Pacman Game

Step 2 and 3 - Breadth-First and Uniform-Cost Search

Intelligent Systems

Master in Software Engineering

Manuel Cura (76546) | Carolina Albuquerque (80038)

deti universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Context

The aim of the first challenge of the Intelligent Systems course is the implementation of breadth-first search and uniform-cost algorithms to allow the pacman agent to find paths across the game board to achieve the goals.

In this work, the strategies used in the implementation of the algorithm will be addressed, as well as the results obtained for the different boards of the pacman game. At the end, the costs of the obtained paths will be analyzed and the results discussed comparing the already three search algorithms implemented.

Breadth-First Search

Breadth-First Search (BFS) is an algorithm for searching tree or graph structures where it starts at a root node and explores level by level, contrary to depth-first search (DFS), visiting all successor nodes on the level before moving to the next level.

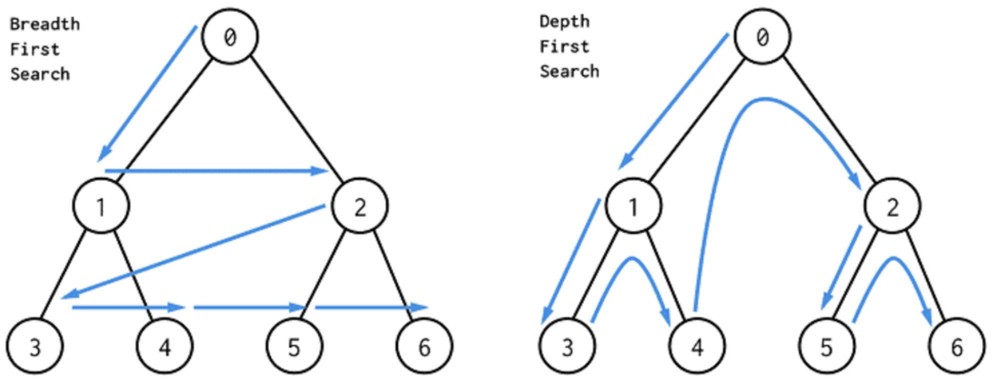


Figure 1 - *Comparison between breadth-first and depth-first algorithms*

Uniform-Cost Search

Uniform-Cost Search (UCS) is an algorithm for searching weighted tree or graph structures.

This algorithm follows the nodes with the smallest cost, always keeping track and updating the least path cost to any of the already expanded nodes. Once the solution is found, the cheapest path is the one that the algorithm selects.

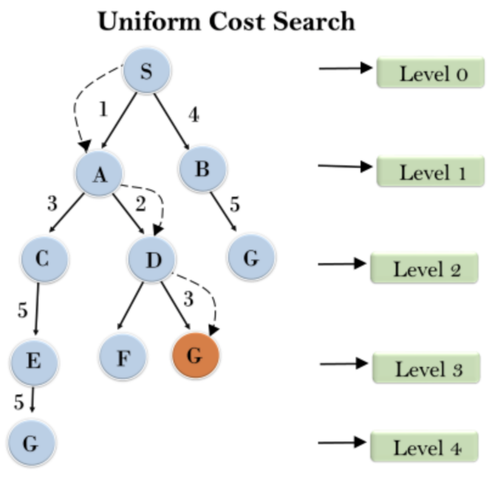


Figure 2 - *Uniform-Cost search algorithm*

Implementation

Similar to the depth-first algorithm, both breadth-first and uniform-cost search algorithms were implemented using the graph search approach in order to avoid expanding already explored nodes, optimizing the agent. All explored nodes are stored in a set and for each successor node of the current node it is checked if this was already explored. With this approach, contrary to the tree search approach, only non explored nodes will be considered and pushed to data structure avoiding the rewriting of actions for an already explored node. It expected a list of actions as a result of algorithm search. During the search, if the goal is achieved, the corresponding list of actions to achieve it must be returned. Otherwise, an empty list of actions is returned to the agent.

Breadth-First Search Algorithm

Data Structures

As breadth-first search algorithm only differs from depth-first search in the order in which the successor nodes are analysed, consequently, this algorithm only differs in the type of data structure. Breadth-First search acts in the base of the first successor node visited is the first to be explored, thus it uses a queue data structure due to its first-in first-out (FIFO) approach.

