Competitive Programming and Mathematics Society

Programming Workshop #5

Topological Sort and Tree Traversal

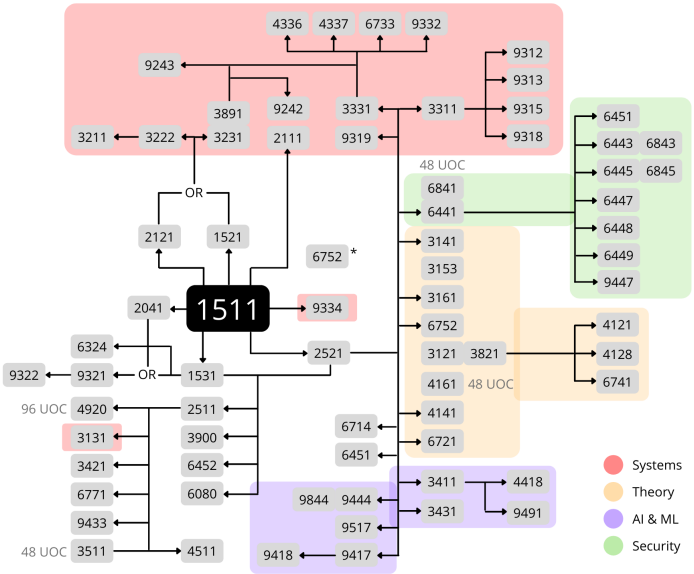
David and Ryan

Scheduling problem

CPMSOC

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

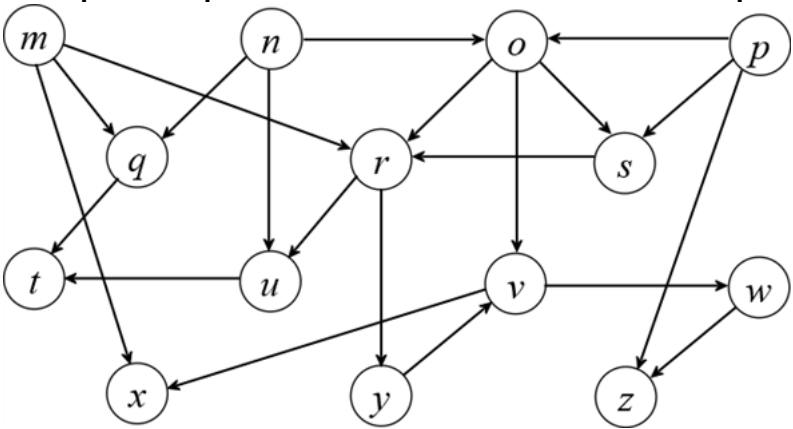
- As a student, you want finish your degree as early as you can!
- As we probably know, certain courses are prerequisite for certain other courses, for example, in order to do COMP2521 (Data Structure and Algorithms), we have to complete COMP1511 (Programming Fundamentals)
- You are a smart genius, but UNSW has 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?


4336
4337
6733
9332
9243
9312
9313
3891
9242
3331
3311
9315
3211
3222
3231
2111
9319
9318
6451
6443
6843
48 UOC
6445
6845
6841
6447
OR
6441
2121
1521
6448
3141
6752 *
6449
3153
9447
3161
2041
1511
9334
6752
4121
6324
2521
3121
3821
4128
9322
9321
OR
1531
4161
48 UOC
6741
96 UOC
4920
2511
4141
3131
3900
6714
6721
3421
6452
6451
6771
6080
3411
4418
9844
9444
3431
9491
9433
9517
48 UOC
3511
4511
9418
9417
Systems
Theory
AI & ML
Security

Scheduling problem

CPMSOC

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



(Imagine that each vertex is representing a course.)

