Ailincă Laura
Apreutesei Rareș
Ardelean Raluca-Nicoleta

PACMAN – Documentation

Project 1

In this project, our Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. To success these objectives we built general search algorithms which applies to Pacman scenarios.

To make our algorithms complete, we implemented the graph search version of DFS, BFS, UCS and A*, which avoids expanding any already visited states. These algorithms are very similar.

Also, we used an auxiliary class for representing a node of graph which is defined as a combination of state, parent, action and cost. We consider that two nodes are equals if their state are the same.

Q1: Finding a Fixed Food Dot using Depth First Search

The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

DFS method has one argument: the problem - in this case the problem is SearchProblem

```
def depthFirstSearch(problem: SearchProblem}
```

For DFS we used a Queue as our data structure for representing the fringe and a List for retain expanded states.

The first steps are to add in fringe the node which has as state the Pacman's start state, parent and action does not exist because this node is the root of graph, and the cost is 0 and to initialize the expanded list to be empty.

If the fringe is empty the algorithm has failed, otherwise we choose a leaf node and remove it from the fringe and then we test if this node contains a goal state.

If it is a goal, then we return a list with the actions that bring Pacman to the goal. For creating this list we travers the resulting tree from DFS from this node to start node using the parent of each node. In the end we reversed the list because we built the path from goal to start state, but we need the path from start state to goal.

If it is not a goal, we add to the expanded list and then will expand the state and if the successors are not already expanded we will add them in fringe.

So, we expand states until the fringe is empty or until Pacman has reached the goal.

Q2: Breadth First Search

Breadth-first search algorithm starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes.

BFS method has one argument: the problem - in this case the problem is SearchProblem

```
def breadthFirstSearch(problem: SearchProblem)
```

This algorithm is very similar with DFS, the only differences are the data structure used for representing the fringe and the conditions for adding successors in fringe.

For BFS we used a Queue as our data structure for representing the fringe and a List for retain expanded states.

Also, we add the successors of expanded state in fringe only if they are not already expanded and if they are not in fringe.

Q3: Varying the Cost Function

Uniform-cost search is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination. Nodes are expanded, starting from the root, according to the minimum cumulative cost.

While BFS will find a fewest-actions path to the goal, we might want to find paths that are “best” in other senses.

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

UCS method has one argument: the problem - in this case the problem is SearchProblem

```
def uniformCostSearch(problem: SearchProblem)
```

This algorithm is very similar with DFS and BFS, the only differences are the data structure used for representing the fringe and the conditions for adding successors in fringe.

For UCS we used a PriorityQueue which will give the least costliest next state from all the successors states of expended states as our data structure for representing the fringe and a List for retain expanded states.

Unlike DFS and BFS, when we add a node in fringe, we must also add the cost of that node.

For each successor of an expanded state we update the frontier only if the successor is not already expanded.

We have 3 cases when update a state in fringe:

1. If state is already in fringe with higher priority we update its priority with a new priority which is equals with its cost + parent cost
2. If state is already in fringe with equal or lower priority, we do nothing
3. If state is not in fringe, we add it

Q4: A* search

A* method has 2 arguments:

1. the problem - in this case the problem is SearchProblem
2. a heuristic function -> heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function is a trivial example which return 0

```
def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic)
```

This algorithm is very similar with UCS, the only difference is when choosing a state from fringe to expand it, we must also take into account the heuristics, not only the cost.

The PriorityQueue which will give the least costliest(cost+heuristics) next state from all the successors states of expended states.

When update the fringe and the state is already in fringe with higher priority, we update its priority with a new priority which is equals with its cost + parent cost + heuristics.

A* finds the optimal solution slightly faster than uniform cost search

Q5: Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

To the CornersProblem class we defined a new attribute "visitedCorners" that is a list with 4 booleans which contains information about the 4 food dots from each corner. Each boolean represents if the dot was eaten (True) or not (False).

In init method we also cheked if the Pacman's starting position is one of the corners, if so, we mark that corner in "visitedCorners" list as True.

In this search problem we defined a state of Pacman as a tuple of Pacman's positions and the list of booleans for dots. So, the start state of the problem is the tuple formed by Pacman's starting position and the initial "visitedCorners" list.

The goal of this Corners Problem is that all 4 elements from the "visitedCorners" to be True. That means Pacman has eat all dots.

The successors of a state of Corners Problem are representing as a list of tuples formatted by the Pacman's next state, the action that bring Pacman to the next state(North, South, East, West) and the cost which is 1 because Pacman would make only a step.

If the Pacman's next position is a wall, that state wont be added to the successors list. Otherwise, it will be checked if the next positions is a corner and in this case the list of booleans of the state will be updated, the element that represents the dot from that corner will became True.

