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. $\mathrm{f}_{\mathrm{Wert}}$ will be used to decide which extension rule to apply. When the population has exceeded pop_max, $\mathrm{f}_{\mathrm{Wert}}$ will indicate that a reduction is to take place. $\mathrm{f}_{\mathrm{Select}}$ 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 $j + 1 = |\text{Slots}|$.

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

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 alternatives relation, Altern, as follows:

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

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, $\text{Erw}_{\vee}$. Typically, the first step our control will take is to apply Altern to the root node, with $i = 1$, then increment i. However, if s_0 is a partial assignment, then i will be the index value of the first \$, rather than 1. After each subsequent application of Altern, the search control will increment i, then perform a constraint analysis on each new successor node of O_{Tree} using $\mathbf{Constr}^{*}$. As such, the search control works one element (X_i) at a time, from left to right across each problem instance, until a solution is found.

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 $1 \leq z \leq k$. Then,

$$\text{rand} = \left\{ \begin{array}{ll} 0 & : \text{pr}_z \\ 0.5 & : \text{pr}_i \mid i \neq \text{z},\ 1 \leq i \leq k \end{array} \right\}$$

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

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

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 $j + 1 = |\text{Slots}|$.

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

Before defining Ext, we will first define an alternative Search Control for the Or-tree-based Search used for generating candidate solutions.

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

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

For each subsequent element of pr, pr$_i$, $\textbf{Constr}^{*}(\text{pr}_i)$ will be calculated for both A and B. If both parent elements constitute valid additions to pr, one will be chosen randomly — generating a single new successor node with a pr vector that reflects the addition of that element.

Otherwise, if neither parent element constitutes a valid addition to pr, Altern will be applied. Leaf selection will occur as in the original search control, using the same f$_{\text{leaf}}$ function and randomized selection from among the set of leaves of equal proximity to a solution. Next, i will be incremented and the process repeated.

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

1. Cross-Over/Mutation:
   (a) $A = \text{f}_{\text{select}}(F)$, $B = \text{f}_{\text{select}}(F - A)$. (We will define $\text{f}_{\text{select}}(F)$ in our Search Process).
   (b) Now that we have selected two parent facts, we will execute our Or-tree-based Search using Search Control$_{\text{Alt}}$.
   (c) The resulting solution produced by this search will then be added to the population.
2. Reduce:
   (a) k facts are removed from F, such that $k = |F|$ - pop_max.

Set-based Search Process

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

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

```
$f_{\text{select}}(F)$ {
  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 = $F_{\text{index}}$
    **if** $F_{(\text{index } + \text{ } 1)}$.ACNF $\geq$ r **then**
      return selected
    **end if**
    index ++ 1
  **end while**
  }
```

Next we define the function, reduce(F), which will be used in Ext to reduce the size a generation that has become too large.

```
reduce($F$) {
  $k = |F|$ - pop_max
  sort($F$, **Eval**)
  $i = 0$
  **while** $i < k$ **do**
    $F = F$ - $F_i$
    $i$ ++ 1
  **end while**
  }
```

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

$$\text{pop}(F) = \left\{ \begin{array}{ll} -1 & : |F| \leq \text{pop\_max} \\ 0 & : \text{else} \end{array} \right\}$$

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.

In order to decide which transition to apply to our set of facts we apply $\mathrm{f}_{\mathrm{Wert}}$.

$$\mathrm{f}_{\mathrm{Wert}}(\mathrm{A, B}) = \left\{ \begin{array}{ll} \text{Cross-Over/Mutate} & : \mathrm{pop}(F) == \text{-1} \\ \text{Reduce} & : \mathrm{pop}(F) == 0 \end{array} \right\}$$

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.
$\mathrm{G}_{\mathrm{Set}}(\mathrm{s})$ = 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.

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:

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

We define "pr is solved" as follows:

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

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 **Constr***, derived from the provided function, **Constr**, we can allow **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 Control.

A_{tree} will begin with a single node, $s_0 = (pr, ?)$, and its expansion will be defined by the recursive relation $Erw_\wedge$. At each tree depth, the division relation, *Div*, will be used to expand the tree and create new nodes. After each application of *Div*, the search control will prune leaves that are irrelevant to our search.

via branch and bound, by eliminating all leaves where F_{bound}(leaf) is less than the current $beta_{\min}$.

After this, the search control will use F_{leaf} to evaluate each remaining leaf at that tree depth. The search control will choose the leftmost leaf in the case that F_{leaf} provides a tie between multiple leaves. The search control will prioritize applying *Div* to the lowest value leaves first.

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

$F_{penalty} : \{pr_1, \ldots, pr_n\} \rightarrow \mathbb{R}$ where $1 \leq i \leq n$

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

Using this we can define F_{leaf} :

$F_{leaf} : \{pr_1, \ldots, pr_n\} \rightarrow \mathbb{R}$

$F_{leaf} = (F_{penalty}(\{pr_1, \ldots, pr_n\}))$

F_{leaf} applies $F_{penalty}$ in order to calculate a numeric value for the search control.

F_{bound} is used by the search control to keep the tree size within reason. It uses β pruning to remove leaves that fail to beat the best found solution so far. We can define F_{bound} using F_{bound}. Once again we can use $F_{bound}{}^*$ to evaluate partial assignments using $F_{penalty}{}^*$.

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

$$\mathrm{F_{bound}} = \left\{ \begin{array}{l} \beta = \infty\text{: if } pr_i \in s_0 \\ \beta = F_{penalty}\text{: else} \end{array} \right\}$$

β_{best} is the smallest β value that F_{bound} or $F_{bound}{}^*$ has evaluated to.

if $\beta_{pr_i} \leq \beta_{best}$ then $\beta_{best} = \beta_{pr_i}$ if $\beta_{pr_i} > \beta_{best}$ then $pr_i = null$, pruning the leaf from the tree.

As our chosen *Div* relation produces all possible subdivisions of a given problem instance, F_{trans} is not used. Additionally, backtracking is not necessary when using an And-tree to solve this particular problem.

A state is marked as *solved* when: $\forall X \in Prob, X_i \cup \{\$\} = \emptyset$ where X_i is some X in *Prob*. It is the state where each course and lab has a slot assigned to it. The state will take on the form (pr, solved).

The search control operation operates in the following order:

1. Apply F_{leaf} to the tree.
2. Apply Div to the tree, prioritizing the branch with the lowest F_{leaf} value. In case of a tie, the left most branch is used. Div is applied to all unsolved branches $(pr, ?)$. It is at this point that all lower leaves are checked for $(pr, solved)$.
3. Apply F_{bound} to the tree, pruning the leaves that are out of bounds if needed.

And-tree-based Search Instance

As before the initial search state is s_0:

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

The goal state is $G_\wedge$ is reached when all remaining leaves are of the form: $(pr, solved)$.