Writing your own Planner

The objective of this tutorial is to design a simple planner based on A* search.

1. Create a simple Java project with PDDL4J
2. Create the main class of your planner
3. Get the planner arguments from the command line
4. Searching for a solution plan
5. Write your own A* search strategy
6. Make your planner configurable by programming

Pre-requisite Installations

For this tutorial you need:

- Java JDK version 8 or higher is installed.
- A text editor such as Sublime or Atom or IDE such as Eclipse NetBean or IntelliJ.

In the following, we will give the commands line so that the tutorial can be done independently of any IDE.

Step 1. Create a simple Java project with PDDL4J

First, open a terminal and create your development directory ASP (A* Planner)

```
mkdir ASP
```

Then, create the sub-directories of your project

```
cd ASP
mkdir -p src/fr/uga/pddl4j/examples/asp
mkdir classes
mkdir lib
```

Finally, get the last binary of PDDL4J and save it in the `lib` directory

```
wget http://pddl4j.imag.fr/repository/pddl4j/binaries/pddl4j-4.0.0.jar
mv pddl4j-4.0.0.jar lib/pddl4j-4.0.0.jar
```

You are now ready to write your own A* planner.

Step 2. Create the main class of our planner

Create and edit a file called `ASP.java` in the directory `src/fr/uga/pddl4j/examples/asp`. The skeleton of this class is given bellow:

```java
package fr.uga.pddl4j.examples.asp;

import fr.uga.pddl4j.heuristics.state.StateHeuristic;
import fr.uga.pddl4j.parser.DefaultParsedProblem;
import fr.uga.pddl4j.parser.RequireKey;
import fr.uga.pddl4j.plan.Plan;
import fr.uga.pddl4j.plan.SequentialPlan;
import fr.uga.pddl4j.planners.AbstractPlanner;
import fr.uga.pddl4j.planners.Planner;
import fr.uga.pddl4j.planners.PlannerConfiguration;
import fr.uga.pddl4j.planners.ProblemNotSupportedException;
import fr.uga.pddl4j.planners.SearchStrategy;
import fr.uga.pddl4j.planners.statespace.search.StateSpaceSearch;
import fr.uga.pddl4j.problem.DefaultProblem;
import fr.uga.pddl4j.problem.Problem;
import fr.uga.pddl4j.problem.State;
import fr.uga.pddl4j.problem.operator.Action;
import fr.uga.pddl4j.problem.operator.ConditionalEffect;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import picocli.CommandLine;

import java.util.Comparator;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Set;

/**
 * The class is an example. It shows how to create a simple A* search
planner able to
 * solve an ADL problem by choosing the heuristic to used and its weight.
 *
 * @author D. Pellier
 * @version 4.0 - 30.11.2021
 */
@CommandLine.Command(name = "ASP",
    version = "ASP 1.0",
    description = "Solves a specified planning problem using A* search
strategy.",
    sortOptions = false,
    mixinStandardHelpOptions = true,
        headerHeading = "Usage:%n",
        synopsisHeading = "%n",
        descriptionHeading = "%nDescription:%n%n",
        parameterListHeading = "%nParameters:%n",
        optionListHeading = "%nOptions:%n")
public class ASP extends AbstractPlanner {

    /**
     * The class logger.
     */
    private static final Logger LOGGER =
LogManager.getLogger(ASP.class.getName());

    /**
     * Instantiates the planning problem from a parsed problem.
     *
     * @param problem the problem to instantiate.
     * @return the instantiated planning problem or null if the problem
cannot be instantiated.
     */
    @Override
    public Problem instantiate(DefaultParsedProblem problem) {
        final Problem pb = new DefaultProblem(problem);
        pb.instantiate();
        return pb;
    }

    /**
     * Search a solution plan to a specified domain and problem using A*.
     *
     * @param problem the problem to solve.
     * @return the plan found or null if no plan was found.
     */
    @Override
    public Plan solve(final Problem problem) {
    }

    /**
     * The main method of the <code>ASP</code> planner.
     *
     * @param args the arguments of the command line.
     */
    public static void main(String[] args) {
        try {
            final ASP planner = new ASP();
            CommandLine cmd = new CommandLine(planner);
            cmd.execute(args);
        } catch (IllegalArgumentException e) {
            LOGGER.fatal(e.getMessage());
        }
    }
}
```

The class ASP extends the abstract class AbstractPlanner that contains the basic methods of any planners.