Dive into Topological sort

CPMSOC

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

CPMSOC

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)

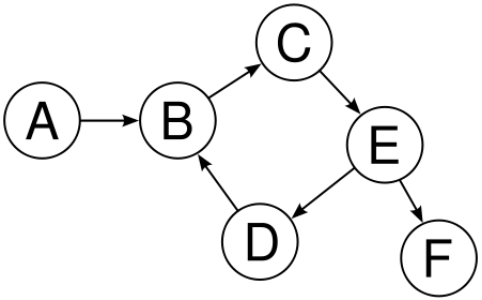
A graph can be either undirected, or directed. This refers to the type of the edges.

- 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.

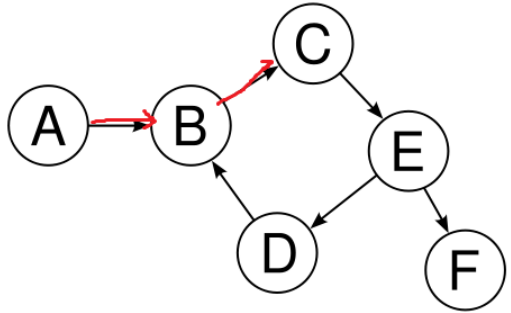
Before we dive into Topological sort

CPMSOC

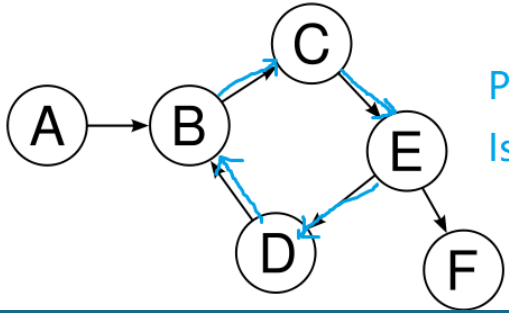
- A path is a sequence of vertices connected by edges.
- A cycle is a path with at least one edge whose first and last vertices are the same.



Acyclic graphs are graphs without cycles.



Path {A, B, C}



Path {B,C,E,D,B}

Is Cycle

Dive into Topological sort

CPMSOC

Then for a classical Topological sort application, we would need a directed, acylic 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 Acylic graph , which is equivalent to with no cycle.

Example

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

$$C = fV_1; V_2; V_3; \quad ; V_1 g$$

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

CPMSOC

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 != VISITED* **then**
        DFS(G, U);
    end
end

