JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

# A Case Study of Fifteen Puzzle Problem Using BFS & DFS

**Daneshwari N. Kori, Vandana S.Bhat, Jagadeesh D. Pujari**

Department of Information Science and Engineering
SDM College of Engineering and Technology, Dharwad.
danukori21@gmail.com, Vandana.Bhat@sdmcet.ac.in, jaggu_dp@gmail.com

## ABSTRACT

BFS is a graph traversal algorithm [2] that begins at the root node and travels the graph to all of its neighbour. DFS (Depth-First Search) is a graph-searching technique that begins at the top and works its way down until it discovers the required node or the node with no children. On tough issues, best-first search frequently has exponential space needs. Although DFS can tackle tough problems with linear space needs, it cannot take advantage of the vast amounts of additional memory available on today's computers. We review the question of whether to utilize best-first or depth-first search. For the first time, using classic BFS, it was possible to uncover optimal solutions to specified difficult difficulties (the whole benchmark set of Fifteen Puzzle issues) thanks to algorithmic innovations (with the Manhattan distance heuristic only). Due to its exponential space requirements, it was believed that this search strategy couldn't solve randomly generated Fifteen Puzzle instances when practical resource restrictions applied.The result, of case study demonstrates that, when hardware and algorithmic advances are combined, the earlier assessment of best-first search can be revised.

**Keywords:** BFS, DFS, Manhattan Distance Heuristic, A* algorithm.

## INTRODUCTION

The Fifteen Puzzle Problem [1] has built up a 4x4 board with 16 block slots, 15 of which include tiles with numbers ranging from 1 to 15. One spot has been left empty. The tile adjacent to the blank spot has the ability to move. To get to the desired state, the tiles or blocks must be put in a specific order.

The domain of Fifteen Puzzle puzzles [3], which are made up of 15 sliding tiles arranged in a four-by-four grid [14]. There are 16!/2=1013 puzzle configurations in the state space. To be more specific, we used the conventional benchmark of 100 randomly produced and supplied problem instances. Except for the Manhattan distance heuristic, none of the compared algorithms possessed any domain-specific information about the challenge. In the early 1980s, IDA* [7] used this approach to solve all 100 issue situations in terms of determining optimal solutions. Meanwhile, we replicated the experiment by assigning 43 million nodes to A* on a system with 2 GBytes of primary storage. It can now address 78 different types of problems, however the majority of them are straightforward, and it is still unable to solve the more challenging ones. We found it more intriguing and hard to attain the desired improvements in search results rather than trying to gain more and more primary storage so that A* [4] could solve them all. With significantly improved heuristic functions, far more efficient searches are possible, and even solving the Twenty-Four Puzzle issues is now possible. This, on the other hand, denotes improvements in the search results, which is an orthogonal strategy. Since the dawn of knowledge-based systems, it has been generally understood that increasing the amount of knowledge provided by humans can greatly increase performance. In fact, developing such effective heuristics before solutions are required will not always be practicable. Another way to improve things is to let the search engine develop heuristic values on its own.Using this method, the computer can improve a specified but less optimal static heuristic.

The current study makes a contribution by demonstrating how successfully this strategy may be used in traditional best-first search. However, even with far greater accessible storage, A* still does not perform well, and it is superior to its unidirectional competitors in several areas. As a result, we decided to investigate heuristic search in this area. BS* [5] is a classic example of bidirectional heuristic search with "front-to-end" assessments. On a 2 GByte system, it can answer 95 problem instances. Those circumstances in which it is still unable to solve are even more numerous. Those circumstances in which it is still unable to solve are far more challenging than those in which it is successful. A related method based on dynamic improvements to the heuristic known as Max-Switch-A* was said to solve 79 of the provided puzzle instances with 256 MB of storage. We determined that the system with 2 GBytes can solve 99 challenges, but not the collection's most difficult problem. The simple and straightforward approach [6] to apply this heuristic to a BS* algorithmic improvement. With 2 GBytes, this combination can solve the entire benchmark set on the provided system. To the best of our knowledge, it can solve it faster than any of the Manhattan distance heuristic methods that have been previously disclosed. In order to make it self-contained, we review some background information on the dynamically updated heuristic and the bidirectional heuristic search method BS*. Then we go into the issues that arise when they're combined, as well as some concrete solutions. The effectiveness of the best combination we found in comparison to the best rivals is finally shown through experimental results.

## A COLLECTION OF PRODUCTION RULES

Step_1: Determine the issues initial and ending states by doing an analysis of the issue.

