PACMAN

Submitted By: Abubaker & ajwad
Roll-Number: i211379 & -

1) **Initialization**: Starting from the initial point, a search is performed using **depth-first search (DFS)**.

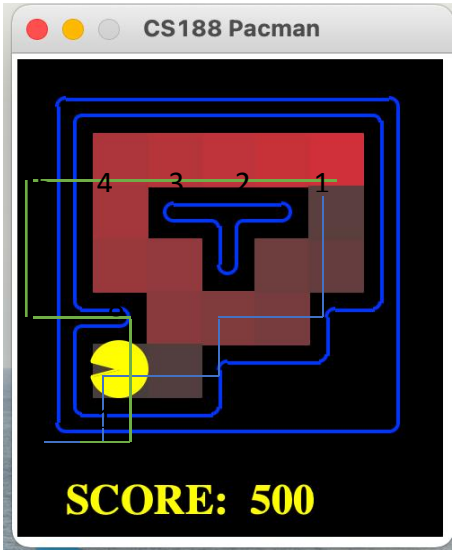
```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
[SearchAgent] using function tinyMazeSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 0
Pacman emerges victorious! Score: 502
Average Score: 502.0
Scores:        502.0
Win Rate:      1/1 (1.00)
Record:        Win
```

2)**Tiny Maze Search**: The algorithm uses a search technique to ensure correctness by utilizing the `searchAgent`. The algorithm starts by performing an initial search from the starting point.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l tinyMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l bigMaze -z .5 -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 390
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

According to the guidelines, the task requires designing a **searching algorithm** where the getSuccessor function will be used to generate successors in a search process. The algorithm needs to maintain and update a **fringe** (frontier) list, which will hold the states for further exploration. After generating the new successors, they will be added to this list, ensuring that the search continues based on the updated frontier.

3) **According to the guidelines**, the task requires designing a **searching algorithm** where the getSuccessor function will be used to generate successors in a search process. The algorithm needs to maintain and update a **fringe** (frontier) list, which will hold the states for further exploration. After generating the new successors, they will be added to this list, ensuring that the search continues based on the updated frontier.



In the context of `getSuccessor`**:**

The **fringe** list contains successors in the **South** and **West** directions. The **West** direction is discarded as it is already explored.

Phase 1: The fringe contains successors in the **South** and **West** directions.

Phase 2: The fringe contains successors in the **South** and **West** directions, with the **West** direction being used.

Phase 3: The fringe contains successors in the **South** and **West** directions, but since the **West** direction is already explored, we proceed with successors in the **South** direction.

Phase 4: The fringe contains successors in both the **North** and **South** directions. The **South** direction is explored, and we continue to the next step.

Phase 5: The fringe contains successors in both **North** and **West** directions. Since the **West** direction is already explored, we continue with successors in the **North** direction.

DFS Algorithm Behavior in Search:

When implementing DFS (Depth First Search), it selects the first node in the fringe (the list of nodes to explore) that has already been visited, as seen in the example.

Phase 1: A fringe node, already in the process of exploration, is selected first. This is because it appears earlier in the queue, meaning it is prioritized before others.

Phase 2: As DFS continues, it explores the next fringe nodes, which are added according to a **push** and **pop** order. If the fringe becomes empty or all paths are explored, DFS backtracks to find alternate paths.

Phase 3: If DFS reaches a node where no further progress can be made, it backtracks and returns to previous levels to continue searching from there.

BFS Algorithm Behavior:

In contrast, **Breadth First Search (BFS)** explores all nodes level by level. It uses a queue to manage nodes in **first-in-first-out** order. This guarantees that nodes closest to the starting point are explored first, ensuring the shortest path is found.

The search continues until all nodes are processed, expanding outward from the starting point.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Understanding BFS (Breadth-First Search)

State Handling: In BFS, different states (or nodes) are processed differently. The algorithm uses a **queue** to explore nodes, ensuring that nodes are visited level by level. This guarantees that the shortest path to a goal is found because it explores all possible paths from the start node before moving deeper into the graph.

