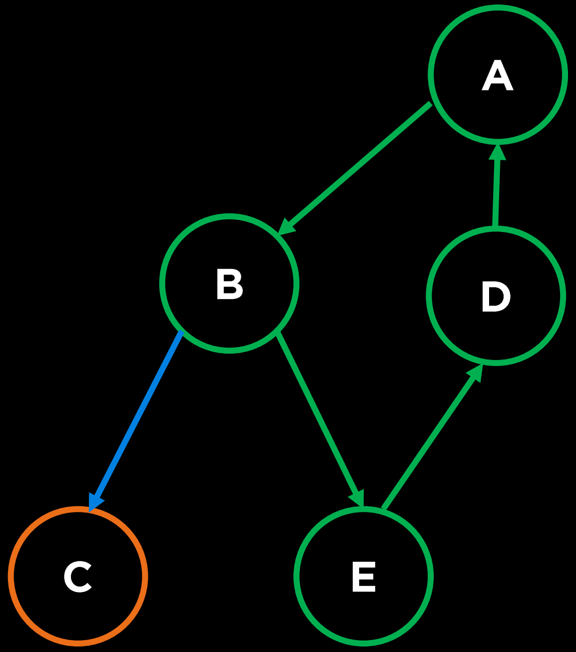


Graph: Terminology

Cycle

A cycle is a simple path in which only the first and final vertices are the same.

`A – B – E – D - A is a cycle.`

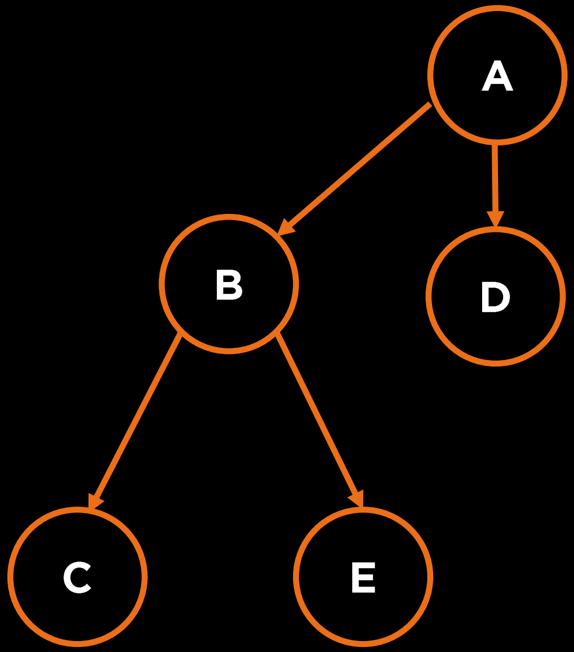


Graph: Terminology

Connected Vertex

Two vertices are connected if there is a path between them.

```
A and C are connected
D and C are not connected
```

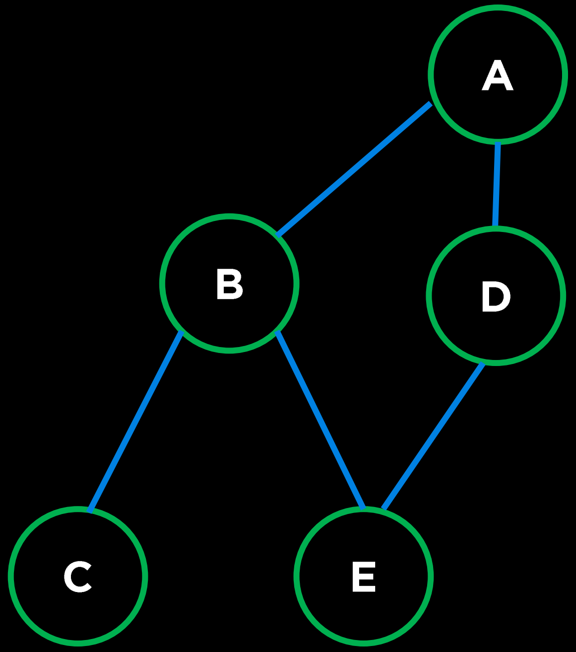


Graph: Terminology

Connected Graph

An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.

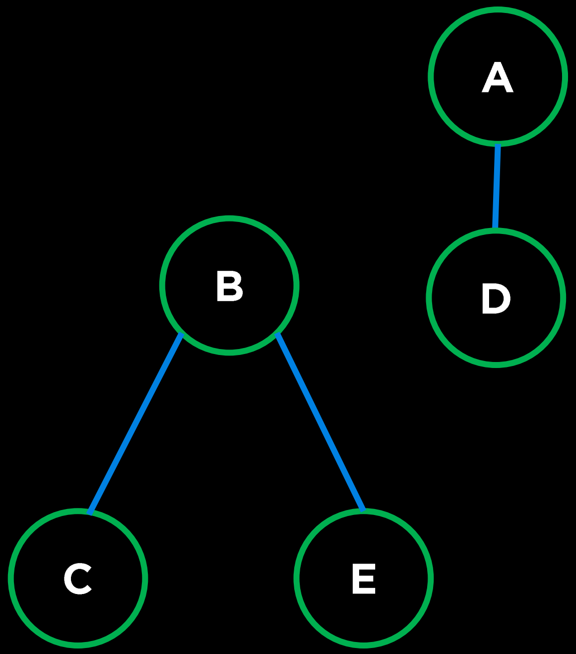
This is a connected graph



Graph: Terminology

Connected Graph

An undirected graph is called a connected graph if there is a path from every vertex to every other vertex.

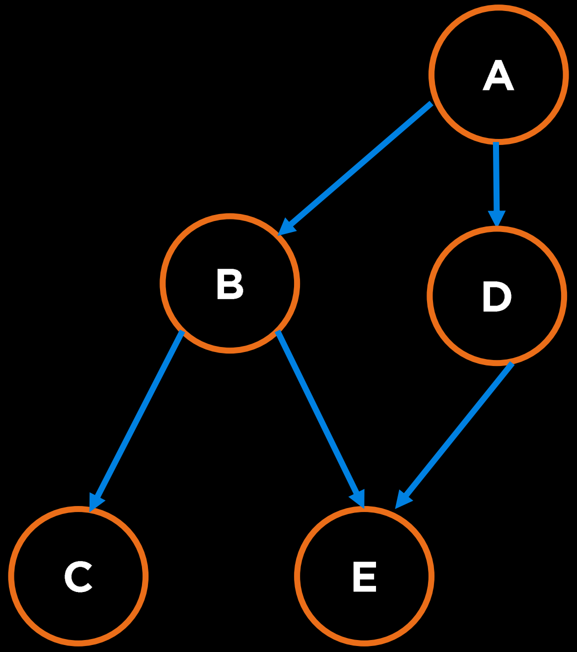


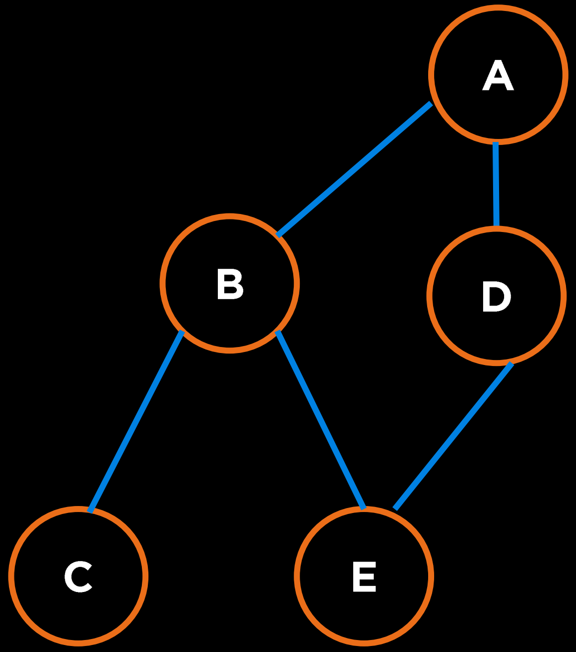

```
This is not a connected graph.

Connected components:
{A,D} and {B,C,E}
```

