



Competitive
Programming and
Mathematics
Society

Programming Workshop #3

Topological Sort and Tree Traversal

David and Ryan

Scheduling problem



CPMSOC

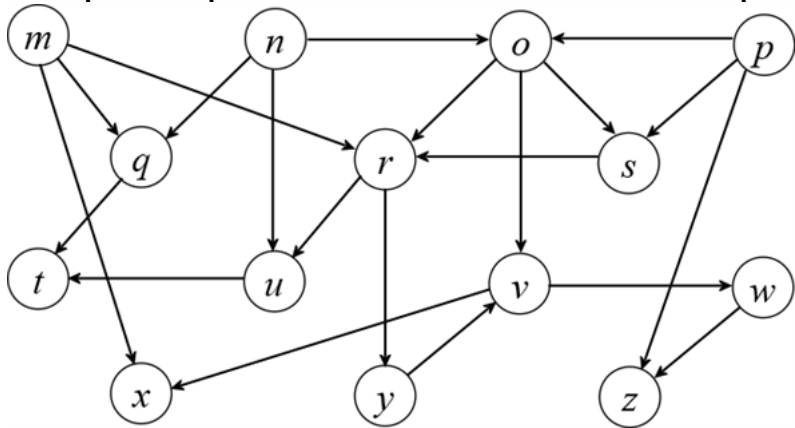


Imagine following **scenario**: You are a UNSW compsci student who is planning for your course schedule, with following constraint

- As a student, you want finish your degree as early as you can!
- As we probably know, certain courses are prerequisite for certain other courses, for example, in order to do COMP2521(Data Structure and Algorithm), we have to complete COMP1511(Programming Fundamentals)
- You are a smart genius, but UNSW got a new policy, you can only take one Comp course each term.
- And because of you are a smart genius, you want take all courses in the next diagram!
- The question would be, can we get a sequence of course that do not violate any "prerequisite" requirement?

Scheduling problem

Hmmm, how should we solve this type problem? What if the prerequisite becomes more complicated?



Imagine each vertex represent a course

Scheduling problem

Before we dive into our "Topological Sort Algorithm", let's generalize what is a scheduling problem. The scheduling problem has to do with arranging for completion of a set of tasks, under the constraint

- When this jobs need to be performed before,
- Might involves function of resource consumed, such as time or other variable.
- **Precedence constraint**, which specify that certain task must be performed before others.

Dive into Topological sort

Back to our original problem, we would observe,

- There is no deadline for our course example.
- All courses takes a term so it's the same, there is no such additional function.
- But we do have **Precedence constraint**, which is a classic **Topological sort** situation.

Before we dive into Topological sort

We still need to refresh some graph basic.

A graph $G=(V,E)$ is a general data structure that consist

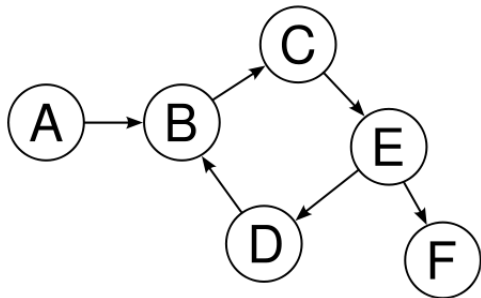
- Set of vertices, (e.g. A set of tasks)
- Set of edges, that each connect a pair of vertices. (e.g. The relation between two tasks)

A graph can be either **undirected**, or **directed**. This refers to the type of the edges.