Code Reuse

Due to the implementation similarities between the depth-first search algorithm and the breadth-first search, it was implemented as a generic function reused for both algorithms.

This function receives different structures by parameters, a stack in the depth-first search case and a queue for the breadth-first search. In that sense, as it followed the pseudo-code provided in the classes' documentation, the code for the breadth-first is a little bit different than the algorithm delivered on the first step, however the same results were obtained. The main difference between this step and the previous one is in the verification of the goal state for each successor node visited. With this approach, if a successor node is considered the goal is returned the list of actions to achieve the solution without expanding the nodes.

```
def genericGraphSearch(problem, structure):
    startNode = problem.getStartState()
    explored = set() # set of explored nodes
    # (Node, list of actions until to achieve the node)
    structure.push((startNode, list()))

    if problem.isGoalState(startNode): # returns actions if is the goal
        return list()

    while not structure.isEmpty():
        currentNode, actions = structure.pop() # returns the element of structure

        if currentNode not in explored:
            explored.add(currentNode)

            for successorNode, action, cost in problem.getSuccessors(currentNode):
                if successorNode not in explored:
                    # copy actions list and append the new action to achieve the goal
                    nextAction = actions.copy()
                    nextAction.append(action)

                     # returns actions if is the goal
                    if problem.isGoalState(successorNode):
                        return nextAction

                    # add the list of actions to achieve the current node
                    structure.push((successorNode, nextAction))
    return list()
```

```
def depthFirstSearch(problem):
    structure = util.Stack() # stack because use LIFO implementation
    return genericGraphSearch(problem, structure)
```

```
def breadthFirstSearch(problem):
    structure = util.Queue() # queue because use FIFO implementation
    return genericGraphSearch(problem, structure)
```

Uniform-Cost Search Algorithm

Data Structure

In order to implement the uniform-cost search algorithm, a priority queue was used, where the priority is defined by the cost of the path to achieve a certain state.

This structure always returns the priority element and when it is updated and if the structure does not contain the node, this node will be added with the respective value for the cost of the path. In situations where the node is already in the priority queue, the value assigned will be updated but only if the new cost of the path for that node is smaller than the previous one.

To achieve this behaviour, the update function of the priority queue already defined in *util.py* was used, avoiding the implementation of two if conditions, since this function already does the desired behaviour, even if the node already exists on the structure or not.

Code

```python
def uniformCostSearch(problem):
    startNode = problem.getStartState()
    explored = set() # set of explored nodes
    structure = util.PriorityQueue() # queue because use FIFO implementation

    # ((Node, list of actions until to achieve the node), cost)
    structure.push((startNode, list()), 0)

    if problem.isGoalState(startNode): # returns actions if start node is the goal
        return list()

    while not structure.isEmpty():
        currentNode, actions = structure.pop() # returns the priority element

        if currentNode not in explored:
            explored.add(currentNode)

            for successorNode, action, cost in problem.getSuccessors(currentNode):
                nextAction = actions.copy()
                nextAction.append(action)

                if problem.isGoalState(successorNode): #returns actions if is the goal
                    return nextAction

                # get the cost of the actions' path
                pathCost = problem.getCostOfActions(nextAction)
                # add the list of actions to achieve the current node
                structure.update((successorNode, nextAction), pathCost)

    return list()
```

The entire code can be checked on GitHub repository.

Results Analysis

Testing the algorithms implemented for several boards, the results obtained are considered positives and the agent performs the algorithm correctly according to the visual output shown in the game window.

