

Simon Chaisouang

[shchaiso@uark.edu](mailto:shchaiso@uark.edu)

010878218

CSCE 5613 Artificial Intelligence

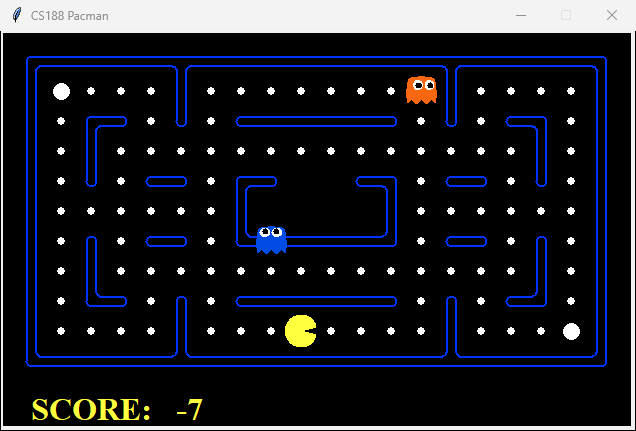
Spring 2025

Late Days: 0

## 1 - Introduction

In this report, the task is given to implement four different search algorithms into a Pacman game. The game environment is set up in Python and a search problem is provided where, from a given starting position, Pacman must find the shortest or optimal path through a given maze to his destination. There are also some additional constraints such as:

* The wall cannot be hit more than 2 times.
* The wall must be hit at least 1 time.

The implementation for the four different search algorithms is contained in the seach.py script. The four different search algorithms that explored in this report are: Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), and A\* Search (A\*). These four algorithms will be implemented and evaluated based on their efficiency and effectiveness in guiding Pacman to his food at the destination. 

**Figure 1. Sample Image of Pacman Game**

## 2 – Search Algorithms Implementations

**Depth-First Search**

The first search algorithm that is introduced and implemented is the Depth-First Search (DFS) Algorithm. DFS aims to explore deeply first but may not always find the best path. The code implementation starts by retrieving the starting state (.getStartState()) from the problem and initializes a stack (LIFO – Last in first out) to store different states, the amount of the times Pacman has hit the wall, and the actions taken. A visited set is initialized to keep track of explored states to prevent redundant searches.

A nested loop is enacted where the algorithm pops the most recent added state from the stack, checks if Pacman has reached the destination and expands the current node by retrieving its successors. An if constraint is added in the outer loop (while) to check is the wall hitting constraint is maintained. The inner loop (for) checks if a successor is a wall, the wall hit counter will increase; otherwise, it remains the same. Uninvited successors are added to the stack along with its updated path and walls hit count. The search continues until the stack is empty or PAX man has reached the food at the goal, at which the function will return the sequence of actions it took to get there. If no valid paths are found, an empty list is returned.

**Breath-First Search**

The Breath-First Seach (BFS) Algorithm aims to find the shortest path possible in an unweighted environment. The BFS implementation is very similar to the DFS, but with the only exception that a queue (FIFO – First in first out) is initialized in place of a stack (FILO) to systematically explore all possible moves at a given depth before moving onto the next level.

**Uniform Cost Search**

The Uniform Cost Search (UCS) Algorithm aims to find the lowest cost path where varying step costs exist. A priority queue (min-heap) is initialized that stores cost, position, wall hit count, and path to be taken. A distance dictionary is also established to track the minimum cost to reach each state while Pacman travels through the maze. A visited set is initialized to reduce redundancy within the explored states.

Similarly to the other implementations, a nested loop is established to check for if the wall hit constraints are fulfilled while updating the cost and priority queue with cheaper (lower cost) paths.

**A\* Search Algorithm**

The last algorithm that is implemented is the A\* Search Algorithm. A\* aims to efficiently find the most optimal path by using a heuristic (Manhattan Distance). The A\* expands upon the UCS implementation by implementing a heuristic function to guide the search towards the goal more efficiently. One notable difference from the previous implementation is priority is calculated by taking the cost to go plus heuristic. This ensures that the algorithm prioritizes the most promising paths. A heuristic function is called where Manhattan Distance is calculated by taking the absolute sum of difference between the x and y coordinates of the current state and goal state.

## 3 - Results

Below we output the results of each search algorithm over three different mazes (tiny, medium, and big).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | DFS | BFS | UCS | A\* |
| Nodes Expanded | 20 | 89 | 90 | 43 |
| Tiny Maze Score | 490 | 502 | 502 | 502 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | DFS | BFS | UCS | A\* |
| Nodes Expanded | 146 | 1321 | 1321 | 484 |
| Medium Maze Score | 366 | 462 | 462 | 462 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | DFS | BFS | UCS | A\* |
| Nodes Expanded | 382 | 1354 | 1325 | 488 |
| Big Maze Score | 304 | 464 | 464 | 464 |

**Figure 2. Tables of results of each algorithm over each maze**

From the results, we can see a noticeable difference between the scores of the DFS algorithm, and BFS, UCS, and A\* algorithms. Although in terms of score, DFS performed noticeably poorly, the algorithm significantly succeeds in maintaining a low node expanded count since it searches for any acceptable path. In practical applications where memory may be low, BFS could provide a valuable use case. For the other three algorithms, in terms of scores all achieve the same results, but A\* does maintain a significantly lower nodes expanded count compared to BFS and UCS. This showcases the algorithms effectiveness in return an optimal path without having to blindly explore like the other two.

|  |  |  |
| --- | --- | --- |
| Tiny Maze | Medium Maze | Big Maze |
| CS188 Pacman | CS188 Pacman | CS188 Pacman |

**Figure 3. Table of sample outputs of A\* Search implementation over three mazes (tiny, medium, big)**