UNIVERSIDADE DA CORUÑA

Grado en Ingeniería Informática
Intelligent systems

Assignment 1: Search strategies

DOCUMENTATION

The code of the project is provided as an IntelliJ IDEA project. This document describes how to use it. It also contains a description of the class structure, which must be preserved in all implementations performed.

Notice: The use of IntelliJ IDEA is not mandatory.

- **How to run the example code:**
  a) If you don't have IntelliJ IDEA, download it from https://www.jetbrains.com/es-es/idea/download/. There's a free version, named Community, that has all the functionality needed for this project.
  b) Open IntelliJ IDEA and import the project. To do so, launch IntelliJ IDEA and select "Import" in the initial screen. A file selection window will appear, which you will use to select the folder where you downloaded the project code to. Once selected, the project will be shown in IntelliJ IDEA's main window.
  c) Verify that you can run the example code by double-clicking the "`example/Main.java`" file in IntelliJ IDEA's main window. The code contained in that class will appear. Look for the "main" method. To the left of it, there should be a green ▶ icon. Click it to launch the execution of the method. A sequence of steps taken to solve the problem (as seen on the SGA2 slides) should appear in the Console output window at the bottom of the screen.
  d) You can find more information about how to use IntelliJ IDEA, including several tutorials, in the official page: https://www.jetbrains.com/es-es/idea/resources/

- **Design description**
  The project provided has the following classes, which here we group by their role. Here's a brief description of each one:
  1. **Problem representation:** The code includes three classes that constitute a template for the problem representation (see SGA1). In order to represent a given problem, you must create subclasses of each one (see example).
     - **State class:** This class defines the representation of a state of a given problem. It is an abstract class, so you can't create objects of this class. Instead, you must create a subclass that inherits from State and implements the required methods. You will then be able to create objects of this new subclass (ex. see class VacuumCleanerState in `example/VacuumCleanerProblem.java` and how it's used in `example/Main.java`).
     - **Action class:** This class represents an action that can be performed on a state for a given problem. The Action class is also abstract so, in order to create objects, you must define a subclass that implements the following methods:
       - ***isApplicable(State st):*** This method determines whether a particular action can be performed on the state that it receives as a parameter. The purpose of this method is to encode the preconditions of actions.
       - ***applyTo(State st):*** This method returns the state that is the result of applying the action on which it is called to the state passed as a parameter. Its purpose is to encode the result of applying each action (transition model).
       - ***getCost():*** Returns the cost of using the action that calls the method.
     - **SearchProblem class:** This class contains the remainder of the search problem definition, which includes the initial state and the goal test. It's an abstract class so, to create instances you must define subclasses that are specific to the problem that you are representing (see example). Subclasses of SearchProblem usually also declare subclasses of State and Action. The following methods are included in this class:
       - ***isGoal(State st):*** Implementation of the goal test. It checks whether the state received as a parameter can be considered a goal in the given problem. It is an abstract method, so all subclasses must override it to provide an implementation specific to the represented problem.
       - ***actions(State st):*** Returns a list of actions that are available for the state received as a parameter. It is an abstract method, so all subclasses must override it to provide an implementation specific to the represented problem.
       - ***result(State st, Action a):*** Returns the result of applying action a to state *st*. This method delegates to the Action class so there's no need to override it for a specific problem.

2. **Search strategy implementation:** The provided code includes two classes, one for uninformed search strategies and one for informed search strategies (with an extra class for encoding heuristics). The classes are as follows:
    - **<u>SearchStrategy</u>:** EsThis interface ensures that all subclasses have a method that solves search problems. All uninformed search strategies must implement this interface (see `example/Strategy4.java`). Methods:
        - ***solve(SearchProblem p):*** Obtains a solution (if that's possible) to the problem received as a parameter. To do so, it performs a systematic search strategy that all subclasses must implement in this method.
    - **<u>InformedSearchStrategy</u>:** Interface similar to SearchStrategy but that, since it deals with informed search, includes a heuristic (as a parameter) in the *solve* method.
    - **<u>Heuristic</u>:** Abstract class that includes a method to compute the value of the heuristic function for a given state (received as a parameter). All subclasses must implement said method.

3. **Example:** The code includes a sample implementation of the basic search strategy #4 (as described in SGA2) that is used to solve the vacuum cleaner world problem (as represented in SGA1).
    - **<u>VacuumCleanerProblem</u>:** Subclass of <u>SearchProblem</u> that encodes the formalization of the vacuum cleaner world problem as described un SGA1. This class defines, in turn, the <u>VacuumCleanerState</u> class (subclass of <u>State</u>) and the <u>VacuumCleanerAction</u> class (subclass of <u>Action</u>).
    - **<u>Strategy4</u>:** Implementation of the basic search strategy described in SGA2. This class implements the <u>SearchStrategy</u> interface and implements the *solve* as described in SGA2.
    - **<u>Main:</u>** Main class that facilitates launching an execution. The main method instantiates a <u>VacuumCleanerProblem</u> object by setting a specific <u>VacuumCleanerState</u> as the initial state. Then it instantiates a <u>Strategy4</u> object, which is then used to solve the previously declared problem. Lastly, the goal state found by the *solve* method is printed on screen.