CPSC 433 — Assignment I

Daniel Gillson
David Keizer
Ethan Higgs
Zheng Liang
Zibin Mei

October 20th 2017

Search Paradigm I — Set-Based Search

Solution Overview

Our approach to solving the problem within the context of set-based search will rely mainly on the use of a genetic algorithm. We will obtain an initial, randomly generated population for our GA via an Or-tree-based Search that will provide candidate solutions to the problem's hard constraints. In this first stage of our solution, we ignore the problem's soft constraints, thereby re-formulating it as a constraint satisfaction problem (CSP).

Once we have obtained an initial population of some size, pop_init, we will consider its individuals to be facts in a Set-based Search. We will also define a constant, pop_max, as an upper bound on the size of any later generation. The set-based search will use our genetic algorithm's cross-over/mutation function, and a reduce function as its extension rules. f_{Wert} will be used to decide which extension rule to apply. When the population has exceeded pop_max, f_{Wert} will indicate that a reduction is to take place. f_{Select} will be used to determine which two facts to use as parents when applying the cross-over/mutation extension rule. The start state will be our initial set of randomly generated facts and our end condition will be a predetermined number of generations or a predetermined time limit, whichever comes first.

In our GA, we will use **Eval** to measure how the well each fact meets soft constraints, **Constr** and **Constr*** to check hard constraints, and roulette-wheel selection as a selection method for cross-over/mutation. **Constr*** is a variant of **Constr** used to evaluate the constraint-compliance of partial solutions, which we assume can be derived from the provided **Constr** function.

Or-tree-based Search Model

We define Prob as a vector of length |Courses + Labs|, whose elements consist of the indices of slots from Slots, or the unassigned symbol, \$. The vector's ordering will preserve the original ordering of Courses + Labs. Therefore, a Prob vector can be read sequentially as: Course/Lab at position i, having value j, has been assigned time slot s_j, where s_j is the member s of the set Slots at the j^{th} index of that set. As such, our definition of Prob is equivalent with the notion of a partial assignment, *partassign*.

Before formally defining Prob, we define D_i, the domain of any element in a problem instance, pr, to be the set: $\{0, \ldots, j\}$ where $j + 1 = |\text{Slots}|$.

$$\begin{gathered}\text{Prob} = \langle C_1\text{slot}, \ldots, C_n\text{slot}, \ldots, L_{11}\text{slot}, \ldots, L_{1k_1}\text{slot}, \ldots, L_{n1}\text{slot}, \ldots, L_{nk_n}\text{slot}\rangle \\ \text{such that } C_i\text{slot}, L_{ik_i}\text{slot} \in D_i \cup \{\$\}\end{gathered}$$

Which can be abstracted into the form:

$$\text{Prob} = \langle X_1, \ldots, X_n\rangle \text{ such that } X_i \in D_i \cup \{\$\}$$

We define our alternatives relation, Altern, as follows:

$$\begin{gathered}\text{Altern} = \{(\langle X_1, \ldots, X_i, \ldots, X_n\rangle, \langle X_1, \ldots, d_{i1}, \ldots, X_{in}\rangle, \ldots, \langle X_1, \ldots, d_{i\ell}, \ldots, X_n\rangle) \mid X_i = \$, \\ 1 \leq i \leq n,\ |D_i| = \ell,\ D_i = \{d_i, \ldots, d_{i\ell}\}\}\end{gathered}$$

We define the notion, "pr is solved" as follows:

$$\text{pr} = \langle X_1, \ldots, X_n \rangle \text{ and } \forall i \text{ such that } 1 \leq i \leq n,\ X_i \neq \$, \text{ and pr is not unsolvable.}$$

We define the notion, "pr is unsolvable" as follows:

$$\text{pr} = \langle X_1, \ldots, X_n \rangle \text{ and } \mathbf{Constr}^*(\text{pr}) = \text{false}.$$

Since we have access to $\mathbf{Constr}^*$, derived from the provided function, **Constr**, we can allow $\mathbf{Constr}^*$ to perform the work of assessing whether any particular problem instance, pr, is compliant with the problem's hard constraints.

Or-tree-based Search Process

Initial Search Control:

This is the search control for the generation of the initial population for the set-based search.

Our O_{Tree} will begin with a single node, (pr, ?). Its expansion will be governed by the recursive relation, $\text{Erw}_{\vee}$. If a partial assignment is provided, pr, is of the form *partassign* and expansion will begin at the first \$ in pr. For all subsequent assignments, the search control will replicate the form of *partassign* wherever it had made an assignment.

