Assignment 3: A goal oriented planning agent

Joost Broekens

November 3, 2023

1 INTRODUCTION

In this assignment you will program your own intelligent agent. The agent is a goal-oriented agent that has to escape a maze by finding the exit. The maze consists of locations that are either occupied by a wall, a key or a door. The agent starts at a starting location, and needs to find the exit. The agent performs actions to do so, perceives the world, and reasons about the world. For this it follows the standard sense, think, decide, act cycle.

To start, you need to first install Eclipse, the standard Java SDK. Download it here: `https://www.eclipse.org/downloads/` Then you need to download the project .zip file for this assignment from Brightspace. Unzip the project into a folder of your choice. Then open Eclipse and import the project (File>Import>Existing projects). Choose the folder you unzipped the file to. This will open the project in Eclipse.

1.1 CODE STRUCTURE

In the project you will find three packages. A package is a *name space* to group classes together, just like in python or c++ when you use import or include. One package contains the logic classes (KB, Predicate, Sentence, Term). One contains the agent's environment classes (Location and Maze). One contains the agent classes (Agent, MyAgent, RunMe, Plan). *It is important to remember that the only class you may and need to work in is MyAgent! Do not create helper classes as this will reduce algorithm readability. You also not change the method headers in any way, otherwise our test will not run.*

The logic package provides you with the functionality to read in and perform basic operations on our own simplified first order logic language, *sfol*, that we invented and are going to use for the agent and knowledge bases. KB is the main knowledge base class, and essentially is a collection of Sentences. A Sentence is a class that contains one logical statement. This is essentially the same as a Prolog rule. However, it has a slightly different syntax. The Predicate class contains one predicate. Again similar to Prolog. The Term class contains one term, which is either a variable or a constant. The syntax and semantics of sentences, predicates and terms is detailed below in the logic section.

The environment classes are rather straightforward. The package contains a Maze class which enables you to read in a maze from a file, generate percepts from-, and execute actions in that maze. The Location class is a helper class to represent one location in the maze.

The rest of the classes are; RunMe: the class with the main program loop; Agent: which is an abstract class (meaning you still need to implement things in the class you extend) containing the basic sense, think, decide, act functionality for the agent; and Plan: a helper class to represent a plan as a sequence of action predicates which you need later when you implement your planner.

The MyAgent class is the class you work in. As you can see it has 6 empty methods that you need to program. These empty methods correspond to the 6 abstract methods in Agent. MyAgent extends the Agent class, which means it has to implement those 6 methods.

Finally, there is a folder in your project in which you can find a prison.txt file (the maze), and several knowledge base files including actions.txt, percepts.txt, programs.txt, and three test files, family1-3.txt, that you may use to test your inference algorithm.

To run the code in Eclipse, click the green arrow with the popup text "Run RunMe". When you run the code now, it should load all the empty agent programs, but otherwise do nothing. To stop running, click the red "Terminate" button in the Java console.

Important. You should now review the topics we covered on search, predicate logic, inference with predicate logic, planning and BDI agents. If you have not done so, please do this first. You will not be able to finish this assignment without a solid understanding of these topics.

Important. Invest time in understanding the existing packages and classes to get an good idea of what they do, and, the many helper methods that are already available to you. This is extremely important as this will facilitate many things you need to do.

2 THE AGENT AND ITS ENVIRONMENT

The agent is a BDI agent. It has three knowledge basis: beliefs, desires and intentions. Further it has three rule sets, also knowledge bases: percept rules (that process percepts from the world), program rules (your main agent logic), and action rules (post conditions of actions). Percept rules are evaluated in the sense step, program rules in the think step and action rules in the act step. Find the knowledge bases in the Agent class and understand how they are defined and instantiated. Also have a look at the sense, think, act methods.

2.1 SIMPLIFIED FIRST ORDER LOGIC

Here we define *sfol*, our simplified first order logic language used by the agent as knowledge representation system. A knowledge base consists of a list of sentences in *sfol*. A Sentence consist of conditions and conclusions of Predicates. A Predicate consist of a name part and zero or more Terms.

