# AI Project Paper

By Team Muk:
Daniel Tetreault (30129879), Jimmy Xu (30174110),
Kyle West (30167781), Guilherme Doherty Marques (30075951),
Kya Lee Broderick (30173946)

## Table of Contents

General Definitions ........ 2
Set-Based Search ........ 2
  Informal Description ........ 2
  Search Model ........ 3
  Search Process ........ 3
  Search Instance ........ 8
  Extension Rules ........ 3
  Or-tree Search Model ........ 9
    Model Description ........ 9
    Defining Altern Relationship ........ 10
    Defining the solved state ........ 11
    Search Control ........ 12
    Or-Tree Search Instance ........ 13
And-Tree-Based Search ........ 22
And-Tree-Based Search ........ 22
    Informal Description ........ 22
    Formal Description ........ 23
    Visualization ........ 25
    Steps ........ 26

## General Definitions

The following are definitions that are shared between our two search paradigms.

$G = Games = \{g_0, \dots, g_n\}$

$P = Practices = \{p_0, \dots, p_n\}$

$GP = Games\ and\ Practices = G \cup P$
$Sl = Slots$

$$Constr(f) = \begin{cases} \text{Fact meets all hard constraints} \wedge Complete(f) & True \\ \text{Otherwise} & False \end{cases}$$

$$PartConstr(f) = \begin{cases} \text{Fact meets all hard constraints*} & True \\ \text{Otherwise} & False \end{cases}$$

$$Complete(f) = \begin{cases} \forall\, sl \in f, sl \neq \$ & True \\ \text{Otherwise} & False \end{cases}$$

$PartEval(f), f \in F'$: Returns the sum of penalties from all violated soft constraints.

$AConstr(a)$: Evaluates whether an individual assignment is valid. This is false if a game is in a practice-only slot, or vice versa.

$Assign = \{(g, sl) | g \in GP, sl \in Sl, AConstr(a)\}$: Ordered pairs representing possible assignments from games or practices to their corresponding slots.

$assign = GP \rightarrow Sl$ such that $\forall g\big(g \in GP \rightarrow \big(g, assign(g)\big) \in Assign\big)$: Surjection mapping a game or practice to a slot. Must be in Assign, to meet constraints.
$partassign: GP \times (Sl \cup \{\$\})$: Initial assignment provided by input

$passign = assign \cup GP \rightarrow \$$: Modified version of assign which allows null assignments

## Set-Based Search

### Informal Description

Our set-based search model is an evolutionary algorithm. We use an Or-Tree-based search to find some initial valid solutions (to create our initial population), then run our evolutionary algorithm of both crossovers and mutations on the set of valid solutions provided to us from the or tree search, to produce new facts for our population. We continue this process until we reach a point where the number of facts we have in the population exceeds some cap, at which point we delete some facts in the population. Between the selective pressure applied from delete, the carry-on of “good” traits from crossover, and the genetic diversity applied from mutation, we can predict that after several generations (iterations of our extension functions) we will likely reach a population containing a fact that is close to the optimal solution to our problem. When the search is complete, we choose the fact in our population with the best evaluation to be considered our “solution”.

## Search Model

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

Let $F$ be the set of all facts. A fact $f$ is a complete, valid schedule. We will first define $F'$ which contains $f'$: both complete and incomplete facts, where there may be some games or practices not assigned to slots, making $F'$ the set of all incomplete and complete schedules. This will be used in our or-tree.

$$F' \equiv \{f' | f' = \{(gp, passign(gp)) | gp \in GP\}, PartConstr(f)\}$$

From $F'$ we will construct our set of facts, which must also satisfy $Constr(f)$, meaning they must be fully assigned. These are fully assigned schedules – complete facts.

$$F \equiv \{f | f \in F', Constr(f), f \in Assign\}$$

## Extension Rules

Our extension rules are the core of the genetic algorithm. They take our set of solutions (our population) and add a new fact. We have crossover, which takes two different facts from the population, and considers their assigned slots to games or practices to build up a new child fact, which is then added to the population. Mutation, which takes a fact and changes a single assignment in it, and adds this new mutated fact to the population, and finally, we have delete, which runs when our population gets too large. When this happens, the worst of the population gets removed.

$$Ext = \left\{A \to B \middle| A, B \in S \wedge \left(\left(Crossover(A, B) \vee Mutation(A, B)\right) \vee Delete(A, B)\right)\right\}$$

We will define a useful variable $a_n$ as the assignment

> $a_n = \left(gp_n, sl_n | \, sl_n = assign(gp_n)\right)$: the individual tuples of valid games/practices and their assigned slot.

## *Crossover(A,B)*

Crossover is an extension function which uses 2 facts (parents) from our current population, to create a new child fact, which is then added in to our population $s$. We use crossover to generate child facts that are made up of 2 fit parent facts, which will hopefully allow us to carry on the "good" traits of 2 parents into a new generation.

Reference our search control section for formal definitions on the contents of A, a set of 2 facts provided to *crossover(A,B)*.

Let $k = |GP|$. In our search control, $f_{select}$ will provide crossover with 2 victims $f_a$ and $f_b$, making

$$A = \{\mathrm{f_a}\,, f_b\}$$

$$A = \{\{a_{a1}, \dots, a_{ak}\}, \{a_{b1}, \dots, a_{bk}\}\}$$

$$A = \{\{(gp_{a1}, sl_{a1}), \dots, (gp_{ak}, sl_{ak})\}, \{(gp_{b1}, sl_{b1}), \dots, (gp_{bk}, sl_{bk})\}\}$$

We will be using an OR tree search to complete this crossover and create a child fact. The OR tree search will select first a random slot from either parent a or parent b. If the selection of the slot from the random parent violates our hard constraints, the OR tree will go on to select the slot from the other parent. If this too violates our hard constraints, the OR tree will select slots at random, until we reach a valid slot. For more information on this process, please reference section SECTION on the or tree. Additionally, note that OR tree will always return a valid slot, as both parents $f_a$ and $f_b$ are valid facts, and OR tree will simply return one of the parents as a child fact if there are no other valid possibilities to create a new fact under.

$OrTree(A, Crossover)$: This is our inputs into our or tree. See *ORTREE* for more details

This or tree will return:

$f_c = \{a_{c1}, \dots, a_{ck}\} = \{(gp_{c1}, sl_{c1}), \dots, (gp_{ck}, sl_{ck})\}$. The child fact (see 1.N OR TREE for more specific definitions on what $f_c$ is composed of, and how we make $f_c$).

As a result, we have

$$B = A \cup \{f_c\}$$

***Mutation(A,B)***

Mutation is an extension function which uses a single fact from our current population, to preform a single point mutation on one of its current slots, creating a new modified fact, which is then added in to our population.

We receive an A from $f_{select}$. Let $k = |GP|$

$$A = \{a_1, \dots, a_k\}$$

$$\mathrm{A} = \{(gp_1, sl_1), \dots, (gp_k, sl_k)\}$$

Let *r* be some random integer such that there exists some slot $sl_r \in a_r$ with $1 \leq r \leq k$. We take this slot $sl_r$ and alter it to $sl'_r = \$$, to update *A* to

$\mathrm{A}' = \{(gp_1, sl_1'), \dots, (gp_k, sl_k') \mid \forall sl'_i \in A', sl'_i = sl_i,$ except for a single $sl'_r \in A'$ where $1 \leq r \leq k,$ such that $sl'_r \neq sl_r,$ and $sl'_r = \$\}$.

