Middle East Technical University

EE 586

Artificial Intelligence

Report for Programming Exercise

Author:
Murat Kumru

Student Number:
1626449

29.11.2017

1 Answers

1 First of all, the whole program is written in Matlab environment. All related information used to describe a puzzle configuration is stored in a data structure called *Node*. Considering remarkable computational advantages in manipulation of matrices and vectors by Matlab, *Nodes* are implemented to be just column vectors with specified order of elements. To illustrate, first *n entities* of a *Node* vector correspond to the numbers on the tiles stored in an ordered fashion. This portion of a *Node* is called *State*. In state vector, 0 (zero) is used as a placeholder for the free spot in the puzzle. The rest of the vector comprises of a *nodeID* (which is unique to each node), *parentID* (serving as a back pointer) and *cost* (to reach that node) as illustrated in Fig. 1. Note that, if A* search algorithm is to be employed, than *Node* vector is appended by a heuristic value and a scalar obtained by adding the heuristic and the cost.

```
if strcmp(method, 'A*')
    startNode = [startState; 1; 0; 0; 0; 0];
    % = [state; nodeID; predecessorID; cost; heuristic; f]
else
    startNode = [startState; 1; 0; 0];
    % = [state; nodeID; predecessorID; totalCost]
end
```

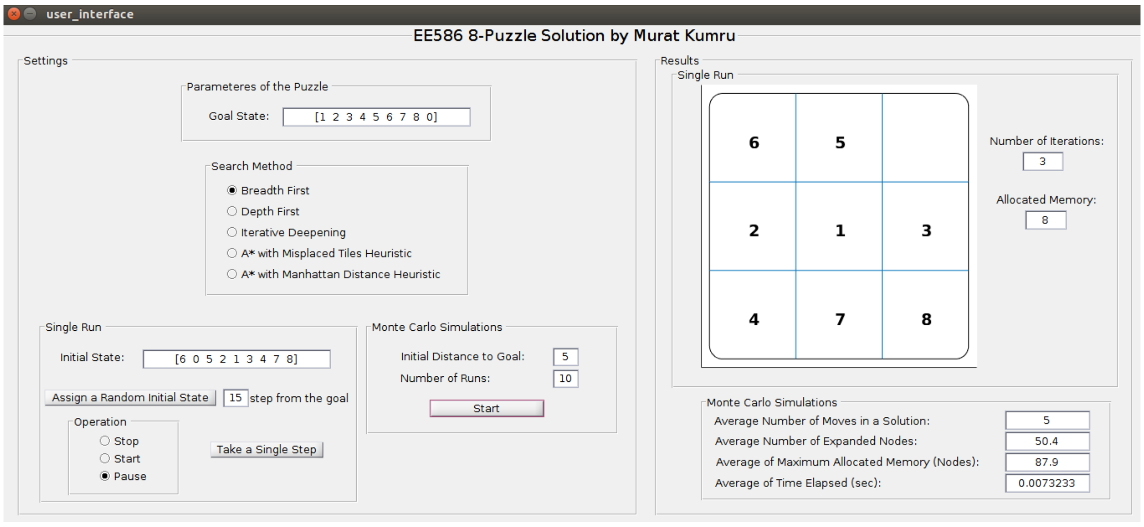
Figure 1: An example code snippet initializing a Node called *startNode*

Furthermore, *"previously visited"* and *"to be explored"* Nodes are also stored in matrices by simply concatenating related *Nodes*.

2 To launch Graphical User Interface (GUI), *user_interface.m* in the project directory is to be opened and run in Matlab environment. Fig. 2 and Fig. 3 depict snapshots taken during different simulations. Via the GUI, a user can

- select the search algorithm to be used,
- track instantaneous state of the puzzle through free-run simulations,
- set any desired goal state,
- set an initial state or choose to assign a random one,
- start/stop/pause a free-run simulation,
- carry out single step iterations (Note that, in this mode, the solution is not obtained before presenting the result to the user. On the contrary, while the algorithm attempts to solve the problem, it simultaneously informs the user about configuration.)