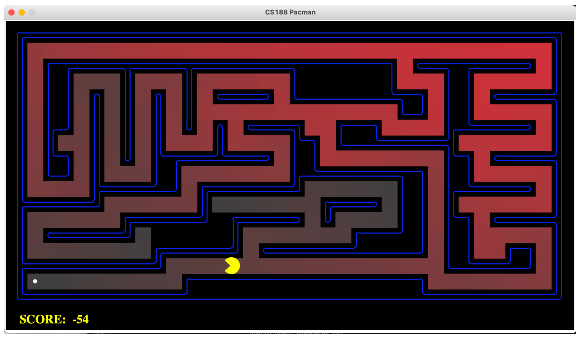


***Figure 3** - Expanded nodes of breadth-first search algorithm for mediumMaze*

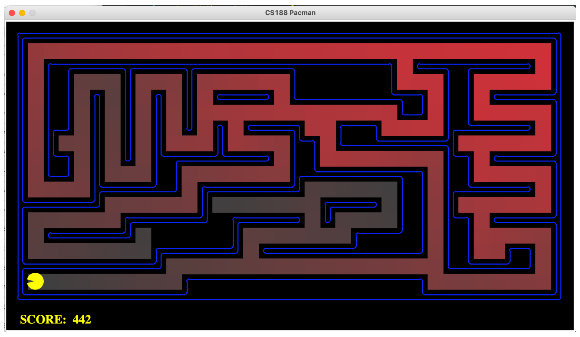
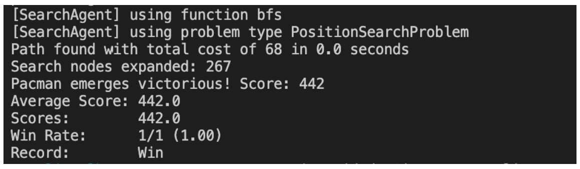
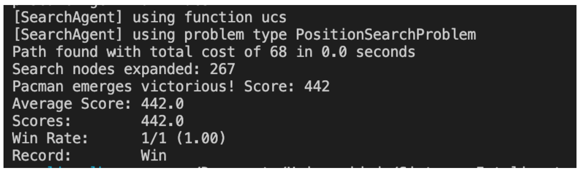


***Figure 4** - Expanded nodes of uniform-cost search algorithm for mediumMaze*



```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 267
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

***Figure 5** - Results of breadth-first search algorithm for mediumMaze*



```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 267
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

***Figure 6** - Results of uniform-cost search algorithm for mediumMaze*

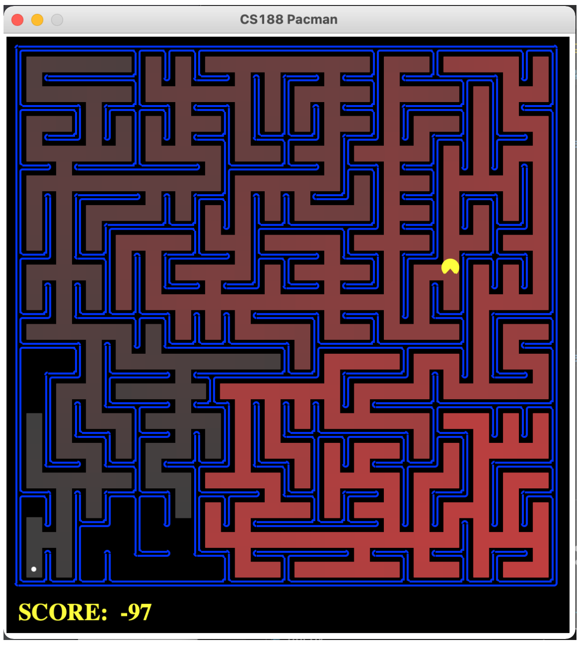


Figure 7 - *Expanded nodes of breadth-first search algorithm for bigMaze*

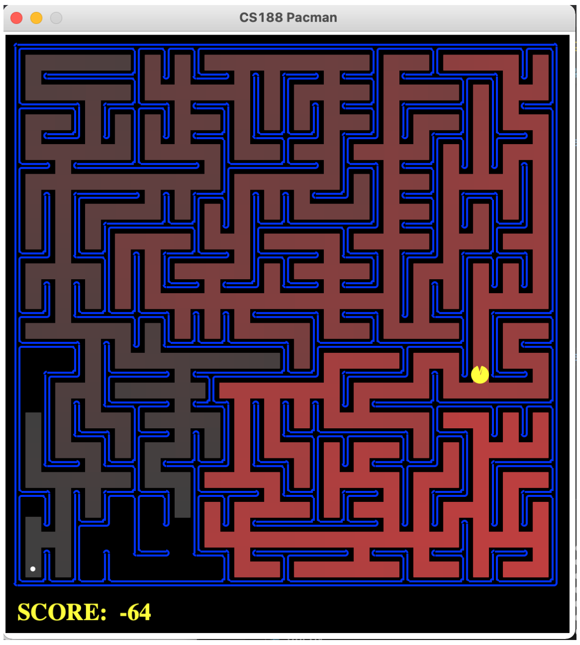
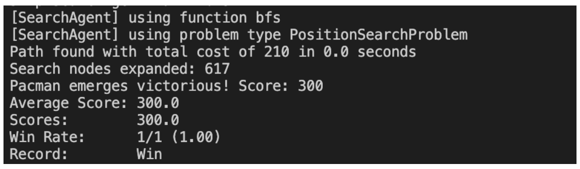
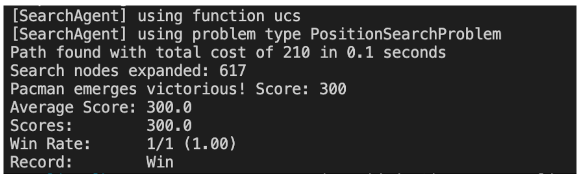


