CS512 Project: Optimization of Timetable with Coloring Graph

Daniel Ying (dty16) Sec 198:512:01-04
James Fu (jf980) Sec 198:512:01-04
Vincent Taylor (vt152) Sec 198:512:01-04
Xiang Li (xl649) Sec 198:512:01-04

December 18, 2021

Submitter: Daniel Ying

Honor Code:

I abide to the rules laid in the Project: Optimization of Timetable with Coloring Graph and I have not used anyone else's work for the project, and my work is only my own and my group's.

I acknowledge and accept the Honor Code and rules of 512 Project.

Signed: Daniel Ying (dty16), James Fu (jf980), Vincent Taylor (vt152), Xiang Li (xl649)

Workload:

Daniel Ying: Compiled the report, recorded Vincent.

James Fu: Coded the algorithm.

Vincent Taylor: Designed the pseudocode, made a video of pseudocode.

Xiang Li: Created the power point.

Together: Discussed the problems, Answered the questions, and Designed the project.

Problem 1

1.1 Introduction:

A graph coloring is an assignment of labels (colors) to the vertices of a graph such that there are no two adjacent vertices with the same color. One of the goal of this algorithm is to assign the minimal number of colors for the vertices in the graph G.

Two of the most common graph coloring problems are vertex coloring and chromatic number.

Vertex Coloring problem: given m colors, we need to find a way of coloring the graph's vertices such that no two adjacent vertices have the same color.

Chromatic Number problem: find the smallest number of colors needed to color G.

1.2 Algorithm Application:

The graph coloring algorithm can be applied on a number of different applications.

1. Time Tables: A list of different subjects with students enrolled in every subject. The question: how to schedule the exam so that no two exams with a common student are scheduled at the same? How many min. time slots are needed to schedule all the exams?

2. Assignment Mobile Radio Frequency: When frequencies are assigned to towers, frequencies assigned to all towers at the same location must be different. How to assign frequencies with this constraint? What is the min. number of frequencies needed?

3. Sudoku: There is an edge between two vertices if they are in same row or same column or same block.

4. Allocating Register: In compiler optimization, register allocation is the process of assigning a large number of target program var. onto small number of CPU registers.

5. Bipartite Graphs: Check if the graph is Bipartite or not.

6. Map Coloring: Color geographical maps of countries and states where no two adjacent country or states (depending on the map) are assigned the same color.

Problem 2

2.1 Pseudo-Code:

2.1.1 Inputs and Outputs

Input: G(V,E) where G is a Graph and V,E are verticies
Output: Minimum number of non-conflicting timeslots required for scheduling courses(Denoted by n)

2.1.2 Description

Step 1: Input the conflict graph G.

Step 2: Compute degree sequence of the input conflict graph G.

Step 3: Assign color1 to the vertex vi of G having highest degree.

Step 4: Assign color1 to all the non-adjacent uncoloured vertices of vi and store color1 into UsedColor array.

Step 5: Assign new color which is not previously used to the next uncoloured vertex having next highest degree.

Step 6: Assign the same new color to all non-adjacent uncoloured vertices of the newly colored vertex.

Step 7: Repeat step-5 and step-6 until all vertices are colored.

Step 8: Set minimum number of non-conflicting time-slots n= chromatic number of the colored graph=total number of elements in UsedColor array.

Step 9: End

2.1.3 PseudoCode Code:

If G(V,E) is an adjacency list:

```
getDegreeQueue  (G(V,E)):
   degreeQueue = [|V|]
   for  each  vertex  v : v  ε V
      degreeQueue.enqueue((V,len(G(V))))
   return degreeQueue
```

```
ColorNonAdjacent(u, G, color):
   for each v in V : vε V
      if(v ≠ u)
         isLinked = false
         for each w in adjcacencyList(v) : adjacencyList(v) ⊆ V
            if(w == u || w.color == u.color):
               isLinked = true
         if(isLinked)
            continue
         v.color = color
```

