Media Engineering and Technology Faculty
German University in Cairo

Agent Searching strategies

Project One

Name:	Hossam Amer
	Mohamed Gad
	Taher Galal
ID:	13-6070
	13-7516
	13-4426
Tutorial:	T-14
Submission Date:	26 November, 2011

1.1 Problem Definition

1.1.1 State Definition

In our program, every flip in orientation will be reflected on the state. The state is defined with the following:

1. Orientation: The orientation that the agent is facing this orientation will have either of the following values: N, E, S, W
2. Position: This has the value of the cell that the agent is in. This will be represented as a tuple of 2 numbers.
3. Holding Gold: This is a value for representing if the agent is holding gold or he still did not find the gold.
4. Killed Wumpus: This represents if the Wumpus was killed or not.

1.1.2 Node Definition

After that we also have a node. The node is defined by the following:

1. Action: This is the actions that was performed to create this action.
2. State: As the node consists of a certain tree. This will represent that each node will have a state.
3. Action Cost: This represents the cost of the actions performed till know.
4. Action Path: This represents the path that will be followed to reach this path.
5. Observation: This represents the observation that occurred in this node.
6. Heuristic: This represents the value returned by the heuristic function to the given node.
7. Depth: This represents the current depth of the given node.

1.1.3 Global Variables Definition

The global variables we used include the following:

1. Successor state: This is a list that contains the successor states of a certain node.
2. Queue: The queue that will contain the nodes that will be popped out of the queue to start search it properly. The type of this queue is determined according to the given searching strategy.
3. Partial nodes: These are nodes that were not fully obtained and may be missing some data. Therefore, it still misses the observation to specify that whether a node valid.
4. Strategies: This is a list that contains the search strategies which are the Breadth First Search, Depth First Search, Uniform Cost and A*.
5. Path to Goal: This is the list that contains the path to reach the goal.
6. Visited: It is a 4 dimensional array to specify if a certain node was visited before or not. It is done to avoid the repeated states

1.1.4 Possible Actions Definition

This subsection will describe the actions used to manipulate the node and the queue.

1. Forward: This is moving forward depending on the orientation and is handled in the method newPosition that will be explained bellow
2. Left: This action will change the orientation of the agent by the method newOrintation explained below.
3. Right: This action will do the same as the Left but the orientation is in the opposite direction.
4. Grab: This action is performed when there is gold in the cell, hence the agent performs a hold to the gold.
5. Attack: This action will throw and arrow and will be performed and added to partial nodes and in the next iteration will either be set to valid or invalid depending on if there was a wampus or not.

1.1.5 Methods Description

The following subsections will describe thoroughly the methods used inside our code.

Agent_Init

This method is called once when our python program is connected. It is used to initialize the variables with some star-up values.

Agent_Start

This method is called every episode. It initializes the values of the global variables specified above. The first step is creating the first partial node. Also the strategy index will be incremented at the start of a new episode. This increment is to assure that the search value that will be used is the next one in the list of strategies. Also the visited states array will be set with false to specify that no state was visited and to make sure that the states that were visited will not be visited again.

Agent_Step

This method is the method that will be called by r_glue repeatedly and will return certain actions.These actions will be performed by the agent and will retrieve certain observations. At the start of this method we will check whether our running mode is agent mode or eagle mode. The next step is if we are in eagle mode we will perform an update to the working set of nodes with the new observations and place them in the successor states list. The next step is to place the nodes into the queue to start queering on them in the right order depending on the search strategy. If it occurs that the queue is empty then this proves that the search strategy failed to find a solution, or the iterative deepening algorithm must be restarted to be performed on the next depth levels. Otherwise we will pop the first element from the queue, mark it as a visited state, and check if it is a goal state. It is specified a goal state if as specified in the project description that the agent is in cell 0,0 and is holding the gold. The process performed when in a goal node we will append the node to the path of nodes. Next we will return an action with a (.) and no more values to observe as well as change to agent mode. Otherwise if this is not a goal cell we will get the first node in the successor nodes and query to get the partial nodes. On the other hand if we are in agent mode we will follow the path that the eagle returned to allow us to reach to the gold and return back with it to the goal.

getCellsNeededForDiscovery

This method will take a certain node and it will get the node that we will need to discover next taking into consideration the position and the orientation of the agent. We will only query on 1 cell which is the cell that is in-front of the agent.

getSuccessorStates

This method will take the parent node and then will start getting the successor nodes. The first part will create a node that will have a grab actions specified in. This action will only happen if the observation that was returned specifies that there is gold in the cell that the agent is in. If this is the case it has the highest priority and we don't need to perform any other action therefore we will return this node only. Otherwise we will start performing all the actions that follow and was specified by in the project. These actions are the following, Forward this action will emulate that the agent is moving forward. This will cause a change and creating a new State that has a position changed while the rest of the elements are the same. On the Other hand if we have an action of either left or right this will move left or right. Hence, we will have a state with the same elements with only the orientation changed. We will also perform a fire of the arrow to create a state with this action. After performing this action if it was not the case after moving forward that there is a wumpus then this node will be taken as a dead end and will not be further explored. Each of these actions will add a cost which will increment the path cost. At the end the list of nodes will be returned as the successor states available.