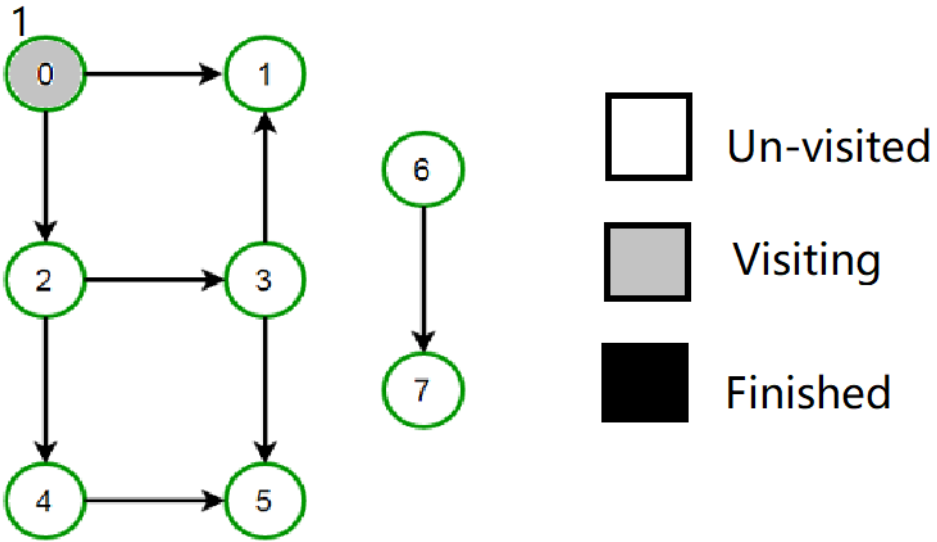
Introducing state

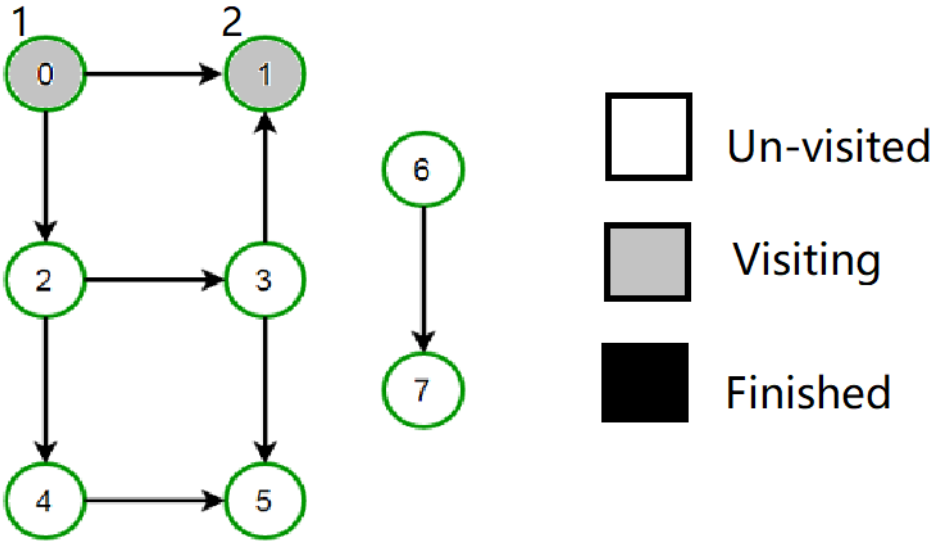
CPMSOC

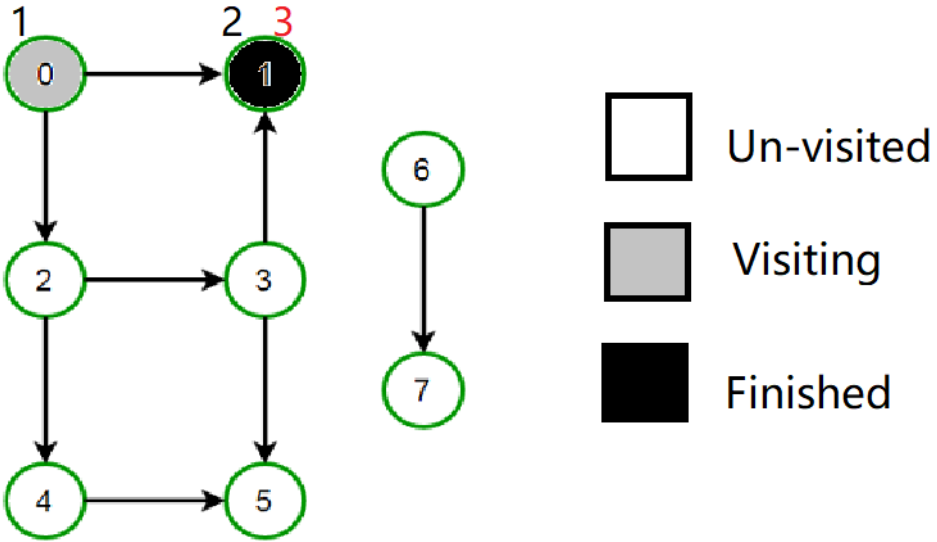
Depth first search explores edges out of most recently discovered vertex v, that still haven't being visited. Once all edges from v being visited, the search will "backtrack" to explore edges from the vertex discovers v.