In general, the search control will construct pr in a left to right fashion, assigning slots to elements of pr from the lowest to the highest index.

In order to decide which leaf to operate on, the search control applies f_{leaf}. It first looks for leaf nodes where the problem is solvable. If there are multiple of these leaf nodes, then one is chosen at random. The sol-entry of the selected node is then changed to "Yes" and the search is finished.

If no such leaf nodes exist, then the search control considers any leaf nodes where the problem is unsolvable. If there are multiple of these nodes, then they are processed from left to right — updating their sol-entries to "No".

If neither of these types of leaf nodes exist, then the search control selects the leaf which has the fewest unassigned courses/labs (\$'s) — therefore the deepest leaf node in the tree. There will always be multiple of these nodes, since the search control works one element at a time, so one will need to be chosen randomly and Altern will be applied to this node.

In order to define f_{leaf}, we must first define a function, sum\$: pr $\Rightarrow \mathbb{Z}$. Which, given some pr, returns the number of \$ entries within it.

Given,

$$\text{P} = \{\{\text{pr}_1, \ldots, \text{pr}_k\} \mid \text{sum\$}(\text{pr}_i) = \text{sum\$}(\text{pr}_j),\ 1 \leq i, j \leq k,\ i \neq j\ \}$$

We define a function rand: pr $\Rightarrow \mathbb{R}$ to deal with multiple deepest leaf nodes.

Let z be a random integer such that $1 \leq z \leq |\text{P}|$. Then,

$$\text{rand} = \left\{ \begin{array}{ll} 0 & : \text{pr}_z \\ 0.01 & : \text{pr}_i \mid i \neq z,\ 1 \leq i \leq |\text{P}| \end{array} \right\}$$

Here $\text{pr}_i = \{\text{X}_1, \ldots, \text{X}_n\}$ and input is a *partassign* $= \{\text{y}_1, \ldots, \text{y}_n\}$, which is either the *partassign* given to the search as the input or s_0, as defined the the search instance.

Now we define f_{leaf}: $\text{pr}_i \Rightarrow \mathbb{R}$

$$\text{f}_{\text{leaf}} = \left\{ \begin{array}{ll} -1 & : \text{pr}_i \text{ is solvable} \\ 0 & : \text{pr}_i \text{ is unsolvable} \\ \text{sum\$}(\text{pr}_i) + \text{rand}(\text{pr}_i) & : \forall X_{\text{i}} = \text{y}_{\text{i}} \text{ or } \text{y}_{\text{i}} = \$ \end{array} \right\}$$

At each tree depth, our search control will apply the process described above in order to either continue the expansion of O_{Tree}, or return a solution.

Or-tree-based Search Instance:

We define our initial search state, s_0, as follows:

$$s_0 = \text{pr} = \langle X_1, \ldots, X_n \rangle \text{ such that, } \forall X_i \in \text{pr},\ X_i = \$$$

However, s_0 will occasionally be equal to a partial assignment, *partassign*.

Our goal state, $G_\vee$, is equivalent with the definition of the notion, "pr is solved", as seen above.

Set-based Search Model

We define a set of facts, F, to be a set of candidate solutions produced by an Or-tree-based Search as described above. As mentioned in the problem overview, we will assume that our initial set F will be of size pop_init.

Before formally defining F, we define D_i, the domain of any element in an individual, F_i, to be the set: $\{0, \ldots, j\}$ where $j + 1 = |\text{Slots}|$.

$$F = \{\langle X_1, \ldots, X_n\rangle_1, \ldots, \langle X_1, \ldots, X_n\rangle_k \mid X_i \in D_i,\ 1 \leq i \leq n,\ k = \text{pop\_init}\}$$

Before defining Ext, we will first define an alternative Search Control for the Or-tree-based Search which will be applied when using the cross-over/mutation extension rule.

Search $\text{Control}_{\text{Alt}}$:

Our O_{Tree}, once again, will begin with a single node, (pr, ?); and its expansion will be governed by the recursive relation, $\text{Erw}_{\vee}$.

If both of the selected parents, A and B, have the same first element, then pr will be initialized with that element's value as its first element, followed by \$'s. Otherwise pr will be initialized with all \$'s, as before.

The only difference between Search $\text{Control}_{\text{Alt}}$ and the initial search control is that the two parents must be considered in f_{leaf}.

