Course Scheduling Algorithm

Adviser: David Eppstein Student: Zhaohua Zeng

June 10, 2017

Contents

Abstract

By modifying the Coffman-Graham algorithm and the Hu's algorithm, we give a scheduling algorithm that can output a four-year study plan for college students in $O(|V|^2)$ time, where $|V|$ is the number of courses in a graph.

We first introduce two related precedence-constrained multi-processor scheduling algorithms, and then in the second section, we describe how we adapt these two algorithms to solve the course scheduling problem we consider. In the third section, we show example schedules we get with small datasets by using our course scheduling problem. Lastly, we conclude our achievements and discuss what we can do in the future to improve the course scheduling problem.

1 Introduction

Consider the problem of scheduling courses for next quarter. To graduate on time, what courses should a Computer Science student choose in the next few quarters? Currently, many students, especially freshmen and transfer students, would like to ask school academic counselors for advice before enrolling to courses of next quarter. Since the number of counselors is limited but many students would go to the office for help, getting suggestions becomes time consuming.

Course scheduling problem is similar to the precedence-constrained mulitprocessor scheduling problem introduced in section 2, which can be solved by the Coffman-Graham algorithm and Hu's algorithm. These two algorithms can generate schedules for tasks as well as guarantee a certain quality. However, some constraints that exist in course scheduling problem is not in the precedence-constrained multiprocessor scheduling problem, which means, these two algorithms cannot resolve the course scheduling problem directly without any modifications. Consequently, during the research, we first construct a skeleton based on the Coffman-Graham algorithm, and then we add constraints and solve them one by one on the skeleton. Lastly, we improve the algorithm with the inspiration from the labeling phase of Hu's algorithm.

Our course scheduling algorithm can generate a reasonable schedule plan as a guidance for students. Using this algorithm, one can develop a tool that can help academic counselors shorten the time spent on arranging student

schedules and provide accurate information. At the same time, students can check what series of courses they are required to take to satisfy some prerequisites and organize their study plans at ease.

Our course scheduling algorithm can:

1. Make a course schedule for the entire undergraduate/graduate path, which can satisfy school/major requirements;

2. Allow users to specify different maximum number of course units for different quarters;

3. Allow users to supply a partial schedule to be completed.

4. Allow users to avoid taking some courses.

2 Related Works

The Coffman-Graham algorithm and Hu's algorithm are related to precedence-constrained multi-processor scheduling problem, which is about minimizing the total makespan of the schedule. The total makespan is the total length of the schedule.

Let V be a set of tasks with indices $1, 2, \ldots, n$, and $G = (V, E)$ be an acyclic directed graph with $|V|$ tasks and $|E|$ edges. An edge $e = (u, v)$ for $u, v \in V$ is in E if u must be executed before v is assigned to a processor. W is the number of processors in the problem. Each task $v \in V$ takes 1 unit of time to finish, each processor w_j can only execute one task at a time. The problem is about scheduling $|V|$ tasks to W multiprocessors. A layer L_k with width W is the assignments in time t_k in the schedule L.

Minimizing the total makespan of a schedule is equivalent to minimizing the time from the start of a schedule to the completion of its last task. However, this problem is NP-complete, so both the Coffman-Graham algorithm and Hu's algorithm suggest heuristic methods for scheduling [1]. Note that the first one minimizing the makespan, and the other one minimizing the total completion time.

There are some similarities between the precedence-constrained multiprocessor scheduling problem and the course scheduling problem. First, we can view each task as a course in our problem, and each course also takes one unit of time to finish within one quarter. Then we can use a quarter as a time unit.

Similar to the precedence-constrained relationship represented by edges between tasks, some courses have other courses as their prerequisites. It means students cannot take a course until they satisfy its prerequisites, so the course could only be assigned to a layer later than its prerequisites' layers. Moreover, although there is no exact number of processors in course scheduling problem, we can use maximum units as the width of a layer instead. Since we want to minimize the total number of quarters in the schedule, the goal of the course scheduling problem is also minimizing makespan. Consequently, we can transform the precedence-constrained multi-processor scheduling problem to the course scheduling problem and develop a new algorithm based on algorithms described below to solve the problem.

Note that the course scheduling problem is not the same as the precedence-constrained multiprocessor scheduling problem. Quarters corresponding to layers in the schedule have varying layer width. Courses also differ from tasks in two aspects: some classes are not offered every quarter, and the prerequisites for a class are not just the AND of all listed prerequisite classes but are instead described by a more complicated logical formula, the conjunctive normal form.

