Cameron Maloney
CSC 349, Theresa Migler
Cal Poly

March 5, 2024

PRACTICAL ALGORITHMS PAPER — Finding ideal course path

1 Problem Description

There are few experiences that are truly universal among university students. Eating, sleeping, going to class (for some), and doing homework may be found in the short list of shared experiences, but even the last two are not necessarily universal. However, I would argue that every student at one time or another has to go through the potentially painful process of choosing classes. Planning future classes can be a daunting task, especially at a university like Cal Poly that offers a wide array of classes in different subject areas. Given the various requriements (Major courses, support courses, GEs, GWR, USCP, electives, minors, etc.), it can be difficult to find what combination of courses is best for finishing a major as quick as possible. For some, finishing their major on time is a challenge, and many find it necessary to work with advisors to navigate through the marshlands of the Cal Poly catalog and make prudent course decisions.

It would be nice if there was a tool that could take in all of the classes that a student has taken, the student's major(s)/minor(s), the time of year, and the amount of courses a student is willing to take in each quarter/season, and output an ideal course path for the student. That is precisely what I have decided to build here. Given that this is an "alpha version" of this tool, I have hard coded some parts so that it tailors to my situation (the courses I have taken, the current time of year, etc.). However, if anyone finds that this tool would be helpful to them, it would be easy to generalize it so that any student may use it.

Note: I will interchange seasons and quarters, but I am using them to refer to the same idea of a period in which certain courses are offered and can be taken. Also, note that my algorithm currently does not keep track of prerequisites as I did not find it necessary for my current situation, but that would also be an easy addition should this tool be generalized in the future.

2 Problem Statement

In order to solve this problem, I made use of multiple algorithms. The code I provided contains 4 different ways of finding a course path, and 2 of them produce ideal solutions (least number of courses) while 2 of them approximate ideal solutions:

1. The first method is a naive course path finder that simply splits the courses up by the seasons/quarters they are offered, then finds a random course list that fits the season requirements in a way that each course would fulfill a requirement that the student has. I included this algorithm to show how choosing courses at whim and not optimizing one's course schedule to choose courses that fulfill multiple requirements can be detrimental if one wants to finish their major on time. In my case, I plan to graduate in Spring of 2025, but using this method would likely cause me to need anywhere from 20 to 32 extra units, meaning I would potentially have to graduate in Spring of 2026.

2. The second method is a brute force search. This method finds an optimal course path, but takes a while to do it. When I say "a while," I mean that most students would have to wait at least for the universe to end for the algorithm to finish, but most would have to wait much much longer than even that. And keep in mind that the brute force algorithm I implemented only tries one season distribution...

3. The third algorithm is one that I call "divide and DP." Basically, using the current time of year and the user's desires for how many courses they are willing to take in each future season, the algorithm will begin by dividing up the course load into seasons. Then, it will perform a dynamic programming algorithm to find the ideal course list for that season given the user's past course information and

requirements they need to fulfill. This does not always produce an ideal course path, but it tends to get close and sometimes reaches one. The reason that this algorithm does not always produce an ideal course path is that for each season, the algorithm is restricted to finding an ideal course path for that specific season.

As an example, the algorithm may find that a specific set of 4 courses in spring satisfies 7 requirements, which is the max that a set of courses can fulfill in spring, so it returns that course list as the ideal. Later, the algorithm finds a course list in fall that fulfills 6 requirements (none of the previous 7), so it returns that as the ideal. However, let's say there is a course in fall that fulfills 3 out of the 7 requirements already fulfilled in spring (this is that case with some courses that fulfill elective, GWR, and USCP requirements). Well now this course won't be considered in fall as the requirements have already been fulfilled in spring, even though the optimal course path would have included that course.