As such, we redefine the set P used by the rand function as follows:

$$\begin{gathered}\text{P} = \{\{\text{pr}_1, \ldots, \text{pr}_k\} \mid \text{sum\$}(\text{pr}_i) = \text{sum\$}(\text{pr}_j),\\ (\text{X}_{i_\ell} = \text{par}_{1_\ell} \text{ or } \text{X}_{i_\ell} = \text{par}_{2_\ell} \text{ and } \text{X}_{j_\ell} = \text{par}_{1_\ell} \text{ or } \text{X}_{j_\ell} = \text{par}_{2_\ell}, \text{ or}\\ \text{X}_{i_\ell} \neq \text{par}_{1_\ell} \text{ and } \text{X}_{i_\ell} \neq \text{par}_{2_\ell} \text{ and } \text{X}_{j_\ell} \neq \text{par}_{1_\ell} \text{ and } \text{X}_{j_\ell} \neq \text{par}_{2_\ell}), \text{ and}\\ 1 \leq i, j \leq k, i \neq j\ \}\end{gathered}$$

Where ℓ is the index of the pr's under consideration at the current tree depth. By our definition, P will either contain all the leaf nodes where the value of the ℓ'th element matches one of the two parents, or P will contain all the leaf nodes where no match is found.

We denote the parents as follows:

$$\text{Par}_1 = \{\text{par}_{1_1}, \ldots, \text{par}_{1_n}\} \text{ and } \text{Par}_2 = \{\text{par}_{2_1}, \ldots, \text{par}_{2_n}\}$$

If both parent elements at the position under consideration constitute valid additions to pr, one will be chosen randomly, according to f_{leaf}. Otherwise, if neither parent element would be a valid addition to pr, a valid option generated by Altern will be chosen randomly.

$$\text{f}_{\text{leaf}} = \left\{ \begin{array}{ll} -1 & : (\text{pr}_i \text{ is solved})\\ 0 & : (\text{pr}_i \text{ is unsolvable})\\ \text{sum\$}(\text{pr}_i) + \text{rand}(\text{pr}_i) & : \text{X}_i = \text{par}_{1_i} \text{ or } \text{X}_i = \text{par}_{2_i}\\ \text{sum\$}(\text{pr}_i) + \text{rand}(\text{pr}_i) + 0.1 & : \text{X}_i \neq \text{par}_{1_i} \text{ and } \text{X}_i \neq \text{par}_{2_i}\\ \infty & : \text{else} \end{array} \right\}$$

Now we will define the set of extension rules, Ext:

1. Cross-Over/Mutation:
   (a) $A = \mathrm{f}_{\mathrm{select}}(F)$, $B = \mathrm{f}_{\mathrm{select}}(F - A)$. (We will define $\mathrm{f}_{\mathrm{select}}(F)$ in our Search Process).
   (b) Now that we have selected two parent facts, we will execute our Or-tree-based Search using Search $\mathrm{Control}_{\mathrm{Alt}}$.
   (c) The resulting solution produced by this search will then be added to the population.
2. Reduce:
   (a) The set F is sorting according to **Eval**: sort(F, **Eval**)
   (b) The first individual in F is then removed: $F = F$ - F_0
   (c) This brings the size of F back down to pop_max, since we only ever add a single individual at a time.

Set-based Search Process

Here we define the function, $\mathrm{f}_{\mathrm{select}}(F)$, which will be used in Ext to pick individuals out of the population in order to perform the crossover operation on them.

$\mathrm{f}_{\mathrm{select}}$: $F \Rightarrow (F_i)$ is an implementation of the roulette-wheel selection algorithm.

```
f_select(F) {
    sum = 0
    for A_i in F do
      A_i.Eval = Eval(A_i)
      sum ++ A_i.Eval
    end for
    for A_i in F do
      A_i.Eval_norm = A_i.Eval / sum
    end for
    sort(F, A_i.Eval_norm)
    for A_i in F do
      for A_j in F and A_j.index < A_i.index do
        A_i.ACNF += A_j.Eval_norm
      end for
    end for
    rand r = randFloatBetween(0, 1)
    selected = NULL
    index = 0
    while true do
      selected = F_index
      if F_(index + 1).ACNF ≥ r then
        return selected
      end if
      index ++ 1
    end while
    }
```

Search Control:

The search control for our Set-based Search will be responsible for producing the initial set, F, tracking the amount of time elapsed, and counting the number of generations that our GA has modelled. The first thing our control will do is start a timer.