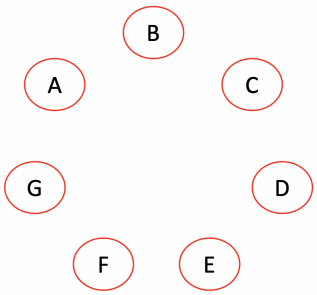
```
Minimum Color Graph(G(V,E)):
    degreeQueue = getDegreeQueue(G)
    colorArray = []
    n = 1
    while degreeQueue.isNotEmpty()
        v = degreeQueue.dequeue()
        if(v.color == null):
            v.color = color_n
            colorNonAdjacent(v,G, color_n)
            colorArray.append(color_n)
            n++
    return len(colorArray)
```

Problem 3

3.1 Visualize the algorithm:

Visualization

RUTGERS

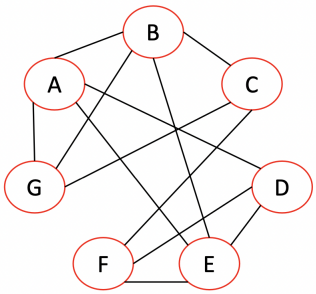


	A	B	C	D	E	F	G
A		X		X	X		X
B	X		X		X		X
C		X				X	X
D	X				X	X	
E	X	X		X		X	
F			X	X	X		
G	X	X	X				

- Draw conflict table, where X means the subjects conflict to each other
- Create graph G for each subject

Visualization

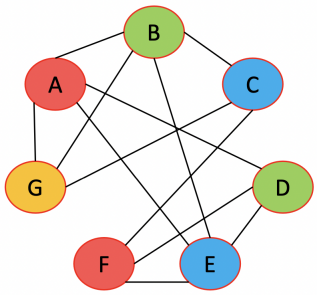
RUTGERS



	A	B	C	D	E	F	G
A		X		X	X		X
B	X		X		X		X
C		X				X	X
D	X				X	X	
E	X	X		X		X	
F			X	X	X		
G	X	X	X				

- Link adjacent nodes that conflict

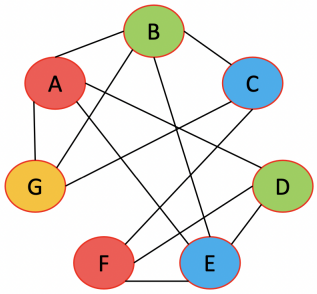
Visualization



	A	B	C	D	E	F	G
A		X		X	X		X
B	X		X		X		X
C		X				X	X
D	X				X	X	
E	X	X		X		X	
F			X	X	X		
G	X	X	X				

- Coloring non-adjacent nodes with RED, then GREEN, BLUE, YELLOW

Visualization



	A	B	C	D	E	F	G
A		X		X	X		X
B	X		X		X		X
C		X				X	X
D	X				X	X	
E	X	X		X		X	
F			X	X	X		
G	X	X	X				

- Nodes(subjects) with the same color can be assigned to a same time slot
- time slot1: A-F
- time slot2: B-D
- time slot3: C-E
- time slot4: G

4

Problem 4

4.1 Implement the algorithm:

The Objective of the code below is to find the time table for the classes provided by a department such that the students and professors would not be in a different class at the same time and at the same time figure out what is the minimum time slot to accomplish this.

Thus with the graph coloring algorithm, we will connect classes with the same professor and students to create the graph G. With the graph G created, we will implement the graph coloring algorithm on the graph G.

The output of the algorithm would be a list of classes each with a color such that no two classes that have the same professors and students at the same time. And, the algorithm would tell us at least how many different colors (the minimum different time slots) do we need to have every professor and students to be able to be in their designated class.

The different between algorithm 1 and algorithm 2, is the approach. Algorithm 1 uses graph coloring algorithm to obtain the a better optimal number of colors (timeslot) needed to allocate the classes.

Algorithm 2 on the other hand uses greedy coloring algorithm that is faster than algorithm 1, but there are cases in which the greedy algorithm does not produce a good optimal number of colors for allocating the the schedule of the professors and students.

