Intel·ligència Artificial
Pràctica 1

por
Sebastian Jitaru i Gerard Llubes

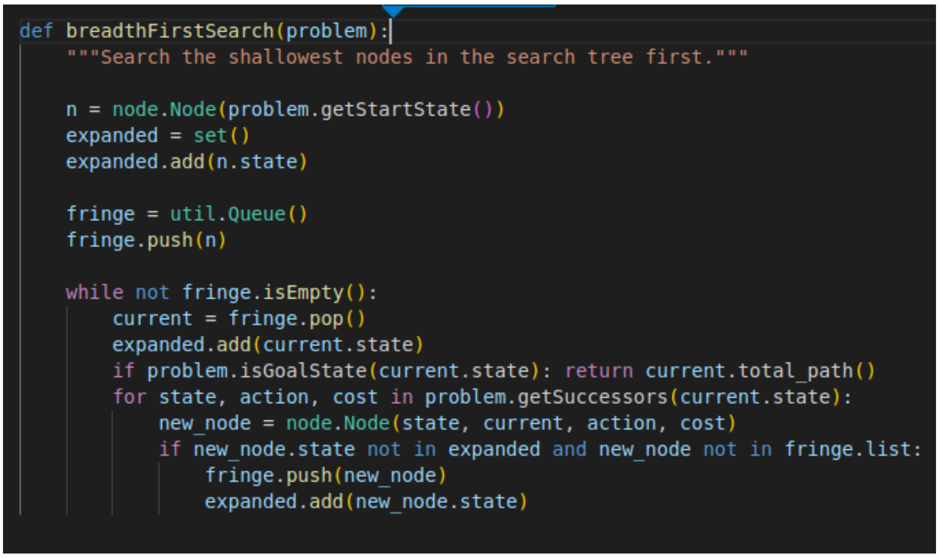
Grau en Enginieria Informàtica
Intel·ligència Artificial
Universitat de Lleida
November 22, 2022

Contents

1 Algorithms descriptions and decisions on implementation

1.1 BFS

Breadth First Search is a search algorithm characterised by level search, in other words, by exploring a whole level before going to the next one.
To do so in our implementation we decided to use a FIFO queue as a fringe. In order to make it easy for us to use the FIFO queue we used the "Util.Queue" python library so that we can pop and push elements in and out of the fringe. in Figure1 you can see our implementation of the algorithm.



```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""

    n = node.Node(problem.getStartState())
    expanded = set()
    expanded.add(n.state)

    fringe = util.Queue()
    fringe.push(n)

    while not fringe.isEmpty():
        current = fringe.pop()
        expanded.add(current.state)
        if problem.isGoalState(current.state): return current.total_path()
        for state, action, cost in problem.getSuccessors(current.state):
            new_node = node.Node(state, current, action, cost)
            if new_node.state not in expanded and new_node not in fringe.list:
                fringe.push(new_node)
                expanded.add(new_node.state)
```

Figure 1: BFS implementation code snippet

Our implementation (Figure1) is a graph implementation, meaning that we keep track of the nodes we've explored (using the expanded set).
This implementations for each node puts all of their successors in the fringe and process them in FIFO order, this way we make sure the exploration goes level by level, because when the successors of the next node are pushed into the fringe, these will go to the bottom of the queue.

1.2 DFS

The implementation of the Depth First Search algorithm is practically identical to the BFS1 one, changing only the data structure used for the fringe for a stack or a LIFO queue. To do so we used the "Util.Stack" library provided by python
In Figure2 you can see our implementation of the DFS algorithm.

```
fringe = util.Stack()
n = node.Node(problem.getStartState())

if problem.isGoalState(n.state): return n.total_path()
fringe.push(n)

expanded = set()

while not fringe.isEmpty():
    n = fringe.pop()
    expanded.add(n.state)
    for state, action, cost in problem.getSuccessors(n.state):
        new_node = node.Node(state, n, action, cost)
        if new_node.state not in expanded and new_node not in fringe.list:
            if problem.isGoalState(state): return new_node.total_path()
            fringe.push(new_node)
```

Figure 2: DFS implementation code snippet

By changing the data structure of the fringe, now when the node successors enter the fringe, these go to the top of the queue instead of the bottom. By doing so the algorithm will explore one branch until the end (if there is an end) before exploring another.

