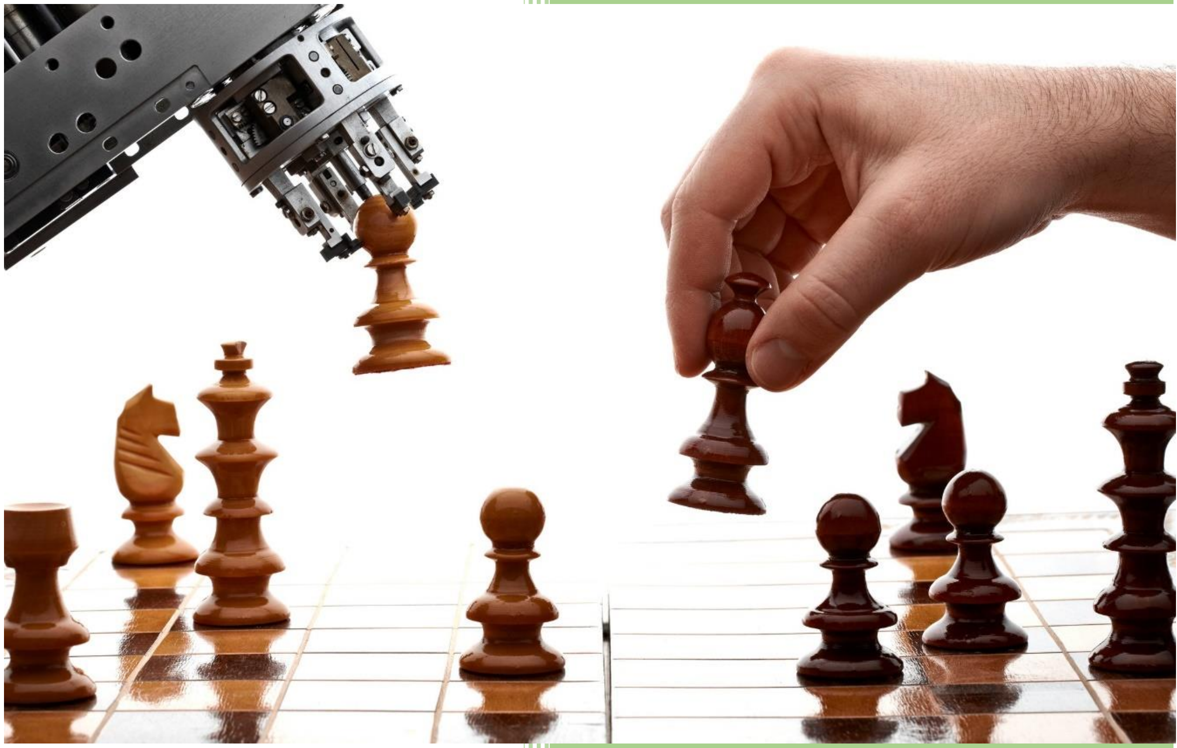
2017

EE586 AI - Take Home Exam



Seyit Yiğit SIZLAYAN-1876861

METU

11/29/2017

Table of Contents

Introduction

This project consists of puzzle solving problem with given parameters. For this project, I have chosen Matlab as programming language to be able to generate GUI more easily.

In the program, the instructor wanted us to implement the algorithms given below:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)
3. Iterative Deepening DFS (IDDFS)
4. A* Heuristic Search

Some algorithms given above has some problem by nature for specific cases. I have added additional functions to overcome these problems which are:

- Memorizing Breadth First Search (BFS_withMemory)
- Memorizing, Depth Limited DFS (DFS_limitedDepth)

I will explain them in detail in further parts.

