Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2012-2013

Homework 2

Due date: 25 March 2013, Monday, 23:55

1 Objective

In this homework, you will solve a deterministic search on maze problem. You will be given a maze including several agents, obstacles and foods. Your aim is to find the actions for each agent that will make the agents eat the food. An example maze is shown in Figure 1. There are agents which are represented as small robots; obstacles which are represented as crosses; and foods which are represented as sandwiches. The details of the method you will follow is explained in Section 2.

2 Specifications

1. Maze is an $n \times n$ environment that an agent can move on it (see Figure 1). The cells are numbered with respect to their row and column numbers. (2,3) corresponds to the cell in the 2^{nd} row and 3^{rd} column.

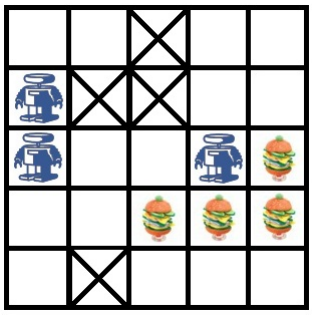

Figure 1: Maze Example

2. An agent is anything that can act in the environment (human, animal or robot). In this homework, you can assume it is a robot designed for a specific task.

3. There will be several (possibly more than one) agents, foods and obstacles in the maze.

4. There are four possible actions for an agent: Up, Down, Left and Right. If an agent is in the (x,y) cell, then Up action moves the agent to the $(x-1,y)$ cell; Down action moves the agent to the $(x+1,y)$ cell; Left action moves the agent to the $(x,y-1)$ cell; and Right action moves the agent to the $(x,y+1)$ cell. Actions are deterministic.

5. Our agents have limited observability. That is, each agent has an $m \times m$ observation scope that is visible to itself. Beyond their scopes, agents can not see the environment. Agents are always at the center of their scopes. m is always less than the size of the maze, i.e., n mentioned in the item #1. Figure 2 exemplifies the observation scope. The red box represents the observation scope for the agent positioned in the $(3, 4)$ cell. All agents will have the same observation scope.

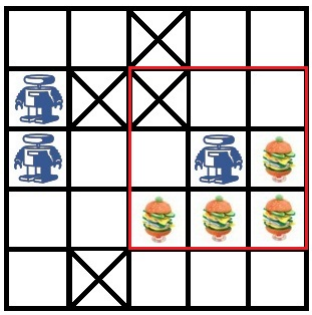

Figure 2: Observation Scope

6. Having observation scope as 1 means that, if an agent is in the (x,y) cell, its observation area is the box whose upper-left corner is ($x-1$,$y-1$) cell and whose bottom-right corner is ($x+1$,$y+1$) cell. You can assume that observation scope will be at least 1. Note that the observation scope of an agent changes when its position changes.

7. The aim for each agent is twofold.

   (a) If there is not any food in its observation scope, it will obey to its priority of actions. Priority of actions for each agent is a 4-tuple of actions. It determines which actions and in which order should be applied in case there is no food. For a tuple (Up, Left, Right, Down), for example, Up action has higher priory than Left action, and so on. An agent seeing nothing in its observation scope will do always Up for this example. If, for some reason such as having an obstacle in the front, agent can not apply its highest priority action, it will apply its second highest priority action, Left in this case. This will continue so on. Note that the agents may get into infinite loops in this configuration, which we will not check. Details are explained later in this section. Priority of actions will be given as input.

   (b) If there is some food in its observation scope, it will try to get close to the nearest food. To be able to do this, you should first find the nearest food, and then apply the action to get nearest to the nearest food.

   If there are more than one nearest food visible to the agent, agent will comply with its priority of foods. Priory of foods will be a 4-tuple of directions. For example, (Up, Right, Down, Left) specifies that, if there are more than one nearest food, the agent will try to get close to the upper-most one. If there are more than one food at the upper-most level, than it will get the rightmost one, an so on. Use manhattan distance as distance metric.