- perform Monte Carlo simulations with desired properties.



user_interface

EE586 8-Puzzle Solution by Murat Kumru

Settings

Parameteres of the Puzzle
Goal State: [1 2 3 4 5 6 7 8 0]

Search Method
- (●) Breadth First
- ( ) Depth First
- ( ) Iterative Deepening
- ( ) A* with Misplaced Tiles Heuristic
- ( ) A* with Manhattan Distance Heuristic

Single Run
Initial State: [6 0 5 2 1 3 4 7 8]
Assign a Random Initial State | 15 step from the goal
Operation: ( ) Stop ( ) Start (●) Pause
Take a Single Step

Monte Carlo Simulations
Initial Distance to Goal: 5
Number of Runs: 10
Start

Results

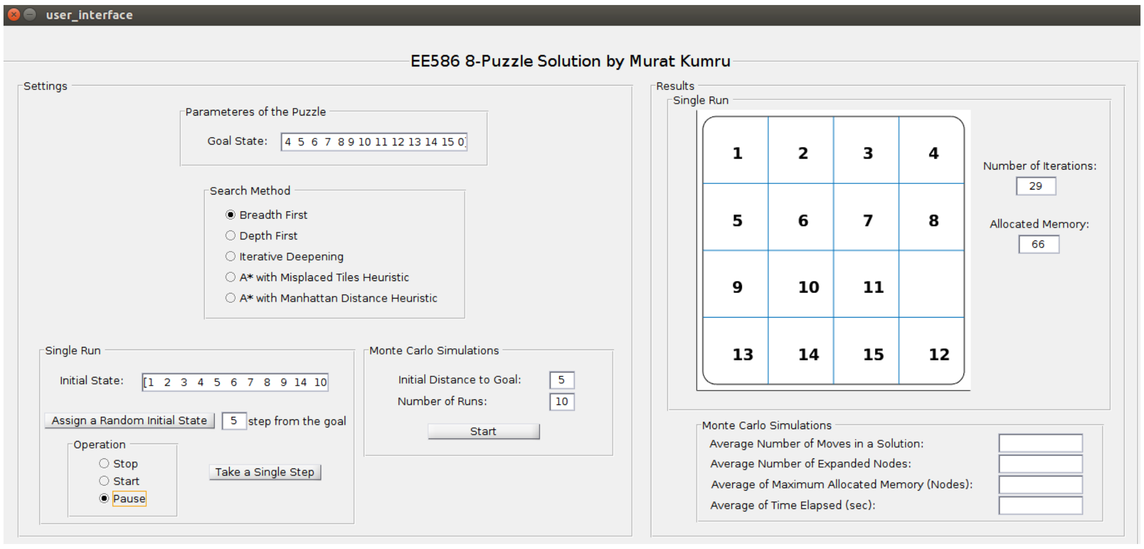
Single Run

6	5	
2	1	3
4	7	8

Number of Iterations: 3
Allocated Memory: 8

Monte Carlo Simulations
Average Number of Moves in a Solution: 5
Average Number of Expanded Nodes: 50.4
Average of Maximum Allocated Memory (Nodes): 87.9
Average of Time Elapsed (sec): 0.0073233

Figure 2: A view from GUI when the puzzle to be solved is 3-by-3



user_interface

EE586 8-Puzzle Solution by Murat Kumru

Settings

Parameteres of the Puzzle
Goal State: 4 5 6 7 8 9 10 11 12 13 14 15 0]

Search Method
- (●) Breadth First
- ( ) Depth First
- ( ) Iterative Deepening
- ( ) A* with Misplaced Tiles Heuristic
- ( ) A* with Manhattan Distance Heuristic

Single Run
Initial State: [1 2 3 4 5 6 7 8 9 14 10
Assign a Random Initial State | 5 step from the goal
Operation: ( ) Stop ( ) Start (●) Pause
Take a Single Step

Monte Carlo Simulations
Initial Distance to Goal: 5
Number of Runs: 10
Start

Results

Single Run

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

Number of Iterations: 29
Allocated Memory: 66

Monte Carlo Simulations
Average Number of Moves in a Solution:
Average Number of Expanded Nodes:
Average of Maximum Allocated Memory (Nodes):
Average of Time Elapsed (sec):

Figure 3: A view from GUI when the puzzle to be solved is 4-by-4

3 The function named as *successors()* can be found in the project directory. It has the ability to handle *L-puzzle* case. A snapshot of the implementation is included in the Appendix section.