2.1 The Coffman-Graham Algorithm

The Coffman-Graham algorithm takes a reduced directed acyclic graph (DAG) $G = (V, E)$ and a positive integer W, the width of the layer, as inputs. The basic Coffman-Graham layering algorithm consists of two phases: the first orders the vertices, and the second assigns vertices to layers [1].

A reduced DAG G means there is no transitive edges in the input graph for the algorithm, that is, a transitive reduction of the original input directed acyclic graph $H = (V', E')$. A directed graph G is said to be a transitive reduction of the directed graph H if G has the same reachability as G and has the minimum number of edges among all sub-graphs of H [2]. DAG's transitive reduction can be done in $O(M(|V'|))$ time, where $M(n)$ is the time for computing the product of two $n \times n$ matrices [2, 3].

Coffman and Graham [4] describe the Coffman-Graham Labeling algorithm: Let $\alpha(v)$ denote the label assigned to a task $v \in G$. Assign $1, \ldots, k$ to the k tasks in G without any immediate successors. Suppose for a $j \leq |V|$, $1, \ldots, j$ integers have been assigned to some tasks. For each task v without any unlabeled immediate successors, let $S(v)$ be the set of immediate successors of v, let $N(v)$ denote the decreasing sequence of integers formed by

ordering the set $\{\alpha(u) : u \in S(v)\}$. Assign $\alpha(v) = j + 1$ to the task v such that $N(v) \leq N(u), \forall u \in S(v)$. Repeat this process until every task in G have been labeled.

In some cases, when two tasks both have all their immediate successors labeled, they are differentiated with a lexicographical order [1].

Figure 1 shows a possible labeling result for a graph by the algorithm. The letter inside each node is the name of the node and the integer behind each node denotes the label assigned to it. The original graph is from Hu [5] to describe the labeling process in Hu's algorithm. To better compare the Coffman-Graham algorithm and Hu's algorithm, here we use the same graph but with different labels to illustrate.

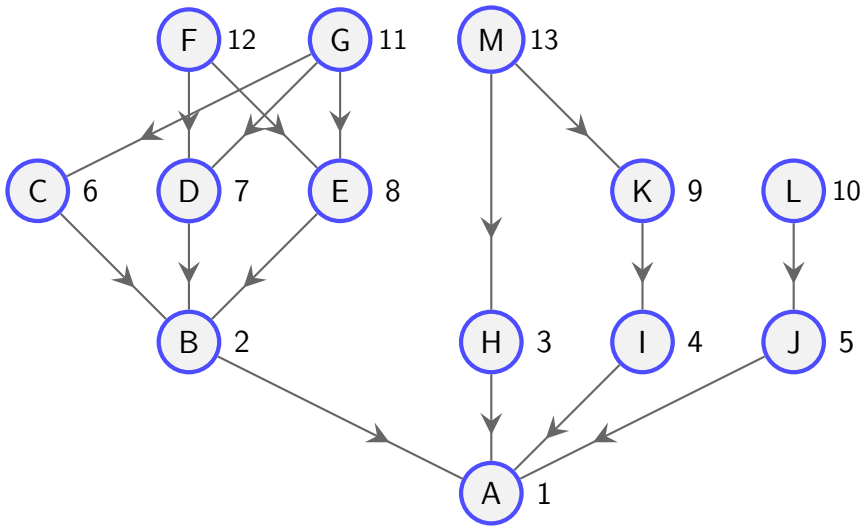

Figure 1: Labeling Process In The Coffman-Graham Algorithm

The scheduling phase of the Coffman-Graham algorithm is illustrated as follows [6, 4]: Whenever a processor is free for assignment, assign the task with the highest label among all ready tasks to it.

Let W be the number of processors, ω be the length of a schedule produced by the algorithm, and ω_0 be the length of an optimal schedule, then $\omega/\omega_0 \leq 2 - 2/W$ [4, 7]. When $W = 2$, the Coffman-Graham algorithm can produce an optimal schedule.

layer	tasks	
0	M	F
1	G	L
2	K	E
3	D	C
4	J	I
5	H	B
6	A	

Table 1: Scheduling Process In The Coffman-Graham Algorithm

2.2 Hu's Algorithm

Given a fixed width w, Hu's algorithm can find a schedule for a finite number of tasks which all require unit time to complete in $O(|V| \log |V|)$ time [6]. McHugh [8] proves that when each task is allowed to have arbitrary completion time, Hu's algorithm minimizes the total completion time for a set of tasks restricted by precedence constraints that are modeled by a singly rooted tree.

