VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

PROJECT REPORT

ARTIFICIAL INTELLIGENCE [SWE4010]

DESIGN OF SEARCH AGENTS USING PACMAN

Team Members:

Registration Number	Name
18MIS1003	Narendra Babu M
18MIS1029	Vaishnavi S
18MIS1085	Prarthana K

Abstract

Pac-Man is a retro game that involves a maze-like environment where a yellow, pie-shaped Pac-Man character, travels around the maze trying to eat dots and while avoiding four hunting ghosts. The maze-like environment makes it possible to test the various search algorithms while defining a winning condition. Thus in our project, we aim to compare various informed and uninformed search algorithms and compare their performances on various maze sizes and also study how various heuristic functions affect the performance of a search algorithm.

Literature Survey

In the paper titled Teaching Introductory Artificial Intelligence with Pac-Man [1], the author says that the pacman environment supports deterministic, stochastic, partially informed and adversarial problem settings. Thus the pacman environment becomes intuitive with grids and objects and is rich with several AI concepts and challenges. Additionally it also helps in visualizing the problems and thereby provides a better understanding of the AI concepts. By visualization additional information is gained which can be used for debugging program failures, increasing efficiency and exploring different solutions. Hence this domain was chosen to implement and compare various search techniques.

The paper titled MM: A bidirectional search algorithm[2] gives 3 different variations of bidirectional search with fractional search and using a priority function. This algorithm can be used in environments where the search space is bigger and the computational power is limited. For our pacman project since the search state is finite, a simple bidirectional search which follows breadth first search is able to find a point of intersection of the forward and backward searches.

In the paper titled Searching Algorithms And A Comparison Methodology, Used In Studying Artificial Intelligence[3], analysis is done to develop a methodology for comparing algorithms which will potentially improve the understanding of concepts and makes it easy to better understand the differences, advantages and disadvantages of each of the search techniques. The methodology is to be visually embedded in a web-based virtual laboratory for further studies.

In the paper titled The Comparison of DFS and BFS Methods on 2D Ising Model [4] , algorithms were explored for finding clustering, the average number of nodes in a cluster, the size, distribution of clusters, and the appearance of an infinite search space. Breadth-first search and depth-first search traversing algorithms were compared and run simultaneously. A program was also written to compare two algorithms within the framework of the conditions of interest like execution speed, simpler implementation, etc.

Proposed Work

In our project, we are comparing the performance of Breadth-First Search (BFS), Depth First Search(DFS), Uniform Cost Search(UCS), Bidirectional Search(BIS), and A* Search with Heuristics.

I. Breadth-First Search (BFS)

BFS is an uninformed search method in which you start traversing the graph layerwise from a specified node (source or starting node) and explore the neighbour nodes (nodes which are directly connected to source node). Then you must go to the next-level neighbour nodes. We continue the process till we attain the desired condition. Here The technique is implemented using a **Queue** data structure, which follows a first-in-first-out approach in which new nodes are appended to the queue's end. This ensures that the shallowest unexpanded node is expanded.

II. Depth-First Search (DFS)

DFS is similar to BFS in terms of the pseudocode or the implementation aspect of it except a Stack is used instead of a queue. From the starting node or root node all the branches are visited along the same node. When a dead end is met the agent backtracks and moves to the next neighboring node. In a similar way all the nodes and branches are covered. The usage of a **stack** data structure in the implementation part ensures the LIFO followed is used which is how the DFS algorithm works.

III. Uniform Cost Search (UCS)

Uniform-cost search is an uninformed search algorithm that finds a path from the source to the destination by calculating the lowest cumulative cost. Starting at the root, nodes are expanded according to the minimum cumulative cost. A **Priority Queue** is used to implement the uniform-cost search, as it can pop the nodes based on the priority of least cost nodes. It gives the lowest cumulative cost the highest priority. Uniform cost search will become equivalent to Breadth First search algorithm, if all the edges have the same path cost.

UCS may be used to solve any graph or tree in which the optimal cost is required. Even if the goal state is reached, the agent will keep looking for other alternative optimal paths. We retain a priority queue and deliver the least expensive next state from all the visited states' nearby states.

The major drawback of the UCS algorithm is that it is unconcerned about the number of steps involved in searching and is only concerned with path cost. As a result, in case of infinite search spaces, this algorithm may become trapped in an infinite loop.