   If there are more than one action to get close to the nearest food, the agent again obeys to its priority of foods. Priority of foods will be given as input.

   An agents eats a food when it is positioned in the same cell with the food. When a food is eaten it will be eliminated from the environment.

8. Agents should not crash to the obstacles. Since minimum observation scope is 1, they will always see the obstacles next to them, and are expected to escape from them. For the case in the Figure 3, the agent is expected to go either left or right with respect to its priority of foods/actions

9. The outside of the maze is forbidden for the agent and is not considered in our problem. If an agent is in the cell $(1, 1)$, for example, it can not do Up or Left since those areas are forbidden. You can assume that edges of the maze have the same effect of the obstacles. So, agents should also not crash to the edges as for obstacles. That is why actually, in Figure 3, we do not consider the Down action.

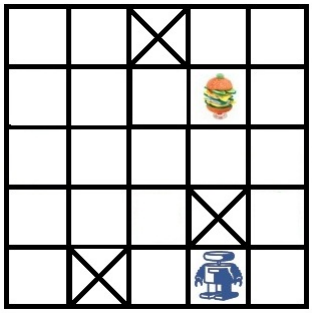

Figure 3: Escape from Obstacles

10. Agents are not informed by the decisions of the other agents. That means agents may attempt to the same cell during the execution, which will result in a crash. For example, two agents may possibly attempt to the same food cell. In that case, all of the crashing agents will be eliminated from the environment. The food will remain in the cell if available.

11. The name of the function you will implement is "search". Its arguments are specified as below.

    (a) The first argument is the environment. Environment is a list of list of Strings. The maze shown in Figure 1 is [["$-$", "$-$", "X", "$-$", "$-$"], ["A", "X", "X", "$-$", "$-$"], ["A", "$-$", "$-$", "A", "F"], ["$-$", "$-$", "F", "F", "F"], ["$-$", "X", "$-$", "$-$", "$-$"]], for example. Note that "A" stands for an agent, "X" stands for an obstacle, "F" stands for a food and "-" stands for an empty cell. Note also that the first list in the list of lists corresponds to the first row, and so on. The first integers in the rows corresponds to the first column.

    (b) The second argument is the observation scope. It is an integer value, determining the observation scope of the agents. Remember that observation scope is same for all of the agents.

    (c) The third argument is the number of steps that you will run your code. It is an integer value. You will be expected to run your code as the number of steps that is specified by this integer. See item #12 for details.

    (d) The forth arguments is the list of the agents in the environment. Each agent is a 10-Tuple $(c_x, c_y, a_1, a_2, a_3, a_4, f_1, f_2, f_3, f_4)$. The meaning of each component is as below.

        - c_x is the row number of the agent
        - c_y is the column number of the agent
        - a_1 is the highest priority action for no food
        - a_2 is the second highest priority action no food
        - a_3 is the third highest priority action for no food
        - a_4 is the fourth highest priority action no food
        - f_1 is the highest priority direction for food
        - f_2 is the second highest priority direction for food
        - f_3 is the third highest priority direction for food

- f_4 is the fourth highest priority direction for food

Each agent will be specified as explained 10-tuple in the input.

12. You are expected to print the latest state of the maze you are given. You will run your code k steps, where k is the third argument of the search function you will write. Note that initial state of the maze is the first argument of the search function. Running your code 1 step means to apply one action for each agent. Running your code 2 steps means to apply two subsequent actions for each agent. For example, the state of the maze in Figure 3 after 1-step is [[“−”, “−”, “X”, “−”, “−”], [“−”, “−”, “−”, “F”, “−”], [“−”, “−”, “−”, “−”, “−”], [“−”, “−”, “−”, “X”, “−”], [“−”, “X”, “A”, “−”, “−”]] (assume observation scope is 3 and priority of foods is (Left, Up, Down, Right)). The state of the maze in Figure 3 after 2-steps is [[“−”, “−”, “X”, “−”, “−”], [“−”, “−”, “−”, “F”, “−”], [“−”, “−”, “−”, “−”, “−”], [“−”, “−”, “A”, “X”, “−”], [“−”, “X”, “−”, “−”, “−”]]. Note that the output format is exactly same as the format of the first argument of the search function.

