Course Scheduling

Aaron Hornby Bryan Lam Cynthia Nguyen
Esther Chung Herman Kwan

October 19, 2018

1 Introduction

1.1 Preface

This paper will describe two search models and processes that could be used as a solution for the Computer Science Department scheduling problem at the University of Calgary. The first of these solutions follows a genetic search algorithm modelled as a set-based search. The second of these approaches uses the And-Tree search paradigm. We will explore each paradigm and present how it can be defined and used to solve our problem.

1.2 Set Definitions

$$Courses = \{c_1, \ldots, c_m\}$$

This is the set of course sections that the department teaches in a particular semester.

$$Labs = \{l_{11}, \ldots, l_{k1_1}, \ldots, l_{m1}, \ldots, l_{mk_m}\}$$

Similarly, this is the set of labs taught. Where $l_{i1}, \ldots, l_{ik_i}$ are the labs associated with course section c_i. Consequently, if c_i has no associated labs, then $k_i = 0$.

$$Slots = \{sl_1, \ldots, sl_n\}$$

This is the set of offered time slots which a course section or lab can be assigned to.

$$Classes = Courses \cup Labs = \{cl_1, \ldots, cl_s\}$$

Furthemore, it is essential, to certain entities defined later, that we assign a unique index i ($1 \leq i \leq s$) to every element in $Classes$. Thus, later on, any references to $cl_i \in Classes$ will represent the course or lab assigned to index i.

2 Solution 1: Genetic Algorithm

2.1 Overview

The main characteristic of a genetic algorithm is that it is based on natural selection, that is, survival of the fittest. This is where the fittest individuals in a given population survive to pass their genes onto the next generations, while the weakest individual genes are killed off. Which is in essence, saying that "good" genes survive, while "bad" genes are removed from the gene pool. A genetic algorithm simulates the survival of the fittest individuals over certain generations by representing individuals as possible solutions to a given problem. Each "solution" has a certain fitness value and those who are more fit will pass on their genes via mating, thus creating another individual possessing those "good" genes. After many generations, we would have ideally come across an individual in the final generation that is close to optimal. We begin by explaining our solution by first defining the search model, search process and search instance. Any predicates and functions not defined in these sections are defined in section 2.5.

2.2 Search Model

$$\mathbb{A}_{set} = (S_{set}, T_{set}) :$$
$$S_{set} \subseteq 2^F$$

$$F = \{\langle t_1, t_2, \ldots, t_p \rangle \mid t_1, t_2, \ldots, t_p \in Slots\}$$

To define S_{set}, we must first define a fact, f. A fact is a vector containing p elements where p is the amount of elements in $Classes$. The index of each of these elements correlates with the element of $Classes$ with the same index. That is to say, cl_1 is assigned the time slot t_1. S_{set} is then a subset of the power set of F, where F is a set of facts.

$$T_{set} = \{(s, s^{'}) \mid \exists (A \rightarrow B) \in Ext, A \subseteq s, s^{'} = (s - A) \cup B\}$$
$$Ext = \{A \rightarrow B \mid A, B \subseteq F, Crossover(A, B), Trimming(A, B)\}$$

The next step to defining our search model is to define T_{set}; the set of transitions. Where S_{set} depends on F, T_{set} depends on the extension rules. The most common extension rules used in genetic algorithms are Crossovers, Mutations and Trimming. In our model, we include two of those, Crossovers and Trimming where our Crossover rule also contains elements of mutations to increase diversity. These two functions are further defined in more detail in Section 2.5.

2.3 Search Process

$$P_{set} = (\mathbb{A}_{set}, Env, K_{set}) :$$

This is the general definition of the set-based search process. We have already defined $\mathbb{A}_{set}$ in the previous section and we do not care for Env in this case (due to our assumption that it remains constant). And so, to define the search process, we define the search control, K_{set} which depends on two functions, f_{wert} and f_{select}.