Graph Types

Graph Types

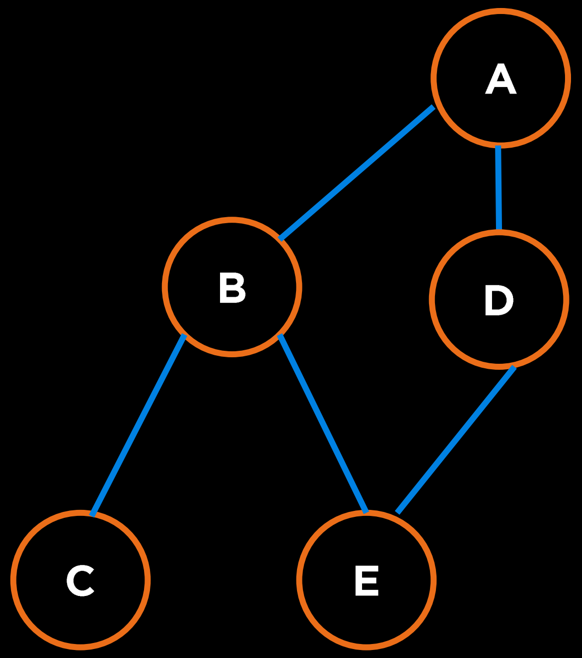
Directed (Digraph) vs Undirected

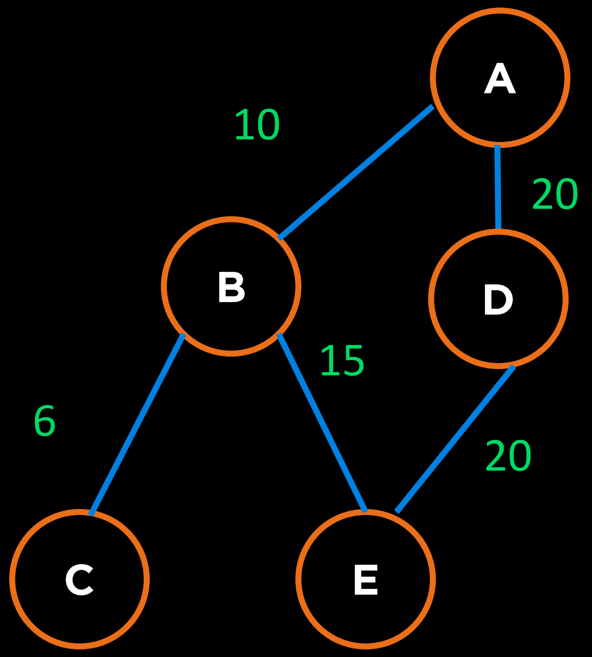




Graph Types

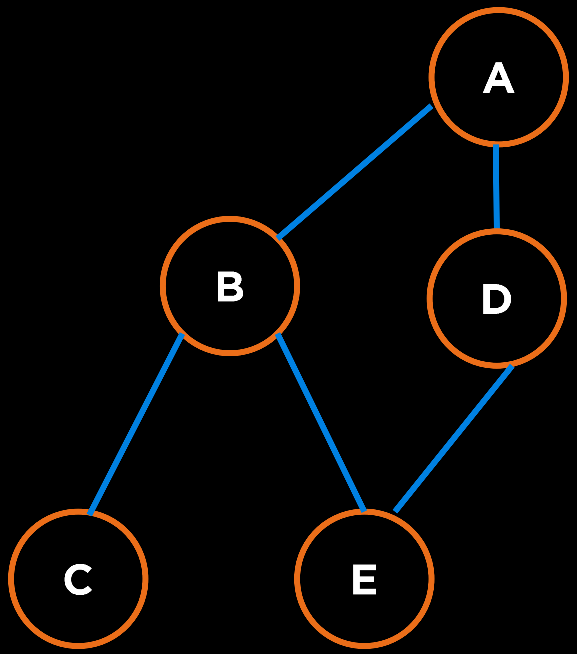
Weighted vs Unweighted

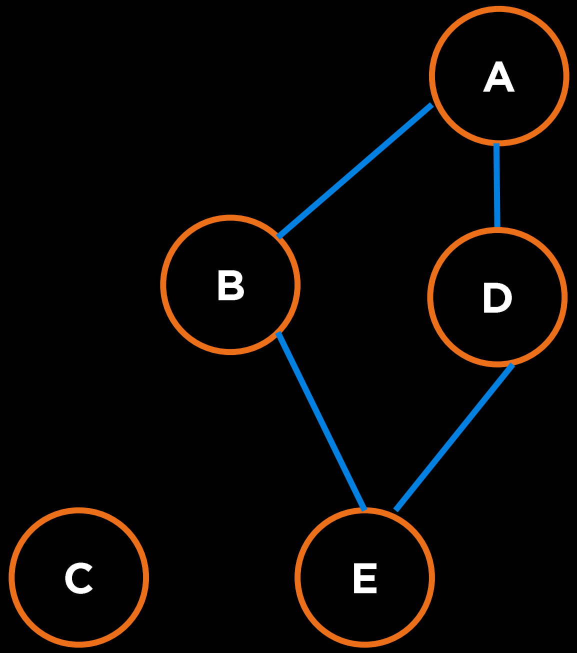




Graph Types

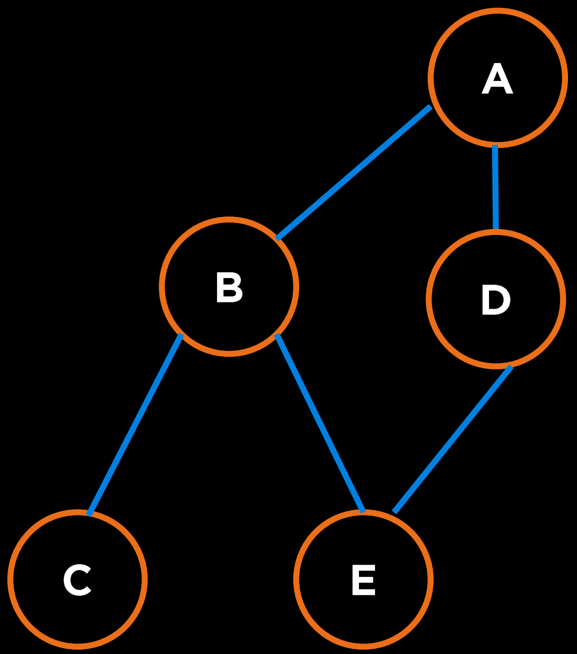
Connected vs Unconnected



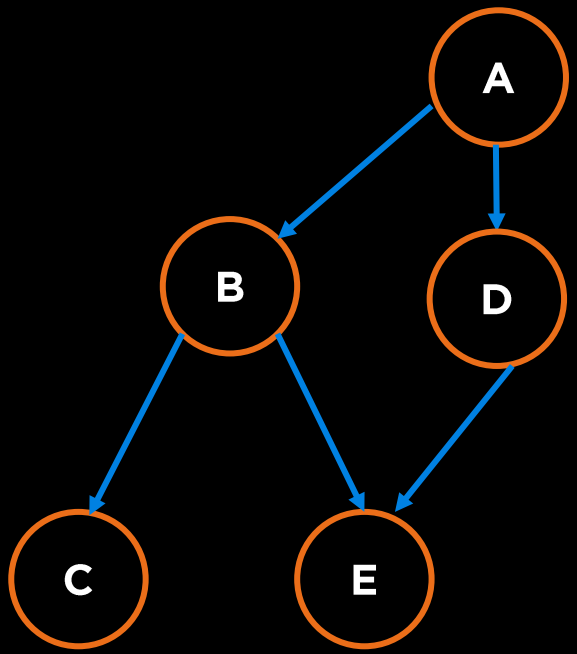


Graph Types

Cyclic vs Acyclic



Cyclic

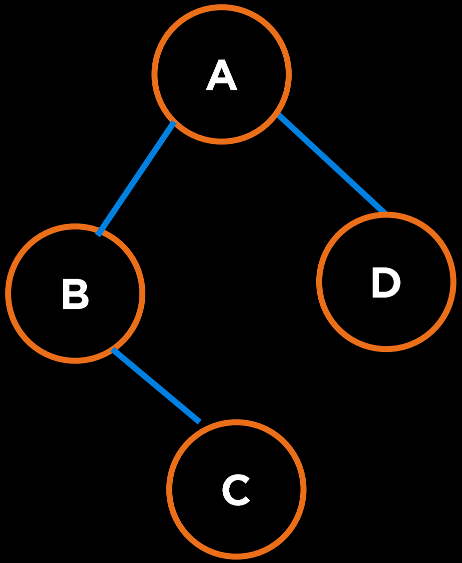


Acyclic

Graph Types

Dense vs Sparse

- The density of a graph is the ratio of $|E|$ to $|V|^2$
- We can assume that $|E|$ is
  - ~ $|V|^2$ for a dense graph [Density ~ 1]
  - ~ $|V|$ for a sparse graph [Density ~ 0]



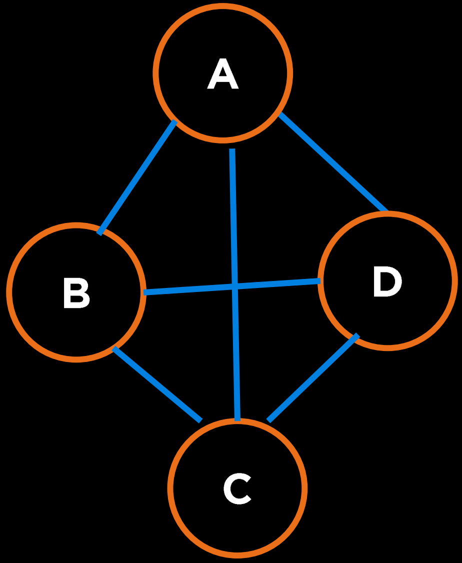
Sparse

Directed Graphs:

$$0 <= |E| <= |V|(|V|-1)$$

Undirected Graphs:

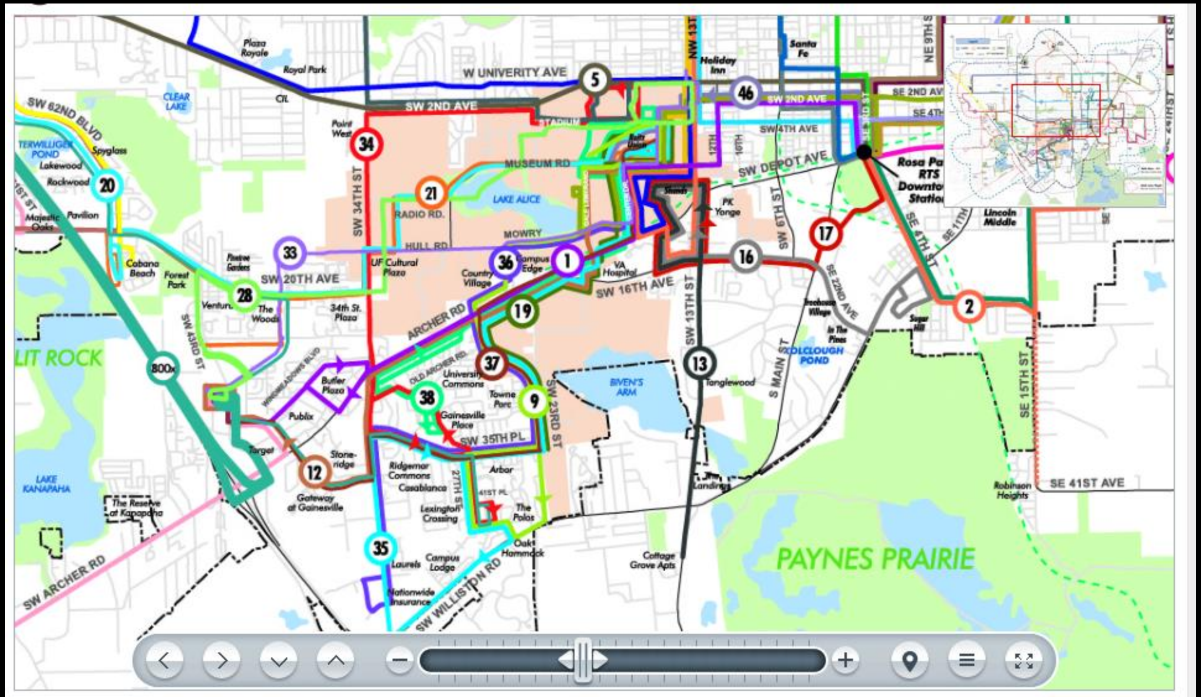
$$0 <= |E| <= |V|(|V|-1)/2$$



Dense

Graphs

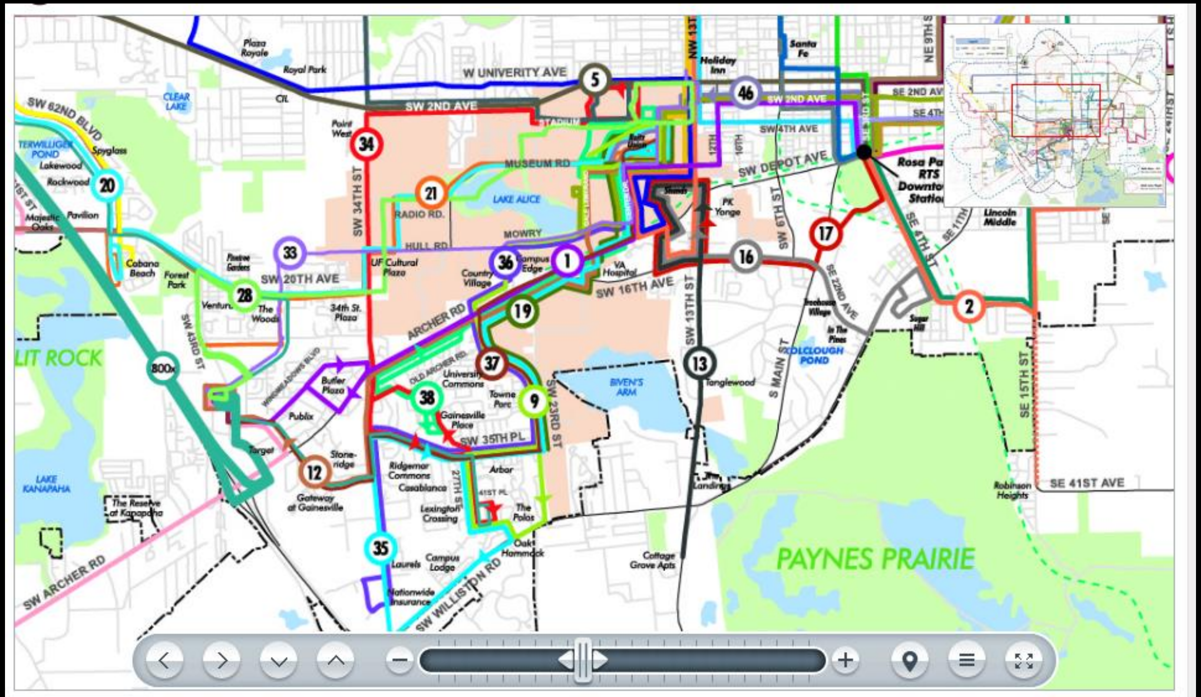
Example



Undirected
Directed
Cyclic
Connected

Graphs

Example



Undirected
Directed
Cyclic
Connected

Graph Problems

Common Examples:

- Social Networks
- World Wide Web
- Maps

Weighted? Directed?

Graph Problems

Common Examples:

- Social Networks (Unweighted, Undirected)
- World Wide Web (Unweighted, Directed)
- Maps (Weighted, Undirected)

Graph Problems

There are lots of interesting questions we can ask about a graph:

- What is the shortest route from S to T? What is the longest without cycles?
- Are there cycles?
- Is there a tour you can take that only uses each node (station) exactly once?
- Is there a tour that uses each edge exactly once?

Graph Problems

Some well-known graph problems and their common names:

- s-t Path. Is there a path between vertices s and t?
- Connectivity. Is the graph connected, i.e. is there a path between all vertices?
- Biconnectivity. Is there a vertex whose removal disconnects the graph?
- Shortest s-t Path. What is the shortest path between vertices s and t?
- Cycle Detection. Does the graph contain any cycles?
- Euler Tour. Is there a cycle that uses every edge exactly once?
- Hamilton Tour. Is there a cycle that uses every vertex exactly once?
- Planarity. Can you draw the graph on paper with no crossing edges?
- Isomorphism. Are two graphs isomorphic (the same graph in disguise)?

Often can't tell how difficult a graph problem is without very deep consideration.

Questions

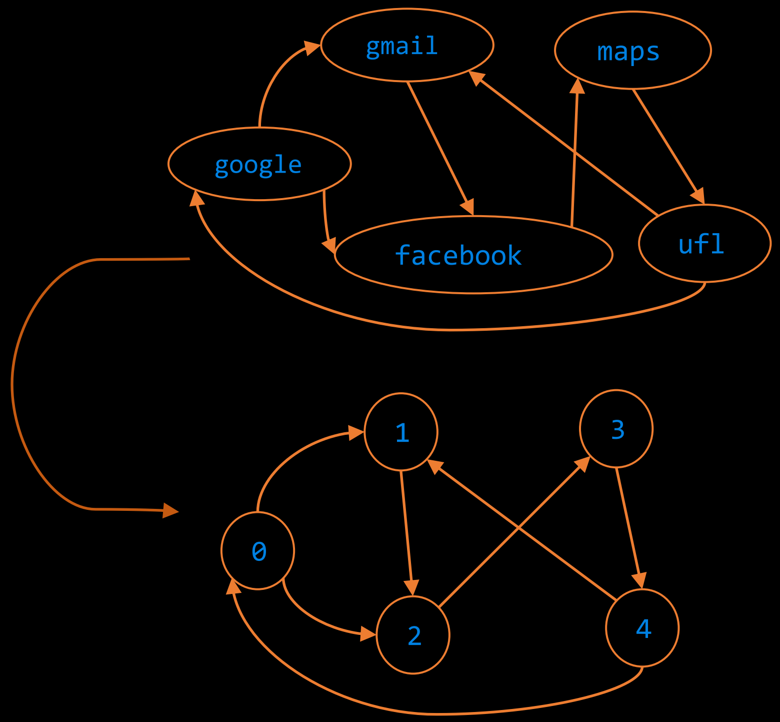
Graph Implementations

Graph API

- No common ADT for Graphs
- Graphs were present before Object Oriented Programming
- API must include Graph methods, including their signatures and behaviors
- Defines how Graph client programmers must think.
- An underlying data structure to represent our graphs.
- Our choices can have profound implications on:
  - Runtime
  - Memory usage
  - Difficulty of implementing various graph algorithms

Common Convention

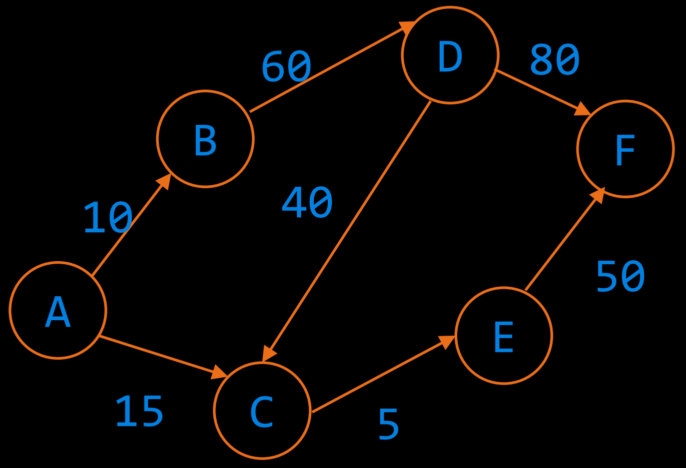
- Map labels to numbers, e.g. If node is called “google.com”, assign it a number, say 0.
- Use a map data structure to achieve this: map<string, int>



Label	Graph_Index
google.com	0
gmail.com	1
facebook.com	2
maps.com	3
ufl.edu	4

```
std::string vertex = "ufl.edu";
int count = 0;

std::map<std::string, int> graph;

if(graph.find(vertex) == graph.end())
        graph[vertex] = count++;
```

