Course Schedule Planning —An insight of topological sort

Overview

This report mainly relates to the graph module of the course contents, and it bases on a typical problem of course schedule planning, where most of the courses have prerequisite relationships, and then provide a walkthrough of topological sort, from the perspectives of what it is, how it works, and how to extend it.

The report is organized in three major parts, introduction, analysis, and conclusion. In introduction, the report will discuss the background of the course schedule planning problem, and why our group choose it as our topic. In analysis part, the report will turn the life problem into mathematical problem, then provide two solutions to achieve topological sort, together with some comments on both solutions. In conclusion part, the report will based on the pros and cons proposed in previous part to make a recommendation, then analyze the weaknesses and limitations of topological sort, and finally propose an extension of topological sort to better solve the course schedule planing problem.

After all the analysis, I will then discuss what I have learned from this project and the value of this project.

Introduction

Course registration is never easy for students, as they have to balance the work load of each semester, consider their preferences, and whether these courses are overlapping with each other. However, the most critical thing is to make sure that you have fulfilled all the prerequisites of the course you wanted to take. This will require an up-front planning dated back to one even several semesters ago.

But the prerequisite relationships could be complicated, for example, one course could have several prerequisites, and each of the prerequisites has its own prerequisites. This sophisticated relationships will lead to a huge raise of the difficulty of planning.

As the course registration for 2023 spring semester just finished, some of the students reported that they met problems in course selection, as they have not fulfill the pre-request conditions. Hence they cannot enroll the desired courses in next semester.

This problem caught our eyes, as it is a quite common mistake that could happen to any student. To make our life easy, our group therefore want to solve this problem.

There is always a simple way of brute force, which is to write down the permutations of all the courses one plan to take. And check if the prerequisites of each course are taken before the it. However, the time complexity of this solution is exponential, as there are N factorial permutations for N course. Hence, this solution is not feasible in real life.

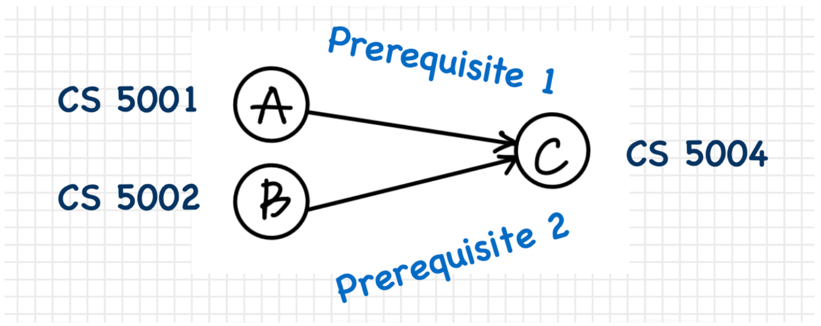
Therefore, the question is that is there a much simpler and less time-consuming way to find a sequence so that the prerequisites of each course are taken before it?

Analysis

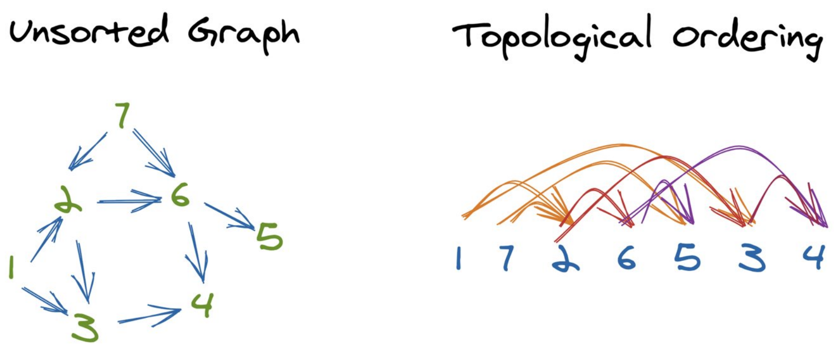
To solve this problem, we first need to turn it into a mathematical problem.

Because of the existence of prerequisite relationship between two courses, an easy observation is made by us that every 2 courses could have this relationship or not. And the relationship is not symmetric, as if A is the prerequisite of B, B must not be the prerequisite of A. Therefore, we naturally relate this problem with directed graph, as the direction of the edge represents the prerequisite relationship's direction.

To further elaborate on this mapping, every course could be represented by a vertex in the graph, and every prerequisite relationship could be represented by an edge in the graph, with the direction pointing from a course's prerequisite to the course itself. As show in the following illustration, CS5001 and CS5002 are the prerequisites of CS5004. Therefore, we use vertices A, B, and C to represent CS5001, CS5002 and CS5004 respectively. And the edges are pointing from CS5001 and CS5002 to CS5004.