Terms are either a variable or a constant. Variables are denoted by starting the term with a capital letter. Constants start lower case. Predicates are of the form:

```
{operator}name{(Term{,Term...})}
```

Where the brackets denote an optional part and the ... denote an optional repetition of the previous. Sentences are of the form $Body > Conclusion$:

```
{Predicate{&Predicate...} >{Predicate{&Predicate...}
```

So, in contrast to Prolog, we are allowed to have multiple conclusions. However *sfol* does not allow lists (pfew!). There are some special characters that denote operations in *sfol*. These are:

```
+ = add fact to believe base
- = delete fact from belief base
* = add goal to desire base (called adopt)
~ = delete goal from desire (called drop, happens automatically with +)
_ = add fact to intentions base (so an action)
```

Operations are placed immediately before an *sfol* term. They are executed whenever a predicate is processed by the agent (in the processFacts method in Agent, see later). Operators are not relevant for the logic and inference method you will program. You can consider operations as actions, to be processed by the agent into the three different knowledge bases it has. Operators can not appear in the conditions of a sentence because they are not facts but operations on predicates. You can see this in the Sentence.parse(...) method of the Sentence class, which checks if the predicate read as part of the conditions is indeed not an action.

Finally, there are three reserved predicates:

```
=(X,Y)   = resolves to true if Term X equals to Term Y
!=(X,Y) = resolves to true if Term X not equals to Term Y
!pred(X,...) = resolves to true if pred(X,...) is not true.
```

Arithmetic, as you can see, is not allowed either (pfew!).

So with *sfol* we can write down things like:

```
parent(joost,leon)
parent(joost,sacha)
parent(peter,joost)
parent(Z,X)&parent(Z,Y)&!=(X,Y)>sibling(X,Y)
parent(X,Y)&parent(Y,Z)>grandparent(X,Z)
```

But also more complex rules involving operations, actions and goals:

```
#If hungry and there is food then add eat to the intention base.
hungry&agent(X)&food(X)>_eat(X)
#If there is no food here, then adopt the goal to move to the food
hungry&agent(X)&food(Y)&!=(X,Y)>*agent(Y)
#An example of an action post-condition rule where two facts are
#retracted from the belief base as a result of
#a successfully executed action.
eat(X)>-food(X)&-hungry
```

This resembles Prolog a lot, but it is some ways richer (multiple conclusions, goals, and actions), and in some way more restricted (no lists, no arithmetic).

2.2 ENVIRONMENT

The agent's environment is a maze defined in prison.txt (example below). The first line gives the maze dimensions. Each location is defined by a character. A # is a wall, a lower case letter is a key, that can be used to open the capital letter door. The * is the starting location opf the agent and the $ is the exit. To define a location with nothing special, the <space> character is used.

```
6 8
######
#* A #
#a # #
#### #
#    #
# ##B#
#b##$#
######
```

All of the environment's dynamics are dealt with in the Maze and Location classes. You do not have to code this yourself. The Maze keeps track of the agent and keeps a list of keys that have been picked up already (this list is used to check if an action is valid, the list is not available to your agent, that is something you need to resolve in your logic). Further, the Maze class stores all the locations, and allows the agent to execute actions at those locations. In predicate form, these actions are:

```
#Moves the agent from Location X to Y, if possible
_goto(X,Y)

#Grabs item I at Location X, if possible
_grab(X,I)

#Opens all doors at Location X with key K, if possible
_open(X,K)

#Look does nothing (idle).
_look
```

The environment checks if an action is possible. For example, $\_open(1\_2, a)$ is not possible if the agent is not at 1_2, it does not have key *a* or there is no locked door at 1_2. An action returns true if it succeeds, otherwise false. The method Maze.executeAction(Predicate action) should be called with an action predicate. If the agent arrives at the exit location, the maze will stop all processing and print a celebration message.

The environment also generates percepts. This is done with the method Agent.generatePercepts(), which returns a knowledge base (KB class) filled with the currently perceived percepts. These percepts can be:

```
# the current location, e.g., at(1_2) (agent at 1,2)
at(X)

#added if the current location is the exit location
exit

#added if a key K is present at the current location, e.g. key(a)
key(K)

#added if a locked door is present
#that blocks a passage that needs key K, e.g. locked(a)
locked(K)

#added if a passage from the current location to Y is present,
#e.g., passage(2_2)
passage(Y)
```

2.3 The agent cycle

The agent cycles through 4 steps. Before each cycle the list of intentions is cleared. The steps are detailed now.

