Graphs

Quick note on my slide decks.

- My slides are simple, I don’t (cannot) spend time on making fancy fonts and effects.
- This is because I would much rather spend that time on adding useful content for u all.
  - not too little
  - and not too much either (there’s always the books and online resources for that)
- I also don’t use some of the features of power point where I could put text on a busy slide one by one.
- So, please try to follow me when I am going thru the slides and don’t try to read all text on a slide, when I may just be talking about the first bullet on that slide (try not to get ahead of me ☺)
- Happy learnings.

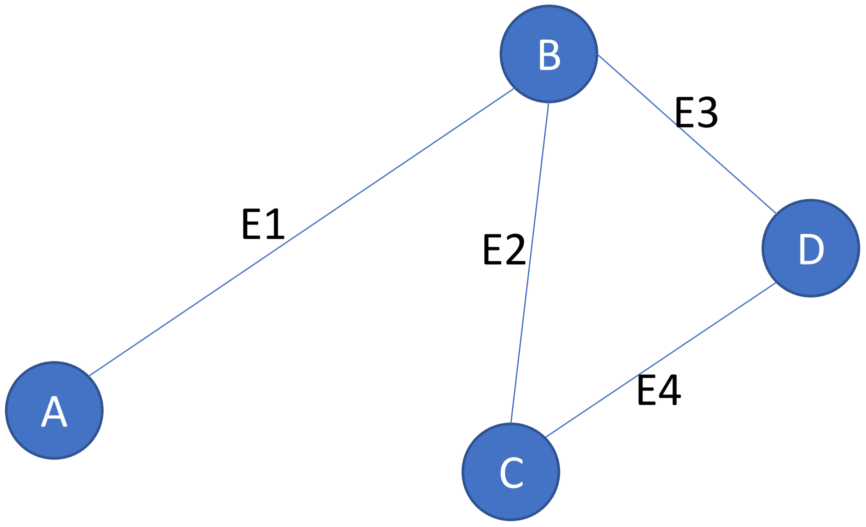
Graph

- Graphs are also a very commonly used data structure.
- Do not confuse them with the graphs u see in Mathematics… there are no x,y coordinates here☺
- Are used to represent a variety of things:
  - Flight connections.
  - Train connections.
  - To generalize, we can say transportation connections
  - And to generalize even further, any routing representation
    - Electrical circuits
    - Web (internet)
    - Social connections (Facebook, Twitter, LinkedIn, etc.)
  - Tasks / job schedules
    - Graphs can depict dependencies among tasks.
  - Etc.

Graph

A graph contains:

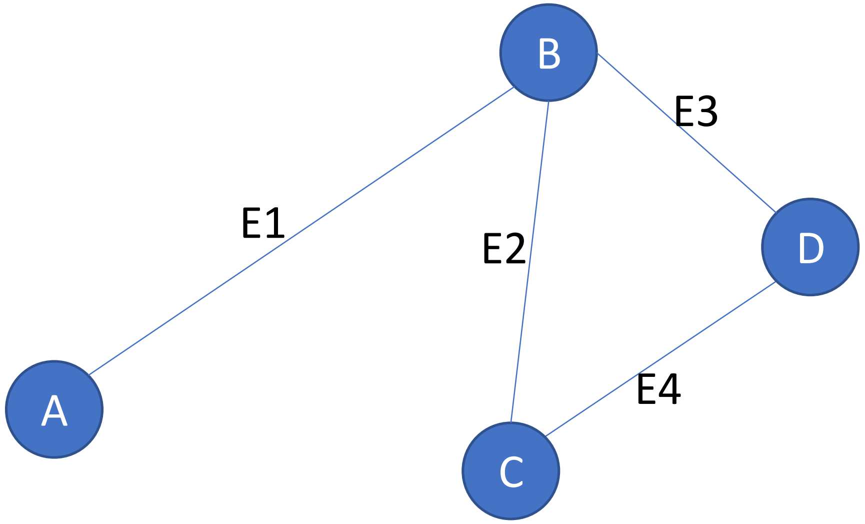
- A set of edges
- A set of vertices



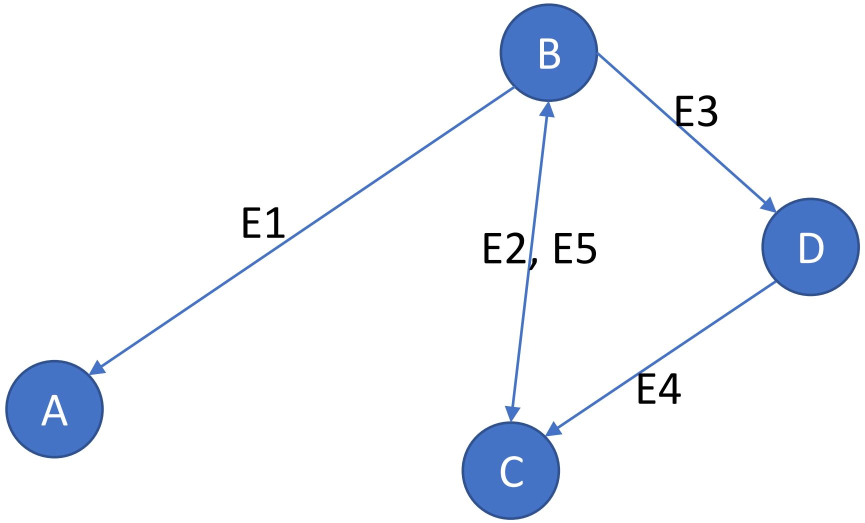
- In the graph shown here, we have
  - Vertices:
    - A, B, C and D
  - Edges:
    - E1, E2, E3, E4
  - An edge is really just a pair of vertices
    - E1: (A,B)
    - E2: (B, C)
    - And so on.

