CS7IS2: Artificial Intelligence Assignment 1

Submitted By

Pallavit Aggarwal - MSc Intelligent Systems
Student Number - 22333721
Email ID - aggarwpa@tcd.ie

The following are the contents of this document

I. Introduction:

One of the fundamental aspects of AI is its ability to find optimal solutions to complex problems. However, finding the best solution to a problem is not always easy, especially when the search space is vast and the solution space is not well-defined. This is where search problems in AI come into play.

Search problems in AI allow us to find optimal solutions to complex problems efficiently. These problems can be categorized into two types:

i) ***Uninformed Search***: Uninformed search algorithms do not have any knowledge about the problem domain. They rely solely on the information provided by the problem representation, such as the initial state, the goal state, and the available actions. Uninformed search algorithms explore the search space systematically, without any bias towards any particular path or solution. Examples of uninformed search algorithms include Breadth-First Search (BFS), Depth-First Search (DFS) etc.

ii) ***Informed Search***: In contrast, informed search algorithms, also known as heuristic search algorithms, use domain-specific information to guide the search towards the goal state more efficiently. This information is provided by a heuristic function, which estimates the distance or cost from the current state to the goal state. Informed search algorithms use this heuristic function to prioritize the nodes that are most likely to lead to the goal state, thus reducing the search space and making the search more efficient. An Example of informed search algorithms is A* algorithm.

iii) In addition to search problems, ***Markov Decision Processes*** (MDPs) are another essential component of AI. MDPs are a mathematical framework used to model decision-making problems where the outcome of each decision is uncertain. Search algorithms are used to find the optimal policy that maximizes the expected cumulative reward over time in MDPs. Examples of search algorithms used to solve MDPs include Value Iteration, Policy Iteration

In this assignment, we implement BFS, DFS as uninformed search algorithms to solve a maze using a start state and goal state, improve upon it using A* and a heuristic and finally explore MDP and implement Value Iteration and Policy Iteration algorithms.

II. Uninformed Search

BFS (Breadth-First Search)

The BFS algorithm works by starting at a vertex, marking it as visited, and exploring all its neighbours. Then, it moves on to the neighbours of the neighbours, marking them as visited and exploring their neighbours, and so on until all vertices in the graph have been visited.

Here is a generic psuedocode for BFS:

BFS(G, s):
1. Initialize a queue Q and mark vertex s as visited
2. Add s to the back of the queue Q
3. While Q is not empty:
4. Remove the vertex at the front of Q and call it v
5. For each neighbour u of v that has not been visited:
6. Mark u as visited
7. Add u to the back of Q

In this algorithm, G represents the graph, and s represents the starting vertex. The algorithm uses a queue data structure to keep track of the vertices to be explored, and a visited array to keep track of the visited vertices.

The time complexity of BFS is O(|V|+|E|), where |V| is the number of vertices and |E| is the number of edges in the graph. The space complexity of BFS is also O(|V|), since the visited array and queue both require space proportional to the number of vertices.

```python
def breadthFirstSearch(problem):
    "*** YOUR CODE HERE ***"
    # initialize the frontier with the starting state
    startState = problem.getStartState()
    frontier = util.Queue()
    frontier.push((startState, []))
    # initialize an empty set to keep track of visited states
    visited = set()

    while not frontier.isEmpty():
        # remove a state from the frontier
        state, actions = frontier.pop()
        # if the state has already been visited, skip it
        if state in visited:
            continue
        # mark the state as visited
        visited.add(state)
        # if the state is the goal state, return the actions to reach it
        if problem.isGoalState(state):
            return actions
        # add the next states to the frontier
        for next_state, action, cost in problem.getSuccessors(state):
            frontier.push((next_state, actions + [action]))

    # if no goal state is found, return an empty list of actions
    return []
    util.raiseNotDefined()
```

In this implementation, the breadthFirstSearch function takes a problem object as input, which represents the search problem in the Pacman game world. The function returns a list of actions that lead to the goal state.

The algorithm starts by initializing the frontier with the starting state and an empty list of actions. It also initializes an empty set visited to keep track of visited states. The main loop of the algorithm runs as long as the frontier is not empty. In each iteration, it removes a state from the frontier and checks if it has already been visited. If it has, the state is skipped. Otherwise, the state is marked as visited and checked if it is the goal state. If it is, the actions that lead to it are returned.

If the current state is not the goal state, the algorithm gets the next possible states from the problem object using the getSuccessors method, which returns a list of (next_state, action, cost) tuples. For each (next_state, action, cost) tuple, the algorithm adds the (next_state, actions + [action]) pair to the frontier, where actions + [action] is the list of actions that led to the current state plus the action that leads to the next state.

If no goal state is found, an empty list of actions is returned.

The time complexity of BFS is O(b^d), where b is the branching factor of the tree and d is the depth of the goal node. In the Pacman game world, the branching factor can be quite large, so BFS may not be practical for very deep search problems. However, it is guaranteed to find the shortest path to the goal state in an unweighted graph.

DFS (Depth-First Search)

DFS (Depth First Search) is a fundamental graph traversal algorithm used to explore a graph by visiting every vertex in the graph. The algorithm starts from a particular vertex in the graph, known as the start vertex, and then traverses as far as possible along each branch before backtracking. It works by maintaining a stack of vertices to visit, and for each vertex, it explores its adjacent vertices, marking them as visited to avoid revisiting them.

Here is a generic psuedocode for DFS:

```
1. DFS(G, start_vertex):
2.   for each vertex v in G:
3.       mark v as unvisited
4.   S = create empty stack
5.    push start_vertex onto S
6.    while S is not empty:
7.      current_vertex = pop from S
8.       if current_vertex is unvisited:
9.          mark current_vertex as visited
10.         for each neighbor v of current_vertex:
11.             push v onto S
```

The time complexity of Depth-First Search (DFS) algorithm is O(V+E), where V is the number of vertices and E is the number of edges in the graph. This is because in the worst-case scenario, the algorithm will visit every vertex and every edge once.

It is worth noting that the time complexity of DFS can be affected by the data structure used to represent the graph. For example, if the graph is represented as an adjacency matrix, the time complexity of checking whether an edge exists between two vertices is O(1), but the space

complexity of the matrix is O(V^2), which can be inefficient for large graphs. On the other hand, if the graph is represented as an adjacency list, the time complexity of checking whether an edge exists is O(log E) (assuming a balanced binary search tree is used), but the space complexity is only O(V+E), which is more efficient for sparse graphs.

The code implemented above with depth-first search (DFS) algorithm searches for a goal state in a problem's state space. The algorithm starts at the problem's initial state, explores all possible paths along each branch of the state space, backtracking when a dead-end is reached, until the goal state is found or all nodes have been explored. The algorithm uses a stack data structure to keep track of nodes to be visited, and a set to keep track of visited nodes, to avoid revisiting nodes and going into an infinite loop.