4 For the initial state given in Fig. 4, the program is run with *Breadth First Search(BFS)* algorithm.

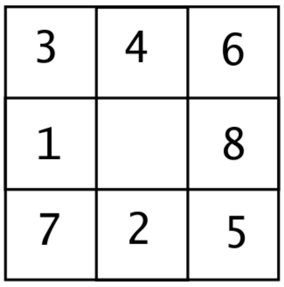


Figure 4: Initial state of the puzzle

(a) 2230 distinct *Nodes* have been visited to find the optimum sequence of the puzzle.

(b) There are 12 tile moves in the optimum sequence.

(c) It takes about 0.85 seconds for the algorithm to end up with the solution.

5 In this step, *Depth First Search (DFS)* is considered.

Depth First Search is implemented with an addition of a simple trick. Stemmed from the nature of this puzzle problem, it is possible to run into the same nodes in separate branches of search. This "problem" is tackled by not discarding a previously visited successor immediately, instead it is modified to compare the current version of the successor the previous one in terms of their costs. However, the implementation do not end up with satisfying results. It took about 1.5 hour for the program to throw an out of memory error when the number of the explored nodes is around 90000. On the other hand, the procedure followed by the algorithm is verified to be consistent with the expectations coming from theoretical analysis of the algorithm.

6 In this step, *Iterative Deepening Search (IDS)* is considered. IDS is built by giving the DFS an incremental control mechanism of maximum allowed depth. The resultant observations are as follows:

(a) 2074 distinct *Nodes* have been visited to find the optimum sequence of the puzzle.

(b) There are 12 tile moves in the optimum sequence.

(c) It takes about 2.5 seconds for the algorithm to end up with the solution.

7 (a) Making use of additional information provided by a heuristic function about *Nodes* position with regard to the goal, *A** is expected to present computational advantage over aforementioned algorithms. It essentially makes *educated* guesses while selecting the next *Nodes* to be explored. Therefore, the more informative or explicit the heuristic function gets, the better performance is to be captured. With this in mind, Manhattan Distance serves a more comprehensive analysis of the problem.

(b) *heuristic_misplaced* and *heuristic_manhattan* functions are included in the directory.

(c) For two heuristic functions, following results are obtained:

Table 1: Comparison of two heuristic functions

	Heuristic: Misplaced	Heuristic: Manhattan
# of Moves	12	12
# of Explored Nodes	113	24
Elapsed Time (msec)	80.3	61.7

8 (a) Desired number of random states at distance d steps from the goal state are produced by *produce_random_puzzle.m* function. A snapshot of the function is included in Appendix section.

(b)-(g) For each search algorithm, Monte Carlo simulations with parameters $N = 20$, $True\ Distance = \{1,\ 3,\ 5,\ 10,\ 15\}$ are conducted. Note that due to the inconvenience in the operation of *Depth First Search*, it is excluded from simulations and comparisons.

Results are presented by Fig. 5- 8. In the figures, data marks indicate the points at which MC simulations are performed, interpolating lines are just added to be able to interpret the findings.

In Fig. 5, it is seen that all of the algorithms are able to find the sequence with minimum number of tile moves. Moreover, in some cases, the averaged number of moves are observed to be less than the corresponding true distance which brings questions into mind. This results from the production process of random initial states. Although we produce random samples by moving tiles in an iterative fashion and preventing any possible loops, it does not guarantee not to have any path reaching to the goal with less number of tile moves.

In Fig. 6-8, with increasing distance between the goal and initial states, the number of stored *Nodes* and time spent to process these nodes increase for all algorithms as expected. With respect to the dominance of exponential characteristic in the allocated memory and number of visited *Nodes*, the algorithms are listed as BFS, IDS, A* with Misplaced Heuristic and A* with Manhattan Heuristic in a descending order. A* is detected to be the best performing algorithm for all cases.

