AI Planner Final Technical Report

Team 15
May 2019

PROMETHEAN

AUTHORS
Ben Wasko | Hayden Nix
Eoin Doherty | Taylor Jesse
Samuel Reed | Evan James

Table of Contents

Introduction and Background

Current manual control of spacecraft is resource intensive, and communication travel time from mission control can inhibit a spacecraft's ability to make time-sensitive decisions. To solve this problem, we have created an artificial intelligence planning system to run onboard a variety of spacecraft. This planning system utilizes a network of executable tasks to create a plan that allows a spacecraft to autonomously transition from its current state, through intermediate states, to a defined goal state. The input to this system is a problem-state file defined in a user-friendly JSON planning language. The language is flexible enough to model any problem state and accommodate user-specified optimizations. The optimized plan is then executed in a simulation system that represents the running state of the spacecraft. To demonstrate the value of the planning system in real-world situations, the simulation system will inject perturbations into the execution of the plan. The planning system is never informed of these perturbations at any point in the planning process. Thus, the planning system will autonomously recognize the perturbation and replan in real time to adapt, demonstrating its robustness and capability of handling unplanned disruptions. Every aspect of this system has been created with the computational limitations of actual spacecraft in mind. The automation of spaceflight is incredibly beneficial and this system serves as a stepping stone for translating these ideas from simulation systems to actual spacecraft.

Summary of Research

The field of planning in artificial intelligence has been highly developed throughout its over forty years of study. While planning is considered a classical AI problem, the applications of these algorithms vary across many problem spaces and use many different approaches. For this reason, a great deal of time was spent researching this field in the beginning stages of Promethean's system development in order to find the best approach for the proposed problem. The team organized the necessary research into two categories: an AI Planning Language and an AI

Planning Algorithm. These results are compiled into two trade studies that can be accessed in Appendix E and Appendix F. In short, the Language Trade Study concluded that creating a new language in JSON would be the most flexible approach. Additionally, the Planning Algorithm Trade Study showed D* to be the best choice due to its dynamic perturbation handling and the fact that it is an extension of A*.

System Overview

Summary

Promethean's main objective is to serve as a foundation for the future of autonomy in spacecraft. Every component of Promethean has been designed to be robust and modular. The parser takes JSON input in the form of the planning language and turns that input into Java objects that can be utilized by the Planner to generate a plan. This plan will outline the steps necessary for the spacecraft to transition from its current state to a goal state. The plan is then executed by the executing agent component in order to authentically simulate the changing of states the spacecraft would go through while enacting the autonomously generated plan. The simulation will also inject predefined perturbations in order to enhance the practicality of the simulation by requiring it to replan and accommodate those perturbations in real time.

System Architecture

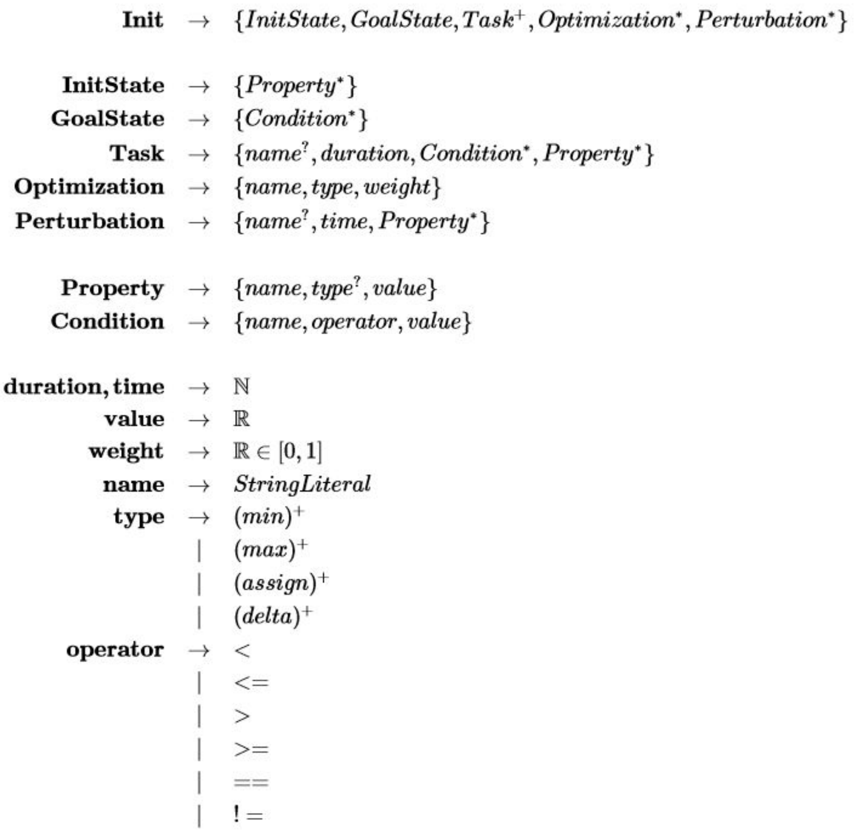
The system architecture of Promethean relies on many different components interacting with each other. Each component has been designed to adhere strongly to object-oriented SOLID design principles in order to reduce cognitive complexity as well as allow future developers to make changes or replace specific components with as little friction as possible.

The API is responsible for facilitating the execution of the separate components, passing all user inputs to those components, and passing system output to the user. The Parser is responsible for parsing the JSON inputs into Java objects. These Java objects, passed from the Parser to the Planner through the API, will be used by the Planner to generate a plan for execution. This plan will be passed from the Planner to the Executing Agent by the API. The Executing Agent will then begin simulating the generated plan. If any perturbations occur, the Executing Agent will detect those perturbations during the simulation and notify the Planner to generate a new plan from the current system state.

The communication between the Executing Agent and the Planner allows the simulation system to handle perturbations in an authentic manner that does not require any user intervention. The entire process of generating a plan and accommodating perturbations will happen autonomously. Appendix A contains architecture diagrams outlining the entire system pipeline.

JSON Planning Language

Figure 1: Promethean Planning Language Grammar



$$
\begin{array}{rcl}
\textbf{Init} & \rightarrow & \{InitState, GoalState, Task^{+}, Optimization^{*}, Perturbation^{*}\} \\
\\
\textbf{InitState} & \rightarrow & \{Property^{*}\} \\
\textbf{GoalState} & \rightarrow & \{Condition^{*}\} \\
\textbf{Task} & \rightarrow & \{name^{?}, duration, Condition^{*}, Property^{*}\} \\
\textbf{Optimization} & \rightarrow & \{name, type, weight\} \\
\textbf{Perturbation} & \rightarrow & \{name^{?}, time, Property^{*}\} \\
\\
\textbf{Property} & \rightarrow & \{name, type^{?}, value\} \\
\textbf{Condition} & \rightarrow & \{name, operator, value\} \\
\\
\textbf{duration, time} & \rightarrow & \mathbb{N} \\
\textbf{value} & \rightarrow & \mathbb{R} \\
\textbf{weight} & \rightarrow & \mathbb{R} \in [0,1] \\
\textbf{name} & \rightarrow & StringLiteral \\
\textbf{type} & \rightarrow & (min)^{+} \\
 & | & (max)^{+} \\
 & | & (assign)^{+} \\
 & | & (delta)^{+} \\
\textbf{operator} & \rightarrow & < \\
 & | & <= \\
 & | & > \\
 & | & >= \\
 & | & == \\
 & | & != \\
\end{array}
$$

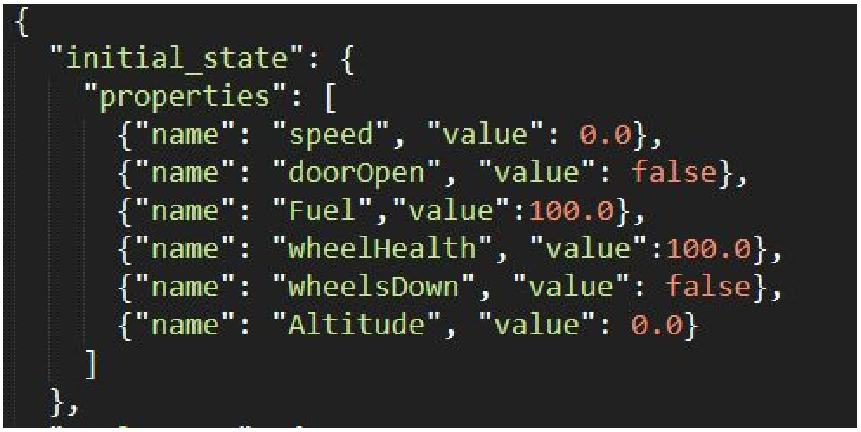
Note: This grammar uses the Regular Expression Syntax where * = 0 or more; + = 1 or more; ? = 0 or 1

Autonomous planning requires a representational model to map real environments to code. Due to the vast diversity in planning problem domains, an extensive trade study was conducted that evaluated various planning language alternatives against the proposed problem (shown in Appendix E). Ultimately, the trade study's findings led to the development of a new planning language because existing representations were too restrictive and specific. Thus a domain agnostic knowledge representation was designed that could model any problem in the form of an initial state with its properties, a specified goal state, and a set of tasks that can be executed. Additionally, Promethean's language has extensions for the user to include optimizations and perturbations that can be injected into the system. **Figure 1** shows the BNF style grammar of the planning language representation.

Additionally, the Language Trade Study results pointed to JSON as the best language format. JSON is a data-interchange format that is used throughout all types of software. Thus, it ranked very high in usability and understandability in the trade study because it's structure is well known across industry. Additionally, JSON has clear and logical delimiters that make it easy for both humans and computers to parse. Lastly, since JSON is a notation that can represent any object in terms of field names and values, it is extremely flexible for extension because it is domain independent.

Figure 2 illustrates a sample JSON representation of the Promethean Planning Language's `InitState`. As specified in the grammar, the `InitState` consists of any number `Property` objects. JSON easily conveys this by encasing object components in curly braces and lists of objects in square brackets. Thus, it is simple for both the user and the machine to extract information from the JSON representation. Further reasoning behind the language's components will be explained in the Data Model Class Breakdown. Appendix C contains a JSON user guide that details how to formulate a JSON file as input into the Promethean Planner and explains the meaning behind each language value.

Figure 2: Sample JSON of Promethean's InitState



```
{
  "initial_state": {
    "properties": [
      {"name": "speed", "value": 0.0},
      {"name": "doorOpen", "value": false},
      {"name": "Fuel","value":100.0},
      {"name": "wheelHealth", "value":100.0},
      {"name": "wheelsDown", "value": false},
      {"name": "Altitude", "value": 0.0}
    ]
  },
```

System Components

Data Model

Overview

Promethean's Data Model provides the basis for representing the planning environment within the system. The goal when designing the Data Model was to abstract domain-specific attributes away such that the system would be flexible in working with any problem space. The Data Model represents the core components of the planning and simulation environment. A system state describes a system's attributes at a given time as a list of properties. Tasks encapsulate the changes from one state to another. Using this model, the Planner can effectively build a graph with system states as the nodes and tasks as the edges. Below is the class breakdown of every Data Model component. The corresponding class diagram can be found in Appendix B.

Class Breakdown

- `SystemState`
    - The `SystemState` class models the environment at a specific time- both internal and external- of the planning executive.
    - `SystemState` components include:
        - `UID`: A system set integer that uniquely identifies each `SystemState`.

        - `time`: The current time of the `SystemState`, denoted in internal system ticks.
        - `properties`: A `PropertyMap` of `Property` objects that describe the `SystemState`. (See `Property` described below).
        - `previousState`: A `SystemState` object that denotes the previous state in the state network before applying a `Task` to get to the current state.
        - `previousTask`: A `Task` object that stores the most recent task applied to reach the current `SystemState`.
        - `gValue/hValue`: Stores best-to-state path cost and heuristic measures calculated by the `TaskWeight` and `Heuristic` classes.
- `Task`
    - The `Task` object represents an action that the executive can take to go from one `SystemState` to another.
    - `Task` components include:
        - `UID`: A system set integer that uniquely identifies each `Task`.
        - `name`: A string that identifies a `Task` object in a human readable format.
        - `duration`: The time in system ticks that it takes for a `Task` object to execute.
        - `requirements`: A list of `Condition` objects that define the constraints that a `SystemState` needs to meet in order to execute the `Task`. (See `Condition` described below).
        - `property_impacts`: A `PropertyMap` of `Property` objects that encapsulate the property changes that the `Task` object applies to the original `SystemState`.
- `TaskDictionary`
    - The `TaskDictionary` object provides a uniform way to encapsulate a set of all possible `Tasks` that are available to the executive.

    - The `TaskDictionary` simply consists of:
        - `tasks`: Which is a Java Hashmap that stores each `Task` object as the value and the `Task name` as the key.
- `GoalState`
    - The `GoalState` object represents the conditions that must be met in order for the end state of the Planner to be achieved.
    - The `GoalState` consists of:
        - `requirements`: A list of `Condition` objects that define the constraints that need to be met in order to reach the system's goal.
- `Property`
    - `Property` is an abstract class that helps describe environmental variables of `SystemStates`, `Tasks`, and `Perturbations`.
    - `Property` components include:
        - `name`: A string that uniquely identifies each `Property` object.
        - `type`: This field currently supports two strings: "assign" and "delta". This allows for `Property` objects to be used in `Task property_impacts`. "assign" is the default value for `SystemState properties`- meaning the `value` field is strictly assigned to the `Property name`. "delta" can be used in `Task` or `Perturbation` objects and it means the old `Property value` changes via the addition of the new `Property value`.
    - Three classes extend `Property`: `NumericalProperty`, `BooleanProperty`, and `StringProperty`. These classes contain the field: `value` which has the corresponding type respective to its name. This design allows for each subclass to write individualized functions depending on the type of `Property`. Examples of each `Property` class instances include:
        - `BooleanProperty: name: "Door Open", value: true`
        - `NumericalProperty: name: "Altitude", value: 1000`

        - `StringProperty: name: “Risk Level”, value: “High”`
- `PropertyMap`
    - The `PropertyMap` object provides a uniform way to encapsulate a set of `Properties`.
    - A `PropertyMap` consists of:
        - `property_map`: Which is a Java Hashmap that stores each `Property` object as the value and the `Property name` as the key.
- `Condition`
    - `Condition` is an abstract class that defines `Property value` constraints that must be met. This object is used to define the preconditions of a `Task` or a `GoalState`.
    - `Condition` components include:
        - `name`: A string that uniquely identifies the `Property` in which the `Condition` describes.
        - `operator:` A string that represents what type of constraint is being imposed on the `Property` such as thresholds, assignments, etc. The Planner currently handles: `“==”, “!=”, “<”, ”<=”, “>”, “>=”` as valid operators.
    - Three classes extend `Condition`: `NumericalCondition`, `BooleanCondition`, and `StringCondition`. These classes mirror the Property class structure in that they also have a field: `value` which has the class’ respective type. Examples of each Condition class include:
        - `BooleanCondition: name: “Door Open”, operator: “==”, value: false`
        - `NumericalProperty: name: “Altitude”, operator: ”>”, value: 1000`
        - `StringProperty: name: “Risk Level”, operator: “!=”, value: “High”`
- `Optimization`

    - The `Optimization` object encapsulates a `Property` that the user wants to optimize over and what kind of optimization it is.
    - `Optimization` components include:
        - `name`: A string that uniquely identifies the `Property` that is being optimized.
        - `type`: A string- either `“min”` or `“max”`- that identifies the optimization in terms of minimizing or maximizing the `Property` value.
        - `weight`: A double from [0.0-1.0] that is the multiplier of the `Optimization` with respect to all properties in the system.
- `StaticOptimizations`
    - `StaticOptimizations` is a class that provides a uniform way to reference all of the system’s `Optimization` objects.
    - `StaticOptimizations` is composed of:
        - `Optimizations`: A static list of `Optimization` objects
- `Perturbation`
    - `Perturbation` objects encapsulate simulated, fixed perturbations that are injected into the Executing Agent. This class is only necessary in artificial environments because real perturbations would be caught by the executive during a `SystemState` comparison. However, since the Planner does not have a physical executive at this stage of the project, the user can create predefined perturbations to test the robustness of the Planner as it handles unplanned state changes.
    - `Perturbation` components include:
        - `name`: A string that provides a human readable identification of the `Perturbation`.
        - `time`: The time- in system tick units- that the `Perturbation` should be injected into the Executing Agent after the start of plan execution.
        - `property_impacts`: A `PropertyMap` of `Property` objects that encapsulate the property changes that the

Perturbation object applies to the current SystemState.