Exploration Mechanism: BFS explores all neighbors of a node before moving to the next level. This method is distinct from other algorithms like DFS, which may go deeper into one path before returning.

Priority Queue in BFS: The text also mentions the **priority queue** in relation to BFS. Typically, BFS uses a regular queue to maintain the order of exploration. However, when modified with a priority queue (often seen in algorithms like **Dijkstra's** or *A search**), it ensures that the most promising states (with the lowest cost or highest priority) are explored first.

Queue Structure: BFS uses a **first-in-first-out (FIFO)** queue to manage nodes, ensuring that nodes at the front of the queue are processed first.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l bigMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

For the stayEastAgent, the task has been considered and planned.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```

And in the stayWestSearch state, the task has been mostly completed.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
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
```

*A Search**

The function used is:
f(n) = g(n) + h(n)

A* search operates based on this formula. It calculates the cost by taking into account the **heuristic** value.

By default, A* search assumes a **null heuristic**, meaning the heuristic value is considered to be zero.

If the heuristic value is zero, the A* search effectively becomes Uniform Cost Search (**UCS**).

When the **nullHeuristic** is used as the default, the heuristic value in the initial state is equivalent to UCS.

The results will be calculated as:
f(n) = g(n) + 0,
which is the same as UCS.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```

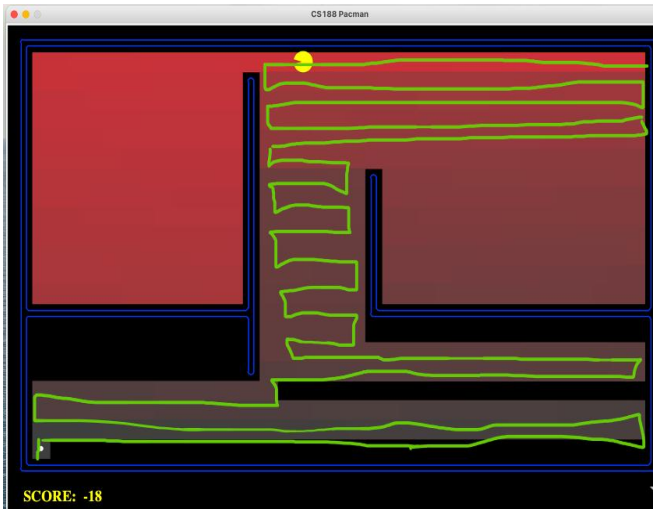
In the **bigMaze**,
when we change the **heuristic** to **Manhattan distance**,
we can observe that the number of states explored is reduced compared to **UCS**

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

openMaze

Run the algorithms you have implemented so far on the **openMaze** and explain what happens.

- **Algorithm: DFS (Depth First Search)**