Figure 8 - *Expanded nodes of uniform-cost search algorithm for bigMaze*



```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 617
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 9 - *Results of breadth-first search algorithm for bigMaze*



```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 617
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 10 - *Results of uniform-cost search algorithm for bigMaze*

The output results shows pacman agent managed the path successfully using both implemented algorithms. Compared with depth-first search algorithms, both breadth-first and uniform-cost search algorithms expand more nodes to achieve the goal state. The agent has a greater perception of the game board and a greater knowledge of possible paths. However, although the previous tests show no significant differences between the two algorithms, through uniform-cost search the agent is able to make more thoughtful decisions in accordance with the shortest paths.

Depth-First vs Breadth-First vs Uniform-Cost Searches

In order to compare the costs of the solutions and the nodes explored for each map according to each search algorithm, the following table was constructed.

Map	Depth-First	Breadth-First	Uniform-Cost
tinyMaze	10	8	8
smallMaze	49	19	19
mediumMaze	130	68	68
bigMaze	210	210	210
openMaze	298	54	54
contoursMaze	85	13	13

Table 1 - *Comparison of cost of the found path to achieve the pacman agent's goal*

Map	Depth-First	Breadth-First	Uniform-Cost
tinyMaze	15	15	15
smallMaze	59	90	90
mediumMaze	146	267	267
bigMaze	390	617	617
openMaze	576	679	679
contoursMaze	85	165	165

Table 2 - *Comparison of the number of nodes expanded to find the path to achieve the pacman agent's goal*

Comparing the previous two tables, sometimes it is more rewarding to expand more nodes in order to obtain better low-cost solutions. Depth-First search expands less nodes compared to other algorithms, however it only returns the best solution in the case of the bigMaze. This board is characterized by long and deep paths, hence obtaining good results with depth-first search.

As mentioned above, depth-first and breadth-first search algorithms are similar, distinguishing only in the order of the node expansion. It is clear that the order of the expansion matters and can affect the results of the algorithms.

Breadth-First and uniform-cost search presents similar results both in the cost of the found path and in the number of expanded nodes during search. Nevertheless, uniform-cost gives priority to the lower cost paths during the search which can be an advantage in other scenarios.

Least Cost Solution

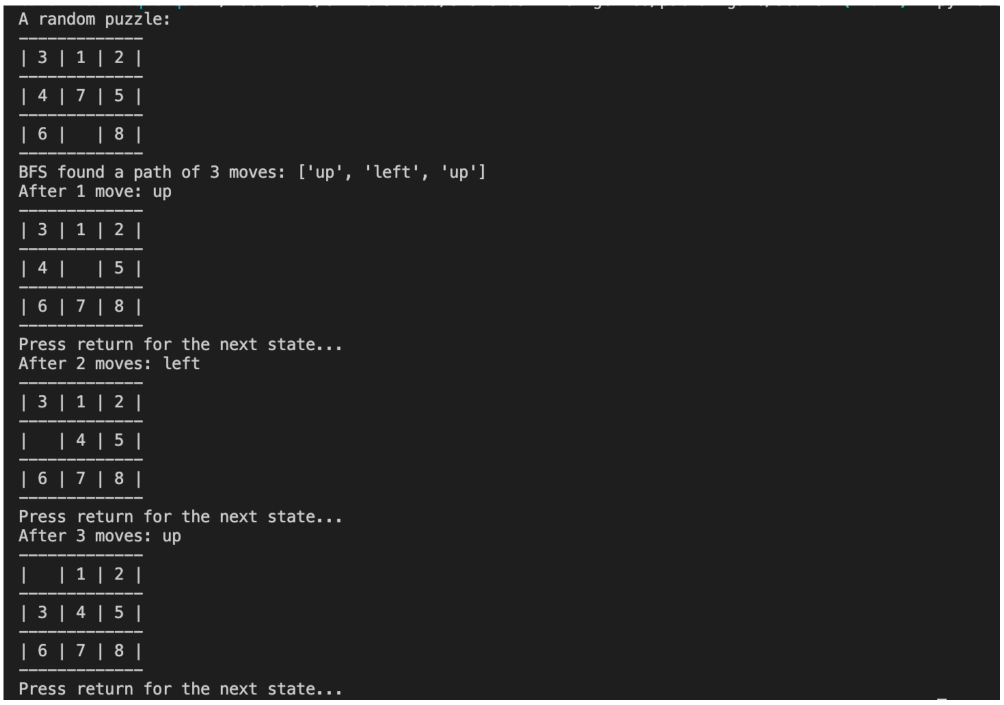
Map	Least Cost	Depth-First	Breadth-First	Uniform-Cost
tinyMaze	8	✗	✓	✓
smallMaze	19	✗	✓	✓
mediumMaze	68	✗	✓	✓
bigMaze	210	✓	✓	✓
openMaze	54	✗	✓	✓
contoursMaze	13	✗	✓	✓