In order to produce F, the control will execute the following algorithm:

produce(F) {
  $i = 0$
  while $i <$ pop_init **do**
    if *partassign* $\in$ input **then**
      individual = Or-tree-based Search execution with s_0 = *partassign*
    else
      individual = Or-tree-based Search execution with s_0 as originally defined in our Or-tree-based Search Instance
    end if
    $F = F +$ individual
    i ++ 1
  end while
  }

Once F has been produced, our GA will begin its work.

We also define a function pop(F) which evaluates the size of the current population, where F is the current set of facts.

$$\text{pop(F)} = \left\{ \begin{array}{ll} -1 & : |\text{F}| > \text{pop\_max} \\ 1 & : \text{else} \end{array} \right\}$$

In order to decide which transition to apply to our set of facts we pick the transition that minimizes f_{Wert}.

$$\text{f}_{\text{Wert}}(\text{A, B}) = \left\{ \begin{array}{ll} \text{pop(F)} & : \text{B} = F - F_0 \\ 0 & : \text{B} = \text{else} \end{array} \right\}$$

f_{Wert} evaluates to -1 if the transition under consideration is reduce and the size of the current generation is larger than pop_max. It evaluates to 1 when considering reduce and the current generation is not too large. Lastly, f_{Wert} evaulates to 0 whenever the cross-over/mutate rule is being considered.

Each time the cross-over/mutate rule is applied, the generation counter will increase by 1. If after any application of an extension rule, the time limit is up or the generation counter reaches the predetermined maximum threshold, the search control will halt the GA. At that point, the most fit individual (according to the **Eval** function) in the current population will be returned as the final, optimized solution to the assignment problem.

Set-based Search Instance

s_0 = Start state for the search instance — composed of pop_init candidate solutions.

$G_{Set}(s)$ = Yes, if and only if enough generations have been reached or enough time has elapsed.

That concludes our description of a Set-based Search solution to the assignment problem.

A simple example:

Slots = $\{s_1, s_2, s_3\}$, Courses = $\{c_1, c_2\}$, Labs = $\{l_{11}, l_{21}\}$

Hard Constraints:

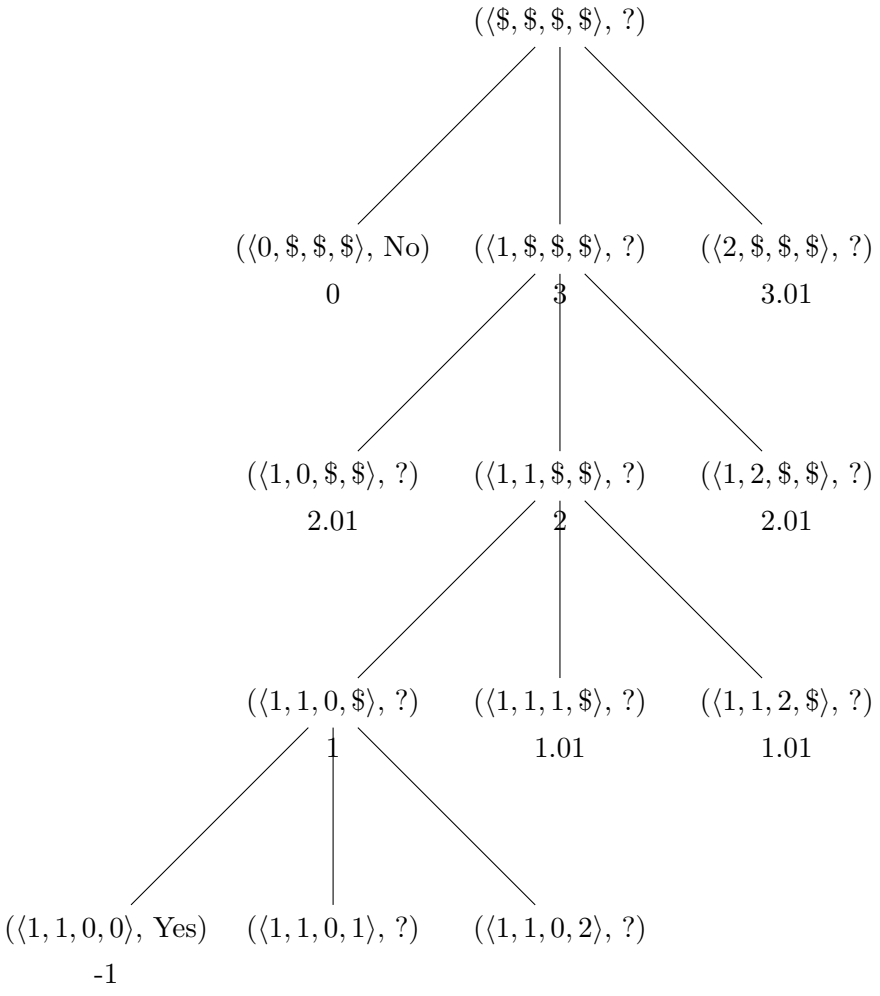
1. **Constr** and **Constr*** both equal false if c_1 is assigned s_1
2. **Constr** and **Constr*** both equal false if l_{21} is assigned s_3