Two methods must be overridden at least:

- The method `instantiate(DefaultParsedProblem problem)` is an abstract method of the class `AbstractPlanner`. This method takes as parameter an instance of parsed problem and return the corresponding instantiated or grounding problem. The problem returned contains all the information related to the problem, i.e., the actions, the initial state, the goal of the problem, etc.
- The method `solve(Problem problem)` is the main method of the planner. The method takes as parameter the instantiated problem returned by the previous method overridden and returns a plan solution of null if no plan was found.

Note

The given skeleton contains also a `main()` method to launch the planner from the command line. We will return to this method in the next section.

Note

Every planner can have a logger instance. This logger is based on Log4j library developed by Apache and specialized in logging. A great benefit of Log4j is that different levels of logging can be set for your planner. The levels are hierarchical and are as follows: *TRACE*, *DEBUG*, *INFO*, *WARN*, *ERROR*, and *FATAL*. Have a look to the web site of Log4j for more details. The declaration of the logger is done line 38. To see an example use of the logger see line 71 of the `main()` method.

Step 3. Get the planner arguments from the command line

Before writing the search algorithm let's first look at how to get the arguments from the command line. Your planner takes as inputs at least a domain file that contains the description of the planning operators and a problem file that define the initial state and the goal to reach. Both files domain and problem rely on PDDL (Planning Domain Description Language). For those who are not familiar with PDDL, first have a look to the tutorial PDDL Tutorial. To deal with complex command line arguments, PDDL4J used picocli library. Picocli allow to create rich command line applications just by adding annotation on the main class of our planner.

Step 3.1. Activate the default command line arguments

By default, the class AbstractPlanner already handles the common arguments of all planners: the domain and the problem description, the time allocated to the search and the log level. The domain and the problem descriptions are mandatory parameters. The log level and the time allocated to search are optional.

To activate the default command line for your planner you have just to add the following annotation before the class declaration:

```java
/**
 * The class is an example. It shows how to create a simple A* search
planner able to
 * solve an ADL problem by choosing the heuristic to used and its weight.
 *
 * @author D. Pellier
 * @version 4.0 - 30.11.2021
 */
@CommandLine.Command(name = "ASP",
    version = "ASP 1.0",
    description = "Solves a specified planning problem using A* search
strategy.",
    sortOptions = false,
    mixinStandardHelpOptions = true,
    headerHeading = "Usage:%n",
    synopsisHeading = "%n",
    descriptionHeading = "%nDescription:%n%n",
    parameterListHeading = "%nParameters:%n",
    optionListHeading = "%nOptions:%n")
```

and complete the `main()` method with the code below:

```java
    /**
     * The main method of the <code>ASP</code> planner.
     *
     * @param args the arguments of the command line.
     */
    public static void main(String[] args) {
        try {
            final ASP planner = new ASP();
            CommandLine cmd = new CommandLine(planner);
            cmd.execute(args);
        } catch (IllegalArgumentException e) {
            LOGGER.fatal(e.getMessage());
        }
    }
```

To test, first compile your planner:

```
javac -d classes -cp lib/pddl4j-4.0.0.jar
src/fr/uga/pddl4j/examples/asp/ASP.java
```

and run it with the command line:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.asp.ASP --help
```

You will obtain the following message:

```
ASP [-hiV] [-l=<logLevel>] [-t=<timeout>] <domain> <problem>

Description:

Solves a specified planning problem using a A* search strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset
600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                                INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

Step 3.2 Adding new command line arguments

Now, we have to add the specific arguments of our planner to allow to choose the heuristic function to used and set the weight of the heuristic. This step is relatively simple and straightforward. We just need to declare two new attributes: one for the heuristic and one for its weight. The heuristic is of type GoalCostHeuristic.Name and the weight of type double. Note that the weight must be greater than 0.

```java
    /**
     * The weight of the heuristic.
     */
    private double heuristicWeight;

    /**
     * The name of the heuristic used by the planner.
     */
    private StateHeuristic.Name heuristic;
```