Essentially, we convert the assignment of one slot in *A* to be null, as this will be the point of mutation.

We will be using an OR tree search to complete the mutation. The OR tree will take the empty slot $sl_r'$ and randomly search until we find a slot $sl \in Sl$ such that the slot can be filled into *A* at $a_r$, and will fit all hard constraints. We prioritize filling the slot $sl_r'$ with some slot $sl_i \in Sl$ such that $sl_i \neq sl_r$. However, if we find no possible slot that fits the constraints and is not $sl_r$, then we will be forced to return $A$ from the or tree as our mutated fact. Otherwise, we will receive a new fact $f_{new} \in F$.

$$Otree(\{A, A'\}, mutation)$$

This tree will return a new complete fact, giving us the following for *B*:

$$B = A \cup f_{new} = \{(gp_1, sl_1), \dots, (gp_k, sl_k)\} \cup \{(gp_1, sl_1''), \dots, (gp_k, sl_k'')\}$$

where $sl_i = sl''_i$ except for the $sl_r \in A$, where we have performed a single point mutation, such that $sl''_r \in Sl,$ but not nessecairly $sl_r = sl_r''$.

Essentially, this extension function returns Our original A from $f_{select}$, unioned with the output of the Or tree.

Note that if the output of the or tree is the same as the original *A,* we are simply just returning the input we received from $f_{select}$. Also, note that this process can be expanded to fit multi-point mutations – although it would require some further considerations into the relationships between slots (MWF, TH, F, for example).

### *Delete(A,B)*

Delete is an extension function which will remove a set of facts from our current population.

We receive *A*, a set of facts, from $f_{select}$(see search control for more specific details on the selected facts) such that

$$A = \{f_1, \dots f_n\}$$

We simply delete these facts, creating a set *B* such that

$$B = \{\}$$

This effectively removes these facts from our population.

## Search Process

Our search process defines how a search can take place. It takes our model, an environment, and a search control.

$$P_{set} = (A_{set}, Env, K_{set})$$

Our environment specifies some static properties, listed below, which are used by our search control.

Our search control is defined as follows:

$$K_{set}: S_{set} \times Env \rightarrow S_{set}$$

$$K_{set}(s, e) = (s - A) \cup B$$

$$Env = \{Cap, P_{mut}, c, g, lim\}$$

$Cap$: A pre-determined integer value for the capacity of complete facts $f$ in our current population

$P_{mut}$: The probability of choosing mutation where $0 \leq P_{mut} \leq 1$

*c:* A decimal representing the percentage $0 < c \leq 1$ of the facts with the lowest *partEval(f)* value in the current population that should be considered for crossover.

*g:* A decimal representing the percentage $0 < c < 1$ of the facts with the highest *partEval(f)* values, in our current population, that should be considered for delete. Note that a *g* value that is too small is unfavorable, as the smaller *g* is, the more often we will be running the delete function.

*lim:* Some integer $0 \leq lim < \infty$, representing how many iterations we run of our algorithm.

Our $f_{wert}$ sets some value to each extension function of crossover, mutation or delete. Note that $s \in S$ is our current population.

$$f_{wert}(Crossover) = f_{wert}(Mutation) = \begin{cases} |s| \leq Cap: & 0 \\ Otherwise: & 1 \end{cases}$$

$$f_{wert}(Delete) = 1 - f_{wert}(Crossover)$$

$f_{select}$ will select the extension rule containing the lowest value from $f_{wert}$

In the event of the tie between crossover and mutation, we break this using $f_{select}$.

Let $RNG$ be a random number generator.

$f_{select}$ uses a $RNG$ to generate a random number $r$ where $0 \leq r \leq 1$.

$f_{select}$ chooses $mutation(A, B)$ when $0 \leq r \leq P_{mut}$

$f_{select}$ chooses $crossover(A, B)$ when $P_{mut} < r \leq 1$

Note that there will never be a tie in values when $cap$ is surpassed, so we do not require an $f_{select}$ tiebreaker when $delete(A, B)$ is the chosen extension rule.

**Selecting Victims**

Let $r$ be some random integer, generated by $RNG$, such that $1 \leq r \leq |s|$. When selecting facts to be victims of crossover or mutation, $f_{select}$ chooses $f_r \in s$, where $s$ is the current state (our current population), making $f_r$ a random fact from our current population.

**For Mutation(A,B),** our victim is simply any random $f_r \in s$.

**For Crossover(A,B),** however, our victims are $f_{r1}$ and $f_{r2}$, where we first ensure that $f_{r1} \neq f_{r2}$ to reduce redundant crossovers, are selected according to their fitness evaluation. We consider only the best *k* facts, such that $k = \lceil c * |s| \rceil$. in the current population *s,* giving us

$f_{r1}, f_{r2} \in \{f_1, \dots, f_k\} \subset s$ where for any $f_j \notin \{f_1, \dots f_k\}$, and $\forall$i s.t. $1 \leq$ i $\leq$ k, $partEval(f_j) \geq partEval(f_i)$. Saying, any fact not in the set of facts $\{f_1, \dots, f_k\}$ has a higher *partEval* value that any fact in the current set of facts. Any fact not in the set of the facts $\{f_1, \dots, f_k\}$ will not be considered as victims for our crossover -- we want to crossover only the best *p%* of our current population.

However, if there are less than 2 distinct facts in the set from the best $p$% of the population, we multiply $p * 2$ to create a new value for $p$ and repeat the process. There is a risk our given *p* value could be small enough that there does not exist 2 facts within this percentage, and we need to account for that.

If for any reason our population is down to 1 fact, we will run $orTree(\{partAssign, partAssign\}, initiate)$ to generate a new fact, before proceeding with selecting victims for our crossover. Note that, due to the fact that $delete$ operates on $\lfloor g * |s| \rfloor$ facts, and $g < 1$, we will never be in a position where our population has no facts following an extension rule.

**For Delete(A,B)** we will do the opposite. We consider only the worst *h* facts, such that

$h$ =⌊g*|s|⌋. Here, we send in the crossover function a set of facts $\{f_h, \dots, f_{|s|}\}$ such that for any $f_j \notin \{f_h, \dots f_{|s|}\}$, and $\forall$h $\leq$ i $\leq$ |s|, $EVAL(f_j) \leq EVAL(f_i)$.

Saying, any fact not in the set of facts $\{f_h, \dots, f_{|s|}\}$ has a lower *partEval* value that any fact in the selected set of facts. Any fact not in the set of the facts $\{f_h, \dots, f_{|s|}\}$ will not be considered as victims for our delete -- we want to delete only the worst *g%* of our current population.

## Search Instance

To define our search instance, we will use $s_0$ for the starting state and $G_{set}$ for the goal conditions, then the search instance is defined as:

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

The starting state $s_0 \in S_{set}$ is an assignment with the hard constraints satisfied. We will use an Or-Tree based search to get these initial states, which is defined below.