Similar to the Coffman-Graham algorithm, Hu's algorithm also has labeling and scheduling phases. However, they are differentiated by their ways of labeling. In Hu's algorithm, the label of a task is a function of the *level* of the task [9]. Let a final task be a task without any successors. Final tasks all receive a label 1 at the beginning of the labeling process. Assume each task requires 1 unit of time to finish, then a task v is labeled with $\alpha(v) = x_v + 1$, in which x_v is the length of the longest path from v to the final task in graph [5]. Figure 2 by Hu gives an example of labeling process in a graph. For a node v in the figure, the letter inside the node is the name of it, and the number that appears behind v is v's label $\alpha(v)$.

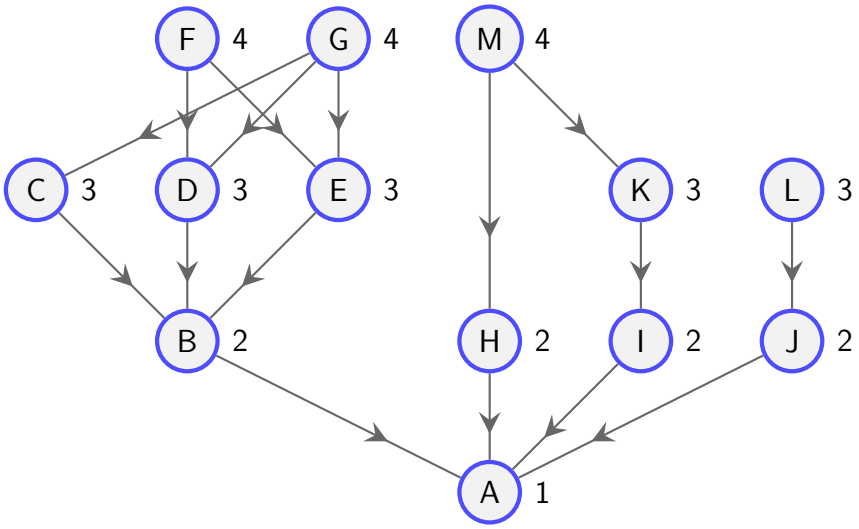

Figure 2: Labeling Process In Hu's Algorithm [5]

The scheduling phase is similar in Hu's algorithm as in the Coffman-Graham algorithm despite the fact that tasks can have the same label in Hu's algorithm. Assume that we will schedule for W processors, and each processor can only execute one task at a time. Whenever a processor is free for assignment, assign the task with the largest label among all ready tasks to the processor. Repeat until all tasks are assigned. When some ready tasks have the same label, break the tie arbitrarily [6]. In other words, it selects a ready task that has the longest distance to a final task. We then assign it to the highest layer we currently have. If the number of tasks in this layer is equal to the layer width W, it creates a new layer above this layer and assigns the task to the new layer. Table 2 shows a schedule generated by Hu's algorithm based on the graph in the figure 2.

layer	tasks	
0	F	G
1	M	C
2	D	E
3	K	L
4	B	H
5	I	J
6	A	

Table 2: Scheduling Process In Hu's Algorithm

3 Proposed Method

3.1 Problem Formulation And Term Definition

3.1.1 Schedule

We represent a schedule L as a list of layers. Each layer stands for a quarter in the actual schedule, and it holds a set of assigned courses. The first layer L_0 in the schedule is the bottom layer.

There is a width bound $W(L_i)$ for each layer $L_i \in L$, which indicates the maximum total units of courses in L_i, and a current width $M(L_i)$ associates with it, which represents the current total number of units in L_i. Since the number of layers is unknown at the beginning, it is required to set a default value for the width bound. If there is no specified width bound for L_i, its width bound is $W(default)$. When initialized, the default value for $M(L_i)$ is 0 since L_i is empty. Every time when a course v is assigned to a layer L_i, we update $M(L_i) = M(L_i) + v.Units$.

3.1.2 Upper Bound

We use an upper bound for the schedule to determine below which layer can an *upper standing only* course be assigned. That is, suppose the upper bound at layer L_u, then we can only assign this course to a layer with index $i \geq u$.

During the assignment, we may assign a course to a new empty layer even if the width of current top layer does not reach the maximum. Thus, we are not able to estimate the exact upper bound when the algorithm starts scheduling. We specify an upper bound layer index u for layers instead, indicating that for the layer $L_i \in L$, if $i \geq u$, then layer L_i can hold a upper standing only course.

3.1.3 Requirements Table

A student should fulfill a list of requirements for school, majors, minors, and specializations in order to graduate. Therefore, we create a requirements table R, which contains a list of requirements. A requirement is in the following format:

> n courses in:
> (Course1, Course2, Course3)
> **AND**
> m courses in:
> (Course4, Course5, Course6)

n and m is the number of courses required in the set of courses. Both the number of sets and the number of the courses in the set are not fixed. Let the name of a requirement be k, the index of a set *(Course1, Course2, Course3)* be i, and the number of courses required for this set be n. Then we get $R_k^i = n$.