4.1.1 algorithm 1: we designed

```
 from queue import PriorityQueue
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    degree: int
    node: Any=field(compare=False)

def get_graph(all_courses: dict, schedules: dict) -> dict:
    graph = dict()
    for courses in all_courses.values():
        for course in courses:
            graph[course] = set(other_course for other_course in courses
                                if other_course != course)

    for courses in schedules.values():
        for course in courses:
            graph[course].update(other_course for other_course in courses
                                 if other_course != course)
    return graph

def first_available_color(neighbor_color: list):
    color_set = set(neighbor_color)
    count = 0
    while True:
        if count in color_set:
            count += 1
        else:
            return count

def greedy_coloring(graph: dict) -> dict:
    color = dict()
    for node in graph:
        neighbor_color = [color[neighbor] for neighbor in graph[node]
                          if neighbor in color]
```

```
        color[node] = first_available_color(neighbor_color)
    return color

def get_degree_queue(graph: dict) -> PriorityQueue:
    degree_queue = PriorityQueue()
    for node, adj_list in zip(graph, graph.values()):
        degree_queue.put((-len(adj_list), node))
    return degree_queue


def color_non_adjacency(current, graph: dict, color: dict, color_count: int) ->
    for node, adj_list in graph.items():
        if node != current and node not in color:
            is_linked = False
            for neighbor in adj_list:
                if (neighbor == current or
                   (neighbor in color and color[neighbor] == color_count)):
                    is_linked = True
                    break
            if not is_linked:
                color[node] = color_count

def graph_coloring(graph: dict) -> dict:
    degree_queue = get_degree_queue(graph)
    color = dict()
    color_count = 0
    while not degree_queue.empty():
        _, node = degree_queue.get()
        if node not in color:
            color[node] = color_count
            color_non_adjacency(node, graph, color, color_count)
            color_count += 1
    return color

def main():
    professor_courses = {'professor A': ['CS501', 'CS512'],
                         'professor B': ['CS507'],
                         'professor C': ['CS502', 'CS515'],
                         'professor D': ['CS513'],
                         'professor E': [],
                         'professor F': ['CS520']}
    student_schedules = {'student A': ['CS501', 'CS512', 'CS520'],
                         'student B': ['CS502', 'CS512', 'CS520'],
                         'student C': ['CS507', 'CS513'],
                         'student D': ['CS501', 'CS512', 'CS515']}

    graph = get_graph(professor_courses, student_schedules)
    for node, adj_list in graph.items():
        print(f'{node}:\t{adj_list}')

    print("Class-> Class V ")
```

```
    for lect in graph:
        print(lect, "->",  graph[lect])


    color = greedy_coloring(graph)

    print()
    print("Coloring for Class:")
    for lect_c in color:
        print("Class: "+ lect_c + " = ", color[lect_c])


if __name__ == '__main__':
    main()
```

4.1.2 algorithm 2: Greedy Coloring

```
def get_graph(all_courses: dict, schedules: dict) -> dict:
    graph = dict()
    for courses in all_courses.values():
        for course in courses:
            graph[course] = set(other_course for other_course in courses
                                if other_course != course)

    for courses in schedules.values():
        for course in courses:
            graph[course].update(other_course for other_course in courses
                                 if other_course != course)
    return graph

def first_available_color(neighbor_color: list):
    color_set = set(neighbor_color)
    count = 0
    while True:
        if count in color_set:
            count += 1
        else:
            return count

def greedy_coloring(graph: dict) -> dict:
    color = dict()
    for node in graph:
        neighbor_color = [color[neighbor] for neighbor in graph[node]
                          if neighbor in color]
        color[node] = first_available_color(neighbor_color)
    return color

def main():
    professor_courses = {'professor A': ['CS501', 'CS512'],
```

```
                        'professor B': ['CS507'],
                        'professor C': ['CS502', 'CS515'],
                        'professor D': ['CS513'],
                        'professor E': [],
                        'professor F': ['CS520']}
    student_schedules = {'student A': ['CS501', 'CS512', 'CS520'],
                        'student B': ['CS502', 'CS512', 'CS520'],
                        'student C': ['CS507', 'CS513'],
                        'student D': ['CS501', 'CS512', 'CS515']}

    graph = get_graph(professor_courses, student_schedules)
    print(graph)

    print("Class-> Class V ")

    for lect in graph:
        print(lect, "->", graph[lect])


    color = greedy_coloring(graph)

    print()
    print("Coloring for Class:")
    for lect_c in color:
        print("Class: "+ lect_c + " = ", color[lect_c])

if __name__ == '__main__':
    main()
```

