Research 1 Archive Paper

Robert Swanson

May 11, 2021

1 Topic

I spent the semester researching the 4-year scheduling problem for college students. I started the semester with 4 weeks of surveying the literature surrounding timetabling (for both high schools and colleges) as well as the more specific and relevant papers focused on scheduling college students 4-year plans. As I read these papers, I gained an understanding for how people went about solving these problems, and categorized the planning systems into complete searches which implement back-chaining algorithms (sometimes using prolog), and incomplete searches which use a genetic algorithm.

2 Experiment

After performing the survey, I defined the question I wanted to explore so that I could start designing and implementing a system to experiment with. The question was as follows: **how do different algorithms perform with generating course plans that optimize according to a set of requirements and preferences defined by the student?**

The potential independent variables that can be used in experiment include:

1. Complete vs Incomplete (GA) Search
2. Tools used
3. The search space
4. Pruning methods
5. The order of search

Tools can change either the algorithm used to solve the problem (eg. using prolog backtracking or java forward-chaining search), or can simple change the efficiency of the system (using python vs java).

The search space can be controlled to attempt to optimize the run-time. For example the search space for a GA that encodes a plan as a binary string including and excluding events operates within a search space of every possible permutation of course events (defined later) which is vastly invalid search space. This search space may potentially work for a GA, which may discover valid plans, but a forward chaining algorithms may need to minimize the initial search space. This could be done by constructing a tree that can be traversed to produce every plan that fulfils certain constraints. For example, a stack overflow post describes an algorithm to develop such a tree based off of the constraint of scheduling events such that none overlap in time. Another such tree could be produced based off of prerequisites, but would likely be far less constraining.

Instead, prerequisite constraints would likely perform better as a pruning mechanism. The distinction between initial search space and pruning the search space is that the former defines which options are considered, and pruning describes when to stop considering a particular traversal. Theoretically, any requirement could serve as a pruning point along the search, but it may not be worth the computational effort to prune a given requirement at each decision point.

Potential dependant variables include:

1. The score of the best plan offered by the system
2. The run-time of the system

3. The run-time for a "good enough" solution
4. The run-time for the first valid plan

3 Input/Output

The first step in designing the system was to determine what inputs would be provided to, and what output would be expected from the system. I decided to encode the inputs and outputs as json so that it could be portable with other applications, especially web services. While designing these formats, I create json schema files that describe the structure and format of both inputs and output files. These schema can be used to validate the files, but also can work with text editors to offer syntax verification and autocompletion while manually creating inputs.

I organized the system input into two files: `catalog.json` and `offerings.json`, and the output into `plan.json`.

The catalog contains information about a coursed offered in a particular catalog year. Such information links information that is not dependant on a particular section or offering to a course. Such information includes a course identifier, name, prefix, number, description, prerequisites, and offering patterns (eg odd springs). Catalogs are organized by department, each containing a list of courses. Following is an example of a `catalog.json` file:

```
{
 "catalogYear": "2021-2022",
 "departments":
  [{
   "name": "Computer Science",
   "description": "",
   "courses":
    [{
     "courseID": "COS 101",
     "name": "Information Tech",
     "prefix": "COS",
     "number": 101,
     "description": "The course..."
    }]
  }]
}
```

The offerings list is organized by semester and then by course. Each course contains a list of offering objects, which contain information specific to an individual section for a specific semester of a course. This includes a course reference number (crn), section number, course type (lecture vs lab), credits, professors, location, number enrolled, max enrollment, optional start and end dates (if they differ from the start and end times for the semester), and then a list of meeting objects. Meeting objects are composed of a list of days the meeting applies to, a start time, and an end time. Meetings describe the events required for a class, which the system will observe when scheduling a plan. Following is an example of a `offerings.json` file:

```
{
 "catalog-year": "2021-2022",
  "spring" : {
   "COS 120H" : [{
    "crn": 3451,
    "section": 1,
    "type": "lecture",
    "credits": 3,
    "professors": ["Dr. X"],
    "location": {
      "building": "Euler",
      "room-number": "217"
    },
    "num-enrolled": 0,
    "max-enrollment": 0,
    "start-date": "2021-03-22",
    "meetings": [{
     "days": ["Monday", "Wednesday"],
     "start-time": "12:00 PM",
     "end-time": "12:50 PM"
    }]
   }]
  }
}
```

