Project 3: State-Space Planner

This project will help you understand the nuances and challenges of heuristic search in a planning context. You will write a program which solves general step-by-step planning problems in several different domains and compare your program to others on a suite of benchmark test problems.

This assignment was originally designed by Dr. Stephen Ware, and we are fortunate to make use of the starter code that he has graciously developed and provided.

Jump to Section

Planning Framework

This project is all about Planning, which is covered in AI: A Modern Approach chapter 10. Reviewing the Planning concepts that we talked about in lecture will, of course, also be helpful for you! Also helpful is reviewing the documentation for Dr. Ware's Planning Framework that you'll be using for this project. Just as before, download the doc folder from Moodle and open index.html to read the documentation of the framework.

There is a lot of documentation, as tradition demands. The most important package to familiarize yourself with for this project is the package ***edu.uno.ai.planning.ss*** – the 'ss' stands for 'State Space' which is the type of planner you are making for this assignment. In addition to this, the `Step` class and the `Problem` class inside of the ***edu.uno.ai.planning*** package are likely to be helpful as well.

The Planning framework is written in Java 11. As before, to work on this project, you'll need to make sure you have a version of the Java Virtual Machine that is at least Java 11. If you have done the previous assignments in this class, you'll be able to do this one too. Install it on your computer, and ensure that Java is on your system path.

Once Java is installed, download and unzip the Planning framework. It contains the following files and folders:

- `planning.jar`, code for representing and solving planning problems.
- `benchmarks`, a folder containing many example planning domain and problem files.
- `planners`, a folder containing several example planning algorithms.
- `ExamplePlanner.zip`, the source code for the Breadth First Search planner.

- `grade.bat`, a Windows batch file for grading your final submission.
- `grade.sh`, a Mac or Linux shell script for grading your final submission.

Note how similar the structure is to your previous programming project!

All of the planning challenges are located in the **benchmarks** folder. A planning challenge is broken up into two parts: a domain file which describes the general kinds of actions that are possible for a specific challenge, and a problem file, which gives specific details about the challenge. Both the domain and the problems are stored in *pddl* files, a Planning Domain Definition Language file.

There are four files that specify domains for this project: *blocks.pddl*, *cake.pddl*, *cargo.pddl*, and *wumpus.pddl.* Each of these files specifies the actions you can take in the domain, as well as any constants that will be present in *every* problem in that domain (e.g., the Table in the blocks domain, or the player, Wumpus, and chest in the Wumpus domain).

Every other file in the benchmarks folder specifies a specific problem. For each domain, there are many problems. i.e., there are several "blocks" problems, several "cargo" problems, etc.

Let's consider one of the example domains, the *cargo* domain (which you can find as `cargo.pddl` in the benchmarks folder):

```
(:domain cargo
    (:action load
      :parameters (?cargo - cargo ?plane - plane ?airport - airport)
      :precondition (and (at ?plane ?airport)
                         (at ?cargo ?airport))
      :effect (and (not (at ?cargo ?airport))
                   (in ?cargo ?plane)))
    (:action unload
      :parameters (?cargo - cargo ?plane - plane ?airport - airport)
      :precondition (and (in ?cargo ?plane)
                         (at ?plane ?airport))
      :effect (and (at ?cargo ?airport)
                   (not (in ?cargo ?plane))))
    (:action fly
      :parameters (?plane - plane ?from - airport ?to - airport)
      :precondition (at ?plane ?from)
      :effect (and (not (at ?plane ?from))
                   (at ?plane ?to))))
```

This domain defines three kinds of actions that are possible, which are:

- **Load** a piece of cargo onto a plane at some airport.
- **Unload** a piece of cargo off of a plane at some airport.
- **Fly** a plane from one airport to another.

You can see each action has three things defined: parameters, preconditions, and effects.

The parameters are similar to the parameters of a function in 'normal' programming, and define the types of things that action deals with and gives a name to each thing (e.g., the load action deals with a cargo object named cargo, a plane object named plane, and an airport object named airport).

The preconditions define what must be true for that action to take place (e.g., load demands that the plane is at an airport, and the cargo is at that same airport).

The effect dictates how the state of the world changes once the action has been carried out (e.g., once the load action has taken place, the cargo is now *not* at the airport, and is instead now in the plane).