$$K_{set}(s, e) = (s - A) \cup B$$

$$Where$$

$$A \to B \in Ext$$

$$A \subseteq s$$

$$\forall (A' \to B') \in Ext,\ A' \subseteq s,\ f_{wert}(A, B, e) \leq f_{wert}(A', B', e)$$

$$f_{select}(\{A' \to B' \mid f_{wert}(A', B', e) \leq f_{wert}(A'', B'', e), \forall (A'' \to B'') \in Ext,\ A'' \subseteq s\},\ e) = A \to B$$

The general definition of K_{set} for a set-based search remains the same for our model. What is more specific is our definition of f_{wert} and f_{select}.

$$f_{wert}(A, B, e) = (|(s - A) \cup B| > |s|,\ ?\,1 : 0)\ +\ (|(s - A) \cup B| < q,\ ?\,10 : 0)$$

Where q is equal to our population limit. Our population limit will be set to a certain number at the beginning of our algorithm. Our f_{wert} can only assign three values, 0, 1 and 10. It is a 1 if the state is growing in size, meaning that our extension rule $Crossover(A, B)$ is being applied as this is the only rule that increases the population. It is a 0 when the state is shrinking in size, meaning that the extension rule $Trimming(A, B)$ is being applied as it's the only rule that shrinks the population. It is 10, when the population shrinks below q, our population limit. In short, f_{wert} gives a 0 for Trimming and 1 for Crossovers, so that Trimming is always prioritized over Crossovers. This way, we can keep our population size constant. It also gives 10 to prevent it from trimming our state below its population limit.

$$f_{select} = (s - A) \cup B\ where,\ \forall b \in (B - s),\ parents(b) \subset fittest(s)$$

$$Where :$$

$$fittest(s) \subset s$$

$$fittest(s) = \{a_1, \ldots, a_{\frac{|s|}{2}} \mid \forall a_j \in s, \forall d \in R, Eval(d) \geq Eval(a_j)\}\, where\, R = (s - fittest(s))\}$$

$$parent(b) = \{p_1, p_2 \mid p_1, p_2 \in s\}$$

f_{select} is the tie breaker for our extension rules. When there are multiple extension rules with the same f_{wert} value then f_{select} applies. We have defined our extension rules so that it'll only be a tie for Crossovers, since there will only be one way to Trim the population. In contrast, there are multiple ways Crossovers can occur in the population and our f_{select} is used to determine how crossovers should occur. In essence, f_{select} is used to choose the fittest individuals in the population to mate.

Our f_{select} chooses the extension rule that leads to a state where all offspring are from the fittest individuals of state s. It also selects so that only 50% of the population actually Crossover. The way fitness is measured in our model is through $Eval$, which gives a number to a fact, the lower the number the better the solution.

2.4 Search Instance

$$Ins_{set} = (s_0, G_{set}) :$$
$$s_0 = \{f_1, f_2, \ldots, f_q \mid \forall f_j,\ f_j = randomAssign(),\ f_j \in F\}$$

Our start state, s_0 is a set of facts of size q. We have defined q in the previous subsection as the maximum population size that will be set during implementation of this algorithm. Each of these facts will be generated by the $randomAssign()$ function. This function generates a valid fact randomly through an OR-Tree-Based search. By doing this, we guarantee our initial population starts off with valid facts. $randomAssign()$ is further detailed in section 2.5.

$G_{set}(s) = True \iff$ This is your final generation

We determine how many generations our search will run for during implementation of this model. When we are in our final generation, our solution will be in the fact with the lowest $Eval$ value.

2.5 Predicate Definitions

2.5.1 Initial Population

An OR-Tree-Based search model is being defined in order to obtain random valid solutions for the initial population. It is also being used for the extension rule $Crossover$. The reason for this is because applying the $Crossover$ operation alone cannot guarantee a valid solution, by using an OR-Tree-Based Search it can guarantee valid solutions.

OR-Tree-Based Search Model

$$\mathbb{A}_v = (S_v, T_v)$$

$$Prob = \{\langle O_1, \ldots, O_p \rangle \mid O_1, \ldots, O_p \in Slots \cup \{\$\}\}$$

$$Altern = \langle pr, pr_1, \ldots, pr_z \mid \forall\ pr_v = \langle O_1, \ldots, O_i, \ldots, O_p \rangle, 1 \leq j < i, O_j \in Slots, O_i = \$ \rangle$$

Altern is defined when $pr = \langle O_1, \ldots, O_j, \ldots, O_p \rangle$ such that $O_j = \$$ and either $j = 1$ or $j \neq 1$ and $\forall O_i$, such that $1 \leq i < j$, $O_i \neq \$$. Then, $\forall sl_a \in Slots$, $pr_a = \langle O_1, \ldots, sl_a, \ldots, O_p \rangle$, (for $1 \leq a \leq n$).