Step_2: Gather information about the starting and ending states.

Step_3: Calculate the production steps for transferring the issue to the desired state using the starting database.

Step_4:Select a few rules from the list of ones that can be used to process data.

Step_5: Before going on to the next state, apply those criteria on the previous state.

Step_6: After applying the rules, find some new created states. As a result, set them to their current state.

Step_7: Finally, take some data about the objective state and retrieve it from the previously used present state.

Step_8: Exit.

## SOLVING OF 15 PUZZLE PROBLEM

Initial state: The Initial state or Starting state.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

Goal state: The Final state or Resulting state.

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

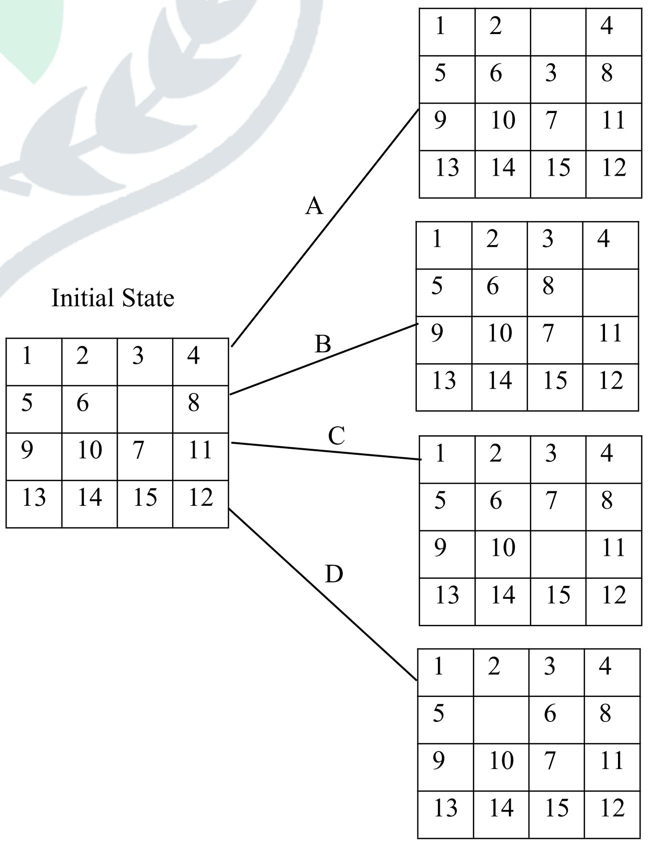
Step_1: From the initial state, f(x) is the first step needed to get to the objective state or goal state. Using the formula

$f(x) = g(x) + h(x)$

where

- $g(x)$ represents the distance from start node.
- $h(x)$ represents the number of misplaced tiles.

Ex: $f(x) = 0 + 4 = 4$



Initial State

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

A

| | | | |
|---|---|---|---|
| 1 | 2 | | 4 |
| 5 | 6 | 3 | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

B

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 8 | |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

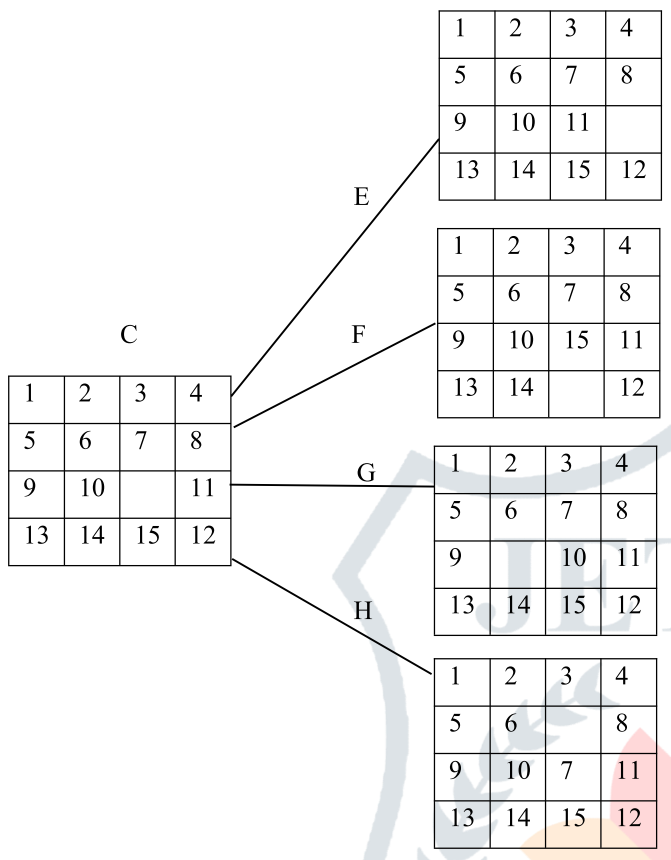
C

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | | 11 |
| 13 | 14 | 15 | 12 |