Planner

Overview

Promethean's Planner tries to create an optimal plan from the given initial state to the given goal state. The Plan created by the Planner consists of two main components: a list of PlanBlocks and a boolean stating whether the goal was found. In order for the Planner to run, the user must provide a stop time to the system so that it does not search forever. If the goal state is not found before the stop time is reached, the Planner will return a plan from initial state to as far as the Planner was able to search. In this case, the boolean is set to false- stating that the Planner did not reach the goal state. Whether the goal is found or not, the Planner with return with an ordered list of PlanBlocks which contain a Task and a SystemState. A PlanBlock's Task is the Task which immediately precedes the selected PlanBlock's SystemState. This is always true except in the case of the first plan block- in this case, the task is applied to the initial state. Promethean's Planner was made with modularity in mind, meaning the user is able to create and provide a new algorithm to the Planner using the Algorithm interface. Promethean's planning system currently implements the Algorithm interface with a slightly altered version of A*. For details on the research that led to using A* Search in the planning system, see the Planning Algorithm Trade Study in Appendix F.

Planning Algorithm

The A* implementation is mostly made up of two classes: AStar and GraphManager. The AStar file contains the high level logic, including checking whether the goal has been found and picking the next best state to search. The GraphManager class contains all of the lower level information about the graph being formed including the current frontier, logic to add states to the frontier, and calculating a state's f_value. The f_value is the heuristic value added to the current best-to-state path cost. Promethean's A* implementation differs from classical A* in the following ways:

State Templates

A `StateTemplate` is the the smallest data representation for states in the frontier. A `StateTemplate` includes references to their previous state and task as well as the f, h, and g values. Only when a `StateTemplate` is chosen out of the frontier to be explored does it get converted into a full `SystemState`. This design choice was made in order to accomodate the limited memory space onboard spacecraft. The team knew that as the algorithm searched for the goal, the frontier would grow very large. By making the frontier hold `StateTemplate` objects rather than `SystemState` objects, Promethean was able reduce memory usage significantly.

CLF

A Control Lyapunov Function can be used to indicate whether Promethean can guarantee finding a goal state while searching an infinite state space. When it comes to search problems, a Lyapunov function is most commonly characterized as a descent function with no local minima except at goal states. The definition given by Theodore J. Perkins and Andrew G. Barto in their paper, *Heuristic Search in Infinite State Spaces Guided by Lyapunov Analysis*[1], states a control Lyapunov function (CLF) is a function $L : S \rightarrow \Re$ with the following properties:

1. $L(s) \geq 0$ for all $s \epsilon S$.
2. There exists $\delta > 0$ such that for all $s \notin G$ there
   is some search operator O▯ such that
   L(s) - L(Succ▯ (s)) $\geq \delta$.

Where S is the set of all possible states and G is the goal or terminal state. Moreover, they give a theorem for A* search, particularly that if there exists a CLF L for a given search problem (SSP), and either of the following conditions are true:

1. the set of search operators descends on L, or
2. the SSP's costs are bounded above zero,

then A* search will terminate, finding an optimal path from s_0 to G. During the Planner's search process, it ensures that these properties stay true within the

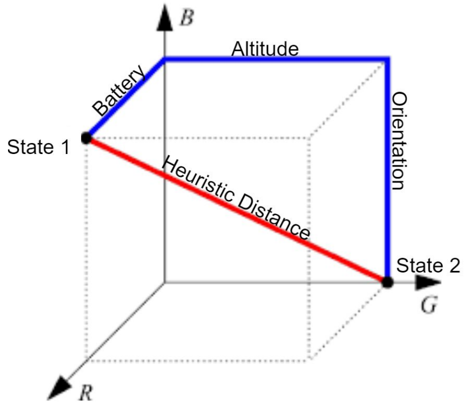
[1] Perkins, T. J., & Barto, A. G. (n.d.). *Heuristic Search in Infinite State Spaces Guided by Lyapunov Analysis* [Scholarly project]. Retrieved September 9, 2018, from http://www-anw.cs.umass.edu/pubs/2001/perkins_b_IJCAI01.pdf

heuristic function by continually checking that each state explored has a smaller heuristic value than its previous state. If this property stays true then at each step of the search, there is a step down the descent function and the goal state gets closer and closer until it is found. This system is in place so that the user can override their time limit by activating the CLF, and as long as the Planner keeps getting closer to the goal it will not terminate the search.

Heuristic

The heuristic function is a 2-norm on the difference vector of property values from any state to the goal state. Since the heuristic is using a 2-norm of a difference vector, the resulting value can be viewed as a multidimensional euclidean distance from one state to the next (**Figure 3** illustrates an example below). Because the euclidean distance is the absolute shortest distance from one point to another, the actual path cost will never be less than the heuristic distance, thus making the heuristic function admissible by definition. An admissible heuristic is what allows for the guarantee of finding the optimal path.

Figure 3: Example of Euclidean Distance Between States



Optimizations

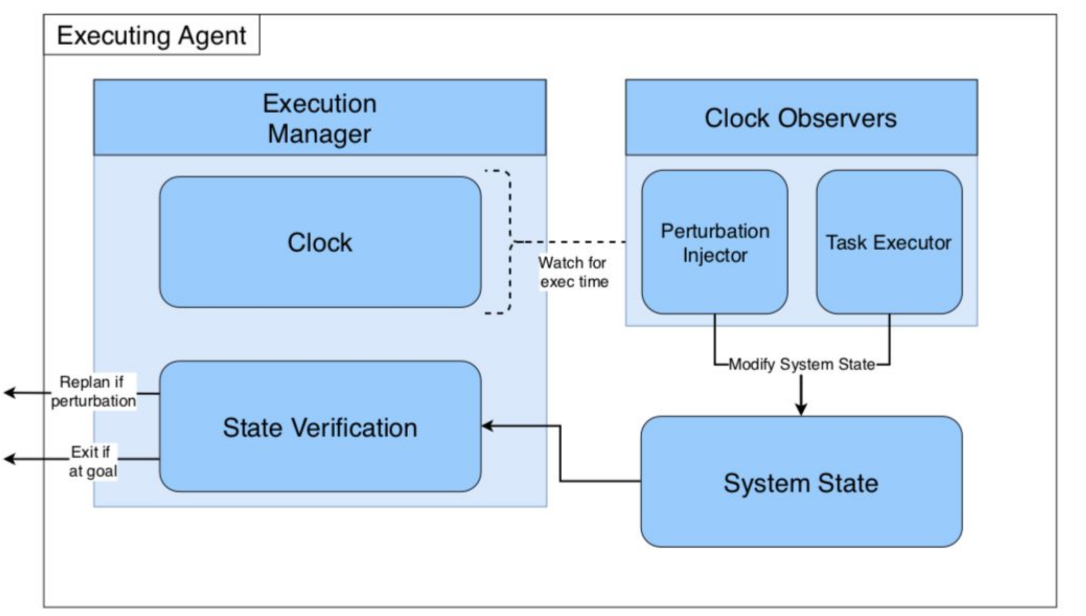
Optimizations are user defined weights that are applied to the path cost function. These weights can be between zero and one and when no optimization is given for a particular property, that property is assumed to have a zero weight. These weights determine the percentage of a property's value that will end up contributing to the overall cost function value. Currently the Planner uses scalar weights for each property, but this can be changed to be a function of the property value itself- ie. weight the optimization on minimizing energy usage more as energy levels decrease.

Executing Agent

Overview

The Executing Agent serves as the main component for executing the plan generated by the Planner within the simulated or physical environment in which the Promethean Planner is running. In the current iteration of this project, every aspect of the system is enacted within a simulation; the Executing Agent's main purpose is to execute the plan as authentically as possible and track the simulated state changes of the system. The Executing Agent also plays a crucial role in allowing the entire simulation to handle perturbations. The Executing Agent continuously compares the simulated system state with the plan generated by the Planner. If there is ever a perturbation in the execution of that plan, the Executing Agent will halt operation and call on the Planner to replan from the current simulated system state. The Executing Agent is composed of four major components: the Execution Manager, the System Clock, the Task Executive, and the Perturbation Injector. Appendix A contains architecture diagrams that outline the Executing Agent's pipeline. An overview of the Executing Agent is shown in **Figure 4**.

Figure 4: Executing Agent Overview



Execution Manager

The System Clock, which is composed of the `Clock`, `ExecutionManager`, and `ClockObserver` classes, produces the heartbeat of the Executing Agent. All of the execution is facilitated through the `ExecutionManager` The entire execution system utilizes the Observer design pattern. It is responsible for initializing, starting, and stopping the Clock. The `ExecutionManager` will stop the Clock when a full plan is completed. The `ExecutionManager` also handles calling the Planner and waiting for a new plan if it is informed that a perturbation has occurred. When execution is complete, the `ExecutionManager` sends a list of all the system states that occurred throughout execution to be serialized into JSON Output.

System Clock

The `Clock` class is a unit based clock that serves as a starting gun for the Executing Agent. Every time the Clock ticks, all of the instances of `ClockObserver` check to see if the current time matches a time that the `ClockObserver` needs to execute a task or perturbation. The `Clock` and `ClockObserver` adhere to the Observer design pattern.

Task Executor

The TaskExecutor is the component of the Executing Agent that is responsible for executing the tasks from the plan generated by the Planner on the current simulated system state. The TaskExecutor is never informed of the perturbations the PerturbationInjector will be injecting. This is to ensure the simulation is kept as authentic as possible. At the end of the execution of a task or perturbation, the TaskExecutor is responsible for comparing the current simulated system state to the associated state from the PlanBlock generated by the Planner. If there is a mismatch it means that a perturbation has occurred and that will be bubbled up to the ExecutionManager.

Perturbation Injector

The PerturbationInjector simulates perturbations that have been defined in the planning language. The PerturbationInjector generates a new state when the clock's time matches a perturbation's time. It then adds that new state back to the static stack of system states that the TaskExecutor uses to generate new states. The PerturbationInjector handles time perturbations differently from other perturbations. In addition to updating the newest state with this time perturbation, the PerturbationInjector also adds the change in time as "lag" to the Clock. The TaskExecutor adds this lag to the duration of the task that is currently being executed. This simulates some unexpected delay in the executing of a task causing that task to finish later than expected.

Parser

The Promethean Planning System includes a parser that parses the user input (see JSON Planning Language described above) into Java objects. This parser utilizes Google's open source GSON library which provides a simple interface for translating JSON into Java. Thus, the parser effectively maps the JSON Planning Language into the planning system's data model which is the foundation of the knowledge base used to create and execute plans.

The parser is a modular component of the Promethean Planning System accessible through the use of an interface. This allows for a plug and play capability that enables additional planning languages to be used in the system. The programmer simply implements the ParserInterface with a mapping from the new language to Promethean's data model. This design allows for languages to be interchangeable and keeps the Planner agnostic to representations outside of its domain. Reference the architecture diagrams in Appendix A for more information on Parser's role in the system.

Application Programming Interface

Traditionally an Application Programming Interface (API) serves the purpose of providing a defined set of functions and subroutines that allow dynamic and flexible interfacing with the application. Promethean's API class is designed to serve as an interface/hub for the user to interact with specific components of the system or to execute specific functionalities of the system. Every system output will be passed through the API and every user input will be passed from the CLI directly to the API to dictate the details of the desired execution. The current available API functions are the following:

- `throwPlannerError`
    - Throws an error from the Planner Component
- `throwParserError`
    - Throws an error from the Parser Component
- `throwOutputError`
    - Throws an error from the Output Component
- `parseInput`
    - Utilize the Parser to parse a JSON input into a list of parsed objects
- `generatePlanFromParsedObjects`
    - Utilize the Planner to generate a `Plan` object from a list of parsed objects
- `generatePlanFromJSON`
    - Utilize the Parser and Planner to parse a JSON input into parsed objects and turn them into a `Plan`
- `generatePlanFromSystemState`
    - Utilize the Planner to create a new `Plan` from a `SystemState` object
- `executeExisitingPlan`
    - Utilize the Executing Agent to execute a simulation using an existing `Plan`
- `executePlan`

  - Utilize the Parser to parse a JSON input into a list of parsed objects that the Planner will use to generate a `Plan` that will be used in a simulation executed by the Executing Agent

Command Line Interface

As an example of how the API can be used, Promethean comes with a simple command line interface (CLI). The CLI allows users to interact with the system using a Java Jar file which can be moved between machines independent to the source code. It currently supports two commands: ‘plan’ and ‘testgen’. The ‘plan’ command allows a user to pass in JSON input and receive a JSON plan output. There is also the option to run the execution simulation following the plan’s completion to test the system’s adaptation to perturbations. The ‘testgen’ command allows a user to provide a JSON initial state and goal state and receive a generated JSON output test case. The user must also specify how many tasks they would like created for the test case. Both commands support different options regarding runtime variables, log verbosity, and output location. The CLI User Guide in Appendix D details all the options.

System Logger

The System Logger is responsible for logging significant events that happen throughout the parsing, planning, and simulated execution portions of the overall system. The System Logger is also responsible for logging errors such as if a perturbation occurs and the system is not able to create a new plan from the perturbed state. The logger is enabled through the CLI and simply sets a flag variable to true. If this flag is set to true all logging functionality throughout the entire system will be enabled. The logger also has the ability to print to the console as execution happens; this can also be enabled through the CLI in a similar way to the log flag. If the System Logger is enabled, it will create a new log file for every runtime session and the name of that file will be representative of the date and time at which the runtime state was executed. All entries in the log file are also time stamped with the corresponding time and date of the entry. Every entry in the log file contains the name of the component that generated that entry.

Planner Output

JSON Output

The Promethean Planning System contains an output module implemented via the `Output` interface. The current system utilizes the `Output` interface to export planning and simulation data for post-processing and visualization. The goal behind designing the system output as an interface was to provide flexibility in the types of output (e.g. JSON, XML, or other data representations) while requiring minimal effort to integrate it into the planning system. The programmer simply needs to implement the `Output` interface provided and the system will handle any subsequent integration.

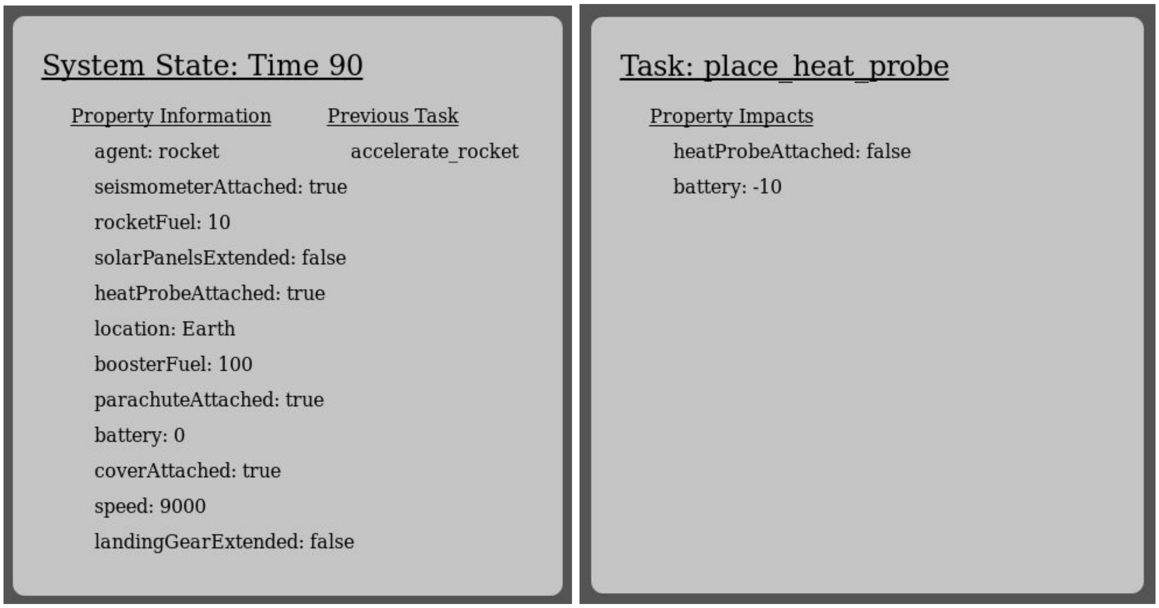
The Promethean Planning System currently implements an output capability, called `JSONOutput`, using the `Outpu`t interface. This implementation leverages the open source Jackson library to serialize the Planner's java objects into a JSON representation. The result is then stored in a designated file. The `JSONOutput` implementation can handle any Java object in the Planner but it is primarily used for storing key components such as `PlanBlocks` and `SystemStates`.