Now let's look at one of the problems in the Cargo domain (`deliver_1.pddl`):

```
(:problem deliver_1
   (:domain cargo)
   (:objects cargo_msy - cargo
             plane_atl - plane
             atl - airport
             msy - airport)
   (:initial (and (at cargo_msy atl)
                  (at plane_atl atl)))
(:goal (at cargo_msy msy)))
```

This file defines several specific details:

- The problem has a name: `deliver_1`
- It specifies which domain to use – the cargo domain (which we defined above) – this is how it knows what actions are available.
- It defines the objects that exist in this particular instantiation of cargo world. Namely, we have:
    - A piece of cargo named `cargo_msy`
    - A plane named `plane_atl`
    - We have two airports: one named `atl` and a second named `msy`
        - Note: other cargo problems will have different amounts of objects. Deliver_5.pddl, for example, five cargos, three planes, and five airports!
- It also defines the initial state of the world:
    - `cargo_msy` begins its life at the `atl` airport.
    - `plane_atl` begins its life at `atl` as well.
- And finally, it defines a goal: we want to find a way to get `cargo_msy` to the `msy` airport.

Perhaps you reading this can think to yourself: ah! This is easy! You **load** the cargo into the plane, you **fly** the plane from atl to msy, and then you **unload** the cargo from the plane at the new destination! But sure, it's easy for your smart human brain

to see the answer, but how do you get the computer to figure it out? Well… that's your job for this assignment!

The planners folder contains five .jar files which are each implementations of common planning algorithms: `bb.jar`, `bfs.jar`, `gp.jar`, `hsp.jar`, and `pop.jar` . To see how any given algorithm performs on this problem, say, HSP, open a terminal window, go to the folder you unzipped, and execute this command:

```
java -jar planning.jar -a planners/hsp.jar -d benchmarks/cargo.pddl -p benchmarks/deliver_1.pddl
```

You can see that the command is made up of the following things:

- `Java -jar planning.jar` ---- This is how you run the provided planning framework.
- `-a planners/hsp.jar` ---- This is specifying which algorithm(s) to use (here, `planners/hsp.jar`, is saying "please use the hsp.jar file which is located in the planners directory"). You may specify as many .jar files as you wish to compare against each other.
- `-d benchmarks/cargo.pddl` ---- This is specifying which domain(s) to use (here, `cargo.pddl`).
- `-p benchmarks/deliver_1.pddl` --- This is specifying which problem(s) to use (here, `deliver_1.pddl`). You can specify more than one to have the algorithms you specified solve multiple problems. The problems should all be in the domain that you specified.

If you run the above command, you should get the following output (or something close to it at least):

```
Planner: HSP
Domain: cargo
Problem: deliver_1
Result: success
Nodes Visited: 4
Nodes Generated: 13
Time (ms): 1
Solution:
(load cargo_msy plane_atl atl)
(fly plane_atl atl msy)
(unload cargo_msy plane_atl msy)
```

This indicates that the algorithm was successful, and that it performed four operations while finding the solution. Those last three lines are a sequence of steps to solve the problem, also known as a "plan." In English it would be:

- First load the cargo onto the plane at the Atlanta airport (atl)
- Then fly the plane to the New Orleans airport (msy)
- Then unload the cargo off of the plane at the New Orleans airport.

Exactly the same as what our smart human brains figured out a few paragraphs ago!

If you provide more than one planner or more than one problem, the program will print out a comparison of each planner's performance on each problem. See the "Grading" section below for a discussion of how the planners are ranked.

Assignment

Your assignment is to write your own planner! It will be graded based on how it performs relative to the other planners provided with this project. There are several families of planning algorithms, and the one that is simplest and easiest to understand is the state-space search family. You can make your own state-space planner by creating a .jar file which contains any class which extends the class `edu.uno.ai.planning.ss.StateSpacePlanner`. Your class will need to override the abstract `makeStateSpaceSearch` method.

The file ExamplePlanner.zip contains the source code for the simplest (and lowest performing) planner, the Breadth First Search Planner. You might want to start with that planner and modify it to create your own.

A "state-space" planner is one that explores the space of possible states. This space is a graph whose nodes are states of the world (for example, where each plane and each box of cargo is located) and whose edges are actions which change the state (for example, updating the location of a plane by having it fly from one airport to another). We start at the initial state and then consider every possible action which could be executed in that state. Each of those actions leads to a new state, so we consider every possible action that could be executed there, and so on, until we find a state where the goal is met.

Your planner's `makeStateSpaceSearch` method will need to return a `StateSpaceSearch` object. For example, the Breadth First Search Planner has a class called `BreadthFirstSearch.java` which extends `StateSpaceSearch`.