IV. Bidirectional Search (BIS)

Bidirectional search is a graph search technique that finds the shortest path from a starting point to a target vertex. It has 2 search agents that conduct two searches simultaneously:

1. Forward search from the source node towards the goal node
2. Backward search from the goal node towards the source node

Bidirectional search divides a single search graph into two smaller subgraphs, one originating at the source node and the other at the goal node. Breadth First search is run simultaneously on both the nodes and the neighboring nodes are explored and two separate graphs are constructed. When these two graphs intersect, the search comes to an end. The solution i.e. the path from source node to goal node is the concatenated path from the source node to the meeting point and the reversed path from the meeting point to the goal node.

solution= path(source, intersection)+ reverse(path(intersection, goal))

Bidirectional search can be driven by a heuristic estimate of the remaining distance from source to goal and vice versa in order to discover the shortest path, just like the A* algorithm. This algorithm is fast and consumes less memory when compared to other algorithms; but to implement this, one must know the goal state well in advance.

V. A Search with Heuristics*

A start is similar to the uniform cost search algorithm. The only difference between the 2 algorithms is that the A* algorithm also adds the heuristic function which is nothing but the distance measure between the 2 nodes along with the cost function. The efficiency of the A* algorithm varies with the choice of heuristics. In our project we have implemented the following heuristic functions

- *Manhattan Distance :* It is the sum of the difference between the x and y points of the 2 nodes. **Manhattan Distance = |x1-x2| + |y1-y2|**
- *Cosine Distance :* Often called the cosine similarity, this measures the cosine of the angle between 2 vectors and measures the similarity between them by determining if they are pointing in the same direction.

  Cosine Distance = xy / ||x||.||y||

- *Chebyshev Distance:* It is also called the maximum metric as it finds the distance between 2 sets of points as the largest difference over any of their axis values. **Chebyshev Distance = max (|xi-yi|)**
- *Euclidean Distance:* It is the most commonly used distance formula and is defined as the straight line distance between any 2 points.

Euclidean Distance = [(x1-x2)^2 + (y1-y2)^2]^0.5

- *Canberra Distance :* This can be considered as a variation of the Manhattan distance. It is just a weighted version of the prior measure. This measure examines the sum of a series of fractional differences between the nodes. **Canberra Distance = |xi-yi| / |xi| + |yi|**

Results

The following table summarizes the performance of all the algorithms when the initial position of the pacman was at (5,5) ie, the top right corner, and the goal state being (1,1) ie, the bottom left corner. The cost of the search appears to be the same while the node expanded differs. As we can see from the table below bidirectional search performs well on 3 of the 4 maze sizes with respect to the nodes expanded.

Maze	BFS (cost,nodes)	DFS (cost,nodes)	UCS (cost,nodes)	A* (cost,nodes)	BIS (cost,nodes)
Tiny	8,15	10,15	8,15	8,15	8,12
Small	19,92	49,59	19,92	19,92	19,54
Medium	68,269	130,146	68,269	68,269	68,185
Big	210,620	210,390	210,620	210,620	210,606

Similarly for the A* algorithm the manhattan heuristic performs well with respect to the nodes expanded. Chebyshev and Euclidean heuristics also perform well with values being at a closer range to the Manhattan heuristics than the other metrics.

Maze	Manhattan	Cosine	Chebyshev	Euclidean	Canberra
Tiny	8,14	8,15	8,13	8,13	8,15
Small	19,53	19,91	19,56	19,56	19,92
Medium	68,221	68,268	68,228	68,226	68,269
Big	210,549	210,619	210,578	210,557	210,620

Conclusion

The Bidirectional Search shows a better performance than the other uninformed search techniques. Although it proves to be costly in the big maze. We can infer from this that as the size of the maze increases the time taken to find the intersection point of the forward and backward searches also increases and the algorithm will fail for infinite search spaces when an intersection point is not found. In such cases, the Meet in the Middle algorithm can be used which is a bidirectional search algorithm that is guaranteed to meet in the middle. It involves the use of epsilon or priority function and fractional search techniques to improve the efficiency of search. The efficiency of the A* algorithm changes with the heuristic function used therefore a better heuristic must be chosen depending on the use case.