```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l openMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.0 seconds
Search nodes expanded: 576
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
```

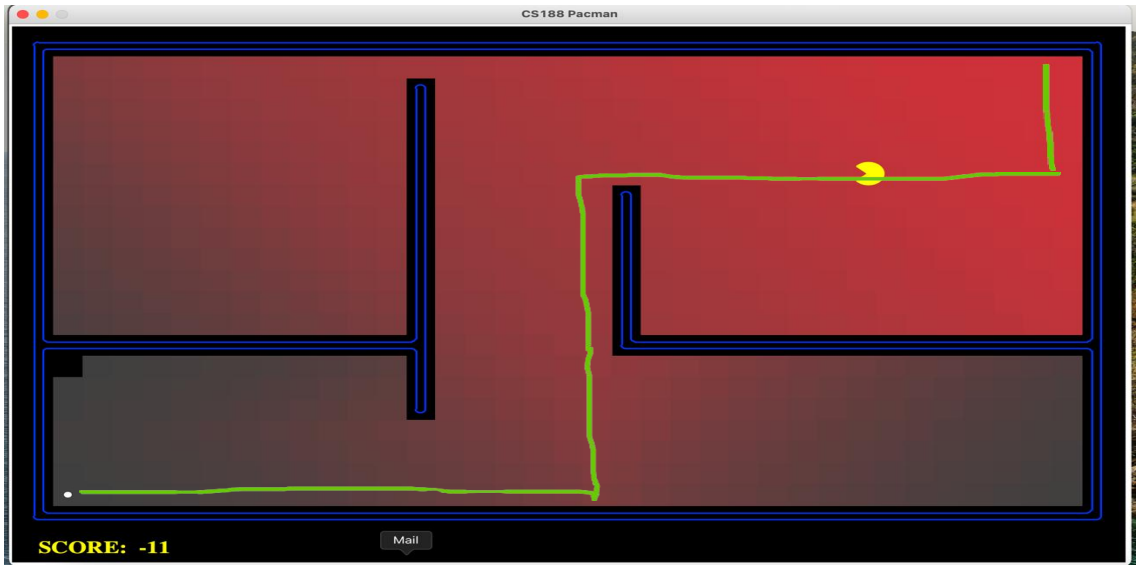
As we observe in practice, the left region appears redder.

For the **successor** function,
the order in which successors are added in the `getSuccessor` method is crucial. The successors are added in the order:
up, down, right, left,
and when popping, the leftmost element is processed first.

In **DFS**, we observe that the algorithm explores a path deeply without considering whether it is moving toward the goal or not (randomly). If it fails to reach the target, it is forced to backtrack one step and explore alternative paths.

This algorithm is **complete** but not **optimal**. While it guarantees exploration, it doesn't guarantee finding the shortest path. As a result, the algorithm is exhaustive but lacks efficiency.

Algorithm: BFS (Breadth-First Search)



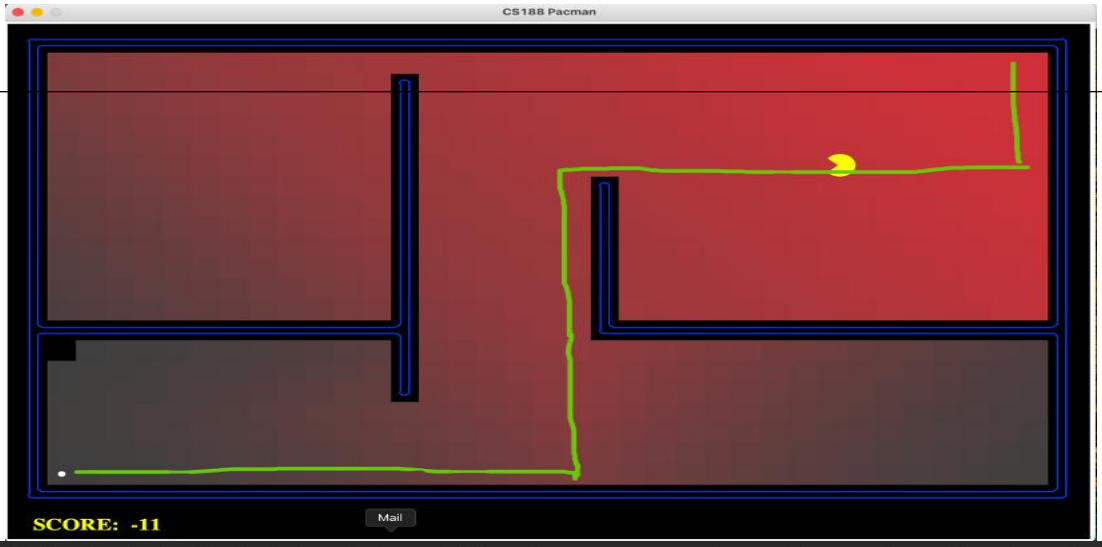

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l openMaze -p SearchAgent -a fn=bfs
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

The **BFS algorithm** first processes all the states at the same depth level (e.g., depth 1). Since it uses a queue, it then processes all the states at the next depth level (e.g., depth 2). This is why the right side of the maze appears more explored.

Additionally, BFS is **complete** because it explores all possible states systematically. However, because it moves from one depth level to the next, it explores more states at shallower depths before going deeper.

Unlike DFS, BFS does not behave randomly and systematically explores all possible successors. If the order of successors were different, the exploration pattern would change. In this case, all four successors are uniformly explored.

Thus, BFS is **complete** but not always **optimal** in terms of efficiency (space or time), especially in cases where there are differences in successor structures.