To complete, we also add the corresponding getters and setters:

```java
  /**
   * Sets the weight of the heuristic.
   *
   * @param weight the weight of the heuristic. The weight must be
greater than 0.
   * @throws IllegalArgumentException if the weight is strictly less
than 0.
   */
  @CommandLine.Option(names = {"-w", "--weight"}, defaultValue = "1.0",
      paramLabel = "<weight>", description = "Set the weight of the
heuristic (preset 1.0).")
  public void setHeuristicWeight(final double weight) {
      if (weight <= 0) {
          throw new IllegalArgumentException("Weight <= 0");
      }
      this.heuristicWeight = weight;
  }

  /**
   * Set the name of heuristic used by the planner to the solve a
planning problem.
   *
   * @param heuristic the name of the heuristic.
   */
  @CommandLine.Option(names = {"-e", "--heuristic"}, defaultValue =
"FAST_FORWARD",
      description = "Set the heuristic : AJUSTED_SUM, AJUSTED_SUM2,
AJUSTED_SUM2M, COMBO, "
          + "MAX, FAST_FORWARD SET_LEVEL, SUM, SUM_MUTEX (preset:
FAST_FORWARD)")
  public void setHeuristic(StateHeuristic.Name heuristic) {
      this.heuristic = heuristic;
  }

  /**
   * Returns the name of the heuristic used by the planner to solve a
planning problem.
   *
   * @return the name of the heuristic used by the planner to solve a
planning problem.
   */
  public final StateHeuristic.Name getHeuristic() {
      return this.heuristic;
  }

  /**
   * Returns the weight of the heuristic.
   *
   * @return the weight of the heuristic.
   */
  public final double getHeuristicWeight() {
      return this.heuristicWeight;
  }
```

To test, your complete command line compile once again your planner:

```
javac -d classes -cp lib/pddl4j-4.0.0.jar
src/fr/uga/pddl4j/examples/asp/ASP.java
```

and run it with for instance the command line:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.ASP
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-
automatic/domain.pddl
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
     -e FAST_FORWARD
     -w 1.2
     -t 1000
```

Now the command line is set. The final command line of your planner is as follows:

```
ASP [-hV] [-e=<heuristic>] [-l=<logLevel>] [-t=<timeout>] [-w=<weight>]
    <domain> <problem>

Description:

Solves a specified planning problem using A* search strategy.

Parameters:
    <domain>              The domain file.
    <problem>             The problem file.

Options:
    -t, --timeout=<timeout>   Set the time out of the planner in seconds (preset
                                600s).
    -l, --log=<logLevel>      Set the level of trace of the planner: ALL, DEBUG,
                                INFO, ERROR, FATAL, OFF, TRACE (preset INFO).
    -w, --weight=<weight>     Set the weight of the heuristic (preset 1.0).
    -e, --heuristic=<heuristic>
                              Set the heuristic : AJUSTED_SUM, AJUSTED_SUM2,
                                AJUSTED_SUM2M, COMBO, MAX, FAST_FORWARD,
                                SET_LEVEL, SUM, SUM_MUTEX (preset: FAST_FORWARD)
    -h, --help                Show this help message and exit.
    -V, --version             Print version information and exit.
```

Step 4. Searching for a solution plan

You finally think we're here. How write my search procedure ? Two possibilities or the search procedure you want to use already exists in PDDL4J. In this case, it's extremely simple just call the right procedure in the `solve()` method. Otherwise, you have to write your own procedure. Let us first consider the first case. The second will consider in last part of this tutorial.

All the search strategies for state space planning already implemented in PDDL4J are available in the package fr.uga.pddl4j.planners.statespace.search.strategy search strategies. Thus, your `solve()` must look like as follows:

