Optimal Course Allocation Using A* Search With Branch and Bound Techniques

Apoorv Khandelwal
Electrical and Computer Engineering
Carnegie Mellon University
akhandel@andrew.cmu.edu

Rohan Aletty
Electrical and Computer Engineering
Carnegie Mellon University
raletty@andrew.cmu.edu

May 3rd, 2013

Abstract

In this paper, we discuss our research on the problem of course allocation. Course allocation can be defined as finding fair and optimal schedules for students given finite capacity courses and students who have heterogenous preferences over those courses. We use A* search using different upper and lower bounds as our heuristics to search for the optimal course allocation and see how different combinations of these bounds perform.

1 Motivation

Course allocation is a difficult problem which has no easy solution. It is natural to limit the number of students that can take a certain course because the learning experience and attention that a student receives from the course is directly related to the number of people in it. However, how should these finite number of seats in a course be allocated to the many students that may want to take it? The optimal solution (assigning a schedule of classes to each student) must attempt to maximize the utility that each student gets from the classes in his or her schedule while being as fair as possible to all students.

Because most universities do not currently have an automated way in which they assign schedules to students, a mechanism that can automate this process would save the time and energy spent doing this manually.

2 Previous Work

There has been much work done in the past dealing with course allocation. One important paper that we will discuss is Finding Approximate Competitive Equilibria: Efficient and Fair Course Allocation [Othman and Sandholm, 2010]. They discuss the Approximate CEEI Mechanism (A-CEEI) which provides a similar way to allocate courses in an optimal manner. The mechanism is a two level search in which the first level (called the master level) searches for an appropriate price level for each course. The other level is the agent level where the agents search for the most preferred affordable bundles given the prices set in the master level search. We will talk about the mathematical details in the subsection below and then go into how our problem is different.

2.1 A-CEEI Theory

We start by assuming that there are N students and M courses. The courses have capacities $q_1, ..., q_j ..., q_M$ and the students have sets of permissible schedules $\Psi_1, ..., \Psi_i, ..., \Psi_N$ which are sets of M-length bit vectors where each vector describes which courses a student would be taking (1 if course i is in the schedule and 0 if not) in that specific scedule. The capacities represent the constraints of the course size and the permissible schedules represent the scheduling constraints of the students (classes at the same time, not meeting prerequisites, etc.) which would not allow certain combinations of classes to go together. In addition, the students have a utility function u_i which maps each schedule in Ψ_i to a real number (preference).

The following procedure details how course allocation would work with this setup in A-CEEI:

1. Students give their preferences as (for each student i) $\hat{u}_i$ representing their preferences over all schedules in Ψ_i.
2. Give each student i a randomly assigned budget b_i^*. Randomness parameter dependent on number of students and max number of courses possible to take.
3. Compute prices for the courses $\{p_j^*\}_{j=1}^M$ and schedules for each student $\{x_i^*\}_{i=1}^N$ such that:
   - Students maximize utility with the chosen schedule, based on their preferences and budget constraints (here x_{ij} is the j^{th} element in that specific schedule for student i)

$$\forall i : x_i^* = \text{arg max}_{x \in \Psi_i} \hat{u_i}(x_i) : \sum_j x_{ij} p_j^* \leq b_i^*$$

   - Minimize the magnitude of the market-clearing error. This basically means that each course should be filled up as much as possible. Formally, this is a minimization of

$$\alpha = \sqrt{\sum_j \xi_j^2}$$

where ξ_j is the following quantity:

$$\xi_j = \begin{cases} \sum_i x^*_{ij} - q_j & \text{if } p^*_j > 0 \\ \max\left[(\sum_i x^*_{ij} - q_j), 0\right] & \text{if } p^*_j = 0 \end{cases}$$

2.2 Differences From A-CEEI

Our representation of the course allocation problem is similar to the one above but there are some key differences. The most prominent difference is that we are modeling the allocation as a multi-item auction. Instead of having a set of predetermined possible schedules (Ψ_i above), we instead bound the student by a minimum and maximum number of units and let them bid on courses that they would want. Thus, rather than have seats in the course be bought with a specific price, students can place possibly different bids on a seat for a course and both get it. With this formulation, it is also possible (and somewhat the point) that students can bid on bundles of courses. Fundamentally this inclusion makes sense as the student can model his bids to represent his inter-course preferences. If a student only wants to take a certain tough course with an easy course, he can bid only on that bundle. This way, the only way the tough course will appear on the schedule is if it comes along with the easy course.

