# IMPORTANT FOR GRAPHS

Wednesday, 1 June, 2022 15:43

How to determine what algorithm need to be used?

- Minimum spanning tree algo (Prims or Krukal) can only be run on undirected graphs.
- If it needs distance Dijkstra/Floyd-Warshall/Bellman-Ford.
- The only algo left is BFS/DFS/Khan.

Properties of graph:

- Weighted <u,v,w> vs unweighted <u,v>
- Directed <u,v> vs undirected <u,v> <v,u>
- Cyclic and acyclic
- Connected (all the vertex are connected) and strongly connected (for every vertex can travel to any vertex on the graph)
- Identifying the above, use case and analysing complexity.

Given a graph:
Unweighted and single source (BFS/DFS)
Directed, unweighted and want all pair (Floyd-Warshall)
Positive edges, all pair (Floyd-Warshall)
Directed, negative edges, single source (Bellman-Ford)

Graph Complexity

Tuesday, 31 May, 2022 01:58

## Graph
### BFS Implementation

MONASH University

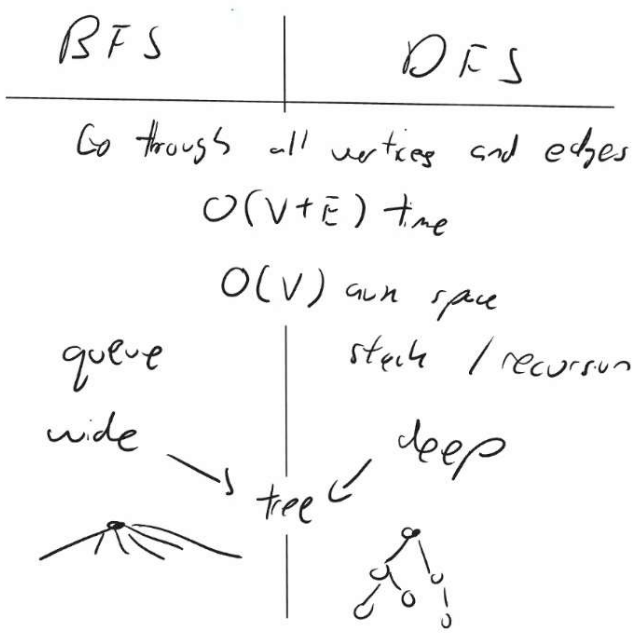
- Complexity?
  - Time is O(V+E)
    - Each vertex is visited once
    - Each edge is visited twice
      - For each <u,v> we visit from u and also from v
  - Space is O(V+E)
    - V maximum for the discovered queue
    - E to stored all of the edges (adjacency list)
  - But don't we need to check the discovered queue for each vertex v?
    - O(V) search through the queue?
    - NO! Implement a Node class with self.discovered = True/ False

152

## Graph
### DFS Implementation

MONASH University

- Complexity?
  - Time is O(V+E)
    - Explanation same as BFS
  - Space is O(V+E)
    - Explanation same as DFS




Dijkstra is O(E log V)

# Directed Acyclic Graph (DAG)

Tuesday, 31 May, 2022 18:14

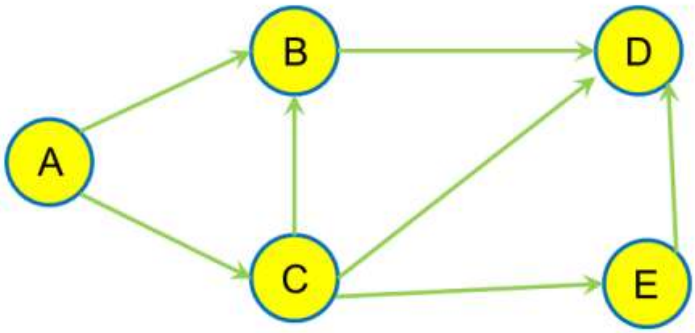
A graph that is directed and has no cycles on the graph.