Graph

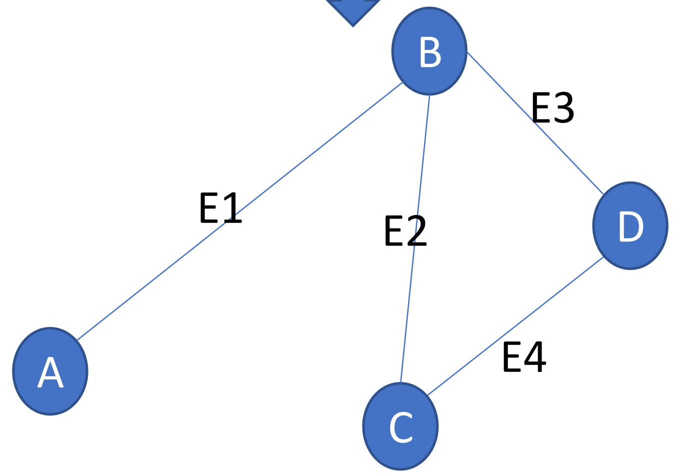
- We can formalize the definition to:
- G = (V, E)
    - Where:
        - V is a set of vertices
        - E is a set of edges (vertex pairs)



- A graph can be
    - Undirected
    - Directed
        - Also called *digraph*

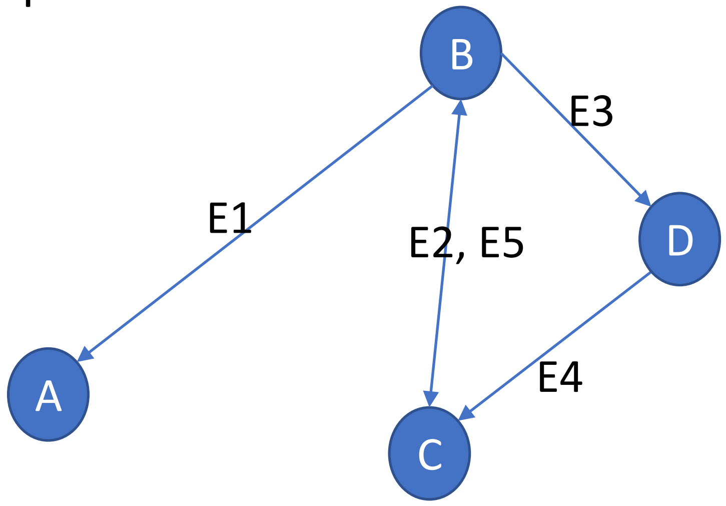


- **Undirected** graph:
  - V(G) = { A, B, C, D }
  - E(G) = { (A,B), (B,C), (C,D), (B,D) }



- **Directed** graph:
  - V(G) = { A, B, C, D }
  - E(G) = { (B,A), (B,C), (C,B), (B,D), (D,C) }
    - Note we need to have (B,C) **and** (C,B)

- As u can tell, in a directed graph, the order of vertices in the edge is important
  - So, (V_1, V_2) is not the same as (V_2, V_1)
- So, we say
  - Directed graph:
    - E is a set of **ordered** pairs of vertices
  - Undirected graph:
    - E is a set of **unordered** pairs of vertices



Graphs vs Trees

- A tree is essentially a graph with certain restrictions:

  - Tree does not have any loops.
  - A parent node has a path to its child(ren), but not the other way around.
  - Each node in a tree has only one parent (except root node).
  - There is only one root node.
  - There are always N-1 edges (N being the total number of nodes in a tree)

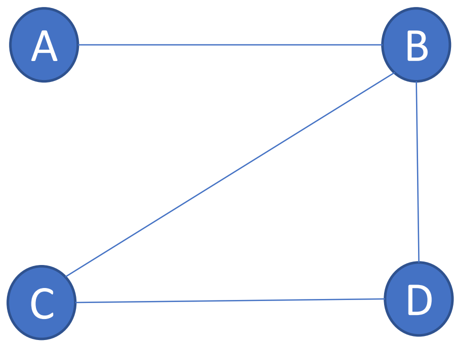
Some definitions

- Adjacent vertices
  - Two vertices are said to be adjacent if there is an edge between them

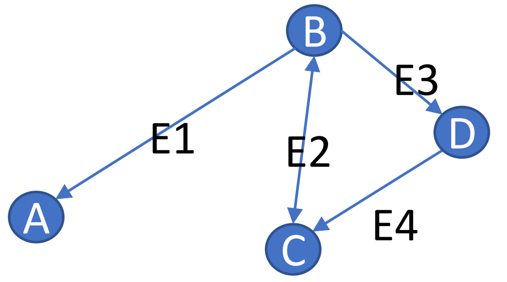
- Path
  - If you can get from a vertex V_a to a vertex V_b , there is a path between those two vertices.
  - In other words, the sequence of vertices that have to be traversed in order to get from V_a to V_b are what constitute this path.

Some definitions

- Degree
  - Undirected graph:
    - *Degree* of a vertex is the number of edges that come out of that vertex.
      - Or go into that vertex (it's the same thing for an undirected graph)
    - Degree of A: 1
    - Degree of B: 3

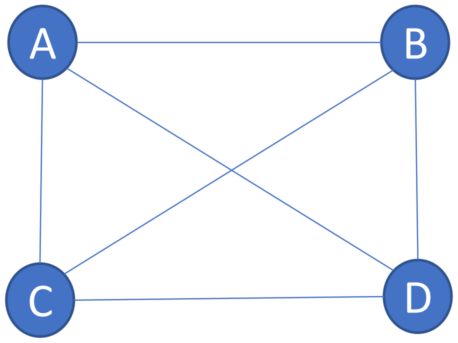


  - Directed graph:
    - *In-degree* of a vertex is the number of edges that are coming into the vertex
    - *Out-degree* of a vertex is the number of edges that are going out of a vertex
    - A: In-degree is 1, out-degree is 0
    - B: In-degree is 1, out-degree is 3
    - C: In-degree is 2, out-degree is 1
    - Note that *sum of in-degree = sum of out-degree*
      - *This is for the whole graph (i.e., all vertices)*