4. The last algorithm is the real algorithm that I will be analyzing in this paper. It is actually composed of two separate algorithms, one for finding the ideal course path, and one for distributing the courses among seasons in an optimal way. I call this algorithm "DP and divide" because it does the reverse of the previous algorithm. It first finds an ideal course list without considering seasons, and then splits that course list into seasons. With the current implementation, it is possible that the ideal course list found does not fit into the user's season requirements, but the algorithm could be easily modified so that it finds the ideal course list that fits the user's season requirements. Regardless, I will explain below the algorithms in their full form, and then simplify the algorithms to a more easily understandable version.

2.1 Finding Ideal Course Path

To solve this problem, we can ignore the seasons and focus on just the courses and the requirements they fulfill. An ideal course path fulfills the most amount of requirements using the least amount of courses. We can treat the amount of requirements that a list of courses fulfills as its value. We can treat each course as having a weight of 1. We want to maximize the value while minimizing the weight. This is very similar to the knapsack problem, which we can solve using dynamic programming. Each entry in our table will need to keep a list of its respective requirements fulfilled (for the dynamic programming formula) and courses used (for the final return value).

2.2 Finding Ideal Way To Distribute Courses Among Seasons

2.2.1 Original problem

We are given 4 quarters, each of which has a capacity (the capacity is the number of courses that the student is willing to take in each quarter). If the student is more than a year out from graduating, they will have to add the capacities for the quarter in this year and for all the rest of the years up until graduation. In my case, I want to take 5 courses in spring 2024 and 4 in spring 2025, so my capacity for spring is 9. However, we will see later that the algorithm that I design to solve this problem allows for users to specify exact capacities for each quarter as many years out as they would like.

Now, we are also given a list of courses that we must somehow distribute optimally among the quarters. I will show soon how this cannot be solved by simply looping through the courses and placing them in seasons.

2.2.2 Simplifying the problem

I want to simplify this problem a bit to get a clearer understanding. Let's treat courses as items and seasons as bins. Let's assume we have n items and k bins. Note that I use 4 in my scenario, but more can easily be added. Also note that I currently have the constraint that each item can be placed in at most one bin, but I am generalizing this problem to make it so that constraint can easily be removed and the problem will remain similar (for those who want to retake a course multiple times). The problem comes down to the following question:

2.2.3 Formalizing the problem

Given n items and k bins, where each bin i has a capacity k_i and each item j has a list of bins L_j that it is allowed to go into, the problem is to find a placement of the items into the bins that maximizes the number of placed items.

Let:

$$\begin{aligned} n &: \text{ Number of items} \\ k &: \text{ Number of bins} \\ k_i &: \text{ Capacity of bin } i, \quad i = 1, 2, \ldots, k \\ L_j &: \text{ List of bins that item } j \text{ is allowed to go into, } \quad j = 1, 2, \ldots, n \\ x_{ij} &: \text{ Binary variable indicating whether item } j \text{ is placed in bin } i \end{aligned}$$

The objective is to maximize the total number of placed items:

$$\text{Maximize} \quad \sum_{j=1}^{n} \sum_{i=1}^{k} x_{ij} \tag{1-1}$$

Subject to the constraints:

$$\text{Each item can be placed in at most one bin:} \quad \sum_{i=1}^{k} x_{ij} \leq 1, \quad \forall j \tag{1-2}$$

$$\text{The capacity of each bin cannot be exceeded:} \quad \sum_{j=1}^{n} x_{ij} \leq k_i, \quad \forall i \tag{1-3}$$

$$\text{A item may only be placed in a bin it is allowed to go into:} \quad x_{ij} = 0, \quad \forall i, j \text{ where } i \notin L_j \tag{1-4}$$

$$\text{Binary constraints:} \quad x_{ij} \in \{0, 1\}, \quad \forall i, j \tag{1-5}$$

2.2.4 Explaining the complexity of the problem

It may seem that one can simply iterate over each bin and then iterate through the courses and place the courses accordingly. This, however, does not work, no matter how you iterate through the bins.

