CmpE 260 - Principles of Programming Languages
Spring 2022 - Project 1

Deadline: 09 May 2022 23:59

1 Introduction

In this project, you will write a logic for an agent to do several tasks in a very tiny version of Minecraft.

2 Knowledge Base

We have already implemented the mechanics of the game for you in `cmpecraft.pro` and `constants.pro`. You should not modify these files. The map for the game is given in `map.txt`. You can change it to test your agent in different settings.

The main data structure for this project is given in `state/3` predicate. After you load your main file (`main.pro`), you can check the current state with the following query:

```
?- state(AgentDict, ObjectDict, Time).
AgentDict = agent_dict{hp:10, hunger:100000, inventory:bag{}, x:3, y:4},
ObjectDict = object_dict{0:object{hp:3, type:stone, x:2, y:1}, ...}
Time = 1

##########
# S TT  S#
# O TT  C#
#    T  T#
#  @    C#
#C      S#
##########
```

This state is generated from the map file `map.txt`. Here, each character represents a different object in the game:

- `#`: Walls and bedrocks.
- `@`: The agent.
- `S`: Stone.
- `T`: Tree.
- `O`: Food.
- `C`: Cobblestone.

The agent dictionary consists of four attributes: `hp` (hit points), `hunger` (hunger level), `x` (x-axis location), `y` (y-axis location), and `inventory`. The agent can collect different items from different objects, and store them in its inventory. The `inventory` is a dictionary that contains the name of the item as the key, and the number of that item as the value.

The object dictionary holds objects in the environment. Each item in this dictionary is also represented as a dictionary which has four attributes: `hp` (how many left clicks are required to mine this object), `type`, `x`, `y`.

Each object yields an item when they are mined, or chopped, or collected. Namely, if the agent mines a stone (i.e. left clicks to the tile that contain a stone until the stone's `hp` drops below zero), the stone yields three cobblestones. This information is provided in `yields/2` in `constants.pro`.

The agent has (is going to have) four main actions: place a cobblestone, left clicking in a specific direction, crafting items, and moving in a direction. These low-level actions are provided for you. In other words, given an

environment state, you can find the next state after executing one of these actions. You can execute `random_move.` to see the agent executing random actions.

3 Predicates

In this section, we will go over the predicates that you are going to implement.

3.1 manhattan_distance(+List1, +List2, -Distance) 10 points

This predicate will compute the Manhattan distance between two lists that contain two items each. The Manhattan distance is the fancy name for ℓ_1 norm:

$$D([x_1, y_1], [x_2, y_2]) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

All test cases will contain valid `List1` and `List2` (i.e. no empty lists).

Examples:

```
?- manhattan_distance([1, 2], [3, 7], D).
D = 7.
?- manhattan_distance([12, 3], [3, 7], D).
D = 13.
```

3.2 minimum_of_list(+List, -Minimum) 10 points

This predicate will unify `Minimum` with the minimum of `List`. All test cases will contain valid `List` (i.e. no empty lists).

Examples:

```
?- minimum_of_list([3, 7, 1, 4], M).
M = 1.
?- minimum_of_list([34], M).
M = 34.
```

3.3 find_nearest_type(+State, +ObjectType, -ObjKey, -Object, -Distance) 10 points

This predicate will find the closest `ObjectType` in the given `State`, and unify the found object's key (in the `ObjectDict`) with `ObjKey`, unify the found object with `Object`, and unify the Manhattan distance between the object and the agent with `Distance`. Here, `State` is a list: `[AgentDict, ObjectDict, Time]`. If there is no object with the given type, the predicate should be `false`. The following examples are valid for the given example map file.

Examples:

```
?- state(A, O, T), State=[A, O, T], find_nearest_type(State, tree, Key, Obj, Dist).
K = 5,
Obj = object{hp:3, type:tree, x:4, y:2},
Dist = 3.
?- state(A, O, T), State=[A, O, T], find_nearest_type(State, stone, Key, Obj, Dist).
K = 0,
Obj = object{hp:3, type:tree, x:2, y:1},
Dist = 4.
```

3.4 navigate_to(+State, +X, +Y, -ActionList, +DepthLimit) 10 points

This predicate will give an `ActionList` that will navigate the agent to `X` and `Y` location. The number of actions in `ActionList` should not be greater than `DepthLimit`, though, it can be less than the depth limit[1]. All test cases will contain valid `X` and `Y`. The following examples are valid for the given example map file.