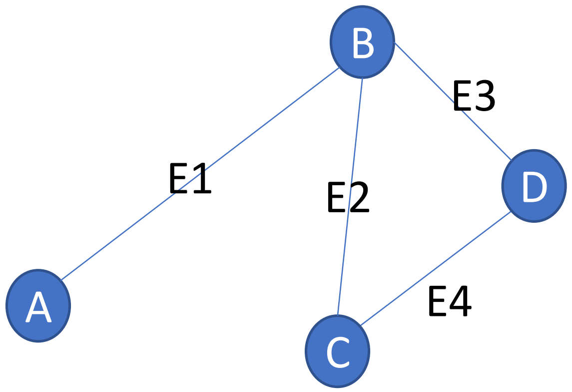
Some definitions

- Complete graph
  - This is a graph in which each vertex is *directly* connected to every other vertex.
  - In other words, the length of the shortest path from any vertex to any other vertex is 1.



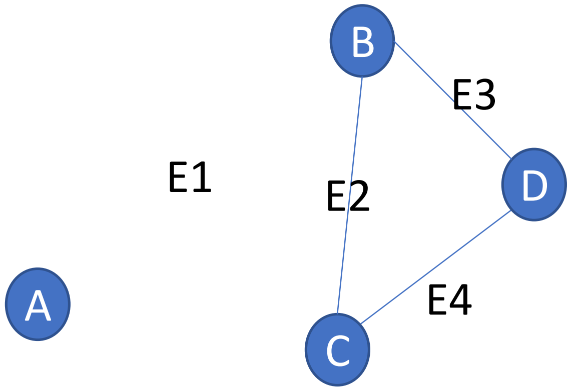
Some definitions

- Connected graph
  - If a path exists between any two vertices in a graph, it is a *connected* graph.
  - Graph on right is a connected graph →
  - Question:
    - Is a complete graph also a connected graph?
    - Is a connected graph also a complete graph?

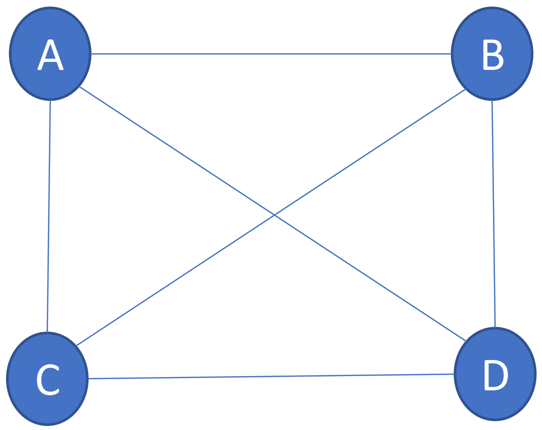


Some definitions

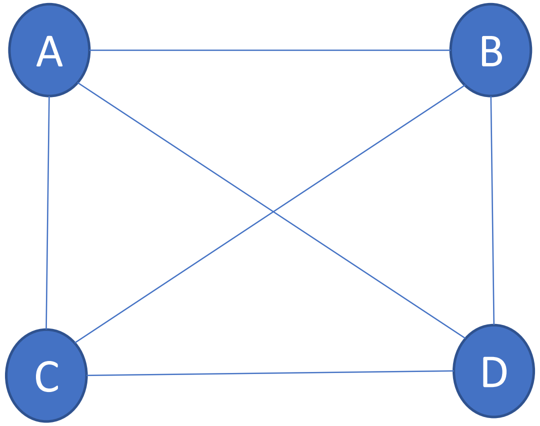
- Disconnected graph
  - If there is at least one vertex where there is no path from any other vertex, it is a *disconnected* graph.



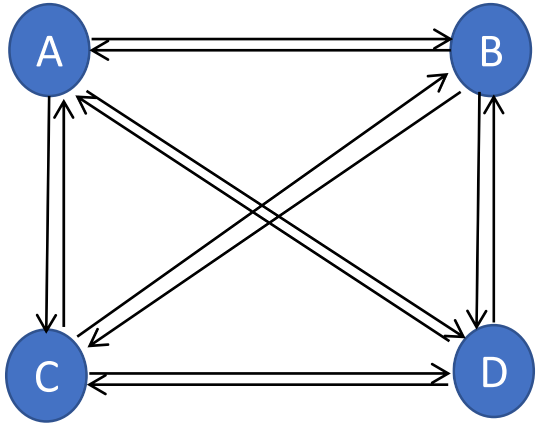
- How many edges are in a *complete undirected* graph with N vertices?



- How many edges are in a *complete undirected* graph with N vertices?
  - N * (N – 1) / 2



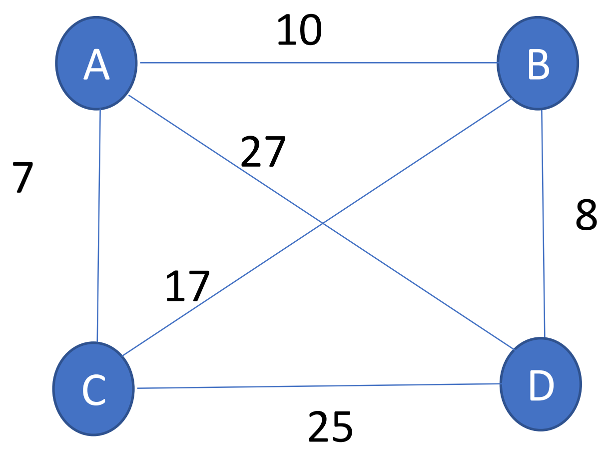
- How many edges are in a *complete directed* graph with N vertices?
    - N * (N – 1)