Resource and Execution Plots

Included in the Promethean repository is a resource and execution visualization. The visualization is written in d3.js, and requires Python 3 to run the web server. All the resources mentioned in this section are in the "visualization" directory of the main promethean.ai project. There is a bash script, named "run.sh" which is executable by default, which can be run to start the web server on localhost:8000. The default file which is being parsed into the visualization is Plan-mission_to_mars_expo.json, which can be changed on line 11 of the chart.js file.

By default, the visualization will load the static representation of the execution plot. Each green bar at the top represents an executed task, and the rust colored bar directly below a given task is the system state the Planner will be in after the task is executed. It is possible to click on either a task or system state bar to open a secondary pane containing information on the selection, pictured in **Figure 5**.

Figure 5



System State: Time 90

Property Information

- agent: rocket
- seismometerAttached: true
- rocketFuel: 10
- solarPanelsExtended: false
- heatProbeAttached: true
- location: Earth
- boosterFuel: 100
- parachuteAttached: true
- battery: 0
- coverAttached: true
- speed: 9000
- landingGearExtended: false

Previous Task

- accelerate_rocket

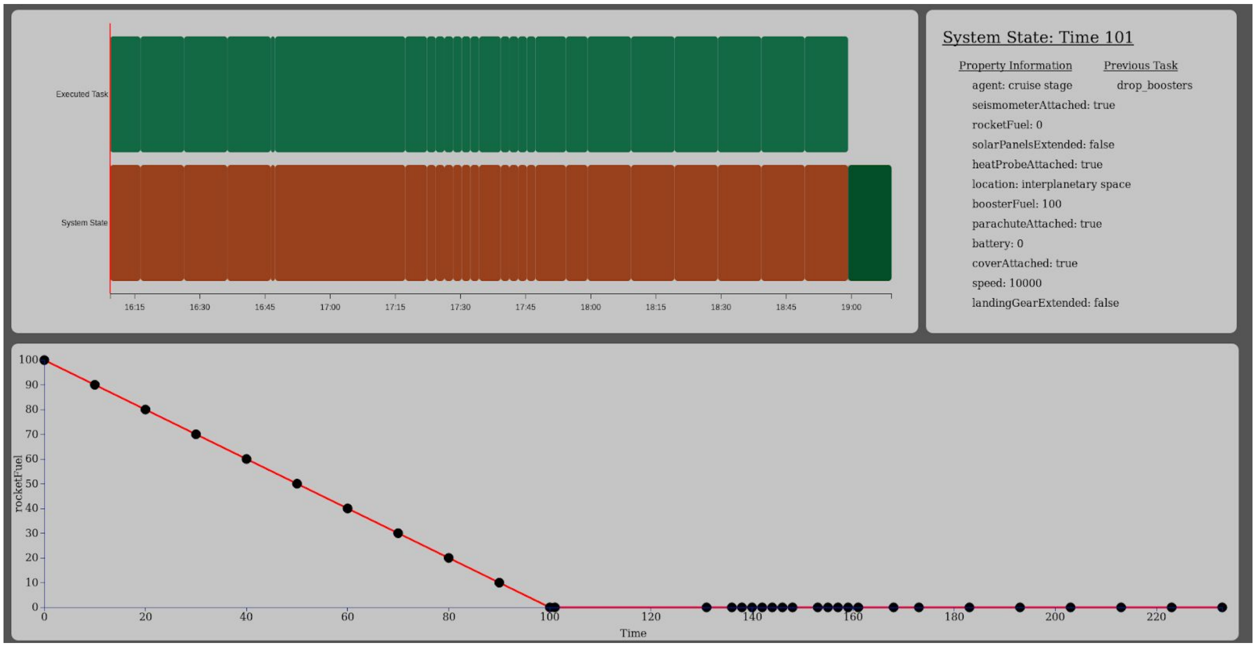
Task: place_heat_probe

Property Impacts

- heatProbeAttached: false
- battery: -10

Each line below the “Property Information” or “Property Impacts” headings can be clicked to open up a graph below the visualization which shows the change of value of the properties over the course of execution. The plot of values has a circle at the time of each task execution to make it easier to tell when a value changes. Each of these circles can be hovered over to display the value at that given time of execution. The fully expanded visualization can be seen in **Figure 6**.

Figure 6



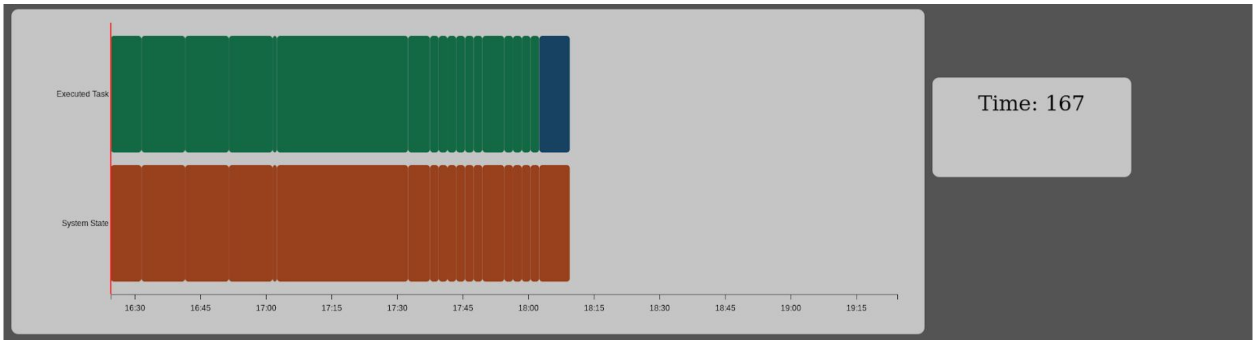
System State: Time 101

Property Information	Previous Task
agent: cruise stage	drop_boosters
seismometerAttached: true	
rocketFuel: 0	
solarPanelsExtended: false	
heatProbeAttached: true	
location: interplanetary space	
boosterFuel: 100	
parachuteAttached: true	
battery: 0	
coverAttached: true	
speed: 10000	
landingGearExtended: false	

There are also two checkboxes at the top of the page labeled “Animate,” which defaults to unchecked, and “Start Line” which defaults to checked. The start line toggles on and off a red line on the plot to show where execution begins. This is useful for visualizations which don’t have many tasks/states, since the population of the bars will begin somewhere in the middle of the chart.

The “Animate” box, when checked, will remove all additional information panels and replace the System State / Task panel with a clock. This will begin ticking upwards from 0, and animate the addition of tasks and system states to the visualization. Tasks will pop in with a blue color, indicating that they are being executed, and will turn green once execution is finished. Because of the way Javascript runs a single thread at a time, and the handling of asynchronous functions (which are needed for the animation), the interaction function which is called to allow clicking on various elements cannot be executed until the animation is complete. Because of this, the page will be refreshed to loop the animation when it is finished, and interaction is initiated by un-checking the “Animate” checkbox at the top of the page. A still image of the animation can be seen in **Figure 7.**

Figure 7



System Testing

Test Plan Overview

The system will be run with minimal supervision, and its components may be modified in the future, so it is important that the code is reliable and efficient. To ensure that the Planner functions this way, the system was tested as a whole with a test case generator, tested with unit tests, and analyzed for code smells and bad practices that could lead to bugs in the future.

JSON Test Case Generator

Packaged along with the Promethean Planning System is the JSON Test Case Generator. This is available to provide test inputs for the Planner. The source for the generator can be found in the TestCaseGenerator directory of ai.promethean.

At a minimum, the `TestCaseGenerator` constructor requires an input `SystemState`, a `GoalState` to plan towards, and a boolean `optimalPath` which defines whether an optimal path through the generated plan should be guaranteed or not. More arguments that can be provided but are not required include: `numTasks`, which defaults to 15 with no argument, specifies how many tasks should be generated in the output JSON; `filename`, which defaults to “generated_test”, allows the output file to have a specified name.

The generator can be called using the `generateTestCase` method, which will perform all the necessary steps to create a valid test case using the internal private methods. This will return an ArrayList of Task objects which will contain the specified number of randomly generated tasks. This ArrayList can then be fed to the `testCaseToJSON` method, which will output a valid JSON file for the generated test case with the filename specified in the `TestCaseGenerator` constructor.

There are various private internal methods which are used in the pipeline to create a test case. The `computePropertyDeltas` method is used to compute the difference in properties between the initial and goal states, and is returned as an ArrayList of `PropertyDelta` objects. These deltas are then passed to and used in the `createCriticalPathTasks` function, assuming the `optimalPath` argument to the `TestCaseGenerator` constructor is true. `createCriticalPathTasks` will create a “critical path” through the network, which will be the guaranteed shortest path from the start to the goal. It is important to note that there will always be a task named “charge” included in the dictionary of generated tasks, regardless of whether the critical path is generated or not. The “charge” task will have a positive delta of +10 to a Property named “Battery”.

Once the critical path is generated, if applicable, the `createRemainingTasks` method is called. This is where the remaining tasks are randomly generated, adding tasks until the original argument number of tasks are created. Inside `createRemainingTasks`, for each task generated, a random duration in the range [1, 50] will be created as well. The generated task will then randomly select up to 5 properties to impact. The list of selected properties is iterated over and a delta value is created. If the selected property is a `BooleanProperty` type, the task will assign the value to True or False randomly. If the selected property is a `NumericalProperty` type, a random number in the range [1, 50] is generated and then a “coin” is flipped to see if the delta will be positive or negative. Once all of the property impacts are generated, the task is added to the ArrayList of randomly generated tasks which is eventually returned.

Once all the tasks are generated, the ArrayList of `Task` objects can be passed to the `testCaseToJSON` method. This is a modular method which converts the ArrayList of `Tasks` into a valid JSON file which is compatible with the defined Promethean JSON Language Task schema. This takes advantage of the `JSONArray` and `JSONObject` from the org.json library.

It is important to note that this method can be rewritten to adhere to any schema which is used in the future of the Promethean project, or entirely scrapped and replaced with a new module used to output to a completely different language. In the development of this `TestCaseGenerator`, it was important to leave the output as a module that can be replaced so the main logic of the generator can be reused in many different applications in the future.

Performance Metrics

This Planner is designed to be run on spacecraft which are considerably less powerful than the machines used for development. To test the Planner in a more realistic environment, it was run on a Raspberry Pi 3B+ with 1GB of RAM and a 1.4Ghz ARM processor. The Planner was able to run all of the sample test cases using this hardware. The memory allocated to the JVM was also limited to 750MB and a test case with no critical path was run. With these constraints, the Planner ran for 5 minutes and expanded over 4.5 million states before running out of memory.

Testing and Continuous Integration

Unit Tests

Unit tests were written using JUnit to test the main functionality of each class in the system. These unit tests address edge cases and check for bugs and unhandled errors. Running existing unit tests after refactoring or changing a component of the system also ensures that the changes made have not broken existing functionality that the rest of the system uses.

Travis Continuous Integration

Travis continuous integration was used with the Promethean remote repository. The Travis pipeline is fairly simple. For every push to the remote repository, Travis automatically lints the code, builds the project from scratch using the settings defined by build.gradle, and runs the unit tests. Travis also checks pull requests for bugs and errors by building and testing the merged version of the code. A pull request will only be able to merge if it passes Travis's checks. This automated testing and building is done using gradle which was used for build and dependency management.

Linter - Code Quality

A linter called SonarQube also checks the codebase for code smells, typos, and bad practices. This linter is run automatically by Travis via gradle. The results of these scans are uploaded to SonarCloud where contributors can view them. Promethean also uses the SonarCloud plugin on our repository to make linter results for pull requests easier to view.

Sample Test Case

An easy to follow test case loosely based on the Mars InSight Lander was created to demonstrate how the Planner works. The plan should launch a rocket from Earth, travel to Mars, decelerate the lander with a parachute and thrusters, land on Mars, and place the heat probe, the seismometer, and the seismometer cover on the ground. To model this, an initial state, goal state, and a list of tasks was defined which can be found in `mission_to_mars.json`. This file defines the following properties:

Property	Property Type
agent	string
location	string
speed	numerical
battery	numerical
rocketFuel	numerical
boosterFuel	numerical
solarPanelsExtended	boolean
landingGearExtended	boolean
parachuteAttached	boolean
seismometerAttached	boolean
coverAttached	boolean

heatProbeAttached	boolean

The initial state is:

Property	Value
agent	“rocket”
location	“Earth”
speed	0
battery	0
rocketFuel	100
boosterFuel	100
solarPanelsExtended	false
landingGearExtended	false
parachuteAttached	true
seismometerAttached	true
coverAttached	true
heatProbeAttached	true
emergencyParachuteAttached	true

and the goal state is:

Property	Value
agent	“lander”
location	“Mars ground”
speed	0
solarPanelsExtended	true
landingGearExtended	true

parachuteAttached	false
seismometerAttached	false
coverAttached	false
heatProbeAttached	false

The possible tasks are:

Task
accelerate_rocket
drop_boosters
cruise
decelerate_with_parachute
extend_landing_gear
decelerate_with_thrusters
land
extend_solar_panels
charge_battery
place_seismometer
place_heat_probe
place_seisemometer_cover
deploy_emergency_parachute

For readability, these tasks have been listed in the order in which they will be executed. This is not a requirement and does not affect the Planner's output.

The Planner returns a plan where accelerate_rocket is scheduled 10 times to increase the rocket's speed to escape velocity. Once that has been reached, the Planner schedules drop_boosters changing the agent from "rocket" to "cruise stage" and the location from "Earth" to "interplanetary space." The plan then schedules the cruise task which changes the agent to “landing capsule” and the location to “Mars atmosphere.” The speed property is still very high, so the Planner schedules decelerate_with_parachute followed by 6 instances of decelerate_with_thrusters. decelerate_with_parachute changes the agent to “lander.” It then schedules extend_landing_gear since that is a requirement to land and decreases its speed further by scheduling 4 more instances of decelerate_with_thrusters. The speed should be 0 after running all of this, so the land task is scheduled changing the location to “Mars ground.” The Planner then schedules the extend_solar_panels task, charge_battery, and place_heat_probe. This one instance of the charge_battery task does not fully charge the battery, but the Planner is designed to optimize based on time. Only one instance of charge_battery is needed to charge the battery enough to place the heat probe. Next, the Planner schedules two more charge_battery tasks followed by place_seismometer and place_seismometer_cover.

There is another test case that adds a perturbation to this plan. The perturbation changes parachuteAttached to false to simulate a parachute failure. The planner detects this change and replans. It executes decelerate_with_thrusters until thrusterFuel is 0. At this point, it deploys the emergency parachute which slows the lander down to 0 but adds significantly more time to the plan. After this, the lander lands and the plan proceeds the same as the plan above.

If this planning system were used to plan the takeoff and landing of an actual spacecraft, the states, properties, and tasks would be much more complicated. These test cases are not as comprehensive as a real-world planning problem definition, but they do serve as simple, easy to follow examples that show how states and tasks can be modelled using our language and how the Planner schedules those tasks.

Further Improvements

One of the main design goals of the entire Promethean system was to have the components be as robust and modular as possible. This modular design allows for future changes and improvements to be easy to implement and the flexibility of the system will allow core components to be swapped out while overall functionality remains the same. The work put into Promethean has been extensive, but the overall time limit for the project has limited what was able to be implemented. The list below details additional improvements which could be made to Promethean if there was more time to work on the project.

- Implementing D* search as a possible search algorithm for the Planner. See the Planning Algorithm Trade Study in Appendix F for the discussion of D*'s potential benefits in planning.
- Further work on the heuristic function and how optimizations are handled. This is a vague goal that can almost always be improved upon.
- Create a user interface to abstract the JSON language creation process from users and give a graphical representation of the CLI.
- Serialize Java Objects to allow the Planner to save currently planned system states and restart planning from those states at a given time.
- Create a database to permanently store the task knowledge base for a specific system.

Conclusion

As the science behind space travel grows exponentially, autonomous spaceflight is an inevitable solution. The goal of the Promethean Planner System was to prototype this idea and provide a foundation for further improvements.

As it is now, the current system provides a domain agnostic JSON language implementation that can model any range of problem spaces. The system's Planner uses the A* Search Algorithm in order to find an optimal path between the executive's initial state and goal state. Lastly, the Executive module provides a planning simulation to model how a spacecraft would execute a plan. The Executive additionally handles system perturbations and replans on spot. Most importantly, Promethean’s modular archectural design allows future programmers to easily create, modify, and extend all components of the system.