3.1.4 Course

A vertex in graph represents a course in our course scheduling problem. A course v contains the following information:

- $v.Units$: Total units v requires. If v has a 4 units lecture section, and a 1 unit lab section, it requires 5 units in total.
- $v.QuarterCodes$: In what quarters the department offers this course. There is a presumption in both the Coffman-Graham algorithm and Hu's algorithm that all tasks are available at any time in the schedule. However, a course may not be offered in every quarter and every year, which means that not every layer can hold this course. Therefore,

knowing when a course will be offered is significant in the scheduling phase.

We store quarter information for the latest 2 years, 6 quarters in total for testing. The latest year is called *odd year*, the the second latest year is *even year*. Each quarter has a quarter code, and if a course is offered in a quarter, we add this quarter code in quarters information. For instance, if year term 2017-2018 is the latest year, then it is an *odd year*, and year term 2016-2017 is the *even year*. The figure 3 below illustrates the relationship between quarters and quarter code in a university on quarter system.

Quarter	Quarter Code
2017 Fall	0
2018 Winter	1
2018 Spring	2
2016 Fall	3
2017 Winter	4
2017 Spring	5

Table 3: Quarter And Quarter Code

Using quarter codes can help matching layer indices with a corresponding quarter using a mod operation. For example, with 0-indexing, if a layer has an index equal to 8, then we calculate $8 \bmod 6 = 2$, so the quarter code for this layer is 2.

- $v.IsUpperOnly$: It denotes if v is an upper only course. If it returns $true$, it means v is an upper standing only course. Suppose there is an integer U representing upper units and $U \geq 0$, then an upper standing course requires a student to achieve at least U units before they can take it.

  Thus, if $v.IsUpperOnly = true$, v can only be assigned to a layer L_i such that

$$\sum_{k=0}^{i-1} \sum_{u \in L_k} u.Units \geq U$$

- $v.Prerequisite$: The prerequisite of v has the following input format:

  Course1
  AND
  (Course2 **OR** Course3)
  AND
  (Course4 **OR** Course5 **OR** Course6)

  This format is in the conjunctive normal form (CNF), which is an AND of ORs, and can be represented as:

  $$\{(Course1) \wedge (Course2 \vee Course3) \wedge (Course4 \vee Course5 \vee Course6)\}$$

  Note that the number of courses in each OR is not fixed. Based on this structure, we have two sequences A_v and B_v for a single course v. A_v represents an AND sequence, which holds a list of OR sets. Each OR set consists of a set of prerequisite courses for v. The OR set with index i in v's AND sequence is A_v^i. A prerequisite course u is in A_v if it is in one of the OR set of A_v.

  B_v in B stores information about how v is satisfied. In B_v, B_v^i refers to the OR set A_v^i in A_v. Initially, B_v^i is null. If a course u is assigned to the schedule, u is in A_v^i, and B_v^i is null, we set $B_v^i = u$, showing that OR set A_v^i is satisfied by u. If and only if all OR sets in A_v are satisfied, then the AND sequence A_v is fully satisfied, and the course's prerequisite is also fully satisfied. We call this a ready course because it is ready to be assigned to the schedule.

- $v.Successors$: A set of courses with OR set indices. For each pair of course w and OR set index i in $v.Successors$, there exists $A_w^i \in A_w$ such that $v \in A_w^i$.

- $v.Requirements$: A set of requirements that v can satisfy. Each element in the set is represented as a tuple (k, i) for k is a requirement name and i is the index of a set in R_k such that $v \in R_k^i$. Having this information in v can help find its satisfying requirements in the requirement table in linear time time after v is assigned.

- $v.CourseValue$: The number of requirements v meets, that is the length of $v.Requirements$.

- $v.DependentIndex$: The largest layer index of v's dependent schedule. Initially, $v.DependentIndex = 0$.

3.1.5 Graph

A course graph for course scheduling is a collection of courses. A course v has incoming edges from each course $u \in A_v$. Similarly, v has outgoing edges to each course $w \in v.Successors$. Note that edges can also be reversed to the find longest path distance required in 3.3.

We define graph to be $G = (V, E)$, a graph G with $|V|$ courses and $|E|$ edges. It is possible for a course graph to have cycles in the graph, but in our cases, we assume that G is a directed acyclic graph. Although it is not required to do the transitive reduction on G, doing so can decrease $|E|$ of G.