Finally, we have the output file which describes the plans generated by the system. A plan file has the potential to contain multiple plans, so it consists of a list of plan objects. These plan objects have a score value and comment string, along with the plan itself. The plan is encoded as a list of term objects. A term objects has a

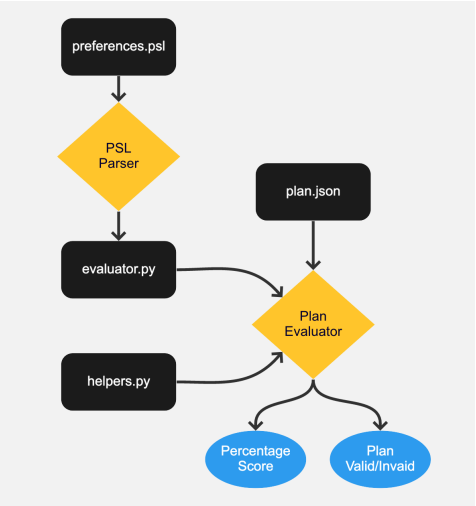
year and term describing itself, and then a list of course reference numbers (CRNs) which describe the sections the student would register for if they followed this particular plan. Following is an example of a `plan.json` file:

```
{
 "plans": [{
  "score": 100,
  "comments": "None at all",
  "terms": [{
   "term": "spring",
   "year": 2021,
   "sections": [ 11013, 23412]
  }]
 }]
}
```

4 Prototype

During the spring I merged the intentions of this class with a project for Language Structures in which I needed to create a domain specific language. The resulting project became an opportunity to prototype the system as I implemented a DSL that could produce plan evaluators based off of user requirements and preferences (as they were described in the PSL format). This section is the documentation I produced for that project.

This project defines a domain specific language (DSL) called PSL (preference specification language) to describe a set of requirements and preferences for a college student's four year plan. A PSL file is translated by the PSL parser into a python file that evaluates a plan (encoded using JSON) according to the preferences and requirements defined. Following is the data flow for this system:



Reference `README.md` for information on command line arguments that need to be passed to the parser and evaluator.

4.1 Blocks

PSL files are composed of one or more labeled blocks. For example:

```
definitions {
   priority strongly 10.
   priority moderately 5.
   priority slightly 1.
}
```

This blocks is labeled 'definitions', which doesn't mean anything to the system now, but could be used for imports in the future.

4.2 Statements

The block body is made up of one or more statements. There are currently 5 types of statements:

1. Requirements: `require <requireable constraint>`
2. Preferences: `prefer <priority name> <constraint>`
3. Priority Definitions: `priority <name> <val>`
4. Condition Blocks: `if <condition> then { <block> }`
5. Context Blocks: `when <condition> then { <block> }`

4.2.1 Requirements

Requirements specify constraints that a valid plan must satisfy. Requirements do not have any impact on the score of a plan, but only invalidate the plan if violated.

4.2.2 Preferences

Preferences specify constraints that can be used to compare the desirability of one plan over another. When a preference is evaluated on a plan, it is scored as a percentage of its priority (defined below). The score of the entire plan is calculated a the sum of these scores divided by the maximum possible score (aka the sum of the priorities).

4.2.3 Priority Definitions

Priority definitions are used to link symbolic names to numeric weights for use with preference scoring. For example in the above code, preferences weighted 'moderately' will have half as much impact on the plan score as preferences weighted 'strongly'.

4.2.4 Condition Blocks

Condition blocks allow for the evaluation of a group of requirements and preferences only if a condition is met. For example, the following snippet demonstrates a requirement to take COS 121, only if taking COS 120, a preference otherwise if taking SYS 120, and otherwise a requirement to take COS 104.

```
if taking "COS 120" then {
   require "COS 121".
} otherwise if taking "SYS 120" {
   prefer moderately "COS 143".
} otherwise {
   require "COS 104".
}
```

If the condition is not met, the preferences and requirements contained have no impact on the validity or score of the plan.

Note: conditional blocks can be arbitrarily nested within other conditional blocks or context blocks without any problem.