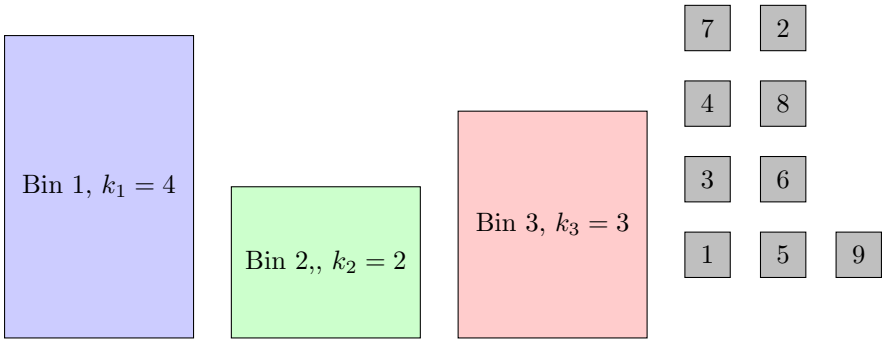


Figure 1: Simple example with 3 bins and 9 items.

Imagine a simple example with 3 bins. Bin 1 has capacity 4, bin 2 has capacity 2, and bin 3 has capacity 3. There are 9 items. Items 1 and 2 can go into bins 1 and 2. Items 3 and 4 can go into bins 1, 2, and 3. Items 5 through 6 can go into bins 2 and 3. Item 7 can only go into bin 1. Item 8 can only go into bin 2. Item 9 can only go into bin 3.

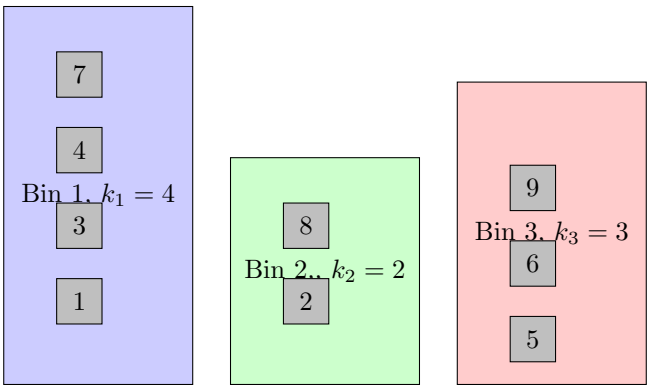


Figure 2: Optimal placement of items in the bins.

One can optimally place the items by placing item 1 in bin 1, item 2 in bin 2, items 3 and 4 in bin 1, items 5 and 6 in bin 3, item 7 in bin 1, item 8 in bin 2, and item 9 in bin 3.

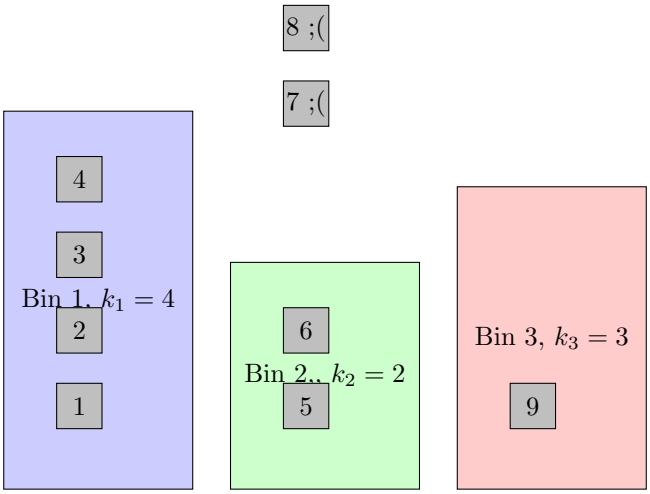


Figure 3: Suboptimal placement of items in the bins (for iteration starting at bin 1.)

However, let's assume you iterate through the bins then iterate through the items. If your iteration starts with bin 1, you will place items 1-4 in bin 1, and will leave item 7 out. If your iteration starts with bin 2, you will place items 1-2 in bin 2, and leave item 8 out. If your iteration starts with bin 3, you will place items 3-5 in bin 3, and leave item 9 out. Note: Items get sad when they feel excluded from bins!

