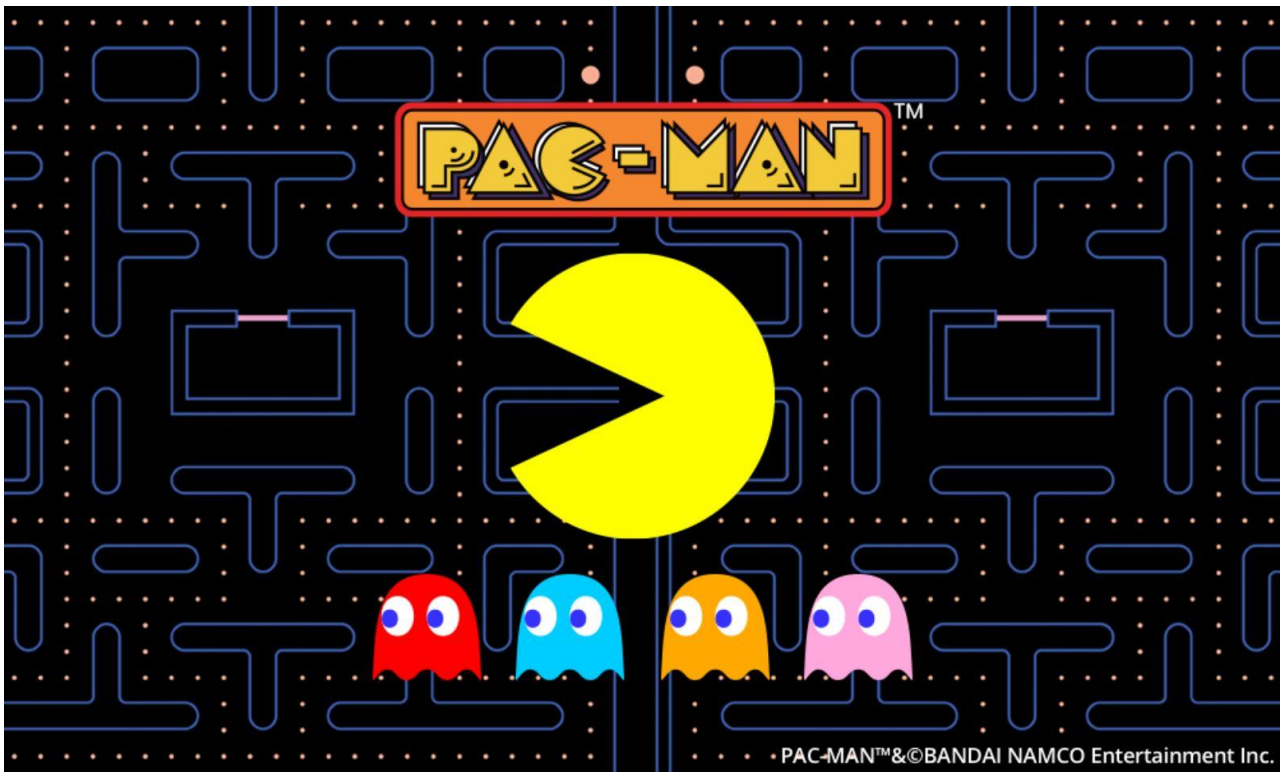
Subject: Artificial Intelligence (CS-860)

Packman agent – Path finding search algorithm techniques.



Course: Artificial Intelligence

Problem Statement

In this assignment, we will embark on a fascinating journey alongside Pacman, our iconic video game character. Our mission is to help Pacman navigate through intricate maze worlds, find his way to specific destinations, and collect food efficiently. To achieve this, we will be tasked with implementing essential search algorithms and designing heuristic strategies tailored for Pacman's universe. The code governing Pacman's behavior is already functional. Our primary task is to focus on the algorithms that will control how Pacman moves. The code is readily available for download, and you'll find everything you need to get started either on your university's LMS section. The codebase includes Python files, some for reading and understanding, some for modifying, and some that you can safely ignore.

Key Files and Components:

- Files we'll edit:
    - search.py (for your search algorithms)
    - searchAgents.py (for your search-based agents)
- Files we should look at:
    - pacman.py (the main file running Pacman games)
    - game.py (defining the Pacman world logic)
    - util.py (providing helpful data structures for search algorithms)

The Quest Begins:

Implementation of search algorithms: Students must implement the search algorithms (DFS, BFS, UCS, A*) in the search.py file. These algorithms should find paths through the Pacman maze while considering maze layouts and constraints. Hence these 4 Problems will be:

- ✓ Problem 1: Finding Food by Depth-First Search (DFS)
- ✓ Problem 2: Breadth-First Search (BFS)
- ✓ Problem 3: Uniform Cost Search (UCS)
- ✓ Problem 4: A Search*

CODE

As the main code “Search.py” is provided to us by the instructor with some other connecting/ reference files such as “commands.py”. Every command performs a different function depending on the different sizes of mazes and different work conditions. The space for the code for the different search algorithms is provided, where we will paste the respective code for making the decent changes according to the search algorithm technique that we want to apply for that respective code.

Here we will discuss one such case because the main code for the whole section will remain the same for every algorithm, just the calculating formula for the movement or the proportion will be changing accordingly to the algorithm. For example, for the DFS (Depth First Search) & the BFS (Best First Search), we only require the cost function value for propagation and no heuristic value is required. But in the case of A* (A-star) technique, heuristic value, and the cost function value both are required.