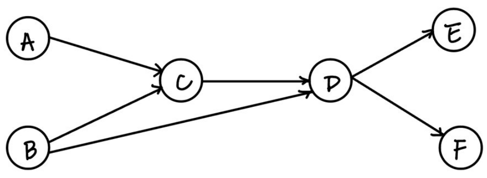


To make it a linear sequence, so that each course's prerequisites exist before the course itself, here we introduce topological sort, which is to sort a messy graph (like shown in the left of the following picture) in a topological way so that each section will only contains edges pointing from this section to the later sections, but no edges in the other way (like shown in the right of the following picture).



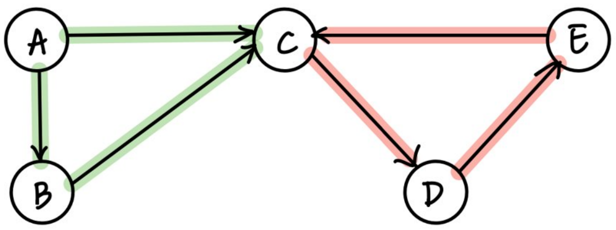
We performed some online research, and I even took an algorithm course from Princeton online resources, to find the algorithms to solve the topological sort problem.

The typical way of topological sorting is depth-first search (referred as DFS), which is to use DFS to search for all vertices and generate a sequence of visited vertices, and finally reserve the sequence to get a valid topological solution. A visited vertex is a vertex with no more unvisited out-going edges and unvisited vertices next to it.



As in the above graph, if we do DFS from vertex A, we will go through A, C, D, E, and then have E as our first visited vertex. Then F as our second visited vertex. And so on so forth, we will get a sequence of visited vertices as E-F-D-C-A-B. When we reverse it, we will get a valid topological solution as B-A-C-D-E-F, where the edges of every vertex are pointing to the later vertices.

However, by using DFS, we will face a problem that when there is a cycle in the graph, we will still get a solution which is invalid while we not even knowing that. Like the following graph illustrates.



First of all, let us define a cycle in a directed graph. It is a walk with the same starting and ending vertex, while the direction of the walk follows the direction of each edge it passes through. For instance, in the above graph, A-B-C is not a cycle as it cannot go back to vertex A if following the direction of the edges. But C-D-E is a cycle as you can go from C to D to E, then back to C.

In the above graph, apparently, there is no topological solution, because C-D-E is a cycle and hence cannot decide which could go first. However, if we perform a DFS from vertex A, it will still generate a sequence of visited vertices, as E-D-C-B-A. When we reverse it, we will get an invalid solution without knowing it, unless we double check the solution in person by ourselves.

Therefore, to solve this, in the code implementation part, we introduce a boolean variable called "valid" to record the status of the graph. If we notice the existence of a cycle, the boolean will because false, and we will just return an empty list, so that we know there is no valid topological solution here.

To check if there is a cycle, it is required to set three status for every vertex. In the code, we use integers to represent the three status. 0 means the vertex is not touched yet. 1 means the vertex is touched, but not popped as a visited vertex yet. 2 means the vertex has been popped as a visited vertex already. Therefore, when in the process of DFS, if we newly touch a 0-status vertex and change its status to 1, we will then iterate the vertices next to it. And we know a cycle exist if we find a 1-status vertex in the iteration process.

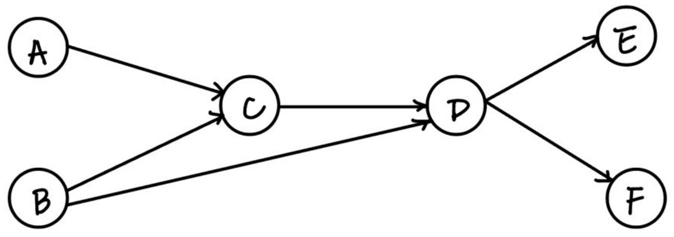
The code file is included in the appendix part of this project.

The time complexity is O(V+E) where V is the amount of vertices, and E is the amount of edges. As we only tough each vertex and each edge once, so the time complexity is O(V+E), which is linear. Meanwhile, since we use a list of lists to record all vertices and their prerequisites, and one extra boolean to record if there is cycle in the graph. So the space complexity is O(V). Compared to the exponential method, DFS saved a lot of time and space.

The advantage of using DFS is that this method is straight forward, and easy to understand. But the disadvantages of DFS, compared to the method we will talking about later, are that it is difficult to implement, that we need extra boolean even loops to check if there is a cycle in the graph, and that the amount of codes is larger, hence hard to understand by others, if only present the codes.

To make the code easier for others to read and understand, also to come up with a simple way to realize the topological sort, we hence introduce anther method as breadth-first-search (BFS).

BFS is to calculate the in-degree of each vertex first, then pop the vertices with 0 in-degree in the solution sequence. Afterwords, remove the vertices with 0 in-degree and their edges from the graph, and then re-calculate the in-degree of the vertices which the in-degree is affected. Then we repeat this process until there is no more vertices' in-degree becomes 0.