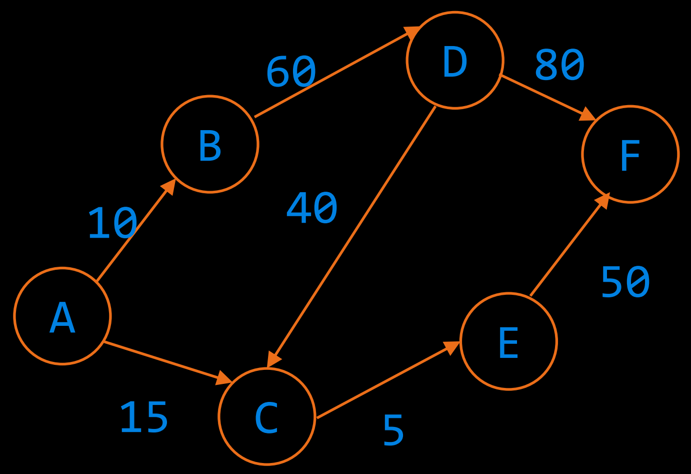
Common Operations



G

- Connectedness
- Neighborhood or Adjacency

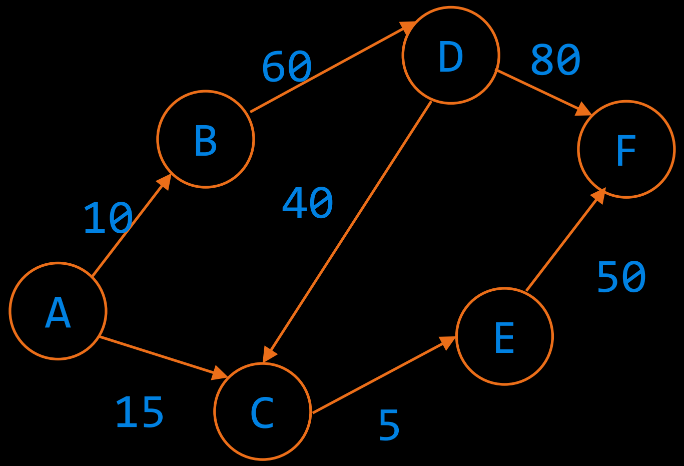
Common Representations



G

- Edge List
- Adjacency Matrix
- Adjacency List

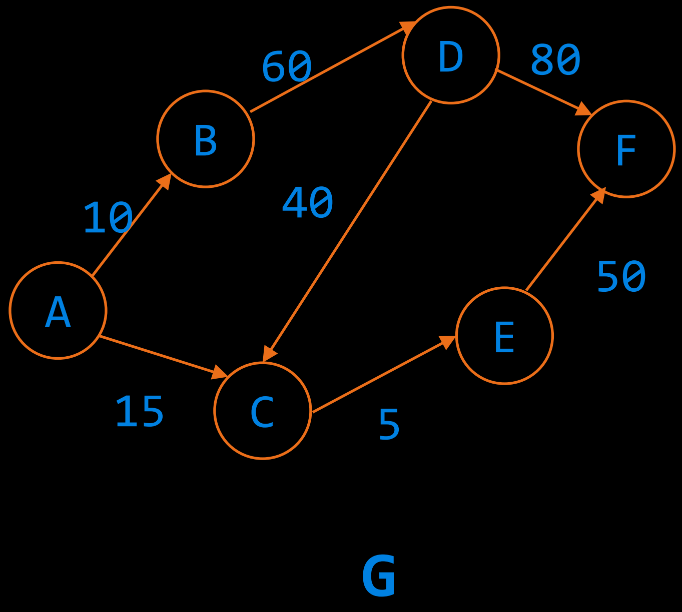
Edge List



G

G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

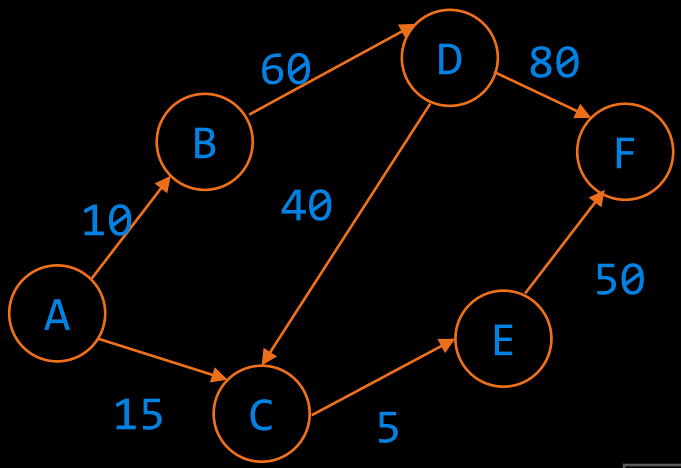
Edge List



A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

Edge List



G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

G

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

Common Operations:

1. Connectedness

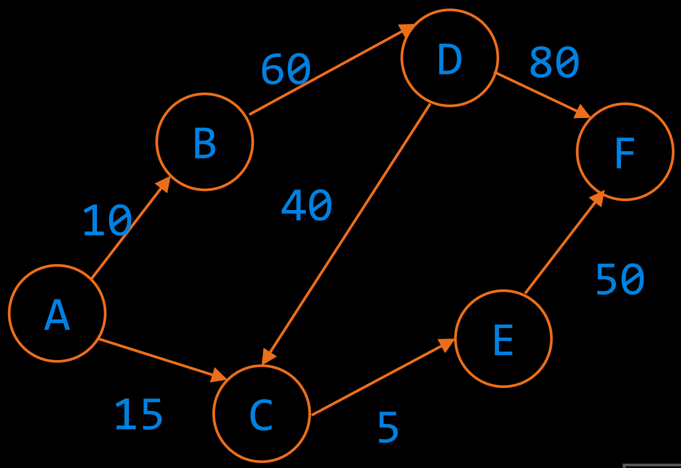
   Is A connected to B?

2. Adjacency

   What are A’s adjacent nodes?

Space: ?

Edge List



G = {(A,B), (A,C), (B,D), (D,C), (D,F), (E,F), (C,E)}

G

A	B	10
A	C	15
B	D	60
D	C	40
D	F	80
E	F	50
C	E	5

Common Operations:

1. Connectedness

Is A connected to B?

~ O(E)

2. Adjacency

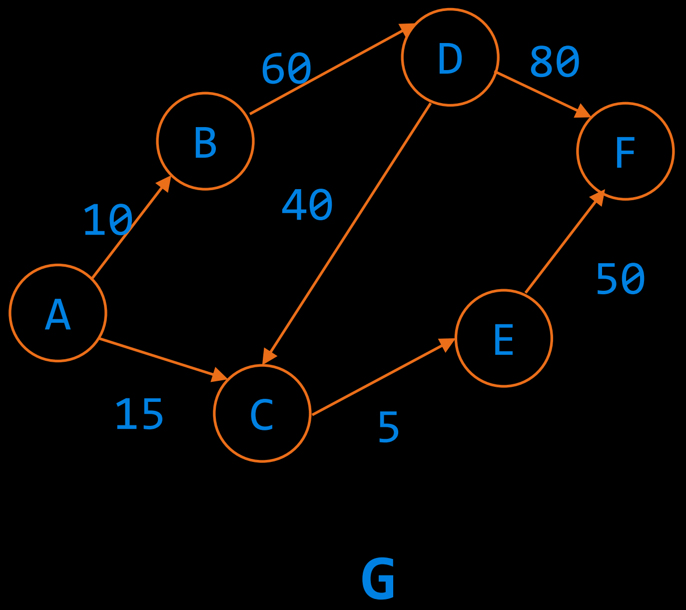
What are A's adjacent nodes?

~ O(E)

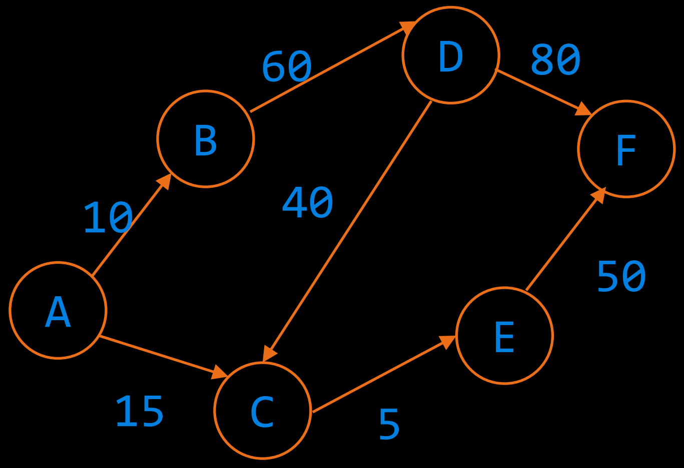
O(|E|) ~ O(|V| * |V|)

Space: O(E)

Adjacency Matrix



Adjacency Matrix

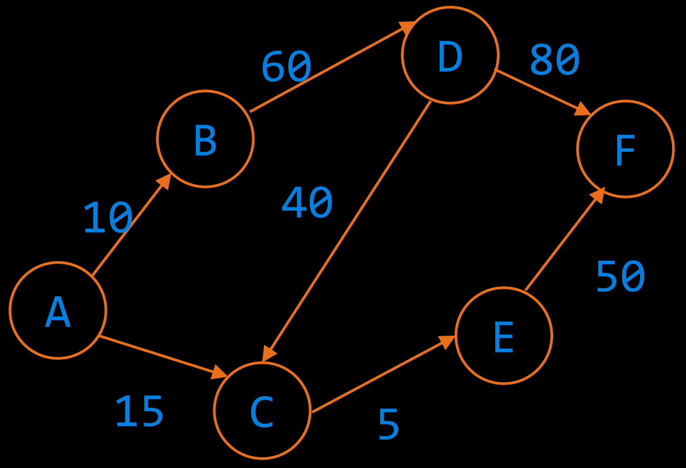


G

Insertion:

```
G[from][to] = weight; (if there is an edge, “from” -> “to”)
G[from][to] = 0;            (otherwise)
```

Adjacency Matrix



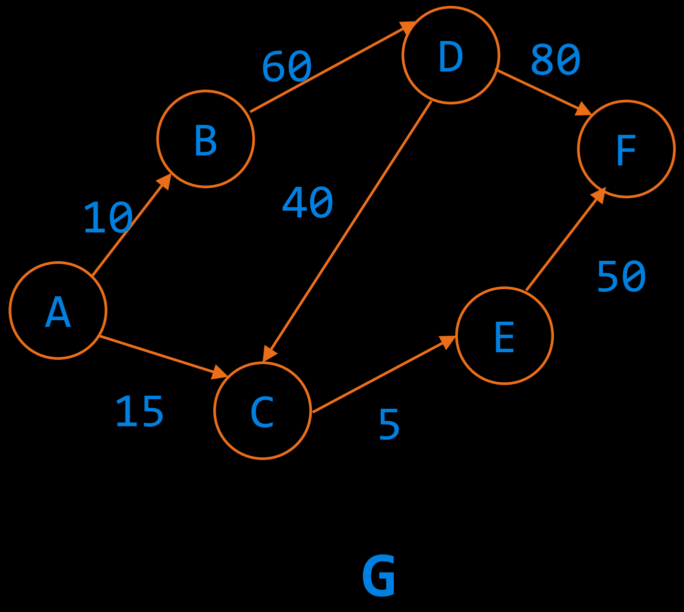
G

	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

Insertion:

```
G[from][to] = weight; (if there is an edge, “from” -> “to”)
G[from][to] = 0;           (otherwise)
```

Adjacency Matrix

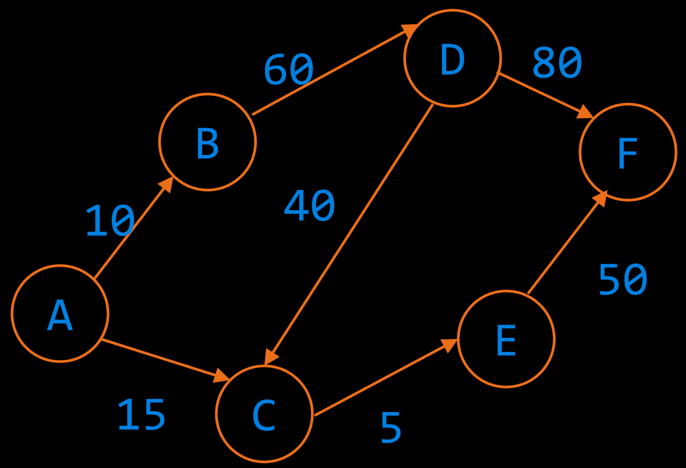


	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

Insertion:

```
G[from][to] = weight; (if there is an edge, “from” -> “to”)
G[from][to] = 0;            (otherwise)
```

Adjacency Matrix Implementation



Insertion:

```
G[from][to] = weight; (if there is an edge, "from" -> "to")
G[from][to] = 0;          (otherwise)
```

Input

7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50

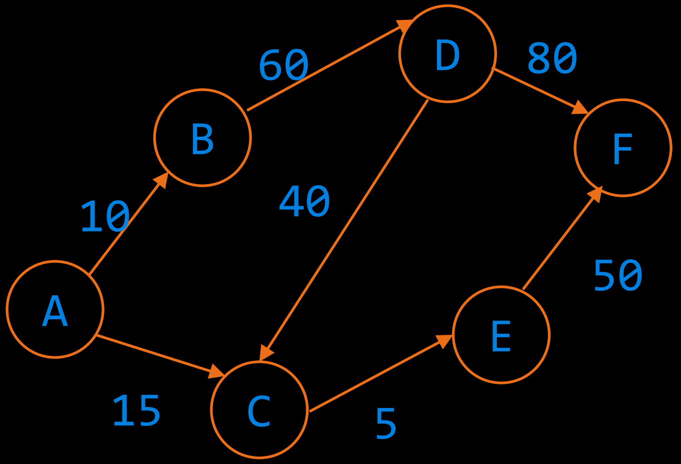
G

Map
A 0
B 1
C 2
D 3
E 4
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

https://www.onlinegdb.com/Hy8M0CnsS

Adjacency Matrix Implementation



Insertion:

G[from][to] = weight; (if there is an edge, “from” -> “to”)

G[from][to] = 0; (otherwise)

```
#include <iostream>
#include<map>
#define VERTICES 6
using namespace std;
int main()
{
        int no_lines, wt, j=0;
        string from, to;
        int graph [VERTICES][VERTICES] = {0};
        map<string, int> mapper;
        cin >> no_lines;
        return 0;
}
```

Input

7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50

G

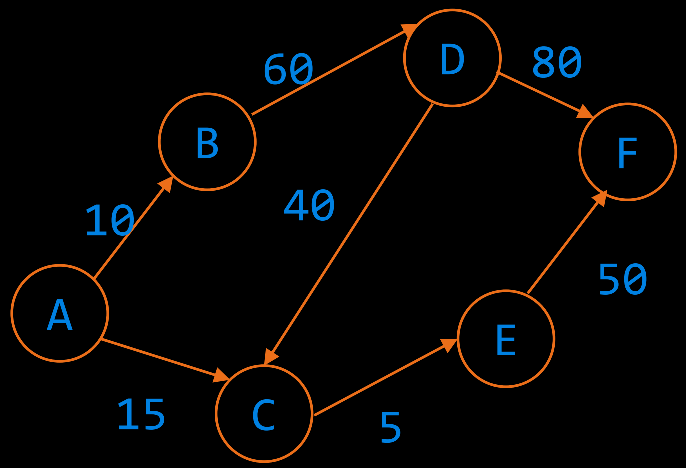
Map

A	0
B	1
C	2
D	3
E	4
F	5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

https://www.onlinegdb.com/Hy8M0CnsS

Adjacency Matrix Implementation



Insertion:

G[from][to] = weight; (if there is an edge, “from” -> “to”)

G[from][to] = 0; (otherwise)

```
#include <iostream>
#include<map>
#define VERTICES 6
using namespace std;
int main()
{
        int no_lines, wt, j=0;
        string from, to;
        int graph [VERTICES][VERTICES] = {0};
        map<string, int> mapper;
        cin >> no_lines;
        for(int i = 0; i < no_lines; i++)
        {
                cin >> from >> to >> wt;
                if (mapper.find(from) == mapper.end())
                        mapper[from] = j++;
                if (mapper.find(to) == mapper.end())
                        mapper[to] = j++;
                graph[mapper[from]][mapper[to]] = wt;
        }
        return 0;
}
```

Input

7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50

G

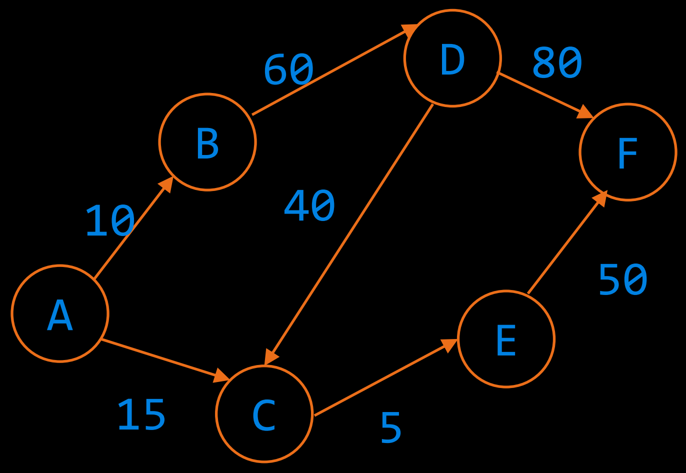
Map

A 0
B 1
C 2
D 3
E 4
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

https://www.onlinegdb.com/Hy8M0CnsS

Adjacency Matrix



G

Map
A 0
B 1
C 2
D 3
E 4
F 5

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Common Operations:

1. Connectedness

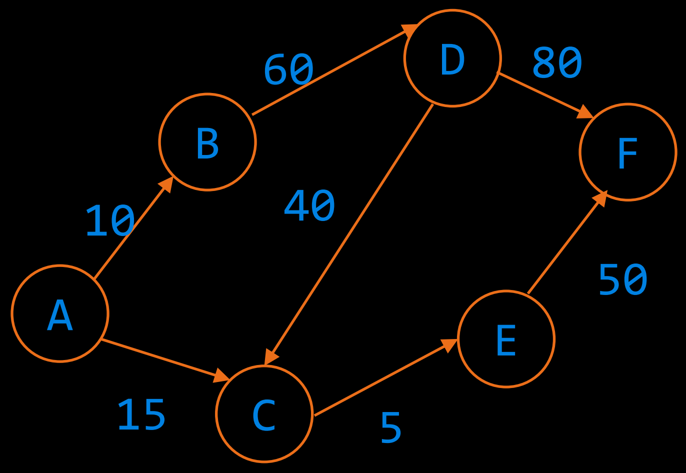
   Is A connected to B?

2. Adjacency

   What are A’s adjacent nodes?

Space: ?

Adjacency Matrix



G

	0	1	2	3	4	5
0	0	10	15	0	0	0
1	0	0	0	60	0	0
2	0	0	0	0	5	0
3	0	0	40	0	0	80
4	0	0	0	0	0	50
5	0	0	0	0	0	0

Map
A 0
B 1
C 2
D 3
E 4
F 5

Common Operations:

1. Connectedness

Is A connected to B?

G[“A”][“B”] ~ O(1)

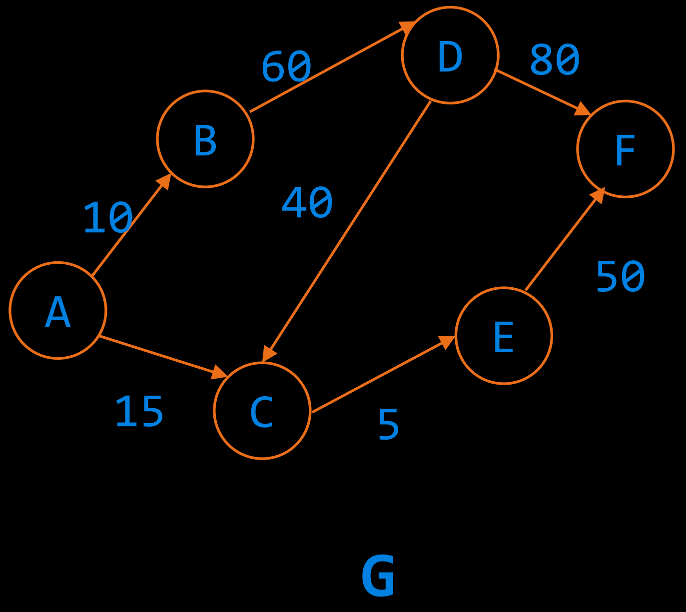
2. Adjacency

What are A’s adjacent nodes?

```
for each element x in G[“A”]
    if x ! = 0
                               ~ O(|V|)
```

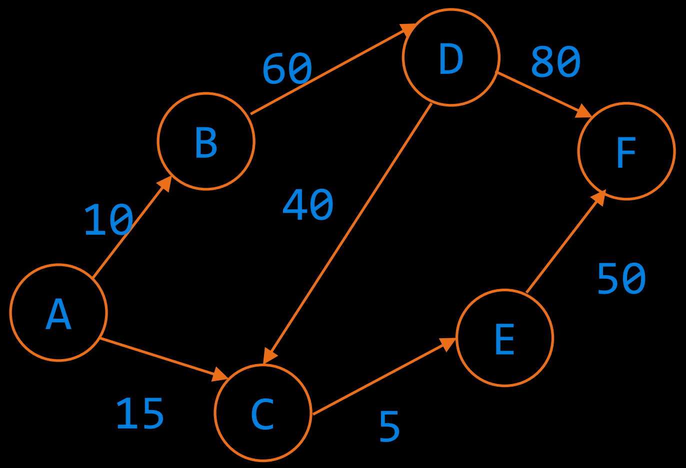
Space: O(|V| * |V|)

Adjacency Matrix Problem



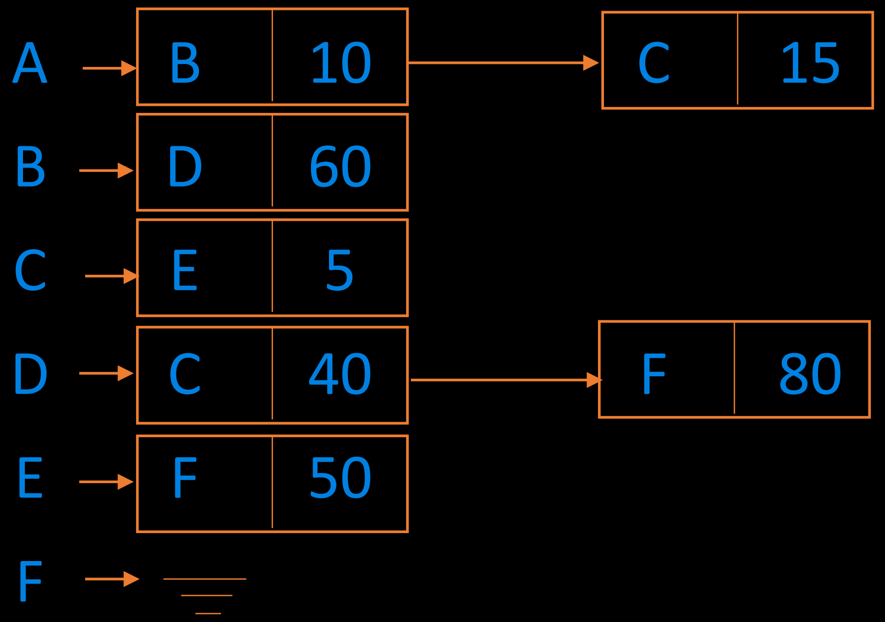
	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0

Adjacency List

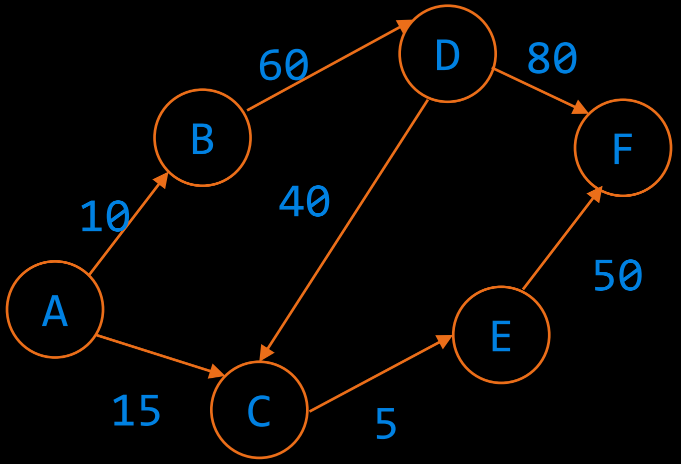


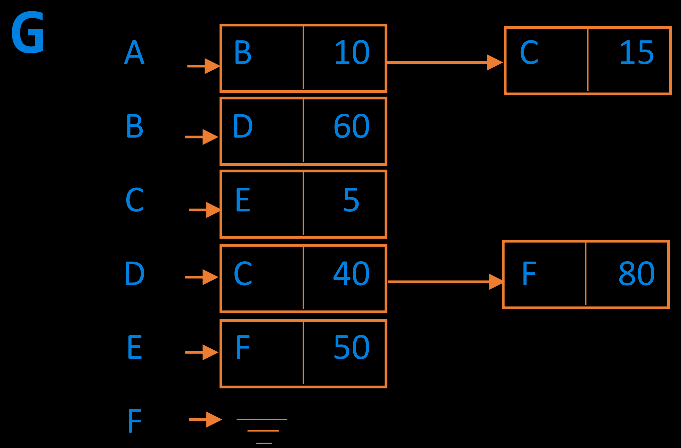
G

	A	B	C	D	E	F
A	0	10	15	0	0	0
B	0	0	0	60	0	0
C	0	0	0	0	5	0
D	0	0	40	0	0	80
E	0	0	0	0	0	50
F	0	0	0	0	0	0



Adjacency List





Common Operations:

1. Connectedness

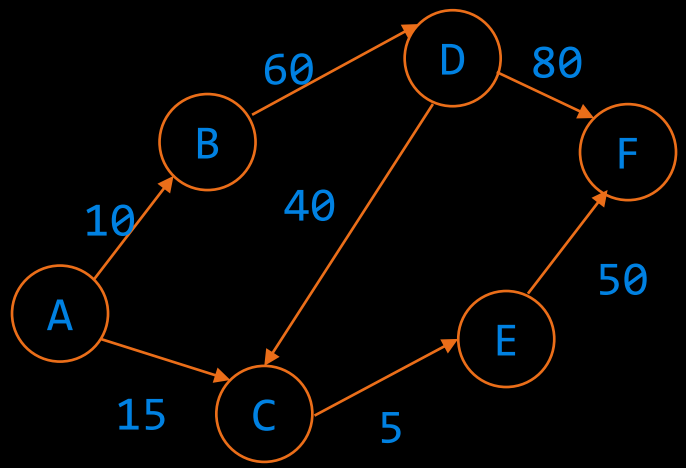
Is A connected to B?

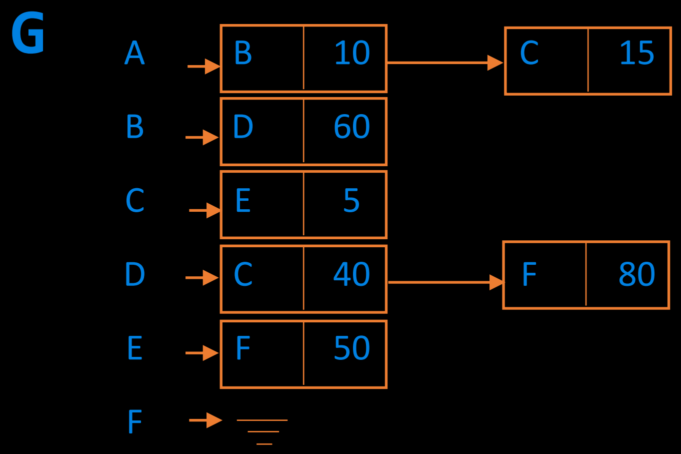
2. Adjacency

What are A's adjacent nodes?