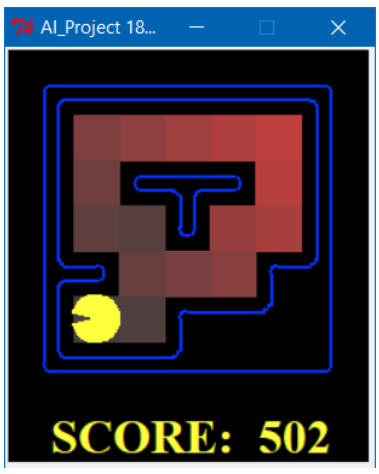
References

1. DeNero, John & Klein, Dan. (2010). Teaching Introductory Artificial Intelligence with Pac-Man
2. R.C. Holte et al., MM: A bidirectional search algorithm that is guaranteed to meet in the middle, Artif. Intell. (2017), http://dx.doi.org/10.1016/j.artint.2017.05.004
3. Петрова, Мария & Atanasova, Desislava. (2018). Searching Algorithms And A Comparison Methodology, Used In Studying Artificial Intelligence. 1767-1775. 10.21125/Iceri.2018.1395.
4. Kapitan DY, Rybin AE, Vasiliev EV, Perzhu AV, Andriuschenko PD. The Comparison of DFS and BFS Methods on 2D Ising Model. InCEUR Workshop Proceedings 2019 (pp. 147-152).

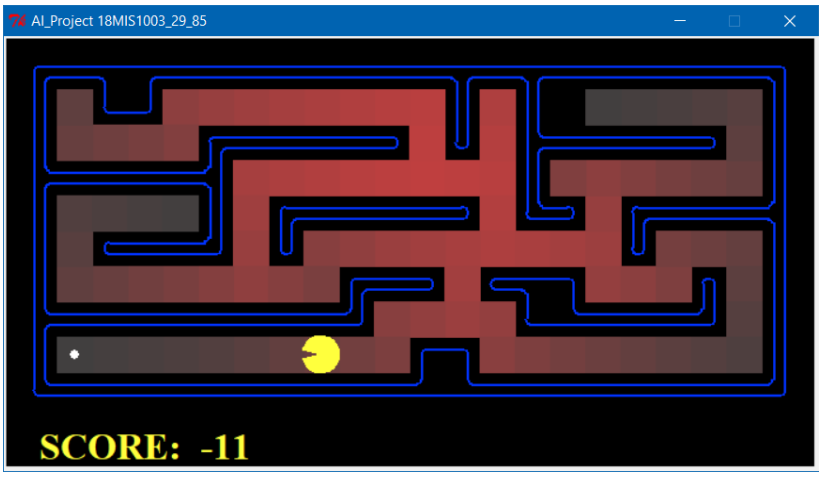
Appendix

Screenshots

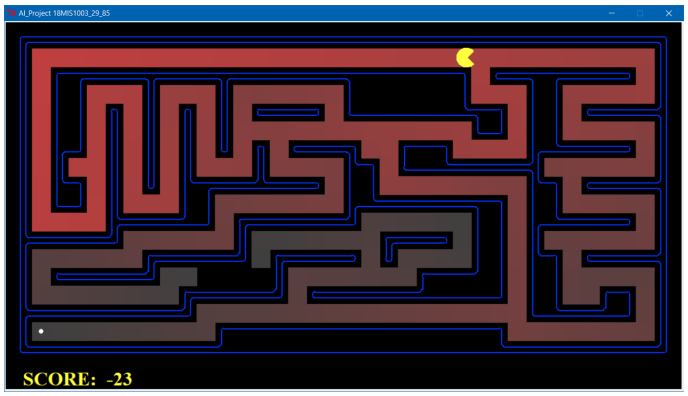
Tiny Maze



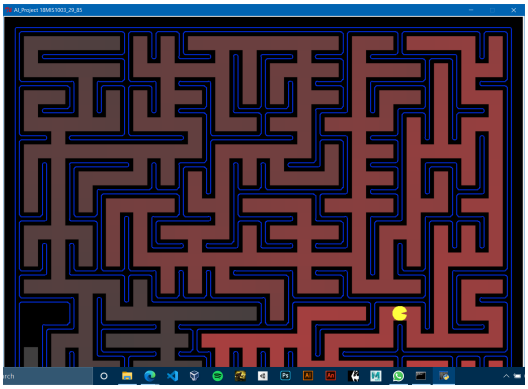
Small Maze



Medium Maze



Big Maze



Breadth-First Search (BFS)

```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
('startnode = ', (5, 5))
(4, 5)
(5, 3)
(3, 5)
(4, 3)
(2, 5)
(5, 3)
(1, 5)
(3, 2)
(2, 5)
(2, 2)
(1, 3)
(3, 2)
(2, 3)
(1, 3)
(1, 1)
('Goal node reached at : ', (1, 1))
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15
```