```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l openMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- **Algorithm: UCS (Uniform Cost Search)**

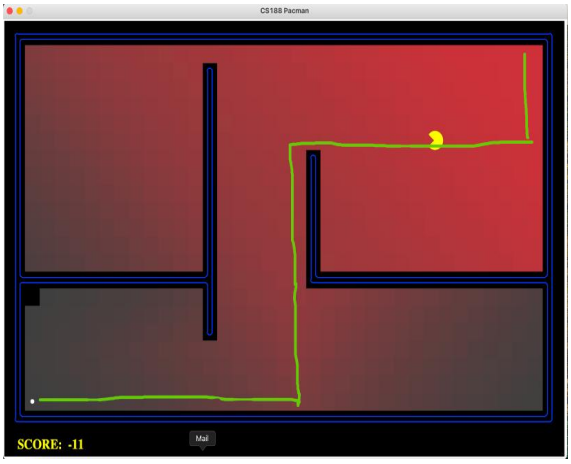
This **algorithm** is similar to BFS.

The difference is that in BFS, we consider all states at the same depth, but here, all states are considered with their respective costs. However, if the costs were different, BFS would follow a different path compared to UCS.

UCS considers costs and initially explores the path with the lowest cost. However, it still does not have a direct sense of the goal (it explores paths somewhat **randomly** based on cost).

This algorithm is **complete** but not **optimal**, as it doesn't directly prioritize the goal.

Algorithm: A*: **nullHeuristic**



```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l openMaze -p SearchAgent -a fn=astar
[SearchAgent] using function astar and heuristic nullHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.1 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

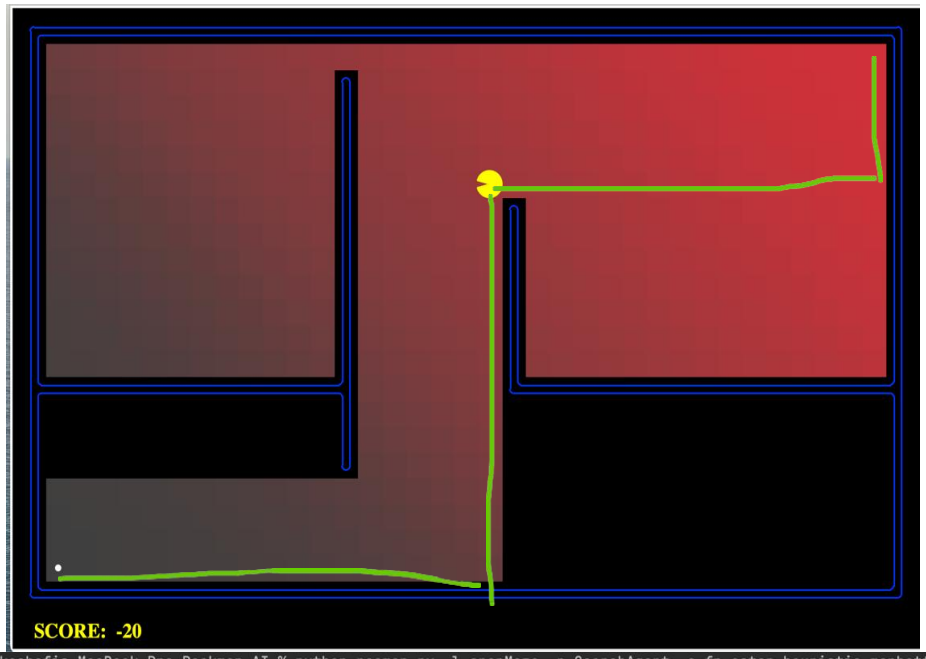
Algorithm: A*

- **nullHeuristic** is considered as having a value of **h(n) = 0**.
- The function **F(n) = g(n) + h(n)** follows the formula, where **g(n)** represents the actual cost and **h(n)** is the heuristic value.
- With **nullHeuristic**, **F(n) = g(n)**, so the algorithm essentially behaves like **UCS** or **BFS**, but this does not introduce any new differences.

In this case, we still don't have a clear goal (non-optimal).

- Additionally, when using **ManhattanDistance** as a heuristic, we are aiming for a goal that is closer in terms of the goal position.
- This method is **complete** and **optimal**

:ManhattanDistance heuristic A