Soft Constraints:

1. **Eval** and **Eval*** both equal 5 if c_1 is assigned s_2
2. **Eval** and **Eval*** both equal 5 if l_{21} is assigned s_1

Or-tree candidate #1 search derivation with the initial search control:

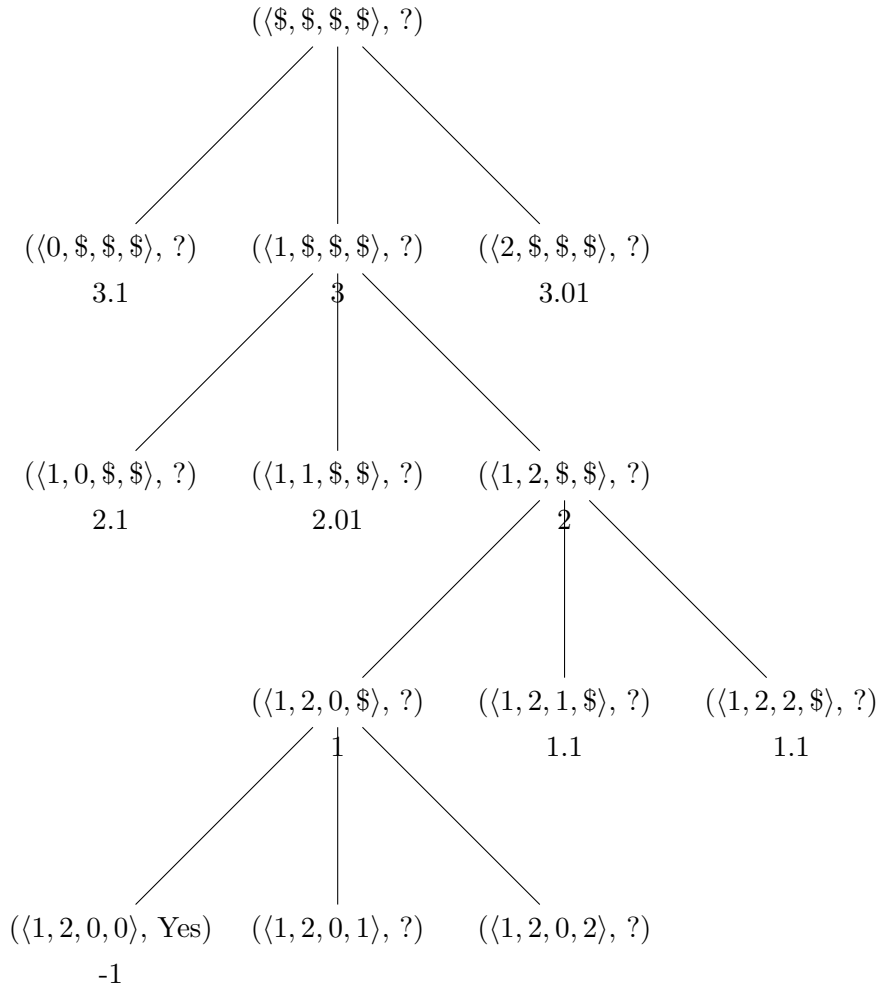
The values underneath each node are f_{leaf} values.



For the sake of brevity, we now assume that we have derived another candidate solution: $\langle 2, 2, 0, 1 \rangle$.

Or-tree cross-over/mutation search derivation with Search Control$_{\mathbf{Alt}}$:

$\text{Par}_1 = \langle 1, 1, 0, 0\rangle$, $\text{Par}_2 = \langle 2, 2, 0, 1\rangle$



According to the Soft Constraints:

1. **Eval**$(\langle 1, 1, 0, 0\rangle) = 10$
2. **Eval**$(\langle 2, 2, 0, 1\rangle) = 0$
3. **Eval**$(\langle 1, 2, 0, 0\rangle) = 5$

Therefore, $\langle 2, 2, 0, 1\rangle$ would be the optimal solution after modelling the GA for a single generation.

