CPSC433 Proposal Paper

October 2024

Team Floette

[redacted]

Jack Graver [redacted]

And-Tree

We chose an And-Tree based search as our first search paradigm for the given problem. In the following we will define the Model and Expansion Function, the Process and the Instance.

Model

Our search model is as follows

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

We must define Prob and Div, and the following are given;

$S_\wedge \subseteq$ **Atree**

where Atree is recursively defined by:

$(pr, sol) \in$ **Atree**for $pr \in$ Prob, $sol \in \{\text{yes}, ?\}$

$(pr, sol, b_1, \ldots, b_n) \in$ **Atree**for $pr \in$ Prob, $sol \in \{\text{yes}, ?\}, b_i \in$ **Atree**

$T_\wedge \subseteq S_\wedge \times S_\wedge$

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

Prob

Prob is defined by a tuple of sets representing assignments to the available slots

$$(GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m)$$

where GS_i represents a set of games assigned to the first slot, and similarly PS_j is for assigned practices. n and m are integers such that $0 \leq n \leq |Games|$ and $0 \leq m \leq |Practices|$. If $n = 0$ or $m = 0$, then no slots of their types are available.

This model will require each slot to be defined as either a game slot or practice slot so that,

$$\text{GS}_i \subseteq \text{Games}$$
$$\text{PS}_j \subseteq \text{Practices}$$

This will ensure independence of all Game and Practice slots.

The notation will allow the subscript i or j to not always be numerical, making it easier to name certain slots that have more importance (for example GS_{1M} refers to the first game slot on a Monday) but they will map back to the slot of the same type in the tuple where the first is 1, second is 2, until we have counted up to n or m.

The order of the slots in the tuple will also represent their time in the week relative to other slots that contain the same type (game or practice). However the relative ordering of slots of different types do not carry any meaning.

Div

Div will allow the choice of one game $g_i \in Games$ or one practice $p_i \in Practices$ to be assigned to any of the slots of its corresponding type such that the assignment does not violate any hard constraints. *Div* will also not allow g_i or p_i to be chosen if it is already assigned to a slot.

This can fit into the $Erw_\wedge$ relation by defining the leaf expansion property as follows.

$\text{Erw}_\wedge((pr, ?), (pr, \text{yes}))$	if pr is solved
$\text{Erw}_\wedge((pr, ?), (pr, ?, (pr_1, ?), \ldots, (pr_n, ?)))$	if $\text{div}(pr, pr_1, \ldots, pr_n)$ holds
$\text{Erw}_\wedge((pr, ?, b_1, \ldots, b_n), (pr, ?, b'_1, \ldots, b'_n))$	if for some $i : \text{Erw}_\wedge(b_i, b'_i)$ and $b_j = b'_j$ for $j \neq i$

If a game g_i is selected to be added, then $g_i \notin GS_{id}$ for any GS_{id}. The leaf expansion property of $Erw_\wedge$ will be,

$$
\begin{aligned}
Erw_\wedge(&((GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m), ?),\\
&((\boldsymbol{GS_1} \cup \boldsymbol{g_i}, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m), ?),\\
&((GS_1, \boldsymbol{GS_2} \cup \boldsymbol{g_i}, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m), ?),\\
&\cdots,\\
&((GS_1, GS_2, \ldots, \boldsymbol{GS_n} \cup \boldsymbol{g_i}, PS_1, PS_2, \ldots, PS_m), ?)).
\end{aligned}
$$

Where the branch with $((\ldots, \boldsymbol{GS_{id}} \cup \boldsymbol{g_i}, \ldots), ?)$ will be a valid branch included in the relation if and only if $GS_{id} \cup g_i$ does not violate any hard constraints.

Similarly, if a practice p_i is selected to be added, then $p_i \notin PS_{id}$ for any PS_{id}. The leaf expansion property of $Erw_\wedge$ will be,

$$
\begin{aligned}
Erw_\wedge(&((GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m), ?),\\
&((GS_1, GS_2, \ldots, GS_n, \boldsymbol{PS_1} \cup \boldsymbol{p_i}, PS_2, \ldots, PS_m), ?),\\
&((GS_1, GS_2, \ldots, GS_n, PS_1, \boldsymbol{PS_2} \cup \boldsymbol{p_i}, \ldots, PS_m), ?),\\
&\cdots,\\
&((GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, \boldsymbol{PS_m} \cup \boldsymbol{p_i}), ?)).
\end{aligned}
$$

Where the branch with $((\ldots, PS_{id} \cup p_i, \ldots), ?)$ will be a valid branch included in the relation if and only $PS_{id} \cup p_i$ does not violate any hard constraints.

