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 crossover/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 crossover/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 crossover/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:

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

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

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

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

$$\begin{gathered}\text{pr} = (X_1, \ldots, X_n) \text{ and there is a constraint } C_i = R_i(X_1, \ldots, X_k) \\ \text{such that } \exists X_{ij} \in \text{pr with a value unequal to \$ and } (X_1, \ldots, X_k) \text{ do not satisfy } R_i.\end{gathered}$$

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.

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.

In order to decide which leaf to operate on, the search control first looks for leaf nodes where the problem is solved. 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. If no such leaf nodes exist, then the search control considers any leaf nodes where the problem is unsolvable. Once again if there are multiple of these nodes, then one is selected at random. The sol-entry is then changed 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. If there are multiple of these nodes, one will 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.

We also want to define a function rand: pr $\Rightarrow \mathbb{R}$.

Here we have a set of problems P = $\{pr_1, \ldots, pr_k\}$ | sum\$(pr_i) = sum\$(pr_j), 1 $\geq$ i,j $\geq$ k

z is a random integer such that 1 $\geq$ z $\geq$ | P |

$$\text{rand} = \left\{ \begin{array}{ll} 0, & : pr_z \\ 0.01, & : pr_i \mid i \neq z,\ 1 \geq i \geq \mid P \mid \end{array} \right\}$$

f_{leaf}: $pr_i \Rightarrow \mathbb{R}$

Here $pr_i = \{x_1, \ldots, x_n\}$ and input is a *partassign* = $\{y_1, \ldots, y_n\}$, which is either the *p*artassign given to the search as the input or y_i such that $\forall y_i \in$ input, y_i = \$

$$f_{leaf} = \left\{ \begin{array}{ll} -1, & : (pr_i \text{ is solved}) \\ 0, & : (pr_i \text{ is unsolvable}) \\ \text{sum\$}(pr_i) + \text{rand}(pr_i), & : \forall x_i = y_i \text{ or } x_i = \$ \\ \infty & : \text{else} \end{array} \right\}$$

Crossover Search Control:

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

This search control is similar to the initial search control. The only difference comes in when there are no solved or unsolvable leaf nodes. The leaf nodes with problems containing the fewest \$'s are considered however the tiebreaker is dependant one the parents used for the crossover. Leaf nodes with a problem, where each entry in the vector is either equal to the entry in the same position of either parent's vector or \$ are then chosen. These are chosen between randomly if multiple leaf nodes fitting this description exist. If none of these exist, then select randomly from the other leaf nodes with the fewest \$'s.

Here the two parents are relevant and will be denoted by $\text{Par}_1 = \{\text{par}_{1_1}, \ldots, \text{par}_{1_n}\}$ and $\text{Par}_2 = \{\text{par}_{1_2}, \ldots, \text{par}_{2_n}\}$

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

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*. This will be stipulated more formally in the search control for our Set-based Search.

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

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

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

1. Crossover:
   (a) A = select(F), B = select($F - A$). (We will define select(F) in our Search Process).
   (b) Now that we have selected two parent facts, we will execute our Or-tree-based Search using the Crossover Search Control.
   (c) The resulting solution produced by this search will then be added to the population.
2. Reduce:
   (a) A fact is removed from the set of facts.
   (b) A $\rightarrow$ B | B = $\emptyset$
3. Populate:
   (a) A new fact is generated by running an Or-tree-based Search using the initial search control. Where the starting state of the tree is either as defined in our Or-tree-based Search instance or *partassign* if *partassign* was given as input.
   (b) A $\rightarrow$ B | A = $\emptyset$

Set-Based Search Process

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

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

```
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
  }
```

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

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

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

$$\text{f}_{\text{Wert}}(\text{A, B, e}) = \left\{ \begin{array}{ll} \text{-2(pop(F))} & : \text{A} = \emptyset \\ \text{2(pop(F))} & : \text{B} = \emptyset \\ -1 & : \text{else} \end{array} \right\}$$