4.2.5 Context Blocks

Context blocks extend off conditions in that contained preferences and requirements are only evaluated if the condition is met, but differs in two distinct ways. First, the condition is not applied to the entire plan, but rather applied to every term within that plan. Second the requirements and priorities contained within the block only evaluates on a version of the plan that contains terms that met the condition. This version of the plan is called the current context, which is pushed to context stack when entering the scope of the when block, and popped off when leaving the scope. For example, if only terms A and B met the condition, then any nested constraints will only look at terms A and B to evaluate. For example:

```
when taking "COS 421" then {
   prefer slightly less credits in
   semester.
}
```

This snippet indicates that the user prefers (with a weighting of "slightly") to take less classes during terms when they are taking COS 421.

Just like conditional blocks, context blocks can be arbitrarily nested within other conditional or context blocks.

4.3 Constraints

Prefer statements are able to operate on any kind of constraint. Constraints that are boolean can be scored as either 0% or 100% met, while continuous constraints can be partially met. However, requirements are either met or unmet, so some constraints cannot be required. Thus, there is a distinction between constraints, and a subset of those constraints which are called requirable constraints. Every requirable constraint is a constraint, but not every constrain is a requirable constraint.

4.3.1 Requirable Constraints

Can use with `prefer` *or* `require`

Num Credits Hours

```
120 credits in plan
120 hours in plan
17 credits in semester
17 hours in plan
```

The `credits` and `hours` tokens are always equivalent in meaning. When required, this constraint indicates that at least the given number of credit hours exist in the current context. If this constraint is required, it makes no functional difference whether the `plan` token or the `semester` token is used.

When preferred, this constraint utilizes a optimization function centered at the provided value, and who's deviance is controlled by whether the `plan` or `semester` tokens were used. This has the effect of providing a 100% score when there are the exact right number of credits, with a decreasing score as the number of credits deviates from the optimum.

Course List

```
"COS 120", "COS 121", "COS 143"
```

This constraint is boolean in nature whether preferred or required. It is met when all the courses from the provided course list are present in the current context.

x of Course List

```
2 of "COS 120", "COS 121", "COS 143"
```

This constraint is boolean in nature whether preferred or required. It is met when at least x courses from the provided course list are present in the current context.

x Upper Division Hours

```
42 upper division credits
42 upper division hours
```

Again, credits and hours are equivalent in meaning. When required, this constraint is met when at least x credits from courses 200 level an above are in the current context. When preferred, an optimization scoring function is used.

Left Before Right

```
taking "COS 120", "COS 121" before "COS 265"
```

Whether required or preferred, this constraint is met if the one or more courses on the left hand side occur before the one course on the right hand side.

4.3.2 Non-requirable Constraints

Can only use with `prefer`

Earlier/Later Classes

```
later classes
later courses
earlier classes
earlier courses
```

The `class` token is equivalent in meaning to the `course` token. This constraint is scored with a sigmoid function operating on the average class start time with the following parameters:

- `earlier`
  - Lower Bound: Soft, 2:00 PM
  - Upper Bound: Soft, 10:00 AM
- `later`
  - Lower Bound: Soft, 10:00 AM
  - Upper Bound: Soft, 2:00 PM

More/Less Courses

```
more classes in semester
less courses in plan
```

The `class` token is equivalent in meaning to the `course` token.

The `semester` token should be used if the current context is a semester and the `plan` token should be used if the current context is the whole plan.

This constraint is scored with a sigmoid function operating on the number of courses in the with the following parameters:

- `semester`

  - `more`: Greater Equal To 6, Deviance 2
  - `less`: Less Than Equal to 5, Deviance 2
- `plan`
  - `more`: Greater Equal To 45, Deviance 16
  - `less`: Less Than Equal to 44, Deviance 16

More/Less Credits

```
more credits in semester
less hours in plan
```

The `credits` token is equivalent in meaning to the `hours` token.

The `semester` token should be used if the current context is a semester and the `plan` token should be used if the current context is the whole plan.

This constraint is scored with a sigmoid function operating on the number of credits in the with the following parameters:

- `semester`
  - `more`: Greater Equal To 15, Deviance 2
  - `less`: Less Than Equal to 14, Deviance 2
- `plan`
  - `more`: Greater Equal To 145, Deviance 16
  - `less`: Less Than Equal to 144, Deviance 16

4.4 Conditions

Conditions are passed into `if` and `when` statements to perform an evaluation on a particular context. Currently we only support conditions based on the existence of a course in a plan.

```
if not taking "COS 121" then {
    ...
}
```

```
when (taking "COS 120" or taking "COS 121") {
    ...
}
```

Conditions can participate in arbitrary Boolean logic expressions including:

- `(<left> and <right>)`
- `(<left> or <right>)`
- `not <cond>`

Parenthesis are necessary when using `and` or `or` to disambiguate order of operations.