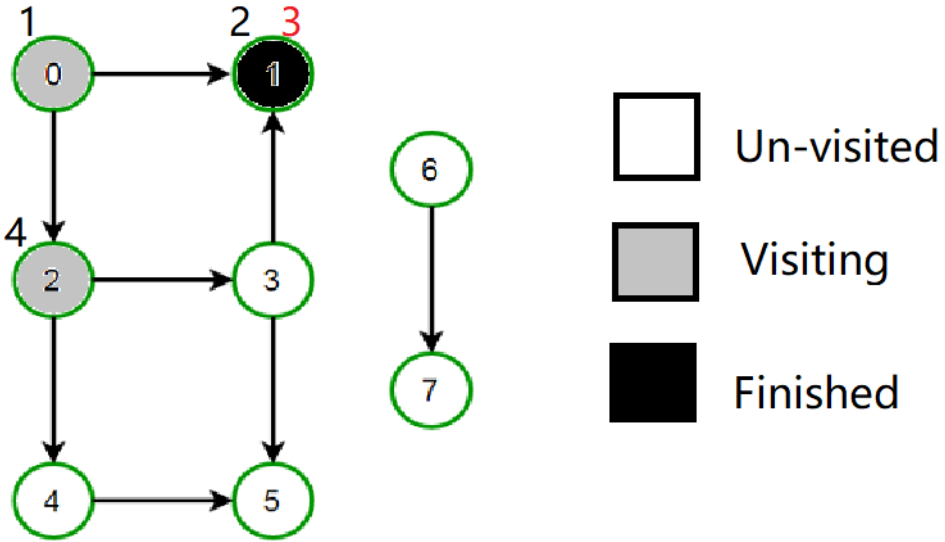
From description above, at an arbitrary step during the execution of DFS algorithm, each vertex have three states,

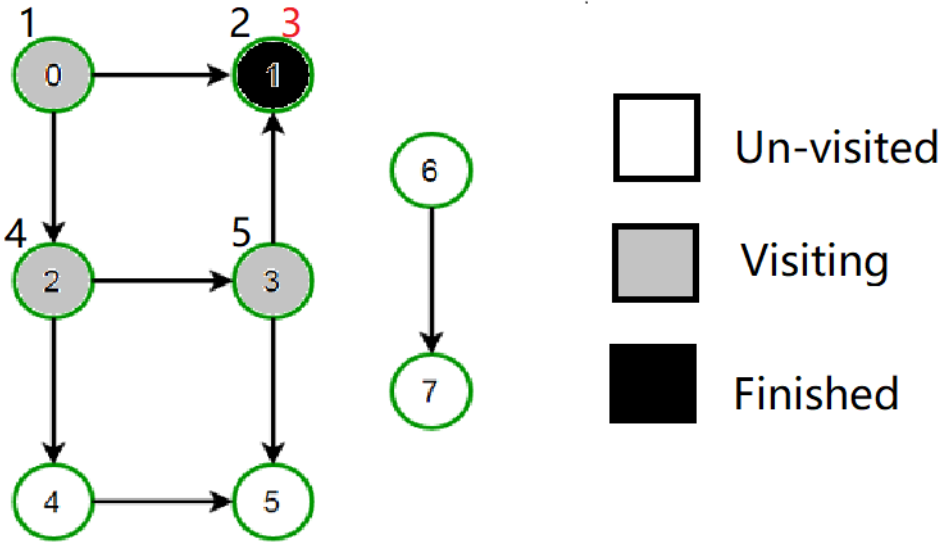
- Un-visited. As name suggests, the vertex not being discover yet.
- Visiting. Vertex have being visited, but not finished.
- Finished. All adjacent, undiscovered vertex being examined completely, DFS would "backtrack" at this point.