Table 3 - *Algorithms that returns the least cost solution for each maze*

Analysing Table 3, depth-first search algorithm only returns one least cost solution (bigMaze) while breadth-first and uniform-cost search often return the least cost solution for each maze. Generally, breath-first and uniform-cost algorithms present better performance compared to depth-first.

8-Puzzle Game

The 8-puzzle game is a sliding puzzle that consists in order numbered square pieces by making strategy moves using the empty space. As breadth-first search algorithm was implemented generically, this algorithm also solves a random 8-puzzle game provided by *eightpuzzle.py* file.



```
A random puzzle:
-------------
| 3 | 1 | 2 |
-------------
| 4 | 7 | 5 |
-------------
| 6 |   | 8 |
-------------
BFS found a path of 3 moves: ['up', 'left', 'up']
After 1 move: up
-------------
| 3 | 1 | 2 |
-------------
| 4 |   | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 2 moves: left
-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 3 moves: up
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
```

Figure 11 - *8-puzzle game solved using breadth-first search*

The previous random puzzle generated was solved with only three moves as seen in Figure 11.

MediumDottedMaze and MediumScaryMaze

In addition to the comparison established between the different search algorithms already implemented, the behaviour of the agent was also analyzed for two other different mazes: MediumDottedMaze and MediumScaryMaze.

- **MediumDottedMaze:** medium maze with more dots
- **MediumScaryMaze:** a little bit different medium maze, more spaced with some ghost agents

As suggested in the problem statement, each maze was tested with two different types of agents which force certain behaviors of the SearchAgent:

- **StayEastSearchAgent:** agent that benefits positions on the East side of the board using a cost function of *1/2^x* for stepping into a position (x,y)

- **StayWestSearchAgent**: agent that penalizes positions on the East side of the board using a cost function of 2^x for stepping into a position (x,y)