```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Ending graphics raised an exception: 0
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

In the case of using ManhattanHeuristic:

If **A** has a goal state and we consider the **Manhattan Heuristic** for estimating the distance, in reality, the states could be divided into different categories. As a result, the heuristic becomes more refined in certain situations, while other situations may result in less progress. For example, states that are farther from the goal will require more steps to reach the goal, and the heuristic will guide the search accordingly.

For comparison, …

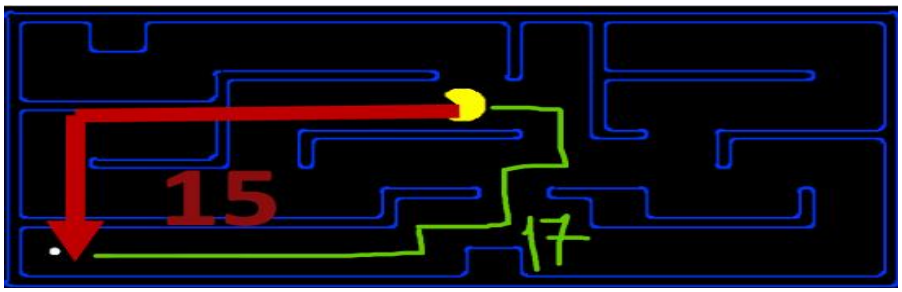
> *A (Manhattan Distance) < BFS, UCS, A* (null heuristic) < DFS: Not.
>
> *A (Manhattan Distance) < DFS < BFS, UCS, A* (null heuristic): has become worse than the number of nodes.

5% Increase in Q values for corner Q, compared to the tinyCorner state (28) versus mediumCorner state (2000)

The state space has been expanded and the ability to use A and heuristic search has been increased.

Explain the heuristic for the corner problem and justify its correctness. The heuristic used here is based on the Manhattan Distance, which estimates the cost to reach the goal by considering the grid's layout and obstacles. In this case, the heuristic provides an admissible estimate because it never overestimates the true cost, satisfying the condition h(n) ⩽ h(n) < 0. If the heuristic is zero (h(n) = 0), it reduces to UCS, while for non-zero heuristics, A* would use it to calculate the shortest path more efficiently.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 252
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:        512.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.2 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```



To optimize the configuration of the state, we define f(n) *based on the formula* f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n)*, where* g(n) *is the cost to reach node* nnn*, and* h(n) *is the heuristic function. The A** algorithm calculates f(n) and selects nodes based on the value of f(n). A smaller f(n) indicates that the node is more likely to be the best path forward.

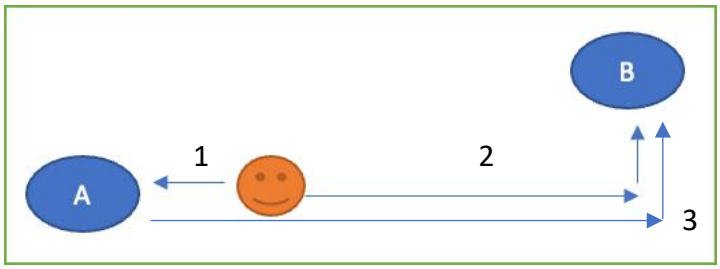
The algorithm prefers states with smaller f(n) values and explores them first, which is why A* guarantees that it finds the optimal path. If h(n) is admissible, meaning it does not overestimate the actual cost, A* will always find the shortest path.