Uniform Cost Search (UCS)

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
('startnode = ', (5, 5))
(4, 5)
(5, 3)
(3, 5)
(4, 3)
(2, 5)
(5, 3)
(1, 5)
(3, 2)
(2, 5)
(2, 2)
(1, 3)
(3, 2)
(2, 3)
(1, 3)
(1, 1)
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 15

C:\Users\Admin\Desktop\Fall21-22\AI\AI_Proj>
```

A* with Chebyshev distance

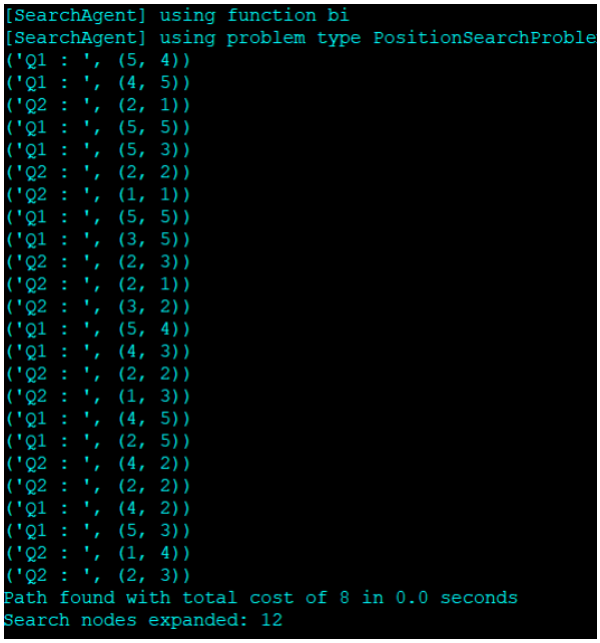
```
[SearchAgent] using function astar and heuristic chebyshevDistance
[SearchAgent] using problem type PositionSearchProblem
('startnode = ', (5, 5))
(4, 5)
(5, 3)
(3, 5)
(4, 3)
(2, 5)
(5, 3)
(1, 5)
(3, 2)
(2, 2)
(3, 2)
(2, 5)
(1, 1)
(1, 3)
Path found with total cost of 8 in 0.0 seconds
Search nodes expanded: 13
```

Depth-First Search (DFS)

```
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
('startnode = ', (5, 5))
(4, 5)
(3, 5)
(2, 5)
(1, 5)
(2, 5)
(1, 3)
(2, 3)
(1, 3)
(3, 2)
(2, 2)
(3, 2)
(5, 3)
(4, 3)
(5, 3)
(1, 1)
('Goal node reached at : ', (1, 1))
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15

C:\Users\Admin\Desktop\Fall21-22\AI\AI_Proj>
```

Bidirectional Search (BIS)



Project Code

```
def depthFirstSearch(problem):
    stack = Stack()
    explorednode = set()
    startnode = problem.getStartState()
    print('startnode = ', startnode)
    stack.push((startnode,[]))
    while stack.isEmpty() is not True:
        currentnode, direction = stack.pop()
        explorednode.add(currentnode)
        if problem.isGoalState(currentnode):
            print('Goal node reached at : ', currentnode)
            return direction
        for (successor, action, stepCost) in problem.getSuccessors(currentnode):
            if successor not in explorednode:
                stack.push((successor, direction + [action]))
        print(successor)


def breadthFirstSearch(problem):
    queue = Queue()
    explorednode = set()
```