Here is the sample code-part that we will be adding for each respective search algorithm for the propagation:

```
138  def aStarSearch(problem, heuristic=nullHeuristic):
139    "Search the node that has the lowest combined cost and heuristic first."
140    astar = util.PriorityQueue()
141    astar.push((problem.getStartState(), []), 0)
142    explored = set()
143
144    while not astar.isEmpty():
145      (node, path) = astar.pop()
146      if problem.isGoalState(node):
147        return path
148      if node in explored:
149        continue
150      explored.add(node)
151      for n, p, c in problem.getSuccessors(node):
152        if n not in astar.heap and n not in explored:
153          astar.push((n, path + [p]), c + heuristic(n, problem))
154    return False
```

This code defines the A* search algorithm, which is used to find the optimal path in a search problem. It combines known costs and heuristic estimates to explore states in a way that efficiently leads to the best solution. This makes A* a widely used search algorithm for various pathfinding and optimization problems. All the other codes will be same as this one in sequence just their value changes at the priority queue (Line-153 for this case). Here, “*astar*” is just used as a variable it can be replaced with anything. Just used here for differentiating the code for ither search algorithms.

Here's a breakdown of what this code does:

- *def aStarSearch(problem, heuristic=nullHeuristic):* This function implements the search algorithm, which is used to find the optimal path in a search problem. It takes two main arguments: problem, representing the search problem, and heuristic, which is an optional heuristic function. The heuristic function provides an estimate of the cost from the current state to the nearest goal state. The default heuristic, when not provided, is set to *nullHeuristic*, which returns a constant value of 0.
- *astar = util.PriorityQueue()*: A priority queue is initialized to store nodes to be explored. Nodes in the queue are prioritized based on their estimated total cost, which is the sum of the known cost to reach the node and the heuristic estimate. Here, "*astar*" is just used as a variable.
- *astar.push((problem.getStartState(), []), 0):* It starts by pushing the initial state of the problem (start state) into the priority queue. The second argument is an empty list, representing the path taken to reach this state. The third argument is the estimated total cost, which is initially set to 0.

The main loop continues as long as the priority queue is not empty:

- *(node, path) = astar.pop()*: It pops the node with the lowest estimated total cost from the priority queue. node represents the current state, and path is the path taken to reach this state.
- If the current state is the goal state (*problem.isGoalState(node)),* the function returns the path taken to reach that state (return path), indicating a successful solution.

If the current state is not the goal state, it explores the successors of the current state:

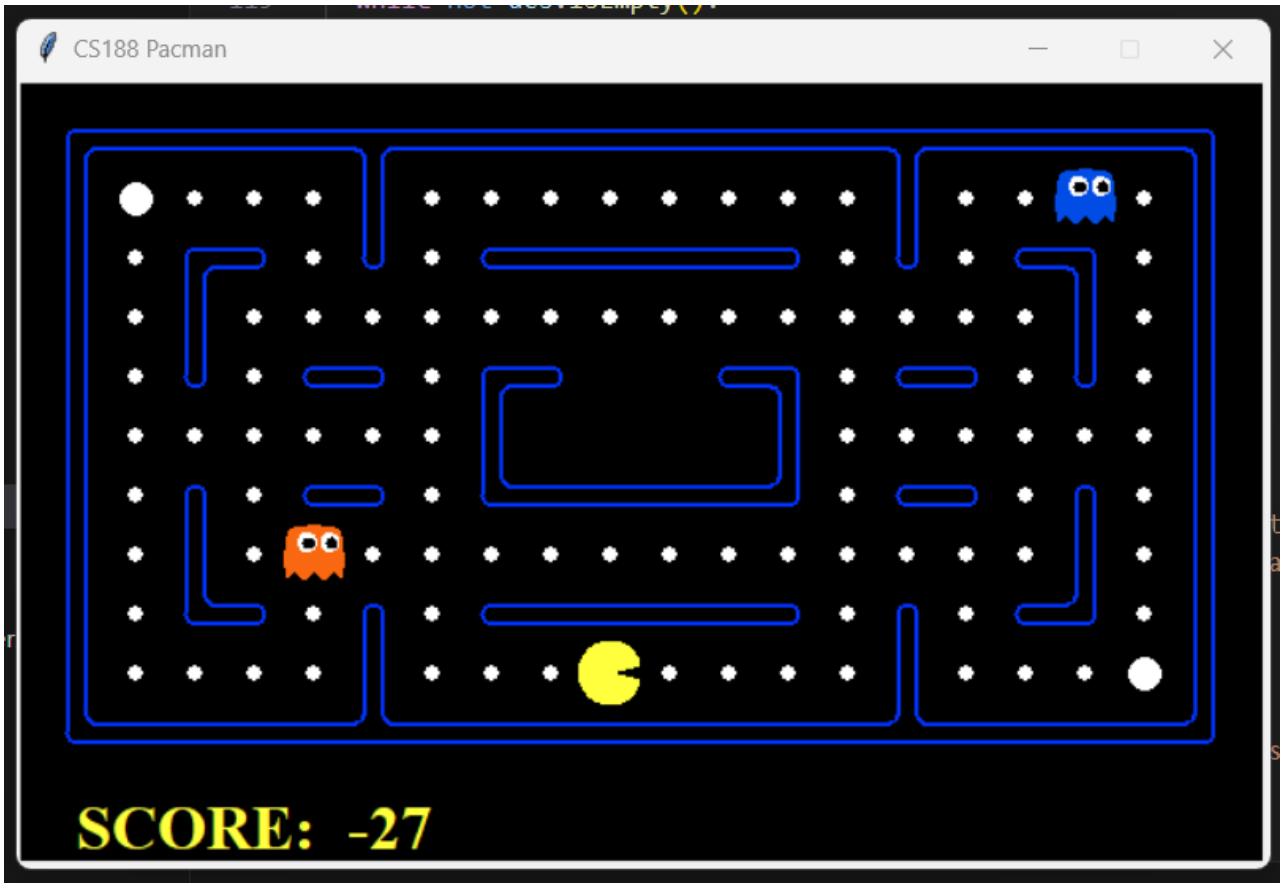
- (n, p, c) are the successor state, the action taken to reach that state, and the cost to move to the successor, respectively.
- It checks if the successor state (n) is not in the priority queue *(astar.heap)* and has not been explored already (not in explored). If both conditions are met, it adds the successor state to the priority queue with an updated path and estimated total cost *(astar.push((n, path + [p]), c + heuristic(n, problem)))*. This step will be different for other search algorithms too for calculating the path strategy.
- The loop continues until a solution is found or until the priority queue is empty. If no solution is found, the function returns False.