Data Structures

Puzzle → NodeClass

To represent a puzzle, I have turned it to its row version and used "-1" to represent blank. For example

$$\begin{bmatrix} 3\,4\,6 \\ 1\quad 8 \\ 7\,2\,5 \end{bmatrix} \rightarrow [3\ 4\ 6\ 1\ -1\ 8\ 7\ 2\ 5]$$

This row is stored as a NodeClass, which is user defined class, in Matlab. It has properties of:

- BackPointer → Holds predecessor, required for the map from the bottom
- State → Holds state of the puzzle
- Depth → Holds position of from the top, required for DFS, IDDFS and A*
- HeuristicScore → Holds the heuristic score for A*

And methods of:

- Constructor → Takes a vector and turns it to NodeClass
- successor_withoutHeuristic → Takes the current state and returns successors list as NodeClass, heuristic is not calculated
- successor_withHeuristic → Takes the current state and returns successors list with calculated heuristic scores with given heuristic function
- heuristicMisplaced → Misplaced tile number given function
- manhattanDistance → Manhattan distance for each entry in the puzzle

Stack

It was required for the DFS, DFS_limitedDepth and IDDFS implementation. It is simple FILO stack class written in Matlab. It has push and pop functions. Stores the values in an array given type of array.

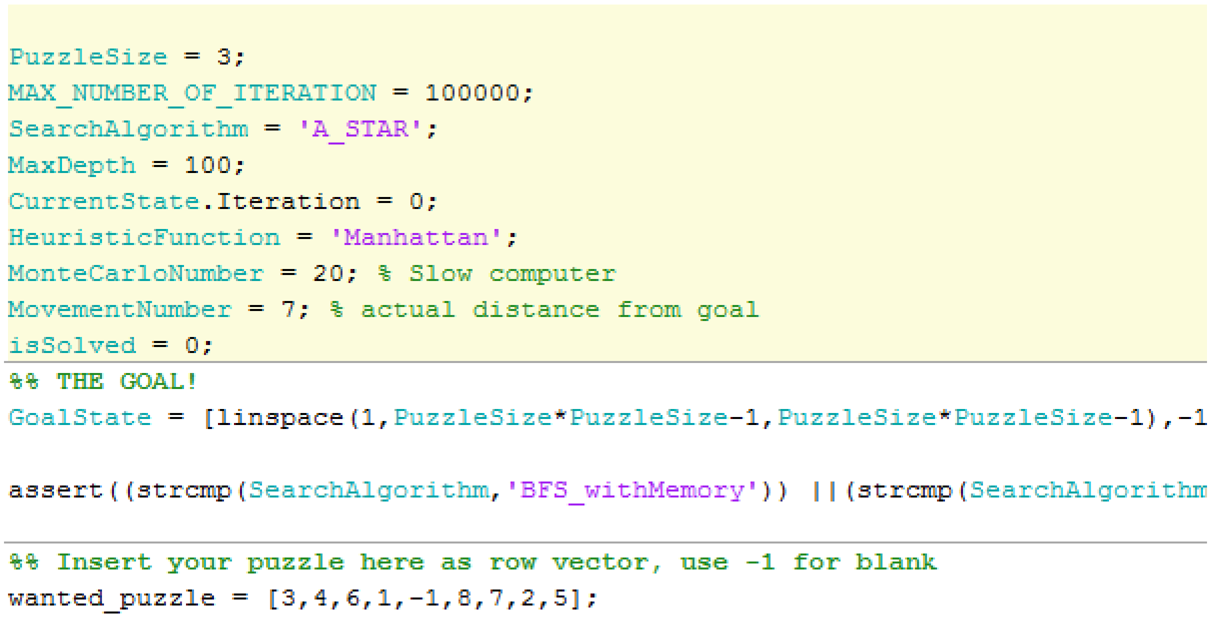
Queue

It was required for BFS_withMemory and BFS_withOutMemory implementation. It is simple FIFO queue class written in Matlab. It has push and pop functions. Stores the values in an array given type of array.

GUI

Due to unfortunate events happening in last 2 weeks, I was not able to implement beautiful GUI.