In theory, rearranging the way that one iterates through the items would eventually yield an iteration that produces an optimal distribution of items. However, the time complexity for this algorithm would be $\mathcal{O}(n!)$, so we need a better option.

3 Algorithmic Solution

3.1 Dynamic Programming Algorithm to Find Ideal Course Path

I use a dynamic programming algorithm to solve this problem. Note that a "requirement" could be anything from "csc349" to "GWR" to "GE Upper-Division D." The definition, base cases, solution, and formula are as follows:

Definition:

$$\text{dp}[i, j] = \text{Max number of requirements that can be fulfilled with } i \text{ courses using course } j \tag{1-6}$$

Base case(s):

$$\text{dp}[1, j] = \text{number of requirements fulfilled by course } j \tag{1-7}$$

Solution:

$$\text{Max}\{\text{dp}[\text{numCourses}, j]\} \quad \text{for } 0 \leq j \leq \text{size of course list} \tag{1-8}$$

Formula:

$$\begin{aligned} \text{dp}[i, j] = \text{Max}\{&\text{dp}[i-1, k] \\ &+ \text{number of unique requirements fulfilled by course } j \\ &(\text{not already fulfilled by the courses in dp}[i-1, k])\} \end{aligned} \tag{1-9}$$

Note: *numCourses* is an input provided by the user for how many courses into the future they want to consider. If *numCourses* is greater than the number of courses in the ideal course path found, the algorithm trims the extra courses found at the end.

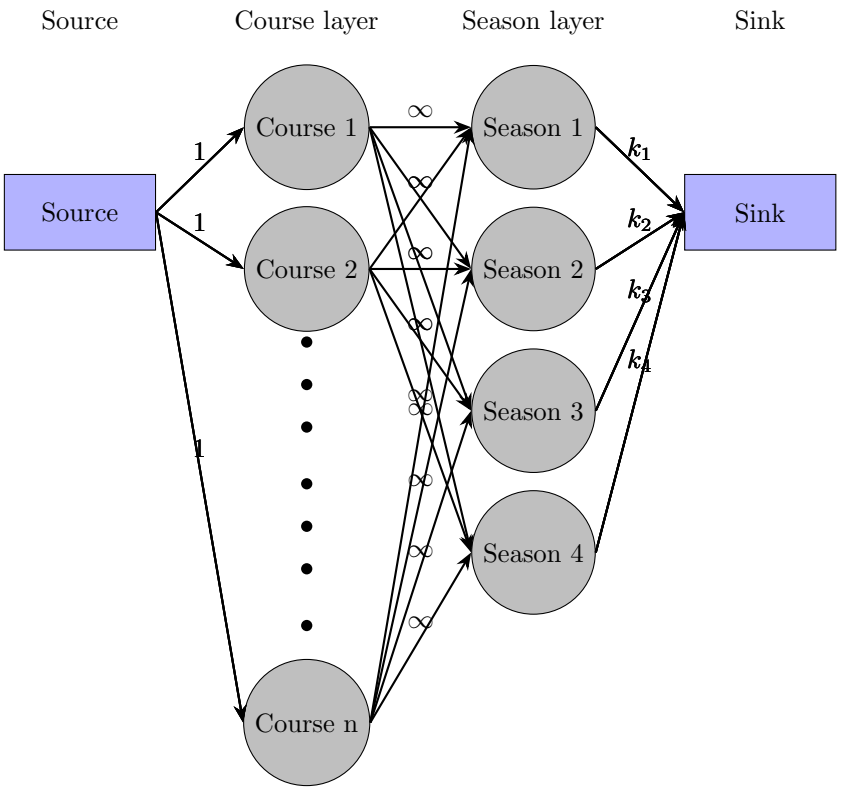
Each item dp[i, j] also stores a respective "list of requirements fulfilled" and "list of courses taken" with it, which are necessary for the formula and the returning of the final result.