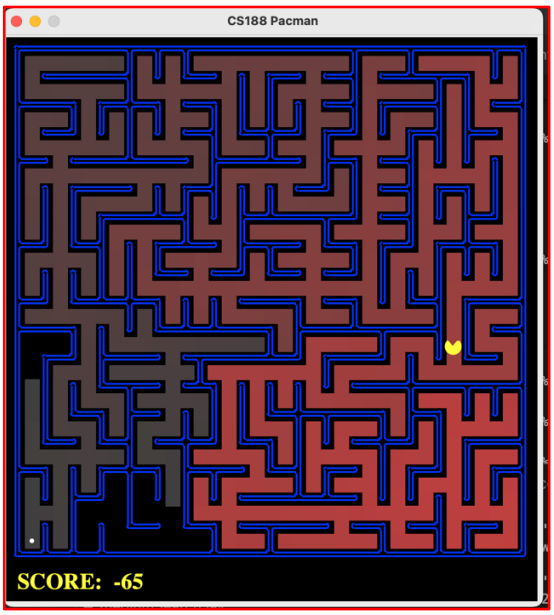
At each iteration, the code pops the next state and its path from the stack, checks if it has already been visited, and marks it as visited. If the state is the goal state, the code returns the path taken to reach it and some metrics. Otherwise, the code expands the state by obtaining its successors, increments the counter for the number of nodes expanded, checks for dead-ends, and pushes each successor's state and path onto the stack. The code updates the maximum memory used, and if the stack is empty and no goal state is found, it returns an empty list indicating that no path was found.

```
# initialize the frontier as a stack with the start state
start_state = problem.getStartState()
frontier = util.Stack()
frontier.push((start_state, []))

# initialize the explored set as an empty set
explored = set()

# loop until the frontier is empty
while not frontier.isEmpty():
    # get the next node from the frontier
    node, actions = frontier.pop()

    # check if the node is the goal state
    if problem.isGoalState(node):
        return actions

    # add the node to the explored set
    explored.add(node)

    # expand the node by generating its child nodes
    for child_node, action, cost in problem.getSuccessors(node):
        # check if the child node has not been explored or in the frontier
        if child_node not in explored and not frontier.contains(child_node):
            # add the child node and its action sequence to the frontier
            frontier.push((child_node, actions + [action]))

# if the goal state is not found, return None
return None
"*** YOUR CODE HERE ***"
```

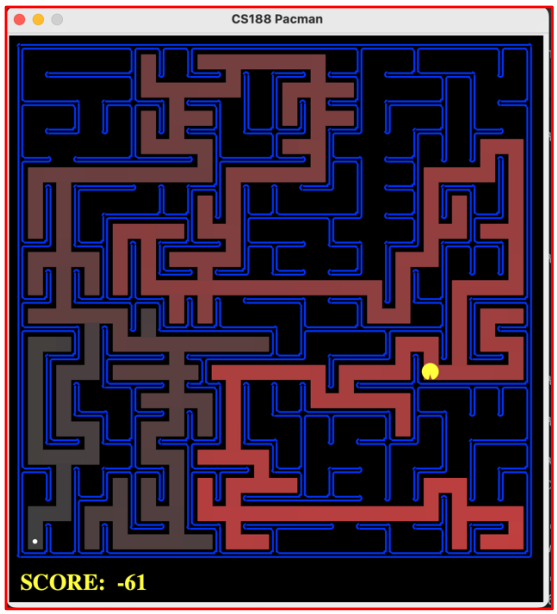
III. Performance Evaluation of DFS and BFS

Both BFS and DFS are judged against 3 different Layouts. The first layout is BigMaze default layout available in the Berkley Project. The other two are variations of BigMaze where in the second one a newer shorter route is available, and the third one has a bigger open area to stress test the algorithm in getting stuck around that area and exploring it endlessly.

i. BigLayout-1 (*Default*)



BFS for BigMaze Layout 1



DFS for BigMaze Layout 1

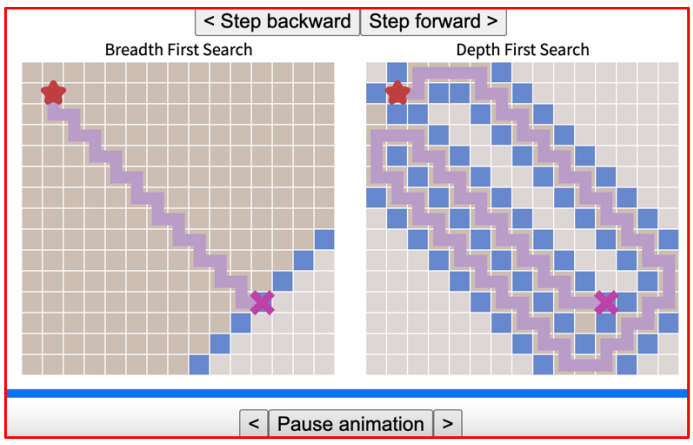
The output from the sys logs is as follows

	BFS	DFS
Path Length	210	210
Search Nodes Expanded	1240	390
Time Taken	0.028 seconds	0.019 seconds
Memory Used	5.3MB	5.3MB
Number of dead-ends encountered	0	40

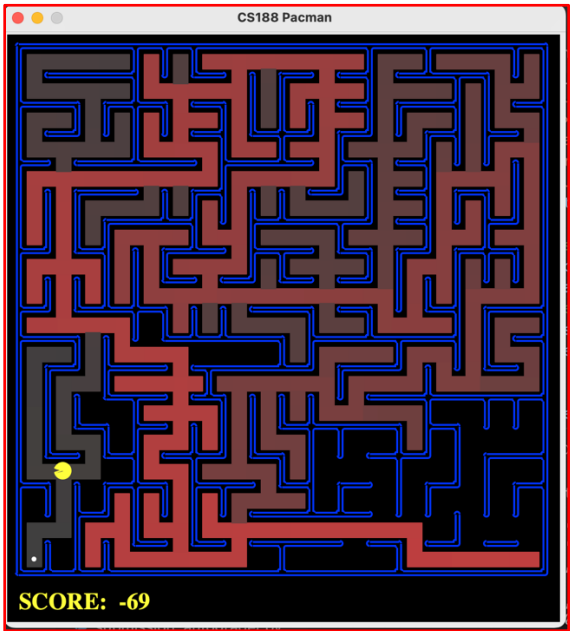
We observe that both take the same path length, but BFS expands more nodes and searches a broader path than DFS. DFS is able to find a path faster than BFS but DFS also encounters a lot of dead-ends as compared to BFS.

Note: A dead end is being calculated as a state from which no further actions are possible and no successors to the state are available. It is given by the following condition in the codebase if len(problem.getSuccessors(current_state)) == 0

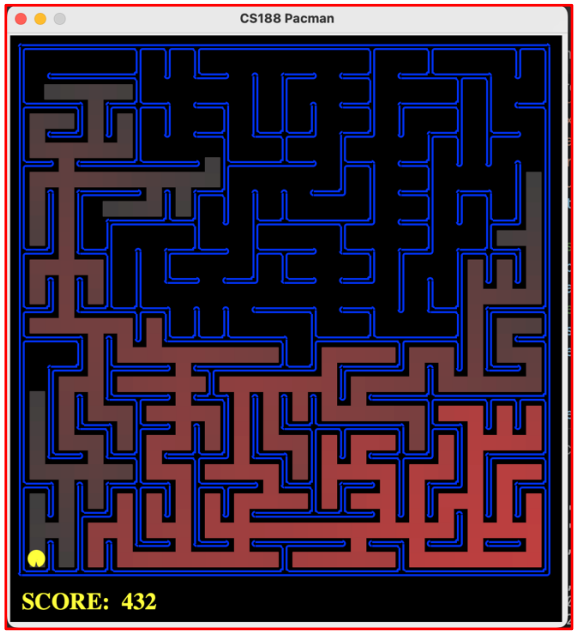
From the lecture slides, the following tutorial helps in visualizing this search node expansion between BFS and DFs much better (https://cs.stanford.edu/people/abisee/tutorial/)



ii. BigLayout-2 (Shorter path also available)