However, the wanted properties for the project still can be determined using "globals.m" formatted file.



```
PuzzleSize = 3;
MAX_NUMBER_OF_ITERATION = 100000;
SearchAlgorithm = 'A_STAR';
MaxDepth = 100;
CurrentState.Iteration = 0;
HeuristicFunction = 'Manhattan';
MonteCarloNumber = 20; % Slow computer
MovementNumber = 7; % actual distance from goal
isSolved = 0;
%% THE GOAL!
GoalState = [linspace(1,PuzzleSize*PuzzleSize-1,PuzzleSize*PuzzleSize-1),-1

assert((strcmp(SearchAlgorithm,'BFS_withMemory')) ||(strcmp(SearchAlgorithm

%% Insert your puzzle here as row vector, use -1 for blank
wanted_puzzle = [3,4,6,1,-1,8,7,2,5];
```

As you can see from the file snapshot given above, one can determine:

- Puzzle size by changing PuzzleSize variable
- Algorithm by changing SearchAlgorithm variable
  - Possible Values
    - BFS_withMemory
    - DFS_limitedDepth
    - A_STAR
    - IDDFS

Since other algorithms takes too much time to solve, I have limited the functions as this. But:

For standard BFS, one can use:

>> globals

>> [map, elapsed_time, visitedNodeNumber, totalNodes] = BFS_withoutMemory(initial_node,heuristic);

>> displayMap(map, elapsed_time, totalNodes);

For standard DFS, one can use:

>> globals

```
>> [ map, elapsed_time, visitedNodeNumber, totalNodes ] = DFS (initial_node,heuristic);
```

```
>> displayMap(map, elapsed_time, totalNodes);
```

- Heuristic function by changing HeuristicFunction variable
    - Possible Values
        - Misplaced
        - Manhattan
- Maximum depth for DFS_limitedDepth by changing MaxDepth variable

For Monte Carlo Simulations:

- Total number of Monte Carlo simulation for specific distance from goal state by changing MonteCarloNumber variable
- Maximum number of movements from goal state by changing MovementNumber variable

→To give a program specific puzzle, enter the row version of the puzzle to wanted_puzzle variable, then run:

```
>>solveGivenProblem
```

This will give performance metrics and map of given problem at the terminal screen

→To run Monte Carlo simulations with given parameters, run

```
>>monteCarlo
```

This will give 3D bar-plots of performance metrics for randomly generated problem. The results will be explained in more details in Monte Carlo Simulations part.

Algorithms

During the implementations of the algorithm, I have tried to avoid for loops as much as I can since I was working in Matlab. Its built-in functions like “ismember” or “isequal” makes the program run faster.

Breadth-First Search (With Memory)

Since problem have loops in it, I have kept the visited node list to avoid loops. This makes the problem faster around 100 times for given problem.

Answers:

a. 1749 nodes are opened
b. 12 movement has been done to reach goal.
c. It takes around 990-1010ms for each trial.

Depth-First Search

It solved the problem in its simplest form in my case but took around 8-12 minutes for one problem. To fix that, I have added depth limitation to the problem. Since the solution is close to the start for given problem, small numbers (14-16) of depth limitation gave good results. However, large numbers(80-100) result in 3 minutes of run-time.

a. Around 15000 depending on the given depth limit. Smaller the depth limit, smaller the opened-node. For values less than 13, the algorithm could not solve the problem.
b. Solution for the given problem took 1000 movement to reach the goal for 80-depth-limit.
    - It is far beyond optimal but I have checked if it has loops. There was no loops in the solution.
c. It tooks around 3 minutes, again 80-depth-limit.

Iterative-Deepening Seach

I have implemented as iterative-deepening depth-first-search. To gain a little time, I have stored the nodes which can be opened but left due to the depth limitation. The stored nodes are opened at the next iteration.

a. During the process, since I have implemented it as a function, at each iteration, all the variables are cleaned. So I have recorded only the maximum number of nodes stored in the memory(changes for each iteration, does not require cumulation) and number of opened nodes(cumulated after each step).

   Maximum Nodes stored in the stack was 2202.

   Total opened Nodes was 5799.

b. It have found the optimal solution which is 12 movement.
c. It took 1.3 to 1.45 seconds for each trial.