The goal $s_{goal} \in S_{set}$ is any assignment of games and practices that meet all hard constraints and maximizes the soft constraints. The goal condition is defined by:

$$G_{set} : S \rightarrow \{yes, no\}$$

Where $G_{set}(s_i) = yes$ if and only if $s_{goal} \subseteq s_i$ where $s_{goal}$ has an $Eval$ of zero, or the amount of times we run our algorithm reaches a set limit $lim \in Env$.

Otherwise, $G_{set}(s_i) = no$.

## Or-tree Search Model

We use an or tree to return a single valid solution. This will be used to develop the initial population for our set based search, to generate a child for crossover from 2 parents, and to create a new mutated fact from a fact with a single empty slot assignment. Our or tree search model will use specific search methods to create each of these 3 outputs, and will return the leaf containing the complete fact fitting all hard constraints, or will run until it has visited every possible node, and determined that there does not fit any valid facts within our constraints.

### Model Description

We first need to define our model $A_\mathrm{V} = (S_\mathrm{V}, T_\mathrm{V})$. Having the given definitions for $S_\mathrm{V}$ and $T_\mathrm{V}$, we only need to define $Prob$ to complete our definition of A. We define $Prob$ as $F'$.

Now, with our definition of *Prob*, we can defined $S_\mathrm{V}$ as

$$S_\mathrm{V} \subseteq \boldsymbol{Otree} \text{ where } \boldsymbol{Otree} \text{ is defined recursively by}$$

$$(pr, sol) \in \boldsymbol{Otree} \text{ for}$$

$$pr \in Prob, sol \in \{yes, ?, no\}$$

$$(pr, sol, b_1, \dots, b_n) \in \boldsymbol{Otree} \text{ for}$$

$$pr \in Prob, sol \in \{yes, ?, no\}, b_i \in \boldsymbol{Otree}$$

And we can define $T_\mathrm{V}$ as

$$T_\mathrm{V} \subseteq S_\mathrm{V} \times S_\mathrm{V}$$

$$T_\mathrm{V} = \{(s_1, s_2) | s_1, s_2 \in S_\mathrm{V} \text{ and } Erw_\mathrm{V}(s_1, s_2)\}$$

Where $Erw_\mathrm{V}$ is a relation on $\boldsymbol{Otree}$ defined by

$$Erw_\mathrm{V}((pr, ?), (pr, yes))$$

$$Erw_\mathrm{V}((pr, ?), (pr, no))$$

$$Erw_{\vee}\left((pr,?),\left(pr,?,(pr_1,?)\ldots,(pr_n,?)\right)\right) \text{ if Altern}(pr,pr_1,\ldots,pr_n) \text{ holds}$$

$$Erw_{\vee}\left((pr,?,b_1,\ldots,b_n),(pr,?,b'_1,\ldots,b'_n)\right)\text{if for an } i\text{: } Erw_{\vee}(b_i,b'_i) \text{ and } b_j = b'_j \text{ for } i \neq j$$

## Defining Altern Relationship

By the definition of *Prob,* we know that

$$Altern \subseteq Prob^{+}$$

Altern is a function that takes a partial schedule prob and generates a new prob with one more assignment. It does this by selecting some arbitrary game or practice that is not assigned, removes that ordered pair, and replaces it with that same game or practice assigned to a slot.

$Altern(prob) = (prob - \{(gp,\$)\}) \cup \{(gp, assign(gp)\}$ where $gp \in GP$

### The environment

We define the environment as

$$Env = (w_{set}, type)$$

**Where $w_{set}$ is a set of 2 facts such that**

$$w_{set} = \{f_1, f_2 | \, f_1, f_2 \in F'\}$$

The specific contents of $f_1$ and $f_2$ depend on the $type$ variable. Let $k = |GP|$

1. If $type = crossover$ then:

We receive 2 facts from $w_{set}$—parent *a* and parent *b*, which will be used to help us develop our child fact

$$f_1 = f_a = \{a_{a1},\ldots,a_{ak}\} = \{(g_{a1}, sl_{a1}),\ldots(gl_{ak}, sl_{ak})| \, \forall sl_i \in a_i, sl_i = assign(g_{ai})\}$$

$$f_2 = f_b = \{a_{b1},\ldots,a_{bk}\} = \{(g_{b1}, sl_{b1}),\ldots(gl_{bk}, sl_{bk})| \, \forall sl_i \in b_i, sl_i = assign(g_{bi})\}$$

2. and by our victim definitions in $f_{select}$,

$$f_a \neq f_b$$

If $type = mutation$ then:

If we have a mutation, then $f_1$ is going to be a complete fact $f_{og}$, wherein $f_{stripped}$ is simply $f_{og}$, with a random one of the slots in $f_{og}$ associated to a game set to null.

$$f_1 = f_{og} = \{a_1, \dots, a_k\} = \{(g_1, sl_1), \dots (gl_k, sl_k) | \forall sl_i \in a_i, sl_i = assign(g_i)\}$$

$$f_2 = f_{stripped} = \{a_1, \dots, a_k\} = \{(g_1, sl_1'), \dots (gl_k, sl_k') | \forall sl_i' \in f_2, \forall sl_i \in f_1, sl_i' = sl_i, \text{except for one random } sl_r \in a_r, \text{where } sl_r = \$\}$$

3. If $type = initiate$

In *initiate*, we do not require any other input facts than *partAssign,* meaning the fact $f_2$ is redundant.

$$f_1 = partAssign$$
$$f_2 = partAssign$$

**And *type* is the type of or tree we will be implementing**

$type$ = one of $crossover, mutation\ or\ initialize$

## Defining the solved state

We now need to define when a *pr* in our tree is solved to a *yes* (we have found a valid complete fact), a *no* (we have violated some hard constraints), or a *?* (when we have yet to violate hard constraints, and we have not reached a complete fact). These are defined as the following:

$$sol = \boldsymbol{yes} \text{ when } partialConstr(pr) \wedge Complete(pr)$$

Our node is labeled as "*yes*" when we reach a complete solution that passes all of the hard constraints

$$sol = \boldsymbol{no} \text{ when } \neg partialConstr(pr)$$

Our node is labeled as "no" when we violate any hard constraints, regardless of if our branch is complete or not

$$sol = ? \text{ when } partialConstr(pr) \wedge \neg Complete(pr)$$

Our node is labeled as "?" when we have not yet violated any partial constraints, however we have not reached a full solution

## Search Control

Now that we have defined the search model $A_{\text{V}}$, we can define the search process,

$$P_{\text{V}} = (A_{\text{V}}, Env, k_v).$$

The search control $K_{\text{V}}$ chooses which leaf to transition to using the following evaluation functions:

**Mutation $\boldsymbol{f_{eval}}$**

When *type=mutation*, we receive a set of facts $f_{og}$ and $f_{stripped}$ (reference the environment in or tree). Let $k = |GP|$, and let $sl_n$be some slot associated with a game or practice $gp_n$ such that $sl_n = assign(gp_n)$. To restate what was already defined in our environment

$$f_{og} = \{a_1, \dots, a_k\}$$

$$f_{og} = \{(gp_1, sl_1), \dots, (gp_k, sl_k)\}$$

$f_{stripped} = \{(gp_1, sl_1{}'), \dots, (gp_k, sl_k{}') | \exists$ a single random $sl_r{}' \in f_{stripped}$ where $1 \leq r \leq k, \boldsymbol{s.t}\ sl_r = \$\}$.