The set of states is defined as:

$$S_v \subseteq Otree$$

Where Otree is recursively defined by:

$$(pr, sol) \in Otree,\ where\ pr \in Prob,\ sol \in \{yes, ?, no\}$$

$$(pr, sol, b_1, \ldots, b_n) \in Otree,\ where\ pr \in Prob,\ sol \in \{yes, ?, no\},\ b_1, \ldots, b_n \in Otree$$

pr is defined as a vector of size p such that each element indicates an assignment of a slot or not. \$ is being used to represent a slot that is unassigned.

OR-Tree-Based Search Process

$$P_v = (\mathbb{A}_v, Env, K_v)$$

$$K_v : S_v \times Env \rightarrow S_v\ where\ K_v(s, e) = s' \Rightarrow (s, s') \in T_v$$

K_v is not more specific than the general definition. What we do define to make it more specific is the definition of f_{leaf}.

$$f_{leaf}\ chooses\ one\ of\ the\ deepest\ leaves\ at\ random$$

For initial population, f_{leaf} chooses at random the deepest node in which to apply *Altern* to.

$$sol = \begin{cases} yes & pr = \langle O_1, \ldots, O_p \rangle\ where\ \forall\ O_j,\ O_j \notin \{\$\}\ \wedge\ Constr(pr) = true \\ no & pr = \langle O_1, \ldots, O_p \rangle\ where\ \forall\ O_j,\ O_j \notin \{\$\}\ \wedge\ Constr(pr) = false \\ ? & When\ neither\ yes\ or\ no \end{cases}$$

OR-Tree-Based Search Instance

$$Ins_v = (s_0, G_v)$$

Let $pr = \langle O_1 \ldots, O_p \rangle$, where $O_1 = \ldots, = O_p = \$$.

The start state is defined as:

$$s_0 = (pr, ?)$$

The goal condition is defined as:

$G_v(s) = yes \iff$
$s = (pr', yes)$, or
$s = (pr', ?, b_1, \ldots, b_n)$, $G_v(b_i) = yes$ for an i, or
All leaves of s have either the sol-entry no or cannot be processed using Altern

2.5.2 Crossover

$$Crossover(A, B) = B = A \cup C \mid krossover(m, d) \in C,\ m, d \in A$$

Where $krossover(m, d)$ is a function that generates a fact using an or-tree-based-search similar to the one used for the $randomAssign()$ function, but differs in its search control.

OR-Tree-Based Search Model

The search model is identical to $randomAssign()$.

OR-Tree-Based Search Process

$$f_{leaf} = \begin{cases} choose\ \ random\ leaf & Constr^*(partm) = false \wedge Constr^*(partd) = false \\ choose\ leaf\ \ from\ m & Constr^*(partm) = true \wedge Constr^*(partd) = false \\ choose\ leaf\ \ from\ d & Constr^*(partm) = false \wedge Constr^*(partd) = true \\ choose\ random\ leaf\ between\ parents & otherwise \end{cases}$$

The only difference between the $randomAssign()$ and $krossover(m, d)$ is that it differs in how f_{leaf} is defined. In this function, f_{leaf} is defined so that, for some index, it chooses randomly between the parents' time slots for assigned classes for that index. This represents the crossover operation, so that the fact created contains the parents' elements. Although, it could be the case that the crossover causes an invalid fact. We define $partm$ and $partd$ as the pr of our current state, where the left most \$ is replaced with the time slot from m and d respectively. We use $Constr^*$ (defined in section 3.3.1) to check for invalid facts during the construction of the fact. If it is the case that $partm$ and $partd$ results in an invalid fact then we simply choose the time slot from the other parent. If both $partm$ and $partd$ result in invalid facts then we choose a random leaf so that \$ is replaced with any time from $Slot$. In this case, this would represent a mutation in our algorithm and would maintain diversity in our population.