Space: ?

Adjacency List





Sparse Graph:
Edges ~ Vertices

Common Operations:

1. Connectedness

   Is A connected to B?

   ```
   for each element x in G["A"]
       if x ! = 'B'
           ~ O(outdegree|V|)
   ```

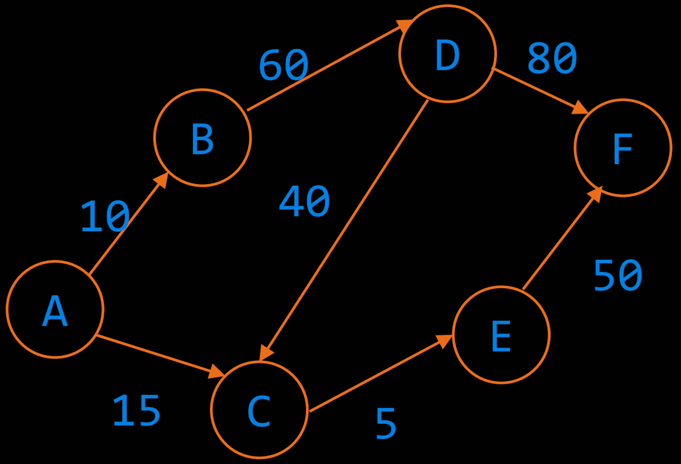
2. Adjacency

   What are A's adjacent nodes?

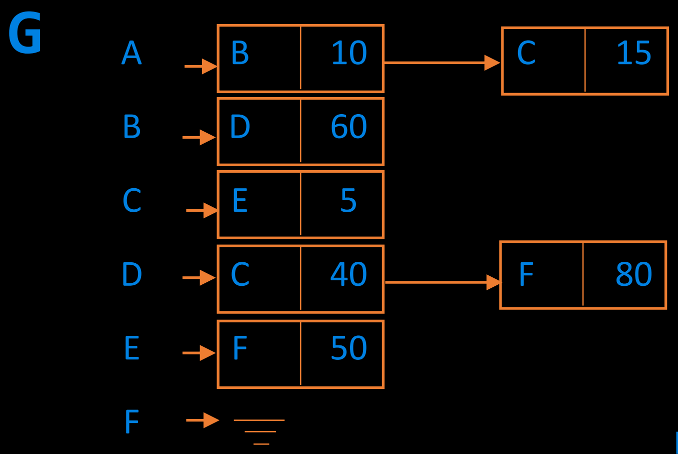
   G["A"] ~ O(outdegree|V|)

Space: O(|V| + |E|)

Adjacency List Implementation



Input
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50



Insertion:

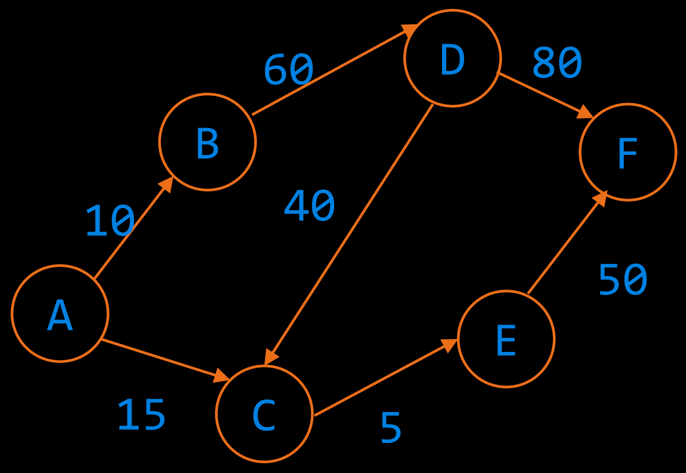
If to or from vertex not present add vertex

Otherwise add edge at the end of the list

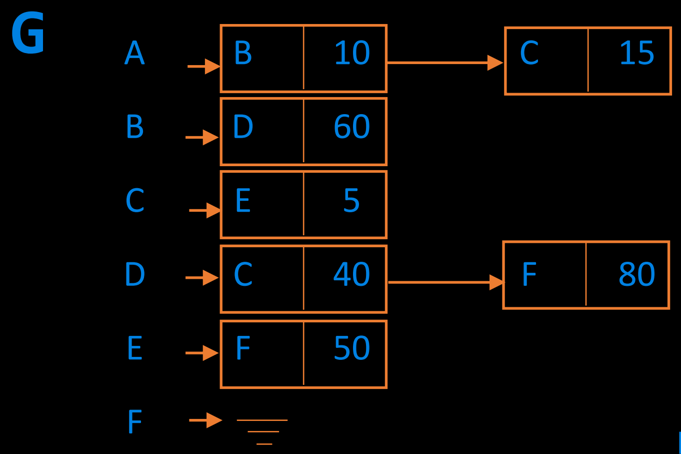
```
#include <iostream>
#include<map>
#include<vector>
#include<iterator>
using namespace std;

int main()
{
        int no_lines;
        string from, to, wt;
        map<string, vector<pair<string,int>>> graph;
        cin >> no_lines;
        for(int i = 0; i < no_lines; i++)
        {




        }
}
```

https://onlinegdb.com/HkJq9iFaI

Adjacency List Implementation



Input
7
A B 10
A C 15
B D 60
D C 40
C E 5
D F 80
E F 50



Insertion:

If to or from vertex not present add vertex

Otherwise add edge at the end of the list

```
#include <iostream>
#include<map>
#include<vector>
#include<iterator>
using namespace std;

int main()
{
        int no_lines;
        string from, to, wt;
        map<string, vector<pair<string,int>>> graph;
        cin >> no_lines;
        for(int i = 0; i < no_lines; i++)
        {
                cin >> from >> to >> wt;
                graph[from].push_back(make_pair(to, stoi(wt)));
                if (graph.find(to)==graph.end())
                        graph[to] = {};
        }
}
```

https://onlinegdb.com/HkJq9iFaI

Graph Implementation

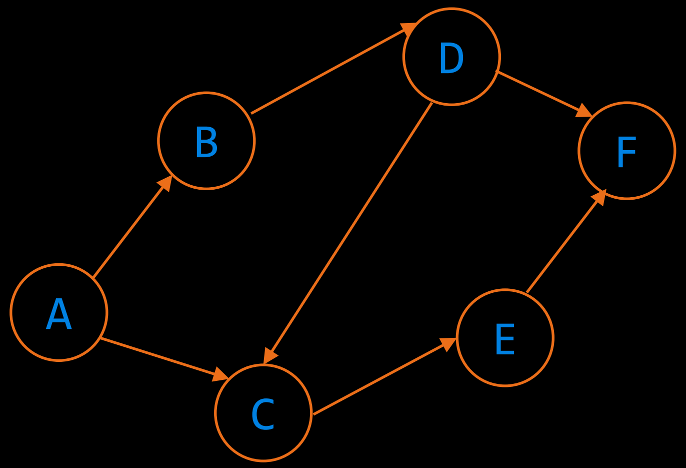
	Edge List	Adjacency Matrix	Adjacency List
Time Complexity: Connectedness	O(E)	O(1)	O(outdegree(V))
Time Complexity: Adjacency	O(E)	O(V)	O(outdegree(V))
Space Complexity	O(E)	O(V*V)	O(V+E)

One Graph API

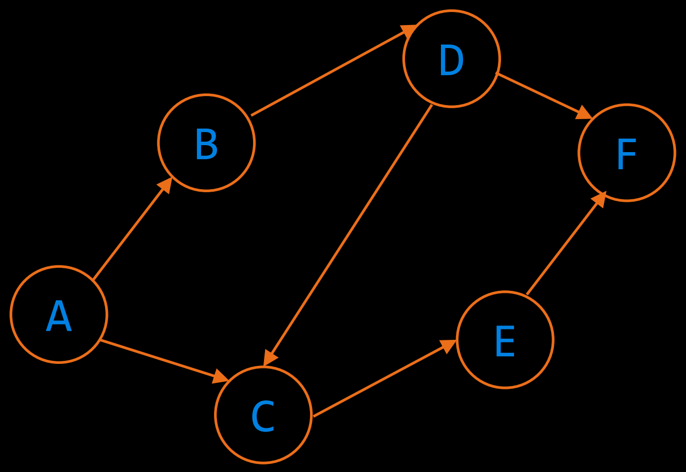
```
class Graph
{
    private:
      //Graph Data Structure

   public:
      Graph();
      Graph(int V); //Creates graph with v vertices
      int V();      //Returns number of vertices
      int E();      //Returns number of edges

      void insertEdge(int from, int to, int weight);
      bool isEdge(int from, int to);
      int getWeight(int from, int to);
      vector<int> getAdjacent(int vertex);
      void printGraph();
};
```

Graph Traversal

Breadth First Search



Breadth First Search

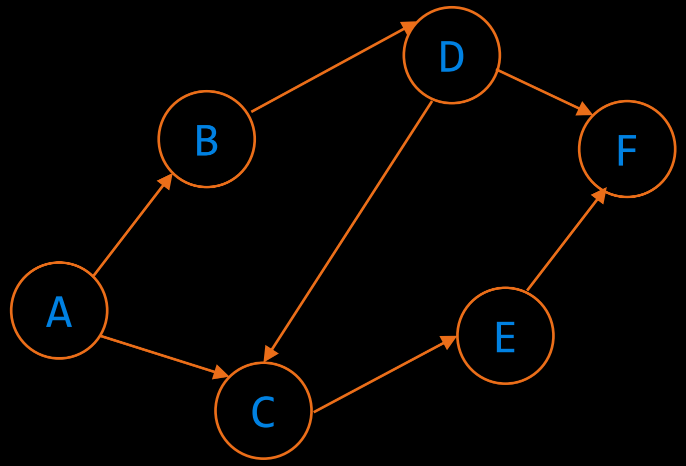


Valid BFS:

A, B, C, D, E, F
A, C, B, E, D, F

Breadth First Search

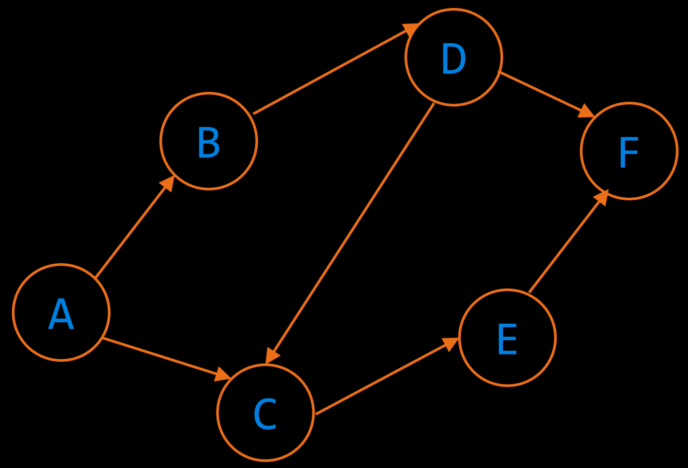
```
Algorithm for Breadth-First Search

1.     Take an arbitrary start vertex, mark it identified,
       and place it in a queue.
2.     while the queue is not empty
3.          Take a vertex, u, out of the queue and visit u.
4.          for all vertices, v, adjacent to this vertex, u
5.                 if v has not been identified or visited
6.                        Mark it identified
7.                        Insert vertex v into the queue.
8.            We are now finished visiting u.
```



https://onlinegdb.com/B12M0og1P

Breadth First Search

```
Algorithm for Breadth-First Search

1.     Take an arbitrary start vertex, mark it identified,
      and place it in a queue.
2.     while the queue is not empty
3.          Take a vertex, u, out of the queue and visit u.
4.          for all vertices, v, adjacent to this vertex, u
5.                 if v has not been identified or visited
6.                        Mark it identified
7.                        Insert vertex v into the queue.
8.         We are now finished visiting u.
```




```cpp
string source = "A";
std::set<string> visited;
std::queue<string> q;

visited.insert(source);
q.push(source);
cout<<"BFS: ";

while(!q.empty())
{
      string u = q.front();
      cout << u;
      q.pop();
      vector<string> neighbors = graph[u];
      std::sort(neighbors.begin(), neighbors.begin() + neighbors.size());
      for(string v: neighbors)
      {
            if(visited.count(v) == 0)
            {
                  visited.insert(v);
                  q.push(v);
            }
      }
}
```

https://onlinegdb.com/B12M0og1P

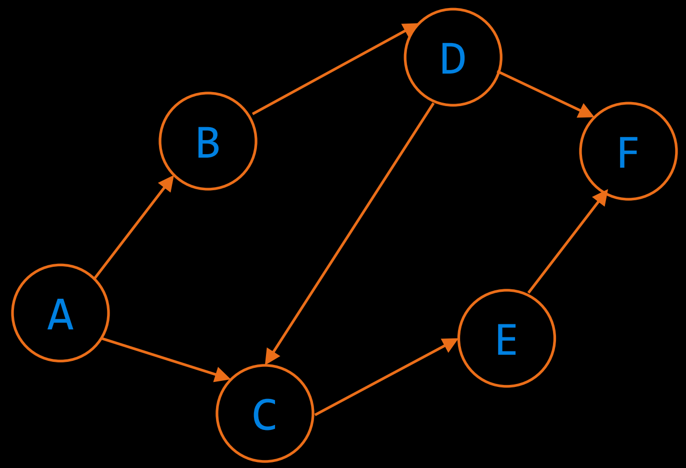
Breadth First Search: Alternate way (7.2.2)

```
Algorithm for Breadth-First Search

1.    Take an arbitrary start vertex, mark it identified,
     and place it in a queue.
2.    while the queue is not empty
3.         Take a vertex, u, out of the queue and visit u.
4.         for all vertices, v, adjacent to this vertex, u
5.                if v has not been identified or visited
6.                       Mark it identified
7.                       Insert vertex v into the queue.
8.        We are now finished visiting u.
```