In summary, this div defines 3 important things:

1. Only allows one game or practice to be selected.

   Reason: This constrains the tree so that we can try and limit the search space.

2. Does not allow branches that violate hard constraints.

Reason: Exploring branches that violate hard constraints will never lead to ones that do not, so we can just remove the possibility of the process selecting these sub trees.

3. Does not allow games or practices to be selected that have already been selected.

   Reason: A game or practice can only be assigned to one slot.

Process

The process is defined as follows

$$P_\wedge(A_\wedge, Env, K_\wedge)$$

To complete the process we must first define f_{leaf} and f_{trans}

f_{leaf}

Defines what leaves will be chosen to be explored first. Since this is an and-tree will need to explore the whole tree at some point but what we set up here will help f_{bound} with pruning.

$$f_{leaf} = \begin{cases} 0 & \text{if pr is the leaf node with the largest depth.} \\ 1 & \text{otherwise.} \end{cases}$$

Since there will always be at least one node at the largest depth we only need to consider ties when there are multiple $f_{leaf} = 0$. To break these ties we will prioritize leaves in the order,

1. Choose the leaf such that $Eval(assign)$ is minimized.
2. Choose the left-most leaf.

We have defined f_{leaf} to be depth first and using the soft constraint evaluations to break ties between similar depth leaves for the following reasons:

- Exploring depth first means we go straight for solutions which may allow f_{bound} to find sub-trees higher up from the solutions to prune.
- Breaking ties based on the soft constraint evaluations means that we will explore the most promising branches first.
- Defaulting to the left-most leaf when equivalent evaluations based on soft constraints gives an unambiguous execution.

f_{trans}

The goal is to pick what game or practice will be used for the next sub-tree division. It will also define when pr is a 'yes' or 'no' solution. The general philosophy will be to pick the most constraining divisions first so that they can be dealt with earlier in the tree, reducing amount of branching required.

We will define the value of f_{trans} to be assigned in the following order.

1. $f_{trans}(pr, ?) = (pr, \text{yes})$
   if $GS_1 \cup GS_2 \cup \cdots \cup GS_n \cup PS_1 \cup PS_2 \cup \cdots \cup PS_m = Games \cup Practices$.
   - This means all games and practices have been assigned to a slot so that the hard constraints have been fulfilled.
2. $f_{trans}(pr, ?) = (pr, \text{no})$
   if Div does not allow any more branches and
   $GS_1 \cup GS_2 \cup \cdots \cup GS_n \cup PS_1 \cup PS_2 \cup \cdots \cup PS_m \neq Games \cup Practices$.
   - This means there are more games or practices to be assigned but none of the assignments will fit the hard constraints.
3. $f_{trans}(pr, ?) = (pr', ?)$ where pr' picks a game or practice as required by div. We will prioritize options that are the most constraining so that games or practices will be chosen with the following priority
   - 3.1. g_i will be selected if $partassign(g_i)$ is defined.
     - Break ties by taking the smallest i
   - 3.2. p_i will be selected if $partassign(p_i)$ is defined.
     - Break ties by taking the smallest i
   - 3.3. g_i will be selected if an associated practice $p_{ik_i} \in PS_1 \cup PS_2 \cup \cdots \cup PS_m$ (p_{ik_i} has been assigned to a slot)
     - Break ties by taking the smallest i
   - 3.4. p_{ik_i} will be selected if an associated game $g_i \in GS_1 \cup GS_2 \cup \cdots \cup GS_n$ (g_i has been assigned to a slot)
     - Break ties by taking the smallest i, then smallest k_i
   - 3.5. g_i will be selected if $notcompatible(g_i, b)$ or $notcompatible(a, g_i)$ is defined.
     - Break ties by taking the smallest i
   - 3.6. p_i will be selected if $notcompatible(p_i, b)$ or $notcompatible(a, p_i)$ is defined.
     - Break ties by taking the smallest i
   - 3.7. g_i will be selected if $unwanted(g_i, s)$ is defined.
     - Break ties by taking the smallest i

3.8. p_i will be selected if $unwanted(p_i, s)$ is defined.