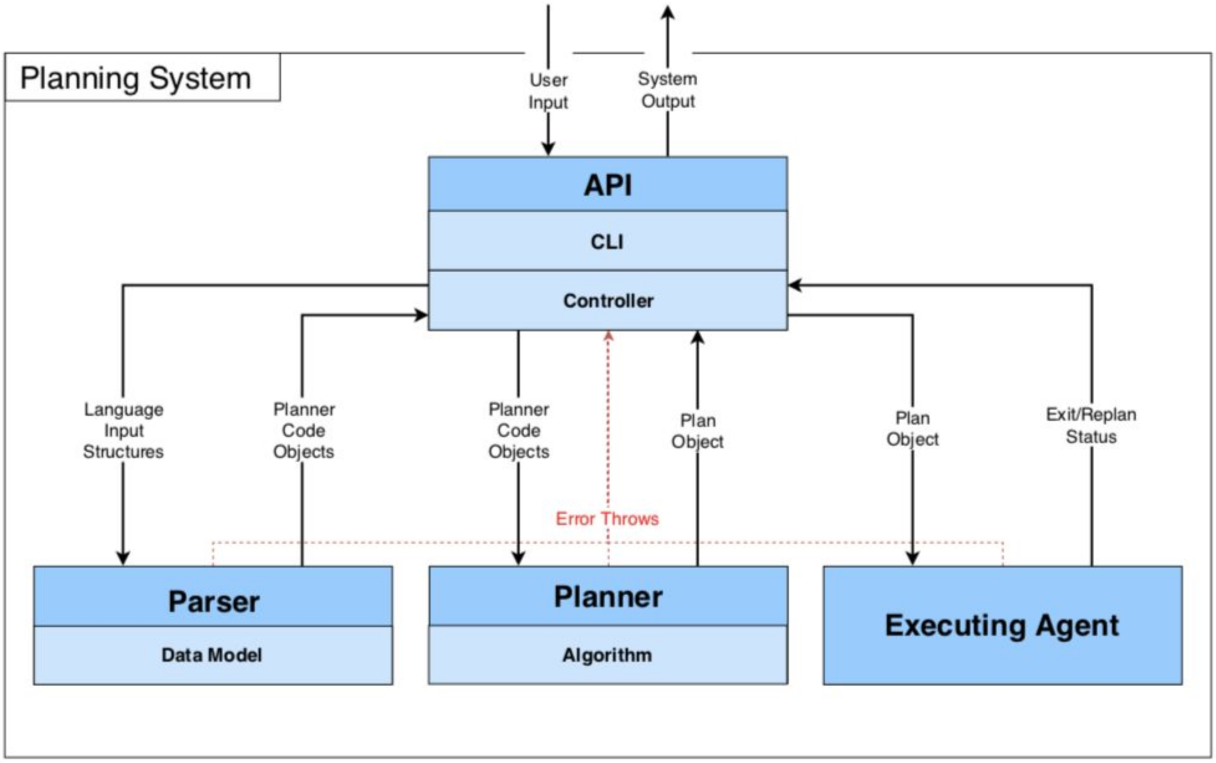
This system is just the beginning. It will act as a proof-of-concept for NASA and is the first step in innovating how we explore space. Autonomous space exploration is the future, and we are happy to be a part of it.

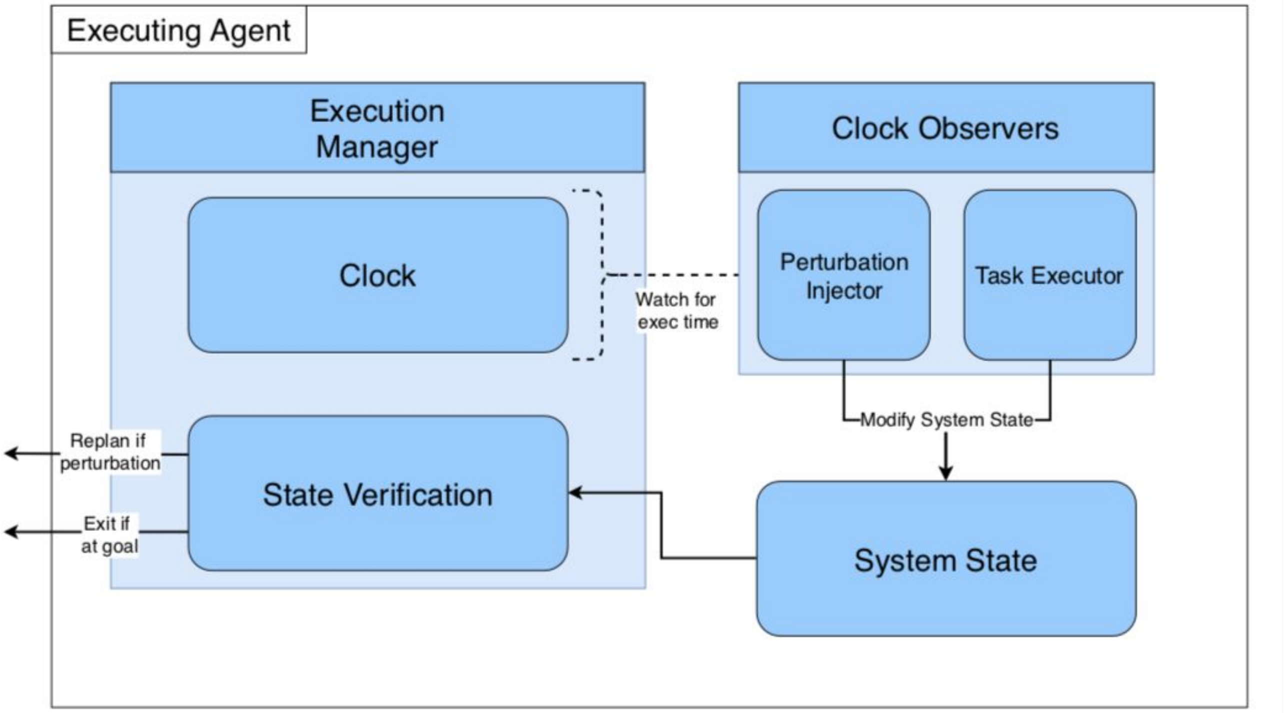
Acknowledgements

We would like to thank our project sponsor from JPL, Marcel Llopis, and our teaching assistant from CU, James Watson, for all of their help in this project. Working on Promethean has been a wonderful experience and we are very grateful for all of the invaluable time and mentorship provided by Marcel and James.

Appendix A: Architecture Diagrams

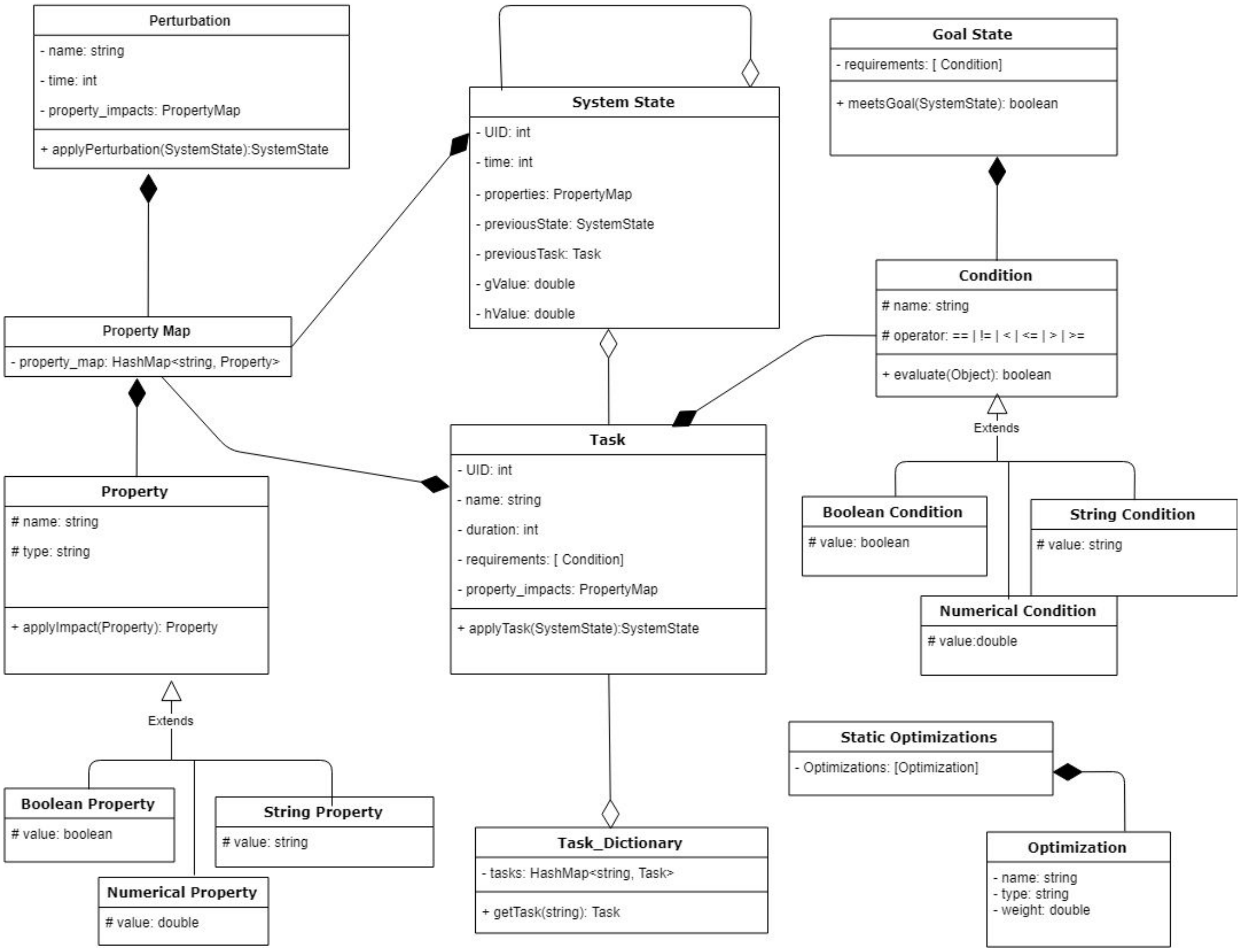
PROMETHEAN


Planning System
User Input
System Output
API
CLI
Controller
Language Input Structures
Planner Code Objects
Planner Code Objects
Plan Object
Plan Object
Exit/Replan Status
Error Throws
Parser
Data Model
Planner
Algorithm
Executing Agent


Executing Agent
Execution Manager
Clock
Watch for exec time
Clock Observers
Perturbation Injector
Task Executor
Modify System State
Replan if perturbation
Exit if at goal
State Verification
System State

Appendix B: Data Model Class Diagrams

PROMETHEAN


Perturbation
- name: string
- time: int
- property_impacts: PropertyMap
+ applyPerturbation(SystemState):SystemState
System State
- UID: int
- time: int
- properties: PropertyMap
- previousState: SystemState
- previousTask: Task
- gValue: double
- hValue: double
Goal State
- requirements: [Condition]
+ meetsGoal(SystemState): boolean
Property Map
- property_map: HashMap<string, Property>
Condition
# name: string
# operator: == | != | < | <= | > | >=
+ evaluate(Object): boolean
Extends
Task
- UID: int
- name: string
- duration: int
- requirements: [Condition]
- property_impacts: PropertyMap
+ applyTask(SystemState):SystemState
Property
# name: string
# type: string
+ applyImpact(Property): Property
Boolean Condition
# value: boolean
String Condition
# value: string
Numerical Condition
# value:double
Extends
Boolean Property
# value: boolean
String Property
# value: string
Numerical Property
# value: double
Task_Dictionary
- tasks: HashMap<string, Task>
+ getTask(string): Task
Static Optimizations
- Optimizations: [Optimization]
Optimization
- name: string
- type: string
- weight: double

Appendix C: JSON Input User Guide

PROMETHEAN

Introduction:

The Planning Simulation System relies on input files fed through the API to the parser. These input files represent the problem space language and will give the Planner all the information it needs to operate correctly. The input JSON file will consist of several entries representing the necessary items for the Planner to execute correctly. These include an initial state, a goal state, a list of all possible tasks and their respective effects on the state, a list of desired optimization and their weight, and all the perturbations the simulation will encounter. Below are descriptions of those specific steps. All JSON schemas adhere to JSON draft-06.

Initialization:

As the language stands in this current draft, the input is designed to have a single “Init” JSON object. The schema for the Init object can be found in Init_schema.json.
The fields for the Init JSON object include:

- Field Name: initial_state, Type: InitState JSON object, Required Field
- Field Name: goal_state, Type: GoalState JSON object, Required Field
- Field Name: tasks, Type: JSON array of Task JSON objects, Required Field
- Field Name: optimizations, Type: JSON array of Optimization JSON objects, Optional Field
- Field Name: perturbations, Type: JSON array of Perturbation JSON objects, Optional Field

See the below sections for a more detailed description of the above Initialization fields.

Initial State:

The initial state field describes all of the properties that represent the beginning state of the system before the Planner’s execution. There can only be one Initial State in the Init JSON. The schema for the initial state can be found in InitState_schema.json.
The fields for the InitState JSON object include:

- Field Name: properties; Type: JSON array of JSON Property objects; Optional Field
    - This field is for the user to input current properties of the state.
    - The Property object adheres to the Property_schema.json
    - The Property object has the following fields:
        - Field Name: name; Type: String, Required Field
            - This field is for the name of the property.
        - Field Name: value; Type: Number/String/Boolean, Required Field
            - This field is for storing the value of the property. This value can be a number, string, or boolean.

Goal State:

The goal state field describes the requirements that must be met for the goal state to be achieved in the planning system. There can only be one goal state. The JSON schema for the goal state can be found in GoalState_schema.json. The fields for the GoalState JSON object include:

- Field Name: requirements; Type: JSON array of JSON Condition objects; Optional Field
    - This field is for the user to input the required property preconditions in order for the goal state to be met.
    - The Condition object adheres to the Condition_schema.json
    - The Condition object has the following fields:
        - Field Name: name; Type: String, Required Field
            - This field is for the name of the condition.
        - Field Name: operator; Type: String, Required Field
            - This field denotes the precondition operator.
            - For a numerical value, the valid operators are: “>” , “>=”, “<”, “<=”, “==”, and “!=”.
            - For a string or boolean value, the valid operators are: “==” and “!=”
        - Field Name: value; Type: Number/Boolean/String, Required Field
            - This field is for storing the value of the condition.

Tasks:

The tasks field in the Init JSON object consist of an array of Task JSON objects. Tasks are implemented adhering to the Task_schema.json file. The order of tasks does not matter and an individual task JSON entry should be entered for every possible task. Property Impacts should be reflective of the effects the task will have on the properties of the state.
The fields for the Task JSON object include:

- Field Name: name; Type: String, Optional Field
    - The name field denotes the name of the Task that is being initialized.
- Field Name: duration; Type: Integer, Required Field
    - This field denotes the duration of the given task. This duration is currently represented as internal system “ticks”.
- Field Name: requirements; Type: JSON array of JSON Condition objects; Optional Field
    - This field is for the user to input the property preconditions to be met in order for the task to execute.
    - The Condition object adheres to the Condition_schema.json
    - The Condition object has the following fields:
        - Field Name: name; Type: String, Required Field
            - This field is for the name of the condition.
        - Field Name: operator; Type: String, Required Field
            - This field denotes the precondition operator.

      - For a numerical value, the valid operators are: “>” , “>=”, “<”, “<=”, “==”, and “!=”.
      - For a string or boolean value, the valid operators are: “==” and “!=”
    - Field Name: value; Type: Number/Boolean/String, Required Field
      - This field is for storing the value of the condition.
- Field Name: property_impacts; Type: JSON array of JSON Property objects; Optional Field
  - This field is for the user to input the changes in properties impacted by the task.
  - The Property object adheres to the Property_schema.json
  - The Property object has the following fields:
    - Field Name: name; Type: String, Required Field
      - This field is for the name of the property.
    - Field Name: type; Type: String, Optional Field
      - This field denotes whether the property impact is a delta or an assignment.
      - The “assign” keyword signifies that the task will change the old corresponding property to the current value field. Boolean and String properties are inherently assignments.
      - The “delta” keyword signifies that the task will add the current property value to the old property value. If the user wants to subtract an amount from the original property, the value field must be negative.
      - If no type field is given, the system defaults to “assign”.
    - Field Name: value; Type: Number, Required Field
      - This field is for storing the value of the property. This value can be a number, string, or boolean.