BFS for BigMaze Layout 2



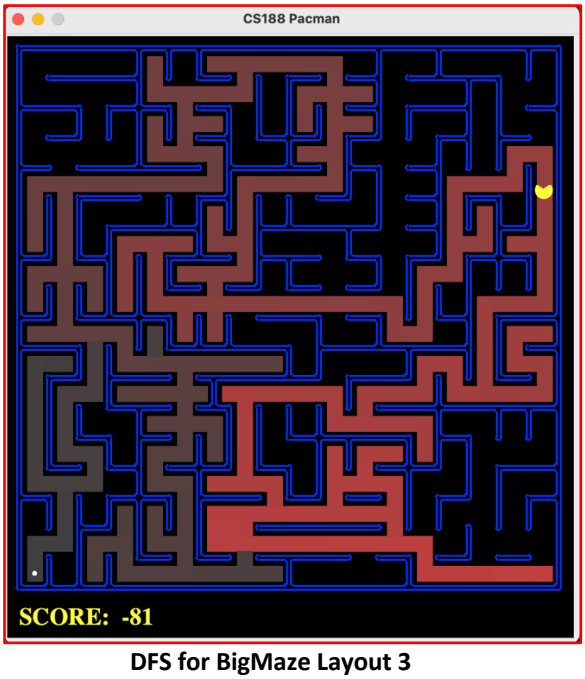
DFS for BigMaze Layout 2

	BFS	DFS
Path Length	78	78
Search Nodes Expanded	764	553
Time Taken	0.017 seconds	0.019 seconds
Memory Used	5.3MB	5.3MB
Number of dead-ends encountered	0	72

We observe that both take the same path length, but BFS expands more nodes and searches a broader path than DFS. In this case the BFS is able to find a path slightly faster than DFS.

iii. BigLayout-3 (Shorter path and open space to stress test against cycles)

BFS for BigMaze Layout 3

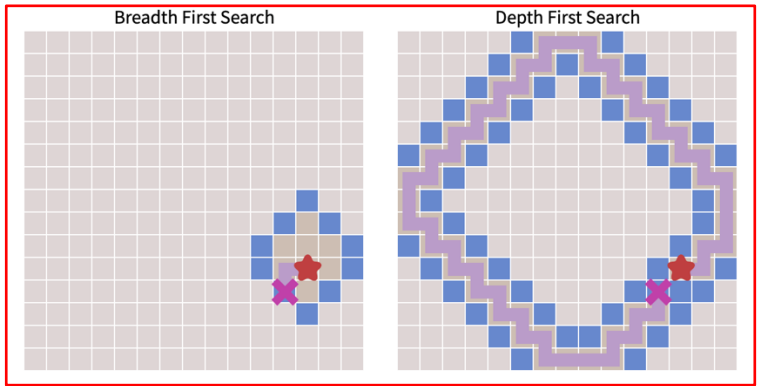


DFS for BigMaze Layout 3

	BFS	DFS
Path Length	78	210
Search Nodes Expanded	784	405
Time Taken	0.019 seconds	0.015 seconds
Memory Used	5.3MB	5.3MB
Number of dead-ends encountered	0	37

In the stress test we observe something different, DFS finds the longer path whereas BFS is able to find the shorter more optimal path. BFS expands more nodes than DFS and the reddening around the paths show how many times the nodes have been explored.

This can also be observed from the animation tutorial, where-in if we place the start state and goal state right next to each other BFS finds the optimal path whereas DFS takes another longer route altogether.



IV. Informed Search

A Search*

A* is considered an extension of Dijkstra's algorithm with the addition of a heuristic function, which estimates the distance between a given node and the goal node. The heuristic function is used to guide the search process towards the goal node, leading to a faster and more efficient search.

Here is a generic psuedocode for A*:
1. Initialize the starting node and the open list with the starting node
2. Initialize the closed list as empty
3. While the open list is not empty:
    4. Find the node in the open list with the lowest f score (f = g + h, where g is the cost to reach the current node and h is the heuristic function)
    5. If the current node is the goal node, return the path
    6. Move the current node from the open list to the closed list
    7. For each neighbour of the current node:
        8. Calculate the tentative g score (the cost to move from the current node to the neighbour node)
        9. If the neighbour is already in the closed list or the tentative g score is greater than the neighbour's g score, continue to the next neighbour
    10. If the neighbour is not in the open list or the tentative g score is less than the neighbour's g score:

11. Set the neighbor's g score to the tentative g score
12. Calculate the neighbor's h score (using the heuristic function)
13. Set the neighbor's parent to the current node
14. If the neighbor is not in the open list, add it to the open list

15. Return "no path" if the open list is empty

```
def aStarSearch(problem, heuristic=nullHeuristic):
    startState = problem.getStartState()
    frontier = util.PriorityQueue()
    frontier.push((startState, []), heuristic(startState, problem))
    explored = set()

    while not frontier.isEmpty():
        node, actions = frontier.pop()

        if problem.isGoalState(node):
            return actions

        if node not in explored:
            explored.add(node)
            successors = problem.getSuccessors(node)
            for successor, action, cost in successors:
                newActions = actions + [action]
                newCost = problem.getCostOfActions(newActions) + heuristic(successor, problem)
                frontier.push((successor, newActions), newCost)

    return []
    util.raiseNotDefined()
```

The implementation above starts by initializing a priority queue called frontier with the start state of the problem, an empty action list, and a cost of 0. The frontier priority queue is ordered by the sum of the cost to reach a node and a heuristic function called heuristic. The explored set is used to keep track of the nodes that have already been explored.

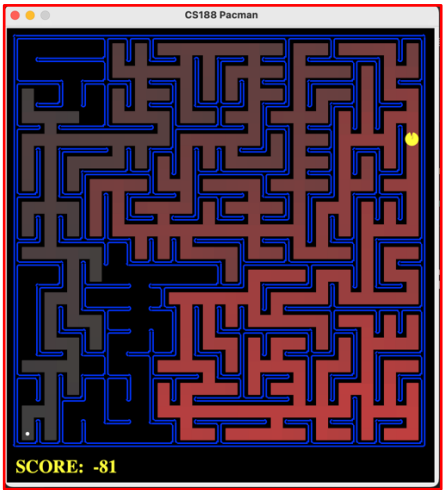
The algorithm then enters a loop where it pops the node with the lowest combined cost and heuristic estimate from the frontier queue. If the popped node is the goal state, the algorithm returns the action list to reach it, along with the statistics. Otherwise, it adds the popped node to the explored set, generates its successors, and adds them to the frontier queue if they have not been explored before. If a successor node has already been explored, it is considered a dead end, and the algorithm increments the num_dead_ends counter.

Finally, if the frontier queue is empty, the algorithm returns an empty action list, indicating that no path to the goal state was found.

