# 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 decision in this part was the structure used to represent the **search-tree**, **opened-list** and **closed-list**. We concluded that the closed list is already in the search tree, so, why would we create an object for it? For the opened-list we used the already implemented data structures in the file *util.py*, in this part of the assignment we used the **stack**.

At the early development we had problems with making the code generic, we realised that the assignment is not only about pacman, we saw that the autograder was using graphs where the states were strings, or tuples etc. Once we designed the class **searchTree**, making the code generic became easier, we tried to create methods with descriptive names so it was easier for us to know what we were doing (methods such as *isInClosedList* etc.).

So, the algorithm uses the class searchTree, this class contains just a list of dictionaries in which we have a variable **childs** which is a list that contains the childs of a state, a variable **parent** which is a state that is the parent of a state, a variable route which is a list of “movements” to get to that state, and variable **node**, it’s usually a tuple with all the information provided by the *problem* argument, if it’s not a tuple its just an state and this means that it’s the parent node. Regarding the methods of the class as we said, these are methods to obtain information of the tree. Apart from this the algorithm uses an opened-list and from here its just the generic algorithm set in the theory slides.

The trickiest part was how to not allow repetition of the states in the search tree, we realised this was important because when we allowed that, the code was going through already expanded states and, in some cases, going into infinite loops.

* 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 distribution **Anaconda**, 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(), None)

**def** **solveSimpleSearch**(problem, utils, heuristic):

start\_state = problem.getStartState()

# Initialize the search-tree with the root-node

search\_tree = SearchTree(start\_state)

# Initialize the opened-list with root-node

openedList = utils

**if** type(openedList) == util.Stack **or** type(openedList) == util.Queue:

openedList.push(start\_state)

**elif** heuristic != None:

openedList.push(start\_state, heuristic(start\_state, problem))

**else**:

openedList.push(start\_state, **0**)

# Iterating

**while** **1**:

# If the open list is empty error

**if** openedList.isEmpty():

**return** None

# Getting the node from the stack to expand it and adding it to the search tree

currentNode = openedList.pop()

**if** search\_tree.isInClosedList(currentNode) **and** **not** search\_tree.isRoot(currentNode):

**continue**

# Checking if the node is in the closed-list, if it is we dont add it

**if** **not** search\_tree.isInClosedList(currentNode):

search\_tree.add(currentNode)

search\_tree.setParent(currentNode)

search\_tree.setRoute(currentNode)

search\_tree.setCost(currentNode)

# Checking if this is the goal

**if** problem.isGoalState(currentNode **if** search\_tree.isRoot(currentNode) **else** currentNode[**0**]):

**return** search\_tree.getNodeRoute(currentNode)

**else**:

# Iterating through the successors and adding them. Excluding the nodes that have been visited

**for** child **in** problem.getSuccessors(currentNode **if** search\_tree.isRoot(currentNode) **else** currentNode[**0**]):

# Checking if the childs are already in the tree

**if** **not** search\_tree.isInClosedList(child):

search\_tree.addChild(currentNode, child)

**if** type(openedList) == util.Stack **or** type(openedList) == util.Queue:

openedList.push(child)

**elif** heuristic != None:

openedList.push(child, search\_tree.getNodeCost(currentNode)+child[**2**]+heuristic(child[**0**], problem))

**else**:

openedList.push(child, search\_tree.getNodeCost(currentNode)+child[**2**])

We can see that the **depthFirstSearch** function, calls **solveSimpleSearch** which is the generic method. There are some highlighted zones, these zones represent the “if decision” done by the code. We can also see that in the **depthFirstSearch** function we call **solveSimpleSearch** with the arguments **problem**, **util.stack** (which is the open list to use) and None (because we don’t use an heuristic).

\*We strongly recommend to review the code in the search.py file

* 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)**
  3. **Answer to question 1.1 (1pt)**
  4. **Answer to question 1.2 (1pt)**
  5. **Answer to question 2 (1pt)**

## Section 2

* 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 3 (1pt)**

## Section 3

* 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)**

## Section 4

* 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 4 (1pt)**

## Section 5

* 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)**

## 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)**