CPSC 433 - Course Scheduling Problem

Konrad Aust Mark Barley Kendra Wannamaker
Tomas Bystersky Jordan Wallace Aaron Mouratidis

October 19, 2015

Contents

1 Introduction

In this paper, we will present two approaches to solving the scheduling problem for the Computer Science Department at the University of Calgary. The first is a set-based search model, with a genetic approach. The second is an And-Tree-Based model that subdivides the schedule and attempts to combine subproblems to find an overall solution.

2 Set Based Search

2.1 Overview

This is a genetic algorithms approach that begins with a set of valid, random solutions, and then for a number of generations uses mutate and crossover operations on some subset of the population to create a gradually improving solution.

The desirability of a given solution is given by the *Eval* function. More desirable solutions are more likely to be crossbred. Whenever the population expands beyond a certain threshold, the solutions with the highest *Eval* values are retired.

Mutations and crossovers are done in such a way that all solutions are kept valid. There will never be a solution in the population that does not satisfy all hard constraints.

The algorithm terminates after a number of generations determined at runtime. After simulating the generations, the final solution is the member of the population with the lowest *Eval* value.

2.2 Search Model

$$\mathbb{A} = (S, T)$$

$$f \in F \iff f = (a_1, ..., a_n)$$

$$a_i \in Slots$$

Where a_i in f means that some course or lab with index i is assigned slot $a_i \in Slots$. $(1 \leq i \leq n)$.

Our courses and labs are indexed based on some arbitrary ordering of the elements in the set $Courses \cup Labs$. The ordering itself does not matter, so long as we have a unique index for every course and lab. There should be n such indices, where $n = |Courses \cup Labs|$

$$S \subseteq 2^F$$

$$T = \{(s, s') | \exists (A \rightarrow B) \in Ext,\, s' = (s - A) \cup B\}$$

$$Ext = \{A \rightarrow B | A, B \subseteq F,\, (Mutate(A, B) \vee Crossover(A, B) \vee Plague(A, B))\}$$

(Function definitions for Mutate, Crossover, and Plague are given at the end of this section.)

2.3 Search Process

$$\mathbb{P} = (\mathbb{A}, Env, \mathbb{K})$$

$$\mathbb{K}(s,e) = (s - A) \cup B$$

Where

$$A \to B \in Ext$$

And:

- If $|s| \geq threshold$
  - Where $threshold$ is the maximum population size we allow.
  - $A =$ The least valuable 40% of facts.
  - $Plague(A,B)$ holds.
- Else, randomly select between one of the two following conditions with equal probability:
- Condition 1: Mutate
  - $A = \{k_1\}$ Where k_1 is a randomly selected member of s
  - $Mutate(A,B)$ holds.
- Condition 2: Crossover
  - $A = \{k_1, k_2\}$ Where
  - $k_1 =$ The member of s with the lowest $Eval$ value.
  - k_2 Is a member of s randomly selected according to the following distribution:
  - $$p_j = \frac{1}{Eval(s_j) \cdot fitSum} \forall s_j \in s - \{k_1\}$$
  - $$\text{where } fitSum = \sum_{h=1}^{N} \frac{1}{Eval(s_h)}, \ N = |s - \{k_1\}|$$
  - Where p_j is the probability that s_j is selected. This is fitness proportionate selection, meaning solutions with a better $Eval$ value are more likely to be selected. The sum of all such probabilities is 1.
  - $Crossover(A,B)$ holds.

We only discard the worst members of the population when necessary, and we ensure that the fittest member is always able to pass on his traits through crossovers, while selecting the other crossover parent randomly with a bias towards fitter individuals. To ensure diversity, we add random mutations.

This search control should build on the most successful solutions while introducing enough randomness to maintain diversity. That's the plan, anyways.

2.4 Search Instance

$$Ins = (s_0, G)$$

$$s_0 = \{s_{0_1}, s_{0_2}, ..., s_{0_q}\}$$

Where $s_{0_i} = RandomSol()$ for $1 \leq i \leq q$, where q is the number of elements in our starting population. This number can be decided at runtime.

$$G(s) = \begin{cases} true & \text{If this is the final generation} \\ false & \text{Otherwise} \end{cases}$$

In other words, we decide ahead of time for how many generations our search will run. Once we have reached our final generation, we quit the search. Our solution will be the member of s with the lowest $Eval$ value.o

2.5 Relation Definitions

2.5.1 RandomSol

RandomSol is an or-tree-based search that finds a random valid class assignment. It is used to build our initial population of facts at the beginning. It selects transitions randomly, prioritizing the ones at the deepest levels of the tree to find solutions more quickly.

If we fail to find a solution, we do not need to run the set based search, as this means that there is no valid solution.