Another difference is that rather than randomly budgeting each student with money, we give each student varying amounts of money depending on how much we value the preferences of certain students. For example, we can give senior students more money to buy courses with than freshman students so that a hierarchy in preference can be created. In our formulation, the students also have hard constraints on the number of units they can take rather than the number of courses. This allows us to weight courses with different units to represent the difficulty of the course.

3 Our Problem

In our setup of the course allocation problem, we assume that there are courses that have a finite capacity and can only have a number of students up to that capacity. Also, students have preferences over which courses they want to take and are also given some amount of money. Each student can place bids on sets of courses with this money. Since each student may want to take a different number of classes, they can enter minimum and maximum number of course units that they wish to take. Given these inputs from the students and the prior knowledge of the course units and size capacity, our mechanism will output an allocation of courses to students that will maximize the total utility over all students.

3.1 Formalized Problem

In the problem, we assume that there are N students (labeled as S_i). Each student has the following:

- M_i money to spend on classes (simulating his or her preferences for bidding)
- f_i, g_i as a lower and upper bound, respectively, on the number of units desired
- b_i total bids associated with the sets of courses that he or she wants to take
  - Bids will be represented by the set of desired courses with the money the student is willing to pay for them
  - Written as $\{T_{i,j}, m_{i,j}\} \implies$ student i's j^{th} bid is $m_{i,j}$ money on courses $T_{i,j}$

It is important to note in the last point above that $\forall i, \sum_j m_{i,j} \leq M_i$ (the sum of the money used in each bid that a student makes sums to less than or equal to the total amount of money that the given student has).

On the other side of the coin, there are P courses (labeled C_k) that can each be represented as having the following:

- s_k as a size limit, the maximum number of students that can be in course C_k
- u_k as the total number of units that course C_k has

In the multi-item auction, we will be auctioning off an s_1 number of C_1's, an s_2 number of C_2's, and so on for each C_i until either no more students have bids on C_i or the size limit of C_i has been reached. In order to categorize whether a student got placed in a course or not, we can create the binary variable x_{ij} which is 1 if student i receives bid j and 0 otherwise.

Our problem can now be defined as follows:

$$\arg\max_{x_{i,j}} \sum_{i=1}^{N} \sum_{j=1}^{b_i} m_{i,j} x_{i,j}$$

$$\text{constraint set 1: } \forall i \in \{1, ..., N\},\ \sum_{j=1}^{b_i} x_{ij} |T_{i,j}| = |\bigcup_{\substack{j=1,\\ j \text{ s.t.}\\ x_{i,j}=1}}^{b_i} T_{i,j}|$$

$$\text{constraint set 2: } \forall i \in \{1, ..., N\},\ f_i \leq \sum_{j=1}^{b_i} x_{i,j} \sum_{k=1}^{P} u_k 1_{C_k \in T_{i,j}} \leq g_i$$

$$\text{constraint set 3: } \forall k \in \{1, ..., P\},\ \sum_{i=1}^{N} \sum_{j=1}^{b_i} x_{i,j} 1_{C_k \in T_{i,j}} \leq s_k$$

3.2 Explanation of Problem

Simply put, we are maximizing the total amount of money spent by students (the total number of student preferences met in order of weight). The first constraint set ensures that no course is repeated among all the bids that are accepted for a given student. The second constraint set limits the students into their desired unit count for a semester. The third constraint set limits the total number of students that can take a certain class to the size limit of that class. Another cool thing that we can do to weight certain students' preferences over others is to give the preferred students more money for bidding. This way, they can naturally bid more for courses that they want than other students can.

We are attacking this problem using branching on bids. Branching on items would have many redundant nodes in the search path since each course C_i would be represented s_i number of times. To branch on bids, we are following this sequence: a descending order of $\frac{m_{ij}}{|T_{ij}|^\alpha}$, where we let $\alpha = 1$. This heuristic is somewhat greedy in that it looks for bids that are spending the most per course in the bid.

3.3 Solution Search and Bounding

To find the optimal allocation of course seats to students, we are using A* Search combined with the branch and bound technique to prune the search space. A* Search uses a heuristic function $h()$ that combines previous knowledge and an estimate of remaining distance to the solution to determine how to traverse the search tree. Since we are trying to maximize the total amount of money spent by all students, our past knowledge can be represented as the total amount of money that we have collected from bids so far. Our estimation on how much money can be collected in the future through accepted bids is represented by a lower and upper bound on the solution at each node.

