Bi-directional search applied to Pacman and Rubiks cube problem.

Arsh Padda 1209620227 | Raj Sadaye 1213175416 | Rushikesh Sargar 1214365814 | Shubham Gondane 1213179615

***Abstract*— For many problems with large action space, it is observed that traditional searches like Breadth First Search(BFS) and Depth First Search(DFS) run in exponential time. We can dramatically reduce the amount of state exploration by using the Bidirectional Search algorithm. In this paper, we first compare the performance of the Bi-directional search algorithm to Breadth-First Search, Depth-First Search, and A* algorithm by using the Pacman food pellet search environment. We further use bidirectional search using the iterative deepening A* algorithm to search for the solution of the Rubiks cube problem.**

I. INTRODUCTION

Bi-directional search alternates between 2 separate search algorithms:

1) A forward search from the source/initial node towards the goal node.
2) Backward search from goal/target node towards the source node.

But some important points to take into consideration are that bi-directional search can only be applied when :

1) Both initial and goal states are unique and completely defined.
2) Branching factor is equal in forward as well as backward direction.
3) Search operators i.e actions are reversible.[3]

Many games and puzzles involve intelligent movement of characters in either static or changing environments. Developing effective path-finding algorithms for games and puzzles is an active area or research.

Bi-directional search is known to be more efficient than unidirectional search when there is no heuristic information available. In such cases, search frontiers typically meet somewhere half-way between the start and goal nodes, greatly reducing the explored part of the graph. However, the first attempts to develop bidirectional heuristic search obtained poor results when compared to A*. Ever since, the efficiency of bidirectional heuristic search has been subject to considerable debate and research, obtaining mixed results in different problem domains.

If BFS is used for implementing bi-directional search, it is guaranteed to be complete and optimal. If b is the branching factor and d is the depth for a search tree then the time and space complexity of bi-directional search when using BFS is $O(b^{d/2})$. When using a heuristics based bi-directional search, it is important to ensure that the forward and backward searches meet in the middle to ensure completeness of the search.

Returned Action by reverse search	Reverse action for correct path
Move Left	Move Right
Move Up	Move Down
Move Right	Move Left
Move Down	Move Up

TABLE I: Reversed move sequence

II. BACKGROUND

Pacman food search problem involves teaching Pacman how to intelligently find his food in the shortest time possible.

This involves running 2 searches:

1) First starting from pacman's initial position to the food pellet position.
2) Second starting from food position to initial pacman position.

In this problem we can perform search in a bi- directional manner by following Algorithm 1.

Algorithm 1 public double Bidirectional(G,src,goal)

```
new queue Q , new set V
Q ← add src
V ← add src
while Q is not empty do
  t ≤ Q.remove ()
  if if t is goal then
    return t
  end if
  for all edges e in G adjacent to t loop do
    u ≤ G.adjacentVertex(t,e)
    if u is not in V then
      V ← add u
      Q ← add v
    end if
  end for
end while
return none
```

One key aspect is that when that the path for the reverse search is returned, to get the correct path from the initial node to final node, the action sequence returned by reverse search has to be reversed as shown in Table I.

The pacman search problem contains an agent (pacman) who is trying to find a path between two points on the map. There are 4 moves pacman can do in each position i.e. North,

Maze	Algorithm	Total cost	Search nodes expanded	Pacman score
tinyMaze	BFS	8	15	502
tinyMaze	DFS	10	15	500
tinyMaze	A star	8	15	502
tinyMaze	Bi-directional	8	12	502
smallMaze	BFS	19	92	491
smallMaze	DFS	49	59	461
smallMaze	A star	19	92	491
smallMaze	Bi-directional	19	31	491
mediumMaze	BFS	68	269	442
mediumMaze	DFS	130	146	380
mediumMaze	A star	68	269	442
mediumMaze	Bi-directional	68	170	442
bigMaze	BFS	210	620	300
bigMaze	DFS	210	390	300
bigMaze	A star	210	620	300
bigMaze	Bi-directional	210	596	300

TABLE II: Comparision of various search algorithms on different Pacman environments

South, East and West. The Bidirectional search does a Breath First Search from both start position and the goal position. The search continues till either we find the goal position from the start position or the two search meet. We tried 4 different mazes namely tinyMaze, smallMaze, mediumMaze and bigMaze. The comparison was done with Breath First Search, Depth First Search and A Star Search. The results are show in Table II.