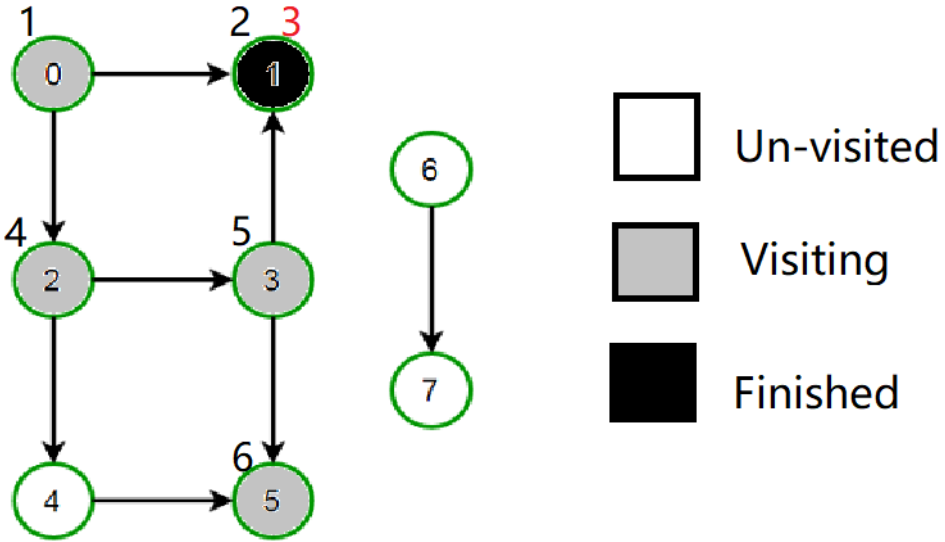

1
0
1
2
3
4
5
6
7
Un-visited
Visiting
Finished

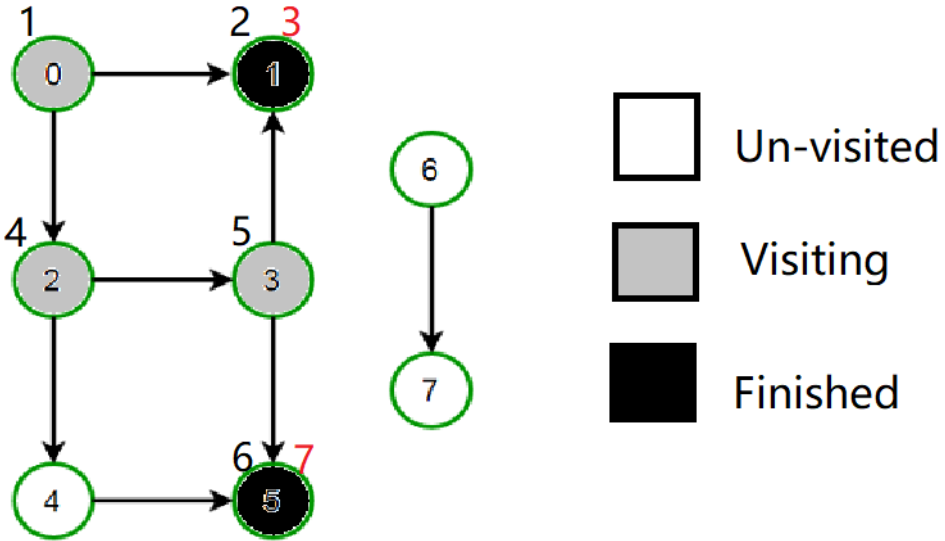

1
2
0
1
2
3
4
5
6
7
Un-visited
Visiting
Finished

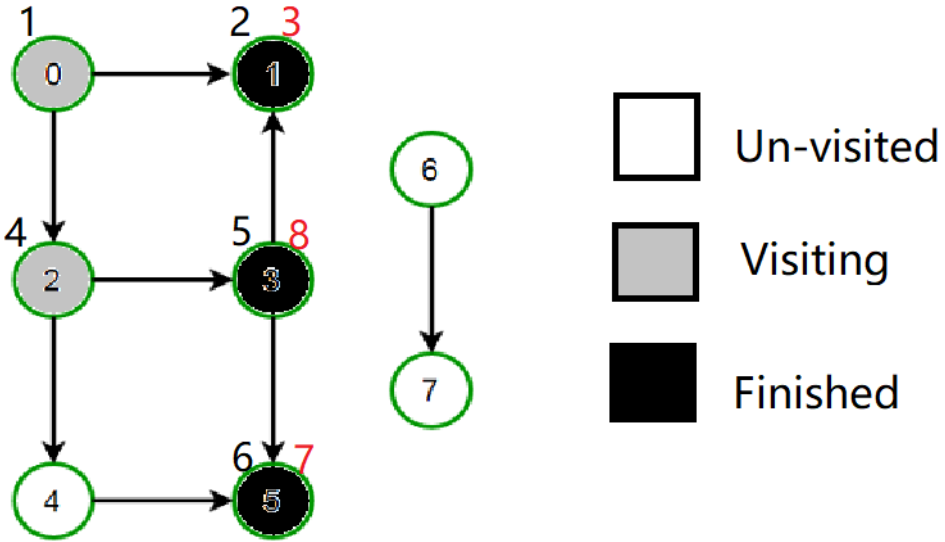

1
2
3
0
1
2
3
4
5
6
7
Un-visited
Visiting
Finished

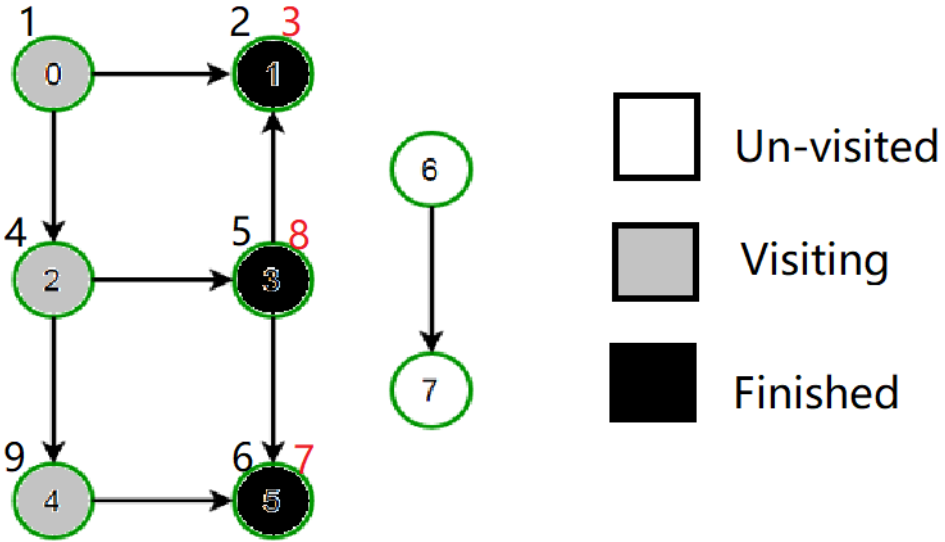

1
2
3
0
1
4
2
3
4
5
6
7
Un-visited
Visiting
Finished

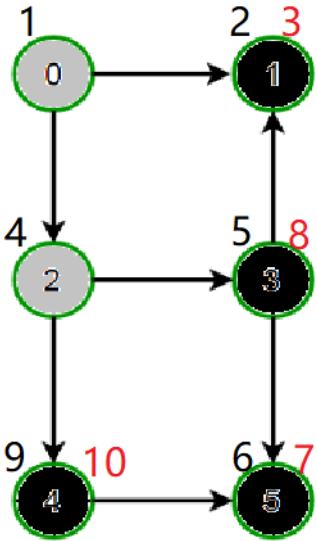

1
2
3
0
1
4
5
2
3
6
7
4
5
Un-visited
Visiting
Finished


1
2
3
0
1
6
4
5
2
3
7
6
4
5
Un-visited
Visiting
Finished


1
2 3
0
1
4
5
2
3
6
7
6 7
4
5
Un-visited
Visiting
Finished

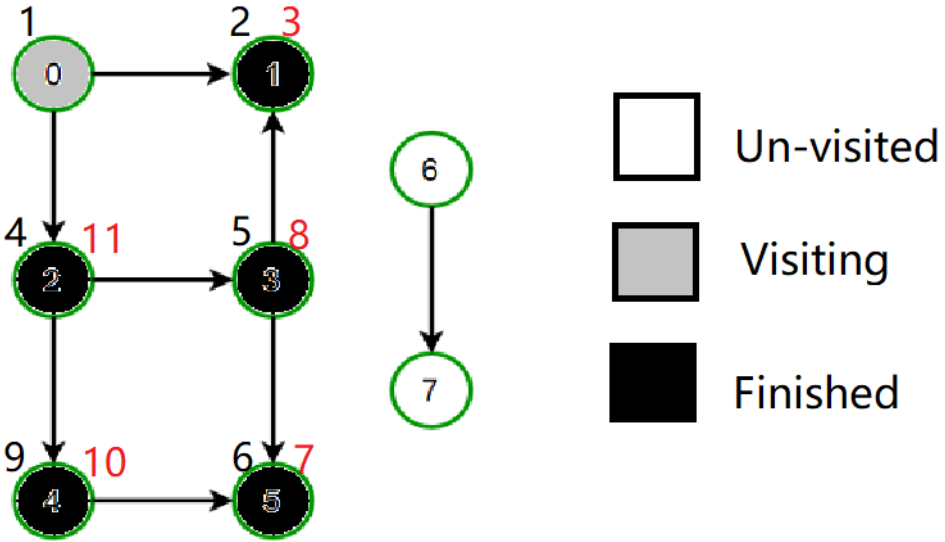

1
0
2
3
1
4
2
5
8
3
6
7
4
5
6
7
Un-visited
Visiting
Finished