Some definitions

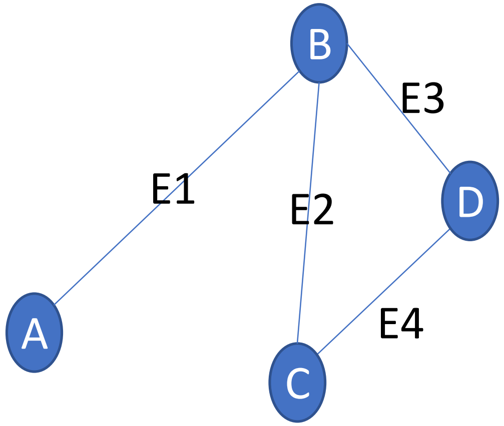
- **Weighted** graph
  - This is a graph in which a *weight* is assigned to all the edges.
  - weight is a generic term here, but it could be like the cost of that edge.

  - Example:
    - If a graph represents the road connectivity between cities (vertices), then the weight of an edge could represent the distance between the cities connected by that edge.
    - Or it could represent the cost of gas for driving a delivery truck from one city to another.
    - Or the amount of goods that can be transported by the trucks.



Representing graphs

- There are many ways to represent a graph in a data structure



1. **Edge list**

   - This is a collection of vertex pairs.
     - (A,B), (B,C), (B,D) (C, D)

   - This could be implemented as an array of vertex pairs, so something like:

     ```
     class VertexPair  ( or class Edge)
     {
             Vertex        v1;
             Vertex        v2;
     };
     ```

     We now use an array of VertexPair to represent the list of edges.

   - Space complexity:
     - Size of this list is E, where E is the number of edges.

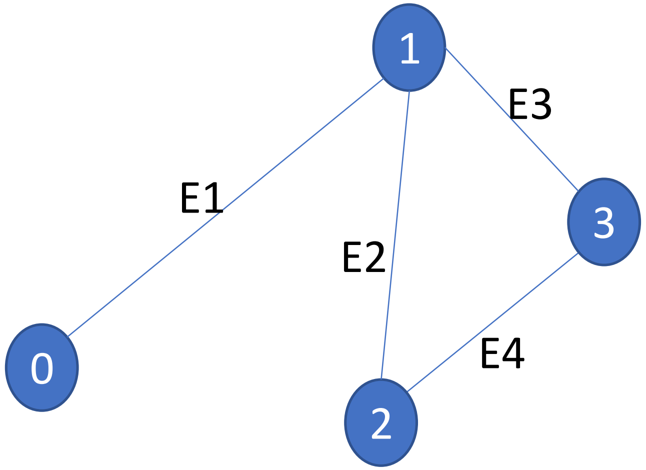
   - Time complexity:
     - Search for an edge in this list: O(E)

   - Enumerating adjacent vertices of a given vertex: O(E)

   - **SUMMARY** : Collection of all edges in the graph

Adjacency matrix

2. Adjacency matrix



- Its a matrix (2 dimensional array) of size V x V
  - If matrix[V_1][V_2] == 0
    - there is no edge between V_1 and V_2
  - Else
    - there is an edge between V_1 and V_2

	0	1	2	3
0	**0**	**1**	**0**	**0**
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0

- Size (space): V X V = ← Change font color to reveal
- Search if exists an edge (V_i, V_j) : ← Change font color to reveal
- Enumerating adjacent vertices of a given vertex V_i : ← Change font color

Adjacency matrix

- Time complexity for look up here is great
- But space complexity (V^2) is an issue with this representation.
  - Imagine the space requirement for a huge graph.
- Can be used for small graphs, but not big ones.

- For a given vertex V_1
  - **In-degree** of V_1 is count of non-zero entries in the **row** of matrix[V_1]
    - If u want to look a bit mathematical, then this is for u ☺. If not, u can ignore it.
    - In-degree of V_1 = $\sum_{j=0}^{v-1} matrix[V_1][Vj]$ ← Assumes if edge (V_1, V_j) exists, then matrix has 1, else 0
  - **Out-degree** of V_1 is count of non-zero entries in the **column** of matrix [V_i][V_1]
    - Out-degree V_1 of = $\sum_{i=0}^{v-1} matrix[Vi][V_1]$ ← Assumes if edge (V_i, V_1) exists, then matrix has 1, else 0
  - These are true for undirected as well as directed graphs (aka digraph)

Representing graphs

3. Adjacency list

- There is a neighbor list for each vertex, i.e.:
  - we store a list for each vertex
  - and this list contains its neighbor vertices.

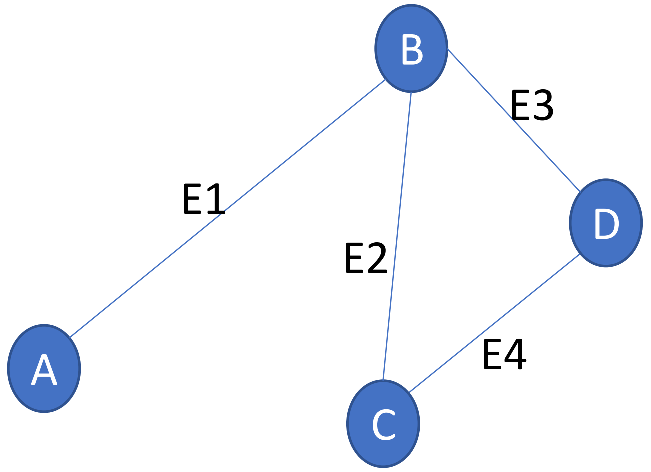
- See next slide for a diagrammatic representation.