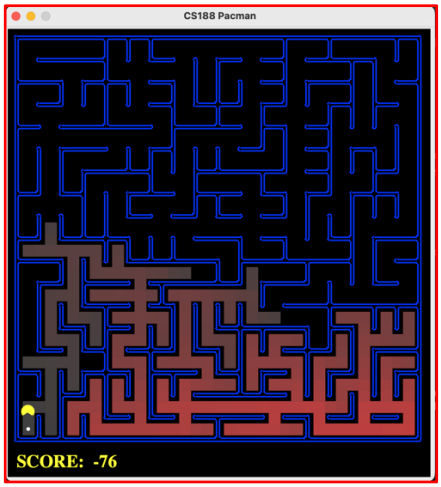
V. Performance evaluation of A*

i. Using Manhattan Distance Heuristic

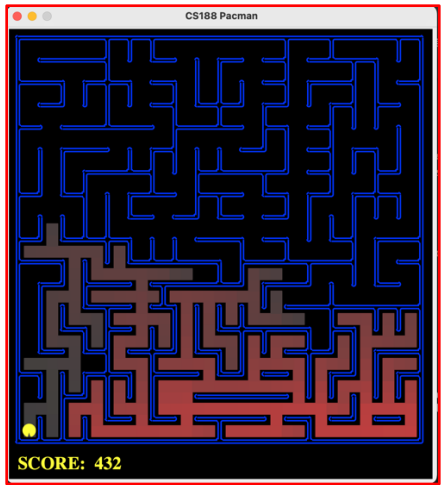
	Layout 1 (default)	Layout 2 (2 paths)	Layout 3 (stress test)
Time Taken	0.02 seconds	0.009	0.01
Memory Used	50 MB	50 MB	50 MB
Path Length	210	78	78
Dead Ends Encountered	548	249	265
Search Nodes Expanded	549	248	258



BigMaze Layout 1

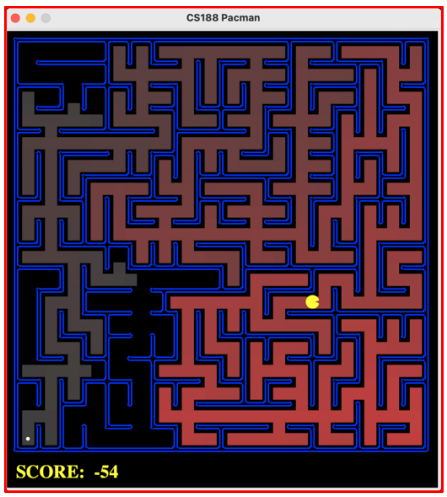


BigMaze Layout 2

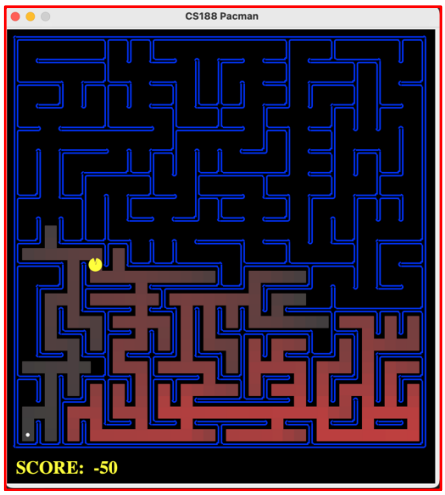


BigMaze Layout3

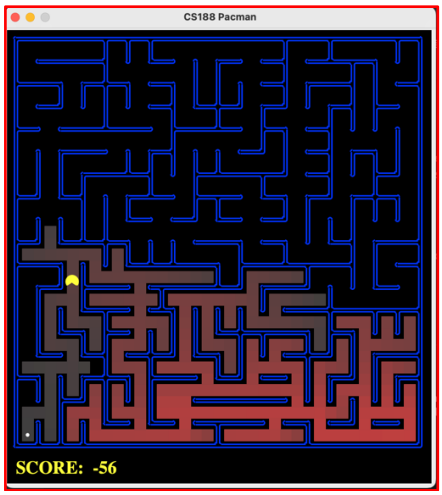
ii. **Using Euclidian Distance Heuristic**



BigMaze Layout 1



BigMaze Layout 2



BigMaze Layout 3

	Layout 1 (default)	Layout 2 (2 paths)	Layout 3 (stress test)
Time Taken	0.02 seconds	0.013	0.01
Memory Used	50 MB	50 MB	50 MB
Path Length	210	78	78
Dead Ends Encountered	556	264	278
Search Nodes Expanded	557	263	271

A* search has to completely explore the grid in the first case when there's only one correct path, but quickly tracks the optimal path and finds optimal path with almost 50% reduction in search nodes when there's an alternate shorter path available. Manhattan heuristic performs better than Euclidian heuristic.

VI. Markov Decision Process (MDP)

Value Iteration

Value iteration finds an optimal policy, which specifies the action to take in each state to maximize the expected cumulative reward over time. Once the optimal policy is found, the optimal path can be obtained by starting in the initial state and following the actions specified by the policy until the goal state is reached. The optimal policy is computed by iteratively updating the values of the states until they converge to their optimal values, which specify the expected reward for following the optimal policy from each state.

Here is a generic pseudocode for Value Iteration:

1. Initialize the value of each state to zero (or some other arbitrary value).

2. For each state, compute the expected value of each action, which is the sum of the rewards for taking that action plus the discounted value of the next state. The discount factor accounts for the fact that future rewards are worth less than immediate rewards.

3. Update the value of each state to be the maximum expected value over all possible actions.

4. Repeat steps 2 and 3 until the values of the states converge to a stable policy.

5. Once the optimal value function is computed, the optimal policy can be obtained by taking the action with the highest expected value for each state.

6. The optimal path can be obtained by starting at the initial state and following the actions specified by the optimal policy until the goal state is reached.

```
num_iterations = 0
for i in range(self.iterations):
    # Initialize the dictionary of new values
    newValues = util.Counter()

    # Compute the new value of each state
    for state in self.mdp.getStates():
        # Find the action with the highest Q-value
        actions = self.mdp.getPossibleActions(state)
        if not actions:
            continue
        maxQValue = float('-inf')
        for action in actions:
            qValue = self.getQValue(state, action)
            if qValue > maxQValue:
                maxQValue = qValue

        # Update the value of the state
        newValues[state] = maxQValue

    if self.has_converged(self.values, newValues):
        num_iterations = i + 1
        print("Convergence reached after", num_iterations, "iterations!")
        break
    # Update the value function with the new values
    self.values = newValues
    num_iterations+=1
```

```
qValue = 0
for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action):
    reward = self.mdp.getReward(state, action, nextState)
    qValue += prob * (reward + self.discount * self.values[nextState])
return qValue
```

The ValueIterationAgent class is a subclass of ValueEstimationAgent and is used to solve MDP problems using the value iteration algorithm. Here's what each method in the class does:

__init__(self, mdp, discount=0.9, iterations=100): This is the constructor method for the class. It initializes the agent with the MDP to be solved, the discount factor, and the number of iterations to perform during value iteration. It also initializes the values dictionary to contain the initial values of each state.

getValue(self, state): This method returns the estimated value of a given state using the values dictionary.

getQValue(self, state, action): This method returns the estimated Q-value of a given state-action pair using the values dictionary.

getPolicy(self, state): This method returns the optimal policy for a given state by selecting the action with the highest Q-value using the getQValue method.

getAction(self, state): This method returns the optimal action for a given state by selecting the action with the highest Q-value using the getQValue method.

valueIteration(self): This method performs the value iteration algorithm for the given number of iterations, updating the values dictionary for each state using the update method.