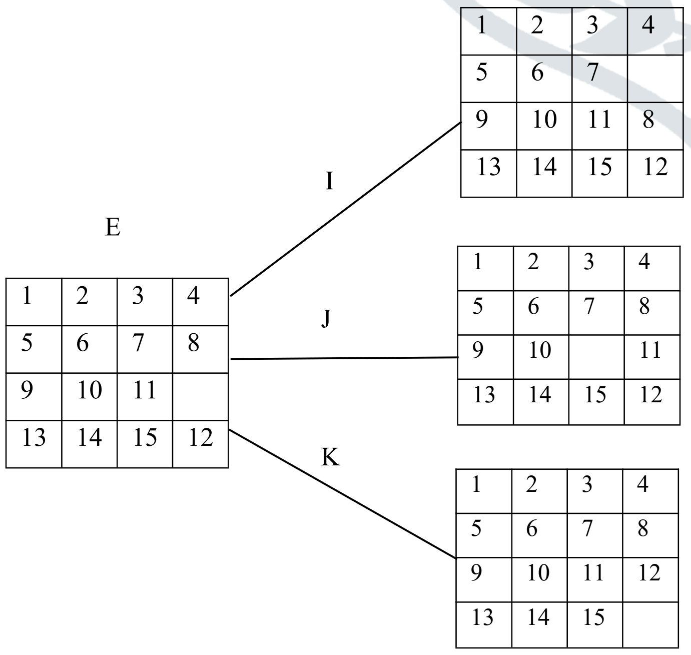
D

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | | 6 | 8 |
| 9 | 10 | 7 | 11 |
| 13 | 14 | 15 | 12 |

Step_2: The step needed to travel from the initial state to the objective state is f(x). The f(x) values of A are 4, B are 4, C are 2, and D are 4. Take C as the current state to reach the following state because 2 is the minimum.



Step_3: In this step, four states can be drawn from tray C. Determine each of their f(x) and select the lowest value. Here, state E has the lowest number 1, therefore use that as the subsequent current state.



Step_4: Three states can be drawn from the tray E in this step. Take the lowest one after computing each of their f(x). In this case, state K has the lowest value. Thus, after a few changes of tiles in various trays' placements, we arrived at the desired state.

The Goal State

K

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | |

Tiles can be divided into separate sets.

➢ Precompute a table for each set.

➢ Don't count moves of tiles that aren't in the set.

➢ Look up the costs for each set in the database.

It has activated a 4x4 board with 16 block slots, of which 15 blocks contain tiles with numbers ranging from 1 to 15. There is one empty spot. The neighbouring tile to the empty space has the capacity to move into it. We need to arrange the tiles in a particular manner in order to reach the objective state.

Estimated Cost c^(x) = f(x) + g^(x).

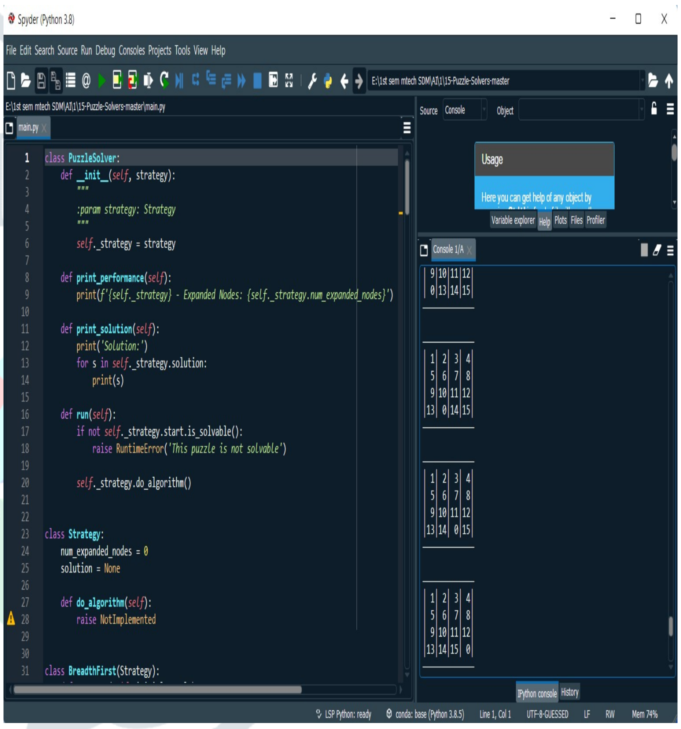
f(x) = Length of the path from the root to node x.

g(x) = Estimated Shortest path length from x to goal node.