## Directed Acyclic Graph (DAG)

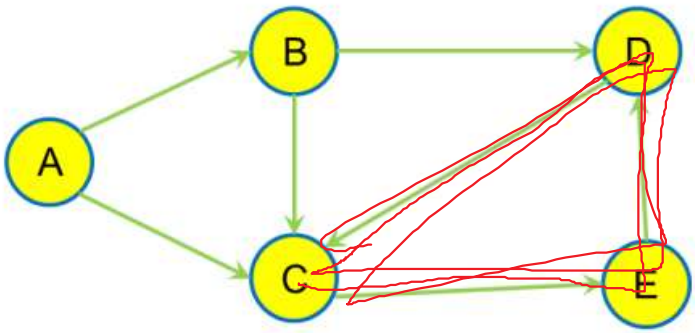
What is it?

MONASH University

- Which graph is a DAG?
  - And where is the cycle?




Graph 1 ✓




Graph 2 ✗

# Topological Sort

Tuesday, 31 May, 2022 18:39

## Topological Sort
### Ordering of Vertices

MONASH University

- A topological sort
  - Permutation of vertices in a DAG
  - Vertex U will appear before vertex V if we have edge <U,V>
    - U<V
  - Vertex U will appear before vertex W is we have edge <U,W>
    - U<W
  - But if we don't have edge <V,W> then V and W are of the same order
    - V==W
- So we have a DAG of your units
- Topological sort of this DAG gives you the order of units to take!

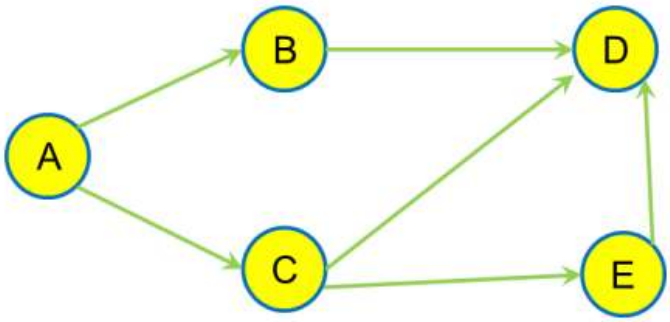
30

## Topological Sort
### Ordering of Vertices

MONASH University

- Which one of these are not a valid topological sort of the DAG?
  1. A, B, C, E, D ✓
  2. A ,C, B, E, D ✓
  3. A, C, E, B, D ✓
  4. A, B, E, C, D ✓

E < C ✗




32

- Topological sort can be done via
  - Kahn's algorithm
  - A modified DFS

Kahn's Algo

## Kahn's Algorithm
For topological sort

MONASH University

- What is the concept like?
  - Start with vertices without incoming edges
  - Delete all outgoing edges from the vertex
  - Add in vertices without incoming edges
  - Repeat!

## Depth-First Search (DFS)
Modified for topological sorting

MONASH University

- Here's how we modify with a stack!

```
28  def dfs_topological(vertex_u):
29      # start for result
30      stack = []
31      # run DFS
32      vertex_u.visited = True
33      for edge in vertex_u.edges:
34          if edge.vertex_v.visited == False:
35              dfs_topological_aux(vertex_v,stack)
36      # output
37      print(stack)
38
39  def dfs_topological_aux(vertex_u,stack):
40      vertex_u.visited = True
41      for edge in vertex_u.edges:
42          if edge.vertex_v.visited == False:
43              dfs_topological_aux(vertex_v)
44      # add to stack
45      stack.push(vertex_u)
```

81

Modifying DFS for topological sorting, only need to add a stack. Complexity would be DFS which is O(V+E) since we have only added a stack.

# Printout

Tuesday, 31 May, 2022 18:11

## Graphs

**Representation**

Adjacency Matrix

- What info can I store?