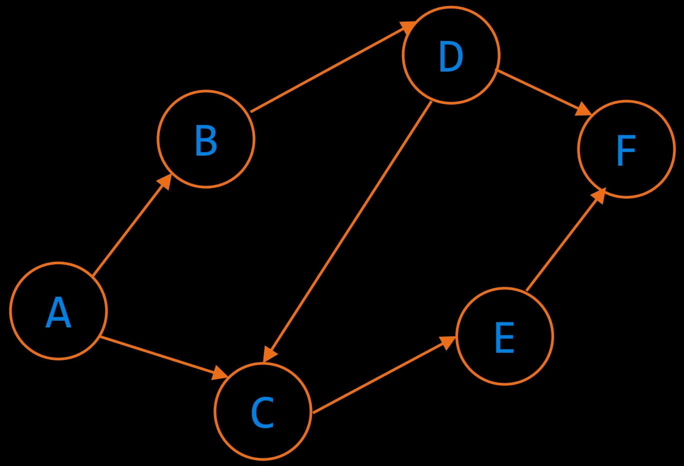
```
// Visited Vertices Alternate
set<string> visited;
visited.insert(source);
if(visited.count(v)==0)
      visited.insert(v);
```

```
void bfs(const Graph& graph, int src)
{
    vector<bool> visited(graph.numVertices);
    queue<int> q;

    visited[src] = true;
    q.push(src);

    while (!q.empty())
    {
        int u = q.front();
        cout << u << " ";
        q.pop();

        for (int v : graph.adjList[u])
        {
            if (!visited[v])
            {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}
```

Depth First Search



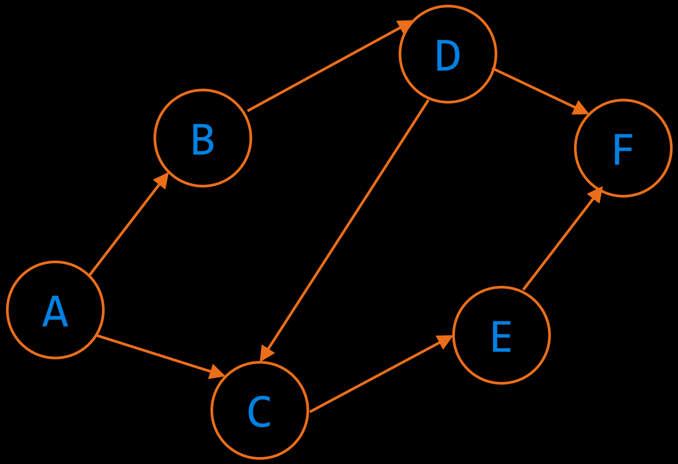
Depth First Search



Valid DFS: A, B, D, C, E, F

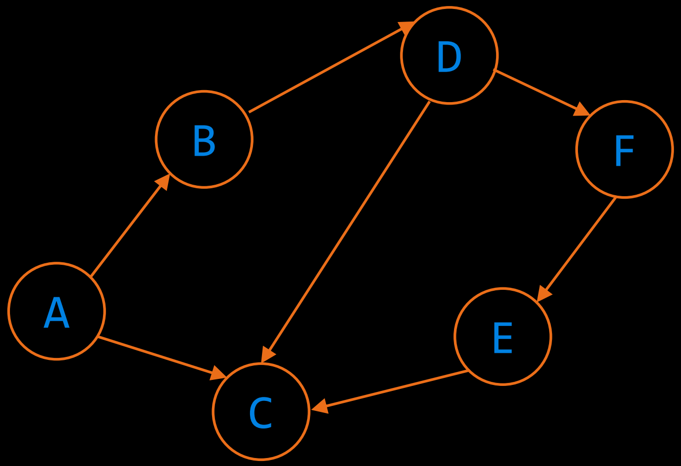
Depth First Search

```
Algorithm for Depth-First Search

1.    Take an arbitrary start vertex, mark it visited, and place it in a stack.
2.    while the stack is not empty
3.          the item on top of the stack is u
4.           if there is a vertex, v, adjacent to this vertex, u, that has not been visited
5.                   Mark v visited
6.                    Push vertex v onto the top of the stack
7.          else
8.                   pop stack
```



Depth First Search – Modified BFS

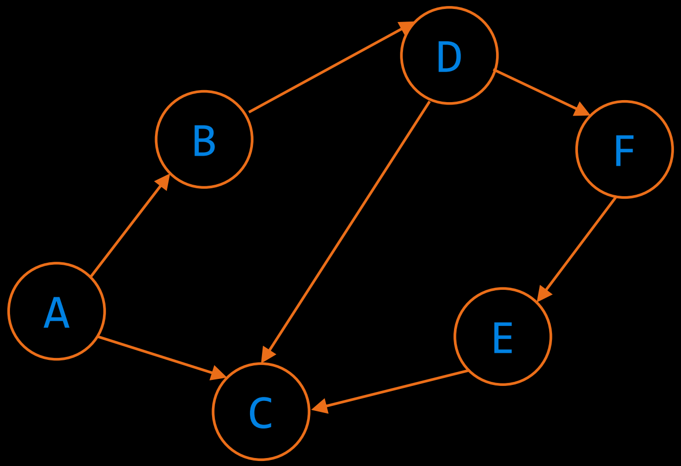
```
Algorithm for Depth-First Search

1.     Take an arbitrary start vertex, mark it identified,
      and place it in a stack.
2.     while the stack is not empty
3.          Take a vertex, u, out of the stack and visit u.
4.          for all vertices, v, adjacent to this vertex, u
5.                if v has not been identified or visited
6.                      Mark it identified
7.                      Insert vertex v into the stack.
8.          We are now finished visiting u.
```



https://onlinegdb.com/SkNolhlkP

Depth First Search – Modified BFS

```
Algorithm for Depth-First Search

1.     Take an arbitrary start vertex, mark it identified,
      and place it in a stack.
2.     while the stack is not empty
3.          Take a vertex, u, out of the stack and visit u.
4.          for all vertices, v, adjacent to this vertex, u
5.                 if v has not been identified or visited
6.                       Mark it identified
7.                       Insert vertex v into the stack.
8.            We are now finished visiting u.
```




```cpp
string source = "A";
std::set<string> visited;
std::stack<string> s;

visited.insert(source);
s.push(source);
cout<<“DFS: ";

while(!s.empty())
{
      string u = s.top();
      cout<<u;
      s.pop();
      vector<string> neighbors = graph[u];
      for(string v: neighbors)
      {
            if(visited.count(v)==0)
            {
                  visited.insert(v);
                  s.push(v);
            }
      }
}
```

https://onlinegdb.com/SkNolhlkP

BFS vs DFS

```cpp
string source = "A";
std::set<string> visited;
std::queue<string> q;

visited.insert(source);
q.push(source);
cout<<"BFS: ";

while(!q.empty())
{
	string u = q.front();
	cout<<u;
	q.pop();
	vector<string> neighbors = graph[u];
	for(string v: neighbors)
	{
		if(visited.count(v)==0)
		{
			visited.insert(v);
			q.push(v);
		}
	}
}
```

```cpp
string source = "A";
std::set<string> visited;
std::stack<string> s;

visited.insert(source);
s.push(source);
cout<<“DFS: ";

while(!s.empty())
{
	string u = s.top();
	cout<<u;
	s.pop();
	vector<string> neighbors = graph[u];
	for(string v: neighbors)
	{
		if(visited.count(v)==0)
		{
			visited.insert(v);
			s.push(v);
		}
	}
}
```

Theoretical Complexity: O(V+E)

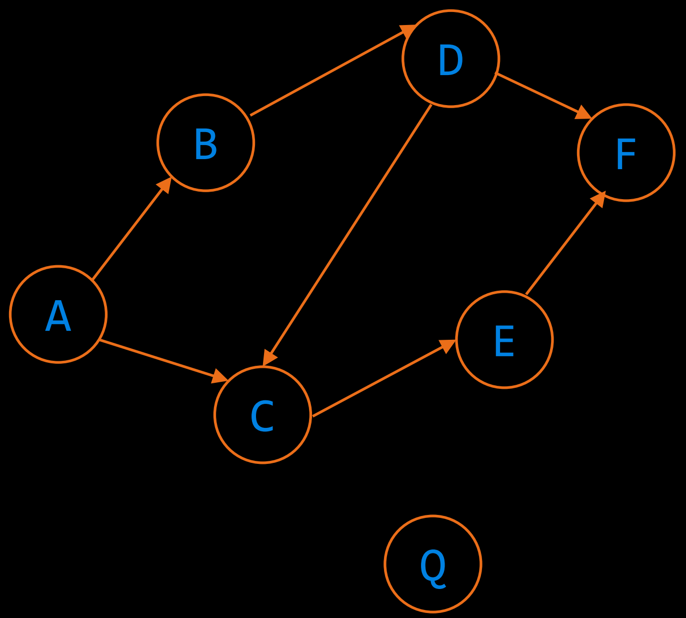
Mentimeter

3176 5158

Questions

s-t Path

Is there a path between vertices s and t?

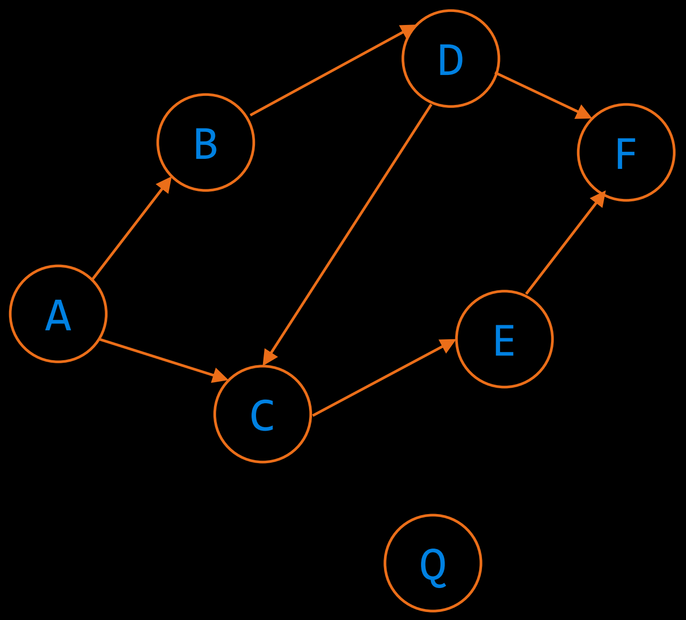


Is there a path between vertices A and C? - Yes

Is there a path between vertices A and Q? - No

s-t Path

Is there a path between vertices s and t?



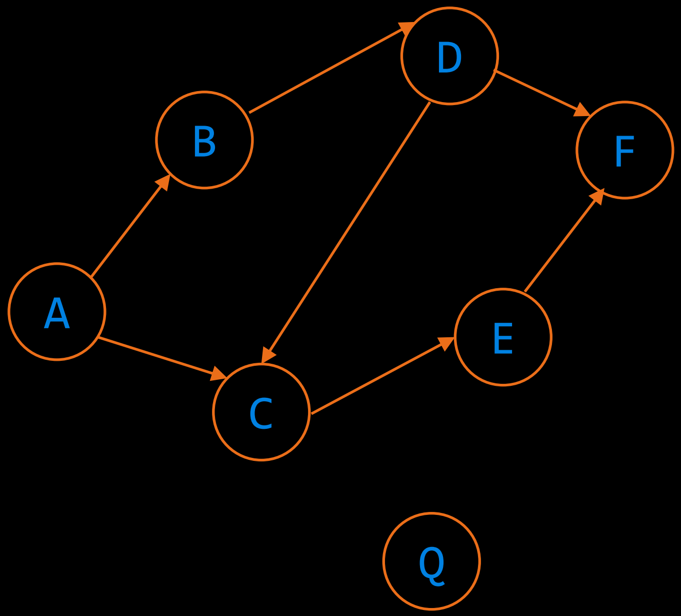
Is there a path between vertices A and C? - Yes

Is there a path between vertices A and Q? - No

Solution

Perform DFS or BFS with source "s" and if we encounter "t" in the path/traversal, then return True otherwise False

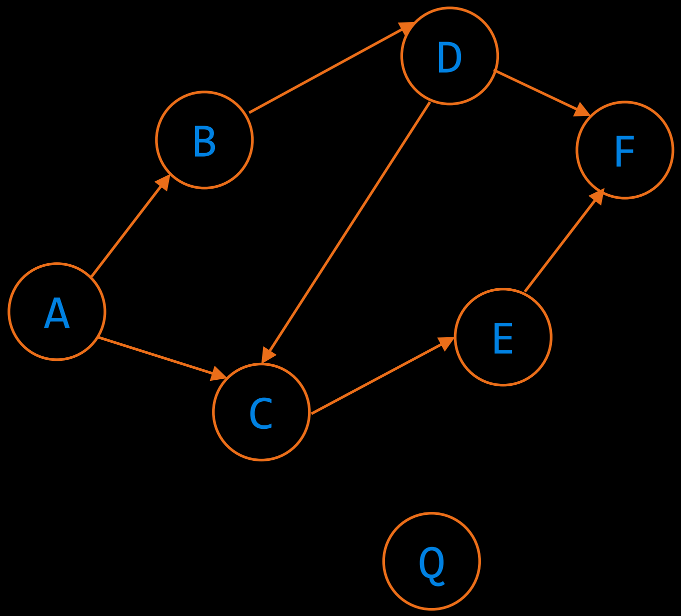
7.2.1 DFS to Find Whether a Given Vertex is Reachable (Iterative)



s-t Path

```
bool dfs(const Graph& graph, int src, int dest)
{
    set<int> visited;
    stack<int> s;
    visited.insert(src);
    s.push(src);
    while(!s.empty())
    {
        int u = s.top();
        s.pop();
        for(auto v: graph.adjList[u])
        {
            if(v == dest)
                return true;
            if ((visited.find(v) == visited.end()))
            {
                visited.insert(v);
                s.push(v);
            }
        }
    }
    return false;
}
```