3.2 Single Course Assignment

For a course v, we define a layer L_i with $M(L_i)+v.Units < W(L_i)$ and (i mod 6) $\in v.QuarterCodes$ to be a valid layer of v. To simplify, we define two functions $Valid$ and $HasDependent$. $Valid(L_i, v)$ returns true if L_i is a valid layer for v. $HasDependent(L_i, v)$ returns true if $i < v.DependentIndex$.

In course scheduling algorithm, after a ready course v is selected, if the highest layer is valid, it marks this layer to be the lowest valid layer. Otherwise, it creates new layers above the highest layer until the new layer is valid, and marks this new layer as the lowest valid layer. Then it starts to scan the schedule from the original second highest layer and renew the lowest valid layer. Whenever it finds a layer that contains any courses in $v.Prerequisite$ or it reaches the upper bound layer, it stops scanning and assigns v to the lowest valid layer.

Algorithm 3.2.1 below shows the scheduling process for a single course v.

Algorithm 3.2.1: AssignCourse

Input : Course v, Schedule L, upperBound index u
Output: the index of the layer where v is assigned

```
1  step = |L| − 1
2  i = step
3  if Valid(L_i, v) = false or HasDependent(L_i, v) = true then
4  |  do
5  |  |  i = i+1
6  |  while Valid(L_i, v) = false and (v.IsUpperOnly = false or
   |  |  i ≥ u)
7  lastStep = i
8  step -= 1
9  while true do
10 |  if HasDependent(L_step, v) = true then
11 |  |  break
12 |  else if v.IsUpperOnly = true and step ≥ u then
13 |  |  break
14 |  else if  v.IsUpperOnly = false and step ≥ 0 then
15 |  |  break
16 |  else if Valid(L_step, v) = true then
17 |  |  lastStep = step
18 |  step = step -1
19 end
20 add new layers to L until |L| − 1 = lastStep
21 add v to L_lastStep
22 return lastStep
```

In algorithm 3.2.1, *step* represents the current scanning layer index. With 0-indexing, $|L|-1$ is the highest layer index in schedule L. *lastStep* represents the current lowest valid layer for course v.

To assign a single course, the algorithm checks at most $|L|$ layers in the schedule. The time complexity for algorithm 3.2.1 is $O(|L|)$.

3.3 Implementation Of Hu's Algorithm

Unlike the precedence-constrained multiprocessor scheduling problem, in which each task is ready if all of its immediate predecessors are assigned into the schedule, our algorithm does not require all courses in the prerequisite to be scheduled.

Although the relationship between each course is not a strict precedence-constrained relationship, inspired by Hu's algorithm, we can reverse the graph, and find the longest path from a course to another course that does not have any successors.

In order to reverse the graph G, we define for each prerequisite course $u \in A_v$, where v is another course, we have an edge (v, u) from v to u, instead of from u to v.

Because we prefer to assign those courses that can meet more requirements first, we also use a heuristic function for labeling. Label of v, $\alpha(u) = \alpha(v) + u.CourseValue$. If v is a ready course which is in the priority queue at the beginning, $\alpha(v) = v.CourseValue$. Because we view $v.CourseValue$ as the weight of edge (v, u), and we assume that the course graph G is an directed acyclic graph, we can use DAG shortest path algorithm with modifications to generate a topological ordering and find the longest path from a course to another course without any successors. When we are looping through all courses in G, we can also remove any courses that do not satisfy

any requirements. Algorithm 3.3.2 below illustrates this procedure.

Algorithm 3.3.2: Labeling

```
   Input  : graph G
   Output: labeled graph G
 1 for each course v ∈ G do
 2 |   if v.courseValue = 0 then
 3 |   |   G.remove(v)
 4 |   else
 5 |   |   α(v) = c.courseValue
 6 |   end
 7 end
 8 for each course v ∈ G in topological ordering do
 9 |   for each u ∈ A_v do
10 |   |   α(u) = max(α(u) + v.courseValue, α(u))
11 |   end
12 end
13 return G
```

In algorithm 3.3.2, the first for loop takes $O(|V|)$ time to initialize labels for all vertices in G, and the second for loop takes $O(|V| + |E|)$ time to visit all vertices and edges in G and correct label of each course. The total time required is $O(|V| + |E|)$.