VII. Performance evaluation of Value Iteration Agent

I run the value Value Iteration Agent for 55 iterations for each Grid of different size and reward distribution for 10 episodes each. The living reward is set to -0.1 and the noise or randomness in action is 0.2 which means it only chooses the right action 80% of the times.

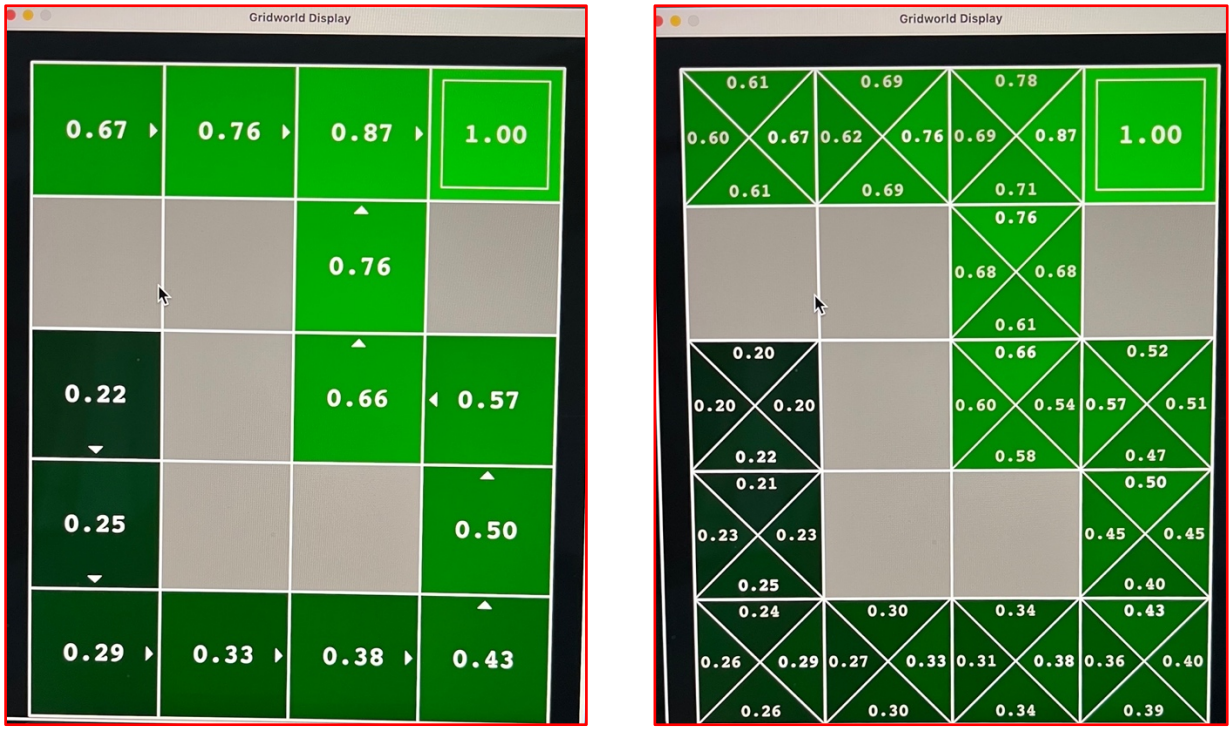
The 3 Grids are as follows:

Grid 1 (Default) :
```
grid = [[' ',' ',' ',+1],
        ['#','#',' ','#'],
        [' ','#',' ',' '],
        [' ','#','#',' '],
        ['S',' ',' ',' ']]
```
Grid 2 (Rewards are different) :
```
grid = [[' ',' ',' ',+1,' '],
        ['#','#',' ','#', ' '],
        [+2,'#',' ',-2, ' '],
        [' ','#','#',' ', ' '],
        [' ',' ',' ','S', ' ']]
```

Grid 3 (Bigger, more maze like to find optimal path and compare with others)
```
grid = [['#', ' ', ' ', '#', '#', '#', '#', ' ', ' ', +1],
   ['S', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', -2, ' '],
   ['#', '#', '#', ' ', '#', '#', '#', '#', '#', ' '],
   [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
   ['#', '#', '#', ' ', '#', '#', ' ', '#', '#', ' '],
```

[‘ ‘, ‘#’, ‘ ‘, -2, ‘#’, ‘ ‘, +1, ‘ ‘, ‘ ‘, ‘ ‘],
[‘.’, ‘#’, ‘ ‘, ‘ ‘, ‘ ‘, ‘ ‘, ‘ ‘, ‘ ‘, ‘ ‘, ‘ ‘, +1]]

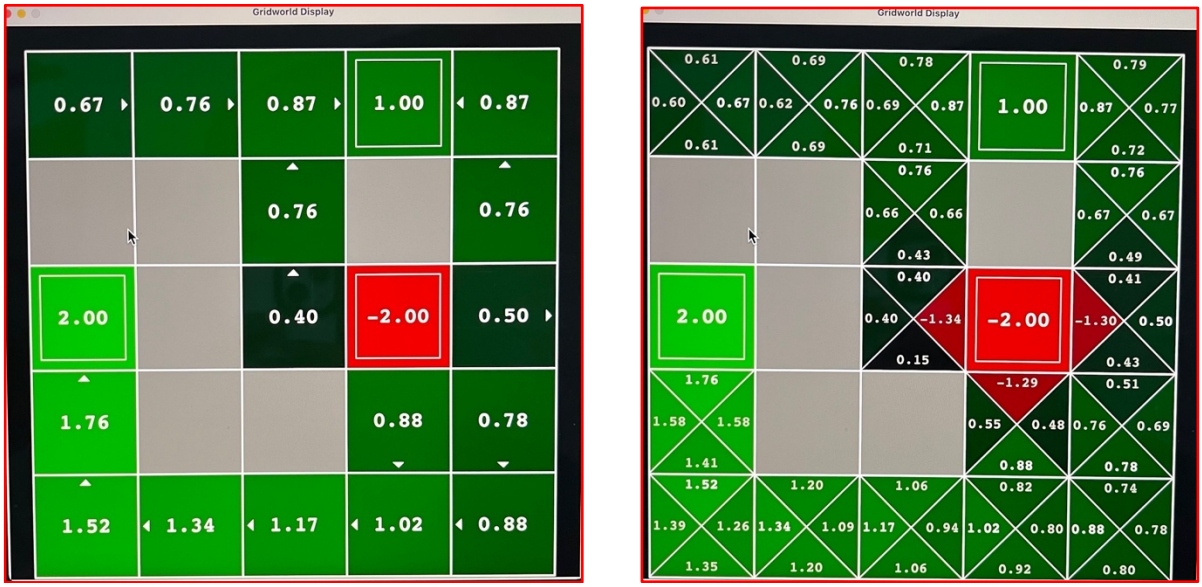
GRID 1 Results:



iterations=55, episodes=10, living cost=-0.1, noise=0.2
Average Rewards accumulated: 0.29814102573395507
Convergence reached after 30 iterations.