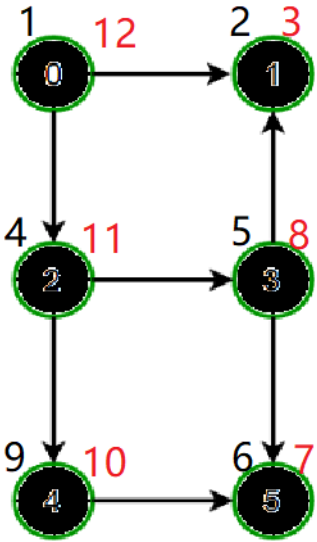

1
2
3
0
1
6
4
5
8
2
3
7
9
6
7
4
5
Un-visited
Visiting
Finished


1
2
3
0
1
4
5
8
2
3
9
10
6
7
4
5


6
7


Un-visited
Visiting
Finished


1
2 3
0
1
6
4 11
5 8
2
3
Un-visited
Visiting
Finished
7
9 10
6 7
4
5


1
12
2
3
0
1
4
11
5
8
2
3
9
10
6
7
4
5


6
7


Un-visited
Visiting
Finished

Implementation of state

CPMSOC

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

CPMSOC

```
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

CPMSOC

```
        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

CPMSOC

```
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

CPMSOC

```
vector<int> top_sort;
while (!starting_nodes.empty()) {
    // extract an arbitary 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:

CPMSOC

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

Let's solve a problem

CPMSOC

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] = [a_i, b_i]` indicates that you **must** take course `b_i` first if you want to take course `a_i`.

- 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

CPMSOC

https://leetcode.com/problems/parallel-courses-iii/
https://codeforces.com/contest/510/problem/C
http://www.usaco.org/index.php?page=viewproblem2&cpid=838

Attendance form :D

CPMSOC

https://forms.gle/UwwCBjbjgESDv2qE8

Further events

CPMSOC

Please join us at:

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