7.2.1 DFS to Find Whether a Given Vertex is Reachable (Recursive)



s-t Path: Recursive

```cpp
bool dfs_helper(const Graph& graph, int src, int dest, vector<bool>& visited)
{
    visited[src] = true;

    if (src == dest)
        return true;

    for (int neighbor : graph.adjList[src]) {
        if (!visited[neighbor]) {
            if (dfs_helper(graph, neighbor, dest, visited))
                return true;
        }
    }
    return false;
}

bool dfs(const Graph& graph, int src, int dest)
{
    vector<bool> visited(graph.numVertices);
    return dfs_helper(graph, src, dest, visited);
}
```

Maze as a Graph: Maze Escape

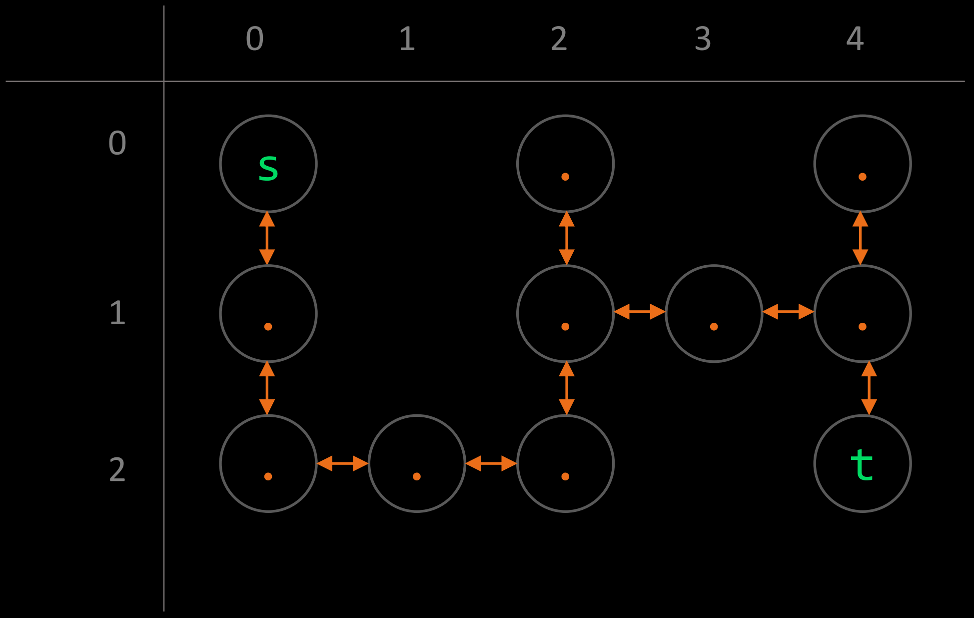
What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```

Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

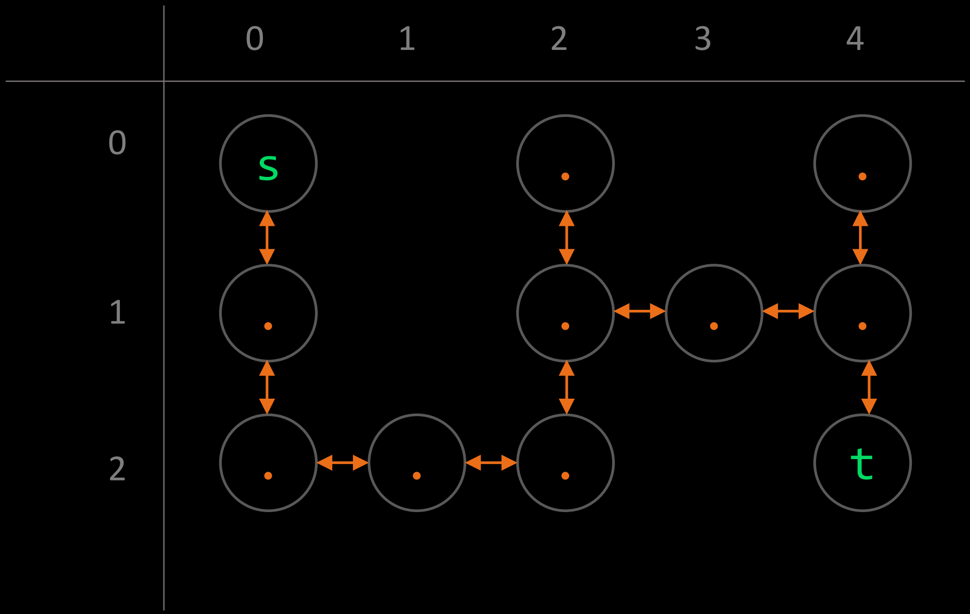
```
s#.#.
.#...
...#t
```



Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```

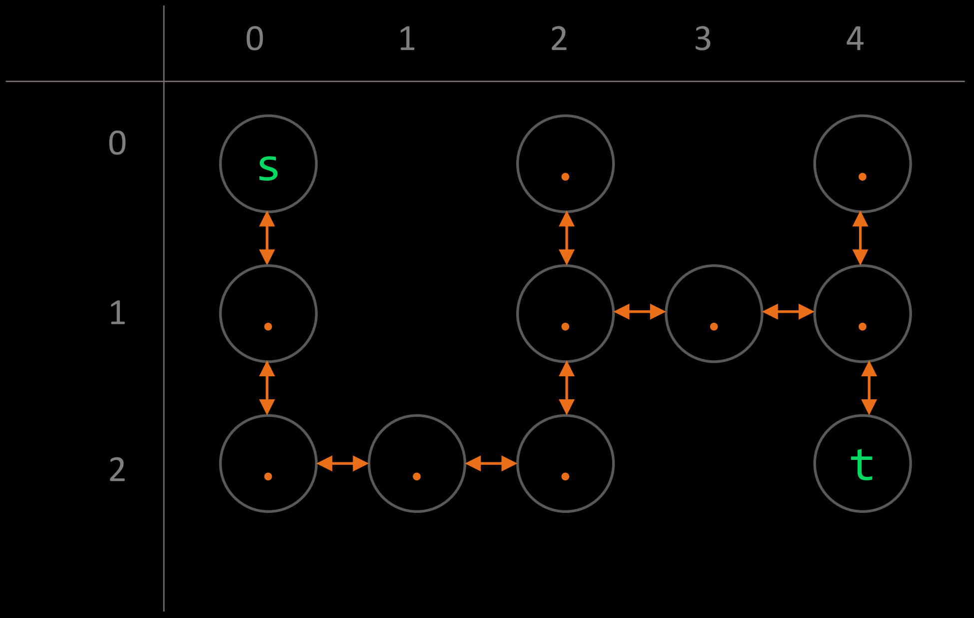



```
General BFS Algorithm

1.    Start from vertex, ‘s’ (0,0), mark it identified, and
     place it in a queue.
2.    while the queue is not empty
3.         Take a vertex, u, out of the queue and visit u.
4.         for all vertices, v, adjacent to this vertex, u
5.               if v has not been identified or visited
6.                     Mark it identified
7.                     Insert vertex v into the queue.
8.        We are now finished visiting u.
```

Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```

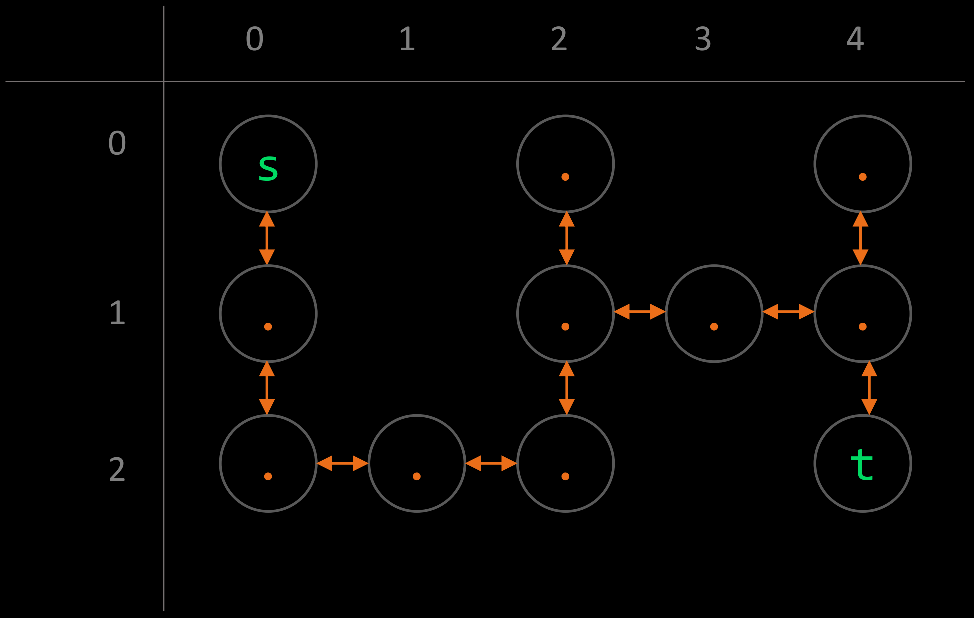



```
General BFS Algorithm

1.    Start from vertex, ‘s’ (0,0), mark it identified, and
     place it in a queue.
2.    while the queue is not empty
3.        Take a vertex, u, out of the queue and visit u.
4.        for all vertices, v, adjacent to this vertex, u
…
```

Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```



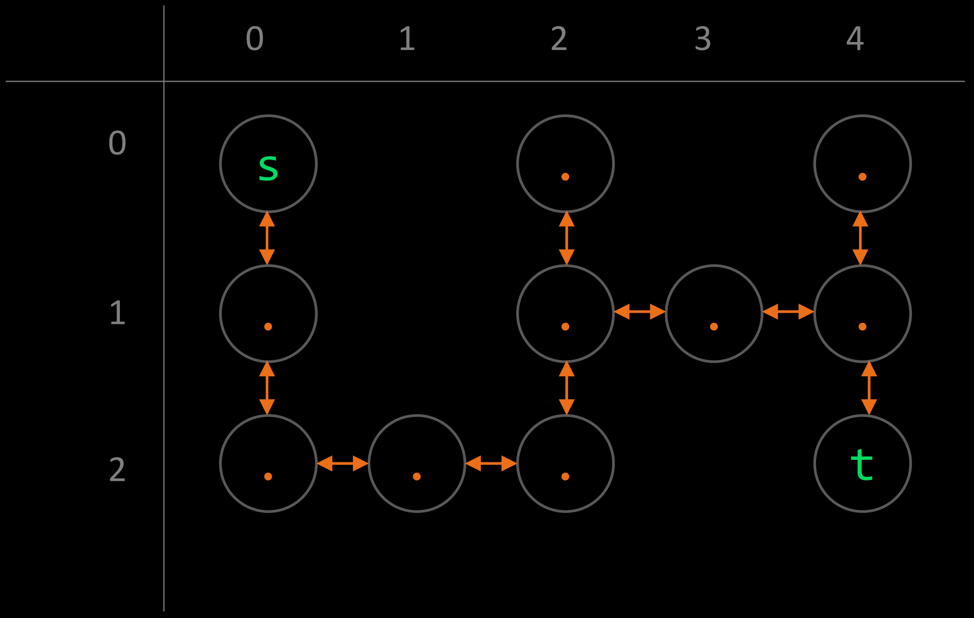

```
General BFS Algorithm

1.    Start from vertex, 's' (0,0), mark it identified, and
      place it in a queue.
2.    while the queue is not empty
3.        Take a vertex, u, out of the queue and visit u.
4.        for all vertices, v, adjacent to this vertex, u
…
```

```
//Keeping track of visited nodes

vector<vector<bool>> visited(3, vector<bool>(5, false));
Visited[0][0] = true;
```

Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```




```
General BFS Algorithm

1.     Start from vertex, 's' (0,0), mark it identified, and
      place it in a queue.
2.     while the queue is not empty
3.         Take a vertex, u, out of the queue and visit u.
4.         for all vertices, v, adjacent to this vertex, u
…
```

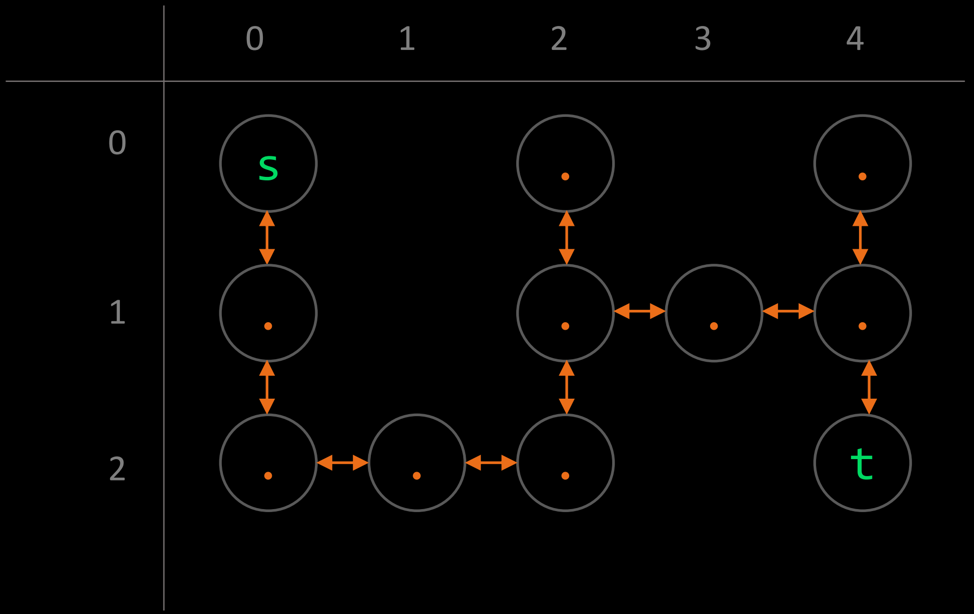
```
//Keeping track of visited nodes

vector<vector<bool>> visited(3, vector<bool>(5, false));
```

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

Maze as a Graph: Maze Escape

What is the shortest path between vertices s and t?

s#.#.
.#...
...#t




```
General BFS Algorithm

1.    Start from vertex, 's' (0,0), mark it identified, and
      place it in a queue.
2.    while the queue is not empty
3.        Take a vertex, u, out of the queue and visit u.
4.        for all vertices, v, adjacent to this vertex, u
…
```