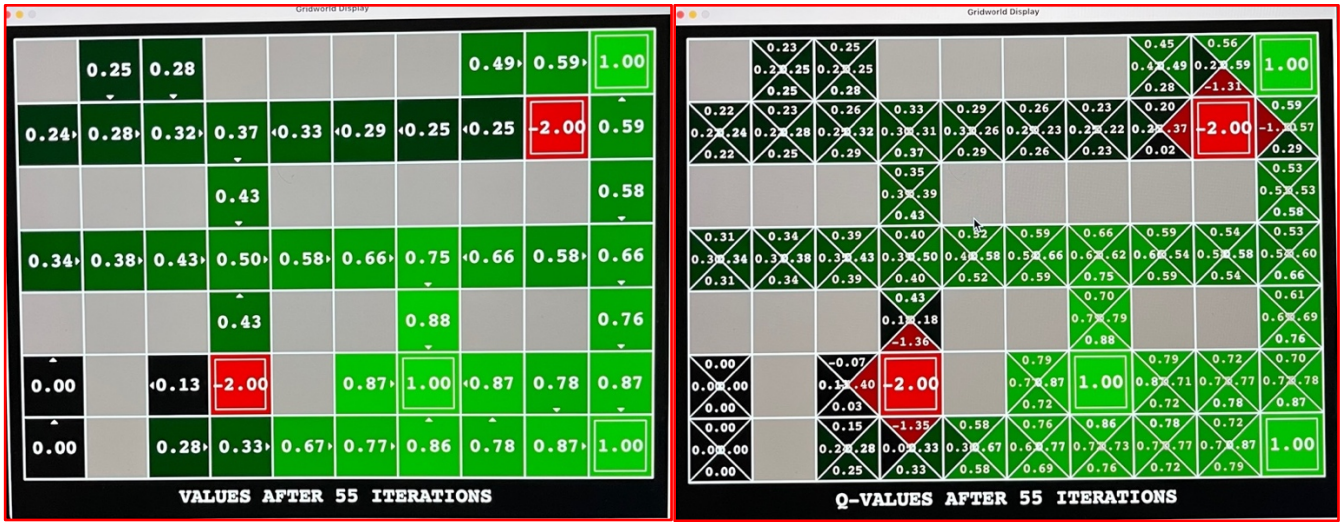
(*Note: These are clicked by phone camera because pressing any other button would change the display of this grid)*

GRID 2 Results:



iterations=55, episodes=10, living cost=-0.1, noise=0.2
Average Rewards accumulated: 1.0557358900200005
Convergence reached after 37 iterations.

GRID 3 Results:



iterations=55, episodes=10, living cost=-0.1, noise=0.2
Average Rewards accumulated: 0.19158531602889894
Convergence reached after 48 iterations.

(*Note: These are clicked by phone camera)*

VIII. Markov Decision Process (MDP) - II

Policy Iteration

Policy iteration is an iterative algorithm that involves two main steps: policy evaluation and policy improvement. In policy evaluation, the value function for a given policy is computed by iteratively applying the Bellman equation until convergence. In policy improvement, a new policy is derived by greedily selecting actions that maximize the expected cumulative reward based on the current value function. The process then repeats until the policy converges to the optimal policy.

It is a model-based reinforcement learning algorithm that learns the optimal policy by iteratively evaluating and improving the policy. It requires knowledge of the transition probabilities and rewards of the environment to compute the value function and derive the optimal policy.

Here is a generic pseudocode for Policy Iteration:

1. Initialize the policy π
2. Repeat the following steps until convergence:
   a. Policy Evaluation:
      i. Given policy π, compute the state-value function Vπ
      ii. Update Vπ for all states using the Bellman equation:
         $V(s) = \sum_a \pi(a|s) \sum_{s', r} P(s', r|s,a) [r + \gamma V(s')]$

   b. Policy Improvement:
      i. For each state s, compute the action-value function Qπ(s,a) for all actions a
      ii. Update the policy for each state using the greedy policy:
         $\pi(s) = \text{argmax}_a Q\pi(s,a)$

3. Return the optimal policy π*

In this pseudocode, π represents the policy being evaluated and improved, Vπ represents the state-value function for policy π, Qπ(s,a) represents the action-value function for state s and action a, P(s',r|s,a) represents the probability of transitioning to state s' and receiving reward r when taking action a in state s, and γ represents the discount factor for future rewards.

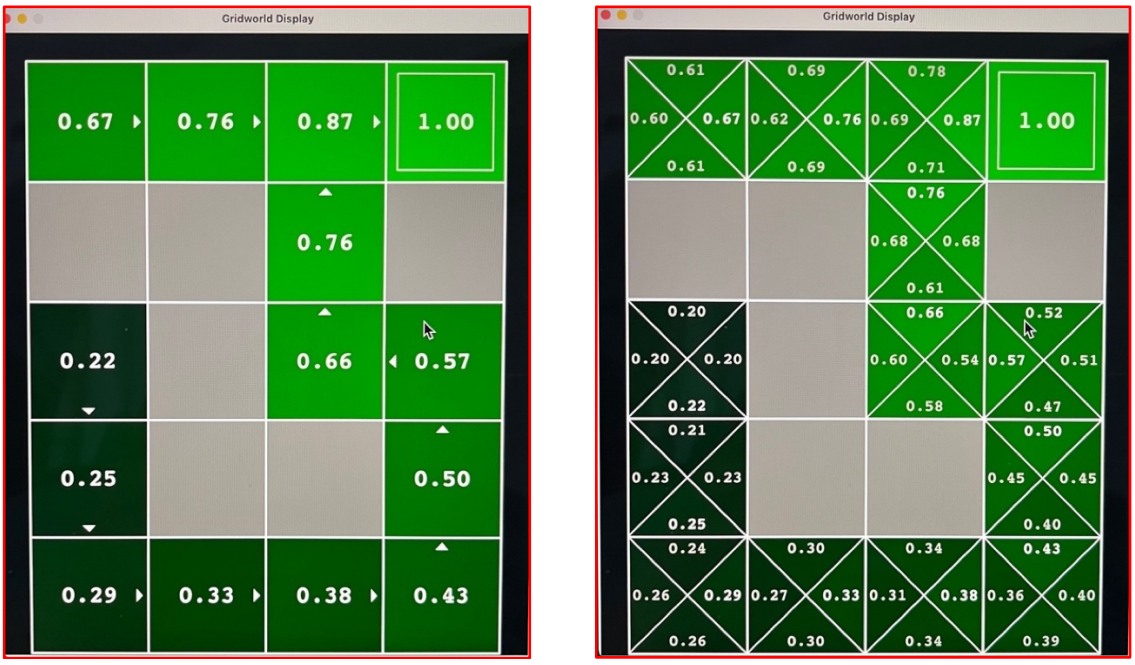
The algorithm consists of two main steps: policy evaluation and policy improvement. In the policy evaluation step, we use the current policy π to compute the state-value function Vπ using the Bellman equation. In the policy improvement step, we compute the action-value function Qπ(s,a) for all actions a in each state s, and update the policy for each state to choose the action with the highest action-value.