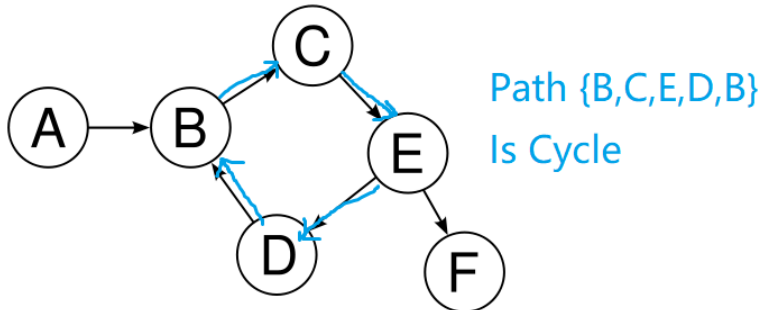
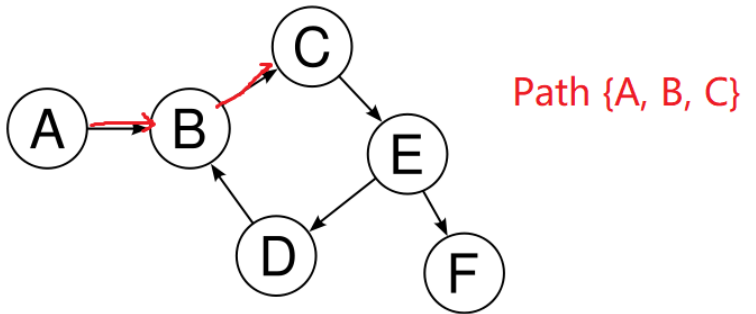
- In an undirected graph, edge is bidirectional. (e.g. Each vertices represent a city, each edge represent a highway)
- In a directed graph, edge is one-way. (e.g. Each vertices represent a phone number, each edge represent a phone call)
- In our example, we would need a directed graph, such that each vertices represent a task, and each edge represent precedence constraint.

Before we dive into Topological sort

- A **path** is a sequence of vertices connected by edges.
- A **cycle** is a path with at least one edge whose first and last vertices are the same.



Acyclic graphs are graphs without cycles.



Dive into Topological sort

Then for a classical Topological sort application, we would need a directed, **acyclic** graph, or a DAG or Digraph as it is sometimes called.

It is obvious why we need a directed graph. However might be less intuitive why do we need a **Acyclic** graph , which is equivalent to with no cycle.

Example

Suppose there exist a cycle C in graph G , then the cycle would be

$$C = \{V_1, V_2, V_3, \dots, V_1\}$$

Then we encounter a contradiction, no matter which vertices we pick as our start, it's **precedence-constraint** can not be satisfied.

Revision on Depth First Search

As it turns out, we can use depth-first search to perform both cycle-detection and a topological sort. Isn't that awesome?

We should have a revision, on the depth-first search algorithm. For the simplicity sake, we would assume that the directed graph G only has one component.

Algorithm 1: DFS(G, V)

Input: DiGraph G , vertices V

Output: depend on the problem

$V.status = VISITED$;

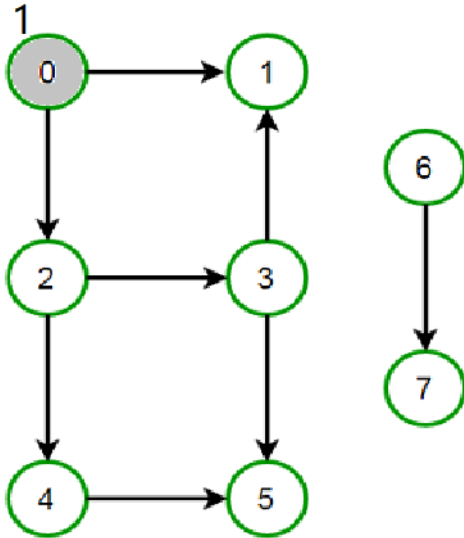
```
for all  $(V, U) \in G.edges()$  do  
    if  $U.status \neq VISITED$  then  
        | DFS( $G, U$ );  
    end  
end
```

Introducing state

Depth first search explores edges out of most recently discovered vertex v , that still haven't been visited. Once all edges from v have been visited, the search will "backtrack" to explore edges from the vertex that discovered v .

From the description above, at an arbitrary step during the execution of DFS algorithm, each vertex has three states,

- Un-visited. As the name suggests, the vertex has not been discovered yet.
- Visiting. Vertex has been visited, but not finished.
- Finished. All adjacent, undiscovered vertices have been examined completely, DFS would "backtrack" at this point.