In the sense step, the world generates percepts and returns to the agent a knowledge base called the percept base. After this, inference is done on the union of the percept rules, the agent's beliefs and the percept base, returning the full list of inferred facts. After inference, operators are processed by the agent in Agent.processFacts(result of inference) to add or delete facts into the appropriate knowledge bases (beliefs, desires, intentions). Every cycle your percept base changes, reflecting the current state the agent is in. If you want to remember things from previous percepts, you have to build rules to do so. This is the purpose of the percept rules. Such rules typically add or delete beliefs, this is done with + and - operators.

In the think step, your agent evaluates the program rules. It does so by doing inference on the union of the program rules and the belief base. The program rules define how and when actions need to be added to the intentions, goals need to be addopted and beliefs needs to be added or deleted. Actions are defined by the operators (see above). The conditions of a rule are the preconditions. The conclusion of the rule contains the action(s) and/or facts that can be derived with inference. After inference, operators are again processed, which in this case also results in adding new intentions to the intention base.

In the decide phase, the agent selects an action from the collection of action predicates in the agent's intention base. It returns the action chosen. This phase will make use of the planning algorithm you develop. However, for testing purposes, you can call set the boolean flag Agent.HUMAN_DECISION to true. This allows you to pick the action by typing a number. To test what happens, you can play the agent choosing an action by typing in the action nr (1...n) in the Eclipse console. Note that this only works after you implemented your inference engine (section 4).

In the act phase, the agent will try one action in the environment. If the action fails, the agent will not have executed the action in the world, and the agent will immediately go to the next cycle. If the action succeeds, the action rules are evaluated using inference on the union of the action rules, the beliefs and the desires. After inference, operators are again processed. This is how action post-conditions are processed. Note that our agent does not process or allow intention operators in this phase. These would be cleared anyway at the start of the next cycle. So, environment actions cannot be post-conditions of actions.

To help you debug, you can set the Agent.DEBUG flag to true on top of the Agent class. If you do, the method Agent.processFacts(...) will print out all facts that are added and deleted from each knowledge base in all phases.

3 Programming the agent in SFOL

In this section you will program a simple agent logic in *sfol*. You will define percept rules, program rules and action rules. Use the percepts.txt, program.txt and actions.txt files to do so.

Your goal in this section is to end up with a simple, reactive, agent program in *sfol* that will learn a model of the maze, decide what actions are appropriate, and process the postconditions of those actions. No goals yet! You can check if the rules you develop parse correctly by running the java project. Of course nothing happens yet, as for this you need to implement the inference and planning algorithms. However that is for later.

3.1 Questions

1. Define four percept rules that process the environment's percepts (see above) and do the following: build a map of the maze in the belief base (store links between locations, whether there is a key at a location, and whether there is a locked door);and, store the current location of the agent in the belief base. Explain your rules.
2. Define three program rules that will propose different actions (see above): open a door at a location with a specific key, whenever you have that key and you are at such location; grab a key whenever you see one; move to a new location if a link to that location is known. Explain your rules.
3. Define three action rules that correctly process the postconditions for your three actions, such that the belief base is updated in a logically correct manner. Explain your rules.

Don't forget to check if your rules parse correctly by running the java project.

4 Programming the inference algorithm

Now we begin with the real algorithmic work in Java. It is useful to look at the forward chaining algorithm in the book, and understand what happens there, as this is what you will be implementing in 4 steps. You will implement your code in the MyAgent class. HINT: you do not *need* to change or add methods, everything you need is there, your 4 methods do not need any global variables as everything they need is in the arguments, and everything they produce is in the returned result. HINT2: forget about the negation predicate ! until you arrive at question 8.

4.1 Questions

4. Substitution is the basic ingredient of inference, as you know. We need to first be able to substitute variables in a predicate, given a particular substitution, and create a new,

substituted predicate. Implement MyAgent.substitute(...). There is no need to explain the code, it is straightforward.