```
    startnode = problem.getStartState()
    print ('startnode = ', startnode)
    queue.push((startnode,[]))
    while queue.isEmpty() is not True:
        currentnode, direction = queue.pop()
        if currentnode not in explorednode:
            explorednode.add(currentnode)
            if problem.isGoalState(currentnode):
                print('Goal node reached at : ', currentnode)
                return direction
            for (successor, action, stepCost) in problem.getSuccessors(currentnode):
                if successor not in explorednode:
                    queue.push((successor, direction + [action]))
            print(successor)


def uniformCostSearch(problem):
    pqueue = PriorityQueue()
    explorednode = set()
    startnode = problem.getStartState()
    print ('startnode = ', startnode)
    pqueue.push((startnode,[]),0)
    while pqueue.isEmpty() is not True:
        (currentnode,direction)  = pqueue.pop()
        if currentnode not in explorednode:
            explorednode.add(currentnode)
            if problem.isGoalState(currentnode):
                return direction
                print('Goal node reached at : ', currentnode)
            for (successor, action, stepCost) in problem.getSuccessors(currentnode):
                if successor not in explorednode:
                    pqueue.push((successor, direction + [action]), problem.getCostOfActions(direction
+ [action]))
            print(successor)



def aStarSearch(problem, heuristic=nullHeuristic):
     pqueue = PriorityQueue()
    explorednode = set()
    startnode = problem.getStartState()
```

```
    print ('startnode = ', startnode)
    pqueue.push((startnode, []), 0)
    while pqueue.isEmpty() is not True:
        (currentnode, direction) = pqueue.pop()
        if currentnode not in explorednode:
            explorednode.add(currentnode)
            if problem.isGoalState(currentnode):
                return direction
                print('Goal node reached at : ', currentnode)
            for (successor, action, stepCost) in problem.getSuccessors(currentnode):
                if successor not in explorednode:
                    pqueue.push((successor, direction + [action]), problem.getCostOfActions(direction
+ [action]) + heuristic(successor, problem))
            print(successor)
    util.raiseNotDefined()


def bidirection(problem):
    q1 = Queue()
    temp_q1 = []
    q2 = Queue()
    temp_q2 = []
    explorednode1 = set()
    explorednode2 = set()
    startnode = problem.getStartState()
    endnode = problem.goal

    q1.push((startnode,[]))
    q2.push((endnode, []))

    while q1.isEmpty() is not True and q2.isEmpty() is not True:
        if q1.isEmpty() is not True:
            currentnode, direction = q1.pop()
            if currentnode not in explorednode1:
                explorednode1.add(currentnode)
                if problem.isGoalState(currentnode) or (currentnode in temp_q2):
                    while q2.isEmpty() == False:
                        node, direc = q2.pop()
                        if node == currentnode:
                            solution = direction + direc.reverse()
                            return solution
```

```
            for(successor, action, stepCost) in problem.getSuccessors(currentnode):
                q1.push((successor, direction + [action]))
                print("Q1 : ", successor)
                temp_q1.append(successor)
        if q2.isEmpty() is not True:
            currentnode, direction = q2.pop()
            if currentnode not in explorednode2:
                explorednode2.add(currentnode)
                if currentnode in temp_q1:
                    while q1.isEmpty() == False:
                        node, direc = q1.pop()
                        if node == currentnode:
                            direction.reverse()
                            solution = direc + rev(direction)
                            return solution
                for(successor, action, stepCost) in problem.getSuccessors(currentnode):
                    q2.push((successor, direction + [action]))
                    print("Q2 : ", successor)
                    temp_q2.append(successor)


def rev(direction):
    j = []
    for i in direction:
        if i == 'North':
            j.append('South')
        elif i == 'South':
            j.append('North')
        elif i== 'East':
            j.append('West')
        else:
            j.append('East')
    return j


def manhattanHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
```

```
    xy1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) **  2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

def chebyshevDistance(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    dx = abs(xy1[0] - xy2[0])
    dy = abs(xy1[1] - xy2[1])
    return max(dx, dy)

def cosineDistance(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    xy = xy1[0]*xy1[1] + xy2[0]*xy2[1]
    x2 = (xy1[0] ** 2 + xy2[0] ** 2) ** 0.5
    y2 = (xy1[1] ** 2 + xy2[1] ** 2) ** 0.5
    return 1 - (xy / (x2*y2))

def canberraDistance(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
      cdist = (abs(xy1[0] - xy1[1]) / abs(xy1[0] + xy1[1])) + (abs(xy2[0] - xy2[1]) / abs(xy2[0] +
xy2[1]))
    return cdist
```