Optimizations:

The optimizations field in the Init JSON consist of a JSON array of Optimization objects. This field is optional because optimizations are not necessary to run the Planner. Optimizations are implemented adhering to the Optimization_schema.json file. Optimizations are property names in which the Planner could minimize or maximize its values while planning. As a general statement the values we will be expecting to minimize are continually increasing properties such as time. Properties that will be maximized can be thought of on a 0-100 scale, such as energy level. So far, the Planner only has one special optimization which is the execution time of the plan. If the user wishes to optimize on time, they must put “duration” in the Optimization name field. Weights determine the percentage of a property’s value that will end up contributing to the overall path-cost value. Numeric properties that are not given an optimization object will default to have zero weight and will not be considered in the path-cost calculation.
The fields for the Optimization JSON object include:

- Field Name: name; Type: String; Required Field
  - The name denotes which property we are optimizing on.
- Field Name: type; Type: String; Required Field

    - This field denotes whether the optimization is to minimize or maximize the property value.
    - This field requires words that include “min” or “max” prefixes and it is case insensitive.
- Field Name: weight; Type: Double; Required Field
    - The weight field denotes the weight (between 0-1) of the property to optimize over in the system.

Perturbations:

The perturbations field in the Init JSON consist of a JSON array of Perturbation objects. This field is likewise optional because perturbations are external to the planning system. Perturbations are implemented adhering to the Perturbation_schema.json file. Property Impacts will be reflective of the effects the perturbation has on the properties of the state. Since the system time is not an explicit property in a System State, the user must use the keyword “time” in the property_impact name if they want to include a time perturbation.

- Field Name: name; Type: String; Optional Field
    - This field is for the user to input the name of the perturbation.
- Field Name: time, Type: Int; Required Field
    - The time field denotes when the perturbation will be injected into the simulation.
- Field Name: property_impacts; Type: JSON array of JSON Property objects; Optional Field
    - This field is for the user to input the property impacts of the perturbation.
    - The Property object adheres to the Property_schema.json
    - The Property object has the following fields:
        - Field Name: name; Type: String, Required Field
            - This field is for the name of the property.
        - Field Name: type; Type: String, Optional Field
            - This field denotes whether the property impact is a delta or an assignment.
            - The “assign” keyword signifies that the perturbation will change the old corresponding property to the current value field. Boolean and String properties are inherently assignments.
            - The “delta” keyword signifies that the perturbation will add the current property value to the old property value. If the user wants to subtract an amount from the original property, the value field must be negative.
            - If no type field is given, the system defaults to “assign”.
        - Field Name: value; Type: Number/String/Boolean, Required Field

This field is for storing the value of the property. This value can be a number, string, or boolean.

Appendix D: CLI User Guide

PROMETHEAN

The CLI supports two commands: ‘plan’ and ‘testgen’. Run either command using the Java jar file. To run the v1.0.0 release:

java -jar PROMETHEAN-v1.0.0.jar <command>

Commands:

- **`plan -`** create (and optionally execute) a plan given input json
  - **`-i, --in-file “path/to/file.json”`**
    - JSON input file for planning system. Must contain at least an initial state, a goal state, and list of tasks
  - **`--in-string “{json: string}”`**
    - JSON string input for planning system
  - **`-x, --execute`**
    - Simulate execution after planning (including perturbations)
  - **`-s <double>, --stop <double>`**
    - Maximum runtime in seconds
  - **`--[no]clf`**
    - If clf is enabled, the planner will not consider the stop time until backtracking occurs
  - **`-v, --verbose`**
    - Enables logging
  - **`-l, --logs`**
    - Write logs to specific directory
  - **`--print-logs`**
    - Print any logs to the command line (enables verbose by default)
  - **`--plan-output`**
    - Directory to write generated plans (planner)
  - **`--states-output`**
    - Directory to write simulated states (exec agent)
- **`testgen -`** generate a test input file given initial and goal states
  - **`-i, --in-file “path/to/file.json”`**
    - JSON input file for the test generator. Must contain an initial state and a goal state
  - **`--in-string “{json: string}”`**
    - JSON string input for test generator

- **`-n, --num-tasks <int>`**
    - Number of tasks to generate for test case
- **`-p, --perturbations`**
    - Number of perturbations to generate for test case
- **`--optimal-plan`**
    - If true, generated test will contain a specific, optimal plan
- **`-v, --verbose`**
    - Enables logging
- **`-l, --logs`**
    - Write logs to specific directory
- **`--print-logs`**
    - Print any logs to the command line (enables verbose by default)
- **`-o, --output`**
    - Output filename for generated test case

Appendix E: Planning Language Trade Study

PROMETHEAN

Introduction & Background

When designing artificially intelligent planning algorithms, a well-defined language must be used to provide the necessary information to the system. Types of information provided may be pre-conditions, post-conditions, event durations, boolean operations, initial states and goal states. Many frameworks exist which are widely used for these types of problems already, each providing some subset of the requirements which are desired for our use case.

The project is to design and produce an AI task planner to be utilized onboard spacecraft -- satellites, rovers, etc. -- for the vehicle to make informed decisions without the need to communicate back to Earth. The language chosen in this trade study will be used to define the various states, conditions, transitions, and perturbations needed for the main decision-making algorithm. The language will be general enough to represent any planning problem that takes this form. Another goal is for the language to be easily expanded and improved upon delivery to JPL.

In this case, the desired language must contain all 10 of the defined “must have” criteria. There are also certain “want” criteria which we would prefer to have but are not integral to the success of the project. This trade study will look at three potential language structures - PDDL, RDF, and XML/JSON - to assess how well they fit (or can be expanded to fit) this use case. A comparison matrix will be used to find the criteria that each language supports, which will motivate the final decision of the language with which to move forward.

Introduction to Criteria and Weights

Must Have Criteria:

- **Tasks must contain Pre-conditions, Parameters, Post-Conditions, and Durations:**
    - A way to represent tasks or actions which the spacecraft can perform must be a part of the language. In order for a task to be represented it must have preconditions, parameters, post-conditions, and durations. Pre-conditions will be sets of values to be met before the task can take place. Parameters will be variables (boolean or numerical) that must be met in the pre and post conditions. The post-conditions will be a set of assignments of what the parameter values will become when the task is complete. Finally, duration must be represented as a time of how long the task will take to complete and thus when assignments should happen for the post-condition.

- **Language must be able to model an initial state and a goal state that contains optimizations:**

  - Autonomous systems must know what state they are starting in and what state they are trying to achieve. For this reason, our language must be able to model both initial and goal states. Initial states will have pre-conditions and specific assignments to a set of variables while goal states will be a set of conditions that the spacecraft must meet in order to achieve that goal state. The goal state will also contain information on what priority of optimizations we have.
- **Can support boolean, integer, and decimal values:**
  - Parameters of states, conditions, and tasks will have associated values. The language must allow these values to be boolean, integers, and decimals to support the assignment/comparison functionality.
- **Can support “and”, “or”, and “not”:**
  - Logical conjunction, disjunction, and negations can be used to compile and modify conditions that make up a state. “And”, or conjunction, is somewhat implicit if a state has a list of conditions/parameters that must all be met. “Not”, or negation, can often be implemented using inverses. “Or”, or disjunction, as well as exclusivity, are not implicit, but alternation is not necessary to represent the generic problem space. However, this language decision will set the foundation of the project, including its extensibility, so the explicit implementation of these logical operators will still be considered.
- **Must represent states; including names, types, and default values:**
  - State, a particular condition that the domain is in at a certain time, must be representable in our language. The states would need to have named variables that represent the descriptive attributes of this state, their data types and their values.
- **Contain information about perturbations: reassignments, timestamps, handle state changes:**
  - Perturbations are important because they allow for adaptation to unforeseen changes in a plan and improve the planning algorithm to take these sudden changes into account. The language should be able to support definitions of various perturbations, each to take place at some undetermined time. A perturbation may take the abstract form of “at X seconds into runtime, the Y value changes from A to B”.
- **Able to represent tasks as a set of tasks:**
  - The language should be able to represent a single task in terms of multiple subtasks. This criterion is ideal because it enables breaking up complex tasks into simpler subproblems. Representing subtasks within a task is inherently recursive and can be solved using a tree-structured language schema.
- **Can support >, ≥, <, ≤, equality and inequality operators:**
  - As specified, the language should support integer and decimal values. To compare these values in preconditions, post-conditions, and goals, the language should support numerical comparison operators. The user should be able to tie these comparison operators together using logical operators.

- **Can support multiple min, max for goals:**
    - It would be useful to optimize different attributes when defining a goal state, so goal states should support min and max statements for more than one attribute. It may not be possible to perfectly optimize all the desired attributes, so the language should be able to specify priorities for each optimization or take optimizations in an ordered list.

__Want Criteria:__

- **Ease of Implementation:**
    - **Existing Planners:**
        - Existing planners specifically designed for a language will provide resources that will ease extending our language and planning algorithm to encompass the desired problem space.
    - **Existing Parser:**
        - An existing parser for the language would eliminate the need to develop and test a new parser. Time is a significant constraint in this project and using a language with an existing parser would ease time constraints.
    - **Previous Use in Planning Systems:**
        - Due to inexperience in this type of work, having previously executed references could prove to be very beneficial. If a language has a history of use in existing planning systems, those planning systems could provide external guidance and help avoid implementation mistakes.
- **Extensibility:**
    - Extensibility concerns how easily we can modify the language. We may need to add features to the language as the project continues, or JPL may need to extend the language in the future. It would be helpful if the language were flexible, so we could easily make the changes we want. Ideally, a language (or it's parser) would have functionality that allows a programmer to modify the language's syntax or define new elements of the language. We will also look for existing tools that can be used to modify or extend the language.

- **Usability:**
    - The usability of the language encapsulates criteria relating to how easy and understandable the language is to use. Usability is very broad and depends on the specific product. In this trade study, we want to focus on how readable the language is. Can the language be represented in an understandable, human readable form? Additionally, we will be looking into whether the existing language implementation has tools that will help for easy maintenance in the future. Such tools could include user interfaces that allow a user to view and extend the language with little effort.

Criteria	Weights (Totals to 1)

Ease of Implementation	0.6
- Existing Planners	- 0.3
- Existing Parser	- 0.2
- Previous Use in Planning Systems	- 0.1
Extensibility	0.35
Usability	0.05

Analysis

PDDL and Expansion:

PDDL (Planning Domain Definition Language) is an AI planning language that was inspired by STRIPS (Stanford Research Institute Problem Solver) and ADL (Action Description Language). PDDL uses multiple components to create a language that describes various sets of tasks and actions. These components include objects, predicates, initial and goal states, and actions. Objects represent entities in the problem space. Predicates represent attributes of the objects being worked with. The initial and goal states represent a starting state and its associated parameters as well as an ending goal state that is attempting to be reached. Actions represent ways in which the state of the world can be changed[1].

Must Have Criteria:

- **Tasks must contain pre-conditions, parameters, post-conditions, and durations:**
    - PDDL supports pre-conditions, parameters, and post-conditions of various types. PDDL has support for durations but it is limited in the sense that PDDL only monitors durations for start time, process time, and stop time. This means that PDDL is unable to support pre-conditions that hold over specified intervals, a requirement in the language we are looking to design[2].
- **Language must be able to model an initial state and a goal state that contains optimizations:**
    - Initial state and goal state are defined in a problem file. Initial state is denoted with the ":init" tag and is a list of atoms that are true in the initial state. The goal state is denoted with the ":goal" tag and is a series of conditions that must be true for the goal state. An optimization can be specified in the problem definition using the ":metric" tag and specifying which attribute to minimize or maximize. Though this is part of the official PDDL grammar, not all existing planners support optimizations[3].

- **Can support boolean, integer, and decimal values:**
    - PDDL does support boolean parameters, in fact most parameters are defined as booleans. PDDL has support for integers and PDDL 3 has support for decimal values[4].
- **Can support "and", "or", and "not":**
    - PDDL supports these logical operators in goal specifications action pre-conditions, and action post-conditions. They can be chained together by nesting expressions using parentheses. Logical operators are used as asserts to check if an action can be performed or to check if the goal state has been reached. PDDL explicitly defines these operators in its grammar using the ":and", ":or", and ":not" tags. If the pre-conditions of an action are met, that action can be used to modify attributes of a state according to the post-conditions[5].
- **Must represent states; including names, types, and default values:**
    - As stated above, the initial state and conditions for the goal state are defined in the problem file. As actions are taken, the values specified in the initial state are modified. Once the values meet the conditions specified in the goal section, the planner has reached a goal state. Names are defined in the ":objects" section of the file, and actions are specified in the ":predicates" section of the file. The types of objects and parameters are specified using the ":types" keyword in the problem file. Values are assigned to objects by applying initial actions to the objects in the ":init" section[6].
- **Contain information about Perturbations; reassignments and timestamps, handle state changes:**
    - PDDL has limited support for perturbations. "Timed effects" are part of the PDDL grammar but only start and end time stamps are supported, no running duration timers are supported which would make it difficult to use for this project's desired purposes[7].
- **Able to represent tasks as a set of tasks:**
    - Actions in PDDL can be composed from smaller actions allowing cleaner structure with less repetition. Objects can also be assigned to sub-objects. Nested objects can be used to encapsulate data and allow for more abstract and easily manageable tree-based structures[8].
- **Can support >, ≥, <, ≤, equality and inequality operators:**
    - PDDL supports numerical comparisons using grammar similar to Lisp. It uses standard comparison operators such as "=", "<", ">", "<=", and ">=" followed by the two atoms or expressions whose values are being compared. The standard grammar for PDDL 3.1 does not specify an operator for inequality but wrapping an equality comparison in a "not" can be used instead[9].

- **Can support multiple min, max for goals:**

    - PDDL does have support for optimizations such as minimize and maximize, but PDDL is only capable of keeping track of one optimization at a time which does not suit the problem space of this project[10].

__Want Criteria:__

- **Ease of Implementation:**
    - **Existing Planners:**

Rating: 9/10

        - Numerous planners created to work with PDDL currently exist[11]. Due to the fact that there are so many existing planners this category receives a 9/10.
    - **Existing Parsers:**

Rating: 7/10

        - There are many existing parsers for PDDL with implementations in C++[12,] Python[13], and other languages. The previous existence of many PDDL parsers results in the score for this category being 7/10
    - **Previous Use in Planning Systems:**

Rating: 10/10

        - PDDL has extensive use in previous planning systems and it is one of the most prolific planning domain languages in use. There is extensive documentation that exists for PDDL. PDDL's extensive history in planning systems results in this score being 10/10.
            - OPTIC Planner: a temporal Planner developed by King's College London[14]
            - Fast Forward Planner: an open source domain independent Planner[15]
- **Extensibility:**

Rating: 6/10

    - Due to its deep history and widespread use, PDDL may prove difficult to extend. PDDL is twenty years old, so it has a very well-defined syntax. In order to modify that syntax, we would likely have to modify an existing PDDL parser or make our own. There are many open source PDDL parsers but modifying one of those parsers could be difficult. We would have to get familiar with an existing code base that likely was not made to be modified in the ways that we need. This is doable, but it would add significantly to our development time which is why this category receives a score of 6/10.
- **Usability:**

Rating: 7/10

    - Many programmers might already know PDDL since it is so widely used. If they do not, there are plenty of resources online to learn. PDDL also has a very minimalist syntax making it fairly easy to read. PDDL is syntactically and conceptually very different from most programming languages, however, so it may be difficult for a new programmer to learn without a lot of research which is why this category receives a score of 7/10.

Conclusion:

PDDL meets many of our criteria but has some major disadvantages. While there are plenty of planners and parsers for PDDL, standard PDDL does not support perturbations, timestamps, and multiple optimizations. We need these criteria, so we would have to modify an existing parser and planner to add them. Multiple optimizations, perturbations, and timestamps are major changes to PDDL's grammar that would warrant major changes to an existing parser and planner interface. It would take a while for the team to get familiar with these existing codebases and modifying them without understanding them completely could lead to bugs. Furthermore, most PDDL planners and parsers only support a subset of the language, so some of the criteria, like optimizing an arbitrary attribute[16], may not be met by the tool we decide to modify. Existing planners and parsers would provide a good starting point for our project but modifying a planner and parser could take more time than the other options in this trade study.