We use these definitions to represent $f_{mutEval}$

$$f_{mutEval}(pr) = \begin{cases} 0 & if\ for\ sl_r' \in pr, sl_r \in f_{og}, sl_r{}' \neq sl_r \\ 1 & otherwise \end{cases}$$

Essentially, if our empty slot $sl_r' \in f_{stripped}$ is replaced with the slot $sl_r \in f_{og}$, then we give the associated *pr* a *partEval* value of 1. We would rather explore a *pr* with a slot that is different that the one associated with original fact that mutation is provided with from $f_{select}$, to help us increase genetic diversity.

**Crossover $\boldsymbol{f_{eval}}$**

When *type=crossover*, we receive a set of facts $\{f_a, f_b\}$—the two parent facts. Let $k = |GP|$, and let $sl_n$ be the empty slot in $pr$ of which we are currently assigning a slot $sl \in Sl$ to, and let $sl_{an} \in f_a$ be the corresponding slot in parent *a*, and let $sl_{bn} \in f_b$ be that of parent *b.*

$$f_{crossEval}(pr) = \begin{cases} 0\ if\ for\ sl_n \in pr, sl_n = sl_{an}\ or\ sl_n = sl_{bn} \\ 1 \quad otherwise \end{cases}$$

Essentially, we are favoring nodes that contain a slot from either parent 1 of parent 2, above slots with random entries.

**$\boldsymbol{f_{leafExt}(s)}$ Crossover and mutation**

When $type = crossover$ or $type = mutation$ we proceed with this $\boldsymbol{f_{leafExt}}$.

When $type = crossover,$ we use $f_{crossEval}$ to calculate our $leafEval$ value, such that $f_{crossEval}(pr) = leafEval.$

When $type = mutation,$ we use $f_{mutEval}$ to calculate our $leafEval$ value, such that $f_{mutEval}(pr) = leafEval.$

$f_{leafExt}$ will return us the node in our state s with the lowest $leafEval$ value. In the case of a tie, it will choose the deepest of the tied nodes. In the case of another tie between these, it will select one of the tied nodes randomly.

**$\boldsymbol{f_{leafInit}(s)}$ initialize**

When $type = initialize,$ we proceed with this $\boldsymbol{f_{leafInit}}$.

This will simply choose the current deepest node in our state s to explore. In the case of a tie between nodes, it will select one of the tied nodes randomly.

$\boldsymbol{f_{trans}}\ ((\boldsymbol{pr}, ?))$ selects the only $Altern$ of pr that holds.

## Or-Tree Search Instance

The $s_0$ of our or tree is the starting state. This will take in a fact we receive from $w_{set}$ to help us initialize our search.

**Mutation $\boldsymbol{s_0}$**

$$s_0 = (f_{stripped}, ?)$$

In our mutation search instance, our starting state $s_0$ is the fact $f_{stripped}$ that the or tree received from the mutation extension function. $f_{stripped}$ is simply a fact with a single null slot, of which our or tree will assign a slot $sl \in Sl$ to.

**Crossover $\boldsymbol{s_0}$**

In our crossover search instance, our starting state $s_0$ is an empty fact $f_{empty}$, as we will build up the empty fact one by one to create our child fact to return to *crossover.*

Let $k = |GP|$ Let $f_{empty} \in F'$ be a fact such that

$f_{empty} = \{a_1, \dots a_k\} = \{(gp_1, sl_1), \dots, (gp_k, sl_k) | \forall s_i \in f_{empty}, s_i = \$\}$ .

It is a fact containing games and practices, all of which are all mapped to a null slot.

$$s_0 = (f_{empty}, ?)$$

**Initialize $s_0$**

In our initialize search instance, our starting state $s_0$ is the fact *PartAssign*, the provided partial assignment of games and practices to slots, with the remaining unassigned games or practices being mapped to null.

$$s_0 = (PartAssign, ?)$$

**Goal**

Note that for all $s_0$ we have defined above, $s_0 = (pr, ?)$

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

1. $s = (pr', yes)$ or
2. $s = (pr', ?, b_1, \dots, b_n), G_\wedge(b_1) = \cdots = G_\wedge(b_n) = yes$ or
3. All leaves of our tree have either the sol-entry $no$ or cannot be processed using $Altern$

We stop our or tree search when we reach either of these conditions.

Now that we have defined $P_\vee$, $A_\vee$ and $Ins_\vee$, we have defined everything needed for our or tree model.

Now that we have defined $P_{\text{V}}$, $A_{\text{V}}$ and $Ins_{\text{V}}$, we have defined everything needed for our or tree model.

## Visualization and walkthrough

For this example, lets first define all initial variables of our GA. In our environment

$$Env = \{Cap, P_{mut}, c, g, lim\}$$

Suppose the following:

$Cap$ = 3

$P_{mut}$ = 0.20

*c:* 0.6

*g:* 0.40

*lim*: 3

Suppose, in our example, $GP = \{gp_1, gp_2, gp_3\}$, and $Sl = \{sl_1, sl_2, sl_3\}$.

Suppose we receive some *partassign* such that

$partassign = \{(gp_1, sl_1), (gp_2, \$), (gp_3, \$)\}$.

1. **Creating initial population of size *CAP***

Given partassign is simply some initial configuration of games to slots provided to us, we can now begin to create our $population_0$ using partassign as a guide for our or tree. To do so, we call

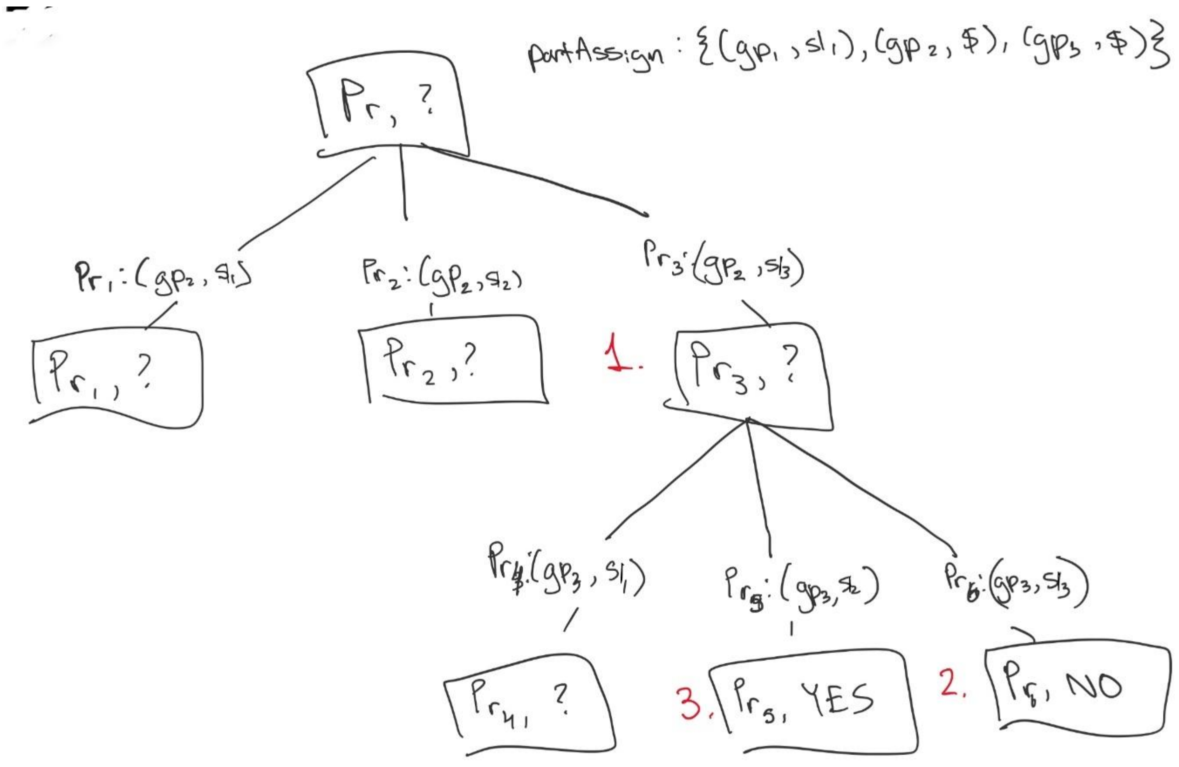
$$orTree(\{partAssign, partAssign\}, initialize)$$