A*-heuristic Search

I have implemented the A* algorithm using the pseudo code given in the [1]. The pseudo code was:

```
      // A*
      initialize the open list
1:    initialize the closed list
```

```
put the starting node on the open list (you can leave its **f** at zero)

while the open list is not empty
    find the node with the least **f** on the open list, call it "q"
    pop q off the open list
    generate q's 8 successors and set their parents to q
    for each successor
        if successor is the goal, stop the search
        successor.g = q.g + distance between successor and q
        successor.h = distance from goal to successor
        successor.f = successor.g + successor.h

        if a node with the same position as successor is in the OPEN
list \
            which has a lower f than successor, skip this successor
        if a node with the same position as successor is in the CLOSED
list \
            which has a lower f than successor, skip this successor
        otherwise, add the node to the open list
    end
    push q on the closed list
end
```

In this implementation, instead of priority-queue, there was open and closed lists. Since there was no priority queue, it must find smallest heuristic for each iteration. I preferred it since otherwise the program will handle large memory copy problem to place a node in a priority-queue which is hard task for matlab.

a. As expected, A* gave the best results. As discussed in the Monte Carlo Simulations Part, its both memory and time metrics was the top. It also could fine the optimal solution.
   The results for “Manhattan Distance” heuristic function was better than the “Misplaced tiles” heuristic function.
b. The functions are implemented as method of the NodeClass class.
c. Results:
    a. Total of 188 nodes are opened and there was up to 534 nodes in the memory (total of open and closed list)
    b. It has found the optimal solution of 12 movements.
    c. 0.6-0.7 second which is the best among others.

Monte Carlo Simulations

How to run Monte Carlo Simulations are give in the GUI part. Since my computer was not too fast, I have run the simulations for 20 puzzles for each distance between 1 and 9. I preferred 9 because I am working in matlab, it is slower by the nature. If I had implemented the algorithms on C++, they would be faster.

The simulation works like this:

1) Create a NodeClass matrix consisting of movement number times Monte Carlo simulation number.
2) Each row requires maximum of row-number movements to reach the goal.
3) All puzzles in the matrix are solved by ‘BFS_withMemory’, ‘IDDFS’ and ‘A-Star with Manhattan Distance’ algorithms, one by one.
4) The elapsed times, maximum opened nodes and maximum nodes stored in the memory is kept as array for each algorithm.
5) The results are plotted on 3D-Stacked-Bar graph for better visualization.

➔ Since DFS even with its modified version, takes too much to compute (3 minutes for each puzzle). Because of that, I have not run the Monte Carlo for the DFS algorithm.