Problem 5

5.1 Type of Data:

The algorithm takes in N number of professors and M number of students and the total classes.

The result would be the classes are giving a time slot (a color) that allows professors and students to take the classes without colliding with another class they are designated to.

5.2 Input format:

The input data shows 6 professors each with their own class to teach, and 4 students that have been registered to the classes taught by the professors.

5.2.1 algorithm 1 and 2:

professor-courses =
'professor A': ['CS501', 'CS512'],
'professor B': ['CS507'],
'professor C': ['CS502', 'CS515'],
'professor D': ['CS513'],
'professor E': [],
'professor F': ['CS520']

student-schedules =
'student A': ['CS501', 'CS512', 'CS520'],
'student B': ['CS502', 'CS512', 'CS520'],
'student C': ['CS507', 'CS513'],
'student D': ['CS501', 'CS512', 'CS515']

5.3 Output Format:

5.3.1 algorithm 1: Our Design

Class-> Class V:
CS501 -> 'CS520', 'CS512', 'CS515'
CS512 -> 'CS520', 'CS501', 'CS502', 'CS515'
CS507 -> 'CS513'
CS502 -> 'CS520', 'CS512', 'CS515'
CS515 -> 'CS512', 'CS501', 'CS502'
CS513 -> 'CS507'
CS520 -> 'CS512', 'CS501', 'CS502'

Coloring for Class:
Class: CS501 = 0
Class: CS512 = 1
Class: CS507 = 0
Class: CS502 = 0

Class: CS515 = 2
Class: CS513 = 1
Class: CS520 = 2

5.3.2 algorithm 2: Greedy Graph Coloring

Class-> Class V:
CS501 -> 'CS520', 'CS512', 'CS515'
CS512 -> 'CS520', 'CS501', 'CS502', 'CS515'
CS507 -> 'CS513'
CS502 -> 'CS520', 'CS512', 'CS515'
CS515 -> 'CS512', 'CS501', 'CS502'
CS513 -> 'CS507'
CS520 -> 'CS512', 'CS501', 'CS502'

Coloring for Class:
Class: CS501 = 0
Class: CS512 = 1
Class: CS507 = 0
Class: CS502 = 0
Class: CS515 = 2
Class: CS513 = 1
Class: CS520 = 2

The output shows we need a minimum of 3 different time slots to allocate the different classes and allow the students and professor to take their designated class without colliding with another class they have.

The difference between the two algorithms is: Objective.

According to the the runtime of the two algorithms, we can see that algorithm 2 the Greedy algorithm has a faster runtime (O(V+E), but due to the nature of its design, the greedy algorithm does not optimize the number of time slots used to allocate the calls (as seen in the figure below).

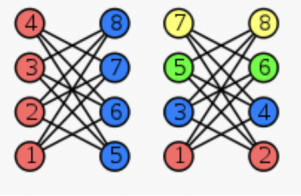


Figure 1: The left is the optimal, right is greedy coloring.

while for the algorithm we designed (check the video submitted), our algorithm though is slower O(VE+E), but it produces the a better optimal number of colors we need to allocate the classes than greedy coloring algorithm.

5.4 Size of data (can it fit in desktop? Need a special server? Is it data at rest or is it streaming data?)

The data should be easily accessible on a desktop, thus the size of the data should be not too big. So there is no need for a special server to store the data.