- For example:
  - Lets say there are N vertices, from 0 to N-1
  - We can have an array of list.
  - Array[0] would have a list that contains neighbors of vertex 0
  - Array[1] would have a list that contains neighbors of vertex 1
  - And so on…

Adjacency list

For this graph, an adjacency list would look like:

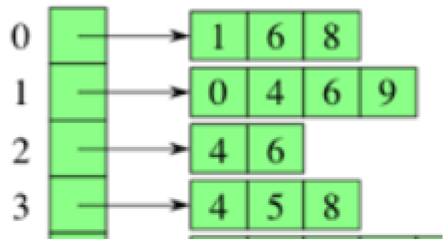
- [0]: B // for vertex A
- [1]: A, C, D // for vertex B
- [2]: B, D // for vertex C
- [3]: B, C // for vertex D



- Size:
  - (V+E)

- Access a vertex's adjacency list: O(1)

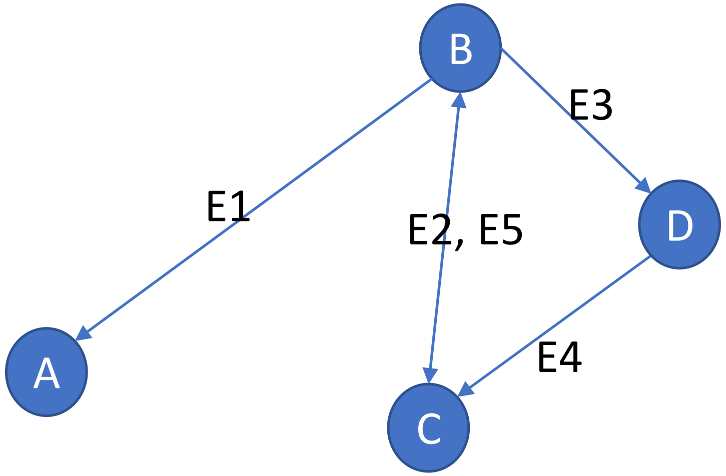
- Search for an edge (V_1, V_2) :
  - O(neighbors of V_1) OR O(neighbors of V_2)
  - For a digraph, it would be O(out-degree of V_1) or O(in-degree of V_2)



- Enumerating adjacent vertices of a given vertex V_1 :
  - O(neighbors of V_1)

Adjacency list

For this *directed* graph, an adjacency list would look like:

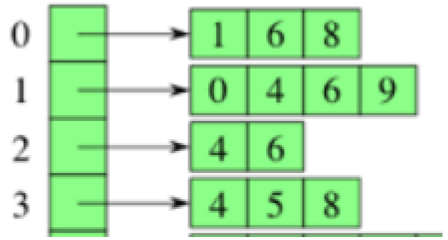


- [0]: // for vertex A
- [1]: A, C, D // for vertex B
- [2]: B // for vertex C
- [3]: C // for vertex D

- Size:
  - (V+E)

- Access a vertex's adjacency list: O(1)

- Search for an edge (V_1, V_2) in this digraph:
  - O(neighbors of V_1) or O(out-degree of V_1)



- Enumerating adjacent vertices of a given vertex V_1 :
  - O(neighbors of V_1)

Graph traversal

- Graphs can be traversed in *breadth* first or *depth* first

  - Breadth first traversal (aka BFS)
  - Depth first traversal (aka DFS)

  - Graph traversal can be a little harder to visualize than for trees.
    - trees do not have loops, which makes it easier.

  - All nodes in a tree have only one parent.

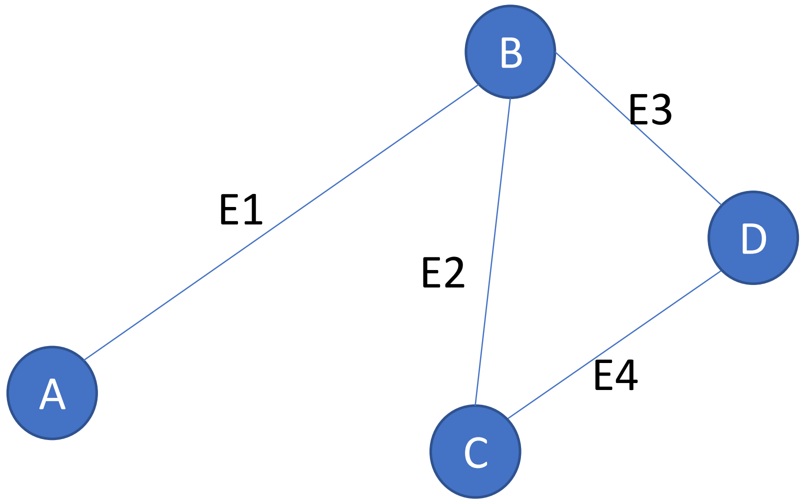
Graph traversal

- In a graph, a given node, say, ‘a’ , *can* have more than one parent.
  - This implies that we can get to that node ‘a’ in more ways than one.
  - This in turn implies that we need to keep track of nodes that we have already visited in a traversal *(so that we don’t visit the same node more than once)*.
  - We do this by marking nodes as “visited” (once we visit them)
  - Tree traversal didn’t require us to keep track of visited nodes, because u can get to a given node in only one way, which is from its **<u>one</u>** parent.

Breadth first search (aka BFS)

- BFS starts at a given vertex.