= Number of non-blank tiles not in the goal position.

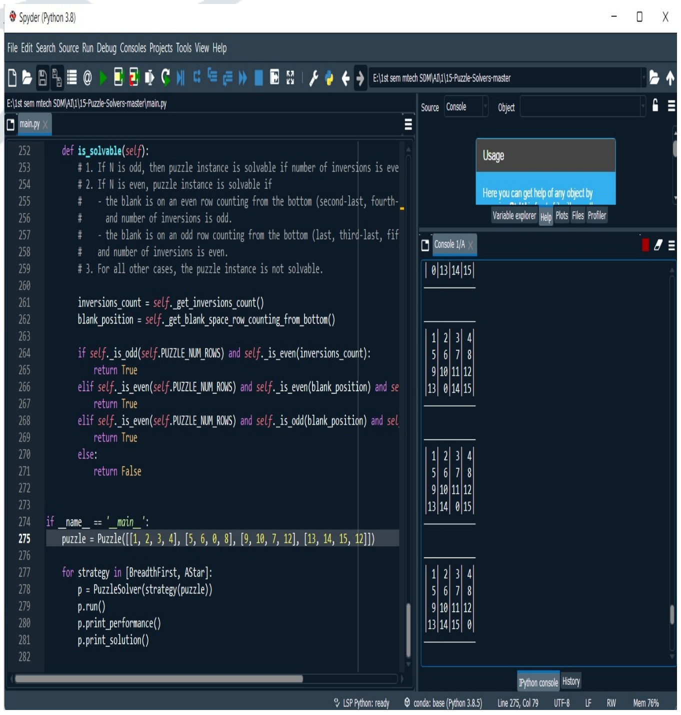
A* is a path-finding algorithm [13] that uses heuristics. When a weighted network and two nodes are provided, the algorithm determines the shortest path between the two nodes. "Two functions are used in an algorithm: g(n) and h(n) (n). The cost from point A to point N, or the sum of the edges linking point A and point N, is g(n). h(n) is a heuristic, or a function that calculates the cheapest route from n to the destination. This function must be acceptable, which means that it should never overestimate the cost. The priority of the node is represented by key(n) = g(n) + h(n)". A lower key value indicates that travelling through that node is less expensive, resulting in a higher priority. The algorithm will prioritize the exploration of nodes. The 15 Puzzle is a well-known heuristic-based algorithm modelling issue [10]. Two popular approaches to this issue include counting the number of misplaced tiles and adding the Manhattan distances between each block and its location in the goal configuration. In order to progress from the initial state to the target state in the puzzle, a tile must be moved to a blank cell at each step. This process is repeated recursively until the goal state is reached. The basic objective is to arrange the tiles to achieve the goal in the most effective way possible. Heuristics Misplaced Tiles [15]. The simplest heuristic [12] for the n Puzzle is to count the number of missing tiles. It fails poorly, though, because the heuristic offers no details on why some tiles are positioned incorrectly, such as the distance between a misplaced tile and its proper location. Manhattan Distance (MD) [13]. Instead, we can combine the MD of each tile's solved position with its present position. It reduces the amount of moves required for each tile to reach its solution position. MD outperforms the prior heuristic and is capable of solving all eight Puzzle cases in a fair period of time. The fundamental problem with MD [16] is that it does not account for tile interactions. The cost bound is incredibly low because it is expected that each tile can travel independently of the others. By incorporating more of the board into the heuristic, we can make it better. Using linear conflicts is one way to do this. We can improve it by including more of the board into the heuristic, and employing linear conflicts is one method to do so. There is a general tendency toward larger lookup tables in exchange for faster search times. To avoid slowing down the actual search, it is just more effective to execute calculations ahead of time. “Using a static additive pattern database with the largest patterns you can build and store is currently the fastest single heuristic for optimally solving the n-Puzzle. A 7-8 partition is sufficient to solve nearly all board states on the order of tens of milliseconds for the 15 Puzzle, and it consumes about 550 MB of storage. With enough storage space, you could even use various database heuristics, such as Walking Distance (WD) [7] and the 5-5-5 pattern database, to their full potential. Even a 5-5-5 partition can solve boards in under a second using only 3 MB of storage, which is quite practical to implement if you have tighter storage constraints. When it comes to pattern databases, WD is fairly efficient on its own, requiring only 25 KB of storage. If you don't want to use databases at all, your best bet is to employ as much MD and ID as possible. At this point, you will be sacrificing a lot of speed for no external storage or precomputation. This is easily sufficient for the 8 Puzzle, but might not be enough for harder instances of the 15 Puzzle” Heuristics exist for each node (a method used for the measuring the distance between the current state and the desired state). Each nodes evaluation function = heuristic + total moves from initial state.