Due to the fact that this algorithm can be used by campuses and/or universities to design and obtain an optimal timetable, the data would most likely be in a database for the faculty and students to access.

The data are class schedule for professors and students, thus it is not necessary for the data to be real-time collected data.

Problem 6

6.1 Three questions you would like to answer with application:

6.1.1 Which data structure is better to store the input? Adjacency matrix or list?

Adjacency list would be better because the algorithm is always iterating through the neighbors of one node, and the algorithm does not care about the specific node to node connection.

6.1.2 What is the time complexity to build the adjacency list?

The time complexity of building an adjacency list is to go through all the nodes and their degree. That will be O(|V| + sum of degree of all nodes) = O(|V| + 2|E|) = O(|V| + |E|).

This will be the optimal time to allocate the input. However, this depends on the input info collected. Thus it is very likely to take more time.

6.1.3 What is the data structure of the output? How should we use it?

Output is a hashmap where the keys are all the courses, and the values are colors (different time sections).

Since we only figured out the courses should be separated out, we still need information such as: all the available time schedules, available classrooms according to the time-schedule, the professor's preference, etc. Moreoever, that will require another program to use the info we got and map all the course into a time schedule table.

Problem 7

7.1 How are our answers presented to the user of our application?

Python dict., where (key, values) pairs are (course, color) pair.

Color is represented as an integer, such that courses with different numbers would mean they are assigned different colors, thus difference time schedules.

And the number of unique colors is the minimum time slots to accommodate the given time schedules of professors and students.

Problem 8

8.1 What algorithms are we using? What are their complexity as a function of the input size?

The algorithm 1 we are using is a algorithm we designed, it has a time complexity of runtime O[|V||E| + |V|].

The algorithm 2 we are using is the called greedy graph coloring, with a time complexity of runtime O[|E| + |V|].

The data structure is Python dict (implemented by hashmap) and priority queue (implemented by heap).

Problem 9

9.1 Do you have access to implementation of algorithms? What language is being used?

As stated before, our data structure are Python dict (implemented by hashmap) and priority queue (implemented by heap). Thus we have quick and organized access to the keys and values stored in the data structure. The language we used to construct this is Python.

Problem 10

10.1 What is novel about our project?

Due to the nature of its design, the greedy algorithm does not optimize the number of time slots used to allocate the calls (as seen in the figure below).

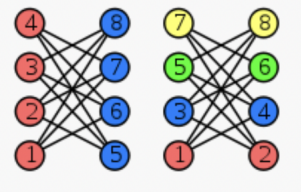


Figure 2: The left is the optimal, right is greedy coloring.

while for the algorithm we designed (check the video submitted), our algorithm produces the a better optimal number of colors we need to allocate the classes than the greedy coloring algorithm.

We often want the optimal answer to a problem, but in reality, an approximated solution is good enough. As our problem, we want to determine the time schedule for each course, since there are plenty of time section in a week's schedule, we do not have to always find the minimum time sections we need. Instead, an algorithm that can come up with a good enough time table in a linear (quadratic) time is much valuable.

Problem 11

11.1 Is your project feasible?

Our project is feasible.

The project is available to almost all individuals with a functioning hardware. It does not have a high demand for the efficiency of the hardware, an old laptop would be able to run the algorithm.

If we look at the time and space capacity, it can run a university or school's classes for the teachers and students fairly well. Now if we look at the necessity for financial or other resources, as stated before, this program can be implemented on most hardwares, no need for a quantum computer to run it.

In the aspect of market demand, if we look at the multiple applications the graph coloring optimizer, there is a market in which this algorithm can be applied to, ranging from universities, to registrar offices.

Also looking at the algorithm data-wise, this algorithm would only require the course information (containing the professor and students' name and the course they had chosen), it does not require any other additional data other than these.

Considering the scope of the work, planning process of the algorithm, many test data, a rather low project budget, and a high build confidence, we believe that the graph coloring algorithm that optimizes the timetable is highly feasible and can be used by campuses.