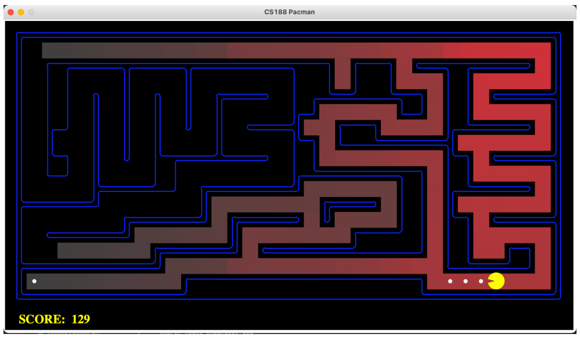


Figure 12 - Solving mediumDottedMaze using a StayEastSearchAgent

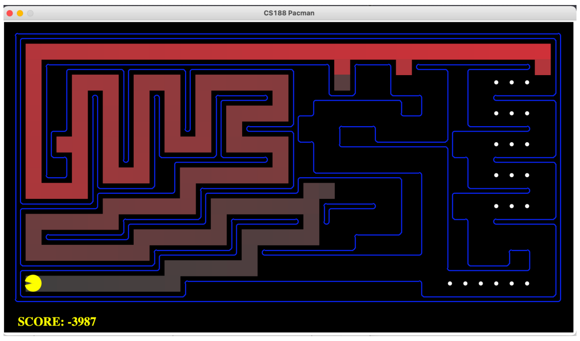
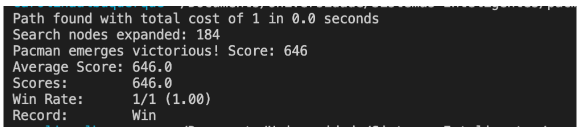
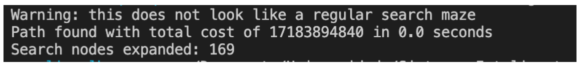


Figure 13 - Solving mediumDottedMaze using a StayWesSearchAgent



```
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 184
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 14 - Results of mediumDottedMaze using a StayEastSearchAgent



```
Warning: this does not look like a regular search maze
Path found with total cost of 17183894840 in 0.0 seconds
Search nodes expanded: 169
```

Figure 15 - Results of mediumDottedMaze using a StayWestSearchAgent

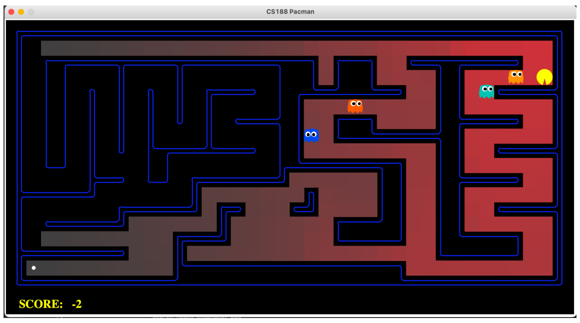


Figure 16 - Solving mediumScaryMaze using a StayEastSearchAgent

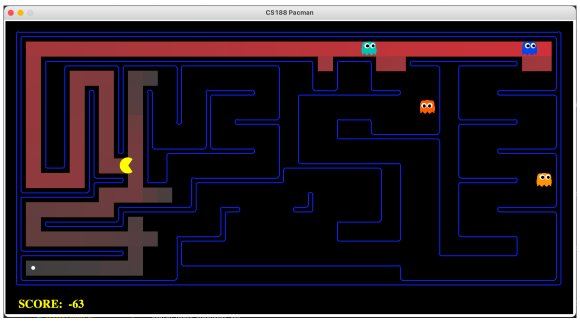
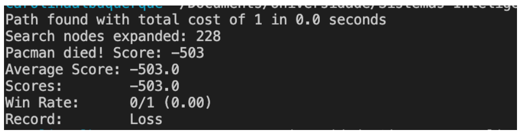
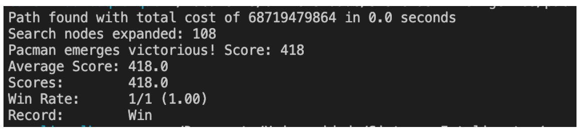


Figure 17 - Solving mediumScaryMaze using a StayEastSearchAgent



```
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 228
Pacman died! Score: -503
Average Score: -503.0
Scores:        -503.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

Figure 18 - *Results of mediumScaryMaze using a StayEastSearchAgent*



```
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 108
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:        418.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Figure 19 - *Results of mediumScaryMaze using a StayWestSearchAgent*

Forcing the search agent to benefit east positions, as mediumDottedMaze contains several dots on the east side of the board, the agent can eat all dots and achieve the final dot goal. On the other hand, when the agent penalizes east positions, all dots positioned on the east side of the board will be ignored and the agent never ends the game.

Testing these special agents in a different maze, mediumScaryMaze that contains ghosts, using the StayEastAgent, this agent will die because when the game starts the ghosts are positioned on the east side of the board. However, the agent can survive longer or even successfully win the game if the agent is a StayWestAgent.

Conclusions

To sum up, both algorithms already implemented have their advantages depending on the application scenario. As seen in the results discussion, breadth-first and uniform-cost search behaved better compared to depth-first search algorithms in almost all test scenarios.

A conclusion drawn from the implementation of these steps is based on the similarity between depth-first and breadth-first search, with only the order of searching being varied, affecting significantly the results obtained.

In addition, uniform-cost can be considered a powerful algorithm since it gives priority to the shortest paths during the search and can optimize the search process.