```
 1      /**
 2       * Search a solution plan to a specified domain and problem using A*.
 3       *
 4       * @param problem the problem to solve.
 5       * @return the plan found or null if no plan was found.
 6       */
 7      @Override
 8      public Plan solve(final Problem problem) {
 9          // Creates the A* search strategy
10          StateSpaceSearch search =
11  StateSpaceSearch.getInstance(SearchStrategy.Name.ASTAR,
12              this.getHeuristic(), this.getHeuristicWeight(),
13  this.getTimeout());
14          LOGGER.info("* Starting A* search \n");
15          // Search a solution
16          Plan plan = search.searchPlan(problem);
17          // If a plan is found update the statistics of the planner and log
18  search information
19          if (plan != null) {
20              LOGGER.info("* A* search succeeded\n");
21
22  this.getStatistics().setTimeToSearch(search.getSearchingTime());
23
24  this.getStatistics().setMemoryUsedToSearch(search.getMemoryUsed());
25          } else {
                LOGGER.info("* A* search failed\n");
            }
            // Return the plan found or null if the search fails.
            return plan;
        }
```

First, we create an instance of the search strategy for the problem to solve and then, we try to find a plan for this problem.

Note

If you need to get the goal node for printing for instance returned by the search strategy you can replace the call to `searchPlan()` line 14 by:

```
final Node goal = search.searchSolutionNode(problem);
Planner.getLogger().trace(problem.toString(goal));
Plan plan = search.extractPlan(goal, problem);
```

Now, your planner is ready to solve problems. After compiling the project run your planner with the command line to make a test:

```
java -cp classes:lib/pddl4j-4.0.0.jar fr.uga.pddl4j.examples.ASP
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-
automatic/domain.pddl
     src/test/resources/benchmarks/pddl/ipc2002/depots/strips-automatic/p01.pddl
     -e FAST_FORWARD
     -w 1.2
     -t 1000
```

The output should be:

```
parsing domain file "domain.pddl" done successfully
parsing problem file "p01.pddl" done successfully

problem instantiation done successfully (90 actions, 46 fluents)

* Starting A* search
* A* search succeeded

found plan as follows:

00: (       lift hoist0 crate1 pallet0 depot0) [0]
01: ( lift hoist1 crate0 pallet1 distributor0) [0]
02: (        load hoist0 crate1 truck1 depot0) [0]
03: (        drive truck1 depot0 distributor0) [0]
04: (  load hoist1 crate0 truck1 distributor0) [0]
05: (unload hoist1 crate1 truck1 distributor0) [0]
06: (  drive truck1 distributor0 distributor1) [0]
07: ( drop hoist1 crate1 pallet1 distributor0) [0]
08: (unload hoist2 crate0 truck1 distributor1) [0]
09: ( drop hoist2 crate0 pallet2 distributor1) [0]

time spent:       0,02 seconds parsing
                  0,03 seconds encoding
                  0,01 seconds searching
                  0,07 seconds total time

memory used:      0,00 MBytes for problem representation
                  0,00 MBytes for searching
                  0,00 MBytes total
```

Step 5. Write your own A* search strategy

Before writing your own A* search strategy, you need to create a class `Node`. This class represents a node of search tree developed by A*.

Step 5.1 Writing your own class Node

For state space planning a Node is a data structure with 5 components:

1. A state, i.e., the state in the state space to which the node corresponds;
2. A parent node, i.e., the node in the search tree that generated this node;
3. An action, i.e., the action that was applied to the parent node to produce this node;
4. A cost, i.e., the cost of the path from the initial state to the node, as indicated by the parent pointer; and
5. A heuristics value, i.e., a estimation of the cost from this node to a solution one.

The easiest way to write your own node class is to inherit the State class that models a state in a compact way. To do so, start creating a file `Node.java` in the repertory `src/fr/uga/pddl4j/examples/asp/` and copy and paste the the skeleton of the class `Node` is given below:

```java
package fr.uga.pddl4j.examples.asp;

import fr.uga.pddl4j.problem.State;

/**
 * This class implements a node of the tree search.
 *
 * @author D. Pellier
 * @version 1.0 - 02.12.2021
 */
public final class Node extends State {

    /**
     * The parent node of this node.
     */
    private Node parent;

    /**
     * The action apply to reach this node.
     */
    private int action;