Sample Maze

A sample maze is provided to us for the better understanding of code and to check the credibility and working of this whole code that who is it working. Here we will be testing it by running the code at first and then applying the following command in terminal (commands are present in command.py file provided with the designated code):

```
python pacman.py
```

Here is the screenshot of that maze which will pop on running the code with testing command.



PROBLEMS

Here we will be applying the Path finding search algorithm techniques for finding the different path strategies for our agent using the cost value and the heuristic values in different sizes of mazes.

Problem-1: Finding food by DFS (Depth First Search)

Code:

```
71  def depthFirstSearch(problem):
72    """
73    Search the deepest nodes in the search tree first
74    [2nd Edition: p 75, 3rd Edition: p 87]
75
76    Your search algorithm needs to return a list of actions that reaches
77    the goal.  Make sure to implement a graph search algorithm
78    [2nd Edition: Fig. 3.18, 3rd Edition: Fig 3.7].
79    """
80    dfs = util.Stack()
81    dfs.push((problem.getStartState(), [], 0))
82
83    while not dfs.isEmpty():
84      node, path, cost = dfs.pop()
85      if problem.isGoalState(node):
86        return path
87      for n, p, c in problem.getSuccessors(node):
88        if n not in dfs.list and n not in problem._visitedlist:
89          bfs.push((n, path + [p], c))
90    return False
```

The code is an implementation of the depth-first search (DFS) algorithm. The DFS algorithm is used to traverse or search trees or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. The code uses a stack data structure to keep track of nodes that need to be explored in the maze. The algorithm starts by pushing the start state of the game onto the stack. It then enters a loop where it pops a node from the stack, checks if it is the goal state, and if not, pushes its successors onto the stack. The algorithm continues until either the goal state is found, or the stack is empty.

1. ***Command: python pacman.py -l tinyMaze -p SearchAgent***

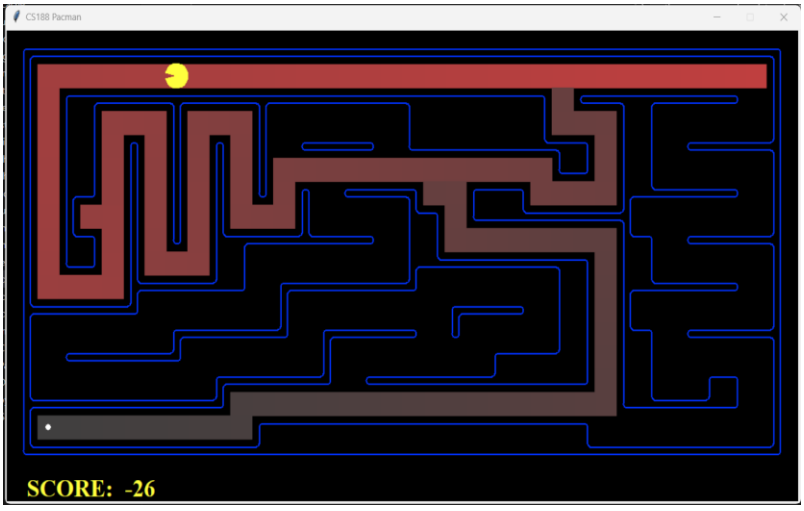
This screen shot here provides the terminal Output that we get to see after running the programs for depth-first search. Here different cases and maze sizes are provided to be run according to the conditions for output.

Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	DFS
Scenario	Tiny Maze
Total Cost	10
Nodes Expanded	15
Average Score	500
Record	Win/Reached Goal

2. ***Command: python pacman.py -l mediumMaze -p SearchAgent***

Here this picture is the screenshot of the terminal output that we will get while successfully sunning the program at this command.

