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 remove 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 function, and a reduce function as its extension rules. The reduce function will remove individuals with large f_{Wert} values from the population until the population has been halved, should it ever reach a size of pop_max. We will use the provided **Eval** function to compute f_{Wert}. Furthermore, there will be no need for an f_{Select} function in our set based search, as transitions to each successive generation will be determined by our GA's selection function. The start state will be our initial set of randomly generated facts and our goal state will be: "Whichever occurs first: 50 generations have been modelled, or an hour has elapsed".

In our GA, we will use $f_{\text{Wert}} = \textbf{Eval}$ as a fitness function, **Cons** and **Cons*** for constraint-checking, and roulette-wheel selection as a selection method. **Cons*** is a variant of **Cons** used to evaluate the constraint-compliance of partial solutions, which we assume can be derived from the provided **Cons** 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:

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

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

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

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

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

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.

Or-tree-based Search Process

Search Control:

Our O_{Tree} will obviously begin with a single node, (pr, ?). Its expansion will be governed by the recursive relation, $\text{Erw}_{\vee}$. After each application of Altern, the search control will perform a constraint analysis on each new successor node of O_{Tree} using **Constr***. Nodes where **Constr*** evaluates to false will be pruned out of O_{Tree} prior to leaf selection.

The search control will pick a random leaf from among the minimal f_{leaf} values.

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.

f_{leaf}: $\{\text{pr}_1, \ldots, \text{pr}_n\} \Rightarrow \text{pr}_i$

$$\text{f}_{\text{leaf}} = \left\{ \begin{array}{ll} -1, & \text{if } (\text{pr}_i, \text{yes}) \\ 0, & \text{if } (\text{pr}_i, \text{no}) \\ \text{sum\$}(\text{pr}_i), & \text{if } (\text{pr}_i, ?) \end{array} \right\}$$

After f_{leaf} values have been calculated for each node, if there is a solved node then the search is finished and the solved problem instance is returned as output.

Otherwise, all unsolvable nodes will be pruned, and a random integer, r, will be generated such that $1 \leq r \leq n$, where n is equal to the number of remaining new successor nodes generated by Altern. Control will then select the r^{th} node and resume the expansion of O_{Tree}.

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

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

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

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

Our O_{Tree}, once again, will obviously 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.

For each subsequent element of pr, if A and B agree upon its value, control will select that value. Otherwise, if they disagree, Altern will be applied.

After each application of Altern, the search control will perform a constraint analysis on each new successor node of O_{Tree} using **Constr***. Nodes where **Constr*** evaluates to false will be pruned out of O_{Tree} prior to leaf selection.

Leaf selection will occur as in the original search control, using the same f_{leaf} function and randomized selection from among the set of leafs of equal proximity to a solution.

Once again, after f_{leaf} values have been calculated for each node, if there is a solved node then the search is finished and the solved problem instance is returned as output.

Otherwise, all unsolvable nodes will be pruned and the control will then select a random leaf as mentioned above and resume the expansion of O_{Tree}.

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

1. Cross-Over/Mutation:
   - (a) $A = \text{select}(F)$, $B = \text{select}(F - A)$. (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 Search $\text{Control}_{\text{Alt}}$.
   - (c) The resulting solution produced by this search will then be added to the population.
2. Reduce:
   - (a) If the size of the current generation equals pop_max, execute reduce(F). (We will define reduce(F) in our Search Process).

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 cross-over/mutation 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 = $F_{\text{index}}$
        **if** $F_{(\text{index + 1})}$.ACNF $\geq$ r **then**
            return selected
        **end if**
        index ++ 1
    **end while**
    }
```

Here we define the function, reduce(F), which will be used in Ext to reduce the size a generation that has become too large. The removal of individuals from the population will be performed quasi-randomly via the select(F) function, in order to avoid implementing an elitist GA.

```
reduce($F$) {
    **if** $|F|$ % 2 == 0 **then**
        target_size = $\frac{|F|}{2}$
    **else**
        target_size = $\frac{|F|+1}{2}$
    **end if**
    $i = 0$
    **while** $i$ < target_size **do**
        individual = select($F$)
        $F$ = $F$ - individual
      $i$ ++ 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.

Next, 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, a generation counter will be instantiated with the value "1". Following that, the search control will simply allow our GA to begin evolving its initial population. If at any point the timer reaches one hour or the generation counter reaches 50, the search control will halt the GA. At that point, the most fit individual 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 50 generations have been modelled, or one hour 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

$A_\wedge = (S_\wedge, T_\wedge)$:

Prob = set of problem descriptions

Div $\subseteq$ Prob$^+$ = division relation for generating sub-problems out of problem instances

$S_\wedge \subseteq A_{\mathrm{Tree}}$

$T_\wedge = \{(s_1, s_2) \mid s_1, s_2 \in S_\wedge$ and $\mathrm{Erw}_\wedge(s_1, s_2)\}$

Where A_{tree} is recursively defined by:

(pr, sol) $\in A_{\mathrm{Tree}}$ for pr $\in$ Prob, sol $\in$ {yes, ?}

(pr, sol, $b_1, \ldots, b_n$) $\in A_{\mathrm{Tree}}$ for pr $\in$ Prob, sol $\in$ {yes, ?}, $b_1, \ldots, b_n \in A_{\mathrm{Tree}}$

$\mathrm{Erw}_\wedge$ is a relation on A_{Tree} defined by:

$\mathrm{Erw}_\wedge$((pr, ?)) = (pr,yes) if pr is solved

$\mathrm{Erw}_\wedge$((pr, ?)) = (pr, ?, (pr_1, ?), …, (pr_n, ?)) if Div(pr, $pr_1, \ldots, pr_n$) holds

$\mathrm{Erw}_\wedge$((pr, ?, $b_1, \ldots, b_n$)) = (pr, ?, b_1', …, b_n'), if for an i:

$\mathrm{Erw}_\wedge(b_i, b_i')$ and $b_j = b_j'$ for $i \neq j$

$\mathrm{Erw}^*_\wedge$ is ignored here, as back-tracking is unnecessary when considering the problem at hand.

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

$\mathrm{Ins}_\wedge = (s_0, G_\wedge)$:

s_0 = (pr, ?)

$G_\wedge(s)$ = yes, if and only if:

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

We need to define:

- Prob
- Div
- $\mathrm{f}_{\mathrm{Leaf}}$
- $\mathrm{f}_{\mathrm{Trans}}$