4.5 Scoring

Preferences are scored as a percentage bounded from $0\% - 100\%$ and then multiplied by the weight/priority passed to the `prefer` statement. This guarantees that a perfectly met constraint will have a score equivalent to the weight, a unmet constraint will have a score of 0, and partially met constraints will fall between. Mathematically stated, the score is evaluated according to the following equation:

$$s = f(x) * w$$

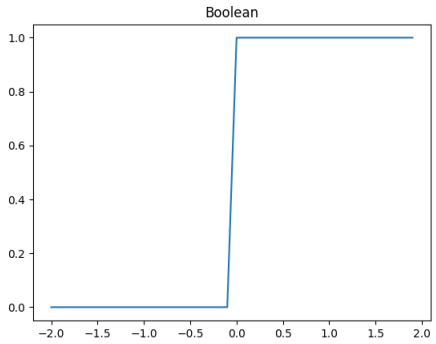
Where s is the score, $f(x)$ is the evaluation of the scoring function, and w is the weight of the preference. There are three functions this system uses to manipulate evaluations into the acceptable range:

1. Boolean
2. Sigmoid
3. Optimal

4.5.1 Boolean

Boolean functions are the simplest of all, and are used to score constraints for preferences which are either met or unmet (eg `taking "COS 120"`).

$$b(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

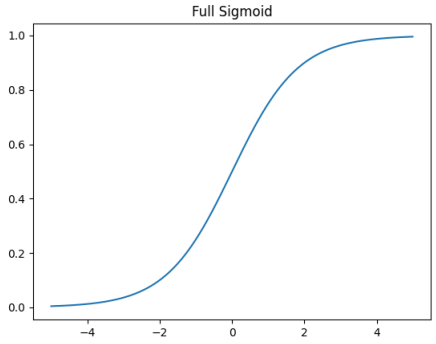


It should only be passed values of either 0 or 1, but if it is passed anything else it behaves as demonstrated by the following graph, who's y axis shows the value of $b(x)$ and who's x-axis shows the value for x.

4.5.2 Sigmoid

Sigmoid functions are used to score constraints for preferences in which the user is trying to maximize or minimize a value (eg `more credits`).

$$s(x) = \frac{1}{1 + 9^{-x}}$$

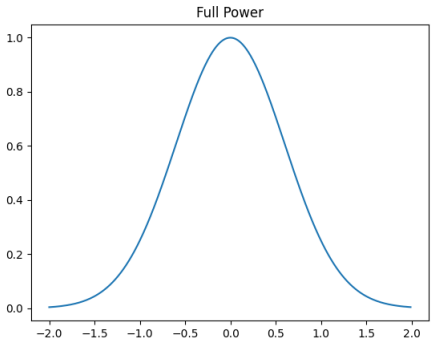


I developed and played with this equation on Desmos. The value 9 was chosen because it made the function have the property that $s(-1) = \frac{1}{4}$ and $s(1) = \frac{3}{4}$, which would be useful for bounding purposes (described later).

4.5.3 Optimal

Optimal functions are used to score constraints for preferences in which the user wants a value as close as possible to a certain value (eg `15 credits in semester`).

$$o(x) = (\frac{1}{4})^{x^2}$$



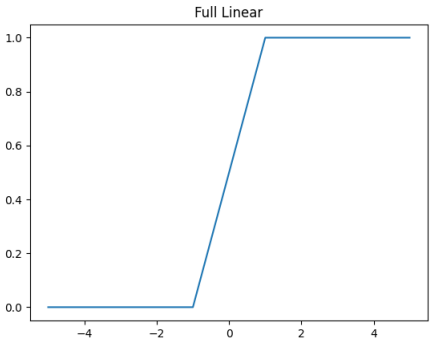
I developed and played with this equation on Desmos. The number $\frac{1}{4}$ was used because it made the optimal function have the property that $o(-1) = \frac{1}{4}$ and $o(1) = \frac{1}{4}$, which would be useful for bounding the function (described below).