[1] If you do not introduce such a depth limit, you might stuck into an infinite loop, depending on your implementation

Examples:

```
?- state(A, O, T), State=[A, O, T], navigate_to(State, 6, 5, ActionList, 3).
false.
?- state(A, O, T), State=[A, O, T], navigate_to(State, 6, 5, ActionList, 4).
ActionList = [go_right, go_right, go_right, go_down]
?- state(A, O, T), State=[A, O, T], navigate_to(State, 6, 5, ActionList, 5).
ActionList = [go_right, go_right, go_right, go_down]
```

3.5 chop_nearest_tree(+State, -ActionList) 10 points

This predicate will generate the necessary actions to find the nearest tree, navigate to that location, and chop it (i.e. left clicking to it four times). After executing the action list, the agent should have the yield of the tree in its inventory. If there is no tree in the state, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, O, T), State=[A, O, T], chop_nearest_tree(State, ActionList).
ActionList = [go_right, go_up, go_up, left_click_c, left_click_c, left_click_c, left_click_c]
```

3.6 mine_nearest_stone(+State, -ActionList) 10 points

This predicate will generate the necessary actions to find the nearest stone, navigate to that location, and mine it (i.e. left clicking to it four times). After executing the action list, the agent should have the yield of the stone in its inventory. If there is no mine in the state, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, O, T), State=[A, O, T], chop_nearest_tree(State, ActionList).
ActionList = [go_up, go_up, go_up, go_left, left_click_c, left_click_c, left_click_c, left_click_c]
```

3.7 gather_nearest_food(+State, -ActionList) 10 points

This predicate will generate the necessary actions to find the nearest food resource, navigate to that location, and collect it (i.e. left clicking to it one time). After executing the action list, the agent should have the yield of the food in its inventory. If there is no food in the state, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, O, T), State=[A, O, T], gather_nearest_food(State, ActionList).
ActionList = [go_up, go_up, go_left, left_click_c]
```

3.8 collect_requirements(+State, +ItemType, -ActionList) 10 points

Given an item type, for example stone_pickaxe, this predicate will generate the necessary actions to collect the required items to craft a stone_pickaxe. After executing the action list, the agent should be able to craft a stone pickaxe. If the current map does not contain all the requirements, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, O, T), State=[A, O, T], gather_nearest_food(State, ActionList).
ActionList = [go_right, go_up, go_up, left_click_c | ...]
```

3.9 find_castle_location(+State, -XMin, -YMin, -XMax, -YMax) 5 points

This predicate will help the agent to find a location to build a castle. Castle is just a fancy name for a three by three cobblestone blocks. An appropriate location for a castle is a three by three area that does not contain any object. If there is no appropriate location, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, 0, T), State=[A, 0, T], find_castle_location(State, XMin, YMin, XMax, YMax).
XMin = 2, YMin = 3, XMax = 4, YMax = 5.
```

3.10 make_castle(+State, -ActionList) 15 points

This predicate will generate an action list to first acquire the necessary requirements for building a castle (i.e. nine cobblestones, so the agent should first mine three stone blocks), then find a castle location, then build the castle by placing cobblestone tiles into a three by three area. If there is no appropriate location or resources, then the predicate should be `false`. The following example is valid for the given example map file.

Examples:

```
?- state(A, 0, T), State=[A, 0, T], make_castle(State, ActionList).
ActionList = [go_up, go_up, go_up, go_left, left_click_c | ...].
```

3.11 Printing states

You can use `execute_actions/3` and `execute_print_actions/3` (same as `execute_actions`) but renders the game state in ASCII) which simulates the action list from the given state and return the final state. For example, `?- state(A, 0, T), State=[A, 0, T], make_castle(State, ActionList), execute_print_actions(State, ActionList, FinalState).`
This will help you see the actions of the agent, and hopefully allow you to debug more easily.

4 Documentation

Please explain what each predicate is for with comments in the code. Codes with no comments might lose points if a predicate is too hard to understand.

5 Submission

You will submit two files: `main.pro`, `feedback.txt`. Your code should be in one file named `main.pro`. First four lines of your `main.pro` file must have exactly the lines below since it will be used for compiling and testing your code automatically:

```
% name surname
% student id
% compiling: yes
% complete: yes
```

The third line denotes whether your code compiles correctly, and the fourth line denotes whether you completed all of the project, which must be `no` if you're doing a partial submission. This whole part must be lowercase and include only the English alphabet. Example:

```
% alper ahmetoglu
% 2012400147
% compiling: yes
% complete: yes
```

We are interested in your feedback about the project. In the `feedback.txt` file, please write your feedback. This is optional, you may leave it empty and omit its submission.

6 Tips and Tricks

Although the project may seem long at the first glance, it can be done in a reasonable amount of time. Therefore, do not panic. Before starting, please carefully examine `cmpecraft.pro` and `constants.pro`. It might be also useful to first run a hard-coded action list, then think about how to generate that action list.

- Do not rush. First think about the requirements, what you need, how you can solve it in a modular way. If you can verbally describe your needs, you can probably implement it easily.
- Try to formalize the problem, then try to convert the logic formulate to Prolog.
- You can use findall/3, bagof/3 or setof/3.
- You can use extra predicates and it is highly recommended. The ones given above are compulsory.
- You can add extra knowledge.
- If a predicate becomes too complex, either divide it into some predicates or take another approach. Use debugging (through trace/1), approach your program systematically.