Search Paradigm II — And-Tree-Based Search

And-tree-based Search Model

For the model, Prob is defined as a vector of length |Courses + Labs|, whose elements consist of the indices of slots from Slots, or the unassigned symbol, \$. The ordering of vector will be the same as the original ordering of Courses + Labs. Therefore, a Prob vector can be read sequentially as: Course/Lab at position i, having value j, has been assigned time slot s_j, where s_j is the member s of the set Slots at the j^{th} index of Slots. As such, the definition of Prob is equivalent with the notion of a partial assignment, *partassign*.

The domain of any element in a problem instance, pr, referred to as D_i, is defined as the set: $\{0, \ldots, j\}$ where $j + 1 = |\text{Slots}|$. With that, Prob is defined as follows:

$$\text{Prob} = \langle C_1\text{slot}, \ldots, C_n\text{slot}, \ldots, L_{11}\text{slot}, \ldots, L_{1k_1}\text{slot}, \ldots, L_{n1}\text{slot}, \ldots, L_{nk_n}\text{slot}\rangle \\ \text{such that } C_i\text{slot}, L_{ik_i}\text{slot} \in D_i \cup \{\$\}$$

Which can be abstracted into the form:

$$\text{Prob} = \langle X_1, \ldots, X_n\rangle \text{ such that } X_i \in D_i \cup \{\$\}$$

We define our division relation, Div, as follows:

$$\text{Div} = \{(\langle X_1, \ldots, X_i, \ldots, X_n\rangle, \langle X_1, \ldots, d_{i1}, \ldots, X_{in}\rangle, \ldots, \langle X_1, \ldots, d_{i\ell}, \ldots, X_n\rangle) \mid X_i = \$, \\ 1 \leq i \leq n,\ |D_i| = \ell,\ D_i = \{d_i, \ldots, d_{i\ell}\}\}$$

We define "pr is solved" as follows:

$$\text{pr} = \langle X_1, \ldots, X_n\rangle \text{ and } \forall i \text{ such that } 1 \leq i \leq n,\ X_i \neq \$, \text{ and pr is not unsolvable.}$$

We define "pr is unsolvable" as follows:

$$\text{pr} = \langle X_1, \ldots, X_n\rangle \text{ and } \mathbf{Constr}^*(\text{pr}) = \text{false}.$$

Since we have access to $\mathbf{Constr}^*$, derived from the provided function, **Constr**, we can allow $\mathbf{Constr}^*$ to perform the work of assessing whether any particular problem instance, pr, is compliant with the problem's hard constraints.

And-tree-based Search Process

Having defined the search model, we are now ready to define the Search Process and Search Instance.

A_{tree} will begin with a single node, $s_0 = (pr, ?)$, and its expansion will be defined by the recursive relation $Erw_{\wedge}$. Each iteration will use Div to expand the tree and create new nodes. After each Div, F_{bound} evaluates all solved leaves via a branch and bound operation, using a *beta* value. The search control uses this β value to prune once a solution has been found. After this, F_{leaf} evaluates all the leaves, and calculates a number, assessing each leaf. The search control will prioritize applying Div to the lowest value leaves first. The search control will choose the left most leaf in the case that F_{leaf} provides a tie between multiple leaves.

F_{leaf} evaluates a penalty score of an assignment, based on the soft and hard constraints. For partial assignments, $F_{leaf}{}^*$ is used, which uses $Eval^*$ and $Constr^*$ instead.

$F_{leaf} : pr_i \rightarrow \mathbb{R}$

$$\mathrm{F_{leaf}} = \left\{ \begin{array}{ll} \infty & : \text{if } Constr(pr_i) = \text{false} \\ Eval(pr_i) & : \text{else} \end{array} \right\}$$

F_{bound} is used by the search control to keep the tree size within reason. It sets the β values of all complete solutions, when the state is (pr, Yes). We can define F_{bound} as follows:

$F_{bound} : \{pr_1, ..., pr_i, ..., pr_n\}$ where $1 \leq i \leq n$

$$\mathrm{F_{bound}} = \left\{ \begin{array}{ll} \beta = \beta & : \text{if } (pr_i, ?) \\ \beta = min(Eval(pr_i), \ldots, Eval(pr_n)) & : \text{else, for all } pr_i \text{ having the sol-entry ``Yes''} \end{array} \right\}$$

As there is is only one Div relation, F_{trans} is not used. Additionally, there is no backtracking in this search control.