Use the previous case as an example, we could calculate the in-degree of each vertex and hence create the following table.

Vertex	A	B	C	D	E	F
In-degree	0	0	2	2	1	1

So we could put A and B in the solution sequence, and remove them and their edges form the graph. Then we need to re-calculate all the affected vertices' in-degree as follows.

Vertex	A	B	C	D	E	F
In-degree	0	0	0	1	1	1

Now as the in-degree of C becomes 0, we could put C into the solution sequence. And remove C and its edge form the graph. Then re-calculate the affected vertices' in-degree as follows.

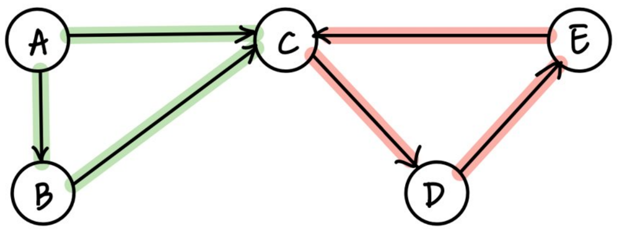
Vertex	A	B	C	D	E	F
In-degree	0	0	0	0	1	1

Now as the in-degree of D becomes 0, we could put D into the solution sequence. And repeat the above steps to get the following table.

Vertex	A	B	C	D	E	F
In-degree	0	0	0	0	0	0

Finally we could put E and F into the solution sequence, and we could get the sequence as A-B-C-D-E-F, which is a valid solution.

While there is a circle in the graph, as previous example (shown as follows), using BFS could easily find out whether the solution is valid or not, by checking whether the length of generated sequence is equal to the number of vertices.



In the above graph, as C-D-E is a cycle, there is no topological solution. When we perform BFS to sort topologically, we will first calculate the in-degree of all vertices, and get the table as follows.

Vertex	A	B	C	D	E
In-degree	0	1	3	1	1

Then we can take A as first in our solution, and remove A and its edges in the graph. As the table shown below.

Vertex	A	B	C	D	E
In-degree	0	0	2	1	1

Then we could do the same thing to B, to reach the following table.

Vertex	A	B	C	D	E
In-degree	0	0	1	1	1

Now, it is easy to notice that all the remaining vertices have an in-degree of 1. So we cannot take and remove anymore vertices in this case. Therefore, the BFS will stop at this stage, and the sequence generated is A-B, which has a length of 2.

By comparing the length of the sequence and the number of vertices, we will immediately know that the sequence is invalid.

The code file is included in the appendix part of this project.

The time complexity is O(V+E) where V is the amount of vertices, and E is the amount of edges. As we only tough each vertex and each edge once, so the time complexity is O(V+E), which is linear. Meanwhile, since we use a list of lists to record all vertices and their prerequisites, and one extra list to record the in-degree of each vertex. So the space complexity is O(V). Compared to the exponential method, BFS also saved a lot of time and space.

The advantage of using BFS is that this method does not need an extra loop to check if the result is valid. We could know immediately by comparing the length of solution against the number of vertices. But the disadvantages of BFS, is that it can only know if the solution is valid when all the sortable vertices are iterated, while in DFS, we could re-arrange the code and once find a loop, we could immediately report the result.

Conclusion

To answer the question that is there a much simpler and less time-consuming way to find a topological sequence, we have two method to perform the topological sort. The time complexity is O(V+E), where V is the number of vertices and E is the number of edges.

Topological sorting is wildly applied in both the industry and life. For example, lots of IDEs will using DFS topological sort to detect whether there is a dead-lock in the program.

Back to the problem of schedule planning, however, one largest drawback of topological sort is that the result is only one of all the possible result. On the one hand, it cannot return all possible sorting results, so that students could compare and choose from them. On the other hand, it cannot consider the preferences of students, for instance, student will prefer take fundamental courses before professional ones. But this requests cannot be considered by the sorting algorithms.

One possible extension to solve this limitation is that we could turn the graph from a un-weighted digraph into a weighted digraph. The weight could represent the workload or easiness of the course, or the preferences of the student. So when performing the topological sort, we will compare each edge's weight and choose the lightest edge first. In this case, we could manipulate the algorithm to be a stable and predictable one, which will give one best solution instead of one possible solution.

What I learned from this project are the two algorithms of topological sort, and the fact that a weighted directed graph could always be an extended solution of un-weighted directed graph. This project broadened my ways of thinking, and I believe that it will be of high value in my future algorithm studying.

Appendix

Code - DFS
https://github.com/kate-yq/NEU_CS5002_final_project_repo

Code - BFS
https://github.com/kate-yq/NEU_CS5002_final_project_repo