# P1: Search

**Title**

**Kevin de la Coba and Juan Luis Sanz**

## Section 1

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

Our main struggle in the assignment was how to make the code generic. First, we created a class to store a search-tree, but we realised that a search-tree is not really needed in the algorithm, instead of that we can use the structure that the **problem** **class** give us. When we were getting the starting node, we realised that this one was given without any additional information (cost and action to get to it), our solution to this problem was to add this information in the node, but before going into any detail let’s see the structure we used:

[state, action, cost, [route to this state]]

We have a **list** in which we include the **current** **state**, the **action** to get there, the **accumulated** **cost** and the **route** from the starting state to the current node. Each node is stored in the closed and opened lists when it is specified by the algorithm.

We save the route to the current state, so when we are in the goal state, we must return the route to this node.

DFS (depth first search) uses a stack. The stack is the one who give us the correct node from the opened-list. We made a generic function that receives a problem and the **opened-list**, so in the DFS function we call this generic function sending as an argument the **stack** data structure.

* 1. **List & explanation of the framework functions used (1pt)**

The exercise was developed in:

* **Ubuntu 20.04**, this was the OS used during the development.
* Python **Anaconda** distribution, this distribution contained all the libraries used in the assignment.
* **VSCode**, this was the code editor, it’s useful because we can debug and run the code with just the editor and also, we can create terminals in it.
  1. **Includes code written by students (0.25 pts)**

**def** **depthFirstSearch**(problem):

**return** solveSimpleSearch(problem, util.Stack())

**def** **solveSimpleSearch**(problem, opened\_list):

start\_state = problem.getStartState()

# Initialize the opened-list with root-node

opened\_list.push([start\_state, None, **0**, []]) # [Starting state, Last action, Cost, Path]

# Initialize the closed-list as an empty list

closed\_list = []

# Iterating

**while** True:

# If the open list is empty error

**if** opened\_list.isEmpty():

**return** None

# Getting the node from the opened list

current\_node = opened\_list.pop()

# Checking if this is the goal

**if** problem.isGoalState(current\_node[**0**]):

**return** current\_node[-**1**]

# If the node is not in the closed list we add it and

# we expand it and we add it to the closed list

**if** **not** any(current\_node[**0**] == node[**0**] **for** node **in** closed\_list):

closed\_list.append(current\_node)

# Iterating through the successors and adding them to the open list

**for** child **in** problem.getSuccessors(current\_node[**0**]):

child\_node = list(child)

# Adding to the child the route to get there

# child path = current node path + action to get to the node

child\_node.append(current\_node[-**1**])

child\_node[-**1**] = current\_node[-**1**] + [child[**1**]]

# Adding to the child the accumulated cost

child\_node[**2**] += current\_node[**2**]

# Adding the node to the closed list

opened\_list.push(child\_node)

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

DFS **is not optimal**, in our case the **solution is reached** but the DFS algorithm does not guarantee a solution. In our implementation, **15 nodes** were expanded having a final **cost of 10** using the tinyMaze, in the mediumMaze **146 nodes** were expanded having a final **cost of 130**, and in the bigMaze **390 nodes** were expanded having a final **cost of 210**.

* 1. **Answer to question 1.1 (1pt)** ***Is the exploration order what you would have expected?***

Well, we were not expecting anything because we do not know the mazes by heart, but as we are not allowing cycles in the algorithm, we were expecting that the pacman could find the solution.

* 1. **Answer to question 1.2 (1pt) *Does Pacman actually go to all the explored squares on his way to the goal?***

No, pacman does not go to all the explored squares.

* 1. **Answer to question 2 (1pt) *Is this a least cost solution? If not, think about what depth-first search is doing wrong.***

No, it is not, and the reason is because DFS does not choose the nodes from the opened-list efficiently. DFS is always expanding the last nodes that have been introduced in the opened-list until the path built by choosing this nodes ends with a wall (that expands a node without childs). If the first node contains the most efficient route to a goal-state then DFS is efficient, otherwise it is not.

## Section 2

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

As we already built the generic function in the *section 1*, we just send to the generic function a **queue** as the opened-list, so BFS (breadth first search) uses a queue as a data structure.

* 1. **List & explanation of the framework functions used (1pt)**
  2. **Includes code written by students (0.25 pts)**

**def** **breadthFirstSearch**(problem):

**return** solveSimpleSearch(problem, util.Queue())

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

**Yes**, it is optimal and **yes**,it reaches the solution. For the tinyMaze it **expands 15 nodes** having a **final cost of 8**, for the mediumMaze it **expands 269 nodes** having a final **cost of 68**, and for the bigMaze it **expands 620 nodes** having a final **cost of 210**.

* 1. **Answer to question 3 (1pt) *Does BFS find a least cost solution? If not, check your implementation.***

**Yes**, it does.

## Section 3

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

For this section we were still using the generic function of the *section 1*, but instead of sending a stack or a queue, we sent a **priorityQueueWithFunction**, this data structure allowed us to have a queue that used a function to calculate the priority. In this problem the priority was just the cost of each node, but every time we were doing a push the data structure itself was computing the priority.