We insert 2 partassigns, knowing the second one will be redundant. Refer to (ORTREE) for further details on the parameters accepted.

Having fed all relevant information to the Or Tree, we begin the or tree process.

**Please note: although our or tree search selects a random $gp_i \in Pr$ to assign a slot to, for the sake of spacing and ease, our diagrams assign a slot to an empty $gp_i$ in numerical order. It is important to know that this may not always be the case, as the decision as far as which game or practice to assign to a slot is made randomly in our tree.**

**Also note: at each definition in the diagram, where $pr_i$: $(gp_n, sl_j)$, we omit the redundant associated $a_n$ components in the set in *pr* for spacing purposes.**



Step 1:

We start with a single leaf node $(pr,\ ?\ )$, which gets expanded into its alternatives $pr_1, pr_2$ and $pr_3$, where $gp_1$ get assigned to $sl_1, sl_2$ and $sl_3$ respectively.

Step 2:

We select one of the possible three nodes randomly. Our selection, in this example, is $pr_3$, which assigns, $sl_3$ to $gp_2$.

Step 3:

$pr_4, pr_5$ and $pr_6$ are the deepest leaves that we can now explore. We select one randomly. Our selection, in this example, is $pr_6$, which assigns slot $sl_3$ to $gp_3$. However, in this example, the fact $f_i = \{(gp_1, sl_1), (gp_2, sl_3), (gp_3, sl_3)\}$ violates a hard constraint. As a result, we set the variable *sol* to *no.*

Step 4:

$pr_4, pr_5$ are now the deepest unassigned leaves that we can now explore. Our next random selection is $pr_5$, which assigns slot $sl_2$ to $gp_3$. This assignment creates the fact $f_i = \{(gp_1, sl_1), (gp_2, sl_3), (gp_3, sl_2)\}$ which, in our example, passes all hard constraints, and is complete. As a result, we can set *sol = yes,* and we are done our search.

We repeat the initialize function $Cap$ times, to create a population of size $Cap$ (in this example, to create a population of size 3).

Let this population be:

$$population_0 = \{f_1, f_2, f_3\}$$

$$population_0 = \{$$

$$\{(gp_1, s_1), (gp_2, s_2), (gp_3, s_3)\},$$

$$\{(gp_1, s_2), (gp_2, s_1), (gp_3, s_3)\}$$

$$\{(gp_1, s_1), (gp_2, sl_3), (gp_3, sl_2)\}\}$$

2. **Choosing an extension function**

Immediately after our initialization of our initial population, we have $cap$ (in this example, 3) chromosomes (facts) in our population. As a result, $f_{wert}(crossover) = f_{wert}(mutation) = 0$, and $f_{wert}(delete) = 1 - f_{wert}(crossover) = 1$.

Given $f_{wert}(crossover) = f_{wert}(mutate), f_{select}$ is tasked with breaking the tie between the two extension rules. To do so, we generate some random number $0 \leq r \leq 1$. Suppose $r = 0.6$

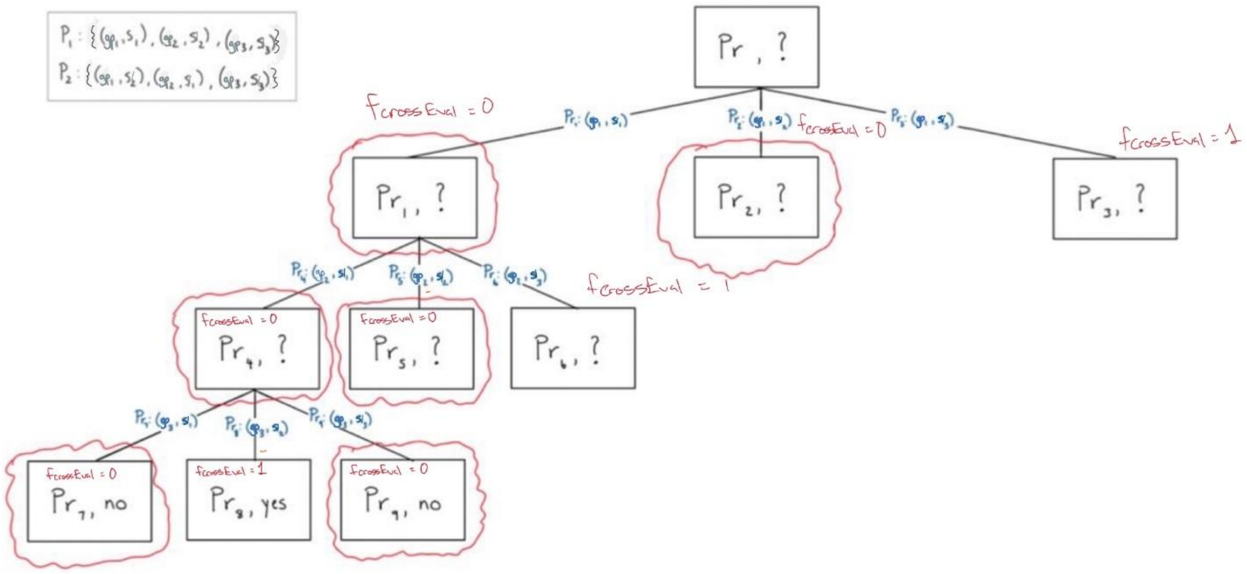
According to our definition of $f_{select}$, and its relationship to $P_{mut} \in Env$, we select $mutate(A, B)$ when $0 \leq r \leq 0.2$, and we select $crossover(A, B)$ when $0.2 < r \leq 1$. Given $r = 0.6$, we are selecting $crossover(A, B)$ for **iteration 1.**

The victims crossover receives is also part of our definition of $f_{select}$, and its relationship to $c \in Env$. In this example, $c = 0.60$. This means that we will be selecting our victims (parents) from the fittest 60% of the population. The number of victims we take is

$\lceil 0.6 * 3 \rceil = 2$. Suppose in our population, $eval(f_1), eval(f_2) < eval(f_3)$. This means that the set $\{f_1, f_2\}$ contains the two parents to be considered from crossover. As a result, we are able to proceed with $Crossover(\{f_1, f_2\}, B)$.