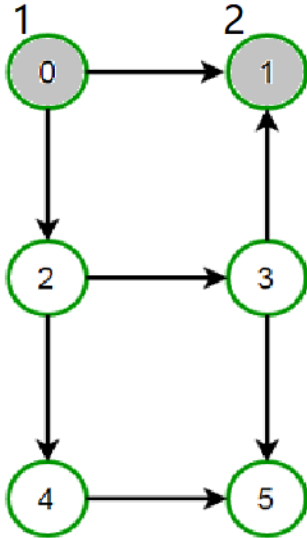
Un-visited



Visiting



Finished



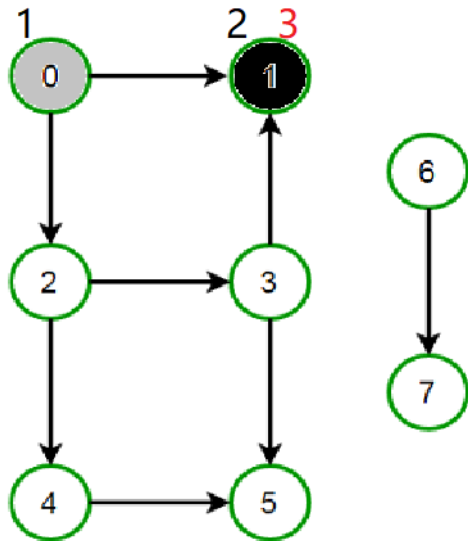
Un-visited



Visiting



Finished



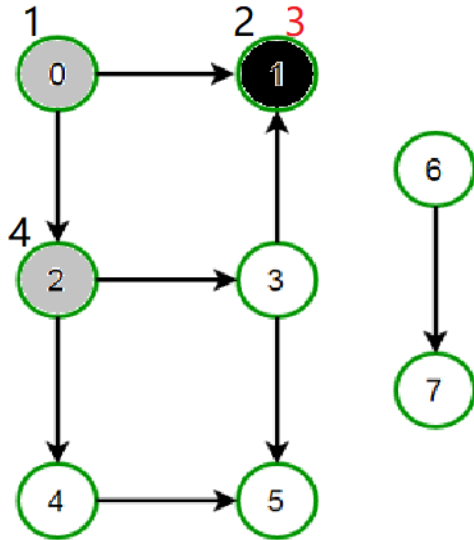
Un-visited



Visiting



Finished



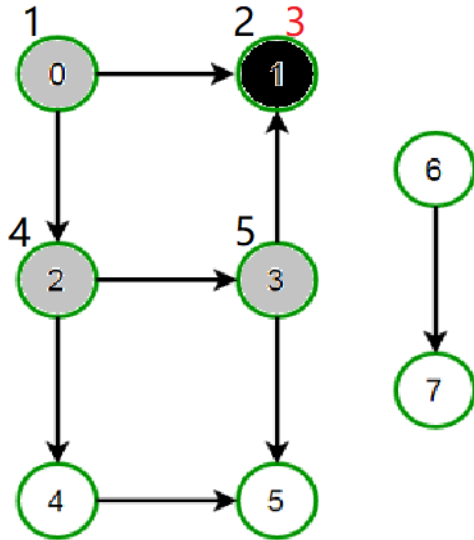
Un-visited



Visiting



Finished



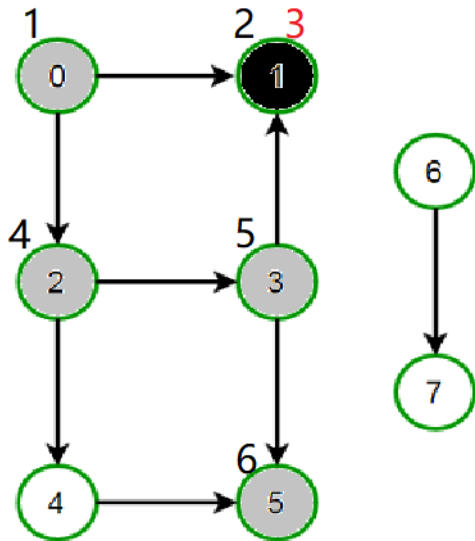
Un-visited



Visiting



Finished



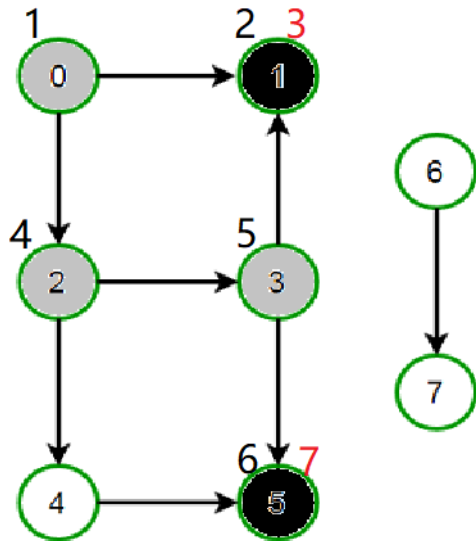
Un-visited



Visiting



Finished



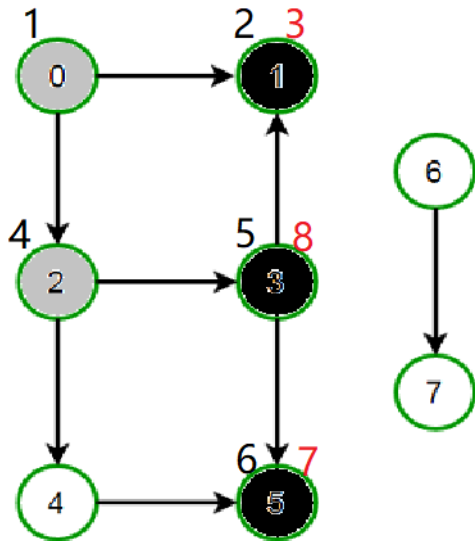
Un-visited



Visiting



Finished



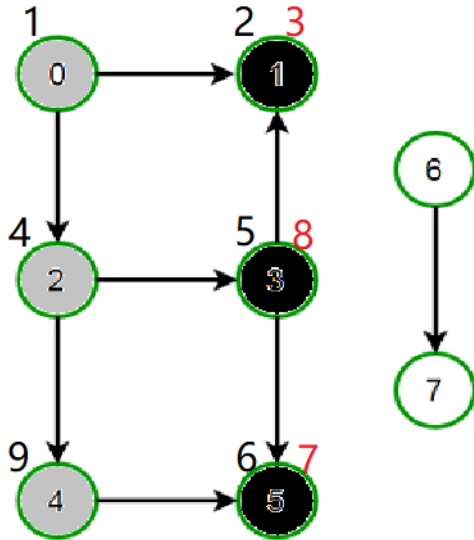
Un-visited



Visiting



Finished



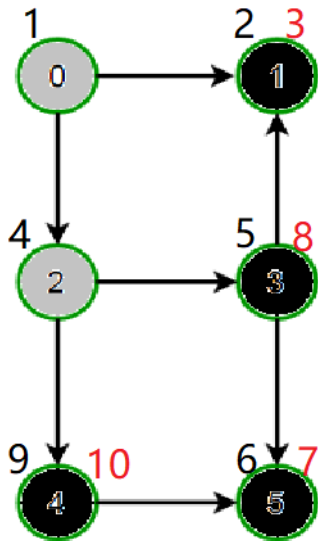
Un-visited



Visiting



Finished



Un-visited

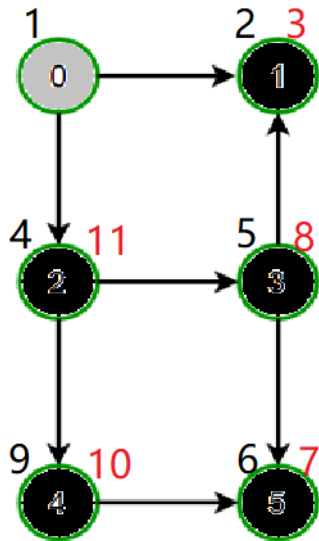


Visiting



Finished





Un-visited

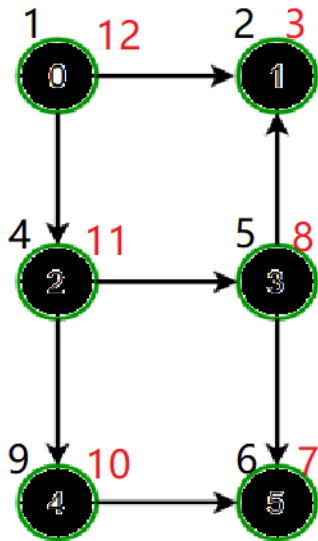


Visiting



Finished





Un-visited



Visiting



Finished

Implementation of state

These states will provide important information about the structure of the graph. To implement the state, we can use a global variable represent the times, and record it on vertex k whenever it finishes on k .

The structure of above algorithm is almost identical to the one we learned from COMP2521. The C++ implementation will be on the next slide.

Implementation of DFS topsort



```
vector<int> outgoingEdges[MAX_V];
vector<int> top_sort;
int state[MAX_V];
// = 0 unvisited
// = 1 visiting