RDF:

Resource Description Framework (RDF) is an extension of XML that defines a set of resources, their properties, and their relationships with other resources. This framework emphasizes the linkages between resources and inherently creates a graph structure where each resource is a node and the relationship between resources is an edge[17]. Extensions of RDF include RDFS (Resource Description Framework Schema) and OWL (Web Ontology Language) which provide additional forms of representation and structure. RDFS and OWL allow a user to create classes and properties to define resources. In this way, RDFS and OWL can be used employ Object Oriented Programming approaches and best practices.

Must Have Criteria:

- **Tasks must contain Pre-Conditions, Parameters, Post-Conditions, and Durations:**
    - This criterion can be met by creating a "Task" class using RDFS. In this "Task" class we could define pre-conditions, parameters, post-conditions, and durations as properties of a task. Additionally, RDF has special constructs for time that we can use to represent durations. An arbitrary number of parameters can be represented using RDF Bag which is a construct that describes a list of items.
- **Language must be able to model an initial state and a goal state that contains optimizations:**
    - Similar to the Tasks criterion above, initial and goal states can be represented by creating a "State" class. Optimizations can be both represented as a property of the "State" class and its own class as well. Additionally, there is a construct called RDF Seq[18] that enforces an ordered list of elements. This can

be useful when representing optimizations because we can implement an optimization ordering that already exists in the RDF language.

- **Can support boolean, integer, and decimal values:**
  - RDF can support numerical values (integer and decimal) as well as strings using RDF XMLLiteral or RDFS Literal. Boolean values can be represented as strings ("true" and "false") using the literal types above, or by using XSD Boolean[19].
- **Can support "and", "or", and "not":**
  - RDF can support "and" and "or" using the RDF Bag and RDF Alternatives[20], respectively. RDF Bag represents an unordered list of values which we could use to imply conjunction between all the elements in the bag. RDF Alternatives represents a list of values which are ordered by some metric, which could be used for disjunction and be helpful with optimization. OWL supports intersectionOf and complimentOf, which allows for negation of comparisons/values.
- **Must represent states; including names, types, and default values:**
  - Since RDF supports classes, a "States" class or types of states could be created. It could include a string name, string types, and a list of values that that describe the state, which could be literals or classes themselves.
- **Contain information about Perturbations; reassignments and timestamps, handle state changes:**
  - Perturbations could also be represented in their own class that we define. RDF supports time values that could be used as timestamps. Additionally, OWL-time[21] is a library that has representations such as IntervalAfter and other interval-based time measurements, which could be especially helpful for perturbations. The changes in state could be represented by a StateBefore field and a StateAfter field, and the actual reassignment would be a functionality of the planner.
- **Able to represent tasks as a set of tasks:**
  - Because RDF has an inherent graph structure, it should not be difficult to extend the language such that a task can be represented as a set of tasks. An example of a way we could implement this structure in RDF is to create a "Task" class and create a corresponding "Subtask" subclass. Since RDF, RDFS, and OWL use Object-Oriented methodologies, "Subtask" would inherit properties of "Task". An additional approach is to make "Subtask" a property of a "Task" class that would contain a list of zero or more subtasks that the singular task is comprised of.
- **Can support >, ≥, <, ≤, equality and inequality operators:**
  - Constructs for greater than, greater than or equal, less than, less than or equal, equal, and not equal exists in the OWL Data Range Extension: Linear Equations[22]. In this extension, there are specific tags for each data

comparison. Additionally, this criterion would be easy to create from scratch; where gt, gte, lt, lte, eq, neq would all be subclasses to a “Relation” class.

- **Can support multiple min, max for goals:**
    - This criterion can be met by using OWL’s restriction tags. Restriction tags define a set of constraints that must be satisfied for a given class or property. There are additional constructs defined in OWL restriction tags, namely minCardinality and maxCardinality[23]. These tags can be used to define our minimum and maximum values and assignments in our goal state. Additionally, OWL restriction tags can be used to define other constraints for certain tasks or states in our language.

<u>**Want Criteria:**</u>

- **Ease of Implementation:**
    - **Existing Planners:**

Rating: 0/10

- There are no existing planners for RDF, as it is primarily used in the semantic web. It was considered that RDF may be compatible with XML planners, but we unfortunately were not able to find existing planners for XML either.

- **Existing Parsers**

Rating: 3/10

- RDF is based on XML, for which there are many parsers. We would be able to leverage an XML parser.

- **Previous Use in Planning Systems:**

Rating: 0/10

- RDF has not been publicly used in planning systems either, as again it is used more for the semantic web.

- **Extensibility:**

Rating: 8/10

- RDF, RDFS, and OWL all employ Object-Oriented approaches and style via their use of classes. For this reason, if Object-Oriented SOLID principles are utilized during the language development, then the language should maintain a high level of extensibility in the future. For example, if the language satisfies the Open/Closed SOLID Principle, then language extension should be just as easy as adding additional classes. When rating RDF’s extensibility from 1-10, we determined it was an 8. Our reasoning is that RDF provides the constructs that allow for easy extension, but it is still up to the developers to use proper Object-Oriented practices to utilize its inherent extensibility.

- **Usability:**

Rating: 10/10

- Since RDF, RDFS, and OWL are graph-structured languages by design, it is highly usable. There are many user interfaces available for both graphical visualization as well as editing. One such example is Protégé[34]- an open-source tool developed by Stanford- which acts as an ontology development environment. This application provides ways to create and visualize languages graphically, as well as create instantiations using an existing language. Additionally, there are many other similar visualization tools available online. We rated the usability of RDF, RDFS, and OWL a 10 out of 10 because it's intrinsic design and available tools make the language highly readable, understandable, and easy to visualize.

Conclusion:

RDF meets all of our required criteria, either through its built-in functionality or through the use of classes. It meets our want criteria of being extensible and usable and is able to make use of XML parsers. There are no existing planners and it has not been used in planning systems before. If our language and planner are truly decoupled, then this should not hinder our progress much. In fact, thanks to RDFs class functionality, it should be straightforward to convert from RDF classes to classes in the planner itself. There is a learning curve associated with learning the syntax/style of RDF, but as mentioned in the usability section there are many GUIs and documentation sites that should help. Our biggest challenge would be if during implementation, we learn that RDFs built-in functionality is not ideal for a certain piece and we need to build new functionality on top (beyond just classes), ensuring it stays compatible with the existing features. Overall, RDF is a viable option for our language.

JSON:

JavaScript Object Notation (JSON) structures data with an attribute and value. Though simple, this format is quite versatile. JSON is able to support values that are numbers, strings, lists, or objects[24]. This allows for representing data in a flexible way and is therefore good for our need of writing a planning language.

Must Have Criteria:

- **Tasks must contain Pre-conditions, Parameters, Post-Conditions, and Durations:**
  - JSON supports both objects and lists which makes representing parameters, conditions, and durations simple; for this reason, it is also convenient for the planner to handle. JSON represents objects through assignments of attributes and values, which in our use case would be parameters, conditions, or durations and their corresponding values or lists of values.
- **Language must be able to model an initial state and a goal state that contains optimizations:**
  - JSON can support this by creating objects and using initial value assignments for the initial state and a list of conditions that need to be met for the goal state. The goal state can be expanded upon with optimizations, which can be put in an ordered list of priority.

- **Can support boolean, integer, and decimal values:**
    - JSON has full native support for many variable types that are built in. JSON supports numbers, strings, booleans, arrays, objects, and a NULL variable.
- **Can support "and", "or", and "not":**
    - JSON has native support for "and" and "or". For "not" JSON "performs a NOT operation on the specified expression and returns the documents that do not meet the expression"[25] which is not a traditional implementation of NOT.
- **Must represent states; including names, types, and default values:**
    - JSON can represent all aspects of a state with its objects as label and value assignments. This includes names, types, and default values.
- **Contain information about Perturbations; reassignments and timestamps, handle state changes:**
    - JSON can handle numerical values and can thus give timestamps. We could have a separate object for perturbations that occur and at what time stamp they should happen. In the JSON we would just have to assign a value to a particular variable and make note of what time the assignment should occur.
- **Able to represent tasks as a set of tasks:**
    - JSON can represent tasks as sets of tasks quite easily since we can make an attribute of the task object another task; in this way, we can nest tasks within one another.
- **Can support >, ≥, <, ≤, equality and inequality operators:**
    - JSON has native support for all these logical comparisons. For example, JSON "Returns any documents that have a value that is less than the specified value in the set. It is also referred to as the less than comparison operator"[26.] This is different than traditional implementations, so an alternative for us would be to create a comparison operator attribute and have the values be the operator we want to use.
- **Can support multiple min, max for goals:**
    - This is something that is not explicitly part of JSON, meaning there is no native support for this feature. In order to implement this, we would assign a label max or min to the variable we are looking to maximize or minimize. Then add support in the code of the parser for syntax like this.

Want Criteria:

- **Ease of Implementation:**
    - **Existing Planners:**

Rating: 0/10

        - JSON was not originally meant to write a planning language, thus there are currently no existing planners that we found for JSON. For this reason, we must give JSON a 0 out of 10.
    - **Existing parsers:**
      Rating: 10/10
        - JSON has plenty of existing parsers that we can use straight out of the box. There are even some that can parse them directly into Java

classes for us. Due to the immense number of parsers that exist for JSON we rate this criterion 10 out of 10[27].

- **Previous Use in Planning Systems:**

Rating: 2/10

- JSON does have previous use in planning systems, but generally it is not as the representation of the language. Instead JSON is usually a helper for the system or just a stepping stone to the full planning language[28]. Since JSON does have some use in planning systems, but not in the way we will utilize JSON, we rate this criterion 2 out of 10.

- **Extensibility:**

Rating: 10/10:

- JSON is very extensible because the only thing that needs to be done to extend the language is add more JSON objects. For this reason, we determined that we would rate the extensibility a 10 out of 10.

- **Usability:**

Rating: 8/10

- The usability of JSON is quite high in general. We decided we would rate this at an 8 out of 10. This is because most people know how to write and read JSON objects and they can easily be translated into objects in any language that supports an object data type.

<u>Conclusion:</u>

JSON is a very flexible way to organize a planning language, because of this it is able to hit all of the required criteria. For our wanted criteria JSON is strong in extensibility, usability, and already has many existing parsers. For these reasons it's a very viable choice for our implementation. One last important fact to look at is how it would be implemented. Writing the language in JSON would be completely custom which would provide a lot of customizability to our structuring. Parsing the language should be easy since we can just use an off the shelf parser to do the whole job for us with trivial amounts of alteration. The parsed JSON will then go into the planner which will be language agnostic and therefore will have constant time across any language implementation.

XML:

XML stores information differently from JSON, stemming from the fact XML is a markup language while JSON is designed to represent objects. In XML everything is stored as an element which contains tags, with data stored within the tags. There is also the ability to define a schema for the data beforehand using the XML Schema Definition (XSD), which is useful for ensuring everything stored in the XML format is uniform in structure. Similar to JSON, all tag names are user-defined.

Must Have Criteria:

- **Tasks Must Contain Pre-conditions, Parameters, Post-Conditions, and Durations**
    - Since tags are all user-defined, it is possible to define all of the above in their own unique tags. XML does not support lists or arrays, so for example there would be a "preconditions" tag which contains many "condition" sub-tags defining the multiple preconditions needed for the action[29].
- **Language must be able to model an initial state and a goal state that contains optimizations**
    - It would be possible to define all the information needed on which to optimize in the XML body. The planner / parser would then be able to extract this information, along with the tags, and use it to optimize a plan of actions.
- **Can support boolean, integer, and decimal values**
    - In the schema file used to describe the XML document, it is possible to define a field as a string, decimal, or integer. The parser could take a "true" or "false" string and easily cast to a boolean, signified by the name of the field.
- **Can support "and", "or", and "not"**
    - There is a styling language for XML called XSL (eXtensible Stylesheet Language) which supports transformations of XML into other formats using XSLT (T = transformations). An element of the XSLT standard is XPath, which is used to navigate through elements in an XML document. Using XPath, specific nodes can be selected using the "and" and "or" operators. The "not" operator is not explicitly defined, but a "not equal to" operator (!=) is available, which can be used in a very similar fashion[30].
- **Must represent states; including names, types, and default values**
    - Names can be defined by the name of the tag holding data. Types and default values are enforced using XSD. In the schema definition for the document, both defaults and data types are explicitly defined. As mentioned above the only data types supported are strings, integers, and decimals, but the parser will be able to cast strings to different data types depending on the name of the tag[31].
- **Contain information about Perturbations; reassignments and timestamps, handle state changes**
    - Perturbations would be able to be implemented in a couple of different ways. One possibility is to have the possible perturbations embedded in the event definition, which would be as simple as adding a "pert" tag in the XML element which contains a time for the perturbation to happen and its effect. Alternatively, there could be an entirely separate file containing all possible perturbations for every event. The testing framework could then either randomly sample the perturbations or apply them at a predetermined time, defined within the pert definition. Regardless of the method of implementation for the perturbations, the planner could then take the change of state into account and change the plan accordingly.

- **Able to represent tasks as a set of tasks:**
    - It is possible to add a "neighbors" tag to all XML elements which contain the names of the neighboring events in a graph structure. This could be leveraged with a parser to represent the tasks as a graph.
- **Can support >, ≥, <, ≤, equality and inequality operators:**
    - Using XPath it is possible to use all the mentioned operators (>, ≥, <, ≤, =, !=)[32]
- **Can support multiple min, max for goals**
    - In the XML document, the goal state could have a list of variables in order of priority, along with an internal tag declaring whether the variable should be min'd or max'd. The parser and planner would then take this information into account when planning the optimal path of actions to take.

Want Criteria

- **Ease of Implementation**
    - **Existing Planners**
      Rating: 0/10
        - It does not look like there exist any publicly available XML based planners, giving this category a 0/10.
    - **Existing Parser**
      Rating: 10/10
        - There are XML parsers built into every modern browser. Java has a few which are included in the JVM. There's Expat, written in C. Python has the xml package. The abundance of XML parsers gives this category a 10/10[33].
    - **Previous Use in Planning Systems**
      Rating: 0/10
        - There is very little information publicly available for XML being used in planning systems, but its flexibility and ability to be extended makes it a very likely candidate to be used. Though it is straightforward to see how XML could be used in a planning system, the lack of public information on this gives this category a 0/10.
- **Extensibility**
  Rating: 10/10
    - XML is extremely extensible by definition – the X in XML stands for "extensible". Since all tags in an XML document are defined by the user, it is trivial to append a new tag which contains any information that needs to be added. This allows any existing document written in XML to be expanded upon with little to no effect on the existing information being represented. This gives the section a 10/10.
- **Usability**
  Rating: 7/10
    - XML is very readable, with all information about an element being contained in a single line. It is slightly less readable than JSON, though, since all data

must be enclosed in opening and closing tags, which add visual clutter. Overall XML is a highly useable language, giving this category a 7/10.

Conclusion:

XML meets all the criteria that are absolutely needed for this project while also fulfilling one of the “want” criteria by having many existing parsers available. This makes the development process slightly easier since it would be possible to leverage elements of an existing parser when designing our own. Since it will be relatively simple to implement the parser and language, the bulk of time can be moved towards designing the actual planning algorithm. XML is flexible, extensible, and meets all the needed criteria, making it a very reasonable language to move forward with.

Comparisons With Alternatives

Decision Criteria	**Language Options**			
Must Haves	**PDDL (Y/N)**	**RDF (Y/N)**	**XML (Y/N)**	**JSON (Y/N)**
Tasks Must Contain Pre-conditions, Parameters, Post-Conditions, and Durations	Y	Y	Y	Y
Language Must be able to model an initial state and a goal state that contains optimizations	Y	Y	Y	Y
Can support boolean, integer and decimal values	Y	Y	Y	Y
Can support "and", "or", "not"	Y	Y	Y	Y
Must represent states; including their names, types, and default values	Y	Y	Y	Y
Contain information for Perturbations: reassignments, and timestamp	N	Y	Y	Y

Able to represent tasks as a set of tasks	Y	Y	Y	Y
Can support >, ≥, <, ≤, equality and inequality operators	Y	Y	Y	Y
Can support multiple min, max for goals	N	Y	Y	Y
Wants	**PDDL Score (Out of 10)**	**RDF Score (Out of 10)**	**XML Score (Out of 10)**	**JSON Score (Out of 10)**
Existing Planners (0.3)	9	0	0	o
Existing Parsers (0.2)	7	3	10	10
Previous use in planning systems (0.1)	10	0	0	2
Extensibility (0.35)	6	8	10	10
Usability (0.05)	7	10	7	8
Want Score	**7.2**	**3.9**	**5.85**	**6.10**
Meets Must Have Criteria	**No**	**Yes**	**Yes**	**Yes**
Decision				**X**

Conclusion

The results of this trade study suggest that a JSON implementation is the best choice for the planning language. JSON can support all the must have criteria thanks to its simple, hierarchical key value structure. JSON supports creating objects with structure to represent states, conditions, perturbations, etc. Parameters can be represented as strings, integers, decimals, and booleans. JSON supports conjunction, disjunction, and negation through native concepts or through contrived tags. Values required for optimization and perturbations can also be represented using tags for max, min, and timestamps that we create. In addition to meeting our must have criteria, JSON meets several of our want criteria. Parsers for JSON are highly available, including parsers that map directly to classes in multiple programming languages. These parses use the hierarchical structure to create these classes, meaning custom keys and values will not require parser adaptation. JSON is very extensible thanks to its support of custom structure and is very usable due to its readability and documentation. As a result, JSON is the best choice for the language moving forward.