1.3 UCS

For the implementation of the Unified Cost Algorithm we used a Priority Queue using the "util.PriorityQueue" library provided by python.
In our implementation we get the successors of a given node, if we haven't explored it yet we update the fringe with the given successor. Then when we pop the next node from the fringe, this will be the node with the least cost. By doing that when we eventually find a solution, we can guarantee that it is the best one.

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"

    fringe = util.PriorityQueue()
    n = node.Node(problem.getStartState())
    fringe.push(n, 0)
    expanded = dict()

    while not fringe.isEmpty():
        current = fringe.pop()
        if problem.isGoalState(current.state): return current.total_path()
        expanded[current.state] = current
        for state, action, cost in problem.getSuccessors(current.state):
            new_node = node.Node(state, current, action, cost+current.cost)
            if state not in expanded:
                fringe.update(new_node, current.cost)
```

Figure 3: UCS implementation code snippet

1.4 A*

The implementation of the A*4 algorithm is practically the same as as the UCS3 but adding heuristics when calculating the cost. The heuristic cost is added to the actual cost of a node. Doing that makes a path that the heuristic predicts to have a high cost not to be popped immediately despite having low actual cost, and could make pop from the fringe the node with the higher actual cost if the Heuristic thought that it was the best path. In other words, A* is like the UCS 3 algorithm but Heuristics help to narrow down which path to follow.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    fringe = util.PriorityQueue()
    n = node.Node(problem.getStartState())
    fringe.push(n, heuristic(n.state,problem))
    expanded = dict()

    while not fringe.isEmpty():
        current = fringe.pop()
        if problem.isGoalState(current.state): return current.total_path()
        expanded[current.state] = current
        for state, action, cost in problem.getSuccessors(current.state):
            new_node = node.Node(state, current, action, cost+current.cost)
            if state not in expanded:
                fringe.update(new_node, current.cost+heuristic(new_node.state,problem))
```

Figure 4: A* implementation code snippet

2 Decision taken on Heuristics

2.1 Food Heuristic

This heuristic gets the Manhattan distance to all the food and stores them in an array, the lower the distance, the better the cost that the heuristic will predict.

```
def get_manhattan_distance(p, q):
    """
    Return the manhattan distance between points p and q
    assuming both to have the same number of dimensions
    """
    # sum of absolute difference between coordinates
    distance = 0
    for a,b in zip(p,q):
        distance += abs(a - b)

    return distance

position, foodGrid = state
unvisited_foods = foodGrid.asList()

if not unvisited_foods:
    return 0

closest_food = min([get_manhattan_distance(position, food) for food in unvisited_foods])

return closest_food
```

Figure 5: Food Heuristic

2.2 Corners Heuristic

This heuristic gets the Manhattan distance to all 4 corners and stores them in an array, the lower the distance, the better the cost that the heuristic will predict.

```
corners = problem.corners # These are the corner coordinates
walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

"*** YOUR CODE HERE ***"
coordinates = state[0]
visited_corners = state[1]
unvisited_corners = []

for one_corner in corners:
    if not one_corner in visited_corners:
        unvisited_corners.append(one_corner)

heuristic_number = 0

while len(unvisited_corners) != 0: # While not empty
    manhattan_distances = []
    # Get manhattan distance to every corner
    for each_corner in unvisited_corners:
        get_manhattan = util.manhattanDistance(coordinates, each_corner)
        manhattan_corner = (get_manhattan, each_corner)
        manhattan_distances.append(manhattan_corner)
    minimum, the_corner = min(manhattan_distances)
    coordinates = the_corner
    heuristic_number += minimum
    unvisited_corners.remove(the_corner)

return heuristic_number
```

Figure 6: A* implementation code snippet

We know that both these heuristics will be consistent because the path will always be equal or greater than the Manhattan distance, this is because Manhattan distance traces a straight line between Pac-Man and the food, but in real life Pac-man has to follow the grid and sort obstacles in order to get to the destination.