3. **Crossover or tree**

In the crossover function, we call $OrTree(\{f_1, f_2\}, crossover)$ to complete the crossover.



Step by Step Example

Step 1: We start with a single leaf node $(pr, ?)$, which gets expanded into its alternatives $pr_1, pr_2$, and $pr_3$, where $gp_1$ gets assigned to $sl_1, sl_2$, and $sl_3$, respectively

Step 2: $(pr_1, ?)$and $(pr_2, ?)$ are the two deepest leaves that last chose an assignment, $f_{crossEval} = 0$, that is consistent with one of their parents. We randomly chose pr1 between them to expand into its alternatives $pr_4$, $pr_5$, and $pr_6$, which assigns $gp_2$ to $s_1$, $s_2$, and $s_3$ respectively.

Step 3: $(pr_4,\ ?)$ and $(pr_5,\ ?)$ are the two deepest leaves that last chose an assignment,$f_{crossEval} = 0$, that is consistent with one of their parents. We randomly chose $pr_4$ between them to expand into its alternatives $pr_7$, $pr_8$, and $pr_9$, which assigns $gp_3$ to $s_1$, $s_2$, and $s_3$ respectively.

Step 4: We notice that $pr_7$ is unsolvable, so we change that leaf to$(pr_7,\ no)$.

Step 5: We notice that$pr_9$ is unsolvable, so we change that leaf to $(pr_9,\ no)$.

Step 6: Since $pr_8$ is the deepest leaf, we choose it to consider next, even though its last assignment is not consistent with either of its parents. Since pr8 can be solved, we solve this leaf and turn into $(pr_8,\ yes)$

4. **Choose next extension function**

Immediately after the crossover, we have $4$ chromosomes (facts) in our $population_1$ . Given we have reached $4 > Cap$ facts, we set $f_{wert}(crossover) = f_{wert}(mutation) = 1$, and $f_{wert}(delete) = 1 - f_{wert}(crossover) = 0$.

As a result, $f_{select}$ will select $delete(A, B)$ as the extension function for iteration 2.

The victims $delete(A, B)$ receives is part of our definition of $f_{select}$, and its relationship to $g \in Env$. In this example, $g = 0.40$. This means that we will be selecting our victims (parents) from the least fit 40% of the population. The number of victims we take is

$\lfloor 0.4 * 4 \rfloor = 1$. Suppose in our population, $eval(f_1), eval(f_2), eval(f_3) < eval(f_4)$. This means that the set $\{f_4\}$ contains the fact to be considered for delete. As a result, we are able to proceed with $delete(\{f_4\}, B)$.

5. **Delete**

Delete will simply remove the fact $f_4$ from our population, creating

$$population_2 = \{f_1, f_2, f_3\}$$

Immediately after its completion.

6. **Choose next extension function**

Following delete, according to the same logic in step 2, we choose a new extension function. Suppose, this time, the function is *mutate(A,B)* and we randomly select a fact from our population such that

$$A = f_1 = \{(gp_1, s_1), (gp_2, s_2), (gp_3, s_3)\}$$

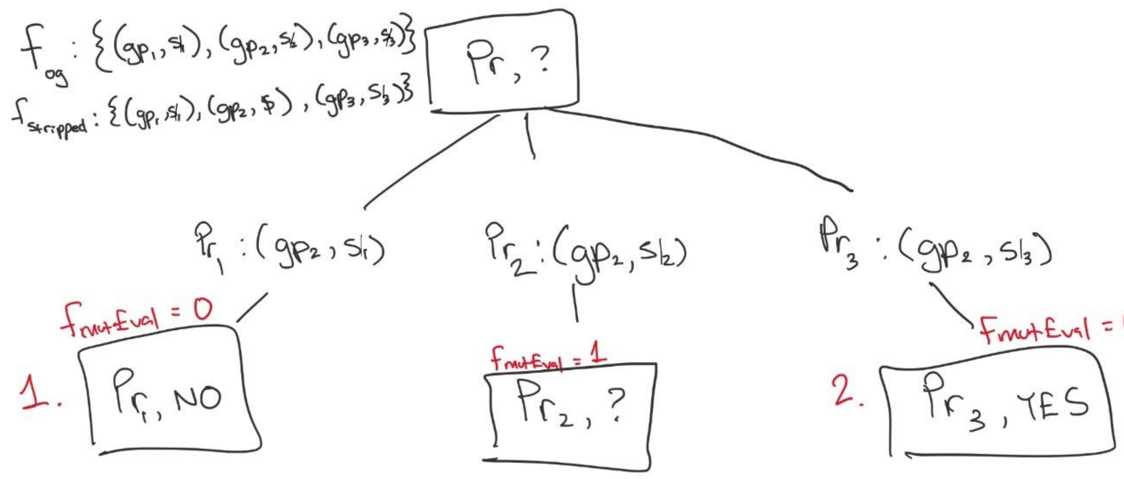
7. **Mutation**

Mutation takes in a fact, in this case $f_1$, and strips a random single slot $sl \in Sl$ from the fact $f_1$ to create $f_{stripped} = \{(gp_1, s_1), (gp_2, \$), (gp_3, s_3)\}$. Let $f_1 = f_{og}$

Mutation then sends these two facts into our or tree as

$$orTree(\{f_{og}, f_{stripped}\}, mutation)$$

From here, we can begin or or tree search, with the initial state $s_0$ being $f_{stripped}$



Step 1:

We start with a single leaf node $(pr,\ ?)$, which gets expanded into its alternatives $pr_1, pr_2$ and $pr_3$, where the unnasigned $gp_2$ get assigned to $sl_1, sl_2$ and $sl_3$ respectively.

Step 2:

$f_{mutEval}(pr_1) = f_{mutEval}(pr_3) = 0$, as the slot being assigned in each of these branches is different than the slot associated with $gp_2$ in our original fact $f_{og}$. As a result, $f_{mutEval}(pr_2) = 1$.

$f_{leafExt}$will need to select and tie break between $pr_1$ and $pr_3$. It does so by selecting one of the possible two nodes randomly. Our selection, in this example, is $pr_1$, which assigns, $sl_1$ to $gp_2$.

Step 3:

In our example, the fact generated from the leaf $f_i = \{(gp_1, sl_1), (gp_2, sl_1), (gp_3, sl_3)\}$ violates some hard constraint. As a result, we set the variable *sol* in $pr_1$ to *no.*

Step 4:

$pr_2, pr_3$ are now the deepest unassigned leaves that we can now explore. Our next random selection is $pr_3$, which assigns slot $sl_3$ to $gp_2$. This assignment creates the fact $f_{mut} = \{(gp_1, sl_1), (gp_2, sl_3), (gp_3, sl_3)\}$ which, in our example, passes all hard constraints, and is complete. As a result, we can set *sol = yes,* and we are done our search.