    * Break ties by taking the smallest i

3.9. g_i with the smallest i

3.10. p_i with the smallest i

This scheme prioritizes finding a yes or no solution (1 or 2), then prioritizes a game or practices based off what is likely to be constraining. Whenever a game or practice is assigned it will also try assigning associated games (3.3) or associated practices (3.4). It will then default to taking an arbitrary game (3.9) and if there are practices with no associated games, an arbitrary practice (3.10).

Instance

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

If the given problem to solve is pr, then we have

- $s_0 =$ (pr, ?) where s_0 is an empty tuple of size = $|Slots|$
- $G_\wedge(s) =$ yes, if and only if
    1. $s = (pr', \text{yes})$ or
    2. $s = (pr', ?, b_1, \ldots, b_n)$, $G_\wedge(b_1) = \cdots = G_\wedge(b_n) =$ yes and the solutions to $b_1, \ldots, b_n$ are compatible with each other or
    3. there is no transition that has not been tried out already

Application Example

The project problem description outlines these available slots:

- M: 13 slots (Game or Practice)
- T: 8 slots (Game only), 13 slots (Practice Only)
- W: 13 slots (Game or Practice)
- R: 8 slots (Game only), 13 slots (Practice only)
- F: 13 slots (Game only), 7 slots (Practice only)

To simplify for our example application, we are going to use the following slots:

- M: 1 slots (Game or Practice)
- R: 1 slots (Game only), 1 slots (Practice only)

For a total of 3 slots across 2 days (Monday and Thursday). This would be represented by the tuple

$$(GS_{1M}, PS_{1M}, \\ GS_{1R}, PS_{1R})$$

And as input we will use the following set for Games and Practices:

$$\begin{aligned} \text{Games} = \quad & \{\text{CMSA U13T3 DIV 01}, \\ & \text{CUSA U13 DIV 01}\} \end{aligned}$$

$$\begin{aligned} \text{Practices} = \quad & \{\text{CMSA U13T3 DIV 01 PRC 01}, \\ & \text{CMSA U18T1 DIV 01 PRC 01}\} \end{aligned}$$

We will also outline the following values for the hard constraints:

- All slots will have a $gamemax(s) = 3$ and $practicemax(s) = 2$
- We will give one $notcompatible(a, b)$ statement as $notcompatible(\text{CMSA U13T3 DIV 01}, \text{CUSA U13 DIV 01})$
- We will give one $partassign(a) = s$ statement as $partassign(\text{CMSA U13T3 DIV 01 PRC 01}) = PS_{1R}$
- We will give one $unwanted(a, s)$ statement as $unwanted(\text{CMSA U13T3 DIV 01}, GS_{1M})$

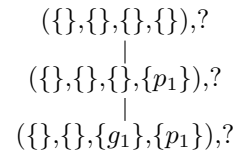
Finally, we will use the following value for a soft constraint, which will help us at the end of the search:

- For all slots s, $gamemin(s) = 0$ and $practicemin(s) = 0$, except for slot PS_{1R} will have $practicemin(PS_{1R}) = 2$

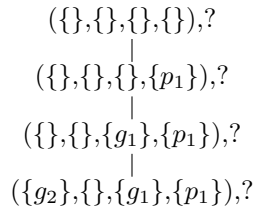
Now we will step through the iterations of the search as we try and find a solution. we start with an empty tree at the root with no Games or Practices assigned, and after Div gives us our first new branches, the tree is as such

$$\begin{array}{c} (\{\},\{\},\{\},\{\}),? \\ \Big| \\ (\{\},\{\},\{\},\{p_1\}),? \end{array}$$

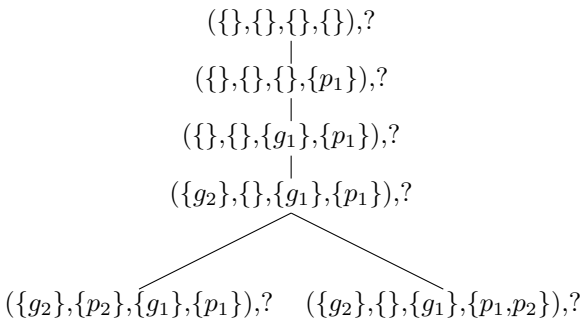
We choose to assign p_1 first because of f_{trans} 3.2. Because of our hard constraints, assigning p_1 to PS_{1M} violates $partassign$. Therefore, the would be second branch is invalid, therefore div will not return this branch. As well, we will select to expand the single branch.