Regarding the alternatives, PDDL did not meet our must have criteria as it is unable to support perturbations and support for multiple min, max for optimizations. While PDDL is somewhat extensible and could be modified to support these, it was not extensible enough to justify the missing must haves. RDF does meet all our must have criteria. Additionally, its extensibility and usability make it a good choice for developing the language. However, it has no previous use in planning. RDF can leverage XML parsers, but they would likely need to be extended to support the custom syntax, which could prove challenging. As a result, its want criteria score is less than that of JSON. Lastly, XML is very similar to JSON. It meets all must have criteria and has plenty of existing parsers. It is very extensible thanks to its tag and value structure. It is also well documented, so somewhat usable, though slightly less readable than JSON. However, JSON is still slightly more usable, slightly more extensible, and has more flexible parsing than XML. As a result, JSON narrowly edges out XML by its want criteria score.

Works Cited

(1) Helmert, Malte. *An Introduction to PDDL*. 16 Oct. 15AD, www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf.

(2) Smith, David E. *The Case for Durative Actions: A Commentary on PDDL2.1*. www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/smith03a-html/PDDL-commentary.html.

(3) Haslum, Patrik. “Writing Planning Domains and Problems in PDDL.” *Australian National University*, users.cecs.anu.edu.au/~patrik/pddlman/writing.html.

(4) Kovacs, Daniel L. “Complete BNF Description of PDDL 3.1 (Completely Corrected).” *University of Huddersfield*, 2011, helios.hud.ac.uk/.

(5) Haslum, Patrik. “Writing Planning Domains and Problems in PDDL.” *Australian National University*, users.cecs.anu.edu.au/~patrik/pddlman/writing.html.

(6) “Writing Planning Domains and Problems in PDDL.” *Linköping University Department of Computer Science*, 2004, www.ida.liu.se/~TDDC17/info/labs/planning/2004/writing.html.

(7) Kovacs, Daniel L. “Complete BNF Description of PDDL 3.1 (Completely Corrected).” *University of Huddersfield*, 2011, helios.hud.ac.uk/.

(8) Kovacs, Daniel L. “Complete BNF Description of PDDL 3.1 (Completely Corrected).” *University of Huddersfield*, 2011, helios.hud.ac.uk/.

(9) Kovacs, Daniel L. “Complete BNF Description of PDDL 3.1 (Completely Corrected).” *University of Huddersfield*, 2011, helios.hud.ac.uk/.

(10) Kovacs, Daniel L. “Complete BNF Description of PDDL 3.1 (Completely Corrected).” *University of Huddersfield*, 2011, helios.hud.ac.uk/.

(11) Li, Xiao. “Classical Planning: #2 TDD a Minimum PDDL Planner.” *Classical Planning: #2 TDD a Minimum PDDL Planner*, xli.github.io/ai/2015/06/28/classical-planning-2-tdd-a-minimum-pddl-planner.html.

(12) “Aig-Upf/Universal-Pddl-Parser.” *GitHub*, Artificial Intelligence and Machine Learning Group - Universitat Pompeu Fabra, 7 June 2018, github.com/aig-upf/universal-pddl-parser.

(13) Bueno, Thiago P. “Thiagopbueno/Pypddl-Parser.” *GitHub*, 28 Nov. 2017, github.com/thiagopbueno/pypddl-parser.

(14) “OPTIC: Optimising Preferences and Time-Dependent Costs.” *Planning at KCL*, King's College London Department of Informatics,

(15) Hoffman, Jörg. “Fast-Forward.” *FF Homepage*, Saarland University, fai.cs.uni-saarland.de/hoffmann/ff.html.

(16) Haslum, Patrik. “Writing Planning Domains and Problems in PDDL.” *Australian National University*, users.cecs.anu.edu.au/~patrik/pddlman/writing.html.

(17) “XML RDF.” *Browser Statistics*, W3Schools, www.w3schools.com/xml/xml_rdf.asp.

(18) “RDF Schema.” *Same Origin Policy - Web Security*, Reuters Limited, www.w3.org/TR/rdf-schema/

(19) “XML Schema Datatype.” *Same Origin Policy - Web Security*, Reuters Limited, www.w3.org/TR/swbp-xsch-datatypes/

(20) “RDF Schema.” *Same Origin Policy - Web Security*, Reuters Limited, www.w3.org/TR/rdf-schema/

(21) “Time Ontology in OWL.” *Same Origin Policy - Web Security*, Reuters Limited, www.w3.org/TR/owl-time/

(22) “Data Range Extension: Linear Equations.” *Same Origin Policy - Web Security*, Reuters Limited, www.w3.org/2007/OWL/wiki/Data_Range_Extension:_Linear_Equations.

(23) “OWL Web Ontology Language Reference.” Same Origin Policy - Web Security, Reuters Limited, www.w3.org/TR/owl-ref

(24) “JSON - Introduction.” *Browser Statistics*, www.w3schools.com/js/js_json_intro.asp.

(25) “Logical JSON Operators.” *The Analytics Maturity Model (IT Best Kept Secret Is Optimization)*, IBM Corporation, www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.swg.im.dbclient.json.doc/doc/r0061295.html

(26) : “Comparison JSON Operators.” *The Analytics Maturity Model (IT Best Kept Secret Is Optimization)*, IBM Corporation,

(27) Joelittlejohn. “Joelittlejohn/jsonschema2pojo.” *GitHub*, 18 Oct. 2018, github.com/joelittlejohn/jsonschema2pojo.

(28) Georgievski, Ilce, Tuan Anh Nguyen, and Marco Aiello. "Combining Activity Recognition and AI Planning for Energy-Saving Offices." *UIC/ATC*. 2013

(29) w3schools. “XML Tutorial.” *w3schools*, 2018, www.w3schools.com/xml/.

(30) w3schools. “XML -- XPath Intro.” *w3schools*, 2018, www.w3schools.com/xml/xpath_intro.asp.

(31) w3schools. “XML-- Schema Tutorial.” *w3schools*, 2018, www.w3schools.com/xml/schema_intro.asp.

(32) w3schools. “XML -- XPath Intro.” *w3schools*, 2018, www.w3schools.com/xml/xpath_intro.asp.

(33) XML Processing Modules.” *Python Docs*, 2018, docs.python.org/3/library/xml.html.

(34) https://protege.stanford.edu/products.php

Appendix F: Planning Algorithm Trade Study

PROMETHEAN

Introduction & Background

A fundamental part of any planning system is the algorithm which is used to schedule and optimize the execution of tasks. This planning algorithm must have certain criteria, such as the ability to reliably return an optimal path from an initial state to a goal state in the form of a list of tasks to be completed, and to optimize on resources defined in the language. There are many algorithms available which have been heavily researched for these purposes, each providing some different subset of the criteria which are being considered.

The project is to design and produce an AI task planner to be utilized onboard spacecrafts (satellites, rovers, etc.) to allow the vehicle to make informed decisions without the need to communicate back to Earth. The planning algorithm chosen in this trade study will be the backbone of this decision-making process. Given definitions of states, tasks, perturbations, and actions, the algorithm will be able to return an optimal set of actions to be taken in order to transition between input initial and goal states. The designs of the algorithm and planning language together should allow for planning to be done on any sufficiently defined problem space.

The algorithm which is chosen at the end of this trade study must satisfy all three of the “must have” criteria. There are also a handful of “want” criteria which would be useful to have but are not critical to the success of the project. Three planning algorithms will be evaluated - Partial Order Planning, SATPLAN, and D* - to assess how well they fit (or can be expanded to fit) this use case. A comparison matrix will be used to find the criteria that each algorithm supports, which will motivate the final decision of the algorithm with which to move forward.

Introduction to Criteria and Weights

Must Have Criteria:

- **Planning algorithm is able to optimize on multiple resources or can be extended to do so:**
    - The ability to optimize on resources is integral to the success of this project. The algorithm should be able to take into consideration one or more resources, given precedence, and find optimal schedules for tasks which minimize or maximize the chosen resources.
- **Planning algorithm is able to output a correct and complete schedule:**
    - This criterion describes the fundamental premise behind all planning algorithms. First, the output schedule of the algorithm must be correct. This means that the schedule’s ordering of the tasks does not violate preconditions of subsequent tasks. An example of an incorrect output would be ordering tasks such that the preconditions of one task are not met at the time of its execution. Secondly, the output schedule must be complete. This means that the schedule contains a complete list of tasks that start from the initial state to the goal state.

- **Planning algorithm takes a task network as input with a goal and initial states:**
    - The language and planning system will model the problem space as a graph of tasks and states. Thus, the planning algorithm must be able to take a graph as input. The algorithm should also take in the initial state and goal state so that it knows where to search in the network.

Want Criteria:

- **Planning algorithm keeps track of resource usage/plots:**
    - The planning system requirements include outputting resource usage plots in order to monitor resource consumption over time. It would be advantageous if this functionality was already built in to existing implementations of the planning algorithm.

- **Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state:**
    - In the case where tasks are independent, meaning that the postconditions do not use the same state, those tasks can be parallelized. In such a situation, the tasks should be executed simultaneously to decrease the amount of time needed to reach the overall goal state.

- **Efficiency:**
    - **Time Complexity:**
        - The computational complexity of the algorithm should be as low as possible since it will likely be running on low-spec hardware. It should output a plan in a reasonable amount of time and the amount of input should not affect its runtime very drastically.

    - **Space Complexity:**
        - The space complexity of the algorithm should be as small as possible since it will be running on a system with low memory. Some algorithms may use a large amount of memory to cut down on runtime, but memory usage during runtime will be prioritized.

- **Ease of Implementation:**
    - This criterion is a general measure of how easy it will be to implement the planning algorithm. Such factors to examine include: the understandability and simplicity of the algorithm, how easy it is to extend its functionality, or whether it is already implemented in the planning system's language.

- **Planning algorithm is able to adapt the plan when perturbations occur or can be extended to do so:**
    - The planning algorithm can account for perturbations specified in the language's problem definition. The planner should be able to replan efficiently after a perturbation occurs. This is different than just starting the algorithm again from a new start state.

Want Criteria	Weights (Totals to 1)
The planning algorithm keeps track of resource usage/plots	0.05
Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state	0.1
Efficiency	0.3
• Time Complexity	0.1
• Space Complexity	0.2
Ease of Implementation	0.35
Planning algorithm is able to adapt the plan when perturbations occur or can be extended to do so	0.2

Analysis

Partial Order Planning:

Partial order planning is slightly different from other graph-based approaches to planning. Partial order planning relies on the principle of least commitment. Each iteration of the algorithm attempts to satisfy a requirement of the goal state by finding a valid action or a series of valid actions that will be added to the plan. These actions are not scheduled in any order unless other actions in the plan require it. Since there is no strict order to the set of actions produced by a partial order planner, parallelizing actions is fairly easy. This flexibility can be useful, but it is more computationally expensive. In addition to finding actions for the plan, the planner must also schedule those actions. A partial order planner must also calculate and store ordering constraints for each action. This algorithm has been used in several planners in the past with notable examples being NOAH (1975), MTC (1987), SNLP (1991), and UCPOP (1992)[8].

Must Have Criteria:

- **The planning algorithm is able to optimize on multiple resources or can be extended to do so:**
    - Partial order planning can be extended to choose actions that optimize multiple resources. The basic algorithm does not include logic to keep track of resource usage, but that can be added to the algorithm in a few ways. Optimization logic can be added to the part of the algorithm that chooses which actions to add to the plan, or it can be added to the scheduling part of the algorithm. To find the most optimal plan, the algorithm should consider

optimizations at both steps of the planning process. If we would like to add an upper or lower limit to a resource, that can be added to the constraint set as well.

- **Planning algorithm is able to output a correct and complete schedule:**
    - Partial order planning is capable of outputting a correct and complete schedule given an initial state, the goal state, and the actions possible in that problem space. Proofs of correctness and completeness are available in appendix A[5].
- **Planning algorithm takes a task network as input with a goal and initial states:**
    - Partial order planning can take a starting state, a goal state, and a task network as input. It handles these inputs slightly differently from other algorithms, however. The goal state will be converted to an agenda containing a list of requirements that plan should meet. The algorithm will remove a requirement from that agenda and choose one or more valid actions from the task network to fulfill that requirement. If the preconditions for the selected action(s) rely on some previously selected actions, an ordering constraint will be added to the plan to ensure that the final, fully-ordered plan is valid.

Want Criteria:

- **The planning algorithm keeps track of resource usage/plots:**

Rating: 0/10

- Partial order planning will produce a solution/plan. This plan can then be mapped to the changes in the state of the system to provide observations of how resource plots change.

- **Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state:**

Rating: 5/10

- Since partial order planning does not order actions until forced to do so, it is more conducive to parallelized plans. For example, tasks that do not have an explicit ordering constraint between one another can be run in parallel. Tasks which have the same ordering constraints can also be run in parallel after those ordering constraints have been satisfied. The process of scheduling tasks in parallel is performed after the actions in the plan have been selected, so we will have to add a scheduler to the planner as well. Implementing an optimal scheduler could be difficult.

- **Efficiency:**
    - **Time Complexity:**

Rating: 3/10

- Partial order planning is a significantly complex algorithm with a high *per-node* cost. It has a very large search space since concurrent actions are allowed[8]. The amortized time of partial order planning relies heavily on the number of edges in the graph which can grow quickly because of the allowance of concurrent actions[5].

- **Space Complexity:**

  Rating: 3/10

  - Partial order planning is more spatially expensive than other planning algorithms. A partial order planner needs to store the actions of a plan as well as metadata about the actions themselves. A separate data structure is used to keep track of any ordering constraints between actions[9]. The algorithm must also track causal links between actions since certain actions may alter the state of the system, invalidating the preconditions of previously chosen actions[9]. The ordering constraint and causal link sets may be small if each of the actions in the plan are independent and do not interfere with each other (modify the same state attribute); however, that will likely not be the case in practice. In the worst case, there exists a time constraint for each action and a causal link for each precondition of every action. Because of all of this extra data, partial order planning does not scale very well.

- **Ease of Implementation:**

Rating: 6/10

  - Partial order planning may be hard to understand, but it is well documented with plenty of pseudocode and reference materials available online[7|8]. Numerous resources, such as previous implementations like those seen in Appendix B 4 a, and possible heuristics that can be used to speed up the algorithm are also available to help with implementation[6].

- **Planning algorithm is able to adapt plan when perturbations occur or can be extended to do so:**

Rating: 0/10

  - Partial order planning has no intrinsic method for handling perturbations. Perturbations could be handled by recalling the algorithm to create a new plan when a perturbation occurs.

<u>Conclusion:</u>

Partial order planning satisfies many of our criteria. It can optimize attributes and produce a correct, parallelizable plan given a starting state, actions and a goal state. Partial order planning also meets a few of our want criteria since implementing parallel schedules would be fairly easy and plenty of documentation exists to help us implement it. This algorithm's weaknesses may outweigh its strengths, however. Since partial order planning produces unordered plans, we will have to research and implement a scheduler as well. It is also computationally and spatially expensive and cannot process perturbations without major modifications. While it does satisfy our required criteria, partial order planning may not suit our needs as well as other planning algorithms.

SATPLAN:

SATPLAN is a task planning algorithm based on the Blackbox planning system, first outlined by Kautz and Selman in 1999[3]. The current iteration, titled SATPLAN-2006, is an updated version of the 2004 variant which took first place at the International Planning Competition at the 14th International Conference on Automated Planning and Scheduling. In SATPLAN, planning problems are defined using STRIPS notation and converted to an instance of the Boolean Satisfiability Problem, where they are subsequently solved using various satisfiability engines. The main engine used in SATPLAN-2006 is Chaff, although it is possible to define other engines to be used to compare efficiencies. SATPLAN is guaranteed to find a path of minimal length through the planning graph, assuming a path exists. It is also able to find paths which contain actions that can be executed in parallel, improving execution times.