As an example, consider two nodes, A and B. Their f(n) values are calculated as follows:

f(a)=g(a)+Manhattan(2)

f(b)=g(b)+Manhattan(2)

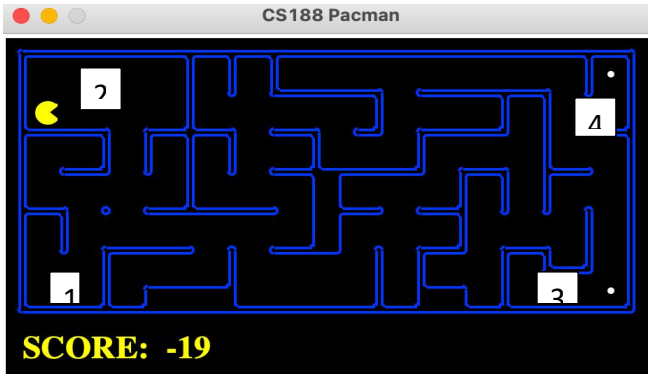
If the heuristic value for both is the same, the algorithm will explore the node with the lower g(n) value first, since that path represents a smaller cost. As the algorithm progresses, it will continue to calculate and compare the f(n) values, and the state with the lowest value will be expanded further, leading to the optimal solution.



The A* algorithm is designed in such a way that the length of its path is equal to the optimal solution.

```
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumCorners -p SearchAgent -a fn=ucs,prob=CornersProblem,heuristic=cornersHeuristic
[SearchAgent] using function ucs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.3 seconds
Search nodes expanded: 1966
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
rojina@Rojinakashefis-MacBook-Pro Packman AI % python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

The process has been completed in a specific manner.



The sequence of distances is: <4 <3 <2 <1
The sequence of nodes with the highest values is: 4 < 3 < 2 < 1
1, 2, 3, 4: f(n) becomes smaller
The sequence of exits from the front queue is: 1, 2, 3, 4 (where f(n) has a smaller value and exits faster)
Observation shows that f(n) is increasing as 1, 2, 3, 4 because f(n) is minimized, which means it exits the queue earlier and thus this is efficient.

7. Eating all the nodes Q

And proving the construction of it:

The cost of the maze is defined as the distance between the points. BFS is used to find the shortest path between the start point and the goal. Nodes are evaluated based on the distance from the start point, using the BFS algorithm. The maze distance is calculated using Manhattan distance, and it provides an optimal solution. The reasoning behind it is that the BFS search explores all nodes, and at each point, the cost is updated.

For example, if Pacman moves in the maze, the distance between two points will be calculated, and the path length increases. As the algorithm progresses, the evaluation function f(n) grows, and the system optimizes the maze traversal by calculating the total cost.

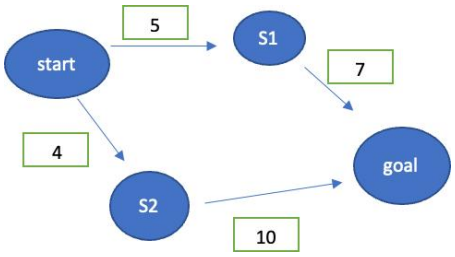
Furthermore, at each stage, the BFS search explores all possible points, updating the costs accordingly.

Search for the nearest dot with the closestDotSearchAgent. It always finds the shortest possible path in the maze. However, it will not work if the agent cannot find a way. The method is implemented as part of an agent's strategy, and this agent performs a search operation to minimize the path to the goal, similar to an example where the agent moves to a nearby point while following the optimal path.

A* and UCS are used for these searches. In this case, the BFS algorithm is employed, as it is effective for such tasks.

The agent will perform an action and not necessarily predict what will happen in the future. The greedy search method will prioritize the next step based on the nearest point.

For example, if a point is 4 or 5 steps away, it might favor the direction with fewer steps in the future. Greedy methods may also have variations based on distance calculations, and the agent always picks the path that seems to lead to the goal most directly at the moment.



Let's assume that the agent is required to move in the direction of the right. The pacman will follow the right side path in the maze, which has many turns and increases the complexity of the search. If the A* algorithm is used, the agent will move toward the right side and continue to explore in that direction. This will lead to an increase in the search cost, and the agent might not find the optimal solution in this scenario. However, when the algorithm's search method is applied, the agent may adjust its path based on real-time decisions.