We repeat these steps until convergence, where convergence is defined as the policy no longer changing. Finally, we return the optimal policy π*.

The computePolicy and getQValue in code are the main functions implementing the above algorithm in an iterative manner as opposed to the general recursive implementations.

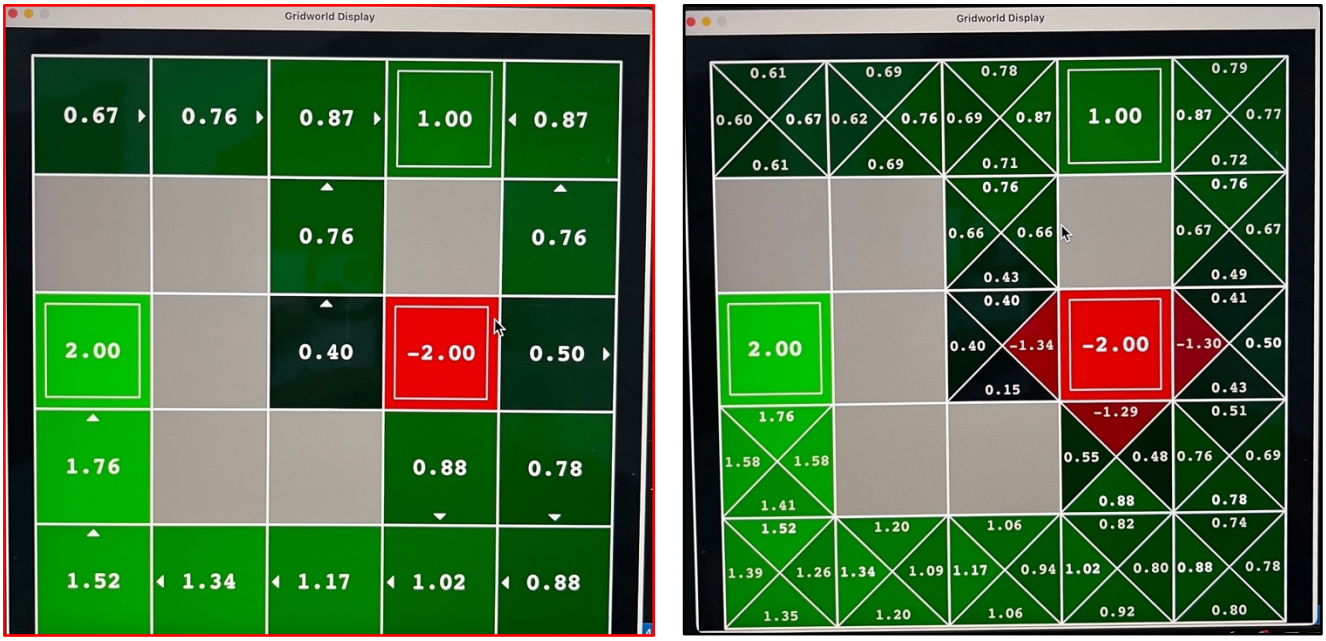
```
def getQValue(self, state, action):
    qvalue = 0
    # Check if the action is one of the possible actions for the given state
    if action in self.mdp.getPossibleActions(state):
        # Iterate over all possible next states and their corresponding transition probabilities
        for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            # Calculate the Q-value using the Bellman equation
            qvalue += prob * (self.mdp.getReward(state, action, nextState) + self.gamma * self.values[nextState])
    return qvalue
```

IX. Performance evaluation of Policy Iteration Agent



iterations=55, episodes=10, living cost= -0.1, noise=0.2
Average Rewards accumulated: 0.26402015554618125
Convergence reached after 4 iterations.

(*Note: These are clicked by phone camera)*



iterations=55, episodes=10, living cost= -0.1, noise=0.2
Average Rewards accumulated: 1.04476458582
Convergence reached after 6 iterations.

(*Note: These are clicked by phone camera)*

iterations=55, episodes=10, living cost= -0.1, noise=0.2
Average Rewards accumulated: 0.24156317189435858
Convergence reached after 6 iterations.

(*Note: These are clicked by phone camera)*

X. Comparing Policy Iteration and Value Iteration

	Policy Iteration	Value Iteration
Grid -1 Average Rewards	0.264	0.298
Grid -1 Convergence Point	4 iterations	30 iterations
Grid -2 Average Rewards	1.044	1.055
Grid -2 Convergence Point	6 iterations	37 iterations

Grid -3Average Rewards	0.241	0.191
Grid -3 Convergence Point	6 iterations	48 iterations

Policy iteration algorithm converges faster than value iteration algorithm because it uses a two-step approach to finding the optimal policy, which makes better use of the available information in the environment.

In policy iteration, the algorithm alternates between two steps: policy evaluation and policy improvement. In the policy evaluation step, the algorithm computes the value function for the current policy, which is the expected total reward that the agent can obtain by following the current policy starting from each state. In the policy improvement step, the algorithm updates the policy by selecting the action that maximizes the expected total reward from the current state, using the value function computed in the policy evaluation step.

This two-step approach is more efficient than the approach used in value iteration algorithm, which updates the value function directly in each iteration without explicitly computing the policy. By explicitly computing the policy in the policy improvement step, policy iteration can make more informed decisions about which actions to take in each state, which can lead to faster convergence.

Another reason why policy iteration can converge faster is that it avoids the potential oscillations that can occur in value iteration. Value iteration algorithm can sometimes oscillate between two or more value functions that are close in value but not identical, which can slow down convergence. Policy iteration avoids this problem by updating the policy directly based on the current value function, rather than updating the value function directly.

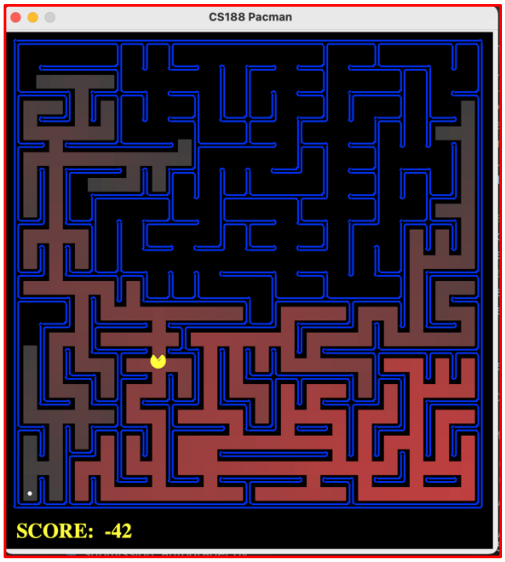
Having said that, the choice between value iteration and policy iteration depends on the specific characteristics of the environment and the computational resources available. Policy iteration is generally preferred when the state space is large which wasn't the case here, but it did because the transition probabilities and rewards of the environment are well-defined and easily computable.