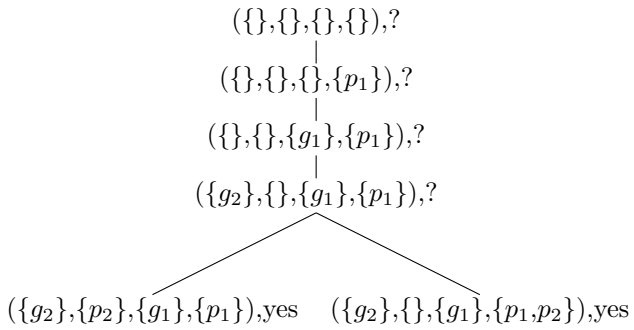
We next choose to assign g_1 because of f_{trans} 3.3. After this iteration, our tree will look as such because assigning g_1 to GS_{1M} violates the hard constraint *unwanted* and therefore is an invalid branch that is not returned by Div.



We next choose to assign g_2 because of f_{trans} 3.5. After this iteration, our tree will look as such because assigning g_2 to GS_{1R} violates the hard constraint *notcompatible* and therefore is an invalid branch that is not returned by Div.



We next choose to assign g_2 because of f_{trans} 3.4. Finally, our div will sol both bottom leaves nodes to yes



After this, we have exhausted all possibilities for Game and Practice assignments, so we evaluate and compare all leaves that sol yes, and we will choose the one that minimizes our soft constraint values. In this situation, the left right has a Eval-value of 0, but the left leaf has an Eval-value of $1 \times pen_{practicemin}$ as PS_{1M} has only 1 practice assigned to it. Therefore, choosing the left leaf we are given a solution of:

$$GS_{1M} = \{g_2\}$$
$$PS_{1M} = \{\}$$
$$GS_{1R} = \{g_1\}$$
$$PS_{1R} = \{p_1, p_2\}$$

Selection Reasoning

When it comes to finding the most optimal solution and-tree is the best search model to use. Comparing And-tree with Or-tree for this type of problem, it's easy to see that or-tree is not sufficient for this problem since we need to find an optimal solution to the problem. Or-tree may be more efficient, however; if the solution found is not optimal there is no point in using it for this problem.

And-tree and set-based search are optimal search models for this problem, however; an optimized and-tree can be much more efficient and find the most optimal solution. This is much better than set-based search which can only find an optimal solution but not the most optimal solution.

Set-Based Search

We chose a Set-Based search as our second search paradigm for the given problem. In the following we will define the Model with F and Ext, the Process and the Instance.

Model

Our search model is as follows

$$A_{Set} = (S_{Set}, T_{Set})$$

We must define F and Ext, and the following are given

$$S_{Set} \subseteq 2^F$$
$$T_{Set} \subseteq S_{Set} \times S_{Set}$$
$$T_{Set} = \{(s, s') | \exists A \to B \in Ext \text{ with } A \subseteq s \text{ and } s' = (s - A) \cup B\}$$

F

A fact (F) can be defined by a tuple of sets representing assignments to the available slots

$$(GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m)$$

where GS_i represents a set of games assigned to the first slot, and similarly PS_j is for assigned practices. n and m are integers such that $0 \leq n \leq |Games|$ and $0 \leq m \leq |Practices|$. If $n = 0$ or $m = 0$, then no slots of their types are available.

Where $GS_i \subseteq$ Games.
And $PS_j \subseteq$ Practices.

This will ensure independence of all Game and Practice slots.

The notation will allow the subscript i or j to not always be numerical, making it easier to name certain slots that have more importance (for example GS_{1M} refers to the first game slot on a Monday) but they will map back to the slot of the same type in the tuple where the first is 1, second is 2, until we have counted up to n or m.

The order of the slots in the tuple will also represent their time in the week relative to other slots that contain the same type (game or practice). However the relative ordering of slots of different types do not carry any meaning.

Each element in GS_i or PS_j represents an assignment of a game or practice to slot i or j assigned by the function $Assign : G \cup P \to S$, where G is the set

of games, P is the set of practices, and S represents the slots. The assignment is valid if it fulfills all hard constraints. A fact is valid if and only if Constr, a function that returns true if every assignment in F meets all hard constraints (outlined in Project Problem).

$$\text{Constr(Assign)} = \begin{cases} \text{true} & \text{if all hard constraints are fulfilled} \\ \text{false} & \text{otherwise.} \end{cases}$$

Ext

The extensions (Ext) allow the creation of new facts, which will be partial solutions to the problem.

$$\text{Ext} \subseteq \{A \to B \mid A, B \subseteq F \text{ and } \text{Add}(A, B) \text{ or } \text{Swap}(A, B)\}$$

Add (A,B)

- Let A be the tuple of all slots before the addition, representing the current assignments:

  $A = (GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m)$ where $s_i \in \text{Games} \cup \text{Practices}$ for all i.