4.5.4 Bounds

There are cases in which a constraint will always be evaluated to a tight range of values, in this case it is an inefficiency of the system to only offer 75% scores for well-met constraints. For this reason, a distinction from **soft bounds** and **hard bounds** are made. The aforementioned equations represent the soft bound equations for sigmoid and for optimal functions because they taper off at a given point, rather than a hard cut off.

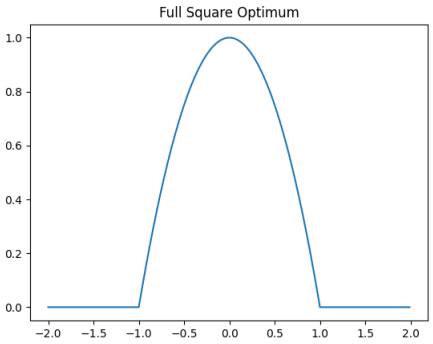
Hard Bound: Sigmoid A sigmoid function using a hard bound becomes a bounded linear function.

$$l(x) = \begin{cases} x & -1 \leq x \leq 1 \\ 0 & x < -1 \\ 1 & x > 1 \end{cases}$$



Hard Bound: Optimal A optimal function using a hard bound becomes a bounded upside-down quadratic function.

$$f(x) = \begin{cases} -x^2 + 1 & |x| \leq 1 \\ 0 & |x| > 1 \end{cases}$$



4.6 Normalization

Both the sigmoid and the optimal function need to be modified to vary most significantly on the expected range of inputs. The sigmoid function had two points at the 1st and 3rd quartiles which are used as base points. Inputs to the sigmoid function are thus normalized according to these two values using the following equation:

$$n(x, l, u) = \frac{x - l}{u - l}$$

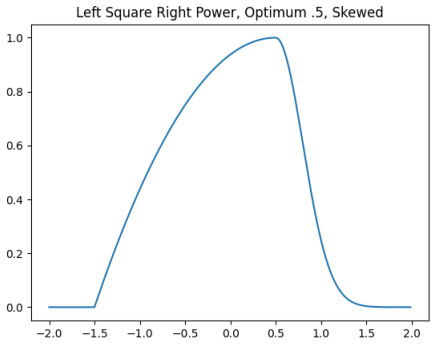
This equation defines l (lower) to be the input value that will score 25% and u (upper) to be the input value that will score 75%. If l>u then the function will give better scores to lower evaluations.

The optimal function had two points which describe the left and right quartiles, and also has a central optimum point that is defined. The following equation is used to normalize the evaluation:

$$n(x, m, d) = \frac{x - m}{d}$$

This equations defines a optimum point at m (mean) and the left and right quartiles to be (respectively) $-d$ and d (deviance).

With the combination of hard and soft bounds, each which can be provided their own parameters, it is possible to use different types of bounds for the two different points of a function. For example, this shows a optimum function where the optimum is 0.5, the left bound is a hard bound with a deviance of 2, and the right soft bound with deviance of 0.5.



5 Lessons from Prototype

Although designing and implementing the prototype took about half the semester to complete, it is my intention to rebuild the evaluator before continuing on to plan generation. This is because certain inefficiencies of the current system.

5.1 Unified System

First and foremost, the current system requires spawning a new process once for each new plan

evaluation, and passing the plan through file I/O into the child process to be evaluated. This problem is compounded by the fact that the evaluator executed by the python interpreter rather than by a compiled binary. Though this would not be a significant problem for a system that only needs to evaluate plans, it will be too slow for a plan generator, whose success depends on quickly evaluating a large number of plans. In response to this, I hope to implement the system in such a way where evaluating a plan does not require spawning any new processes. It would be viable to have a constant number of additional helper processes (eg. a prolog logic server), though the problem of I/O latency would still be present. Currently, my intention is to implement all parsers, evaluators, and plan generators in the same java program. Such a system would simply take a `.psl` file, `catalog.json` file, and `offerings.json` file, and would produce a `plans.json` file. This system would then be able to take advantage of shared memory to quickly generate and evaluate plans without copying all the plan data, and could also take advantage of multi-threading to concurrently evaluate multiple plans at once.

An alternative to this design that I did consider was utilizing a prolog interface both for plan evaluation and for plan generation. Such a system would parse .psl and json files in java, but would make calls using JPL to a prolog logic engine. This would involve generating a set of facts based off of the requirements and preference provided in the `.psl` file, which would likely be more intuitive for requirements, and quite a bit less intuitive for preferences (which are non-boolean). This alternative offered the possibility of using prolog's optimized backtracking implementation to search the space.