Must Have Criteria:

- **The planning algorithm is able to optimize on multiple resources or can be extended to do so:**
  - The Boolean Satisfiability Problem finds a series of actions to carry out which eventually satisfy an equation of the form (T/F) AND (T/F) AND ... etc.
    E.g. ~Rotating(wheelFL) AND ~Rotating(wheelRL) AND Rotating(wheelFR) AND Rotating(RR): to make a rover turn left
    The possible actions are represented as a network, where the algorithm will find the optimal shortest path. Because of the way the network is parsed, it is not possible to explicitly optimize a certain resource. Instead the planner guarantees the shortest possible path is found from start to goal on the network, in number of steps taken overall.

- **Planning algorithm is able to output a correct and complete schedule:**
  - In the definition of SATPLAN, it guarantees a complete and correct schedule of tasks will be found assuming one exists. SATPLAN's completeness and correctness is detailed in[2].

- **Planning algorithm takes a task network as input with a goal and initial states:**
  - The Blackbox planner on which SATPLAN is based takes a task network definition in STRIPS where any two defined states can be the initial and goal states for the plan, satisfying this requirement

Want Criteria:

- **The planning algorithm keeps track of resource usage/plots:**

Rating: 0/10

  - It would be possible to have another task running which keeps track of how much resources the planner is using, but there is no native implementation.

- **Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state:**

Rating: 10/10

  - The definition of SATPLAN describes how it can find actions which do not interfere with each other and parallelize them to be executed simultaneously[3].

- **Efficiency:**
  - **Time Complexity:**

  Rating: 7/10

    - SATPLAN is well known for its speed. While the exact time complexity is not publicly available, it can be assumed in the worst case the algorithm will take O(n*e) time, where n is the number of nodes (states) and e is the number of edges. In practice, since only a small subset of the possible nodes will be expanded and processed, the time complexity will rarely hit its upper bound.

  - **Space Complexity:**

  Rating: 6/10

    - SATPLAN does relatively well on space needed for computation. The Blackbox planning algorithm, which is the backbone of SATPLAN, will take an input task network and find the possible paths of actions to take. The path(s) are then encoded into memory to do mutex propagation on, which means that substantial memory space is only used for the subset of paths which are useable.

- **Ease of Implementation:**

Rating: 4/10

  - SATPLAN is based on STRIPS, which is a fairly easy to implement task network definition. Unfortunately, it has already been decided that the planning language for this project will be JSON, which means a parser must be implemented to bridge the gap between the JSON definition and STRIPS. Another hiccup is how esoteric STRIPS is and the need to explicitly define all pre- and post-conditions, resource effects, and time constraints for each possible action to be taken. The combination of these factors severely limits how easy the implementation of SATPLAN would be.

- **Planning algorithm is able to adapt plan when perturbations occur or can be extended to do so:**

Rating: 0/10

  - SATPLAN does not have a native way to handle perturbations but can be extended to do so by restarting the planner and creating an entirely new action schedule when a perturbation is encountered.

Conclusion:

SATPLAN is not a viable option for the planning algorithm because it does not meet all must have criteria. SATPLAN is unable to optimize on specific resources but instead optimizes on minimizing the number of steps through a network from initial states to goal states. This is incompatible with our task network representation and need to optimize on multiple, ordered resources. SATPLAN is a complete, plan-graph based algorithm and has the added benefit of finding parallelizable tasks but is not appropriate for this specific planning problem.

D* (Lite) Search:

D* Search- also known as Dynamic A* Search- is an adaption of the A* algorithm that handles partially known environments using incremental searching. D* is “functionally equivalent to the A* replanner”[1] in that its basic premise uses A* cost-based search but the cost parameters can change during problem solving. The motivation behind the D* Search algorithm was to aid in 3D space path finding in robotics. Thus, this algorithm is both quick because it is used to solve real-time problems as well as adaptable to changes in the environment. Additionally, D* Lite Search is a revision of the D* Search algorithm that is also being considered because it achieves the same purpose as D* but is known to be faster and simpler.

Must Have Criteria:

- **The planning algorithm is able to optimize on multiple resources or can be extended to do so:**
    - D* Search is an adaption of A* Search, meaning it can likewise optimize on resources by using a problem specific cost function. The cost function will give D* Search a set of metrics in which to score specific moves (or tasks). Thus, the cost function could be designed to rank multiple resources in order to find the optimal set of tasks. Given this cost function, D* and D* Lite Search will always find the path (or set of tasks) with the lowest total cost- which is a combination of the cost function and the distance to the goal state. A proof showing that D* and D* Lite are optimal is detailed in[1].

- **Planning algorithm is able to output a correct and complete schedule:**
    - Given the correct setup, D* and D* Lite will output both a correct and complete schedule. To begin, a schedule is correct if it can get from the initial state to the goal state using only legal tasks and intermediate states. This criterion primarily relies on the initial graph setup. If D* is given a state graph representing only legal moves from state to state, then the algorithm will maintain its correctness. The basic premise behind this statement lies in that given a correct and legal state graph, D* will only output a *subgraph* of the original. Therefore, the outputted schedule is also correct.
    A schedule is complete if it starts from the initial state and ends with the goal state. D* and D* Lite ensures a completed path from the starting node to the ending node, given that such a path exists. Proofs for D* and D* Lite’s completeness are documented in “Robotic Motion Planning: A* and D* Search”[1] and “D* Lite.”[4].

- **Planning algorithm takes a task network as input with a goal and initial states:**
    - Similar to A* Search, D* Search takes a graph, a starting node, and ending node as input. Thus, if the task network was modelled such that each node represented a state and each edge was a task connecting states, D* Search could handle the task network with initial and goal states as proper input.

Want Criteria:

- **The planning algorithm keeps track of resource usage/plots:**

Rating: 0/10

- D* and D* Lite algorithms do not specifically keep track of resource usage. However, these algorithms would keep track of the path (or plan) from the start state to the goal state. Thus, it would be trivial to implement tracking the changes in resources along the resulting path.

- **Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state:**

Rating: 0/10

- No previous implementation was found showing that D* could produce a schedule where tasks are executed in parallel. This is most likely because D* is famously used to search in 3D space where movement cannot be executed in parallel. While this does not mean it is impossible to extend D* to parallelize tasks, there is no implementation to base our solution off of, making it out of scope for this project.

- **Efficiency:**
    - **Time Complexity:**

Rating: 6/10

- Though no specific big-O notation exists for D* there are a couple things we were able to find out about it and D* Lite. In dynamic environments the D* algorithm is more computationally efficient than the more known A* algorithm, making it a good candidate for our system with perturbations[1]. As for D* Lite, the execution time is dependent upon the frequency of perturbations.

- **Space Complexity:**

Rating: 4/10

- Though no specific information could be found for the big-O notation of either D* or D* Lite, there is information showing D* Lite as more efficient compared to D*. Three different machine and implementation agnostic tests were performed to compare D* and D* Lite. D* Lite was at least or more efficient than D* in all three tests[4]. This comparison was done by counting vertex accesses, vertex expansions, and heap percolates of both algorithms.

- **Ease of Implementation:**

Rating: 6/10

  - D* Search is considerably more complex than its predecessor- A* Search. The dynamic and online nature of the algorithm involves significantly more steps and requires a high degree of understanding in its underlying theoretical mathematics. Given this, it may take a good deal of time to understand and then implement D* Search. On the other hand, D* Lite is a refined version of D* which cuts out much of the complexity.
    It's also important to note that both D* and D* Lite are predominantly used in pathfinding in 3D space and the re-planning is done in real time as the robot encounters obstacles. Thus, it may be difficult to reformulate the original planning problem to fit into a D* algorithm.
    However, may implementations of D* and D* Lite Search already exist in languages such as Python and C++. These implementations are referenced in Appendix B.

- **Planning algorithm is able to adapt plan when perturbations occur or can be extended to do so:**
  Rating: 10/10
  - D* and D* Lite were specifically designed to handle partially known environments where changes may occur, thus these algorithms can adapt when perturbations appear. Under A* Search, if the system underwent any changes along the way, A* would have to restart completely and re-plan. D* Search modifies this approach by incrementally searching and "intelligently caching intermediate data for speedy replanning"[1]. This approach saves a significant amount of time compared to the naive A* method. A more detailed explanation of how D* re-plans online can be found at[1].

<u>Conclusion:</u>

D* is a complex, but very useful algorithm. It is shown to be much more efficient than simpler algorithms like A* in dynamic environments and can find an optimal set of tasks for our spacecraft despite the constant changing of its environment. D* is complete, it can search over a task network, and can gracefully handles perturbations. There is even a variant of D*, D* Lite, a simpler and more efficient version of D*, which is most likely what would be implemented. D*, like A*, needs a heuristic function in order to find an optimal path through a task network. This will prove to be one of the larger challenges of D* because without a proper heuristic the time and space efficiency start to lower. A heuristic function will need to include assumptions of the problem space as well as a way to optimize on resources of the user's choice. Another challenge to D* will be writing and understanding it in general. The algorithm is very complex and will take a lot of time to get running. Fortunately, if we implement the simpler A* first before expanding to the full D* we will at least have something to fall back on. In our research D* was never used as a planning algorithm, but rather as a search through a physically changing space. This makes D* a risky choice to implement, since we will have no references to planning problems, but its efficiency and perturbation management may prove to be very advantageous.

Comparisons With Alternatives

Decision Criteria	Algorithm Options		
Must Haves	**Partial Order Planning (Y/N)**	**SATPLAN (Y/N)**	**D* (Lite) Search (Y/N)**
The planning algorithm is able to optimize on multiple resources or can be extended to do so	Y	N	Y
Planning algorithm is able to output a correct and complete schedule	Y	Y	Y
Planning algorithm takes a task network as input with a goal and initial states	Y	Y	Y
Wants	**Partial Order Planning Score (Out of 10)**	**SATPLAN Score (Out of 10)**	**D* (Lite) Search Score (Out of 10)**
The planning algorithm keeps track of resource usage/plots (.05)	0	0	0
Can produce a schedule of tasks to be executed in parallel if postconditions do not use the same state (.1)	5	10	0
Time Complexity (.1)	3	7	6
Space Complexity (.2)	3	6	4
Ease of Implementation (.35)	6	4	6
Planning algorithm is able to adapt plan when perturbations occur or can be extended to do so (.2)	0	0	10
Want Score	**3.5**	**4.3**	**5.5**
Meets Must Have Criteria	**Y**	**N**	**Y**
Decision	**N**	**N**	**Y**

Conclusion

The results of this trade study support D* or D* Lite as the best alternative to use as the basis for the planning algorithm implementation. D* meets all “must have” criteria. Fundamentally rooted in A* Star, D* is a graph search algorithm that can correctly identify a path (or plan) from an initial state to a goal state. This algorithm additionally has a high degree of flexibility because it leverages a problem specific heuristic function. Thus, D* can adapt to optimize on anything the programmer specifies. Regarding the “want” criteria, D* is the only algorithm that directly handles perturbations that are encountered during runtime. This attribute is very advantageous with regards to time because it would not require the planner to restart from scratch. Additionally, D* and D* Lite have multiple implementations to reference and its foundation in A* Search makes it easier to understand and implement compared to the other algorithms examined in this trade study.

Appendix A: Rejected Algorithms

1. We considered a couple of planning algorithms (AEMS2 and HSVI) detailed in this paper about online planners for POMDPs: https://www.jair.org/index.php/jair/article/view/10559/25275. Our planner will be offline and does not need to account for that much uncertainty, so the algorithms detailed there do not fit our needs very well.
2. The heuristic algorithm used by the HSP planner (https://www.cs.toronto.edu/~sheila/2542/s14/A1/bonetgeffner-heusearch-aij01.pdf) was also considered, but the heuristic of relaxing the delete list seemed difficult to adapt for our purposes. It would increase memory overhead significantly, and adding perturbations could be difficult if older preconditions were still being considered by the heuristic function.
3. Forward Progression and Backward Regression Algorithms (Russell, S., & Norvig, P. (2010). Chapter 10. Artificial Intelligence: A Modern Approach (3rd ed.). S.l.: PEARSON.) were considered because they are foundational algorithms for many planners. We decided to reject these algorithms because of time and space constraints. Using basic progression or regression with uninformed search has exponential worst case time complexity which far exceeds our time limitations for our planning system.
4. Operations Research Scheduling (A.K.A. Job Shop Scheduling) Algorithms (Russell, S., & Norvig, P. (2010). Chapter 11. Artificial Intelligence: A Modern Approach (3rd ed.). S.l.: PEARSON.) were considered because they relate to scheduling jobs over a period time given a set of resources which aligns with the current problem. However, we discounted these algorithms because the preconditions for them cannot be met with our formulation of the problem. For example, job shop scheduling takes a list of jobs as input and then forms an optimal plan out of those jobs. Our planning algorithm must first find the jobs that need to be executed first. See Appendix B for more information about how job shop scheduling may still be of use in the planning system.

Appendix B: Miscellaneous

1. A* Search
   (Russell, S., & Norvig, P. (2010). Chapter 3. Artificial Intelligence: A Modern Approach (3rd ed.). S.l.: PEARSON.) was considered because it is widely used in planning systems. This algorithm was thoroughly looked at because it is already implemented in many languages and it meets many of the planning requirements. A* Search could be a good baseline for the planner because it utilizes the same cost function as D* Search but is simpler to implement.

2. Operations Research Scheduling (A.K.A. Job Shop Scheduling) Algorithms
   (Russell, S., & Norvig, P. (2010). Chapter 11. Artificial Intelligence: A Modern Approach (3rd ed.). S.l.: PEARSON.)
   This set of algorithms cannot be applied directly to the planning system because it's preconditions cannot be met with the original problem formulation. However, it's possible that these algorithms could be of use if another algorithm outlined above found a list of necessary tasks in order for the system to reach the goal state. Given this list of jobs, operations research scheduling algorithms can be applied in order to optimize over specific resources. The source above outlines how job shop scheduling can optimize over resources (including time or physical resources) and how it can create a plan of jobs to be executed in parallel. Both of these features are key requirements for our planner.

3. D* and D* Lite Search Implementations
   A. Python Implementation of D* Lite Search:
      https://github.com/mdeyo/d-star-lite/blob/master/d_star_lite.py
   B. Python Implementation of D*/D* Lite Search:
      https://github.com/avgaydashenko/d_star/blob/master/d_star.py
   C. C++ Implementation of D* Lite Search:
      https://github.com/ArekSredzki/dstar-lite/blob/master/Dstar.cpp

1. Partial Order Planning Implementations
   A. C++ Implementation
      https://github.com/buele/pop

Works Cited

1. Choset, Howie. “Robotic Motion Planning: A* and D* Search.” CS CMU, CMU Robotics Institute, www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf.
2. Kautz, Henry, et al. “SatPlan: Planning as Satisfiability.” *Rochester University CS*, www.cs.rochester.edu/u/kautz/papers/kautz-satplan06.pdf.
3. Kautz, Henry, and Bart Selman. “Unifying SAT-Based and Graph-Based Planning.” *Rochester University CS*, www.cs.rochester.edu/u/kautz/papers/ijcai99blackbox.pdf.
4. Koenig, Sven, and Maxim Likhachev. “D* Lite.” IDM Lab, idm-lab.org/bib/abstracts/papers/aaai02b.pdf.
5. Minton, Steven, John Bresina, and Mark Drummond. "Total-order and partial-order planning: A comparative analysis." *Journal of Artificial Intelligence Research* 2 (1994): 227-262.
6. Nguyen, XuanLong and Subbarao Kambhampati. “Reviving Partial Order Planning.” *IJCAI* (2001).
7. Poole, David, and Alan Mackworth. “8.5 Partial-Order Planning.” *Artificial Intelligence - Foundations of Computational Agents*, Cambridge University Press, 2010, artint.info/html/ArtInt_209.html.
8. Simmons, Reid. “Planning, Execution & Learning 1. Partial Order Planning.” *Carnegie Mellon School of Computer Science*, 2001, www.cs.cmu.edu/~reids/planning/handouts/Partial_Order.pdf.
9. Weld, Daniel S. “An Introduction to Least Commitment Planning.” *Computer Science and Engineering*, University of Washington, 1994, homes.cs.washington.edu/~weld/papers/pi.pdf.