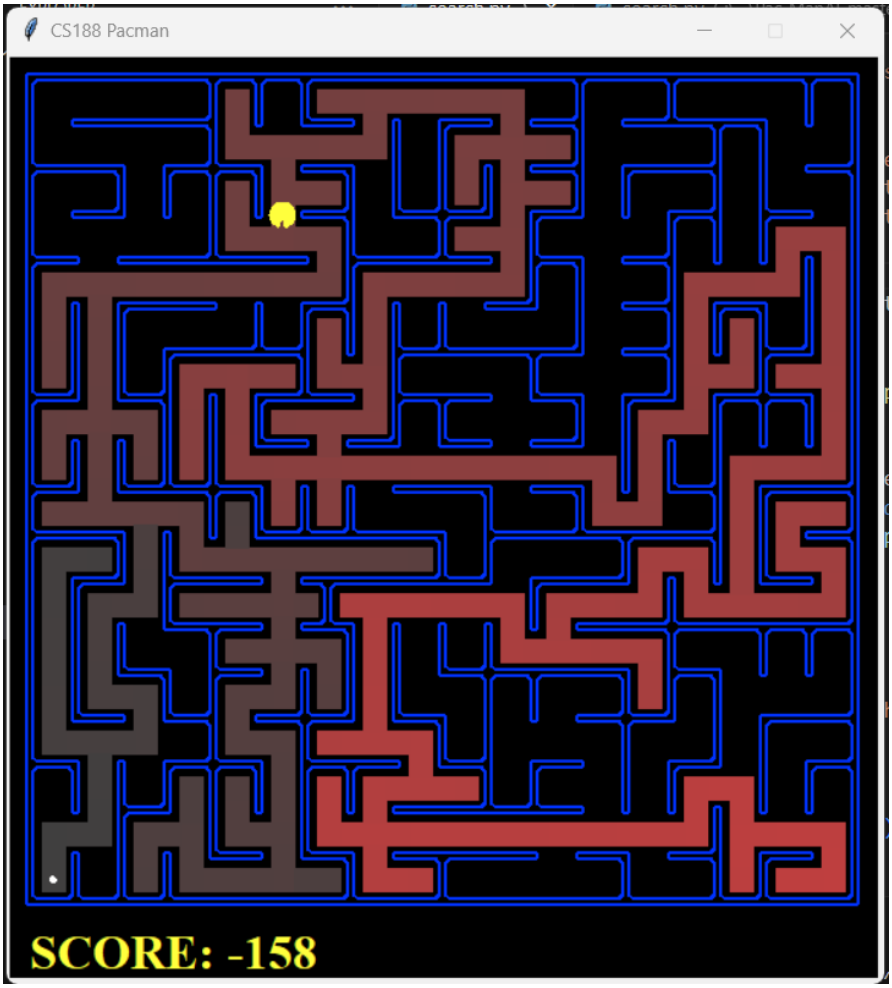


Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	DFS
Scenario	Medium Maze
Total Cost	130
Nodes Expanded	146
Average Score	380
Record	Win/Reached Goal

3. Command: python pacman.py -l bigMaze -z .5 -p SearchAgent

This screen shot here provides the terminal Output that we get to see after running the programs for depth-first search. Here different cases and maze sizes are provided to be run according to the conditions for output.



Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	DFS
Scenario	Big Maze
Total Cost	210
Nodes Expanded	390
Average Score	300
Record	Win/Reached goal

Problem-2: BFS (Breadth First Search)

Code:

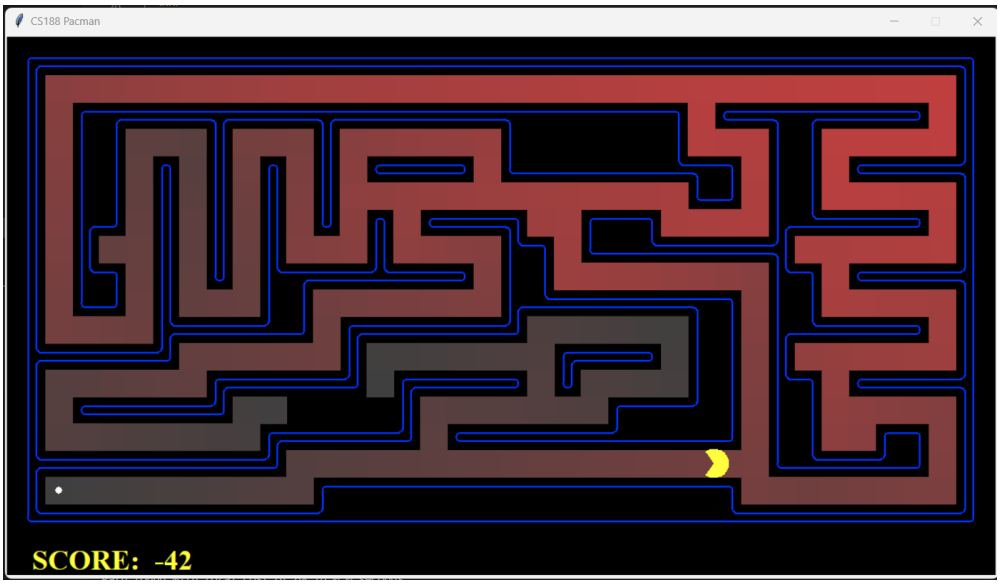
```
93  def breadthFirstSearch(problem):
94      """
95      Search the shallowest nodes in the search tree first.
96      [2nd Edition: p 73, 3rd Edition: p 82]
97      """
98      bfs = util.Queue()
99      bfs.push((problem.getStartState(), [], 0))
100     explored = set()
101
102     while not bfs.isEmpty():
103       node, path, cost = bfs.pop()
104       if problem.isGoalState(node):
105         return path
106       if node in explored:
107         continue
108       explored.add(node)
109       for n, p, c in problem.getSuccessors(node):
110         if n not in bfs.list and n not in explored:
111           bfs.push((n, path + [p], c))
112     return False
113
```

The code here is an implementation of the breadth-first search algorithm. It is a graph traversal algorithm that 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. While using BFS for traversal, any node in the graph can be considered as the root node. The code starts by initializing a queue `bfs` with a tuple containing the start state, an empty list `[]` for path, and `0` for cost. It also initializes an empty set `explored` to keep track of visited nodes. The while loop continues until `bfs` is empty. In each iteration, it pops a node from the front of the queue and checks if it is the goal state.

If it is, it returns to the path to reach that state. If not, it adds the node to `explored` and expands its successors by adding them to `bfs`. The successors are added only if they are not already in `bfs` or `explored`. Finally, if no goal state is found, it returns False.

1. ***Command: python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs***

This screen shot here provides the terminal Output that we get to see after running the programs for breadth-first search. Here different cases and maze sizes are provided to be run according to the conditions for output.