// = 2 finished

void findTopsort(int node) {
    if (state[node] == 1) {
        // ahhhhhhh theres a cycle
        // terminate because topsort is impossible
        return;
    }
    if (state[node] == 2) {
        return;
    }
}
```

Implementation of DFS topsort



```
state[node] = 1;
for (int child : outgoingEdges[node]) {
    findTopsort(child);
}
state[node] = 2;
// node is added to topsort when all dependancies are already added
top_sort.push_back(node);
}

// make sure topsort covers all components of the graph
for (int node = 1; node <= V; node++) {
    dfs(node);
}

reverse(top_sort.begin(). top_sort.end());
```

Implementation of Kahn's topsort



```
vector<int> outgoingEdges[MAX_V];
int num_prereqs[MAX_V];

for (int node = 1; node <= V; ++node) {
    for (int child : outgoingEdges[node]) {
        num_prereqs[child]++;
    }
}

vector<int> starting_nodes;
for (int node = 1; node <= V; ++node) {
    if (num_prereqs[node] == 0) {
        starting_nodes.push_back(node);
    }
}
```

Implementation of Kahn's topsort



```
vector<int> top_sort;
while (!starting_nodes.empty()) {
    // extract an arbitrary starting node
    int curr_node = starting_nodes.back();
    starting_nodes.pop_back();

    for (int child : outgoingEdges[curr_node]) {
        num_prereqs[child]--;
        if (num_prereqs[child] == 0) {
            starting_nodes.push(child);
        }
    }
}
```