Or-Tree-Based Search Model

$$\mathbb{A} = (S, T)$$

S and $T$are no more specific than the general definition given for Or-Tree-Based search.

$$pr \in Prob \iff pr = (a_1, ..., a_n)$$

Where $a_i \in (Slots \cup \{\$\})$. a_i represents the slot assignment for the course or lab with index i. An assignment of \$ indicates that no assignment has been made thus far.

As in the Set Based Search, our courses and labs are indexed based on some arbitrary ordering of the elements in the set $Courses \cup Labs$. The ordering itself does not matter, so long as we have a unique index for every course and lab. There should be n such indices, where $n = |Courses \cup Labs|$

$$Altern(pr, pr_1, ..., pr_M)$$

Holds where:

$$pr = (a_1, ..., a_i, ..., a_n)$$

a_i is the first element in this vector with a value of \$.

For each $s_j \in Slots$, we create a pr_j such that:

$$pr_j = (a_1, ..., a'_i, ..., a_n)$$

$$a'_i = s_j$$

Or-Tree-Based Search Process

$$\mathbb{P} = (\mathbb{A}, Env, \mathbb{K})$$

$$\mathbb{K}(s, e) = s'$$

Where $\forall x \in S,\ f_{leaf}(x) \geq f_{leaf}(s')$

Where $s'$is uniformly randomly selected among all nodes with the highest depth.

Traversing the nodes in a random order should in theory lead us to a random solution.

Or-Tree-Based Search Instance

$$Ins = (s_0, G)$$

$$s_0 = partassign$$

$G(s)$ is no more specific than the general definition. Our criteria for when a solution is solved is the following:

$$sol = \begin{cases} yes & \forall x \in pr,\ x_i \neq \$,\ Constr(pr) = true \\ no & \forall x \in pr,\ x_i \neq \$,\ Constr(pr) = false \\ ? & otherwise \end{cases}$$

2.5.2 Mutate

A mutation takes a single individual, and messes with a number of their course assignments, randomly switching them around. The purpose of this is to introduce randomness to preserve diversity.

$$Mutate(A, B)$$

$$A \subseteq F,\ B \subseteq F$$

$$A = \{f\},\ B = \{f, f'\}$$

1. Start with $f' = f = (a_1, ..., a_N)$
2. Repeat the following a number of times:
   (a) Pick a uniformly random a_i from f'
   (b) Set a_i to some randomly selected $s \in Slots - \{a_i\}$, where $Constr(f') = true$

The number of switches in a mutation is constant and decided ahead of time.

2.5.3 Crossover

$$Crossover(A, B)$$

$$A = \{M, F | M, F \in F, M \neq F\} \ B = \{M, F, C\}$$

Where M, F, and C are the Mother, the Father, and the Child of the crossover operation, respectively. C is created by merging traits of M and F using an or-tree-based search.

Or-Tree-Based Search Model

This search model is identical to that of RandomSol()

Or-Tree-Based Search Process

$$\mathbb{P} = (\mathbb{A}, Env, \mathbb{K})$$

$$\mathbb{K}(s, e) = s'$$

Where s'satisfies

- $\forall x \in s,\ f_{leaf}(x) \geq f_{leaf}(s')$
- $s'$is otherwise selected uniformly randomly.

We define $f_{leaf}(r)$ as follows:

$$f_{leaf}(r) = \begin{cases} 1 & r \in M \ \wedge \ r \in F \\ 2 & r \in M \ \vee \ r \in F \\ 3 & otherwise \end{cases}$$

Or-Tree-Based Search Instance

This search instance is identical to that of RandomSol()

2.5.4 Plague

Plague is very simple. It takes a set of facts, and removes them. We use this function to trim our population, preventing it from getting too large.

$$Plague(A, B)$$

$$A \subseteq F$$

$$B = \{\}$$

2.6 Example

Let's consider a simple scenario where:

- $Slots = \{s_1, s_2, s_3\}$
- $Courses = \{c_1, c_2\}$
- $Labs = \{l_{11}, l_{21}\}$
- $not - compatible = \{(c_1, l_{11}), (c_2, l_{21})\}$
- $unwanted = \{(c_1, s_3)\}$
- $partassign = (\$, \$, \$, \$)$

Let us index our courses and labs in the following order: $(c_1, c_2, l_{11}, l_{21})$.

In the interest of keeping this section brief, we will not specify values such as$pen_coursemin$, or $Preference$, and will simply give the $Eval$ values of each fact. The operations that happens are as follows:

1. Generate a Random Population of Solutions

We use RandomSol() to generate the following facts:

- $f_1 = (s_1, s_2, s_{3,}s_3),\ Eval(f_1) = 7$
- $f_2 = (s_2, s_3, s_{1,}s_1),\ Eval(f_2) = 10$

2. Mutate a Randomly Selected Fact

We choose f_2 randomly. After applying the mutation operator, the result is the addition of f_3:

- $f_3 = (s_2, s_2, s_1, s_1),\ Eval(f_3) = 5$

3. Crossover of Two facts

The next operation is a crossover. We pick $f_3$because it has the lowest $Eval$ value, and we randomly select f_1 to be the other parent. This results in the addition of the newly born f_4.