OR-Tree-Based Search Instance

The search instance is identical to $randomAssign()$.

2.5.3 Trimming

$Trimming(A, B) = B,\ where\ B = A - w\ where, \forall g \in A, w \in A, Eval(w) \geq Eval(g)$

Our trimming rule is used to trim our population so that our population stays at a constant size. It removes a single fact from A, and our f_{select} prioritizes that rule until our population size returns to normal.

2.6 Example

$Courses = \{c_1, c_2\}$
$Labs = \{l_{11}, l_{21}\}$
$Slots = \{sl_1, sl_2, sl_3\}$
$Classes = \{c_1, c_2, l_{11}, l_{21}\}$
$populationLimit = 4$
$coursemin(sl_1) = 1$
$labmin(sl_2) = 1$

We index our courses and labs so that they are in the following order: $(c_1, c_2, l_{11}, l_{21})$

1. Generate our initial population
$randomAssign()$ is used to randomly generate valid facts up to our $populationLimit$. We know they are valid because this function uses an or-tree-based search to generate them. In this example, we apply $randomAssign()$ to generate 4 facts for our initial population.

$$s_0 = \{f_1, f_2, f_3, f_4\}$$

$$f_1 = \langle sl_2, sl_3, sl_3, sl_2 \rangle \qquad Eval(f_1) = 1$$
$$f_2 = \langle sl_1, sl_2, sl_3, sl_1 \rangle \qquad Eval(f_2) = 1$$
$$f_3 = \langle sl_2, sl_3, sl_3, sl_1 \rangle \qquad Eval(f_3) = 2$$
$$f_4 = \langle sl_3, sl_2, sl_1, sl_1 \rangle \qquad Eval(f_4) = 2$$

The Eval value of each fact is represented below to help simplify and visualize this example better. For simplification of this example, we say that each soft constraint broken increases the $Eval$ value by 1. In this case, the soft constraint is represented by the the $coursemin(sl)$ and $labmin(sl)$, when the minimal number courses or labs are not schedule into these slots, it will increase the eval value by 1. In this case, f_1 eval value is 1 because there are 0 courses scheduled for the time slot s_1.

2. Crossover

$$s_1 = \{f_1, f_2, f_3, f_4, f_5\}$$
$$f_5 = \langle sl_1, sl_3, sl_3, sl_2 \rangle \qquad Eval(f_5) = 0$$

Our model is defined so that only half of the population that is the fittest gets to mate randomly among that group. In this case, only f_1 and f_2 are chosen among the group. The result of this operation is the addition of f_5 to the population.

3. Trim the population

$$s_1 = \{f_1, f_2, f_3, f_5\}$$

Our population now has more than the population limit and so the trim operator is then used. f_3 and f_4 have the worst $Eval$ value and so f_4 is randomly chosen between the two.

4. Final Generations

$$Solution = f_5$$

In this example, we only ran one generation, but more should be run to include more coverage of good and near-optimal solutions. Since this is the final generation, we then choose the best solution in this state, which is f_5 as it has the lowest $Eval$ value in our population.

3 Solution 2: AND-Tree-Based Search

3.1 Overview

The main idea behind this paradigm is to break down a problem into subproblems, whose solutions can be combined (if collectively compatible), to produce a solution for the original problem. In our application of this paradigm, the root problem is to find a slot assignment to each course or lab, such that all hard constraints are satisfied and the fulfillment of soft constraints is optimized. The search process will examine the search space and evaluate valid solutions to find an optimal one.

3.2 Search Model

$$\mathbb{A}_\wedge = (S_\wedge, T_\wedge)$$

A problem is described as a vector of size p. The i^{th} element of such a vector corresponds to the assignment of a slot to the course or lab with index i. If no slot is assigned to that course or lab yet, a \$ will be put in place.

$$Prob = \{\langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$p}\rangle \mid sl_{\$1}, sl_{\$2}, \ldots, sl_{\$p} \in Slots \cup \{\$\}\}$$

Where $sl_{\$i}$ (for $1 \leq i \leq p$) corresponds to either $assign(cl_i)$ (where $cl_i \in Classes$ and has the assigned index i) or \$ indicating no assignment yet.