3.4 Course Scheduling Algorithm

Due to the limitation of finding an upper bound without getting an actual schedule, we have to input a range of upper bounds to the algorithm first, and then generate $m - n + 1$ schedules. After that, we check the validity of schedules and pick the best schedule that has the minimum number of layers among all valid schedules it generates. Algorithms required for scheduling

are presented below:

Algorithm 3.4.3: CourseScheduling

Input : graph G, upper bounds range $[m, n]$, initial units $initUnit$, upper standing units $upperUnit$

Output: best schedule $bestL$

```
1  Labeling(G)
2  bestL = Null
3  for u ← m to n do
4      L = getSchedule(G, u)
5      totalUnit = initUnit
6      if |L| < |bestL| then
7          for i ← 0 to u do
8              if ∃v ∈ L_i such that v.IsUpperOnly = true and
                totalUnit ≤ upperUnits then
9                  reject L
10                 break
11             totalUnit = totalUnit + M(L_i) end
12             if L is not rejected then
13                 L = bestL
14     end
15     return bestL
16
```

Algorithm 3.4.4: GetSchedule

Input : graph G, upper bound index u
Output: Schedule L

Create priority queue Q using course label as key
Create schedule with an empty layer L
for *each course* $v \in G$ **do**
    if v *is a ready course* **then**
        Q.enqueue(v, v.label)
end
while Q *is not empty* **do**
    current = Q.extractMax()
    if *current satisfies any requirements* **then**
        assingnedIndex = AssignCourse($current, L, u$)
        ExpandQueue($current, Q, assingnedIndex$)
        for *each* $(k, i) \in v.Requirements$ **do**
            $R_k^i = max(0, R_k^i = R_k^i - 1)$
        end
end
return L

Algorithm 3.4.5: ExpandQueue

Input : course v, priority queue Q, assignedIndex *index*

for *each course* $u \in v.successors$ **do**
    allNotNull=true
    for $i \leftarrow 0$ **to** $|A_u| - 1$ **do**
        if $B_u^i = Null$ **then**
            if $v \in A_u^i$ **then**
                $B_u^i = v$
                $u.DependentIndex = max(u.DependentIndex, index)$
            else
                allNotNull = false
            end
    end
    if $allNotNull = true$ **then**
        Q.enqueue((u, $\alpha(u)$))
end

Algorithm 3.4.4 presents how it generates a schedule with a specific upper bound u. After the course v is assigned, algorithm 3.4.5 visits each successor u of v and updates B_u. If u is ready, it inserts u to the priority queue. The priority queue Q is implemented with a max-heap structure with course label as the key. Since each enqueue or dequeue operation takes $O(\log |V|)$ time and runs $O(|V|)$ times, the total time required on Q is $O(|V| \log |V|)$. In the while loop, each time AssignCourse 3.2.1 takes $O(|L|)$ time to assign a course, and ExpandQueue 3.4.5 visits all outgoing edges (successors) of the course. Therefore, it takes $(O(|V| \cdot |L| + |E| + |V| \log |V|)$ time to get a schedule. Suppose each layer holds at least one course, then $|L| \leq |V|$ and the total running time is $O(|V|^2)$.

Algorithm 3.4.3 shows how it gets the best schedule within a range of upper bounds from layer index m to n. The labeling phase in it takes $O(|V| + |E|)$ time. The time required for algorithm 3.4.4 still dominates, so the total time is $O(|V|^2)$.

4 Experimental Result

In the research, we implement the course scheduling algorithm using Python [1]. We verify that the algorithm is working correctly by using a small artificial dataset. Furthermore, with data collected on UCI website [2], we simulate plans for new students and continuing students based on some restrictions. We then explain the schedule generated by the algorithm and assess its quality.

In table 5 , 6 and 7 below, the first column, layer, shows the layer index. Course columns present the courses assigned to a layer. One layer corresponds to one quarter and the layer 0 is the starting quarter.

4.1 Example Graph Result

To better illustrate the algorithm, we use the same graph we have when we introduce the Coffman-Graham algorithm and Hu's algorithm, and then add restrictions to make it a graph for the course scheduling.

Information required for courses is described in table 4.

[1] https://github.com/jennyzeng/CourseScheduling

[2] http://catalogue.uci.edu/ and https://www.reg.uci.edu/perl/WebSoc

Name	Units	QC	Upper	prereq
A	4	{0,1,2,3,4,5}	false	[{B},{H,I,J}]
B	4	{1,3,4,5}	false	[{C,E},{D}]
C	3	{1,2,4,5}	false	[{G}]
D	4	{0,1,3}	false	[{F},{G}]
E	2	{0,1,3,4}	false	[{F,G}]
F	4	{0,3}	false	[]
G	2	{0,3}	false	[]
H	2	{3}	false	[{M}]
I	3	{1,2,4,5}	true	[{K}]
J	5	{0,1,3,4}	true	[{L}]
K	3	{0,3}	false	[{M}]
L	1	{0,1}	false	[]
M	4	{0,1,2,3,4,5}	false	[]

Table 4: Example Course Table

The third column, *QC*, represents the quarter codes, and the last column, *prereq*, shows the prerequisite information for the course. With the prerequisite information, we can get the successors information then.