The upper bound computed at each node, first of all, is necessary for maintaining our priority queue of nodes in the frontier of the A* search. Also, if the upper bound computed at a node is lower than our best found solution so far in the search, then we can prune that path of our search (any solution we find in this path will be less than the upper bound, which is less than our current solution). Furthermore, computing a lower bound at each node can allow us to increase our best found solution so far (any solution we find in this path would be greater than the lower bound, which is greater than our current best solution so far in the search). This can lead to faster pruning later because the upper bound may be lower than this currently increased global optimum.

The search for an optimal scheduling can be narrowed by keeping a lower bound and upper bound on the solution. We initially implemented naive lower and upper bounds. However, these bounds were too naive and simple and thus, did not prune our search space. So instead, we moved on to the following sophisticated bounds, taking the maximum of the lower bounds and minimum of the upper bounds.

Lower Bounds:

- Greedily add remaining bids in decreasing order of (price / number of courses) while enforcing all constraints
- Greedily add remaining bids in decreasing order of (price / number of units) while enforcing all constraints

Upper Bounds:

- For each course, fill all its remaining spots by choosing the maximum-priced remaining bids that contain that course
- For each student, break up his remaining bids into unit-sized chunks and allot the students remaining units with the maximized-priced chunks

4 Test Cases

To test whether the functioning of our solver, we first devised a couple test cases. The first test case is a basic case where all students get their first preferences. The next case has some students not get their first preferences. Our basic tests contain only bids on single courses.

4.1 Test Case 1

```
10 students => S1, ... , S10
10 classes => C1, ... , C10
capacity: 2 students each => s1 = 2, ... , s10 = 2

for all k = 1:P,  u_k = 2
for all i = 1:N,  f_i = 4, g_i = 4

money vectors (M_is) => (bids)
Mi = [B1, ... , B10]

M1 = [0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.3]
M2 = [0.3, 0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05]
M3 = [0.05, 0.3, 0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05]
M4 = [0.05, 0.05, 0.3, 0.3, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05]
M5 = [0.05, 0.05, 0.05, 0.3, 0.3, 0.05, 0.05, 0.05, 0.05, 0.05]
M6 = [0.05, 0.05, 0.05, 0.05, 0.3, 0.3, 0.05, 0.05, 0.05, 0.05]
M7 = [0.05, 0.05, 0.05, 0.05, 0.05, 0.3, 0.3, 0.05, 0.05, 0.05]
M8 = [0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.3, 0.3, 0.05, 0.05]
M9 = [0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.3, 0.3, 0.05]
```

```
M10 = [0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.05, 0.3, 0.3]

Optimal Allocation
C1 => S1, S2
C2 => S2, S3
C3 => S3, S4
...
C9 => S9, S10
C10 => S10, S1
with maximized score = 6
```

4.2 Test Case 2

```
(similar to example 1)

Money for each student:
m1 = 3, m2 = 3
m3 = 2.5, m4 = 2.5
m5 = 2, m6 = 2
m7 = 1.5, m8 = 1.5
m9 = 1, m10 = 1

S1 => wants C1, C2, C7
S2 => wants C2, C5
S3 => wants C2, C8, C9
S4 => wants C2, C6
S5 => wants C3, C9, C10
S6 => wants C4, C9
S7 => wants C4, C3
S8 => wants C2, C8, C6
S9 => wants C1, C7, C10
S10 => wants C1, C5

M1 = [1.1,  0.9,  0,  0,  0,  0,  1,  0,  0,  0]
M2 = [0,  1.5,  0,  0,  1.5,  0,  0,  0,  0,  0]
M3 = [0,  1,  0,  0,  0,  0,  0,  0.6,  0.9,  0]
M4 = [0,  1.3,  0,  0,  0,  1.2,  0,  0,  0,  0]
M5 = [0,  0,  0.5,  0,  0,  0,  0,  0,  0.7,  0.8]
M6 = [0,  0,  0,  1,  0,  0,  0,  0,  1,  0]
M7 = [0,  0,  0.7,  0.8,  0,  0,  0,  0,  0,  0]
M8 = [0,  0.5,  0,  0,  0,  0.5,  0,  0.5,  0,  0]
M9 = [0.2,  0,  0,  0,  0,  0,  0.4,  0,  0,  0.4]
M10 = [0.4,  0,  0,  0,  0.6,  0,  0,  0,  0,  0]

Optimal Allocation
C1 => S1, S10
C2 => S2, S4
C3 => S5, S7
C4 => S6, S7
C5 => S2, S10
C6 => S4, S8
C7 => S1, S9
C8 => S3, S8
```

```
C9 => S3, S6
C10 => S5, S9
with maximized score = 16.7
```