A problem, $pr = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$p} \rangle \in Prob$, is considered solved (and its sol-entry can be updated to 'yes') when one of two conditions is satisfied:

1. $\forall sl_{\$i}$ in $pr,\ sl_{\$i} \neq \$$ or
2. $\exists sl_{\$i}$ in $pr,\ sl_{\$i} = \$$ and either $Constr^*(pr) = false$ or $Eval^*(pr) \geq bestEval$

   where $Constr^*$, $Eval^*$ and $bestEval$ are defined in section 3.3.1.

Condition 1 is trivial, since if the vector contains no \$, then it cannot be divided anymore in the future and thus the problem is marked as solved. Condition 2 allows us to reduce the number of search steps. It indicates that if we arrive at a sub-problem (with only a partial assignment) that either violates a hard constraint or is less optimal than our best valid solution found previously (if any), then we can also mark this problem as solved. This prevents dividing this problem in the future and undergoing unnecessary steps by checking over sub-problems that will still either violate a hard constraint or have a higher penalty value.

The division relation,

$$Div(pr, pr_1, \ldots, pr_n),\ where\ Div \subseteq Prob^+$$

is defined when $pr = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$j}, \ldots, sl_{\$p} \rangle$ such that $sl_{\$j} = \$$ and either $j = 1$ or $j \neq 1$ and $\forall sl_{\$i}$, such that $1 \leq i < j$, $sl_{\$i} \neq \$$. Then, $\forall sl_a \in Slots,\ pr_a = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_a, \ldots, sl_{\$p} \rangle$, (for $1 \leq a \leq n$).

Less formally, a problem pr can be divided into sub-problems where each sub-problem is a vector constructed from pr where the leftmost \$ in pr is replaced with a different element in $Slots$.

The set of states is defined as:

$$S_\wedge \subseteq Atree$$

Where Atree is recursively defined by:

$$(pr, sol) \in Atree,\ where\ pr \in Prob,\ sol \in \{yes, ?\}$$

$$(pr, sol, b_1, \ldots, b_n) \in Atree,\ where\ pr \in Prob,\ sol \in \{yes, ?\},\ b_1, \ldots, b_n \in Atree$$

The set of transitions is defined as:

$$T_\wedge = \{(s_1, s_2) \mid s_1, s_2, \in S_\wedge \; and \; Erw_\wedge(s_1, s_2) \; or \; Erw^*_\wedge(s_2, s_1)\}$$

Where $Erw_\wedge$ and $Erw^*_\wedge$ are relations on Atree generally defined by:

$$Erw_\wedge((pr, ?), (pr, yes)), \; if \; pr \; is \; solved$$

$$Erw_\wedge((pr, ?), (pr, ?, (pr_1, ?), \ldots, (pr_n, ?))), \; if \; Div(pr, pr_1, \ldots, pr_n) \; holds$$

$$Erw_\wedge((pr, ?, b_1, \ldots, b_n), (pr, ?, b_1^{'}, \ldots, b_n^{'})), \; if \; for \; an \; i : Erw_\wedge(b_i, b_i^{'}) \; and \; b_j = b_j^{'} \; for \; i \neq j$$

$$Erw_\wedge \subseteq Erw^*_\wedge$$

$$Erw^*_\wedge((pr, ?, b_1, \ldots, b_n), (pr, ?, b_1^{'}, \ldots, b_n^{'})), \; if \; for \; all \; i \; either \; Erw^*_\wedge(b_i, b_i^{'}) \; or \; b_i = b_i^{'} \; holds$$

3.3 Search Process

$$P_\wedge = (\mathbb{A}_\wedge, Env, K_\wedge)$$

We can disregard Env, since we assume that no external factors will change during the search.

$$K_\wedge : S \times Env \rightarrow S \; where \; K(s, e) = s' \Rightarrow (s, s') \in T$$

Furthermore, the transition from s to s', can be decided with the help of two functions, f_{leaf} and f_{trans}. First, f_{leaf} is used to select the next leaf in the tree to work on. Let us define $nextLeaf$ as the leaf in s with the smallest f_{leaf} value. Formally, for every leaf problem pr_l in s (with sol-entry = ?), $f_{leaf}(nextLeaf) \leq f_{leaf}(pr_l)$, where $nextLeaf \neq pr_l$. If two leaves have the same f_{leaf} value, then $nextLeaf$ is chosen as the leftmost leaf.