Results

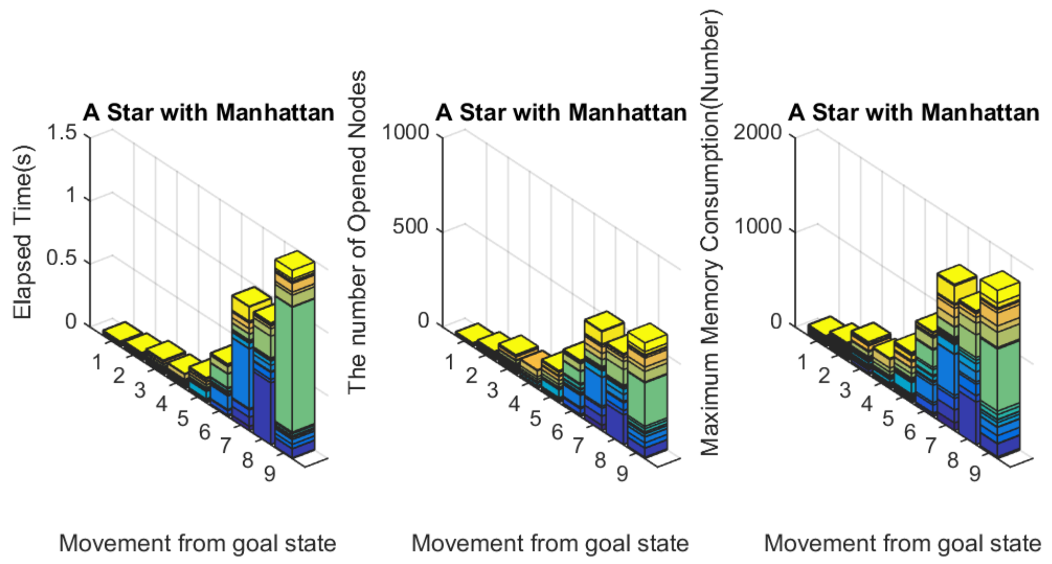


Figure 1: A-Star with Manhattan for 3X3 puzzle

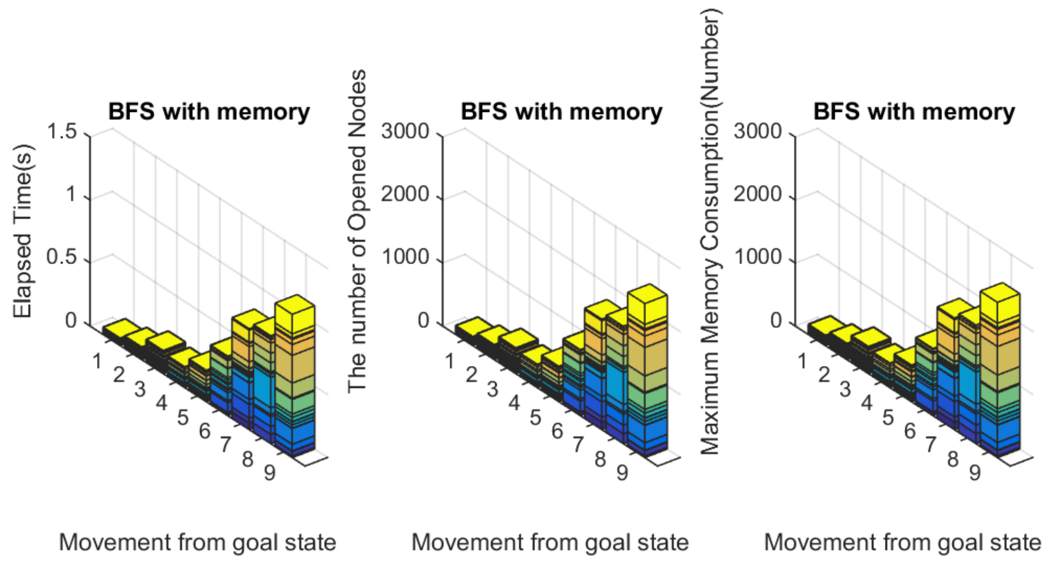


Figure 2: BFS with memory for 3X3 puzzle

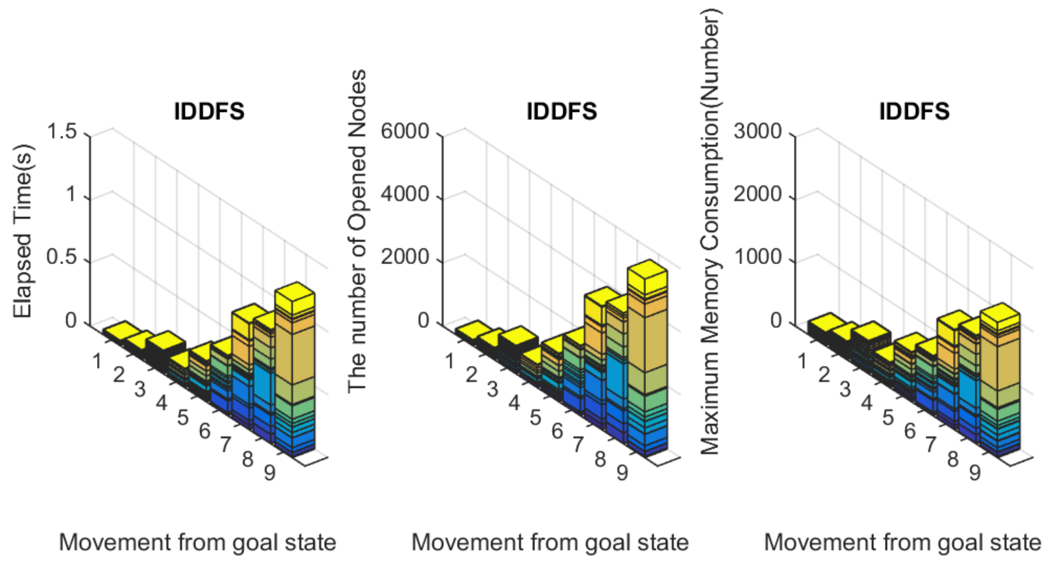


Figure 3: IDDFS with 3X3 puzzle

As one can easily see, the better performance is taken from A* algorithm. Also with the increased number of distances, the required memory and process