The search control operation operates in the following order:

1. Apply Div to the tree, prioritizing the leaf node with the lowest F_{leaf} value. In case of a tie, the deepest, leftmost leaf node is used.
2. Apply F_{bound}.
3. Apply F_{leaf}. If $F_{leaf} \geq \beta$, then prune the leaf.
4. We check leaf nodes to see if they are solved: We mark a leaf node, pr_i as (pr_i, Yes) if $\forall X_j \in pr_i$, $X_j \neq \$$ and $F_{leaf}(pr_i) \neq \infty$
5. We choose the smallest F_{leaf} value of the leaves that are marked $(pr, ?)$ and apply Div again, starting the process over again.

And-tree-based Search Instance

As before the initial search state is s_0:

$$s_0 = pr = \langle X_1, ..., X_n \rangle \text{ such that } \forall X_i \in pr, X_i = \$ \text{ or}$$
$$s_0 \text{ is some partial assignment given as an input to the search.}$$

We set $\beta = \infty$ at the beginning of the search.

The goal state is G_{and} is reached when all leaf nodes are marked (pr, Yes).

The optimal solution is the leaf node with the lowest F_{leaf} value.

A simple example:

Slots = $\{s_1, s_2, s_3\}$, Courses = $\{c_1, c_2\}$, Labs = $\{l_{11}, l_{21}\}$

Hard Constraints:

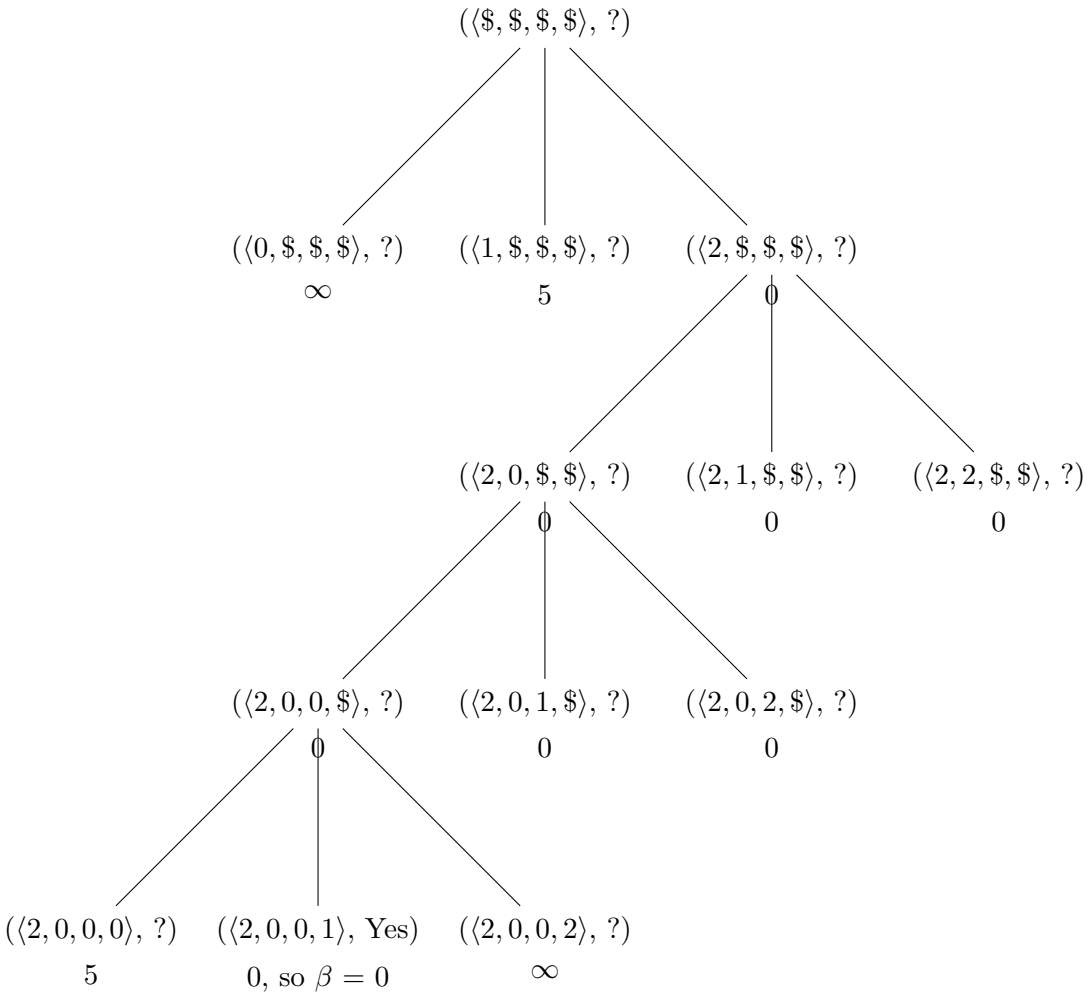
1. **Constr** and **Constr*** both equal false if c_1 is assigned s_1
2. **Constr** and **Constr*** both equal false if l_{21} is assigned s_3