Your planner must be named according to your UNO student username. Because my university e-mail address is bsamuel@cs.uno.edu, my username is `bsamuel`.

Your search needs to generate StateSpaceNode objects. A state space node has three things:

- A parent node, which is the state that came before this state
- A plan, which is a sequence of steps
- A state, which represents the current state after executing the plan

You will keep generating state space nodes until you find one where the problem's goal is met.

A StateSpaceSearch object gives you the first (or root) node of the search:

```
Node 0
```

```
Parent: null
Plan: null
State: (and (at cargo_msy atl) (at plane_atl atl))
```

This node has no parent, because it is the very first one. It has no plan (or, alternatively, a plan with 0 steps) because we have not yet considered any steps. Its state is simply the problem's initial state, where both the cargo and the plane are at the Atlanta airport.

Notice that a state space node object has a method called `expand`. This is how you generate more nodes. Each call to expand counts as "one operation." Every planner is limited to 10,000 operations, and five minutes, when solving a problem.

A `StateSpaceProblem` conveniently provides you with a list of all possible steps (in its `steps` field). Let's consider the step `(load cargo_msy plane_atl atl)`. Calling `expand` from Node 0 with that step would generate a new node:

```
Node 1
Parent: Node 0
Plan: (load cargo_msy plane_atl atl)
State: (and (in cargo_msy plane_atl) (at plane_atl atl))
```

This node has the root node as its parent. Its plan has 1 step (the step we just took). Its state has been updated to reflect the fact that the cargo is no longer at the airport but is now in the plane.

Now let's consider the step `(fly plane_atl atl msy).` Calling `expand` from Node 1 with that step would generate a new node:

```
Node 2
Parent: Node 1
Plan: (load cargo_msy plane_atl atl) (fly plane_atl atl msy)
State: (and (in cargo_msy plane_atl) (at plane_atl msy))
```

This node has Node 1 as its parent. Its plan has 2 steps (both of the steps that we have taken so far). Its state has been updated to reflect the fact that the plane is now at the New Orleans airport.

Finally, let's consider the step `(unload cargo_msy plane_atl msy)`. Calling `expand` from Node 2 with that step would generate a new node:

```
Node 3
Parent: Node 2
Plan: (load cargo_msy plane_atl atl) (fly plane_atl atl msy) (unload cargo_msy plane_atl msy)
State: (and (at cargo_msy msy) (at plane_atl msy))
```

The problem's goal is met in this state, which means this node's plan is a solution to the problem! To generate this solution, we had to "visit" 3 nodes: 0, 1, and 2. We generated 4 nodes: 0,1,2, and 3. We called the expand method 3 times, so we performed 3 operations. To verify if a plan is a solution to the problem, you can

make use of a line like this, invoking the `isSolution` method of your problem object, and passing it in a node's plan:

```
problem.isSolution(current.plan)
```

Importing and Exporting an Eclipse Project

The file ExamplePlanner.zip contains the source code for the Breadth First Search Planner. You can import the project into Eclipse by following these steps:

- Start Eclipse
- File > Import
- Under the General category, choose Existing Projects into Workspace and click Next.
- Choose the Select archive file option.
- Browse for RandomSolver.zip
- Click Finish.

Make sure to change the name of the planner from "Example" to your student username. You should probably also rename the class to something other than ExamplePlanner. You can do that by right-clicking on the class and choosing `Refactor > Rename.`

You have no doubt noticed the pattern of how these projects are set up by now. However, this time around there is a key difference: **the .zip file you downloaded has two *projects* in it**. As opposed to before, where it only had one project but two files. One project is named "Example Planner" and the other is named "Test Planners". The second project provides an *unofficial* way to test your planner from inside an IDE like Eclipse, and it is there for your convenience. This is not how your planner will be graded. When you are done, you must still export your planner as a jar file, and it will be graded as described below in the next section.

To export an Eclipse project as a jar file, follow these steps:

- Right click on the project that contains **your** planner (not the "Test Planners" project) and choose `Export`
- Under the `Java` category, choose `JAR file` and click `Next`.
- Choose the destination for the exported file, making sure that it is named with your student username and ends in `.jar`.
- Click `Finish`.

You must also export and submit your source code:

- Right click on the project that contains **your** planner (not the "Test Planners" project) and choose `Export`.
- Under the `General` category, choose `Archive File` and click `Next`.
- Choose the destination for the exported File, making sure that it is named with your student ID and ends in `.zip`.
- Click `Finish`.