A* always grows the fringe node with the smallest evaluation function. When a tile in this pattern is exchanged with a blank tile, BFS moves the blank tile to a new state. Heuristic search algorithms are intended to return the best path from one state to another. We faced a number of challenges, most notably how to effectively employ the dynamic Max heuristic [8] in a BS*-like algorithm. This demonstrates that bidirectional heuristic search was required for this outcome, as no known unidirectional BFS can resolve all the cases of the given problem (A* being the best in terms of enlarged nodes). It is essential to be able to use it both ways. They find the optimal solution cost as a side effect. However, there are cases where all that is required is a cost estimate for the best solution.


Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
E:\1st sem mtech SDM\AI\1\15-Puzzle-Solvers-master
E:\1st sem mtech SDM\AI\1\15-Puzzle-Solvers-master\main.py
main.py
Source Console Object
Usage
Here you can get help of any object by
Variable explorer Help Plots Files Profiler
Console 1/A
IPython console History
LSP Python: ready conda: base (Python 3.8.5) Line 1, Col 1 UTF-8-GUESSED LF RW Mem 74%

```
class PuzzleSolver:
    def __init__(self, strategy):
        """
        :param strategy: Strategy
        """
        self._strategy = strategy

    def print_performance(self):
        print(f'{self._strategy} - Expanded Nodes: {self._strategy.num_expanded_nodes}')

    def print_solution(self):
        print('Solution:')
        for s in self._strategy.solution:
            print(s)

    def run(self):
        if not self._strategy.start.is_solvable():
            raise RuntimeError('This puzzle is not solvable')

        self._strategy.do_algorithm()


class Strategy:
    num_expanded_nodes = 0
    solution = None

    def do_algorithm(self):
        raise NotImplemented


class BreadthFirst(Strategy):
```

```
| 9|10|11|12|
| 0|13|14|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13| 0|14|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13|14| 0|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13|14|15| 0|
```


Spyder (Python 3.8)
File Edit Search Source Run Debug Consoles Projects Tools View Help
E:\1st sem mtech SDM\AI\1\15-Puzzle-Solvers-master
E:\1st sem mtech SDM\AI\1\15-Puzzle-Solvers-master\main.py
main.py
Source Console Object
Usage
Here you can get help of any object by
Variable explorer Help Plots Files Profiler
Console 1/A
IPython console History
LSP Python: ready conda: base (Python 3.8.5) Line 275, Col 79 UTF-8 LF RW Mem 76%

```
    def is_solvable(self):
        # 1. If N is odd, then puzzle instance is solvable if number of inversions is eve
        # 2. If N is even, puzzle instance is solvable if
        #    - the blank is on an even row counting from the bottom (second-last, fourth-
        #      and number of inversions is odd.
        #    - the blank is on an odd row counting from the bottom (last, third-last, fif
        #    and number of inversions is even.
        # 3. For all other cases, the puzzle instance is not solvable.

        inversions_count = self._get_inversions_count()
        blank_position = self._get_blank_space_row_counting_from_bottom()

        if self._is_odd(self.PUZZLE_NUM_ROWS) and self._is_even(inversions_count):
            return True
        elif self._is_even(self.PUZZLE_NUM_ROWS) and self._is_even(blank_position) and se
            return True
        elif self._is_even(self.PUZZLE_NUM_ROWS) and self._is_odd(blank_position) and sel
            return True
        else:
            return False


if __name__ == '__main__':
    puzzle = Puzzle([[1, 2, 3, 4], [5, 6, 0, 8], [9, 10, 7, 12], [13, 14, 15, 12]])

    for strategy in [BreadthFirst, AStar]:
        p = PuzzleSolver(strategy(puzzle))
        p.run()
        p.print_performance()
        p.print_solution()
```

```
| 0|13|14|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13| 0|14|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13|14| 0|15|

| 1| 2| 3| 4|
| 5| 6| 7| 8|
| 9|10|11|12|
|13|14|15| 0|
```

## DRAWBACKS

1. For the purpose of storing the various trays, this issue calls for a lot of space.
2. The time complexity of the tasks is higher.
3. When moving tiles within the trays, the user must be extremely careful.
4. This method can be used to conquer very challenging puzzle games.