Soft Constraints:

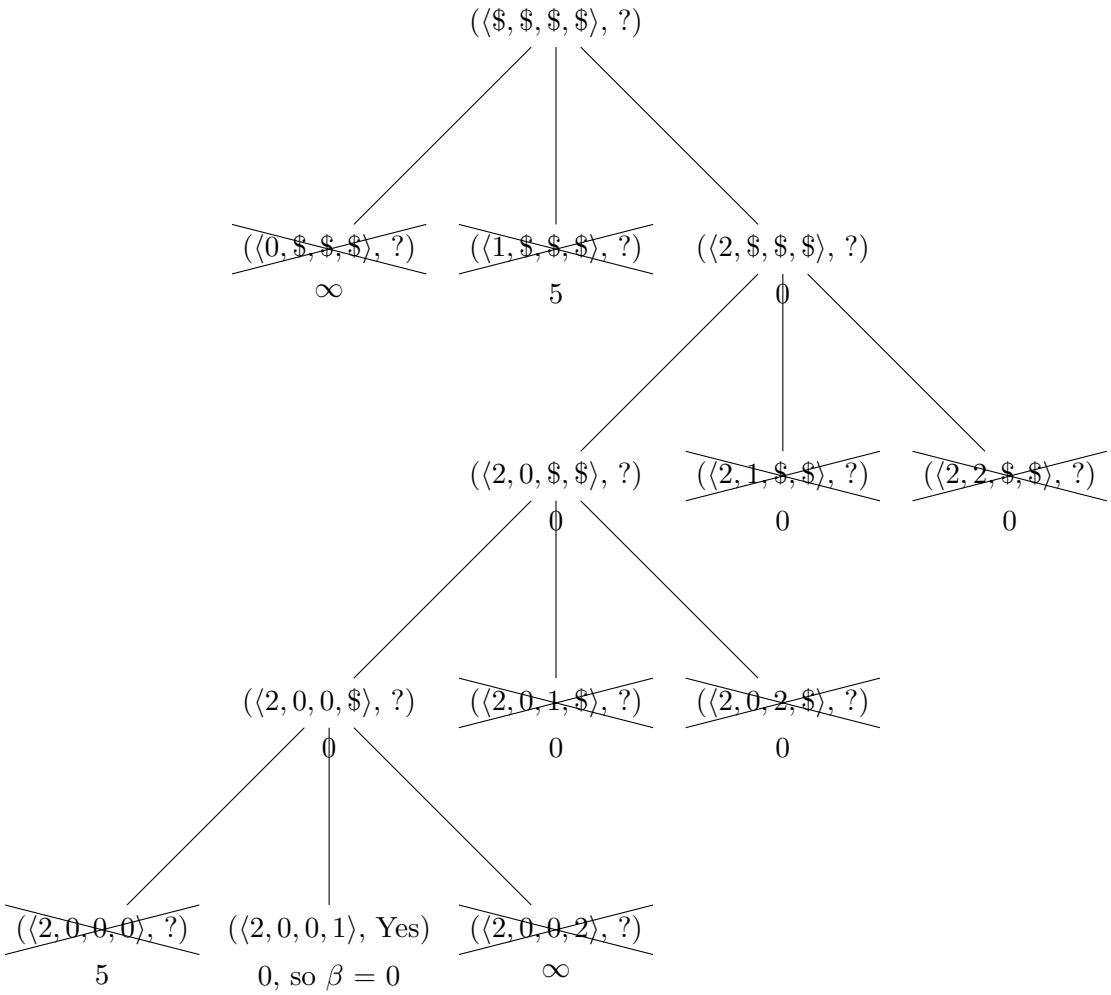
1. **Eval** and **Eval*** both equal 5 if c_1 is assigned s_2
2. **Eval** and **Eval*** both equal 5 if l_{21} is assigned s_1

And-tree search derivation:

The values underneath each node are f_{leaf} values. Also, $\beta = \infty$.



Since a solution has been found and β updated, the search control prunes all leaf nodes where $f_{leaf} \geq \beta$.



In this example, all nodes are pruned after the first solution is found, therefore the final, optimized solution is $\langle$ 2, 0, 0, 1 $\rangle$.