- Let b be a game or practice that will be assigned to one of the slots in A, such that resulting tuple B does not violate hard constraints. The resulting tuple B after applying the add operation will be:

  $$\begin{aligned} B = (&GS'_1, GS'_2, \ldots, GS'_n, PS'_1, PS'_2, \ldots, PS'_m) \\ &\text{where } GS'_i = GS_i \cup \{b\} \text{ for some slot } GS_i \text{ in } A, \\ &\text{or } PS'_i = PS_i \cup \{b\} \text{ for some slot } PS_i \text{ in } A. \end{aligned}$$

Swap (A,B)

- Let A be the tuple before the swapping, representing the current assignments, such that:

  $A = (GS_1, GS_2, \ldots, GS_n, PS_1, PS_2, \ldots, PS_m)$ where $s_i \in \text{Games} \cup \text{Practices}$ for all i.

- Let b be a game or practice that will be assigned to one of the slots in A, such that the resulting tuple B does not violate the hard constraints. A swap will be made if the value of the evaluation function is minimized (i.e. there is a more optimal solution when that meets more soft constraints). The resulting tuple B after applying the swap operation will be:

  $$\begin{aligned} B = (&GS'_1, GS'_2, \ldots, GS'_n, PS'_1, PS'_2, \ldots, PS'_m) \\ &\text{where } GS'_i = GS_j \text{ and } GS'_j = GS_i \\ &\text{for some slot } GS_i, GS_j \text{ in } A \text{ and } \text{Eval}(B) < \text{Eval}(A), \\ &\text{or } PS'_i = PS_j \text{ and } PS'_j = PS_i \\ &\text{for some slot } PS_i, PS_j \text{ in } A \text{ and } \text{Eval}(B) < \text{Eval}(A). \end{aligned}$$

Process

The process is defined as follows

$$P_{Set} = (A_{Set}, Env, K_{Set})$$
$$K_{Set} : S_{Set} \times Env \to S_{Set}$$
$$K_{Set}(s, e) = (s - A) \cup B$$

To create the search process we must first define f_{wert} and f_{select} to get S_{Set} and T_{Set} by process definition. A_{set} is the search model previously defined. The *Env* is simply the environment of the process.

f_{wert}

We must define f_{wert}, a function that assigns a value to the Ext rules available. We want to perform an Add if possible, to add a game or practice into a slot. Otherwise, we will choose to perform a Swap.

$$f_{\text{wert}}(\text{A,B},Env) = \begin{cases} 0 & \text{if the Ext transition is function Add (A,B)} \\ 1 & \text{f the Ext transition is function Swap (A,B)} \end{cases}$$

f_{select}

The f_{select} will be used as a tiebreaker when comparing Add and Swap extension rules.

$$f_{\text{select}}(\{A' \to B'\}, Env) = A \to B$$

where $A' \to B'$ is the set of possible Ext with equal f_{wert} values, and $A \to B$ is the selected extension. We will tiebreak Add and Swap as follows:

Add - We will prioritize options that are the most constraining so that games or practices will be chosen with the following priority

1. g_i will be selected if $partassign(g_i)$ is defined.
   - Break ties by taking the smallest i
2. p_i will be selected if $partassign(p_i)$ is defined.
   - Break ties by taking the smallest i
3. g_i will be selected if an associated practice $p_{ik_i} \in PS_1 \cup PS_2 \cup \cdots \cup PS_m$ (p_{ik_i} has been assigned to a slot)
   - Break ties by taking the smallest i
4. p_{ik_i} will be selected if an associated game $g_i \in GS_1 \cup GS_2 \cup \cdots \cup GS_n$ (g_i has been assigned to a slot)

   - Break ties by taking the smallest i, then smallest k_i

5. g_i will be selected if $notcompatible(g_i, b)$ or $notcompatible(a, g_i)$ is defined.
   - Break ties by taking the smallest i
6. p_i will be selected if $notcompatible(p_i, b)$ or $notcompatible(a, p_i)$ is defined.
   - Break ties by taking the smallest i
7. g_i will be selected if $unwanted(g_i, s)$ is defined.
   - Break ties by taking the smallest i
8. p_i will be selected if $unwanted(p_i, s)$ is defined.
   - Break ties by taking the smallest i
9. g_i with the smallest i
10. p_i with the smallest i

Swap - We will prioritize swaps that are produce the Ext $A \rightarrow B$ which minimizes the number of soft constraints violated.