Rubiks Cube problem consists of a 3 x 3 x 3 cube, with different colored stickers on each of the exposed squares of the subcubes, or cubies. Any 3 x 3 x 1 plane of the cube can be rotated or twisted 90, 180, or 270 degrees relative to the rest of the cube. In the goal state, all the squares on each side of the cube are the same color. The puzzle is scrambled by making a number of random twists, and the task is to restore the cube to its original goal state. The problem is quite difficult. In fact, there are 4.3252 x 10^{19} different states that can be reached from any given configuration.Iterative deepening A* algorithm can be used to solve this problem. [2]

Iterative deepening A* (IDA*) is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph. It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm. Since it is a variant of the depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus doesnt go to the same depth everywhere in the search tree. Unlike A*, IDA* doesnt utilize dynamic programming and therefore often ends up exploring the same nodes many times. IDA* is a memory constrained version of A*. It does everything that the A* does, it has the optimal characteristics of A* to find the shortest path but it uses less memory than A*.

The traditional Iterative Deepening A*(IDA*) search algorithm is uni-directional. We implement a bidirectional version of the IDA* algorithm and apply that to the Rubik's cube problem.

III. IMPLEMENTATION DETAILS

A. Pattern Databases

A heuristic as a function can computed by an informed search algorithm. Any function can also be computed by a table lookup, given sufficient memory. In fact, for reasons of efficiency, heuristic functions are commonly precomputed and stored in memory. These tables only need to be computed once for each goal state, and its cost can be amortized over the solution of multiple problem in- stances with the same goal. of multiple problem instances with the same goal. The use of such tables is called pattern databases is due to (Culberson and Schaeffer 1996), who applied it to the Fifteen Puzzle.[4]

B. Heuristic

For each cubic, compute the minimum number of moves required to correctly position and orient it, and sum these values over all cubies. This is the three-dimensional version of Manhattan distance used for sliding-tile puzzles. The heuristic considered here is to take the maximum of the sum of the Manhattan distances of the corner cubies, and the edge cubies, each divided by four.

Consider the eight corner cubies, the position and orientation of the last cubic is deter- mined by the remaining seven, so there are exactly 8! x 37 = 88,179,840 possible combinations. Using a breadth-first search from the goal state, these states can be enumerated, and record in a table the number of moves required to solve each combination of corner cubies.

Similarly, if we consider the 12 edge cubies, the number of possible combinations for six of the twelve edge cubies is 12!/6! x 26 = 42,577,920. Compute the corresponding heuristic table for the remaining six edge cubies. The heuristic used for the our implementation is the maximum of all three of these values: all eight corner cubies, and two groups of six edge cubies each.

For our implementation we are not generating the pattern databases table ourselves, we are using an existing implementation of pattern databases that generates the heuristic

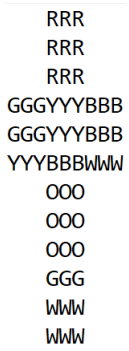

Fig. 1: State representation for Rubik's cube

table for eight corner cubies and heuristic table for two sets of six edge cubies.

C. Action Model

We do not follow the traditional actions of the Rubik's cube problem. We have implemented a modified action model which includes actions that have the following characteristics:

- The cube can only be rotated in clockwise direction.
- Every action corresponds to a colored face being rotated clockwise.
- Every face can be rotated upto three times in clockwise direction
- An action is represented by pair:
  <Color, Number of Rotations (i.e. 1 every time)>
  An example of an action sequence is : R1W1Y1

D. Representation of state space

A Cube Node consists of 4 parts:

- State: Which is represented as a string indicating the position of 54 cubies.
- Heuristic.
- g-value: For calculating bound of the search algorithm.
- Path: Sequence of actions taken from initial state to reach the current state.

Example for state representation can be seen in Figure 1

E. Features of the Bi-directional Algorithm

We maintain 2 priority queues ordered according to the value of the heuristic. These priority queues store the fringe for each search. The start node is added to one queue and goal node is added to the second queue. We also maintain two hashsets for explored nodes. Also two hashsets are used to store paths(the states visited). The bound for top-down search is the heuristic value of the start node. The bound for bottom-up search is the heuristic value for the goal node.

Alternatively, both the approaches explore their successors withing the respective bounds considering the heuristic values of the successors from the pattern databases. Each approach looks if the current node has already been explored by the other approach. If such match is found both the matched nodes are returned as the solution.

Algorithm 2 Bi-directional IDA* Algorithm

```
while Solution is not found do
  while Both the priority queues are not empty do
    Select node from the TD queue
    if Not explored earlier then
      add in explored
      add its state in path
      if BU path contains the current node then
        Find out the node in BU explored
        return Both the nodes
      end if
      Generate successors if they are not explored add
      them in the priority queue
    end if
  end while
end while
```

The solution returned has two parts, one returned by the top-down approach and other by the bottom up approach. The path consist of series of actions performed to reach that particular node from respective source node.(start node in case of Top-down approach and Goal node in case of bottom up approach).
e.g. O1W1W1B1B1B1 as the action consist of face and clockwise turn (i.e. 1) By merging the similar consecutive actions this can be formatted as 01W2B3.