5 Results

Apart from hard-coded test cases, we also decided to test our solver on randomly generated problems. We were interested in generating the following parameters:

- Number of students among whom courses must be allocated
- Number of distinct courses to be allocated among the students
- For each course, the maximum student capacity the course can handle
- For each student, the minimum number of units he can be allocated
- For each student, the maximum number of units he can be allocated
- For each student, the amount of money he has with which to bid on courses
- For each course, the number of units the course contributes to a student taking the course
- For each student, a measure of the price for each of his bids
- For each student, a measure of the number of units for each of his bids

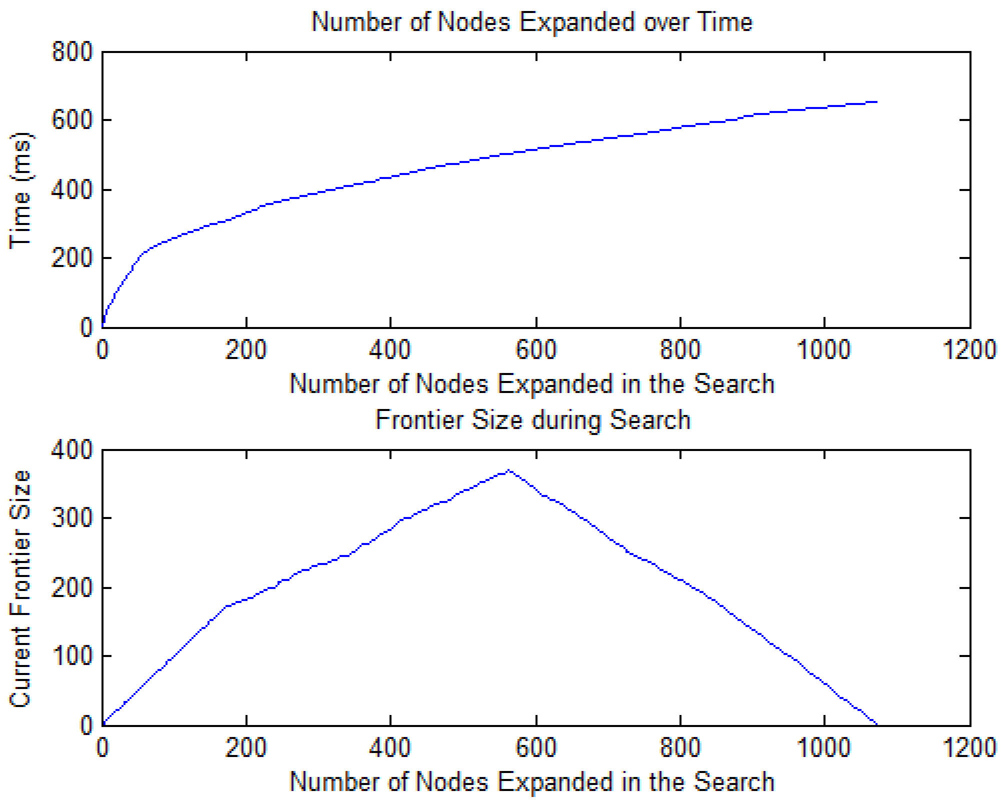
We had two methods for generating likely parameters for the students' preferences and course constraints. One way to generate these is via a maximum entropy model (specifying a minimum and maximum for the constraint and letting the parameter take a value uniformly in this range). A more reasonable approach, we found, was by invoking the Central Limit Theorem. Because each parameter's value can be thought of as the sum of numerous independent random variables, we also generated these parameters by specifying a mean and standard deviation for each one and sampling the corresponding Gaussian distribution. While specifying the means for each of the parameters, we used the following heuristic (it is an inequality because bids can overlap in the courses they contain):