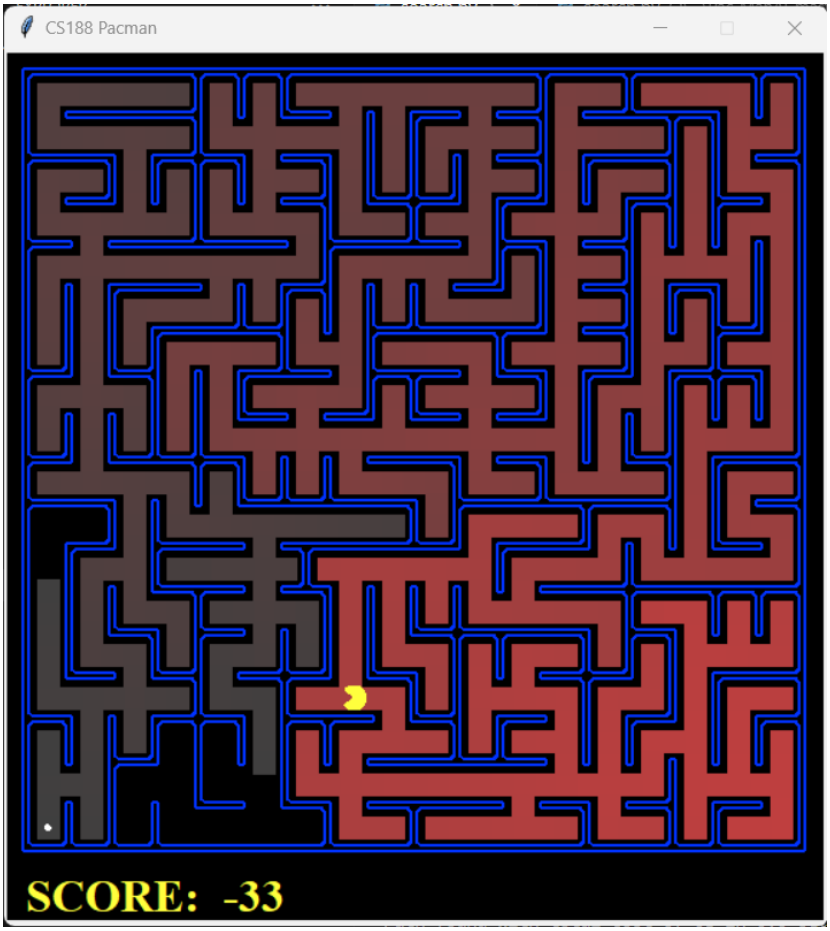


Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	BFS
Scenario	Medium Maze
Total Cost	68
Nodes Expanded	269
Average Score	442
Record	Win/Reached Goal

2. ***Command: python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5***

This screen shot here provides the terminal Output that we get to see after running the programs for breadth-first search. Here different cases and maze sizes are provided to be run according to the conditions for output

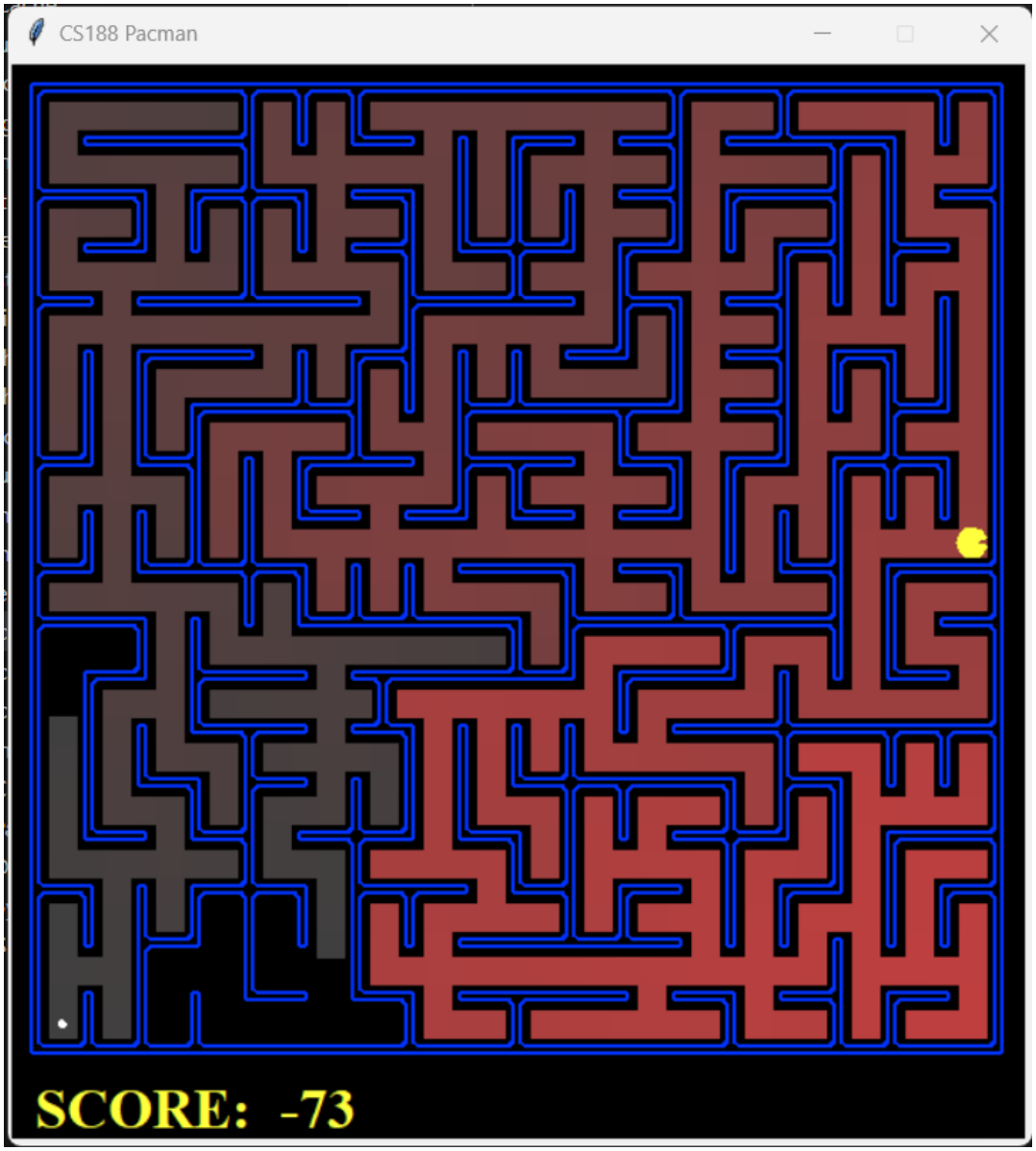


Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	BFS
Scenario	Big Maze
Total Cost	210
Nodes Expanded	620
Average Score	300
Record	Win

3. Command: python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime

This screen shot here provides the terminal Output that we get to see after running the programs for breadth-first search. Here different cases and maze sizes are provided to be run according to the conditions for output.