XI. Comparison of BFS, DFS, A*, and Value and Policy Iteration together meaningfully

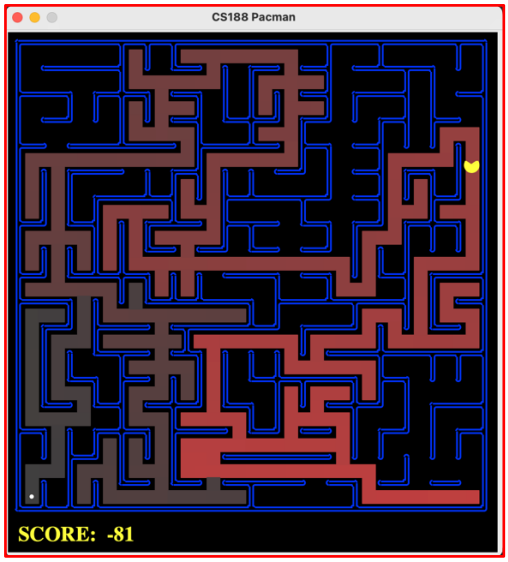
There are multiple metrics which can be used to evaluate BFS and DFS and even A* like
a) Execution Time – the time taken by the algorithm to find an optimal solution. b) nodes expanded – which is the value of nodes the algorithm needs to expand to find the optimal solution, c) Memory Usage , d) Optimality as in the path length and finally e) Robustness which is the total number of dead-ends or actions with no successors encountered.

I choose Path length as a metric for optimality, Total number of dead-ends to judge robustness, time taken as another value to distinguish these algorithm and finally search nodes expanded and I consider the BigMaze layout 3 gride.

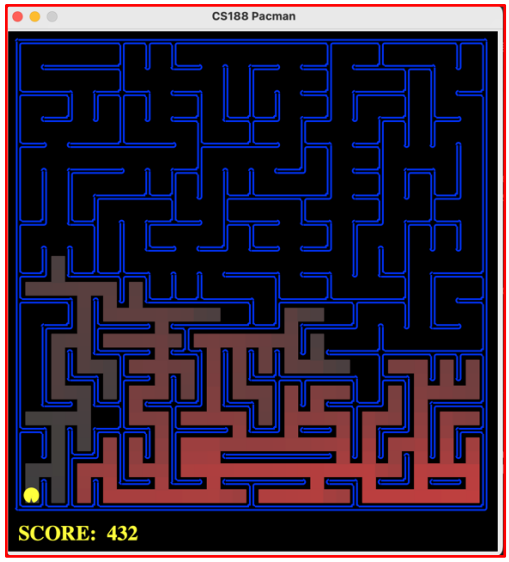
	BFS	DFS	A*
Path Length	78	210	78
Time Taken	0.019 seconds	0.015 seconds	0.010 seconds
Nodes Expanded	784	405	258
Dead-ends	0	37	265



BFS



DFS



A*

The above images show the maze with node search intensity colored in Red that DFS (middle image) explores a longer deeper route. **A* explores lesser nodes and reaches the optimal solution the fastest.**

Cautionary note:
In practice, the choice between any algorithm would depend on the size of the state space, the complexity of the environment, and the available computational resources. The implementation style of the algorithm and the environment of the problem state and the resources would also make a big difference.

Now for MDP algorithms, they find the optimal path by finding the optimal policy or the next best state based on an action and a rough estimate of the rest of the states. This pre-existing knowledge of states is defined by the bellman optimality equation.

A careful comparison of policy iteration and value iteration algorithms under the same environments and reward structures above, showed that policy iteration algorithm consistently performed better

	Policy Iteration	Value Iteration
Grid -1 Average Rewards	0.264	0.298
Grid -1 Convergence Point	4 iterations	30 iterations
Grid -2 Average Rewards	1.044	1.055
Grid -2 Convergence Point	6 iterations	37 iterations
Grid -3Average Rewards	0.241	0.191
Grid -3 Convergence Point	6 iterations	48 iterations

Since the computation of MDP and its fitness doesn't match BFS, DFS or A*, comparing them apples-to-apples will not be accurate. *Noise could be kept to 0 and checked, but that again would depend on the implementation of the algorithms and the grid size.*

In such cases, the comparison can be made based on the performance of the combined approach. For example, the comparison can be based on the quality of the solution found, the computational resources required to find the solution, or the robustness of the approach to different scenarios. However, if the comparison is being made purely between search

algorithms and MDP algorithms, then it may not make sense to compare them directly as they solve different types of problems.

XII. Conclusions and Take-aways

Search algorithms are used to traverse a search space to find a solution to a problem. They do not necessarily consider the long-term consequences of their actions, and they may need to explore a large portion of the search space before finding a solution.

On the other hand, MDP algorithms are used to find optimal policies for decision-making problems where the outcomes depend on both the agent's actions and the environment's probabilistic transitions. MDP algorithms consider the long-term consequences of the agent's actions and aim to find a policy that maximizes some reward or utility function.

DFS is typically faster than BFS in terms of runtime because it explores nodes deeply before backtracking. This means that DFS may be more efficient in terms of space complexity since it only needs to store a single path at a time, whereas BFS needs to store all the paths to the frontier in the queue.

However, DFS may not be efficient in terms of finding the optimal solution or the shortest path. DFS may get stuck in an infinite loop or a deep branch before finding the goal, while BFS guarantees to find the shortest path to the goal. Therefore, BFS may be a better choice when finding the optimal solution is important.

BFS can fail if there is an infinite path or loop in the graph. For example, if there is a cycle in the graph, BFS may get stuck exploring the same nodes repeatedly and never reach the target node. In such a case, BFS will not terminate and may consume all available memory resources.

DFS can fail if the search space is very large and the depth of the solution is much greater than the depth at which the goal node is located. This situation is known as a "depth-bound violation." If the depth limit is not set, DFS can get stuck exploring an infinite path or loop as well.

A* performed better because A* uses a heuristic function, it is an informed search algorithm. In contrast, BFS and DFS are uninformed search algorithms that do not use any information about the problem other than the structure of the graph or grid. This means that A* can make more informed decisions about which nodes to explore next, which can result in faster and more efficient search. It also guarantees to find the optimal solution if an admissible heuristic function is used.

However, BFS or DFS may find the optimal solution faster than A* as it can be more computationally expensive than BFS or DFS if the heuristic function is complex or if there are many nodes to explore.

MDPs are often more suitable for certain types of decision-making problems compared to heuristic searching as they are used to model decision-making problems where there is uncertainty about the outcomes of actions. MDPs are well-suited for problems where the environment can change over time, and the agent needs to adapt its policy accordingly. MDPs can handle problems with large state spaces by using value iteration or policy iteration

algorithms, which can converge to the optimal policy in polynomial time. In contrast, heuristic searching algorithms may be impractical or infeasible for problems with large state spaces.

Now, in some scenarios, search algorithms and MDP algorithms may be used together. For example, in a robotics navigation problem, a search algorithm may be used to find a path from the current location to the goal location, while an MDP algorithm may be used to find an optimal policy for the robot to follow along that path.

References

1) https://inst.eecs.berkeley.edu/~cs188/sp20/project1/ : CS188 open source pacman project which is used to implement these algorithms.

2) https://cs.stanford.edu/people/abisee/tutorial/ : Sailors Tutorial Page

APPENDIX

1) https://github.com/aggarpa/AI---Assignment-1/blob/dev/search/search.py - Search Algorithms Implementations

2) https://github.com/aggarpa/AI---Assignment-1/blob/dev/reinforcement/valueIterationAgents.py - Value Iteration Algorithm Implementation

3)https://github.com/aggarpa/AI---Assignment-1/blob/dev/reinforcement/policyIterationAgents.py - Policy Iteration Algorithm Implementation