3 Performance Analysis

3.1 Search Algorithms

	DFS	**A*(Euclidean)**	**BFS**	**A* (Manhattan)**	**UCS**
bigMaze	0.1	0.0	0.1	0.0	0.1
contoursMaze	0.0	0.0	0.0	0.0	0.0
mediumMaze	0.0	0.0	0.0	0.0	0.0
openMaze	0.1	0.1	0.1	0.0	0.1
smallMaze	0.0	0.0	0.0	0.0	0.0
testMaze	0.0	0.0	0.0	0.0	0.0
tinyMaze	0.0	0.0	0.0	0.0	0.0

Table 1: Time taken for each algorithm

	DFS	**A*(Euclidean)**	**BFS**	**A* (Manhattan)**	**UCS**
bigMaze	210	210	210	210	210
contoursMaze	13	13	13	13	49
mediumMaze	68	68	68	68	130
openMaze	54	54	54	54	158
smallMaze	19	19	19	19	49
testMaze	7	7	7	7	7
tinyMaze	8	8	8	8	10

Table 2: Path cost for each algorithm

	DFS	A*(Euclidean)	BFS	A* (Manhattan)	UCS
bigMaze	620	557	620	549	390
contoursMaze	170	60	170	49	49
mediumMaze	269	226	269	221	144
openMaze	682	550	682	535	315
smallMaze	92	56	92	53	59
testMaze	7	7	7	7	7
tinyMaze	15	13	15	14	14

Table 3: Expanded nodes for each algorithm

	DFS	A*(Euclidean)	BFS	A* (Manhattan)	UCS
bigMaze	300	300	300	300	300
contoursMaze	497	497	497	497	461
mediumMaze	442	442	442	442	380
openMaze	456	456	456	456	352
smallMaze	491	491	491	491	461
testMaze	503	503	503	503	503
tinyMaze	502	502	502	502	500

Table 4: Score for each algorithm

3.2 Corners

	UCS corners	**A* corners)**
bigMaze	0.4	0.3
contoursMaze	0.1	0.0
mediumMaze	0.0	0.0

Table 5: Time for each algorithm

	UCS corners	**A* corners)**
bigMaze	162	162
contoursMaze	106	106
mediumMaze	28	28

Table 6: Path cost for each algorithm

	UCS corners	**A* corners)**
bigMaze	7949	6490
contoursMaze	1966	1653
mediumMaze	252	239

Table 7: Expanded nodes for each algorithm

	UCS corners	**A* corners)**
bigMaze	378	378
contoursMaze	434	434
mediumMaze	512	512

Table 8: Score for each algorithm

3.3 Food

	DFS	**A*(Euclidean)**
bigSearch	NAN	NAN
greedySearch	0.7	0.4
mediumSearch	NAN	NAN
oddSearch	NAN	NAN
openSearch	NAN	NAN
smallSearch	NAN	NAN
testSearch	0.0	0.0
tinySearch	NAN	NAN
trickySearch	NAN	NAN
mediumDottedMaze	NAN	NAN
bigCorners	NAN	NAN
mediumCorners	3.5	NAN
tinyCorners	0.1	0.1

Table 9: Time for each algorithm

	DFS	**A*(Euclidean)**
bigSearch	NAN	NAN
greedySearch	16.0	16.0
mediumSearch	NAN	NAN
oddSearch	NAN	NAN
openSearch	NAN	NAN
smallSearch	NAN	NAN
testSearch	7.0	7.0
tinySearch	NAN	NAN
trickySearch	NAN	NAN
mediumDottedMaze	NAN	NAN
bigCorners	NAN	NAN
mediumCorners	106.0	NAN
tinyCorners	28.0	28.0

Table 10: Path cost for each algorithm

	DFS	**A*(Euclidean)**
bigSearch	NAN	NAN
greedySearch	538.0	692.0
mediumSearch	NAN	NAN
oddSearch	NAN	NAN
openSearch	NAN	NAN
smallSearch	NAN	NAN
testSearch	12.0	14.0
tinySearch	NAN	NAN
trickySearch	NAN	NAN
mediumDottedMaze	NAN	NAN
bigCorners	NAN	NAN
mediumCorners	1475.0	NAN
tinyCorners	231.0	252.0

Table 11: Expanded nodes for each algorithm

	DFS	**A*(Euclidean)**
bigSearch	NAN	NAN
greedySearch	614.0	614.0
mediumSearch	NAN	NAN
oddSearch	NAN	NAN
openSearch	NAN	NAN
smallSearch	NAN	NAN
testSearch	513.0	513.0
tinySearch	NAN	NAN
trickySearch	NAN	NAN
mediumDottedMaze	NAN	NAN
bigCorners	NAN	NAN
mediumCorners	434.0	NAN
tinyCorners	512.0	512.0

Table 12: Score for each algorithm