We now have a new population of

$$population_3 = \{f_1, f_2, f_3, f_{mut}\}$$

## 8. Final population

Considering *lim=3,* and we have completed 3 iterations of our extension rules, we have now reached the state $s_{goal}$, and have completed our set based search.

## And-Tree-Based Search

Re-iteration of Definition Legend:

$G = Games = \{g_0, \dots, g_n\}$

$P = Practices = \{p_0, \dots, p_n\}$

$GP = Games\ and\ Practices = G \cup P$
$Sl = Slots$

$$Constr(f) = \begin{cases} \text{Fact meets all hard constraints} \wedge Complete(f) & True \\ \text{Otherwise} & False \end{cases}$$

$$PartConstr(f) = \begin{cases} \text{Fact meets all hard constraints*} & True \\ \text{Otherwise} & False \end{cases}$$

$$Complete(f) = \begin{cases} \forall\, sl \in f, sl \neq \$ & True \\ \text{Otherwise} & False \end{cases}$$

$PartEval(f), f \in F'$: Returns the sum of penalties from all violated soft constraints.

$AConstr(a)$: Evaluates whether an individual assignment is valid. This is false if a game is in a practice-only slot, or vice versa.

$Assign = \{(g, sl) | g \in GP, sl \in Sl, AConstr(a)\}$: Ordered pairs representing possible assignments from games or practices to their corresponding slots.

$assign = GP \to Sl$ such that $\forall g\big(g \in GP \to \big(g, assign(g)\big) \in Assign\big)$: Surjection mapping a game or practice to a slot. Must be in Assign, to meet constraints.
$partassign: GP \times (Sl \cup \{\$\})$: Initial assignment provided by input

$passign = assign \cup GP \to \$$: Modified version of assign which allows null assignments

## And-Tree-Based Search

### Informal Description

Our other search paradigm is an And-Tree search, which utilizes a branch and bound search algorithm to find the best assignment of games and practices to slots assuming a valid input.

We start the search with whatever partial solution we are given. From there, we build from this parent solution by dividing the problem into $n$ different sub-solutions, where each sub-solution assigns the next game/practice to one of the $n$ slots. Each sub-solution is then checked if it violates $PartialConstr$, which checks if the sub-solution violates one of the hard constraints. If this partial solution does violate $PartialConstr$, then we say that the sub-problem is solved, since we cannot build a valid full solution from it.

After this, we look at all the leaves we have in our tree that have not been solved yet and calculate a lower bound for each leaf based off what the best $Eval$ evaluation you can theoretically get from turning that partial solution into a full solution; We call this function $PartialEval$. We continue the search on the various leaves until we reach a leaf that is a valid full solution to our problem. At this point, if it is a better solution than whatever we have in our Environment ($Env$) we replace it with the better one.

While we are searching through our tree, if we ever come across a partial solution that has a larger $PartialEval$ than the $Eval$ we get from the solution we have stored in our $Env$, we can prune, and instantly say that leaf is solved, since it is impossible for any full solutions built off it to be better than whatever solution we have stored in our $Env$.

At the end of the search, whatever solution is stored in our $Env$ is the optimal solution to our problem. If no solution is stored in our environment, we can conclude that there are no solutions to the problem given.

## Formal Description

We start our definition of the And-Tree search process by defining our model $A_\wedge = (S_\wedge, T_\wedge)$. Using the definitions for $S_\wedge$ and $T_\wedge$ given in lecture, we only need to define $Prob$ for the definition $A_\wedge$ of to be completed.

We define the problem space $Prob$ as the following:

$$Prob := \{f' | f' : GP \rightarrow Sl \cup \{\$\}, PartialConstr(f)\}$$

- Where $f'$ is an assignment of Games & Practices to slots or null assignments “\$” and $f'$ meets all hard constraints and Prob is a set of assignments.

To manage hard constraints on an assignment $f$ we define $PartialConstr(f)$ as follows:

$$PartialConstr(f') := \begin{cases} true & No\ hard\ constraints\ are\ currently\ violated\ by\ f' \\ false & Otherwise \end{cases}$$

Now that we have defined $Prob$, we want to define when $Prob$ is solved (sol-entry “yes”) or unsolved (sol-entry “?”). Before we can do that, however, we must define the function $PartialEval$, as well as our $Env$.

$$PartialEval(f') := \begin{matrix} Counts\ how\ many\ soft\ constraints\ are\ currently\ violated\ by\ f'\ and\ sum's\ their \\ penalties\ together, not\ including\ the\ penalties\ for\ gamemin\ and\ practicemin \end{matrix}$$

- $PartialEval(f')$ will be used as a way for us to measure a problem in a leaf.

$$Env = \left\{ (Bestsol, assign) | assign \in Prob, Bestsol = \begin{cases} \infty & assign = \$ \\ PartialEval(assign) & Otherwise \end{cases} \right\}$$

$Bestsol$ is set to infinity at the start to signify that no assignments have been made.

Our initial environment is $e_0 = (\infty, \$)$.

Now that we have defined everything in our search model, we need to determine when a problem is marked as sol-entry "yes" or sol-entry "?". Let $pr \in Prob$ , and

$$e = (Bestsol, assign) \in Env.$$

We say $pr$ is marked as sol-entry "yes" if and only if:

$$\neg PartialConstr(pr) \lor pr : GP \rightarrow Sl \lor (PartialEval(pr) > Bestsol)$$

- pr doesn't meet the hard constraints or
- pr is a complete schedule or
- pr has a worse (higher) evaluation score than our current best schedule.

And that $pr \in Prob$ is never marked as sol-entry "?" if it can't be expanded.

Now, to define how we can divide a leaf into its sub-problems we have the following definitions and their descriptions:

1. Let $prob \in Prob\ s.t. \exists i \in \mathbb{N}, 1 \leq i \leq n$ with $prob(a_i) = \$$ (Note: If this isn't true then prob is solved.)
    - Define a subproblem as an element of the larger problem such that it does not have an assignment at an index "i" in the range of possible assignments.
2. Let $h \in \mathbb{N}, 1 \leq h \leq n\ s.t. \forall j \in \mathbb{N}, 1 \leq j \leq n, prob(a_j) \neq \$ \land prob(a_h) = \$$.
    - Then, define an "h" and a "j" such that the subproblem has a partial assignment at index "j" and no assignment at index "h".
3. $Divide(prob) := \{f: GP \rightarrow Sl \cup \{\$\} | f(a_i) = prob(a_i), \forall i \in \mathbb{N}, 1 \leq i \leq n, i \neq h, f(a_h) \neq \$\}$
    - Then, we define a "Divide" function that maps a partial assignment of Games and Practices to an available set of Slots such that this assignment corresponds to a subproblem for every index of that subproblem "prob" that can be expanded upon.

4. $Div(prob, pr_1, \dots, pr_n)$ holds if and only if $Divide(prob) = \{pr_1, \dots, pr_n\}$.
   - Finally, we define Div as the function that determines whether a subproblem is expandable by checking the result of Divide(subproblem).

Now that we have defined $A_\wedge$ and $Env$, we only need to define our search control $K_\wedge$ to complete our Process definition $P_\wedge = (A_\wedge, Env, K_\wedge)$.