We have already defined a method to select the two parents for the crossover extension and we don't need to pick a fact for the populate extension to operate on, however we do need to control which fact the reduce extension rule is applied to. We do this by using **Eval** to evaluate how well each fact follows the soft constraint. The fact with the maximum **Eval** score is used for the reduction.

f_{select} $(\{\text{A}_1 \rightarrow \text{B}_1, \ldots, \text{A}_\text{m} \rightarrow \text{B}_\text{m}\}, \text{e}) = \text{A}_\text{i} \rightarrow \text{B}_\text{i}$

where $\text{A}_\text{i} \neq \emptyset$ and $\textbf{Eval}(\text{A}_\text{i}) < \textbf{Eval}(\text{A}_\text{j})$, $\text{i} \neq \text{j}$, $1 \geq \text{i, j} \geq \text{m}$ or $\text{i} < \text{j}$

Every time after the crossover rule is applied, the generation counter will increase by 1. If after any application of an extension, 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.

And-tree 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, *partassgin*.

D_i, defined as the domain of any element in a problem instance, pr, to be the set: 0,...,j where $j + 1 =$ —Slots—. With that, Prob is defined as follows:

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

Which can be abstracted into the form:

$$\text{Prob} = ¡X_1,...,X_n¿ \text{ such that } X_i \in D_i \cup \{\$\}$$

The divide relation, Div, defined as:

$$\text{Div} = \{((X_1, ..., X_i, ..., X_n), (X_1, ..., d_{i1}, ..., X_{in}), ..., (X_1, ...d_{il}, ..., X_n)) —$$
$$X_i = \$, 1 \leq i \leq n, |D_i| = l, D_i = \{d_i, ..., d_{il}\}\}$$

"pr is solved" is defined as follows:

pr = $(X_1, ..., X_n)$ and $\forall i$ such that $1 \leq i \leq n$, $X_i \neq \$$, and pr is not unsolvable.

"pr is unsolvable" is defined as follows:

pr = $(X_1, ..., X_n)$ and there is a constraint $C_i = R_i(X_1, ..., X_k)$ such that $\exists X_{ij} \in$pr with a value unequal to \$ and $(X_1, ..., X_k)$ do not satisfy R_i.

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

Search Process

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

The And_{tree} will begin with a single node, $s_0 = (pr, ?)$, and its expansion will be defined by the recursive relation Erw_{and}. Each iteration will use Div to expand the tree and create new nodes. After each Div, F_{bound} prunes leaves that are irrelevant for our search via a branch and bound operation, using a *beta* value. After this, F_{leaf} evaluates all the leaves, and calculates a number that will correspond to the state. 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} 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}.

$F_{penalty} : \{pr_1, ..., pr_n\} \rightarrow \mathbb{R}$ 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} :

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

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

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 $F_{penalty}{}^*$.

$F_{bound} : \{pr_1, ..., pr_n\} \rightarrow pr_i$ 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 there is is only one Div relation, F_{trans} is not used.

There is no backtracking in this search control.

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.

Search Instance:

As before the initial search state is s_0:

$s_0 = pr =< X_1, ..., X_n >$ such that $\forall X_i \in pr, X_i = \$$

The goal state is G_{and} is reached when all branches are marked with $(pr, solved)$.

And-Tree-Based Search Process

$P_\wedge = (\ A_\wedge, K_\wedge)$:
$A_\wedge$ = the Search Model
$K_\wedge$ = the Search Control

$K_\wedge$ uses two functions: f_{Leaf}, and f_{Trans}, which compare all leaves of the tree representing

the state and select one, and select one of the transitions available to the selected leaf, respectively.

And-Tree-Based Search Instance

$Ins_{\wedge} = (s_0, G_{\wedge})$:
$s_0 = (pr, ?)$
$G_{\wedge}(s) =$ yes, if and only if:

- s = (pr', yes) or,
- s = (pr', ?, b_1, ..., b_n), $G_{\wedge}(b_1) = \ldots = G_{\wedge}(b_n) =$ yes and the solutions to b_1, ..., b_n are compatible with each other. Or,
- there is no possible transition that has not already been attempted and analyzed