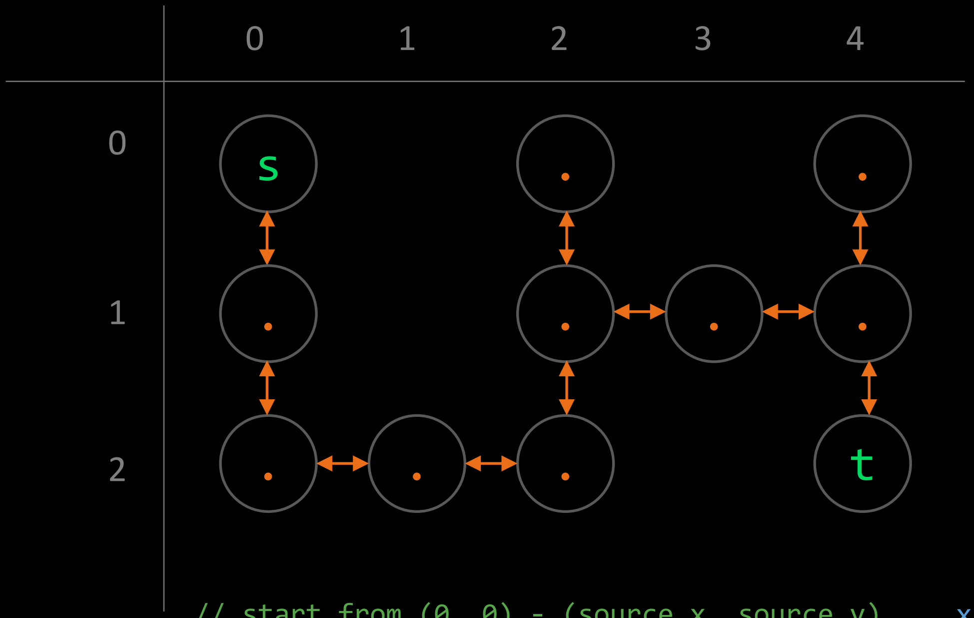
	0	1	2	3	4
0	T	F	F	F	F
1	F	F	F	F	F
2	F	F	F	F	F

```
//Keeping track of visited nodes

vector<vector<bool>> visited(3, vector<bool>(5, false));
visited[0][0] = true;
```

Maze as a Graph: Maze Escape

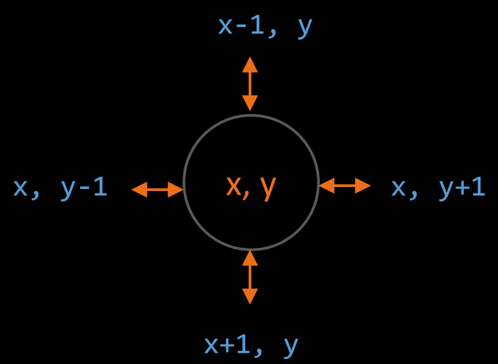
What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```




```
General BFS Algorithm

1.    Start from vertex, ‘s’ (0,0), mark it identified, and
     place it in a queue.
2.    while the queue is not empty
3.         Take a vertex, u, out of the queue and visit u.
4.         for all vertices, v, adjacent to this vertex, u
…
```

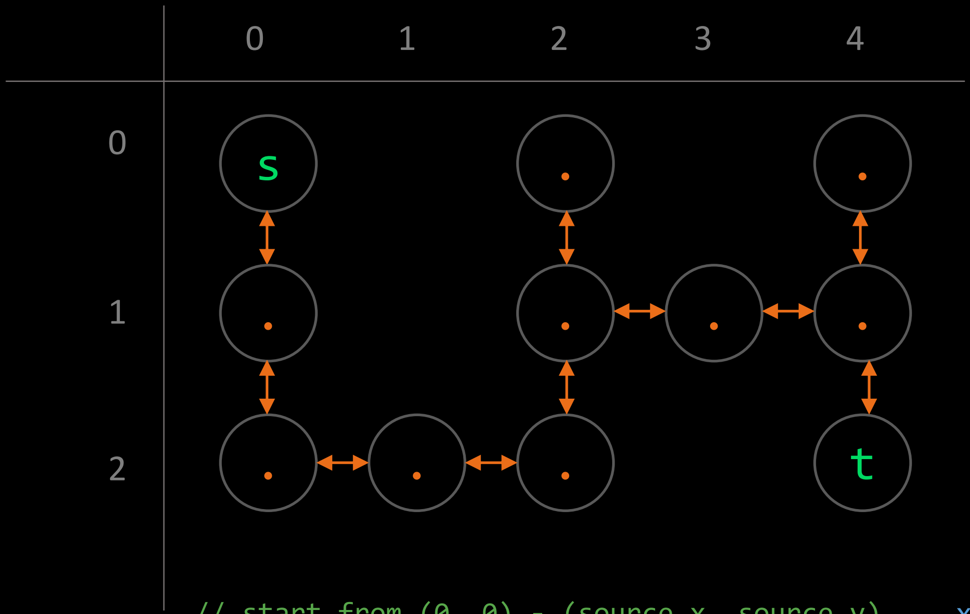
```
// start from (0, 0) - (source_x, source_y)
int x = 0, y = 0;

// down, up, right, left
std::vector<int> dx = {1, -1, 0, 0};
std::vector<int> dy = {0, 0, 1, -1};
```



Maze as a Graph: Maze Escape

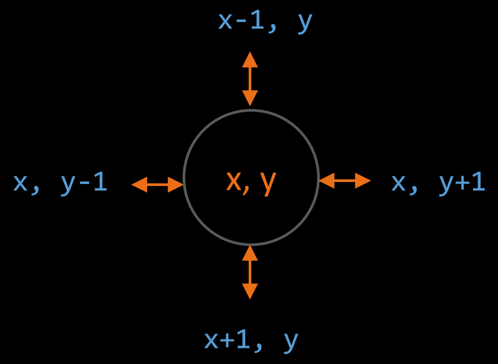
What is the shortest path between vertices s and t?

```
s#.#.
.#...
...#t
```




```
// start from (0, 0) - (source_x, source_y)
int x = 0, y = 0;

// down, up, right, left
std::vector<int> dx = {1, -1, 0, 0};
std::vector<int> dy = {0, 0, 1, -1};
```

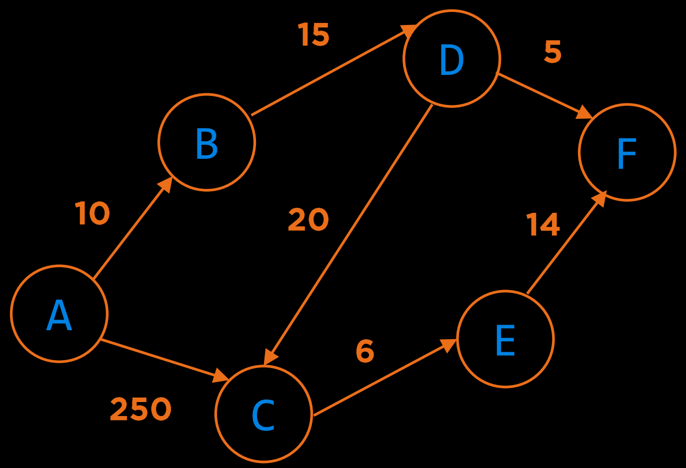
```
General BFS Algorithm

1.    Start from vertex, 's' (0,0), mark it identified, and
     place it in a queue.
2.    while the queue is not empty
3.         Take a vertex, u, out of the queue and visit u.
4.         for all vertices, v, adjacent to this vertex, u
…
```




```
// start from (0, 0) - (source_x, source_y)
int x = 0, y = 0;

// down, up, right, left
std::vector<int> dx = {1, -1, 0, 0};
std::vector<int> dy = {0, 0, 1, -1};

for(int i = 0; i < 4; i++) {
    int nx = x + dx[i];
    int ny = y + dy[i];
    // check if nx, ny is in the grid
    if(nx < 0 || ny < 0)
            continue;
    …
    …
}
```

Shortest Weighted s-t Path

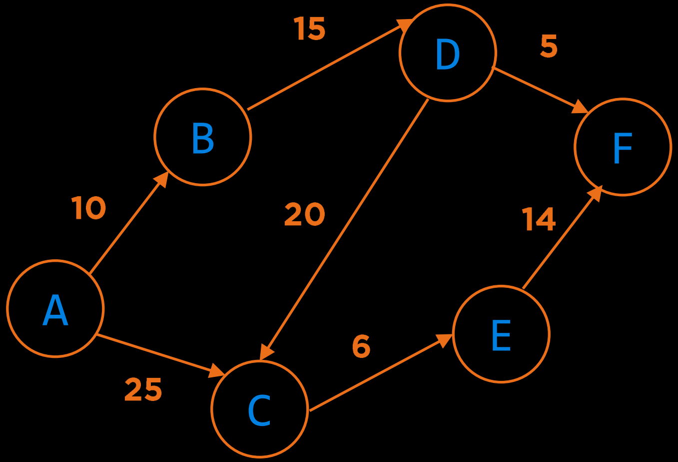
What is the shortest weighted path between vertices s and t?

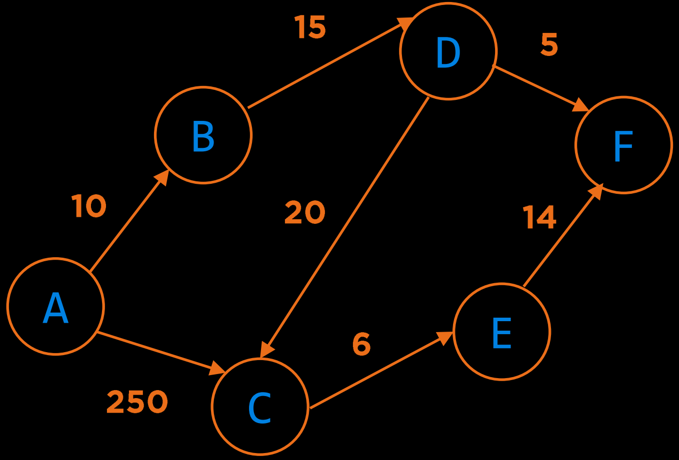


Problem with s-t Path

What if the edges are weighted?

The algorithms do not consider the weights.



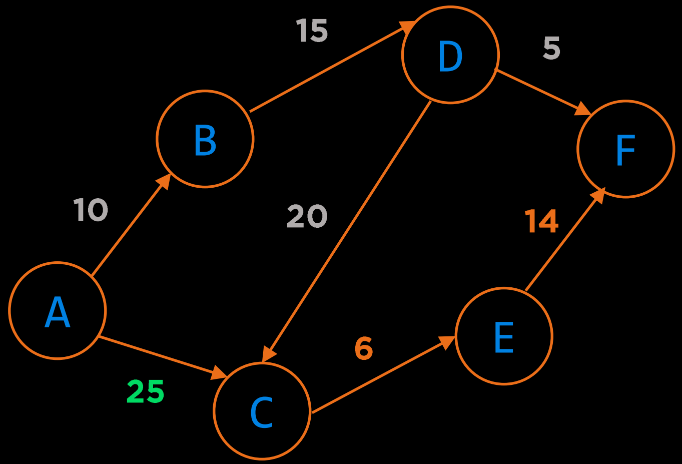


Problem with s-t Path

What if the edges are weighted?

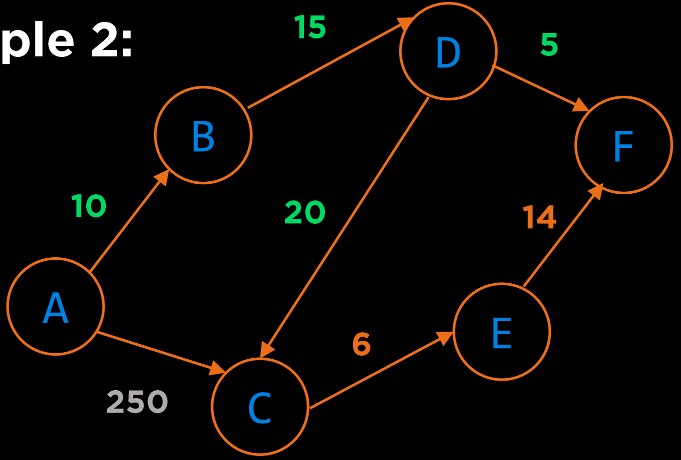
The algorithms do not consider the weights.

Example 1:



Example 1: Path for A to C will be A-B-D-C for a DFS traversal which will have a total cost of 45 against 25 for the path directly from A-C.

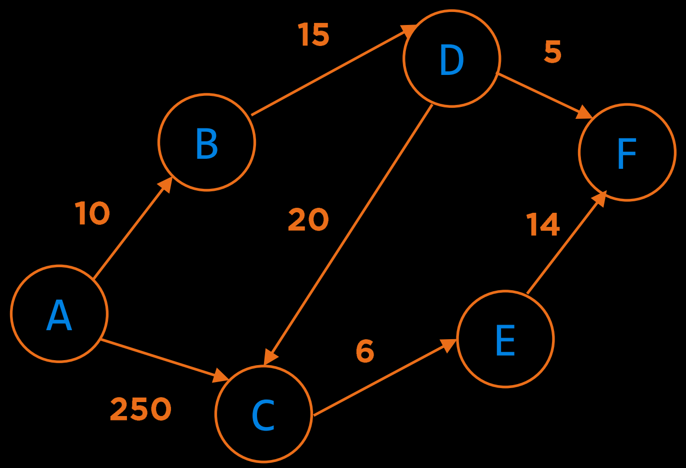
Example 2:



Example 2: Path for A to C will be A-C for a BFS traversal which might have a total cost of 250 against 45 for the path directly from A-B-D-C.

Shortest Weighted s-t Path

What is the shortest weighted path between vertices s and t?



- Dijkstra's Algorithm
  - Single Source: Path to all vertices
  - Directed Graphs
  - No negative weights allowed
  - No negative weight cycles allowed
- Bellman Ford
  - Single Source: Path to all vertices
  - Negative Weights allowed
  - No negative weight cycles allowed
- Floyd-Warshall
  - All pair shortest paths
- A* Search