Our search control $K_\wedge$ chooses which state to transition to by using the following functions:

$$f_{eval}((pr, ?)) := \begin{cases} 0 & if\ pr\ is\ solved \\ 1 + PartialEval(pr) & Otherwise \end{cases}$$

$f_{leaf}(s)$ gives us a leaf with the minimal $f_{eval}$ score in state $s$. In the event of a tie, choose the deepest leaf of the tied leaves. In the event of a second tie, randomly choose one of the tied deepest leaves to proceed with.

$f_{trans}((pr, ?))$ selects the only $Div$ of $pr$ that satisfies definition 4.

Whenever we come across a leaf $(pr, ?)$ such that $pr: GP \rightarrow Sl \wedge Contr(pr)$, we check our current environment $e = (Bestsol, assign) \in Env$. If $Eval(pr) < Bestsol$, then we take $e' = (Eval(pr), pr)$ as our new environment. Once all leaves are solved, we take our current $e = (Bestsol, assign) \in Env$, and whatever $assign$ is, will be our best solution. If $assign = \$$, then there are no valid solutions to the problem.

Now that we have defined out search process $P_\wedge$, we now need to define our search instance $Ins_\wedge(s_0, G_\wedge)$. Using the natural definition of $Ins_\wedge$. We need to define our initial state $s_0$ and our Goal $G_\wedge$.

Our initial state is defined as follows:

$$s_0 = (partassign, ?)$$

- Where partassign is an initial assignment as provided by a valid input.

Our Goal $G_\wedge$ is defined as:

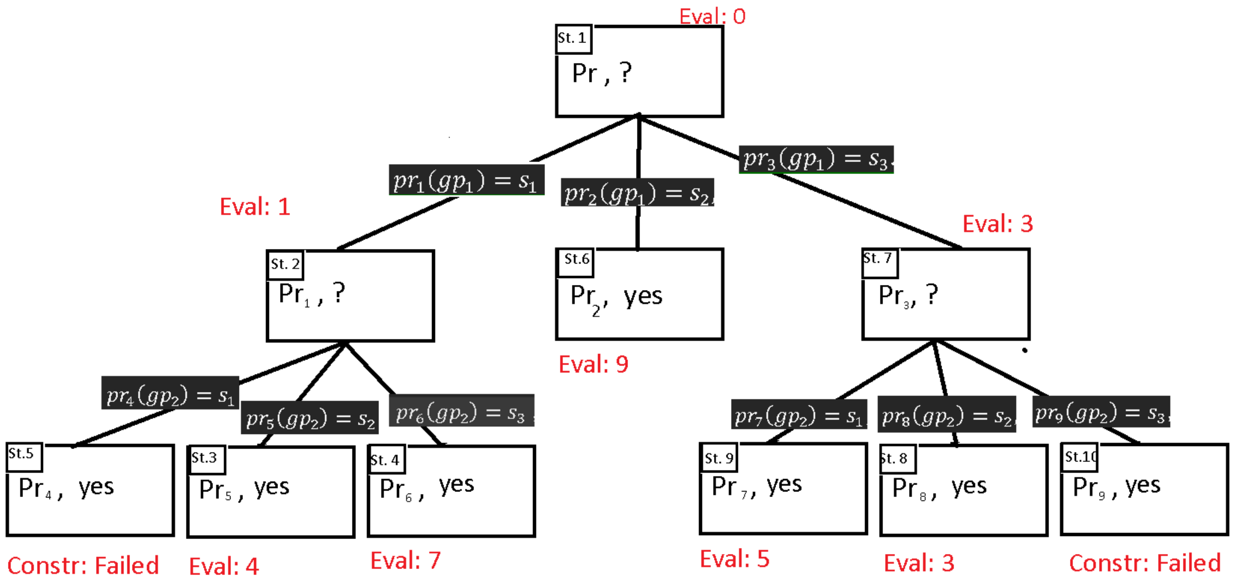
$G_\wedge(s) = yes, if\ and\ only\ if$

1. $s = (pr', yes)\ or$
   - The root is sol-entry “yes”, which means there’s no valid solution.
2. $s = (pr', ?, b_1, \dots, b_n)$, $G_\wedge(b_1) = yes$ and the solutions to $b_1, \dots, b_n$ meet our hard constraints, or
   - All branches are sol-entry “yes” and are compatible, with the best solution stored in $e = (Bestsol, assign) \in Env$

3. We have explored all leaf nodes, they are all marked sol-entry "yes" by our definition, and they have no further possible transitions, again with our best solution stored in $e = (Bestsol, assign) \in Env$.

Now that we have defined our Process $P_\wedge$, and Search Instance $Ins_\wedge$, we have defined everything necessary for our And-Tree search model.

## Visualization



## Steps

1. We start with $s_0 = (partassign, ?)$, so $[pr, ?]$ which as we specified, has a default $PartialEval = 0$. So, we divide it into its sub-solutions and assign $gp_1$ to $s_1$, $s_2$, and $s_3$ respectively: $pr_1(gp_1) = s_1$ , $pr_2(gp_1) = s_2$, $pr_3(gp_1) = s_3$.
2. Sub-solution $s_1$ derives $[pr_1, ?]$ which has $PartialEval = 1$ and can still be divided further into sub-solutions. Splitting into: $pr_4(gp_2) = s_1$ , $pr_5(gp_2) = s_2$, $pr_6(gp_2) = s_3$, which we will find all to be full assignments and so they are given priority in order of evaluation.
3. $pr_5$ is solved (sol-entry "yes") and so we mark it as such: $[pr_5, yes]$, with it's $Eval = 4$. Currently our best (lowest) score, we store it in our $Env$.
4. $pr_6$ is solved (sol-entry "yes") and so we mark it as such: $[pr_6, yes]$, with it's $Eval = 7$, failing to beat our best solution, so we keep our environment unchanged.
5. $pr_4$ is violating a hard constraint, so we mark it as sol-entry "yes" so we can ignore it.
6. Back to $pr_2$ we find that it has a $PartialEval = 9$, which is worst than the solution we have stored in our environment and all other $PartialEvals$, so we needn't expand it and so mark it as sol-entry "yes" so we can ignore it.

7. $pr_3$, like $pr_1$ and $pr_2$, is derived from $s_1$. We find that it can be divided further and since $PartialEval(pr_3) = 3$ it's sub-solutions might be better than our current best one stored in the environment. So we mark it sol-entry "?": $[pr_3, ?]$ and split from it into: $pr_7(gp_2) = s_1, pr_8(gp_2) = s_2, pr_9(gp_2) = s_3$, which we will find all to be full assignments.
8. $pr_8$ is solved (sol-entry "yes") and so we mark it as such: $[pr_8, yes]$, with it's $Eval = 3$, which beats our current best solution, so we replace the solution we stored in our environment with it.
9. $pr_7$ is solved (sol-entry "yes") and so we mark it as such: $[pr_7, yes]$, with it's $Eval = 5$, which does not surpass our best score, so we leave it.
10. $pr_9$ is violating a hard constraint, so we mark it as sol-entry "yes" so we can ignore it.
11. With that we are done searching as we have expanded our tree as much as possible and have stored the best solution among it's leaves in our environment, giving us the best solution of $pr_8$.