- In this graph, BFS starting at
  - A would give us:
    - A, B, C, D
  - D would give us:
    - D, B, C, A OR D, C, B, A

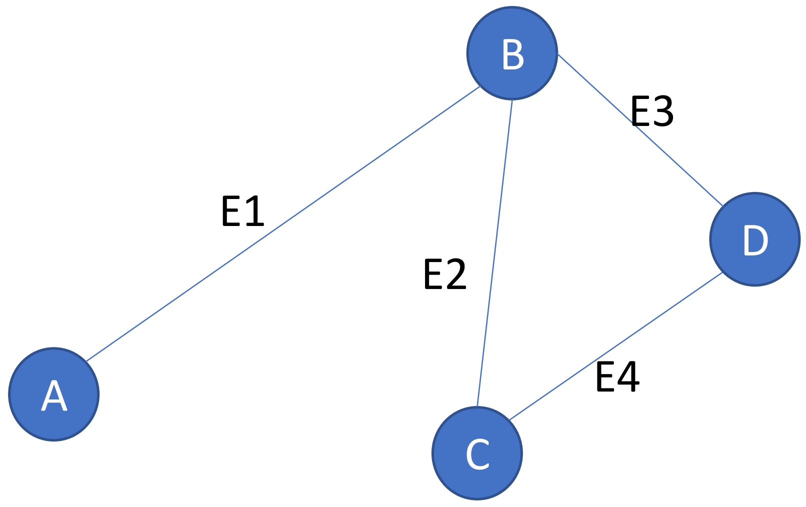


Breadth first search (aka BFS)

- Pseudocode for BFS **NOTE**: Lets only think of **undirected** graph for now

```
BFS ( Vertex v )  {
        Queue q;
        q.Push( v );  // q : d
        while ( ! q.Empty() )
        {
                Vertex a = q.Pop();  // q: empty  / b  q: c,
                Mark  a  as  visited
                Print a
                AddUnvisitedNeighbors( q, a ) // q: b, c / q: c, a
        }
}

AddUnvisitedNeighbors( Queue q, Vertex  a )  {  // add unvisited neighbors of a
        for each vertex w connected to a
                if ( w not visited yet)
                        q. Push ( w )
}
```



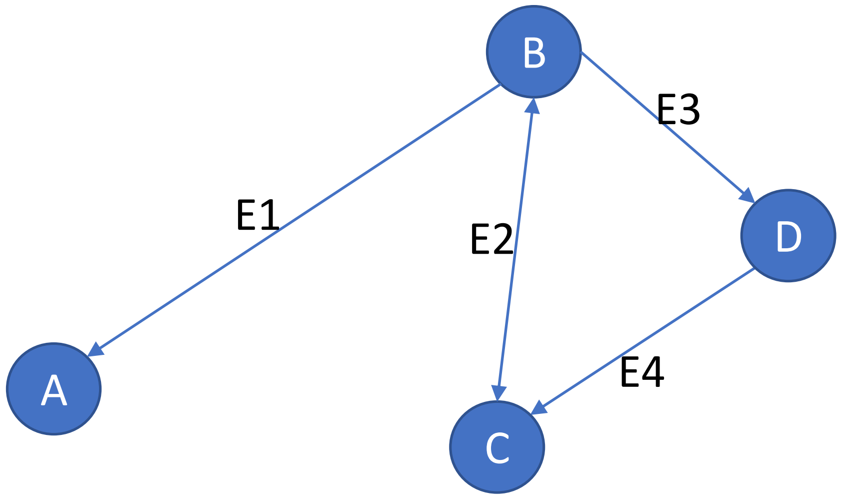
If we call BFS(A), then we have
Queue : A   Visited: empty   Printed : empty
Queue :   Visited: empty   Printed : empty

Breadth first traversal

- Pseudocode for BFS **NOTE**: Now lets think of **directed** graphs

```
BFS ( Vertex v )  {
        Queue q;
        q.Push( v );
        while ( ! q.Empty() )
        {
                Vertex a = q.Pop();
                Mark  a  as  visited
                Print a
                 AddUnvisitedNeighbors( q, a )
        }
}

AddUnvisitedNeighbors( Queue q, Vertex  a )  {
        for each vertex w having a as incoming vertex
                if ( w not visited yet)
                        q. Push ( w )
}
```

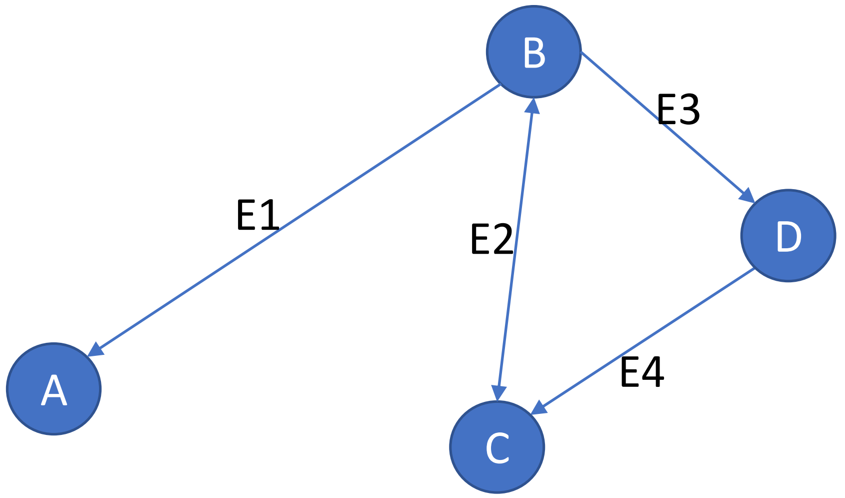


If we call BFS(A), then we have
Queue : A   Visited: empty   Printed : empty
Queue :   Visited: empty   Printed : empty

Depth first traversal

```
DFS ( Vertex v )  {
        Stack s;
        s.Push( v );
        while ( ! s.Empty() )
        {
                Vertex a = s.Pop();
                Mark  a  as  visited
                Print a
                 AddUnvisitedNeighbors( s, a )
        }
}

AddUnvisitedNeighbors(Stack s, Vertex  a )   {
        for each vertex w having a as incoming vertex
                if ( w not visited yet)
                        s.Push( w )
}
```



If we call DFS(A), then we have

Stack : A   Visited: empty   Printed : empty

Stack :   Visited: empty   Printed : empty

Depth first traversal

We can write this as a recursive function (where do we get the stack in that case ?)