* 1. **List & explanation of the framework functions used (1pt)**
  2. **Includes code written by students (0.25 pts)**

**def** **uniformCostSearch**(problem):

# priorityFunction = lambda state: state[2]

**def** **priorityFunction**(state):

**return** state[**2**]

openlist = util.PriorityQueueWithFunction(priorityFunction)

**return** solveSimpleSearch(problem, openlist)

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

UCS (uniform cost search) it’s only optimal if the accumulated cost of the successor is bigger or equal than the path of the current node. In pacman that condition is fulfilled, so we can say that **yes**, **it is optimal**. As we don’t allow cycles by eliminating repeated states, pacman **will reach a solution always**. The cost using the tinyMaze was **15** with a **final cost of 8**, using the mediumMaze was **269** with a **final cost 68,** and using the bigMaze was **620** with a final cost 210.

## Section 4

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

As we did in the previous sections, we used the generic function, and as we did in the *section 3*, we used as a data structure the **priorityQueueWithFunction**, but this time the function was different. A\* uses a heuristic, the value of the heuristic in node of the function must be added to the accumulated cost in each node, so our priority function was computing that value.

* 1. **List & explanation of the framework functions used (1pt)**
  2. **Includes code written by students (0.25 pts)**

**def** **aStarSearch**(problem, heuristic=nullHeuristic):

# priorityFunction = lambda state: state[2] + heuristic(state[0], problem)

**def** **priorityFunction**(state):

**return** state[**2**] + heuristic(state[**0**], problem)

openlist = util.PriorityQueueWithFunction(priorityFunction)

**return** solveSimpleSearch(problem, openlist)

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

If the heuristic is admissible then A\* it’s optimal. The null heuristic is admissible but trivial because it returns 0, so we could say that in that case A\* is optimal, but the Manhattan distance heuristic it is not optimal because in some states being n a state in the problem and g(n) the accumulated cost.

The expanded nodes for the with the Manhattan distance heuristic in the tinyMaze were **14** with a **final cost of 8**,in the MediumMaze **221** with a **final cost of 68**, in the bigMaze **549** with a **final cost of 210**.

* 1. **Answer to question 4 (1pt) *What happens on openMaze for the various search strategies?***

A\*, UCS and BFS find the shortest solution, A\* expands 535 nodes having a final cost of 54, UCS and BFS expand 682 nodes having a final cost of 54, and DFS is the worst one, because it has the longest solution with a cost of 212 having 576 nodes expanded. DFS is always exploring the first successor so this makes that the path found it is not optimal.

## Section 5

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**

In this section we had several issues and problems in order to solve it:

1. Pacman was not going back after eating one food. When we were testing our first implementation, we realised that the path that the algorithm was finding was ending in the first food it found. This was because our “state” (when we refer to state, we mean the position of the pacman) included the position of the pacman, so once the closed-list had this position, it cannot be expanded again.
2. Pacman did not find the most efficient route. In order to solve the previous problem, we **deleted** the data in the closed list after one food was found. We realised that this was not a good practice because the algorithm was modified and because the route was not efficient.
3. We were expanding too many nodes. Before we continued, we were thinking about a way to solve the first 2 problems, we decided that each state would be unique by assigning it a number, later in the code we were getting the position of each index by functions. The problem is that each position was visited plenty of times.

So, how did we fix this? We realised that a position may have been visited more than once, but each time the position is visited something must have been changed. Then we also realised that we were focusing on modifying the node (when we refer to node, we refer to the complete information about a state which includes, position, last action, and cost) and we didn’t need to modify this! We needed to modify the state so every time a node is revisited it is because a food has been found before. So, we modified the state, so it contained: the position and a list that included the corners that have not been visited.

* 1. **List & explanation of the framework functions used (1pt)**
  2. **Includes code written by students (0.25 pts)**

**def** **\_\_init\_\_**(self, startingGameState):

...

# We define a starting state which includes the starting positions

self.starting\_state = (self.startingPosition, starting\_position\_corners)

**def** **getStartState**(self):

**return** self.starting\_state

**def** **isGoalState**(self, state):

# Checking the lenght of the corners in the state

# If list containing the corners is empty, return true

**return** len(state[**1**]) == **0**

**def** **getSuccessors**(self, state):

successors = []

# Getting the position in the state

(x, y) = state[**0**]

**for** action **in** [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

dx, dy = Actions.directionToVector(action)

nextx, nexty = int(x + dx), int(y + dy)

**if** **not** self.walls[nextx][nexty]:

# Building the structure of the child state

child\_state = ((nextx, nexty), state[**1**].copy())

# If the postion is in a corner, we remove that

# position from the not visited corners

**if** child\_state[**0**] **in** child\_state[**1**]:

child\_state[**1**].remove(child\_state[**0**])

# Building the structure of the child node

child\_node = (child\_state, action, **1**)

successors.append(child\_node)

self.\_expanded += **1** # DO NOT CHANGE

**return** successors

* 1. **Screenshots of executions and test carried out analyzing the results (1pt)**
  2. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**

**Yes**, it is optimal and **yes**, it reaches the solution.

In the tinyCorners **252** **nodes** were expanded with a **final cost of 28**, in the mediumCorners **1966** **nodes** were expanded with a **final cost of 106**, and in the bigCorners **7949** **nodes** were expanded with a **final cost of 162**.

## Section 6

* 1. **Personal comment on the approach and decisions of the proposed solution (1pt)**
  2. **List & explanation of the framework functions used (1pt)**
  3. **Includes code written by students (0.25 pts)**
  4. **Screenshots of executions and test carried out analyzing the results (1pt)**
  5. **Conclusions on the behavior of pacman, it is optimal (y / n), reaches the solution (y / n), nodes that it expands, etc (1pt)**
  6. **Answer to question 5: heuristics (1pt)**

## Section 7

Personal comments on the development of this practice

## Memory grade (40% of practice)

**Total points (X / 31.5)**