Here is the table that shows the statistics for this iteration:

Statistics	**Conclusions**
Algorithm	BFS
Scenario	Big Maze + Frame-Time=0
Total Cost	210
Nodes Expanded	620
Average Score	300
Record	Win/Reached goal

Problem-3: UFS (Uniform Cost Search)

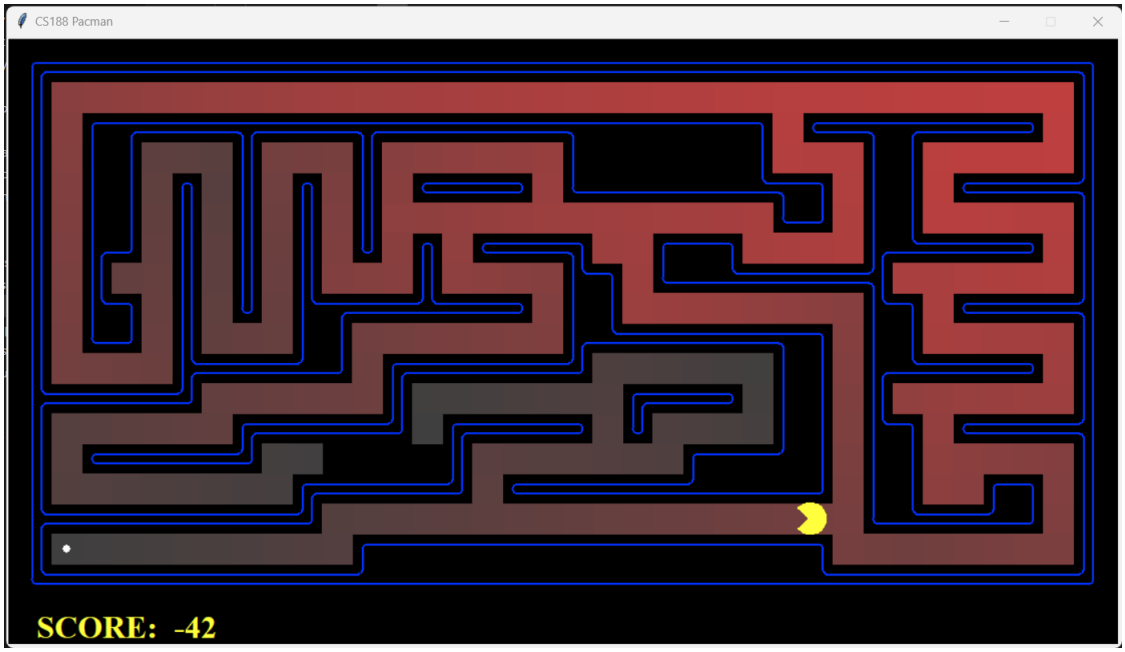
Code:

```
114  def uniformCostSearch(problem):
115    "Search the node of least total cost first. "
116    ucs = util.PriorityQueue()
117    ucs.push((problem.getStartState(), []), 0)
118    explored = set()
119
120    while not ucs.isEmpty():
121      (node, path) = ucs.pop()
122      if problem.isGoalState(node):
123        return path
124      if node in explored:
125        continue
126      explored.add(node)
127      for n, p, c in problem.getSuccessors(node):
128        if n not in ucs.heap and n not in explored:
129          ucs.push((n, path + [p]), c)
130    return False
```

The code provided is an implementation of the Uniform Cost Search (UCS) algorithm in Python. This algorithm is used to find the node of least total cost first. The algorithm works by maintaining a priority queue of nodes to be explored, with the node with the lowest cost being explored first. The algorithm continues to explore nodes until it reaches the goal state or there are no more nodes to explore. The UCS algorithm guarantees that it will find the optimal solution if one exists. The code you provided initializes a priority queue and pushes the start state onto it with a cost of 0. It then enters a loop where it pops the node with the lowest cost from the priority queue and checks if it is the goal state. If it is, it returns the path to that node. If not, it adds the node to a set of explored nodes and expands its successors, adding them to the priority queue if they have not been explored before. If there are no more nodes in the priority queue, the algorithm returns False.

1. ***Command: python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs***

This screen shot here provides the terminal Output that we get to see after running the programs for Uniform Cost search. Here different cases and maze sizes are provided to be run according to the conditions for output.

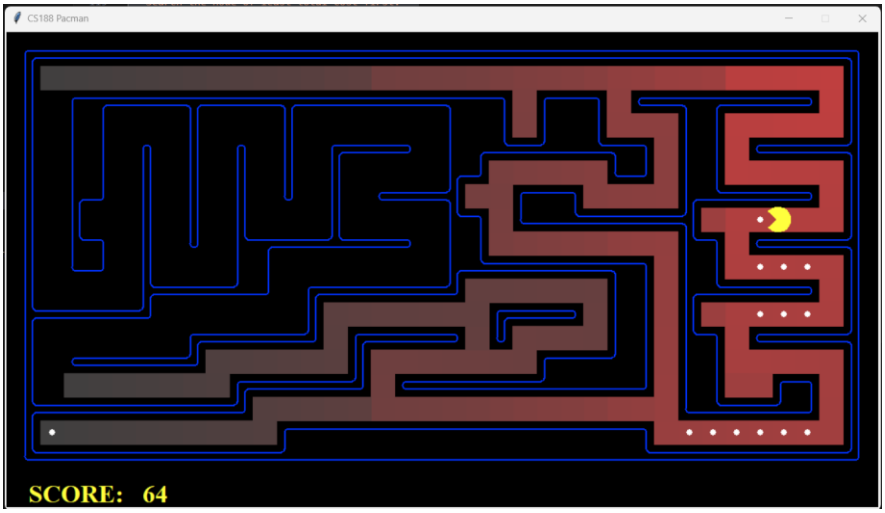


Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	UCS
Scenario	Medium Maze
Total Cost	68
Nodes Expanded	269
Average Score	442
Record	Win/Reached Goal

2. Command: python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

Here, attached picture is the screenshot of the terminal output that we will get while successfully running the program at this command.