- $f_4 = (s_2, s_2, s_3, s_3),\ Eval(f_4) = 3$

4. Trimming of the Population

We now have too many facts in our population! We kill off the worst one. In this case, that is f_2, since it has the highest $Eval$ value. Our population is now:

$s = \{f_1, f_3, f_4\}$

5. Finished with Generations

We only ran three generations in this example, but you should probably run more. Now that we are finished, we return the solution with the lowest $Eval$ value in our population. That value is f_4.

- Solution $= f_4 = (s_2, s_2, s_3, s_3)$

And we are done!

3 And-Tree-Based Search

3.1 Overview

This is an And-Tree-Based approach that, at each level of the tree, inserts one more course or lab into every possible slot and create a new leaf for each one. Each of those newly created leaf nodes is evaluated to determine which we should choose next. We first prioritize branches where we can change the sol entry. Our second priority depends if we have previously found a valid solution to compare soft constraint evaluation against. If we have not found a valid solution we will do a depth first search, to find the first valid solution as soon as possible. Once we find our first solution we will switch to a breadth first search beacuse our goals states can now include "violating our constraints", "complete assignment of all course and labs", and "the leaf evaluates to less than our current best solution".

In order to trim the size of the tree we use the $Eval$ function which evaluates the current leaf based on an analysis of the violations of the soft constraints. Once a leaf is reached where the $Eval$ function evaluates to a worse value than the current best solution, the leaf's sol entry is changed to a Yes state. If a new complete solution is found, and has an $Eval$ value better than the current best solution, the we update our best $Eval$ with our new solution.

When all branches of the tree end in a Yes state leaf, the best solution has been found and is returned.

3.2 Search Model

$$A = (S, T)$$

$$pr \in Prob \iff pr = (X_1, ..., X_n)$$

Where $X_i \in (Slots \cup \{\$\})$. X_i represents the slot assignment for the course or lab with index i. An assignment of \$ indicates that no assignment has been made thus far.

As in the Set Based Search, our courses and labs are indexed based on some arbitrary ordering of the elements in the set $Courses \cup Labs$. The ordering itself does not matter, so long as we have a unique index for every course and lab. There should be n such indices, where $n = |Courses \cup Labs|$

$$Div(pr, pr_1, ..., pr_M)$$

Holds if:

$$pr = (X_1, ...X_p....X_n)$$

Where $X_p$is the first instance of \$ in pr and for each $s_j \in Slots$

$$X'_p = s_j$$

$$pr_j = (X_1, ..., X'_p, ..., X_n)$$

$$T = \{(k, k')|k, k' \in S\ Erw(k, k')\ or\ Erw*(k', k)\}$$

$$S \subseteq Atree$$

Where $Atree$ is defined recursively by:

$$(pr, sol){\in}Atree$$

$$pr{\in}Prob,\ sol \in \{yes, ?\}$$

$$(pr, sol, b_1, \dots, b_M){\in}Atree$$

$$b_1, ..., b_M \in Atree$$

$$\text{"}pr\text{ is solved"} \iff pr = (X_1, ..., X_n) \text{ with}$$

$$X_k \neq \$ \text{ for } 1 \leq k \leq n \text{ or}$$

$$Eval(pr) \leq Eval(w)$$

Where $w = min[Eval(pr_i)]$ where $pr = (X_1$,..., $X_n)$ with all $X_j \neq$ \$, $1 \leq j \leq n$.

In other words, w is the current best solution.

Erw is no more specific than the general definition for And-tree search.

3.3 Search Control

$$\mathbb{P} = (\mathbb{A}, Env, \mathbb{K})$$

$$\mathbb{K}(s, e) = s'$$

Where s'satisfies

- $\forall x \in s,\ f_{leaf}(x) \geq f_{leaf}(s')$
- If there are multiple transitions that satisfy the above, s' is the leftmost node.

We define $f_{leaf}(r)$ as follows:

$$f_{leaf}(r) = \begin{cases} 0 & sol \text{ value can be changed} \\ depth_{max} - depth(r) & w = null \\ depth(r) & w \neq null \end{cases}$$

f_{trans}expands pr based on the Div relations to produce pr_1 to pr_n as children of pr.

$depth_{max}$ is the maximum depth of the tree.

As above, w is our current best solution. It is the leaf node with $sol = yes$ which satisfies $Constr$ and has the lowest $Eval$ value out of any we have traversed so far.