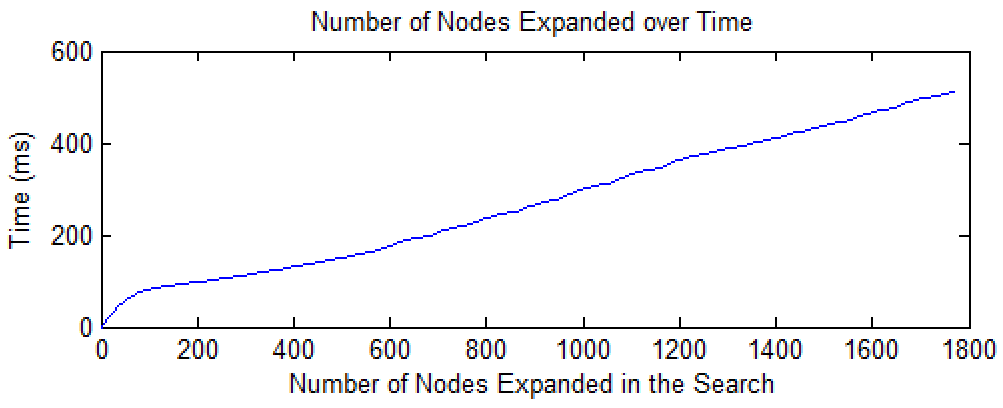
$$courses\left(\frac{capacity}{course}\right) \leq people\left(\frac{bids}{person}\right)\left(\frac{courses}{bid}\right)$$

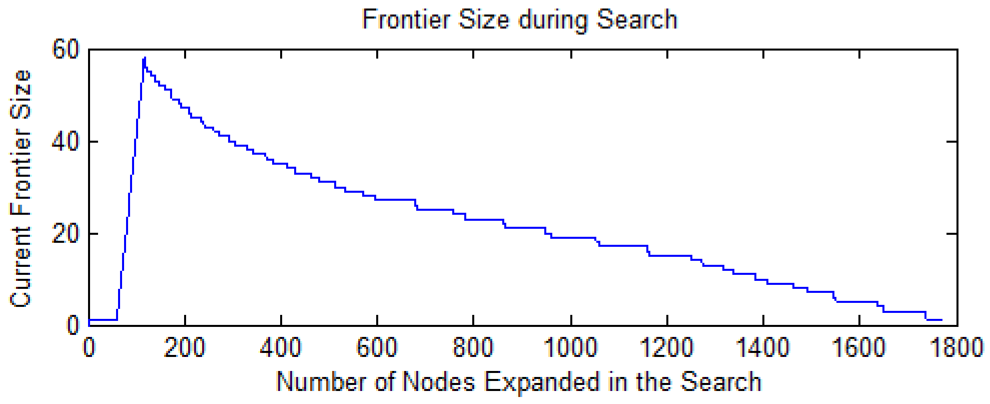
$$\text{where } \frac{bids}{person} \approx \left(\frac{money}{person}\right)\left(\frac{bids}{money}\right) \text{ and } \frac{courses}{bid} \approx \left(\frac{courses}{unit}\right)\left(\frac{units}{bid}\right)$$

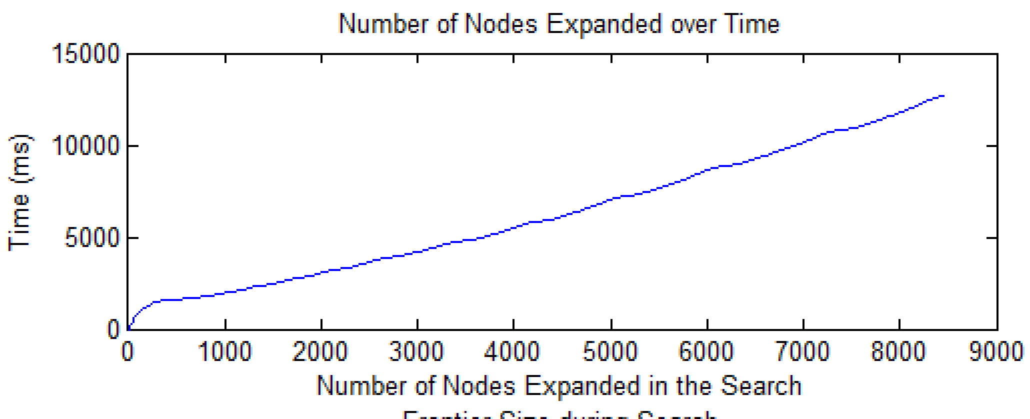
Keeping these parameter choices in mind, we plotted some interesting attributes about our searches for increasing problem sizes. The first plot is for a problem with 10 students and 10 courses, the second contains 50 students and 10 courses, and the third contains 100 students and 10 courses. A key observation is that the amount of time per expanded node in the search becomes fairly constant for nodes later in the search. But for larger problem instances, the time spent on the first few expanded nodes is disproportionately large. This is explained by the fact that our upper and lower bounds take longer to compute on the first few nodes, since there are more remaining bids at the beginning of the search.