time increases for all three cases. The least aggressive increase also seems on A*.

Results for larger problems:

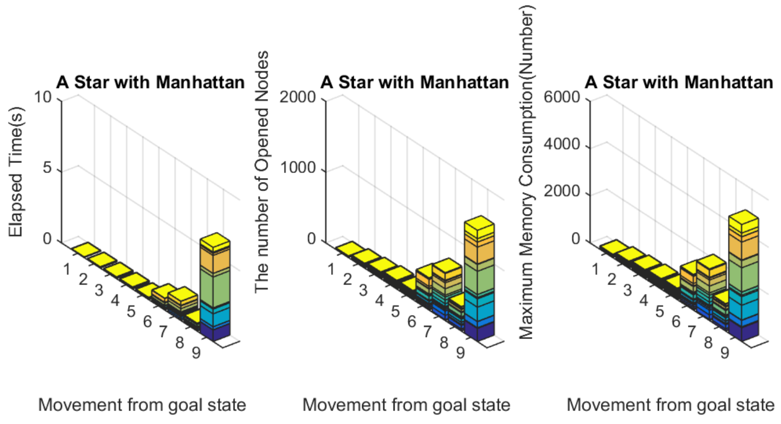


Figure 4: A 4X4*

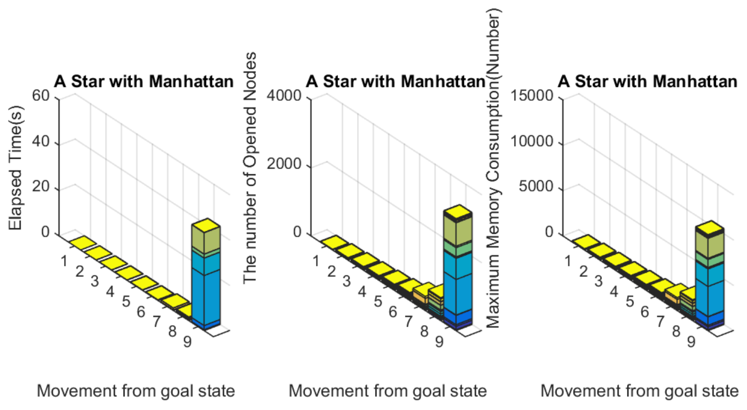


Figure 5: A 5X5*

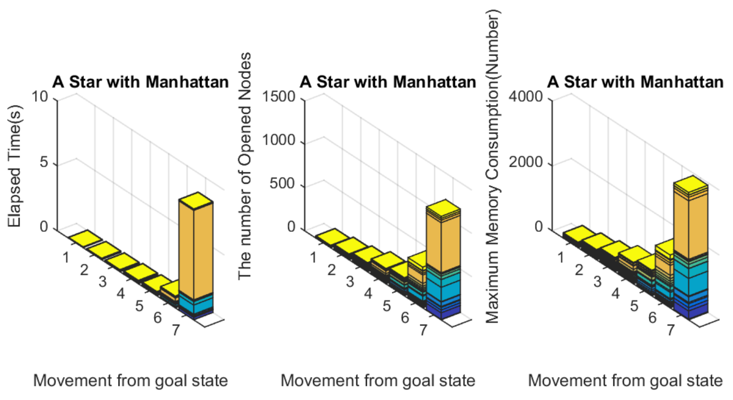


Figure 6: A 7X7 (movement is up to 7)*

In A* case, the memory does not increase aggressively, just doubles itself at most.