    /**
     * The cost to reach this node from the root node.
     */
    private double cost;

    /**
     * The estimated distance to the goal from this node.
     */
    private double heuristic;

    /**
     * The depth of the node.
     */
    private int depth;

    /**
     * Creates a new node from a specified state.
     *
     * @param state the state.
     */
    public Node(State state) {
        super(state);
    }

    /**
     * Creates a new node with a specified state, parent node, operator,
     * cost and heuristic value.
     *
     * @param state     the logical state of the node.
     * @param parent    the parent node of the node.
     * @param action   the action applied to reached the node from its
parent.
     * @param cost      the cost to reach the node from the root node.
     * @param heuristic the estimated distance to reach the goal from the
node.
     */
    public Node(State state, Node parent, int action, double cost, double
heuristic) {
        super(state);
        this.parent = parent;
        this.action = action;
        this.cost = cost;
        this.heuristic = heuristic;
        this.depth = -1;
    }

    /**
     * Creates a new node with a specified state, parent node, operator,
cost,
     * depth and heuristic value.
     *
     * @param state     the logical state of the node.
     * @param parent    the parent node of the node.
     * @param action    the action applied to reached the node from its
parent.
     * @param cost      the cost to reach the node from the root node.
     * @param depth     the depth of the node.
     * @param heuristic the estimated distance to reach the goal from the
node.
     */
    public Node(State state, Node parent, int action, double cost, int
depth, double heuristic) {
        super(state);
        this.parent = parent;
        this.action = action;
        this.cost = cost;
        this.depth = depth;
        this.heuristic = heuristic;
    }

    /**
     * Returns the action applied to reach the node.
     *
     * @return the action applied to reach the node.
     */
    public final int getAction() {
        return this.action;
    }

    /**
     * Sets the action applied to reach the node.
     *
     * @param action the action to set.
     */
    public final void setAction(final int action) {
        this.action = action;
    }

    /**
     * Returns the parent node of the node.
     *
     * @return the parent node.
     */
    public final Node getParent() {
        return parent;
    }

    /**
     * Sets the parent node of the node.
     *
     * @param parent the parent to set.
     */
    public final void setParent(Node parent) {
        this.parent = parent;
    }

    /**
     * Returns the cost to reach the node from the root node.
     *
     * @return the cost to reach the node from the root node.
     */
    public final double getCost() {
        return cost;
    }

    /**
     * Sets the cost needed to reach the node from the root node.
     *
     * @param cost the cost needed to reach the node from the root nod to set.
     */
    public final void setCost(double cost) {
        this.cost = cost;
    }

    /**
     * Returns the estimated distance to the goal from the node.
     *
     * @return the estimated distance to the goal from the node.
     */
    public final double getHeuristic() {
        return heuristic;
    }

    /**
     * Sets the estimated distance to the goal from the node.
     *
     * @param estimates the estimated distance to the goal from the node to set.
     */
    public final void setHeuristic(double estimates) {
        this.heuristic = estimates;
    }

    /**
     * Returns the depth of this node.
     *
     * @return the depth of this node.
     */
    public int getDepth() {
        return this.depth;
    }

    /**
     * Set the depth of this node.
     *
     * @param depth the depth of this node.
     */
    public void setDepth(final int depth) {
        this.depth = depth;
    }

    /**
     * Returns the value of the heuristic function, i.e.,
     * <code>this.node.getCost() + this.node.getHeuristic()</code>.
     *
     * @param weight the weight of the heuristic.
     * @return the value of the heuristic function, i.e.,
     * <code>this.node.getCost() + this.node.getHeuristic()</code>.
     */
    public final double getValueF(double weight) {
        return weight * this.heuristic + this.cost;
    }

}
```

Step 5.2 Writing your own A* search

A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance traveled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node (see the pseudocode A* for more details).

At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes $f(n) = g(n) + h(n)$ where :

- *n* is the last node on the path,
- *g(n)* is the cost of the path from the start node to *n*, and
- *h(n)* is a heuristic that estimates the cost of the cheapest path from *n* to the goal.

For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node.

Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open set or fringe. At each step of the algorithm, the node with the lowest *f(x)* value is removed from the queue, the *f* and *g* values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower *f* value than any node in the queue (or until the queue is empty). The *f* value of the goal is then the length of the shortest path, since *h* at the goal is zero in an admissible heuristic. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node (see the `extract` procedure below).

Consider the implementation of A* now with PDDL4J and the new `solve()` procedure:

```java
    /**
     * Search a solution plan for a planning problem using an A* search
strategy.
     *
     * @param problem the problem to solve.
     * @return a plan solution for the problem or null if there is no
solution
     * @throws ProblemNotSupportedException if the problem to solve is not
supported by the planner.
     */
    public Plan astar(Problem problem) throws ProblemNotSupportedException
{
        // Check if the problem is supported by the planner
        if (!this.isSupported(problem)) {
            throw new ProblemNotSupportedException("Problem not
supported");
        }

        // First we create an instance of the heuristic to use to guide
the search
        final StateHeuristic heuristic =
StateHeuristic.getInstance(this.getHeuristic(), problem);

        // We get the initial state from the planning problem
        final State init = new State(problem.getInitialState());

        // We initialize the closed list of nodes (store the nodes
explored)
        final Set<Node> close = new HashSet<>();

        // We initialize the opened list to store the pending node
according to function f
        final double weight = this.getHeuristicWeight();
        final PriorityQueue<Node> open = new PriorityQueue<>(100, new
Comparator<Node>() {
            public int compare(Node n1, Node n2) {
                double f1 = weight * n1.getHeuristic() + n1.getCost();
                double f2 = weight * n2.getHeuristic() + n2.getCost();
                return Double.compare(f1, f2);
            }
        });

        // We create the root node of the tree search
        final Node root = new Node(init, null, -1, 0,
heuristic.estimate(init, problem.getGoal()));

        // We add the root to the list of pending nodes
        open.add(root);
        Plan plan = null;

        // We set the timeout in ms allocated to the search
        final int timeout = this.getTimeout() * 1000;
        long time = 0;

        // We start the search
        while (!open.isEmpty() && plan == null && time < timeout) {

            // We pop the first node in the pending list open
            final Node current = open.poll();
            close.add(current);

            // If the goal is satisfied in the current node then extract
the search and return it
            if (current.satisfy(problem.getGoal())) {
                return this.extractPlan(current, problem);
            } else { // Else we try to apply the actions of the problem to
the current node
                for (int i = 0; i < problem.getActions().size(); i++) {
                    // We get the actions of the problem
                    Action a = problem.getActions().get(i);
                    // If the action is applicable in the current node
                    if (a.isApplicable(current)) {
                        Node next = new Node(current);
                        // We apply the effect of the action
                        final List<ConditionalEffect> effects =
a.getConditionalEffects();
                        for (ConditionalEffect ce : effects) {
                            if (current.satisfy(ce.getCondition())) {
                                next.apply(ce.getEffect());
                            }
                        }
                        // We set the new child node information
                        final double g = current.getCost() + 1;
                        if (!close.contains(next)) {
                            next.setCost(g);
                            next.setParent(current);
                            next.setAction(i);
                            next.setHeuristic(heuristic.estimate(next,
problem.getGoal()));
                            open.add(next);
                        }
                    }
                }
            }
        }

        // Finally, we return the search computed or null if no search was
found
        return plan;
    }
```

The method `isSupported()` tests if the problem to solved is supported by the method `solve()`. The planner ASP can solve only ADL problem. The code to test if the problem is ADL is given below:

```
        /**
         * Returns if a specified problem is supported by the planner. Just
    ADL problem can be solved by this planner.
         *
         * @param problem the problem to test.
         * @return <code>true</code> if the problem is supported
    <code>false</code> otherwise.
         */
        @Override
        public boolean isSupported(Problem problem) {
            return
    (problem.getRequirements().contains(RequireKey.ACTION_COSTS)
                    || problem.getRequirements().contains(RequireKey.CONSTRAINTS)
                    ||
    problem.getRequirements().contains(RequireKey.CONTINOUS_EFFECTS)
                    ||
    problem.getRequirements().contains(RequireKey.DERIVED_PREDICATES)
                    ||
    problem.getRequirements().contains(RequireKey.DURATIVE_ACTIONS)
                    ||
    problem.getRequirements().contains(RequireKey.DURATION_INEQUALITIES)
                    || problem.getRequirements().contains(RequireKey.FLUENTS)
                    ||
    problem.getRequirements().contains(RequireKey.GOAL_UTILITIES)
                    ||
    problem.getRequirements().contains(RequireKey.METHOD_CONSTRAINTS)
                    ||
    problem.getRequirements().contains(RequireKey.NUMERIC_FLUENTS)
                    ||
    problem.getRequirements().contains(RequireKey.OBJECT_FLUENTS)
                    || problem.getRequirements().contains(RequireKey.PREFERENCES)
                    ||
    problem.getRequirements().contains(RequireKey.TIMED_INITIAL_LITERALS)
                    || problem.getRequirements().contains(RequireKey.HIERARCHY))
                    ? false : true;
        }
```

The method `extractPlan()` extracts a solution plan from the search space by backward chaining the path from the goal node to the root node. The code is given below:

```java
    /**
     * Extracts a search from a specified node.
     *
     * @param node    the node.
     * @param problem the problem.
     * @return the search extracted from the specified node.
     */
    private Plan extractPlan(final Node node, final Problem problem) {
        Node n = node;
        final Plan plan = new SequentialPlan();
        while (n.getAction() != -1) {
            final Action a = problem.getActions().get(n.getAction());
            plan.add(0, a);
            n = n.getParent();
        }
        return plan;
    }
```

Finally, you have to change the call to the method `searchPlan()` in the method `solve()` by the explicit call to your `astar()` procedure. Your new `solve()` method is:

```java
/**
 * Search a solution plan to a specified domain and problem using A*.
 *
 * @param problem the problem to solve.
 * @return the plan found or null if no plan was found.
 */
@Override
public Plan solve(final Problem problem) {
    LOGGER.info("* Starting A* search \n");
    // Search a solution
    final long begin = System.currentTimeMillis();
    final Plan plan = this.astar(problem);
    final long end = System.currentTimeMillis();
    // If a plan is found update the statistics of the planner
    // and log search information
    if (plan != null) {
        LOGGER.info("* A* search succeeded\n");
        this.getStatistics().setTimeToSearch(end - begin);
    } else {
        LOGGER.info("* A* search failed\n");
    }
    // Return the plan found or null if the search fails.
    return plan;
}
```

Step 6. Make your planner configurable by programming

By default, your planner is configurable by programming (see MISSING REF for more details) because it inherits the class AbstractPlanner. But only for the common configurable properties of all planners, i.e., the domain, the problem, the timeout and the log level.

> **Note**
>
> The common configurable properties and their values are defined in the interface Planner.

In order to allow the new properties of your planner to be configurable by programming, you have to :

1. declare for each new property a name and a default value
2. redefined the setter and the getter method to set and get the configuration of your planner
3. redefined a method `getDefaultConfiguration()`
4. redefined the method `hasValidConfiguration()`
5. redefined the constructors of your planner and deal with the class PlannerConfiguration

Step 6.1 Declaration of new properties

In the case of your planner, you have two new properties that you want configure: the heuristic and the weight the weight associated with it. The following code:

```
    /**
     * The HEURISTIC property used for planner configuration.
     */
    public static final String HEURISTIC_SETTING = "HEURISTIC";

    /**
     * The default value of the HEURISTIC property used for planner
configuration.
     */
    public static final StateHeuristic.Name DEFAULT_HEURISTIC =
StateHeuristic.Name.FAST_FORWARD;

    /**
     * The WEIGHT_HEURISTIC property used for planner configuration.
     */
    public static final String WEIGHT_HEURISTIC_SETTING =
"WEIGHT_HEURISTIC";

    /**
     * The default value of the WEIGHT_HEURISTIC property used for planner
configuration.
     */
    public static final double DEFAULT_WEIGHT_HEURISTIC = 1.0;
```

Step 6.2 Setting and getting the configuration of your planner

To deal with the two properties and make your planner configurable by programming, it is necessary to redefined the setter and the getter of the class AbstractPlanner. This can be done in your case using the code below:

```
    /**
     * Returns the configuration of the planner.
     *
     * @return the configuration of the planner.
     */
    @Override
    public PlannerConfiguration getConfiguration() {
        final PlannerConfiguration config = super.getConfiguration();
        config.setProperty(ASP.HEURISTIC_SETTING,
this.getHeuristic().toString());
        config.setProperty(ASP.WEIGHT_HEURISTIC_SETTING,
Double.toString(this.getHeuristicWeight()));
        return config;
    }

    /**
     * Sets the configuration of the planner. If a planner setting is not
defined in
     * the specified configuration, the setting is initialized with its
default value.
     *
     * @param configuration the configuration to set.
     */
    @Override
    public void setConfiguration(final PlannerConfiguration configuration)
{
        super.setConfiguration(configuration);
        if (configuration.getProperty(ASP.WEIGHT_HEURISTIC_SETTING) ==
null) {
            this.setHeuristicWeight(ASP.DEFAULT_WEIGHT_HEURISTIC);
        } else {

this.setHeuristicWeight(Double.parseDouble(configuration.getProperty(
                ASP.WEIGHT_HEURISTIC_SETTING)));
        }
        if (configuration.getProperty(ASP.HEURISTIC_SETTING) == null) {
            this.setHeuristic(ASP.DEFAULT_HEURISTIC);
        } else {

this.setHeuristic(StateHeuristic.Name.valueOf(configuration.getProperty(
                ASP.HEURISTIC_SETTING)));
        }
    }
```

The code is quite simple. It call the method `getConfigration()` and `setConfiguration()` from the parent class AbstractPlanner. and set of get the new properties to an instance of the class PlannerConfiguration.

Step 6.3 Defining the default configuration of your planner

By convention all planner have a static method which returns the default configuration of a planner. In your case, the method `getDefaultConfiguration()` calls the eponymous method from the parent class AbstractPlanner. Then, in the same way as the previous method `getConfiguration()` it creates an instance of the class PlannerConfiguration with the default values.

```
     *
     * @return the default arguments of the planner.
     * @see PlannerConfiguration
     */
    public static PlannerConfiguration getDefaultConfiguration() {
        PlannerConfiguration config = Planner.getDefaultConfiguration();
        config.setProperty(ASP.HEURISTIC_SETTING,
ASP.DEFAULT_HEURISTIC.toString());
        config.setProperty(ASP.WEIGHT_HEURISTIC_SETTING,
            Double.toString(ASP.DEFAULT_WEIGHT_HEURISTIC));
        return config;
    }
```

Step 6.4 Defining the method that checks if a configuration is valid or not

The `hasValideConfiguration()` method calls the eponymous method of the parent class AbstractPlanner and adds the checks on the added properties. For your planner, it checks that the weight associated to the heuristic is strictly greater than 0 and that a heuristic has been chosen among the GoalCostHeuristics already defined in the library

```
    /**
     * Checks the planner configuration and returns if the configuration
is valid.
     * A configuration is valid if (1) the domain and the problem files
exist and
     * can be read, (2) the timeout is greater than 0, (3) the weight of
the
     * heuristic is greater than 0 and (4) the heuristic is a not null.
     *
     * @return <code>true</code> if the configuration is valid
<code>false</code> otherwise.
     */
    public boolean hasValidConfiguration() {
        return super.hasValidConfiguration()
            && this.getHeuristicWeight() > 0.0
            && this.getHeuristic() != null;
    }
```

Step 6.5 Redefining the constructors of your planner

Redefining the constructors of your planner to create your planner from a PlannerConfiguration can be done with the code below:

```java
    /**
     * Creates a new A* search planner with the default configuration.
     */
    public ASP() {
        this(ASP.getDefaultConfiguration());
    }

    /**
     * Creates a new A* search planner with a specified configuration.
     *
     * @param configuration the configuration of the planner.
     */
    public ASP(final PlannerConfiguration configuration) {
        super();
        this.setConfiguration(configuration);
    }
```

Note

The final code of the planner code is available here.