## CONCLUSION

Even with the outdated MD heuristic, the conventional benchmark for Fifteen Puzzle problems can now locate the best answers using best-first search. Generally speaking, it addresses these issues faster, especially the trickier ones. This result was obtained using a dynamically improved heuristic, but algorithmic changes were necessary. Our biggest challenge was figuring out how to employ the dynamic Max heuristic effectively in a BS*-like algorithm. This further demonstrates the need for a bidirectional heuristic search to reach the desired outcome, as no known unidirectional BFS (A* being the best in terms of enlarged nodes) is capable of resolving all the instances of the given problem. Being able to use it both ways is essential.

## FUTURE SCOPE

When people are waiting for a solution in the seconds to minutes, solving problems in "real-time" can make a significant difference. Traditional BFS is capable of effectively resolving reasonably complex problems on the machines currently in use.

## REFERENCE

[1] M.Sinthiya and Dr.M. Chidambaram, "A STUDY ON BEST FIRST SEARCH", International Research Journal of Engineering and Technology (IRJET), Volume: 03, no. 10, June-2016.

[2] Andreas Auer and Hermann Kaindl, " A Case Study of Revisiting Best-First vs. Depth-First Search", Proceedings of the 16th Eureopean Conference on Artificial Intelligence, no. 6, Valencia, Spain, August 22-27, 2004.

[3] J. Culberson and J. Schaeffer, 'Searching with pattern databases', in Advances in Artificial Intelligence, ed., G. McCalla, 402–416, Springer Verlag, Berlin, (1996).

[4] R. Dechter and J. Pearl, 'Generalized best-first strategies and the optimality of A ', J. ACM, 32(3), 505–536, (1985).

[5] H. Kaindl and G. Kainz, 'Bidirectional heuristic search reconsidered', Journal of Artificial Intelligence Research (JAIR), 7, 283–317, (1997).

[6] H. Kaindl, G. Kainz, R. Steiner, A. Auer, and K. Radda, 'Switching from bidirectional to unidirectional search', in Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99). San Francisco, CA: Morgan Kaufmann Publishers, (1999).

[7] R.E. Korf, 'Depth-first iterative deepening: An optimal admissible tree search', Artificial Intelligence, 27(1), 97–109, (1985).

[8] R.E. Korf and A. Felner, 'Disjoint pattern database heuristics', Artificial Intelligence, 134, 9–22, (2002).

[9] R.E. Korf and L.A. Taylor, 'Finding optimal solutions to the TwentyFour Puzzle', in Proc. Thirteenth National Conference on Artificial Intelligence (AAAI-96), pp. 1202–1207. Menlo Park, CA: AAAI Press / The MIT Press, (1996).

[10] J.B.H. Kwa, 'BS : An Admissible Bidirectional Staged Heuristic Search Algorithm', Artificial Intelligence, 38(2), 95–109, (1989).

[11] E.L. Lawler and D. Wood, 'Branch-and-bound methods: a survey', Operations Research, 14(4), 699–719, (1966).

[12] G. Manzini, 'BIDA*: an improved perimeter search algorithm', Artificial Intelligence, 75(2), 347–360, (1995).

[13] I. Pohl, 'Bi-directional search', in Machine Intelligence 6, pp. 127–140, Edinburgh, (1971). Edinburgh University Press.

[14] A. Reinefeld and T.A. Marsland, 'Enhanced iterative-deepening search', IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 16(12), 701–709, (July 1994).

[15] L.A. Taylor and R.E. Korf, 'Pruning duplicate nodes in depth-first search', in Proc. Eleventh National Conference on Artificial Intelligence (AAAI-93), pp. 756–761. Menlo Park, CA: AAAI Press / The MIT Press, (1993).

[16] W. Zhang and R.E. Korf, 'Depth-first vs. best-first search: new results', in Proc. Eleventh National Conference on Artificial Intelligence (AAAI-93), pp. 769–775. Menlo Park, CA: AAAI Press / The MIT Press, (1993).