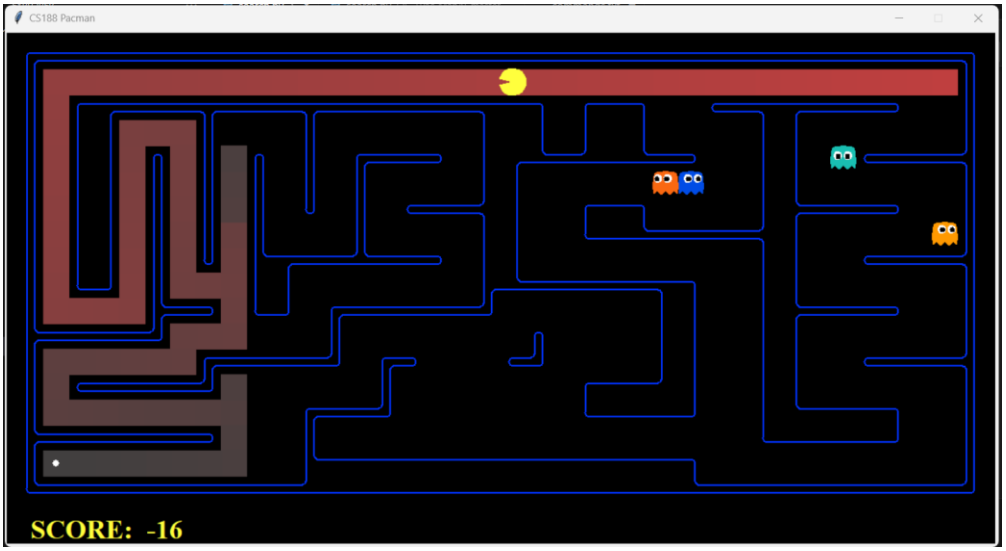


Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	UCS
Scenario	Medium Dotted Maze
Total Cost	1
Nodes Expanded	186
Average Score	646
Record	Win/Reached Goal

3. ***Command: python pacman.py -l mediumScaryMaze -p StayWestSearchAgent***

This screen shot here provides the terminal Output that we get to see after running the programs for uniform cost search. Here different cases and maze sizes are provided to be run according to the conditions for output.



Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	UCS
Scenario	Medium Scary Maze
Total Cost	68719479864
Nodes Expanded	98
Average Score	418
Record	Win/Reached Goal

Problem-4: A* Search (A-star)

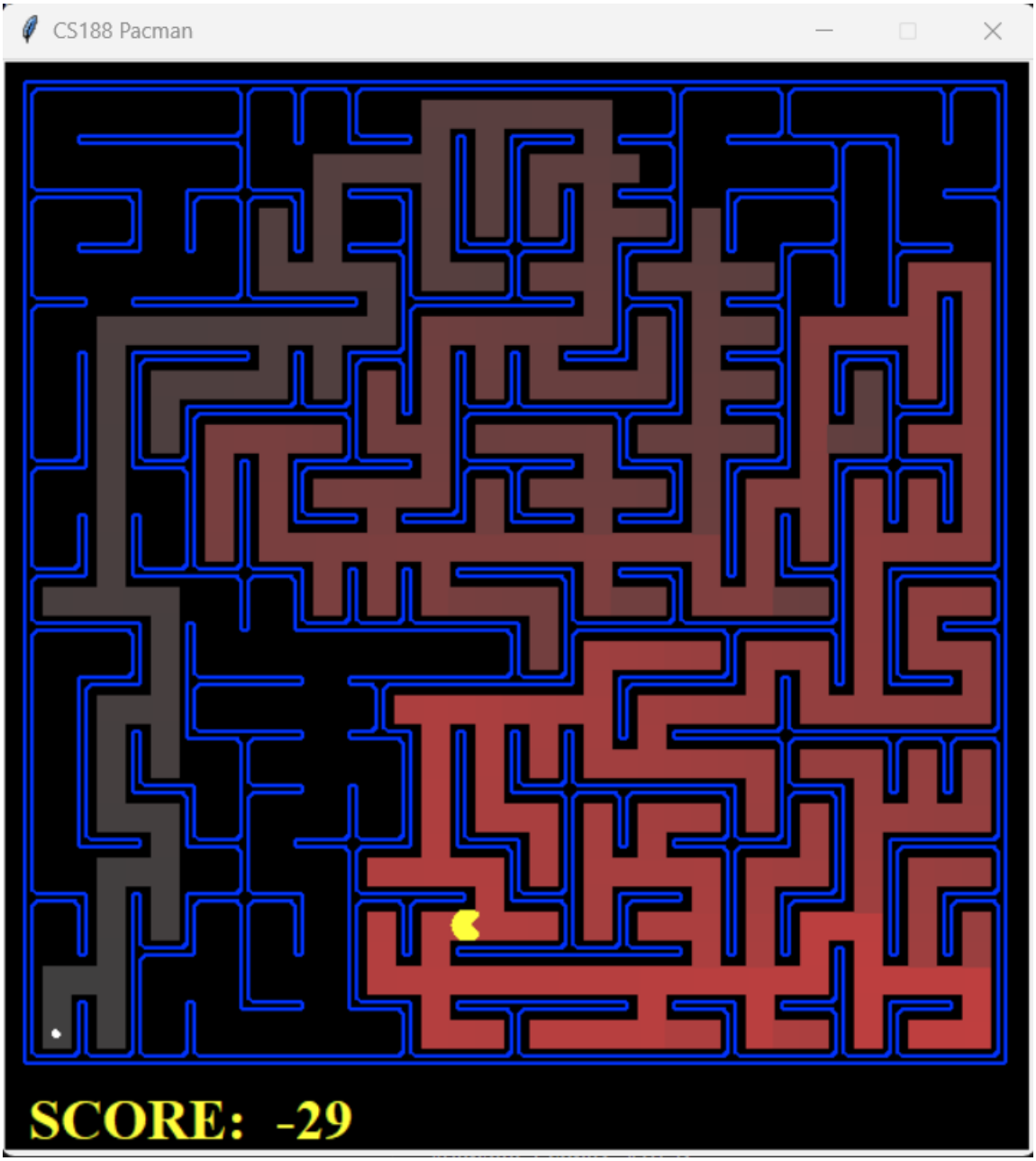
Code:

```
138  def aStarSearch(problem, heuristic=nullHeuristic):
139      "Search the node that has the lowest combined cost and heuristic first."
140      frontier = util.PriorityQueue()
141      frontier.push((problem.getStartState(), []), 0)
142      explored = set()
143
144      while not frontier.isEmpty():
145          (node, path) = frontier.pop()
146          if problem.isGoalState(node):
147              return path
148          if node in explored:
149              continue
150          explored.add(node)
151          for n, p, c in problem.getSuccessors(node):
152              if n not in frontier.heap and n not in explored:
153                  frontier.push((n, path + [p]), c + heuristic(n, problem))
154      return False
```