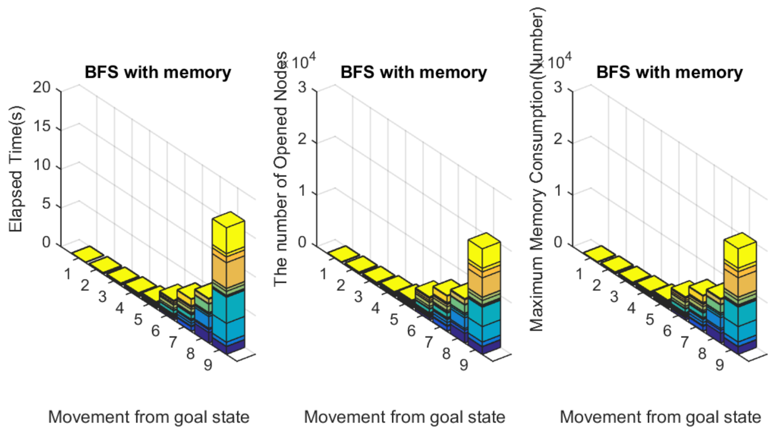


Figure 7: BFS for 4X4

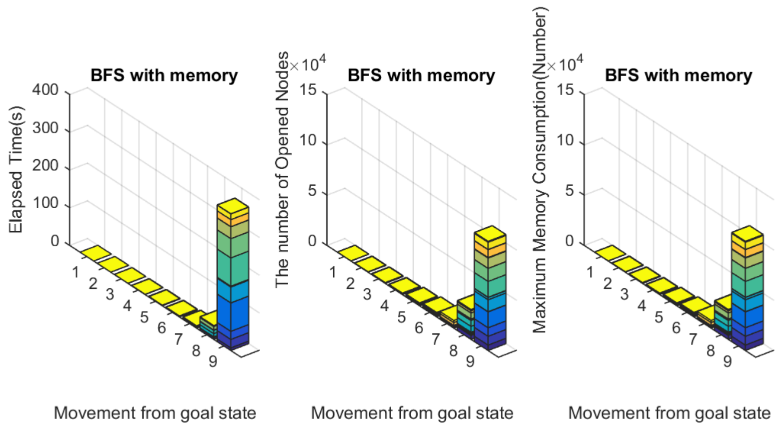


Figure 8: BFS for 5X5

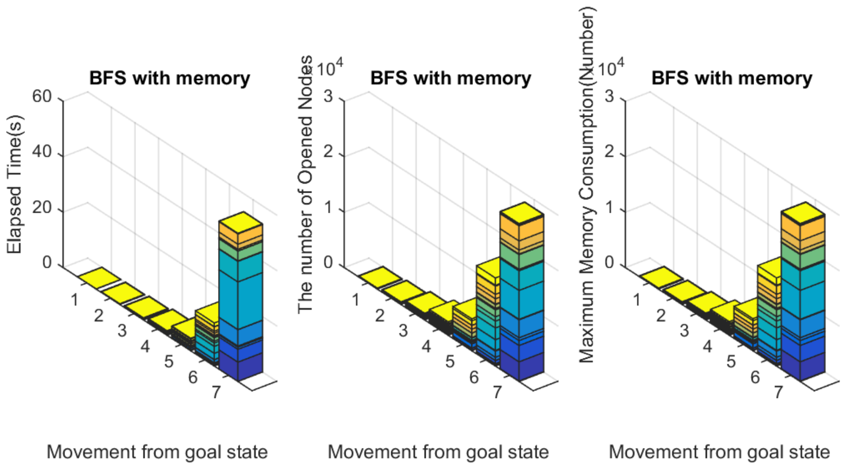


Figure 9: BFS for 7X7 (movement is up to 7)

In BFS case, both process time and memory requirements almost larger 10 times with increased number.