1. Swap (A,B) producing the lowest Eval(assign) will be selected
2. If all Swap(A,B) have the same soft constraint penalty, the swap will be randomly selected.

Instance

$$Ins_{set}(s_0, G_{set})$$

- $s_0, s_{goal} \in 2^F$
  - where s_0 is an empty tuple of size = $|Slots|$
  - where s_{goal} is after all hard constraints are satisfied, and swapping has begun. And one of the following happens:
    1. Swapping creates a state with zero soft constraint penalties
    2. After $2 \times |Slots|$ iterations
- $G_{set} : S \rightarrow \{\text{yes}, \text{no}\}$
- $G_{set}(s_i) =$ yes if and only if $s_{goal} \subseteq s_i$ or there is no extension rule applicable in s_i

Application Example

We will be using all of the example values used in the And-Tree Application Example. We can step through the iterations of the search as follows, as we start with the empty tuple with no assignments

$$(\{\}, \{\}, \{\}, \{\})$$

The next iteration will choose to perform an Add because Add is the lowest value in our hierarchy and will always be picked before a Swap if possible. We want to assign p_1 with the add because f_{select} 3.2. Because assigning p_1 to PS_{1M} violates *partassign* hard constraint, it is not a possible Ext and we choose to assign p_1 to PS_{1R}

$$(\{\}, \{\}, \{\}, \{p_1\})$$

Next, we will choose to perform an Add with g_1 because f_{select} 3.3. Because assigning g_1 to GS_{1M} violates *unwanted* hard constraint, it is not a possible Ext and we choose to assign g_1 to GS_{1R}

$$(\{\}, \{\}, \{g_1\}, \{p_1\})$$

Next, we will choose to perform an Add with g_2 because f_{select} 3.5. Because assigning g_2 to GS_1R violates *notcompatible* hard constraint, it is not a possible Ext and we choose to assign g_2 to GS_{1M}

$$(\{g_2\}, \{\}, \{g_1\}, \{p_1\})$$

Next, we will choose to perform an Add with p_2 because f_{select} 3.4. Because there are no hard constraint violations with adding p_2 to either PS_{1M} or PS_{1R}, we will randomly choose to assign p_2 to PS_{1M}

$$(\{g_2\}, \{p_2\}, \{g_1\}, \{p_1\})$$

Now, we have assigned all Games and Practices to slots. We will next choose to perform a Swap because we are still violating soft constraints because PS_{1R} has a penalty of $1 \times pen_{practicemin}$. We choose to perform a swap of g_2 from PS_{1M} to PS_{1R}.

$$(\{g_2\}, \{\}, \{g_1\}, \{p_1, p_2\})$$

Finally, because we are still not violating any hard constraints, and we are not receiving any soft constraint penalties, we can terminate the search. Giving us

a solution of:

$$GS_{1M} = \{g_2\}$$
$$PS_{1M} = \{\}$$
$$GS_{1R} = \{g_1\}$$
$$PS_{1R} = \{p_1, p_2\}$$

Reasons For Model

When comparing set-based search to and-tree and or-tree, set-based search model is an effective way to optimize a solution in this problem.

When it comes to or-tree it is nearly impossible to find an optimal solution since it can only find one solution in the entire tree. Furthermore, if we try to optimize it to find an optimal solution we would need to make an A* or-tree or "hack" the properties of an or-tree which are both extremely tricky to implement. Therefore, we created the set-based model to simulate an or-tree with a fact which continually changes. In this model, we do not keep track of previous history and also do not stop when 1 solution is found. Through the Swap Ext rule, we aimed to our optimize solution.

Compared to and-tree the set-based model should be more efficient in finding a optimal solution, as the and-tree requires an exhaustive search. And-tree will give a optimal solution for the problem, however; it will take more time and memory for an exhaustive search to find this solution. We aimed to make our set-based search a middle ground between the two search models, as it can find an optimal solution unlike or-tree and be much more efficient than an and-tree.