Let $leaf = (pr_l, ?)$, then:

$$f_{leaf}(leaf) = \#\; of \; \$ \; entries \; in \; pr_l$$

Once we have identified $nextLeaf$, f_{trans} will be used to choose the appropriate transition. There are two types of transitions to choose from, depending on the characteristics of $newLeaf$.

1. Can the sol-entry of $newLeaf$ be updated to 'yes'?
2. Should $newLeaf$ be expanded further using the Div relation?

Priority should align with updating the sol-entry of a leaf, if we can. Thus, we use case 1, if $pr_{newLeaf}$ is solved, in accordance with the definition of a solved problem in section 3.2. Otherwise, f_{trans} follows case 2, where $newLeaf$ is expanded as defined by Div.

3.3.1 Additional Definitions

$$Constr^* : Prob \rightarrow \{True, False\}$$

The definition of $Constr^*$ is context-dependent.

If $pr \in Prob$ is located at the root node, then:

$$Constr^*(pr) = True \iff \forall sl_{\$a} \; in \; pr, (such \; that \; sl_{\$a} \neq \$), \; sl_{\$a} \; satisifes \; all \; the \; hard \; constraints$$

Otherwise:

Let $pr = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$j}, \ldots, sl_{\$p} \rangle$ such that $sl_{\$j} = \$$ and $j \neq 1$ and $\forall sl_{\$i}$, such that $1 \leq i < j$, $sl_{\$i} \neq \$$. Then,

$$Constr^*(pr) = True \iff sl_{\$j-1} \; satisfies \; all \; the \; hard \; constraints$$

Less formally, $Constr^*$ is used check if all hard constraints are satisfied for a partial assignment. For optimization, we only need to check over the differences between a child vector and its parent vector. The reason being, is that we already performed the check over the inherited elements in previous steps and it would be unnecessary to perform them again. Since the root problem has no parent, we must check over all non-$ assignments. For every other node, it only differs from its parent by a replacement of the leftmost $ in parent vector with a slot.

$Eval^*$ is an extension of $Eval$ to work on partial assignments (i.e. those that include some $ elements).

$$Eval^* : Prob \rightarrow \mathbb{N}$$

$$Eval^*(pr) = \sum_{i=1}^{p} penalties(sl_{\$i})$$

where $pr = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$p} \rangle$, and

$$penalties(sl_{\$i}) = \begin{cases} 0 & if \; sl_{\$i} = \$ \\ soft \; constraint \; penalty \; for \; assigning \; sl_{\$i} \; to \; cl_i & otherwise \end{cases}$$

Also, throughout the search process, we keep track of our best solution so far (if any) as the leaf with a 'yes' sol-entry, $Constr$ satisfied and with the lowest $Eval$ value. This $Eval$ value is saved in a variable denoted $bestEval$, while the solution vector can be denoted $bestSol$. Everytime, the search finds a more optimal solution, these two fields will be updated.

3.4 Search Instance

$$Ins_\wedge = (s_0, G_\wedge)$$

It should be noted that some pre-processing will occur before initializing the search. In particular, the forced assignments specified by *partassign* must be checked for any contradictions (e.g. partassign(course1) = slot1 and partassign(course1) = slot 2). If a contradiction is found, we won't start the search since we know that no valid solution will ever be found. Otherwise, we can initialize the search as follows:

Let $pr = \langle sl_{\$1}, sl_{\$2}, \ldots, sl_{\$p} \rangle$, where $sl_{\$i} = partassign(cl_i)$ (or $sl_{\$i} = \$$ if there is no partassign for cl_i).