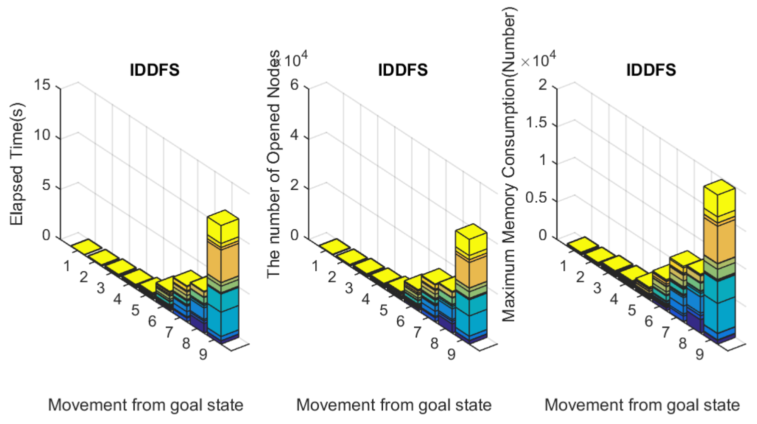


Figure 10: IDDFS for 4X4

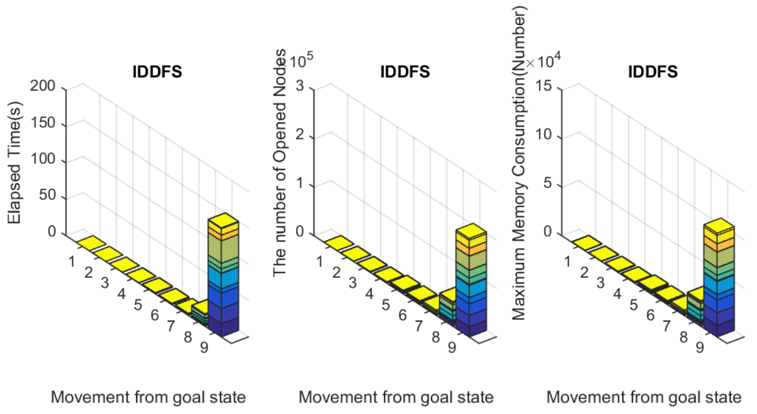


Figure 11: IDDFS for 5X5

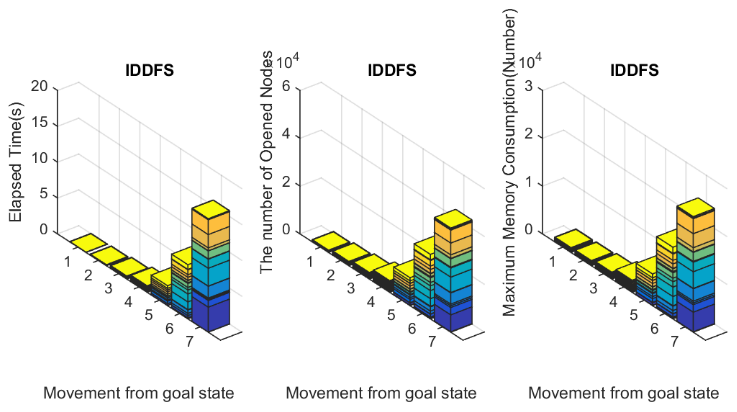


Figure 12: IDDFS for 7X7 (movement is up to 7)

As one can easily see, the most aggressive increase in both memory and process time requirements happens in IDDFS algorithm.

References

[1] R. Eranki, "Pathfinding using A* (A-Star)," 2002. [Online]. Available: http://web.mit.edu/eranki/www/tutorials/search/. [Accessed 28 11 2017].