Grading

This is how I will grade your project: First, I will download your jar file to the planners folder. Then I will run a command to see how your planner compares against all of the provided planners. This doesn't have to be a mystery to you, you can run the command yourself before you submit your work – they are provided in the .zip file from Moodle (grade.bat and grade.sh). I will substitute your `jar` file anywhere it says "`your.jar`" (and you should do the same, if you wish to test this on your own machine).

In other words, your planner's performance will be compared against the performance of all of the other planners you have been provided with on all of the provided benchmark problems.

Your grade will be determined as follows (depending on whether you are enrolled in 4525 or 5525):

Your planner…	CSCI 4525 Grade	CSCI 5525 Grade
Outperforms HSP Planner	A+ (105%)	A (100%)
Outperforms GP Planner	A (100%)	B(80%)
Outperforms POP Planner	B (80%)	C (70%)
Outperforms BB Planner	C (70%)	D (60%)
Outperforms BFS Planner	D (60%)	F (50%)
Throws an Exception	F (50%)	F(0%)

You can see that your grade is all about "outperforming" other planners. What does it mean for one planner to "outperform" another? Here are the criteria:

- A planner that solves more problems is ranked higher.
- If two planners solve the same number of problems, the planner which generates shorter solutions is ranked higher. That is to say, a two step plan to solve the problem is better than a three step plan.
- In case of a further tie, a planner which performs fewer operations is ranked higher (i.e., calls the 'expand()' method fewer times).

- In case of a further tie, a planner which takes less time to solve its problem is ranked higher.

The following requirements must be observed when submitting your project or it will not be graded:

- Submit your project on Moodle by the deadline.
- Submit your jar file, named for your student username (e.g., mine would be `bsamuel.jar`). If your planner does not load correctly or throws an exception, you will receive no credit for this project.
- Submit the source code for your planner as an Eclipse archive via the method described above. It must be named for your student username (e.g., `bsamuel.zip`). If I cannot import your source code, or if you export it incorrectly, I will not grade it.
- You must perform your search using the provided `StateSpaceNode` object and its expand method. Do not attempt to circumvent the 10,000 operations limit (for example, by creating your own version of the Assignment class which does not obey the limit). Circumventing the 10,000 operation limit or 5 minute time limit will be considered cheating.

About the Planners

Here is some information about the other algorithms you may find helpful when designing your planner:

- *BFS* is a state space planner that uses breadth first search. First it will consider every possible plan with 1 step, then every possible plan with 2 steps, then every possible plan with 3 steps, and so on. This is obviously very inefficient!
- *POP* is a least-commitment partial-order planner. Its planning process is more similar to how humans form plans. It uses various logic concepts that we discussed, such as unification, to express ideas like "fly the plane from *somewhere* to New Orleans." This algorithm is very complicated, but is briefly described in your textbook in section 10.4.4
- *GP* is an implementation of the GraphPlan algorithm. The state space of a planning problem is huge, but it can be approximated by a much smaller data structure called a Plan Graph. Plan Graphs can be used in place of the state space to find solutions. See AI: A Modern Approach section 10.3
- *BB* is an implementation of the BlackBox planner, which translates a planning problem into a SAT problem, solves the SAT problem, and then translates the solution back into a plan. This version of the algorithm uses the DPLL solver from the previous project.
- *HSP* is an implementation of the Heuristic Search Planner. It uses A* search with a simple heuristic called "ignore the delete list." This means that once a fact

becomes true, the heuristic assumes it will never becomes false again (obviously that is not really the case, but remember: a heuristic is just a guess; it doesn't have to be exactly right). HSP is the fastest and also the simplest of all the planning algorithms provided for this project. See AI: A Modern Approach section 10.2.3 for details on HSP.

Class Competition

Back by popular demand! Every planner which runs without throwing an exception will be entered into a class-wide competition. The first, second, and third place winners of this competition will receive bonus points on this project.

Help and Hints

So, to expand a StateSpaceNode I need to specify a step, and the preconditions and effects of Step objects are represented as Propositions. That's fine, but when I look up the documentation for Proposition I just see weird "Atom" fields. How do I get a proposition's literals?