newPosition

This method performs the returning of a new position after performing a forward action. This forward action will change the position related to the orientation. First of all we will get the orientation of the agent. Then according to the orientation a certain shift will be performed to retrieve a new position. If the position is not possible a false will be returned to specify that this forward action is invalid.

newOrintation

This method outputs the change of the orientation relative to either a left or a right through the use of a list to specify the direction faced and performing either and add to the index or a subtract with the remainder to simulate a circular list.

actionCost

This method calculates the cost of performing a certain action.

createPathToGoal

This method will append a C if we reach the goal to the path to goal in the goal node.

updateWorkingNodeSet

This method will update the partial nodes with the observations to specify the following, we will check if this node was visited before then the node will be removed from the partial nodes. Then a check will occur to see if there is a wumpus and the cell the agent is in and at the same time the gold is in the same cell and the wumpus was not killed this state is not possible because the wumpus needs to be killed first. If it was the case that the arrow was thrown and the wumpus was in this cell and it was also a pit this also specifies a invalid state. The last check is if the action was do nothing then it is fine. before returning the nodes we will remove all the invalid nodes from this set.

setVisited

This will set a visited array to set if this state was visited in a 4 dimensional array.

isVisted

This method will check if the state was visited or not.

add_node

This is a method that will take the node and the priority that the node needs to be inserted in the heap and will insert it in the correct order so it can be popped in the correct order.

enqueue

This method will enqueue the nodes in the queue at a certain place relative to the search method that will be performed.

goal

This method will check if the agent is in cell 0,0 and holding the gold. Hence, performing the goal test.

1.2 Heuristic functions Description

We use the Manhatten distance heuristic, since it is always useful to calculate the minimum distance to reach the goal and back. In addition, we made sure this heuristic is admissible. It is important to point out since, initially, the agent does not know the position of the goal, the heuristic value is set to one during acquiring the goal.

1.3 Search Strategies Implementation

A common note for all the search strategies, that we first query the environment about the state that the agent facing north, in position (0, 0), no gold acquired and we add it into the partial nodes.

1.3.1 Breadth First Search

The data structure used for this algorithm is a normal queue. Hence, the nodes come in a first in first out layout fashion. We set the nodes as visited once we enqueue them in the queue. This is done to avoid the repeating states and it speeds up the performance.

1.3.2 Depth First Search

We use here a last in first out queue to implement this algorithm. We do the same thing we did with the case of the breadth first search in terms of marking the visiting nodes.

1.3.3 Iterative Deepening Search

Here, we use a last in first out queue. If the node's depth equal to the iteration number, we do not expand this node. However, we check the current depth compared to the maximum depth, if it is still in range, we reset everything as the method agent_start does.

The maximum depth is set to WIDTH*HEIGHT*2*5. It is obvious that it is WIDTH*HEIGHT to cover the whole grid, and we double this factor by two to make it for the whole trip going back and forth. Finally, it is multiplied by 5 to cover the possible actions we have.

As done with the rest of the searching strategies, we marked the nodes visited to be able to see the result. That is because, if we used the normal implementation of this algorithm, it will run for a lot of time.

1.3.4 Uniform Cost Search

Priority queue is used here for the uniform cost search. This is to take in the account the path costs of every node during insertion. We marked here the visited too to boot up the performance.

1.3.5 A-star Search

The same thing as the uniform cost search done here, except that when we send the priority we take into account the heuristic value too.

1.4 Dealing with Repeated States

To accommodate for the repeated states we did the following: there is a four dimensional array where the 4 dimensions are position of the agent, orientation, holding gold or not and if the wumpus was killed or not. From the mentioned table, we are able to know whether the agent visited this stat before. In addition, there are two methods to set and check whether a given node is visited.

1.5 Comparing Search Strategies

The breadth first search is running 1 second, and the number of expanded nodes is in the order of 108 nodes. However, for the case of the depth first search is running around 1 second with around 500 expanded nodes. While, for the iterative deepening search, it expands more nodes (around 550), and the elapsed time for it is 2 seconds. In case of the uniform cost search, it took around 0.5 seconds to run, and it expanded 180 nodes. Finally, the A star search took 0.3 seconds, it expands 130 nodes.

From the numbers above, it is clear that the iterative deepening always expands more nodes. In addition, we concluded the optimality of the A-star and the uniform cost search. While, for the iterative deepening, it was not optimal nor the shortes path due to using the set visited property which prevents from expanding more nodes. It can be changed to ancestor visitor check. For all the algorithms, we used the set visited property, it is clear that it boosted up the performance a lot. The DFS usually gets a longer action path than the other searching strategies, since it expands the deeper nodes first.

1.6 References

The sources listed below helped us to implement our project:

1. Lectures and university materials.
2. http://docs.python.org/library/queue.html