The A* Search function is a search algorithm that finds the node with the lowest combined cost and heuristic first. The function takes two parameters: problem and heuristic. The problem parameter represents the problem to be solved, while the heuristic parameter is an optional parameter that represents the heuristic function. The function uses a priority queue to store nodes and their associated costs. It starts by pushing the start state of the problem onto the priority queue with a cost of 0. It then enters a loop that continues until the priority queue is empty. In each iteration of the loop, it pops the node with the lowest cost from the priority queue and checks if it is the goal state. If it is, it returns the path to that node. If not, it adds the node to a set of explored nodes and expands its successors. The successors are added to the priority queue if they are not already in it or in the set of explored nodes. The cost of each successor is calculated as the sum of its path cost and its heuristic value. If no path to the goal state is found, it returns False.

- ***Command: python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic***

This screen shot here provides the terminal Output that we get to see after running the programs for A* search. Here different cases and maze sizes are provided to be run according to the conditions for output.



Here is the table that shows the statistics for this iteration:

Statistics	Conclusions
Algorithm	A*
Scenario	Big Maze
Total Cost	210
Nodes Expanded	466
Average Score	300
Record	Win/Reached Goal

Comparison Table

Algorithm	Maze Type (with Conditions)	Cost	Nodes (Explored)	Optimality	Efficiency
DFS	Small	10	15	Not Optimal	Efficient Nodes
	Medium	130	146	Not Optimal	Efficient Nodes
	Big	210	390	Not Optimal	Efficient Nodes
BFS	Medium	68	269	Optimal	Higher Efficient nodes
	Big	210	620	Optimal	Higher Efficient nodes
UCS	Medium	68	269	Optimal	Higher Efficient nodes
	Medium Dotted	1	186	Optimal	Higher Efficient nodes
	Medium Scary	6871947986	98	Optimal	Lower Efficient nodes
A*	Big	210	549	Optimal	Efficient Nodes

In the context of algorithms, “Efficiency” refers to how well an algorithm balances finding an optimal solution and the number of nodes expanded. An optimal solution guarantees finding the shortest path, while a non-optimal solution may not always find the shortest path. The number of nodes expanded can vary based on the problem’s characteristics and heuristics used. Efficiency is a measure of how well an algorithm performs in terms of computational resources used. It is important to note that efficiency is not the same as optimality. Optimality refers to the best possible solution according to a given criterion, while efficiency refers to how well an algorithm performs in terms of resource usage. Therefore, an algorithm can be efficient without being optimal, but an optimal algorithm is always efficient.

Conclusion

DFS, BFS, UCS, and A* Search are all search algorithms that have their own advantages and disadvantages. DFS explores deeper into the search tree before backtracking, making it less efficient for finding optimal solutions. BFS explores all nodes at a given depth level before moving deeper into the search tree. It guarantees finding the shortest path, but it can be inefficient in terms of memory and time for larger search spaces. UCS is similar to BFS in that it guarantees finding the shortest path by considering the cost of actions. However, it can handle different cost functions, making it more versatile. A* Search combines the advantages of UCS and a heuristic function (in this case, Manhattan distance). It is effective in finding optimal solutions faster than UCS, as seen in the Big Maze example.

In summary, the choice of the search algorithm depends on the specific problem and requirements. If optimality is crucial and memory constraints are not an issue, BFS or UCS with an appropriate cost function is a good choice. If optimality is essential but memory is a concern, A* with an admissible heuristic is preferred. If optimality is not a strict requirement and you need a quick but suboptimal solution, DFS may be sufficient. Remember that these results are specific to the problems and heuristics used in your examples, and the performance of these algorithms can vary for different scenarios and heuristics.

The main two outcomes of that we should keep in mind are:

- The choice of search algorithm depends on the specific problem and requirements.
- The performance of these algorithms can vary for different scenarios and heuristics.

References

Software Used for assignment:

- VS (Visual Studio)
- Python 3.11

Online Resources for understanding of Searches:

- https://stackoverflow.com/questions/2604022/pathfinding-algorithm-for-pacman
- https://github.com/topics/pacman-game?l=python&o=asc&s=stars

For understanding purposes:

- OpenAI. (2023). ChatGPT (Sep 25 version) [Large language model]. https://chat.openai.com/chat.