Ultimately, I chose against this possibility because the potential benefits of intuitive search were countered by my lack of familiarity with prolog, the overhead of translating from Java objects to prolog facts (which I learned to be time-consuming while implementing such an interface for the prototype). I also suspect that Prolog's back-tracking approach to searching would not be appropriate for a search space of such magnitude, so I judged that it might be more optimal to create my own forward tracking search.

5.2 System Behavior

As I designed the Python-generating interface in Java, I found much of my time was spent on deciding when I should fill in magic variables (eg parameters to scoring functions) and when I should pass up configuration to higher levels of abstraction. After deciding where to define those values, I then had to decide what those values should be, which was hard coded into the interface. Such decisions should be abstracted to one place such as a configuration file, or in configuration objects so that it there can be a single point of truth for the behavior of the system.

5.3 Preference Expression

After spending a large amount of time implementing specific expressions in the translator, I found that it would be better to change the syntax style to be use tokens more generically. For example, instead of of implementing unique expressions for the constraints `less courses` and `earlier courses`, it would be better to conceptualize a syntax that translates more generically to how the evalutor conceputalizes it: operations (eg equal, less) on one or more plan features (eg credits).

Restructuring the syntax in this way could also result in a broader reconceptualization of constraints that could be useful later on when implementing pruning decisions in the search space. This is because there will be different optimal pruning behaviors for different kinds of constraints. For example, constraints that require less than a certain number of credits do not need to be reconsidered if credit counts decrease, and they have the ability to prune a search path if and only if credit counts have increased. Additionally, it may be helpful to conceptualize a "difficulty of computation", or "likelihood of pruning", or some combination of the two, for constraints that could be used to order evaluation of constraints to prune a plan as quickly as possible. In the same way, optimizations could be made to batch

checks that are less likely to result in pruning (eg check pruning conditions after every 5 courses instead of after adding each course to a plan).

6 Considering Plan Generation

Although I had hoped to end the semester with a system capable of generating plans, I only got as far as plan evaluation. However, I did spend time thinking about and reading up on plan generation and have developed some thoughts relating to how to approach the problem. Many of these thoughts have already been expressed in Section 2 including the options of genetic algorithms and complete searches, and desire to constrain initial search space.

6.1 Constraining Initial Search Space: Event Overlap

In the search for constraining the initial search space as much as possible, it was desirable to find a method to include the constraint of preventing overlapping sections from being scheduled. I found an algorithm described on StackOverflow which constructed an event graph with the following properties:

- Each node A points to each node B that can be attended after event A, which conflicts with every other event node A points to
- Any traversal through the graph describes a sequence of events without event conflict (no event starts while another event is occurring)
- A complete traversal through the graph, where each complete path is also expanded to produce all combinations that conform to that ordering (with only but not all events in the path) will result (with redundancy) in every valid

Such a graph could be traversed in order to produce the search space for plan generation. Such a search space has the advantages of producing only conflict-less plans, which cuts down dramatically from the brute force search space. It also results in searching plans with early events first, which would be optimal for finding plans that result in the earliest possible graduation. The search space does however suffer from the trait that it would produce a large number of plans that include redundant sections (eg same class in two different semesters) for a course or not enough events for a course section (eg Monday event but not Wednesday event). While I havn't thought through this problem completely yet, the worst case solution to these problems would be pruning traversals that violate these expectations. However, it would be more optimal to encode such logic into the original search space.

6.2 Constraining Initial Search Space: Prerequisites

A common approach in the literature (eg PERCEPOLIS) for plan generation was to create a tree describing prerequisites for courses. A traversal resembling a breadth first search is then performed on that tree to fill in semester "slots". This search would also favor plans with earlier graduation, but would also spend much of its time pruning plans that run into event conflicts (this would be more a problem when courses have less offerings, and less of a problem for larger schools). This algorithm resembles the human approach to scheduling, and would likely be effective at finding "good enough" solutions quickly.

7 Moving Forward

I hope to implement a fully Java-based evaluator over the summer, but if I can't, then I'll continue on with the existing prototype. In the Fall I plan on developing plan generation with different search spaces and pruning mechanisms and comparing their performance. This will include run-time for certain, and may also include test with humans to compare the "real" desirability of generated plans.