- True/False or 1/0 for unweighted *(check if edge exists)*
- Edge Weight for unweighted

- Time and space complexity?

- **Time → O(1)** to check edge exists *(since we can just check based on row and column number)* / **O(V)** to traverse all adjacent vertices.
- **Space → O(V^2)** since our matrix size depends on number of vertices, 1 vertex can connect to N-1 number of vertices.

Adjacency List

- What info can I store?

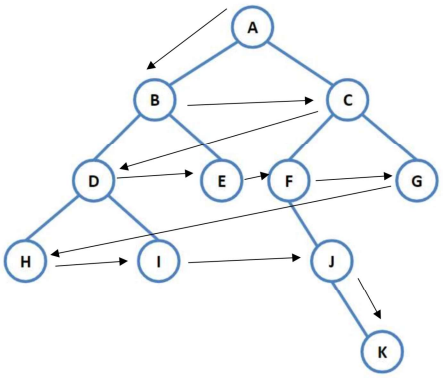
- A list of Vertex Objects under Graph class, each Vertex has a list of Edge Objects under Vertex class. Each Edge contains values for u, v, w.
- u → starting vertex, v → destination object, w → weight.

- Time and space complexity?

- **Time → O(X)** check edge exists if all sorted, and to retrieve all adjacent vertices of vertex *(X = number of adjacent vertices).*
- **Space → O(V+E)** since all vertices will be stored in the array and the length of edge list for each vertex depends on whether the edge connects the vertex or not.

## BFS & DFS

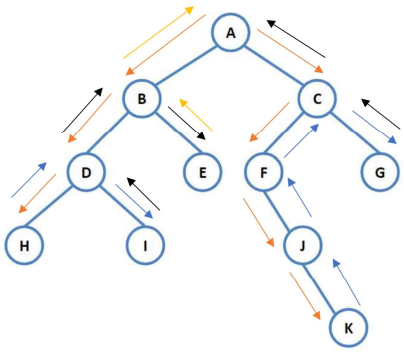
**Breadth-First Search**




**Time Complexity → O(V+E)**
- Each vertex visited once
- Each edge visited twice *(Go to u first, then back to v)*

**Space Complexity → O(V+E)**
- V maximum for discovered queue
- E to store all edges
- Basically, your adjacency list

**Depth-First Search**



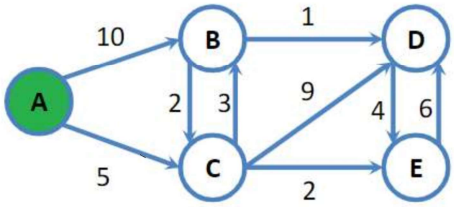

**Time Complexity → O(V+E)**
- Each vertex visited once
- Each edge visited twice *(Go to u first, then back to v)*

**Space Complexity → O(V+E)**
- V maximum for discovered queue
- E to store all edges
- Basically, your adjacency list

**Notes:**
- These two graph algorithms are **IMPORTANT** before you start learning any new graph algorithms in FIT2004
- A lot of the algorithms are basically just modifications from either BFS or DFS.
- Exam will not touch so much about this.

## Dijkstra

**How does Dijkstra work?** Important for Exam and Understanding




1. **(Only do once) Initialize all distances to each vertex from target source to infinite, with target source vertex having value 0**
2. **Update all edges into list (if value of vertex's distance in list not infinite, compare and get minimum value to be updated)**
3. **Find smallest distance from most recently visited vertex to discovered vertex.**
4. **Repeat steps 2 and 3 until all vertices are visited.**

**What is needed to be modified?**

- BFS Algorithm
- Use Min-Heap Priority Queue as Data Structure

## Dijkstra

```
1  discover_queue = MinHeap()
2  discover_queue.append([source,0])
3
4  while discover_queue is not empty:
5      u = discover_queue.serve()
6      u.visited = True
7      for each <u,v,w> in u.edges:
8          if v.visited = True:
9              pass
10         else:
11             if v.discovered = False:
12                 discover_queue.append([v, u.distance+w])
13                 v.discovered = True
14             else:
15                 if v.distance > u.distance+w:
16                     discover_queue.update(v, u.distance+w)
17                     v.discovered = True
```