Implementation of Kahn's topsort



CPMSOC



```
if (top_sort.size() < V) {  
    // top sort doesnt contain all nodes  
    // => some nodes still have prereqs  
    // => theres a circular dependancy  
    // => graph has a cycle  
    // ahhhh! stop here because our topsort isn't valid - there is no valid topsort  
}
```

Consider:

- Which do you find easier to understand? Which do you find easier to implement?
- Time complexity analysis: $\mathcal{O}(V + E)$
- Does this still work when the graph has many components?

Let's solve a problem

<https://leetcode.com/problems/course-schedule-ii/>

210. Course Schedule II

Medium



7094



244



Add to List



Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you **must** take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return *the ordering of courses you should take to finish all courses*. If there are many valid answers, return **any** of them. If it is impossible to finish all courses, return **an empty array**.

Example 1:

More challenging problems

<https://leetcode.com/problems/parallel-courses-iii/>

<https://codeforces.com/contest/510/problem/C>

<http://www.usaco.org/index.php?page=viewproblem2&cpid=838>



CPMSOC



Attendance form :D

<https://forms.gle/UwwCBjbjgESDv2qE8>



CPMSOC



Further events

Please join us at:

- Next weeks Maths workshop! (Tuesday 12-2)
- Guest speaker David Angell!
- Programming weekly blog posts!



CPMSOC