The start state is defined as:

$$s_0 = (pr, ?)$$

The goal condition is defined as:

$G_\wedge(s) = yes \iff$
$s = (pr', yes)$, or
$s = (pr', ?, b_1, \ldots, b_n)$, $G_\wedge(b_1) = \cdots = G_\wedge(b_n) = yes$ and the solutions to $b_1, \ldots, b_n$ are compatible with each other, or
There are no remaining transitions that have not already been tried out.

3.5 Example

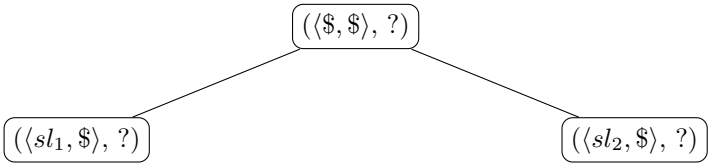
$Courses = \{c_1\}$
$Labs = \{l_{11}\}$
$Slots = \{sl_1, sl_2\}$
$coursemax(sl_1) = 2$
$coursemax(sl_2) = 2$
$labmax(sl_1) = 2$
$labmax(sl_1) = 2$
$unwanted(c_1, sl_2)$
We index our courses and labs so that they are in the following order: (c_1, l_{11})

STEP 1:

$(\langle \$, \$ \rangle, ?)$

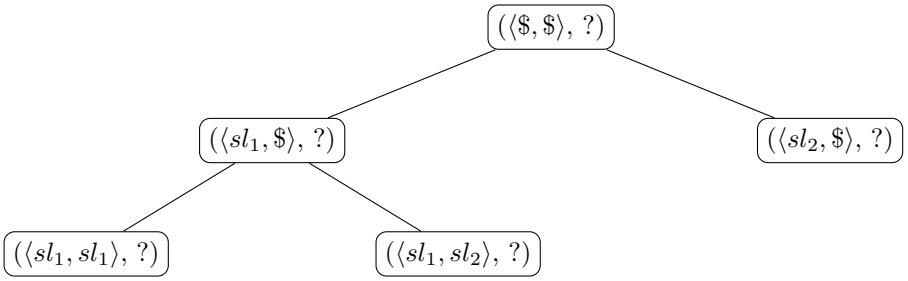
Initialize the root.

STEP 2:



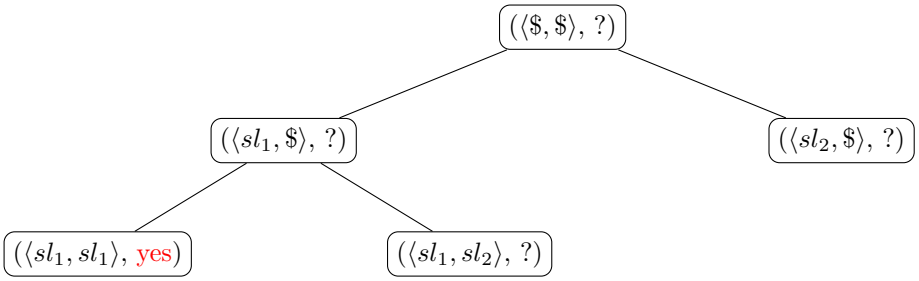
f_{leaf} selects the root since it has lowest number of \$. The problem is not solved, so we expand this leaf.

STEP 3:



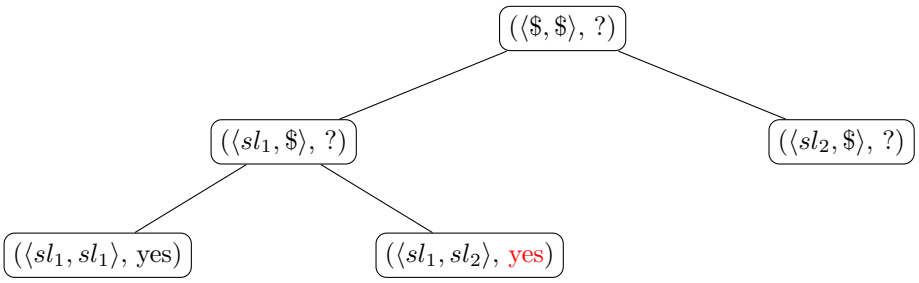
f_{leaf} selects this leaf since it has lowest number of \$ and is the leftmost. The problem is not solved, so we expand this leaf.