There are two requirements for the schedules:

1. first requirement:
    - 1 course in {A}
    - 4 courses in {A, B, C, J, L}
    - 2 courses in {D, E, F, G}
2. second requirement:
    - 2 courses in {H, M, K, L, I}
    - 3 courses in {F, G, D, E}

Based on the input courses, we can get the graph in figure 3. For each course v in the graph, incoming edges with the same color indicate that they are in the same OR set for v's prerequisite. The number on the right of v shows the label of it.

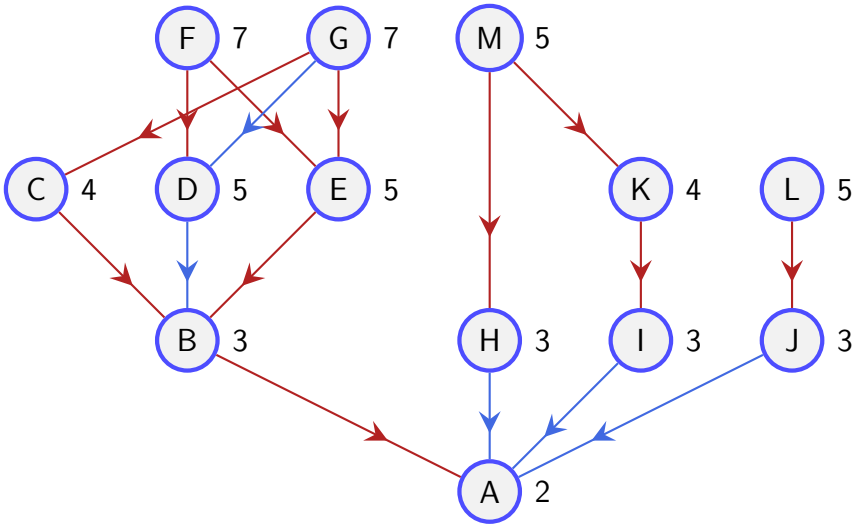

Figure 3: Labeling Process In Course Scheduling Algorithm

Table 5 below represents the schedule that the algorithm would output if the upper standing units requirement is 10. The last column, Max, in the table shows the width of each layer. Layers 2 to 4 do not have a specified width so they all have default width 10.

Layer	Courses			Max
0	F	G		6
1	D	L	C	8
2	M	J		10
3	B			10
4	A			10

Table 5: Scheduling Process In Course Scheduling Algorithm

Courses E, H, I and K are not in the schedule because when they are selected from the priority queue, the requirements they can meet are already satisfied.

4.2 Four Year Plan For A New Student

We test the quality of our algorithm by simulating a four year course schedule for a new student majoring in Computer Science and specializing in Intelligent Systems. Some other specifications are:

- Besides major requirement, the student is also required to satisfy General Requirements;
- The student take at most 13 units at the first quarter and at most 18 units for the rest of the quarter;
- The student has 0 default unit at the beginning of assignment;
- Upper bound U ranges from 0 to 10, inclusive;
- First quarter of schedule is 2017 fall quarter.

We get a four year schedule using the course scheduling algorithm for a new student. The result is represented in the table 6.

Layer	Courses			
0	ICS 31	MATH 2A	ICS 6B	ICS 90
1	ICS 32	MATH 2B	ICS 51	ICS 6D
2	ICS 33	STATS 67	WRITING-LOW1	GEII-1
3	ICS 45C	ICS 6N	CS 151	CS 122A
4	ICS 46	CS 178	GEII-2	HIST 40B
5	CS 177	HIST 40C	ICS 53+53L	POLSCI 21A
6	CS 161	CS 171	CS 141	HIST 40A
7	CS 116	CS 164	CS 112	CS 175
8	CS 165	IN4MATX 43	WRITING-LOW2	GEIII-1
9	GEIII-2	GEVI-1	GEVII-1	GEVIII-1
10	ICS 139W			

Table 6: Schedule For An Incoming CS Student

The algorithm generates a schedule with 10 layers, equivalent to 3 years and 2 quarters. It prefers to assign major courses rather than general requirement (GE) courses first because every GE course has a small label value: they only satisfy one requirement, and they do not have any successors. If we only focus on the makespan, the schedule has a good quality.