5. The next step is unification. We need this because we want to know for which substitution s a particular predicate p unifies with another predicate q. Implement MyAgent.unifiesWith(...). Explain your code.
6. Now we are ready to implement a recursive method to find *all* valid substitutions for a vector of conditions of a rule, given a set of facts. MyAgent.findAllSubstitions(...) has a complex set of arguments, so have a good look at. Try to understand why these arguments are passed in the way they are, before starting your implementation. Explain in your report how you dealt with the reserved predicates $= (X, Y)$ and $! = (X, Y)$.
7. Now you are ready to implement MyAgent.forwardChain(KB). This has become a simple matter of adding to a collection of facts those facts that result from applying to to the conclusion of that sentence all valid substitutions for the conditions of that sentence, for all sentences in the KB, and doing this until no new facts are produced. HINT: remember that fully bound operators need to be added as facts to the collection during inference, but, these facts cannot be used as facts to unify with condition predicates (as they are operators, not true facts). HINT: the exception is the addition action. Explain in your report how you dealt with the belief addition operator during inference.
8. Add the ability to deal with the negation operator ! to MyAgent.findAllSubstitutions(...) and MyAgent.unifiesWith(...). You may assume that a negated predicate does not *proof* the predicate is false. Instead you may implement a shortcut based on the predicate unifying with your current fact list when you encounter it in a condition. If you coded the 4 methods nicely, this addition is relatively straightforward. Explain how you dealt with it in your report.

Test your inference method on the family rules as defined in the family1-3.txt files. Load each rule set (see code in RunMe) and run. When you understand the beliefs that it generates, you can continue with your agent program (percepts.txt, program.txt, actions.txt). You should be able to set the agent to manual action selection by setting the Agent.HUMAN_DECISION to true. See how it produces intentions. Play around with it until you are convinced that your inference works as expected and your agent program infers what it should do. Can *you* find the exit by moving around the agent in the maze?

5 PLANNING

Now that you have played around with the agent, set it back to the agent making decisions. If you run it, you will see that the agent picks a random action from the list of intentions. It has no other way to select an action from the list of intentions. This will change. You will implement an iterative deepening planning algorithm that searches through the agent's belief-action space for the shortest sequence of actions that will make a particular goal predicate true. The goal predicate fed to the planning algorithm is pulled from the desires knowledge base. Once all this is done, the agent will make a plan for each desire, and pick the first action from the plan belonging to the first desire in its desire base. You can inspect the code that does this in the Agent.decide(...) method. Of course your agent program should produce desires, i.e. goals. Let's do that first.

9. Add rules to your program.txt file, so that the agent will adopt appropriate goals. Think about what the agent wants to achieve as a reaction to what happens to it. For example, it might be a good idea to adopt the goal haveKey(K), if you see a door that needs that key. Invent some 3 goals that are useful for the agent to pursue and add them as rules to your program. Also, make sure that you revise your percept and action rules, if needed. Remember that if you set a goal, and you want to plan towards it, you must be able to predict it's effects in terms of action post conditions.

Verify that your agent indeed produces appropriate goals, by setting the agent to manual and debug mode and taking actions yourself. You should be able to see what goals are adopted and dropped in different situations.

Now you are ready to implement the planning algorithm. You implement an iterative deepening algorithm. This has two parts, a recursive depth first search method, MyAgent.depthFirst(...), and the iterative deepening loop around that, MyAgent.idSearch(...).

10. Implement the idSearch outer loop in MyAgent.idSearch(...). It should return a Plan, which is just a helper class to easily cope with a Vector (=list) of Predicates. Note that MyAgent.idSearch(...) is called in Agent.decide(...) with a max depth of 7. Do not change this depth (computational cost explosion). Explain your algorithm.

11. Implement the recursive depth first search algorithm MyAgent.depthFirst(...). Have a good look at the arguments and what it returns. HINT: it helps to view the depth first search as a process that really is going through a search tree with nodes being local belief states, and edges being possible actions in those states. HINT2: make use of think() and act() using the local belief state for the node you are in, using the appropriate knowledge base arguments. You can use *null* if an argument for think() or act() is not relevant during planning. Explain your algorithm.

If all went well, you can now retry your agent, and it should propose plans for each desire the agent has, and execute the first action from the first plan. There is one thing you should probably do: make the goals strategic! That means, the agent should adopt a clever goal if it knows it can achieve it, and, it should probably not adopt another goal before it has reached the first.

12. Make tweaks to your agent's program rules, percepts rules and action rules, so that it comes up with a (near) optimal strategy for the given maze. Be careful though, it should of course work for a different maze of similar difficult as well! Explain your tweaks.

This concludes the assignment. Make sure you answer all questions in a report (pdf). Then upload a *single zip file* containing no folders and ONLY: the report pdf, your MyAgent.java, your percepts.txt, program.txt and your action.txt file. No projects or git repos!. Upload the zip containing the 5 files to the assignment in Brightspace.