Q6: Corners Problem: Heuristic

Heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs.

The admissibility isn't enough to guarantee correctness in graph search – we need the stronger condition of consistency.

The heuristic method has 2 arguments:

1. state: The current search state (the data structure described above that we chose in our search problem)
2. the problem: in this case the problem is CornersProblem

```
def cornersHeuristic(state: Any, problem: CornersProblem)
```

Our heuristic is based on Manhattan distance from a state to each corner which is not visited. After the calculation of distances, we choose the maximum.

Before calculate the distances, we checked if the state is goal and in this case we retuned 0.

Heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs.

The admissibility isn't enough to guarantee correctness in graph search – we need the stronger condition of consistency.

The heuristic method has 2 arguments:

1. state: The current search state (the data structure described above that we chose in our search problem)
2. the problem: in this case the problem is CornersProblem

```
def cornersHeuristic(state: Any, problem: CornersProblem)
```

Our heuristic is based on Manhattan distance from a state to each corner which is not visited. After the calculation of distances, we choose the maximum.

Before calculate the distances, we checked if the state is goal and in this case we retuned 0.

Q7: Eating All The Dots

This problem aims to make the pacman eat all the food in as few steps as possible. This problem again, uses the A* algorithm but this time we need a new heuristic, which we need to implement.

We get the start position of the pacman in start_position and a list of the food coordonates in food_list. We initialise our heuristic with 0.

First thing first, if there is no food then we are done and we return 0. If not, we return the maximum distance between the current position of the pacman and the farthest food. We do this by going through the list of food coordonates and calculating the distance with mazeDistance, which gives us the real distance. Then if the distance is grater then the heuristic, the heuristic takes it's value. After all the distances were calculated we return the final heuristic value.

Q8: Suboptimal Search

In this problem we will implement a greedy like algorithm in which the pacman always goes to the closest food.

We gave the new problem a goal, which is that of always finding the closest food. We calculate the minimum distance from pacman's current location to the food pieces, using the manhattanDistance function. If pacman's current state is the same as the goal we calculated then the problem is solved and we return True.

Next step is to implement the findPathToClosestDot in which we just called A* on the problem.

Project 2

In this project we will have to take into consideration the presence of ghosts. We will have to implement minimax and expectimax searches and also the evaluation function.

Q1: Reflex Agent

We have to implement the evaluation function which tells our pacman which moves are the best for him to do. We have to take in consideration his current position, food positions and also positions of the ghosts.

We will be using the next state of the game (successorGameState), the possible positions of pacman (newPos), the new layout of food (newFood) and the next state of the ghosts (newGhostStates). We initialise the value with 0 and we call it score. We get the position of the closest ghost using the manhattanDistance between newPos and the new position of the ghosts and getting the minimum value. We then go through a list of all the food coordonates and calculate the distance between newPos and all the foods. If there are no food pieces we return 0, and if the next move results in a win we return infinity. Also if the action done by pacman is Stop we deduct 50 from the score. We then calculate a value using the current score, the distance to the closest ghost, the closest food and the score:

$$\frac{currentScore + distanceToClosestGhost}{10 * distanceToClosestFood + score}$$

Q2: MINIMAX Agent

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally.

The main function of this algorithm is the **getAction(self, gameState: GameState)** function from where we start with calling the **max_value (gameState, depth)** function with the next arguments:

1) gameState which specifies the full game state including the food, capsules, agent configurations and score changes
2) 0, because the first node (root) is at depth 0, considering the fact that we are on the first move in the game

Also, in the getAction function we return the minimax action

Moreover, we defined two other functions:

1) **Max_value function**
   - Here The maximizer tries to get the highest score possible maximizing any opportunity to win the game
   - Max value nodes represent the fact that Pacman must make the next move

2) **Min_value function**
   - Here minimizer tries to do the opposite and get the lowest score possible to find the gameplay that minimizes any tendency to win a game
   - Min value nodes represent the fact the adversaries (all the ghosts) must make the next move

The max_value function implementation:

- in a list (availableActions) we store the legal actions for the current agent (Pacman, id = 0)
- **next we test for terminal states** => if availableActions list is empty (we have nowhere else to go) or the game state is a winning state or in a losing state we return the utility (score for the game state we are in using the evaluationFunction function)
- if the game is not in a terminal state, we prepare for calculating the maximum value for the node from its children's nodes => initialize a variable **v** (with -infinity value) in which we will store the maximum value and an **action** variable for the best move
- as we traverse through the list of available actions, we store in a **tuple** the maximum value and the best action comparing the current state with the next state
- for the calculation of the **next state,** we recursively call the **min_value** function with the next parameters:

1) gameState = generateSuccessor which returns the successor state after the specified agent (in this case it has the ID 0 because Pacman is always agent 0) takes the action and the current action from the list
2) ghostID will be 1 because after Pacman moves, the first ghost has to move next
3) and the current depth

The min_value function implementation:

- in a list (availableActions) we store the legal actions for the current agent (the current ghost, id = ghostID from 1 to maximum number of ghosts)
- **next we test for terminal states** => if availableActions list is empty (we have nowhere else to go) we return the utility (score for the game state we are in using the evaluationFunction function)
- if the game is not in a terminal state, we prepare for calculating the minimum value for the node from its children's nodes => initialize a variable **v** (with +infinity value) in which we will store the minimum value and an **action** variable for the move
- as we traverse through the list of available actions, we store in a **tuple** the minimum value and the best action comparing the current state with the next state
- for the calculation of the **next state,** we recursively call the **max_value** function with the next parameters:
    1) gameState = generateSuccessor which returns the successor state after the specified agent (ID corresponding to the current ghost) takes the action and the current action from the list
    2) the depth will be increased with one if we reached the number of maximum ghosts , otherwise if we didn't reach the number of maximum ghosts we call the min function recursively function with the same **depth** and with the next ID (the next ghost to move)

Q3: Alpha-Beta Pruning

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.

Conceptually, alpha-beta pruning is this: if you're trying to determine the value of a node n by looking at its successors, stop looking as soon as you know that n's value can at best equal the optimal value of n's parent.

This algorithm involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning.

The two-parameter can be defined as:

a) Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
b) Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

This algorithm is very similar to the Minimax algorithm. The only difference is the using of an Alpha and a Beta variable that we initialize with maximum respectively the minimum value of float in the **get_action** function.

Step 1: At the first step the Max player will start first move from the root node where α= -∞ and β= +∞, these value of alpha and beta passed down to the next nodes (for the first action of the ghosts) where again α= -∞ and β= +∞, and those nodes pass the same value to their child (the second move for Pacman) and so on until we reach a parent for a leaf. We move down, depth first, until we reach the left most leaf node. Here, we have to explore both the leaf nodes to determine the maximum value for max's turn. The maximum value is found, which is the value that alpha is set at. Now, the node's value, is used to back track. When we reach node the parent node, the value of beta is updated to become the value of alpha as well.

When we reached this step, we will do max/min between the initial value of alpha and beta and the value of the leaf. To decide whether it's worth looking at its right node or not, we will check the condition **v > b** or **v < a**. If the condition is true, we return the current value and action because they are the best for that game state, and we don't have to explore the right branch anymore (we prune).

Otherwise, we compute the alpha value in the **max_value** function as being the maximum between its current value and the new computed value, and on the other hand, we compute the

beta value in the **min_value** function as being the minimum between its current value and the new computed value. Also, when we call recursively the max_value function and the min_value function, we'll send these two variables as well (*they propagate downwards in the tree*).

Q+: Hole Agent

For this question we implemented a new type of layout, one that includes the concept of "*holes*". **A hole** is represented by a cell of the layout that you cannot access unless the Pacman agent eats a so called *"jump capsule"* which allows him to *"jump"* over this cell and continue its normal search for the rest of the food using the Minimax Algorithm using another evaluation function.

This evaluation function is a better version of the *Q1 evaluation function* taking into consideration not only the Pacman, the food, and the ghost position, but also where the *jump capsules* are.

Firstly, we need to know the Pacman's current position, a list for the ghosts' state, where the food is found and also where the jump capsules are positioned.

First of all, if the current state is a winning state, we return the maximum value possible because it is the best state, and on the opposite if the current state is a losing state, we return the minimum value possible (- infinity).

The next step is to find the closest piece of food for the current position, by traversing the food list and use the **maze distance** between each piece of food and the current position of Pacman.

If we have a better evaluation function it means that we have to take into account, more parameters. So, we compute the next distances:

1) maze distance from the current position of Pacman to all the ghosts
2) maze distance from the current position of Pacman to all the jump capsules
3) minimum distance between Pacman and a ghost

Of course, every parameter has its own "gravity, importance" like chess the strategical advantages, they do not have all the same role-importance in the estimation -evaluation of a state, so we will assign them different weights in the final formula.

Finally, the formula for the score takes the current's game state score and takes into consideration the next factors:

1) taking a piece of food is more important than keeping a maximum distance to the ghost

2) keeping a minimum distance to a jump capsule is more important than minimizing the distance to a piece of food
3) the number of pieces of food remaining is also more important than the closest piece of food

So, the score formula looks like this:

$$score = currentScore - (20 * distToJump + 1.5 * minFoodDist + 2 * (\frac{1.0}{distanceToClosestGhost}) + 100 * nrOfRemainingFood)$$