DFS (Vertex v)

{

Mark v as visited

Print v

for each vertex *w* having *v* as incoming vertex

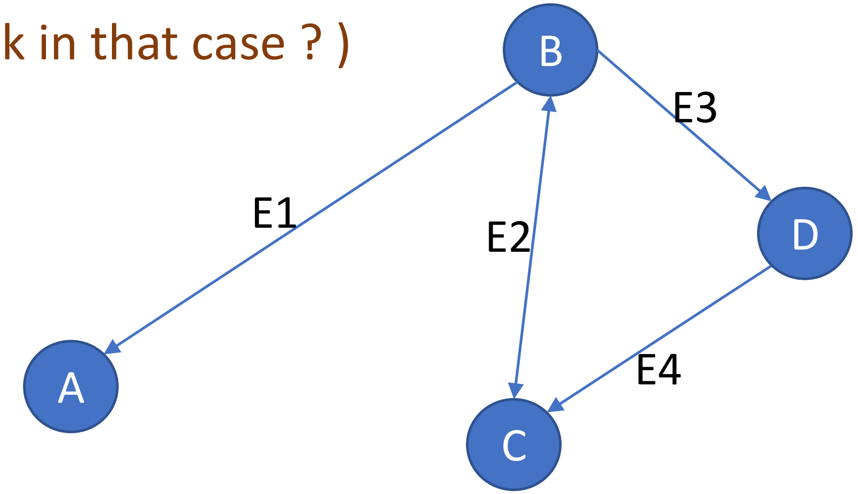
{

if (*w* not visited yet)

DFS(*w*) ← Recursive call to DFS

}

}



If we call DFS(A), then we have
Stack : A Visited: empty Printed : empty
Stack : Visited: empty Printed : empty

Recursive BFS?

- We looked at iterative and recursive versions of DFS.

- Can we do recursive BFS?

- Why does it seem easy to do recursive DFS than BFS?

Topological Sort

- A *topological sort* (new term we are looking at) of a directed acyclic graph (DAG) is essentially a "certain" ordering of its vertices.

- We will look at what that ordering really is, and look at a more formal definition, as well as an algorithm to determine this ordering.

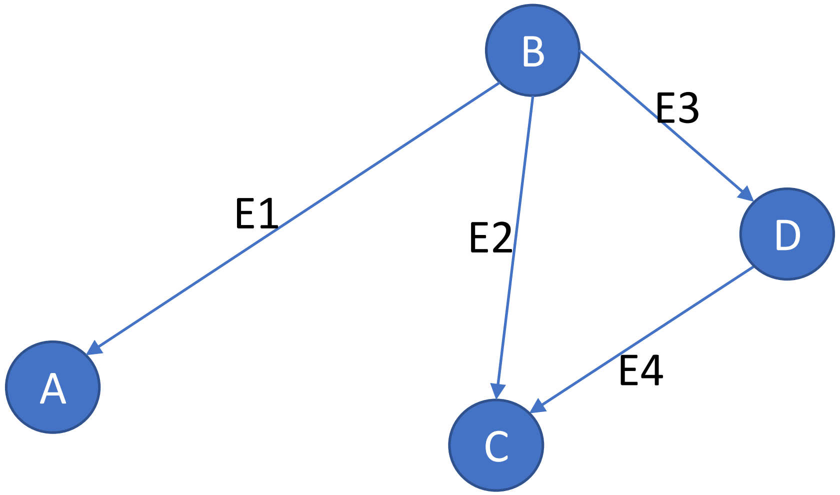
- But first, on the next slide, lets get a high level idea of it in a non-technical language, before we go all mathematical and computer sciency ☺

- gregarious

Topological Sort of a directed graph

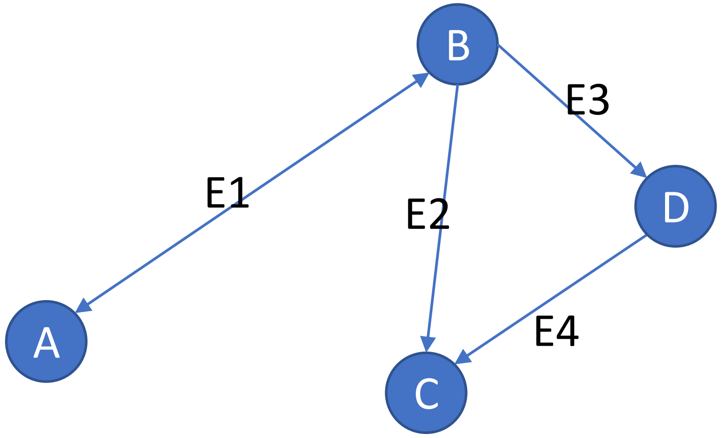
- Lets think as if the vertices in the directed graph represent some job to be done.
- And if vertex B has an edge going into vertex A, then u think of it as B needs to happen before A can happen, i.e., A is dependent on B.
  - So, in the topological sort, **B has to be placed before A**
  - And for the same reason, B has to be placed before D and C.
  - And D before C … which means **both** B and D have to be before C
  - In fact, B is NOT dependent on any vertex, so this means B has to be the **first vertex** in this sort.

- So, a topological sort here could be:
- B A D C
- OR
- B D A C



Topological Sort of a directed graph

- Now, earlier we said that this sort can happen on a directed *acyclic* graph.