Line 4 → O(V)
Line 6 → O(log V) since we are now using Min Heap
Line 7 → O(V)
Line 17 → O(log V) since we are now using Min Heap

**Time Complexity** → O(V^2 log V) = O(E log V), since in dense graph → E approx. equals to V^2

*Note: can make it O(E + log V) with Fibonacci Heap → You will learn this in FIT3155

**Single Source, Single Target**

- Here, we want to find the vertices that we have traversed to get to the target.
- We then use backtracking to record the vertex
- Store vertex info into an instance variable, then when iterated through, reverse to get the values.

**Why Important?**

- Will come out in assignment (implementation) and exam (tracing)
- One of the first algorithms to meet in order to understand weighted graphs and how they work.

**Why are these important?**

- **BFS & DFS** → Basic Fundamentals of all your Graph Algorithms
- **Dijkstra** → Most commonly used Graph Algorithm

**What's Next?**

- **Directed Acyclic Graph**
- **Floyd-Warshall & Bellman Ford**

Past Year Questions

Wednesday, 1 June, 2022 02:47

# Graph Representation

## Question 16

For each of the following operations, determine its time complexity.

In this question, 3 Marks

- *V* refers to the number of vertices in the graph
- *E* refers to the number of edges in the graph
- *N(x)* refers to the number of neighbors of vertex-*x*.

Assume that in the adjacency list representation, the interior lists are unsorted.

| | |
|---|---|
| Determining if an edge from *u* to *v* exists in an adjacency matrix | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |
| Determining if an edge from *u* to *v* exists in an adjacency list | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |
| Finding all neighbors of *u* in an adjacency matrix | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |
| Finding all neighbors of *u* in an adjacency list | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |
| Performing a complete BFS traversal in an adjacency matrix | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |
| Performing a complete BFS traversal in an adjacency list | · O(V+E) · O(N(u)) · O(V^2) · O(1) · O(V) |

```
You are working in the X that uses a very large number of approval. For almost every approval
there is a set of other approvals that have to be submitted before this one can be submitted.
There are even approvals to request other approvals.

The set of approvals used has grown out of control over decades. It has now become so complex
and convoluted that some suspect that there are approvals that can never be submitted because
they require another approval to have been previously submitted, which in turn requires the
very approval that we were trying to submit in the first place!

Your task is to write an algorithm that can test whether this is actually the case or whether
the requirements for approval are still well-defined and feasible.

What graph is this? this needs to be a directed acyclic graph (DAG) [1 marks]
What algorithm can you use to determine this property? DFS, kahn's algorithm (BFS) [1 mark]
    - give pseudocode [2 mark]
How can we store the graph efficiently? adjacency list [1]
Complexity? O(V+E) [2 mark]

7 marks
```

Markdown

What graph is this?
- Directed acyclic graph because it can't have a cycle and it needs to have direction.

What algorithm can you use to determine this property?
- DFS, kahn's algorithm (BFS modification)
- When doing the depth first search, we keep track of the visited vertex. Each vertex should be visited only once since there shouldn't be a cycle. Once a vertex is visited twice, we have found that there is a cycle in graph G.

How can we store the graph efficiently?
- Adjacency list

Complexity?
- O(V+E) BFS or DFS

**Kahn's Algorithm**
For topological sort

MONASH University

- What is the concept like?
  - Start with vertices without incoming edges
  - Delete all outgoing edges from the vertex
  - Add in vertices without incoming edges
  - Repeat!

If found another vertex that is in discovered list then we have found a cycle. Kahn's algorithm can't go through all of the vertices so it can detect the cycle.