    Please be careful about the strings that you print and check the examples carefully in Section 3. Note that if any agent is eliminated from the environment for some reason such as crash, it will not be printed in the state of the maze anymore. Note also that any eaten foods will also be eliminated from the environment and will not be printed in the state of the maze anymore. If there remains no agent in the maze, you will continue to run your code by changing nothing in the maze.

3 Examples

- Main >> search [[“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “ − ”, “F”], [“ − ”, “X”, “ − ”, “A”, “ − ”], [“ − ”, “ − ”, “F”, “ − ”, “ − ”], [“ − ”, “A”, “ − ”, “ − ”, “ − ”]] 1 5 [(5, 2, “$Left$”, “Up”, “$Right$”, “$Down$”, “Up”, “$Right$”, “$Left$”, “$Down$”), (3, 4, “$Left$”, “$Down$”, “$Right$”, “Up”, “$Right$”, “$Down$”, “Up”, “$Left$”)]
  [[“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “ − ”, “ − ”], [“A”, “X”, “A”, “ − ”, “ − ”], [“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “ − ”, “ − ”, “ − ”]]

- Main >> search [[“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “ − ”, “F”], [“ − ”, “X”, “ − ”, “A”, “ − ”], [“ − ”, “ − ”, “F”, “ − ”, “ − ”], [“ − ”, “A”, “ − ”, “ − ”, “ − ”]] 1 100 [(5, 2, “$Left$”, “Up”, “$Right$”, “$Down$”, “Up”, “$Right$”, “$Left$”, “$Down$”), (3, 4, “$Left$”, “$Down$”, “$Right$”, “Up”, “$Left$”, “$Down$”, “Up”, “$Right$”)]
  [[“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “ − ”, “F”], [“ − ”, “X”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “F”, “ − ”, “ − ”], [“ − ”, “ − ”, “ − ”, “ − ”, “ − ”]]

  {- note that they are crashed, after 2 steps the maze remain same -}

- Main >> search [[“ − ”, “ − ”, “A”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “ − ”, “F”], [“ − ”, “X”, “ − ”, “A”, “ − ”], [“ − ”, “ − ”, “F”, “ − ”, “ − ”], [“ − ”, “A”, “ − ”, “ − ”, “ − ”]] 1 6 [(5, 2, “$Left$”, “Up”, “$Right$”, “$Down$”, “Up”, “$Right$”, “$Left$”, “$Down$”), (3, 4, “$Left$”, “$Down$”, “$Right$”, “Up”, “$Left$”, “$Down$”, “Up”, “$Right$”), (1, 3, “$Right$”, “$Left$”, “$Down$”, “Up”, “$Right$”, “Up”, “$Left$”, “$Down$”)]
  [[“ − ”, “ − ”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “X”, “A”, “ − ”], [“ − ”, “X”, “ − ”, “ − ”, “ − ”], [“ − ”, “ − ”, “F”, “ − ”, “ − ”], [“ − ”, “ − ”, “ − ”, “ − ”, “ − ”]]

  {- note that agent#3 continues after the crash -}

4 Submission

Submission will be done via COW. You should upload a single Haskell file called "$hw2 - e1234567.hs$". The **name of the file should not have any spaces**. Here e1234567 indicates your student ID you should fill it accordingly.

You should test your codes in inek machines with hugs before submitting. Since black box method will be used in evaluation, be careful about the name of functions, data structures etc.

Late submission: At most 3 late days are allowed. After 3 days, you get 0.

5 Grading

This homework will be graded out of 100.

6 Cheating Policy

We have zero tolerance policy for cheating. People involved in cheating will be punished according to the university regulations. Both parties involved in cheating get 0 from all of the 6 homework's and will be reported to the university's disciplinary actions committee. Note that this may also affect your condition to enter to the final examination.