- Lets look at a graph with cycles, and see what issue we would run into:
  - This graph has and edge (A,B) as well as (B,A).
  - This is a cycle A → B → A
  - This means
    - A needs to come before B in a topo sort
    - But also that B needs to come before A.
    - This is not possible to resolve, and is known as a *cyclic* dependency.
    - So this graph is not a DAG.
    - And we cannot do a topo sort on this.

Topological Sort

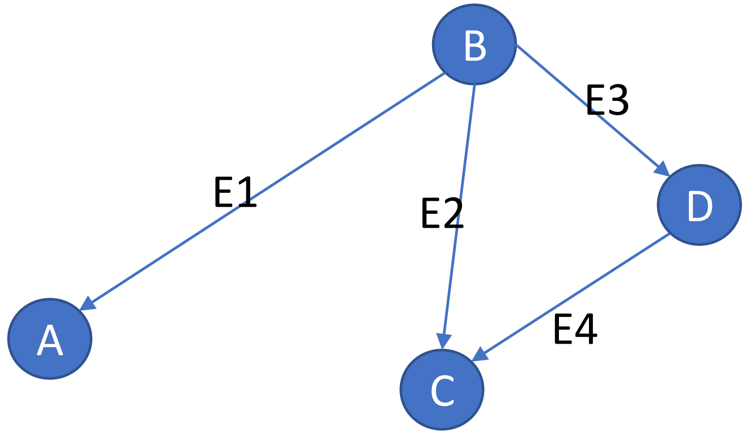
- So, to put it a little formally:

  - Topological sorting of a directed acyclic graph is where
    - a linear ordering of the vertices is created
    - such that if there is an edge (V_1, V_2),
      - then in the ordering, V_1 occurs before V_2.

- Linear ordering = layout the vertices one after another (order matters)

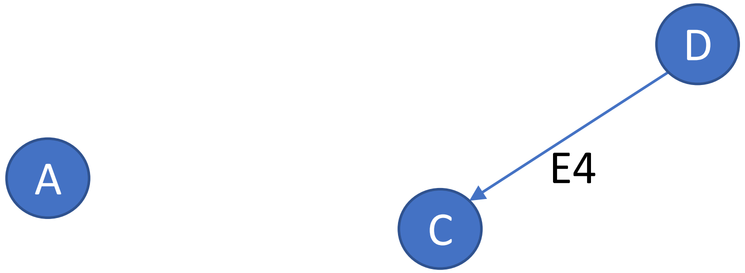
Topological sort algorithm

- Start from a vertex that has no incoming edges (B)
  - This means it has no dependencies, and can be the first one in the sort.
    - **Note**: If u cannot find such a vertex, u have a loop (cycle).
  - Print this vertex to output list
  - output list : B

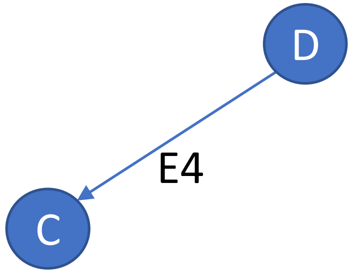


Topological sort algorithm

- So, we took out vertex B and any edges going out of B.
- Find the next vertex with no incoming edges.
  - We have A and D.
  - Lets pick A
  - Print A
  - Remove
    - Vertex A
    - And any edges going out of A (none)

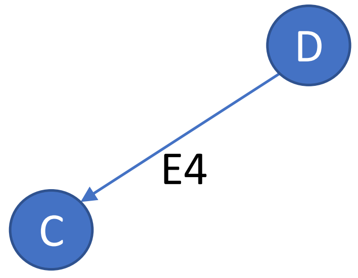


  - output list : B A



Topological sort algorithm

- So, we took out vertex A and any edges going out of A.
- Find the next vertex with no incoming edges.
  - We have D.
  - Print D
  - Remove
    - Vertex D
    - And any edges going out of D



  - output list : B A D

Topological sort algorithm

- Now we are left with only vertex C
  - Print C
  - Remove C
  - No more vertices left.
  - We are done.

  - output list : B   A   D   C

- Lets take a look at the algorithm again, this time with the complexity in mind.
- Our first step was:
  - Start from a vertex that has no incoming edges

- In order to achieve this, we need to have these:
  1. computed the in-degree of all vertices in the graph.
     - This would be a one time step (initialization)
     - O (E)
  2. Find a vertex with in-degree zero
     - This is a linear search in the array computed in step 1 above.
     - This means a linear search in an array of size V (number of vertices)
     - And this would need to be done V times
     - O(V^2)
  3. Reduce the in-degrees of adjacent vertices
     - This is where we removed the edges going out of the current selected vertex
     - O(E)
  4. Mark vertex
     - O (V)

- So, overall complexity here is O (V^2) + O (E) … which is quadratic, and hence not that good.

- The step that causes the quadratic complexity is the search for vertex with in-degree 0

- So, lets see if we can improve that step.
- During initialization, we add the following:
  - Create a queue. O (1)
  - Initialize it with in-degree 0 vertices. O (V)
- In the step where we reduce the in-degrees of vertices, if the in-degree becomes 0, push that vertex onto this queue.
  - Pushing onto queue is O (1)
  - Do this for a total of V vertices.
    - O (V)

- So, we reduced O (V^2) to O (V), for a total time complexity of
  - O (V) + O (E)

LAB

- Write a function that will create one (or both) of the following graphs (adjacency list shown)
  - a: b c
  - b: d e
  - c: f g
  - d: h i
- **OR**
  - a: b
  - b: c
  - c: d c1
  - d: d1
  - d1: d2
  - c1: c2

- Now write functions for:
  - Depth first traversal : print vertices
  - Breadth first traversal: print vertices

Further topics

- Spanning Tree
  - Minimum Spanning Tree
    - Kruskal’s algorithm
    - Prim’s algorithm

- Shortest path algorithms
  - Dijkstra’s algorithm