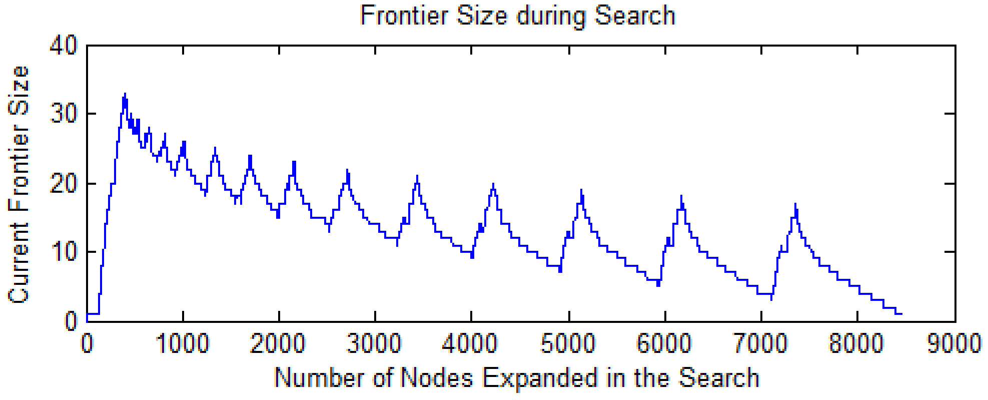
Additionally, there is a dramatic change in the behavior of the frontier size over time from smaller sized problems to larger problems. While the frontier size trajectory for smaller instances fits with previous results, it is not yet clear the reason for the regular fluctuation in frontier size for larger problems.


Number of Nodes Expanded over Time
Time (ms)
800
600
400
200
0
0
200
400
600
800
1000
1200
Number of Nodes Expanded in the Search
Frontier Size during Search
Current Frontier Size
400
300
200
100
0
0
200
400
600
800
1000
1200
Number of Nodes Expanded in the Search


Number of Nodes Expanded over Time
Time (ms)
600
400
200
0
0
200
400
600
800
1000
1200
1400
1600
1800
Number of Nodes Expanded in the Search


Frontier Size during Search
Current Frontier Size
60
40
20
0
0
200
400
600
800
1000
1200
1400
1600
1800
Number of Nodes Expanded in the Search


Number of Nodes Expanded over Time
Time (ms)
15000
10000
5000
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Number of Nodes Expanded in the Search


Frontier Size during Search
Current Frontier Size
40
30
20
10
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Number of Nodes Expanded in the Search

6 Discussion

Future work can be done in figuring out the reason for the transition in performance of the search's frontier size for different-sized input problems. Furthermore, domain-independent upper bounds can be tried as well for the search, for example, relaxing the requirements that the solutions are integers to just forcing them to real numbers. Moreover, the tradeoffs associated with different choices of lower and upper bounds can be explored by plotting the time complexity and space (frontier size) for combinations of these. Finally, instead of the current method of computing the lower and upper bound at each node, a policy could be determined experimentally for which nodes to try to prune via the bounds and which ones should be skipped to reduce the overal runtime.

7 Sources

1. Finding Approximate Competitive Equilibria: Efcient and Fair Course Allocation, Abraham Othman, Eric Budish, and Tuomas Sandholm, *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, van der Hoek, Kaminka, Lesprance, Luck and Sen (eds.), May, 1014, 2010, Toronto, Canada, pp. XXX-XXX.

2. E. Budish. The Combinatorial Assignment Problem: Approximate Competitive Equilibrium From Equal incomes. Technical Report, University of Chicago Booth School of Business, 2011.