You see the weird "Atom" fields because Proposition itself is an abstract class that several other classes implement. That said, in all of the domains and problems for this project, you will only encounter two kinds of propositions: a literal by itself or a conjunction of literals. You can use Java's `instanceof` keyword to tell the difference between the two. Here's a helpful method to iterate through all the literals in a proposition:

```
private static List<Literal> getLiterals(Proposition proposition) {
	ArrayList<Literal> list = new ArrayList<>();
	if(proposition instanceof Literal){
		list.add((Literal) proposition);
	}
	else{
		for(Proposition conjunct : ((Conjunction) proposition).arguments){
			list.add((Literal) conjunct);
		}
	}
	return list;
}
```

Alternatively, if you want to avoid allocating lists on the heap every time, you can use the Consumer interface to invoke a function on each literal:

```
private static void forEachLiteral(Proposition proposition, Consumer<? super Literal> consumer) {
	if(proposition instanceof Literal){
		consumer.accept((Literal) proposition);
	}
	else{
		for(Proposition conjunct : ((Conjunction) proposition).arguments){
			consumer.accept((Literal) conjunct);
		}
```

```
        }
}
```

When we discussed HSP, a big part of determining the heuristic was giving every literal a value and updating it, but in the documentation I see that literals don't have a 'value' field. How can I keep track of the values of all of my literals?

Well, you could do something similar to how I recommended you keep track of the Chess state evaluations back in Project 1, by making a 'wrapper' class that associates a State with a double. Here you would do the same but link together a Literal and a value. But for this project, maybe try the following on for size: a `HashMap`.

If you've never used a HashMap before, well, they're awesome. They are basically a way to map an object of a certain type to a value/object of another type. This is referred to as a "key/value pair" – So, for example, if you wanted a way to map a Literal to a Double (i.e., a way to associate each Literal with its own Double), you could declare a HashMap like this:

```
private final HashMap<Literal, Double> cost = new HashMap<>();
```

Which is declaring and initializing a new variable of type HashMap called `cost` that is using objects of type Literal as its keys, and objects of type Double as the values associated with any given key. Or in simpler terms: a data structure that, when given a Literal as input, provides you with a Double back.

Of course, just declaring it and initializing it will create an empty HashMap; it's up to you to fill it up (and update it) with the appropriate values.

I'll leave it to you to read up on HashMaps (https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html) but the methods that will likely be most useful for you are `get()` (to retrieve the current value of any given key in the hashmap) and `put()` (to set/update the value of any given key).

How do I deal with infinite values?

Some of the heuristics we discussed, like HSP, require you to initialize certain values to infinity (which is not actually a number, but rather, a concept). Some people use a number's maximum value to represent infinity, but this can lead to bugs if you are not careful.

```
int i = Integer.MAX_VALUE; // Represents infinity
i = i + 1; // Infinity + 1 should be infinity...
System.out.println(i); // ... but it's not, because an integer overflow occurs.
```

Java's double data type has a built-in representation for infinity that works the way we want it to:

```
double d = Double.POSITIVE_INFINITY; // Represents infinity
d = d + 1; // Infinity + 1 should be infinity...
```

```
System.out.println(d); // ... and it is!
```

How do you recommend I get started on this project?

Once we've covered State Space Planners in lecture, you will happily find that conceptually they are the easiest planners to understand (and, fingers crossed, the easiest to implement as well). You are also encouraged to read ahead in the book (Chapter 10) if you are itching to get started and we haven't covered state space planners in lecture yet.

You should also download the documentation and starter code. For the documentation, focus on gaining familiarity with the *package edu.uno.ai.planning.ss* – the 'ss' stands for 'State Space' which is the type of planner you are making for this assignment. In addition to this, the `Step` class and the `Problem` class inside of the *edu.uno.ai.planning* package are likely to be helpful as well.

And then, check out BreadthFirstSearch.java. Note how it receives the problem from ExamplePlanner.java, and how it uses that problem to access all possible steps that can be taken, as well as how it checks to see if the plan stored in any given node is a solution to the problem (and if it is, return that plan).

When you are feeling ready to start, either edit the name of BreadthFirstSearch.java or create a new file with a name indicative of the strategy you are attempting to implement (I highly recommend attempting HSP – it is the happy confluence of "easiest" to implement and also "best performing" here – take advantage of it)!

Then, update the name of the class to match the new file name, and then in ExamplePlanner.java change the StateSpaceSearch object you create and return from BreadthFirstSearch to match whatever you changed its name to. While you are at it, change the String on line 23 of ExamplePlanner.java to be your user name too.

And then, ah, you simply implement a form of state space search with an efficient heuristic! And again, we'll chat in lecture about a good, recommended strategy for how to actually do this!

Good luck!!!!