STEP 4:



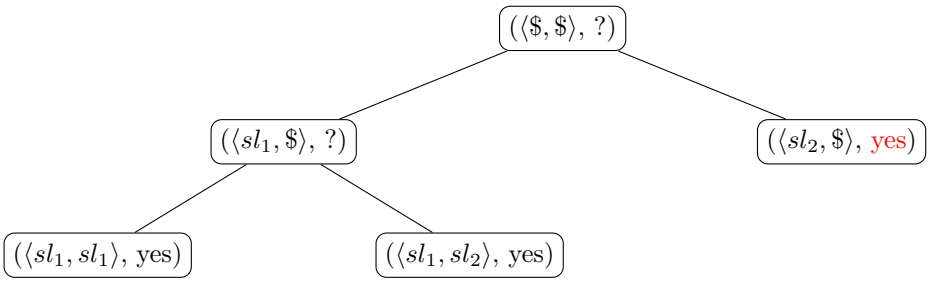
f_{leaf} selects this leaf since it has lowest number of \$ and is the leftmost. This leaf problem was solved, but the assignment vector is invalid since it violates the hard constraint that a course and associated lab can't be assigned to the same slot.

STEP 5:



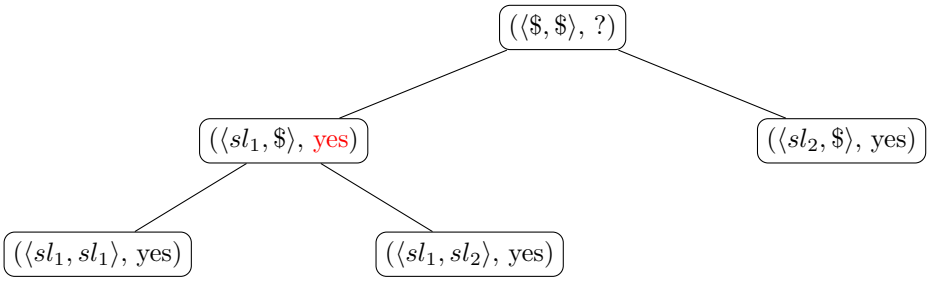
f_{leaf} selects this leaf since it has lowest number of \$. This leaf problem was solved, and the assignment vector is the first valid solution we have found. So we update $bestEval$ and $bestSol$ accordingly.

STEP 6:



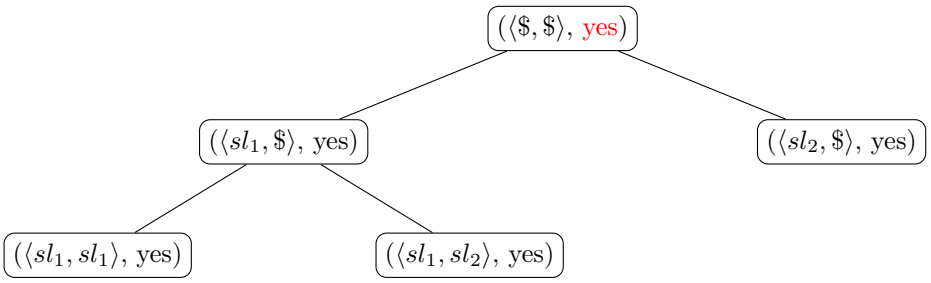
f_{leaf} selects this leaf since it has lowest number of \$. This leaf problem was solved, since the assignment vector violates the hard constraint, $unwanted(c_1, sl_2)$.

STEP 7:



Now that all leaves are solved, we can change the parent nodes to be solved as well, until every node is solved.

STEP 8:



All node problems are solved (and are compatible), thus our search ends and we can output our optimal solution $bestSol$ with a value of $bestEval$.

4 Conclusion

In this paper, we outlined two approaches that can be used to solve the Computer Science Department scheduling problem at the University of Calgary. The first of these approaches used a Genetic Algorithm based paradigm and the second approach used an And-Tree based paradigm. By outlining these approaches, we hope to provide optimal solutions in an efficient amount of time in which we can implement.