To get the complete path to the goal from the start state, we need to append top-down path and the reversed bottom-up path. While reversing the bottom up path the action needs to be reversed as well, a clockwise should be reported as anticlockwise and viceversa.
For example:
Path Received from Top-Down approach B1R2
Path Received from Bottom-Up approach O1W2B3
Bottom Up path when reversed becomes B1W2O3
Thus the final path would be B1R2B1W2O3

Reversal of clockwise1 i.e anticlockwise1 could be achieved by having 1 rotation where 3 are done and having 3 rotations where only 1 is performed.

F. Language and System specifications

The Bidirectional Search Algorithm for the Pacman gridworld has been done on the available environment.
Programming Language : Python 2.7
Operating System : Win10

The Bidirectional Search Algorithm for Rubik's cube solver has been programmed in Java 10.0.2
Operating System: Win10
RAM: 16GB

Test Case	Total nodes visited for IDA*	Total nodes visited for Bi-directional IDA*
Cube02	390	13
Cube03	1905	535
Cube04	124	584
Cube05	486	271

TABLE III: Comparison of Bi-directional IDA* search with traditional A* search for the Rubik's cube problem

Algorithm 3 IDA * Algorithm

```
Function: ida_star(root)
bound := h(root)
path := [root]
while TRUE do
  t := search(path, 0, bound)
  if t = FOUND then
    return (path, bound)
  end if
  if t = ∞ then
    return NOT_FOUND
  end if
  bound := t
end while
return none
End function

Function: search(path, g, bound)
node := path.last
f := g + h(node)
if f > bound then
  return f
end if
if is_goal(node) then
  return FOUND
end if
min := ∞
for succ in successors(node) do
  if succ not in path then
    path.push(succ)
    t := search(path, g + cost(node, succ), bound)
    if t = FOUND then
      return FOUND
    end if
    if t < min then
      min := t
      path.pop()
    end if
  end if
end for
return min
End function
```

IV. RESULTS AND DISCUSSIONS

A. Highlights of the algorithm

- It will always find the optimal solution provided that it exists and that if a heuristic is supplied it must be admissible.
- Heuristic is not necessary, it is used to speed up the process.
- Various heuristics can be integrated to the algorithm without changing the basic code.
- The cost of each move can be tweaked into the algorithms as easily as the heuristic.
- Uses a lot less memory which increases linearly as it doesnt store and forgets after it reaches a certain depth and start over again.

B. Overview of Results

- If we observe results from table III we can infer that Bi-directional IDA* search expands lesser nodes than the traditional IDA* star.
- The time and space complexity for a bi-directional search algorithm is usually O($b^{d/2}$). However in the case of a heuristic based search implemented in this paper, the complexity depends on the following attributes:
  - Heuristic value of the starting node.
  - The available actions from each state for a node.
- For the available computation power, The algorithm works faster for the simpler cubes and as the state space grows for the complex problems the time complexity increases

V. CONCLUSION

The Bi-directional search applied to the Pacman path finding problem performs better than BFS, DFS and A* algorithms. The Bi-directional IDA* search algorithm applied to the Rubik's cube problem finds the optimal solution always and the use of heuristic affects the complexity of the search. Also, as seen from the results, the Bi-directional IDA* expands a fewer number of nodes as compared to IDA*.

REFERENCES

[1] **Base paper**: Holte, Felner, Sharon, Sturtevant, & Chen. (2017). MM: A bidirectional search algorithm that is guaranteed to meet in the middle. Artificial Intelligence, 252(C), 232-266.

[2] Korf, R. E. 1997. In Proc. 14th National Conference on Artificial Intelligence (AAAI), 700705.

[3] Pulido, Francisco Javier, L. Mandow, and JL Prez de la Cruz. "An analysis of bidirectional heuristic search in game maps."

[4] Culberson, Joseph C., and Jonathan Schaeffer. "Searching with pattern databases." Conference of the Canadian Society for Computational Studies of Intelligence. Springer, Berlin, Heidelberg, 1996.

VI. Notes about team effectiveness

- The team had good communication and synchronization throughout this project implementation.
- Implementation started with background research and approaches to solve the problems where every member contributed his part proactively and proposed various approaches to solve the problems.
- Everyone showed a good participation in the team meetings and in the development phase of this project.
- Once the plan was created each member was assigned with a tasks where half of the team worked on the Pacman implementation and the other half worked on the Rubik's cube.
- Everyone was keen and responsible towards the task and has contributed evenly towards the implementation of this project.