3.2 Ford-Fulkerson method (Specifically Edmonds-Karp Algorithm) for Optimal Course Distributions

The problem stated above is modeled as a graph where:

- Each course/item is a node. Each season/bin is a node. There are also two extra nodes, a source node and a sink node.
- The source node is connected by edges to the course nodes with an outgoing edge of capacity 1.
- The course nodes are connected by an edge to season nodes with an outgoing edge of infinite capacity.
- The season nodes are connected by an edge to the sink node with their corresponding capacities.
- Edges of capacity 0 are added in opposite directions to each of the previously mentioned edges (these will serve as back edges, which are a necessary component of the algorithm).

The following is an example of a graph with a source and sink node, 4 seasons, and n courses. More seasons can be added as needed. Note that I am not including back edges at the moment as the back edges are not important until the algorithm begins.



This graph can be treated as a flow network, where we want to find the max flow in the graph. Given that the source is connected to the sink by edges to the course nodes with capacity 1, the max flow in the network will be the flow that passes through the max number of courses. No courses will be repeated (in this version of the graph) since the capacity from the source to each course is 1. Similarly, season capacities will be respected as once a season's capacity is met, there will be no more paths from the source to the sink through that season.

After setting up this graph, the max flow in the network is found using the Ford-Fulkerson algorithm/method. Then, for each course node, we analyze each of its outgoing edges and find the one with positive flow. The node on the other end of the edge with positive flow will be the node corresponding to the season where we should place the course node. This leads us to populate all course lists optimally.

Now, I will explain the algorithm in more detail:

I use the Ford-Fulkerson method (specifically the Edmonds-Karp algorithm) to find the max flow in the flow network The algorithm involves traversing the residual graph, finding augmenting paths using BFS, and updating the flow in the network until no more augmenting paths can be found. Here I will explain some terms:

Augmenting Path:

• An augmenting path is a path in the residual graph from the source to the sink along which additional flow can be sent. Starting from the source, the algorithm searches for an augmenting path using BFS (Breadth-First Search) in the residual graph. The presence of an augmenting path implies that there is room to increase the flow in the network.

Residual Graph:

• The residual graph is a modified version of the original flow network (the graph passed in as a parameter) that accounts for the current flow. It contains edges with capacities that represent the remaining capacity for additional flow. For each original edge in the network, the residual graph contains a backward edge with capacity equal to the current flow on the original edge. More specifically, consider an edge (u, v) with capacity C and current flow F. In the residual graph, the forward edge (u, v) has residual capacity C - F and the backward edge (v, u) has residual capacity F. The residual graph is updated during each iteration of the algorithm based on the flow adjustments along the augmenting path. Here's some pseudocode to explain

how exactly the residual graph is updated:

```
for each edge (u, v) in the augmenting path:
    residual_forward_edge_c(u, v) = capacity(u, v) - flow(u, v)
    residual_backward_edge_c(v, u) = flow(u, v)
    flow(u, v) += min_capacity
    flow(v, u) -= min_capacity
```

4 Conclusion

After using the algorithm, I was able to find my specific ideal course list. I used this to sign up for courses for next quarter. You can run the algorithm yourself using the code in the github link I provided. You can also see detailed comments for all parts of the code, especially the algorithms.

I found that the use of algorithms provided a significant improvement over the naive method of simply choosing courses that meet requirements randomly. I also found it to be much faster than brute force searching all course combinations (in my case, it takes about 2 seconds to run, so I do not have to wait for the universe to end).

I have provided a link to a github repository with my code, as well as some other links that I used to help me throughout the process (the most helpful of which was the cal poly catalog, which I used to create Course objects for all Cal Poly courses). Overall, I believe that with a little more time to generalize this tool, it could even be something that Cal Poly offers as a useful option for students who are looking for the quickest way to finish their majors/minors.

5 Addendum - useful links

Ford-Fulkerson Algorithm page on Brilliant

Edmonds-Karp algorithm page on Wikipedia

Cal Poly course catalog

Github repository that contains the code for the algorithms I mentioned above

Submitted by Cameron Maloney on March 5, 2024.