However, this schedule is too idealized and does not consider the work load of a single course. In the layer 9, all courses in it are General Requirement (GE) courses. This is because the algorithm assigns all major courses first to the schedule and later use the GE courses to fulfill any available slots in the schedule. In the layers 3 and 7, the schedule may be too stressful for students. In layer 3, it requires a sophomore to take 2 lower division major course and 2 upper division major courses. In the layer 7, all four courses are major courses, which can be stressful for many students. Moreover, it is recommended to take the lower division writing course in the first two years, but the algorithm assigned them to be some very late layers since they do

not satisfy many requirements. Adding priority for these courses manually may help select them earlier in the scheduling phase. A better schedule for a CS student should balance the number of layers and the workload.

4.3 Plan For A Continuing Student

Suppose that the user is a Junior student who has taken many courses in the school. Instead of getting a four-year plan, the user may prefer to map some courses that can meet the unsatisfied requirements to the schedule. We simulate a plan for a senior standing Computer Science(CS) student specialized in Intelligent Systems. After inputting the courses that the student has taken before, suppose the student still has the following requirements:

- Need 1 course in GEII requirement;
- Need 2 courses in GE-IV requirement;
- Need 1 course in GE-VII requirement;
- Need 1 course in {CS 177, CS 179} (CS 177 or CS179) to fulfill the requirement for the specialization in Intelligent Systems;
- Need ICS 53+53L to fulfill the requirement for Lower-Division CS major courses;
- Need 1 history course in {HIST 40A, HIST 40B, HIST 40C} (HIST 40A or HIST 40B or HIST 40C) to fulfill the University Requirement;
- Need ICS 139W to fulfill the CS upper writing requirement;
- Need 3 more Upper-Division CS major courses.

Note that all courses in the requirement for the specialization in Intelligent Systems are also in the Upper-Division requirements, and a history course can fulfill both the University Requirement and the GE-IV requirement. Assume the student takes 16 units per quarter, and the first quarter is 2017 Fall Quarter. We set the upper bound layer to be the layer 0 because the student has an upper standing. Table 7 shows the schedule generated by the course scheduling algorithm for this student.

code	courses			
0	CS 141	CS 177	HIST 40A	GEII-2
1	HIST 40B	ICS 53+53L	GEVII-1	
2	ICS 139W			

Table 7: Schedule For A Junior CS Student

This schedule is optimal because it prefers to pick those courses that can meet a maximum number of requirements (e.g. CS 177 and HIST 40A).

5 Conclusion And Future Work

Through examining two task scheduling algorithms, the Coffman-Graham algorithm, and Hu's algorithm, we are able to develop our scheduling algorithm that works for course scheduling in time $O(|V|^2)$. Looking at the experimental result, the schedule we generate for a student is valid and achieve a good makespan. With this algorithm, programmers can develop an application to assist students and counselors in making a study plan.

However, building such a schedule is still time and space consuming, and applying better data structures may reduce the total running time. Furthermore, in this report, we can only discuss the quality based on the experimental result and assess it by comparing it with the schedule a student would choose in reality. It is still important to find a way to calculate its quality theoretically.

Although the result is workable for a Computer Science student, it is still too idealized. We should develop a better heuristic function in labeling process to get a more accurate priority for ready courses. For instance, we can change labels of some courses every time when a course is assigned to the schedule but increase the total running time of the algorithm.

If we are able to access the database for courses and student requirements for a college, we may get a better understanding of the relationship between courses, and how requirements are satisfied. Rather than collect this part manually, we may get the major requirements directly, and test the quality of schedule on many different majors. Through getting the real plans of stu-

dents, we can better assess and improve the quality of our course scheduling algorithm.

References

[1] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Chapter 9: Layered Drawings of Digraphs," in *Graph Drawing: Algorithms for the Visualization of Graphs*, pp. 265–302, Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1998.

[2] A. V. Aho, M. R. Garey, and J. D. Ullman, "The Transitive Reduction of a Directed Graph," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 131–137, 1972.

[3] R. Tamassia, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007.

[4] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[5] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, vol. 9, no. 6, pp. 841–848, 1961.

[6] J.-T. Leung, "Some Basic Scheduling Algorithms," in *Handbook of Scheduling*, Chapman & Hall/CRC Computer & Information Science Series, Chapman and Hall/CRC, Apr. 2004. DOI: 10.1201/9780203489802.ch3 DOI: 10.1201/9780203489802.ch3.

[7] S. Lam and R. Sethi, "Worst Case Analysis of Two Scheduling Algorithms," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 518–536, 1977.

[8] J. A. M. McHugh, "Hu's precedence tree scheduling algorithm: A simple proof," *Naval Research Logistics Quarterly*, vol. 31, no. 3, pp. 409–411, 1984.

[9] Y. Huo and J. Leung, "Online Scheduling of Precedence Constrained Tasks," *SIAM Journal on Computing*, vol. 34, pp. 743–762, Jan. 2005.