COURSE SCHEDULING

November 10, 2022

Karthik Nandan Dasaraju (2021CSB1081) ,
Pavithran Goud Gattu (2021CSB1088) ,
Kola Sai Datta (2021CSB1106)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Avadesh Gaur

Summary: The aim of this project is to design a time table that schedules the classes for all branches of students such that there is no overlapping of classes in the same time slot. The input will be taken from a file with a defined format and a neat time table will be generated in a new file. We have used backtracking algorithm with a DSATUR[1] heuristic to solve this problem.

---

1. Introduction

Our project, Course Scheduling, is an application of the well known graph coloring problem. The graph coloring, also known as vertex coloring, is to find an assignment of colors to all the vertices of the graph such that, no neighbours in the graph have same color. Formally, if $\boldsymbol{V}$ is the set of all vertices in the Graph $\boldsymbol{G(V, E)}$, where $\boldsymbol{E}$ is the set of all edges, we need to find a function $\boldsymbol{f : V \rightarrow C}$ where $\boldsymbol{C}$ is the set of colors available such that,

$$\boldsymbol{f(v_1) \neq f(v_2)} \ \forall(v_1, v_2) \in \boldsymbol{E}.$$

Numerous algorithms and heuristics are available to solve this problem. We choose to approach this with DSATUR[1] heuristic on top of the backtracking algorithm. The next section details on how scheduling the courses is connected to solving the graph coloring problem.

2. Understanding

The courses available to enroll, branches present, courses each branch takes and the sessions available will be taken as input. For example, courses could be CS201, MA101 and branches could be Computer science and engineering, Mathematics and Computing. The courses, each of these branches are enrolled in will also be taken as the input. CS201 for CSE and MA101 for the MNC branch. The sessions available are the time slots available for holding the classes.

Each course will be represented as a node in the graph i.e the vertices of the graph are the courses available to enroll. A connection (edge) will be established between the two courses if there is a batch enrolled in both the courses. For example, as shown in figure 1, The two courses CS201 and CS203 are connected since the CSE branch has to take both these courses. Colors being assigned to nodes are the sessions available. Again, as in figure 1, the color red indicates the session 10:00 AM to 10:50 AM. The graph coloring problem becomes apparent now, if the two courses CS201 and CS203 had a same color, that demands the CSE students to attend both these courses in the same time slot(10:00 AM - 10:50 AM) which is impossible.

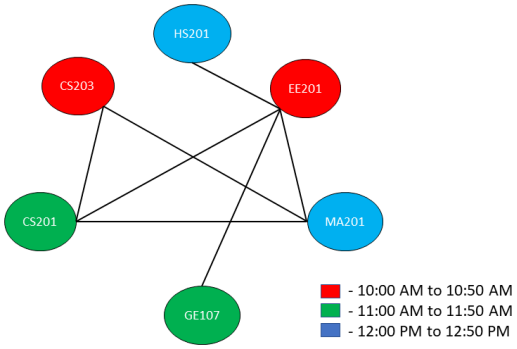

Figure 1: A Colored graph .

The problem now reduces to coloring each course(assigning a time slot) so that students enrolled in any neighbouring courses will have classes in different time slots.

3. Algorithmic Implementation

The following sections entail the initialization and working of the algorithm.

3.1. Initialization

We choose to use the adjacency matrix representation of the graph. Graph[i][j] will be 1 if course i and j are connected, 0 otherwise. Courses are represented with a data type-"course_node", each of these contain index(course index), color(an index for color), colored(a bool type), neighbours(an array of neighbours), edges(number of edges for the course). Table 1 shows this concisely.

name	index	color	colored	neighbours	edges
CS201	**0**	red	1	2, 4	2
CS203	**1**	–	0	1,2,0	3
EE201	**2**	green	1	0, 3, 5	3
GE107	**3**	–	0	1	1

Table 1: Sample data of courses.

Further, 'batch_data' variable contains the list of branches along with the courses they are enrolled in. We used a Max-Priority Queue to implement the DSATUR[1] heuristic, which will be discussed later.

3.2. Algorithm

Here is the pseudo-code for the backtrack algorithm.

Algorithm 1 **BackTrack()**

```
1: if heap.size ≠ 0 then
2:    course = Extract_Max(heap)
3:    for i in colors do
4:       if check_color(course, i) then
5:          course.color = i
6:          course.colored = 1
7:          if BackTrack() ≠ 1 then continue;
8:          return 1
9:       end if
10:   end for
11:   return 0
12: end if
13: return 1
```

BackTrack() is a recursive algorithm that checks if a color is assignable to vertex and if it is, assigns until all vertices are colored. It starts by picking a vertex (*line 2*) and checks if an available color is assignable to it (*line 4*). If it is, that color is assigned to the picked course and its state is changed to 1 (*line 5, 6*). The algorithm then calls itself recursively to continue the process of picking nodes and assigning colors to them (*line 7*). Note that every time we call Extract_Max() to pick a course, the variable heap.size decreases by one. If the last call finds the heap empty, it returns 1 at which point all the parent calls will return 1, indicating that a successful assignment had been found. If none of the colors is assignable to the course, the code returns 0(*line 11*) which makes the parent call pick a different color for the previously assigned vertex.

The purpose and execution of ***Extract_ Max()*** will be discussed in the subsequent sections. Below is the pseudo-code for the check_color algorithm.

Algorithm 2 **check_ color**(course.color, color)

```
1: for i in course.neighbours do
2:    if i.colored == 1 and i.color == course.color then
3:       return 0
4:    end if
5: end for
6: return 1
```

check_ color takes in arguments, the color to be assigned and the course node being assigned to. It checks if the neighbours of the course have the same color, in which case the assignment should not be done and the function exits while returning 0.

3.3. DSATUR[1] Heuristic

A normal backtrack would pick the course (*line 4 in Algo 1*) without any emphasis on the run-time or the recursive calls required to reach the solution. We introduced DSATUR[1] heuristic, which picks the courses based on the number of edges for that particular course. If we pick a course which has the maximum number of edges attached to it and assign a color, we would be restricting a large number of other courses and hence they no more can be assigned with the same color. Picking a course with more edges, will remove many possibilities and recursive calls. This heuristic is termed as the DSATUR[1] heuristic.

We have implemented a Max-Priority queue for this heuristic. Before ***BackTrack()*** is called, a max-heap is built and any calls to ***Extract_ Max()*** will restructure the heap to always have the course with maximum number of edges at the top.

4. Conclusions

Graph coloring has wide area of applications and we have explored many heuristics to solve this problem. We wanted to explore if we could implement the DSATUR[1] heuristic using a Priority queue. We have done many test with and the algorithm perfectly finds a solution whenever possible.

5. Bibliography and citations

Some of the links we have used: https://en.wikipedia.org/wiki/Graph_coloring
https://www.geeksforgeeks.org/graph-coloring-applications/

[1]

References

[1] Daniel Br61az. New methods to color the vertices of a graph. *Commun. ACM*, pages 251–252, 1979.

A. Appendix

A sample output file had been uploaded at: http://txt.do/ttpoh
Some images explaining the graph coloring were uploaded at:
https://ibb.co/xCGdB94
https://ibb.co/wg1XZgJ
https://ibb.co/F52ZR6h