In other words, we expand nodes deeper down the tree until we have found a valid solution. We then switch to a breadth-first method of traversing the tree. This aids us in pruning the tree, by cutting off any branches with a worse $Eval$ value than our current best solution.

3.4 Search Instance

$$Ins = (s_0, G)$$

If the given problem to solve is pr, then we have
$s_0 = (pr, ?)$, where $pr = partassign$.
And $G(s) = yes$ if and only if:

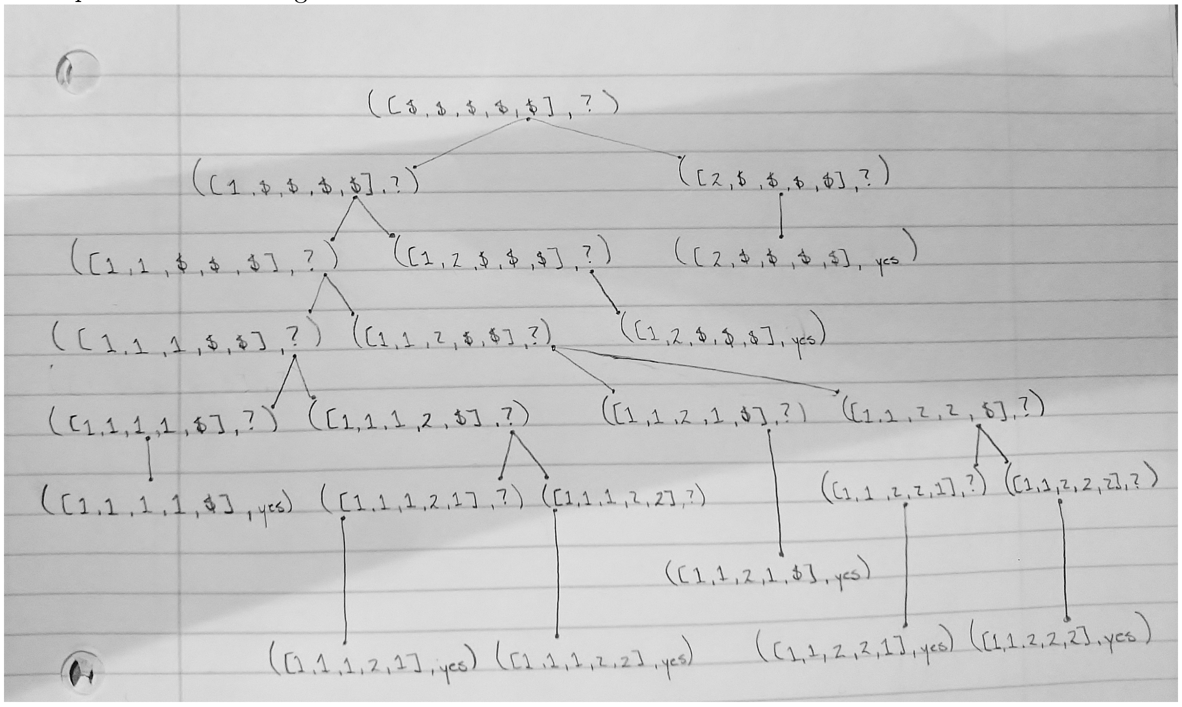
- $s = (pr', yes)$ or
- $s = (pr', ?, b_1, ..., b_M), G(b_1) = ... = G(b_M) = yes$ and the solutions to $b_1, ..., b_M$ are compatible with each other or there is no transition that has not been tried out already.

3.5 Example

Let's consider a simple scenario where:

- $Slots = \{s_1, s_2\}$
- $Courses = \{c_1, c_2, c_3\}$
- $Labs = \{l_{11}, l_{21}\}$
- $unwanted = \{(c_2, s_2)\}$
- $partassign = (\$, \$, \$, \$, \$)$

Let us index our courses and labs in the following order:
$(c_1, c_2, c_3, l_{11}, l_{21})$
We end up with the following search tree:


([$,$,$,$,$],?)
([1,$,$,$,$],?)
([2,$,$,$,$],?)
([1,1,$,$,$],?)
([1,2,$,$,$],?)
([2,$,$,$,$],yes)
([1,1,1,$,$],?)
([1,1,2,$,$],?)
([1,2,$,$,$],yes)
([1,1,1,1,$],?)
([1,1,1,2,$],?)
([1,1,2,1,$],?)
([1,1,2,2,$],?)
([1,1,1,1,$],yes)
([1,1,1,2,1],?)
([1,1,1,2,2],?)
([1,1,2,2,1],?)
([1,1,2,2,2],?)
([1,1,2,1,$],yes)
([1,1,1,2,1],yes)
([1,1,1,2,2],yes)
([1,1,2,2,1],yes)
([1,1,2,2,2],yes)