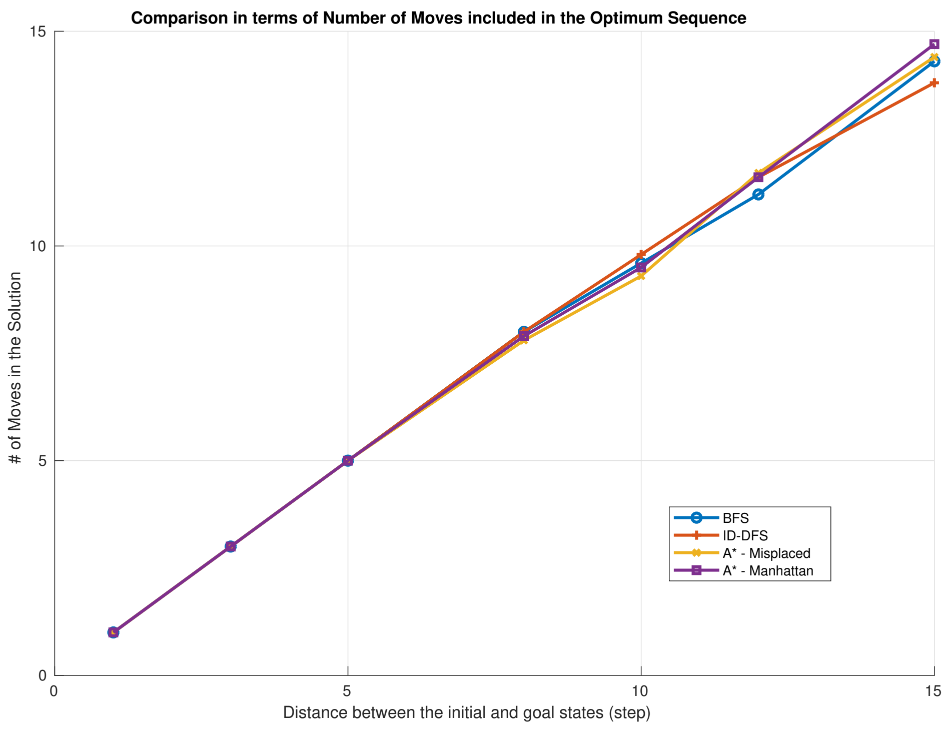


Figure 5: Comparison w.r.t. average number of moves in solutions

9 In this step, A* Search algorithm with Manhattan Distance heuristic is scrutinized for puzzles with various dimensions. For each size, random initial states are generated to be 15 steps away from the goal. Number of Monte Carlo simulations are 20 for all scenarios. Puzzle sizes of 3-by-3, 5-by-5, 8-by-8, 10-by-10, 12-by-12, 15-by-15, 20-by-20, 25-by-25, and 30-by-30 cases are examined. Corresponding results are revealed by Fig. 9. It is observed that the number of visited and queued *Nodes* does not depend on the size of the puzzle. On the other hand, the amount of utilized storage increases by increasing puzzle dimension since storing a *Node* of a rather large puzzle takes more memory compared to the smaller ones.

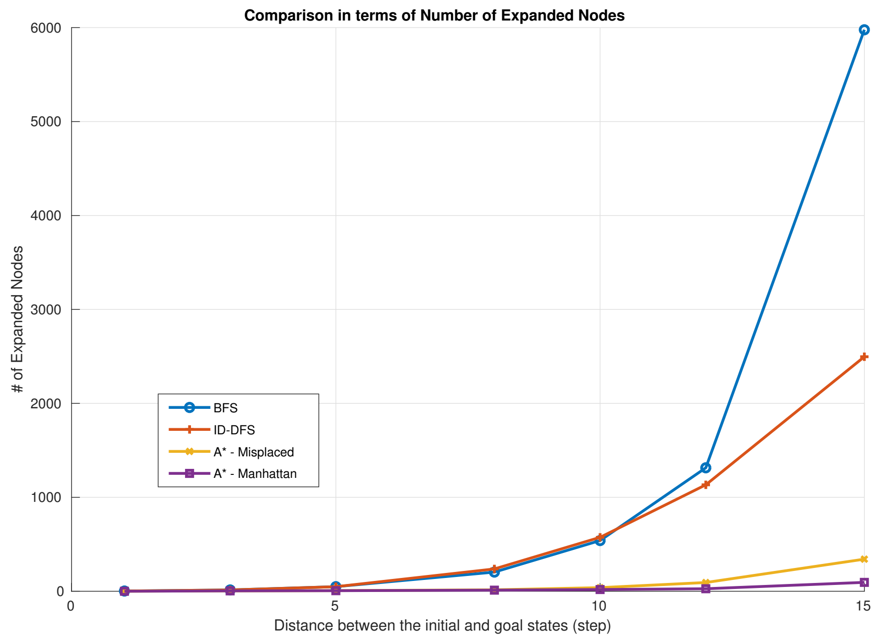


Figure 6: Comparison w.r.t. average number of visited node throughout process

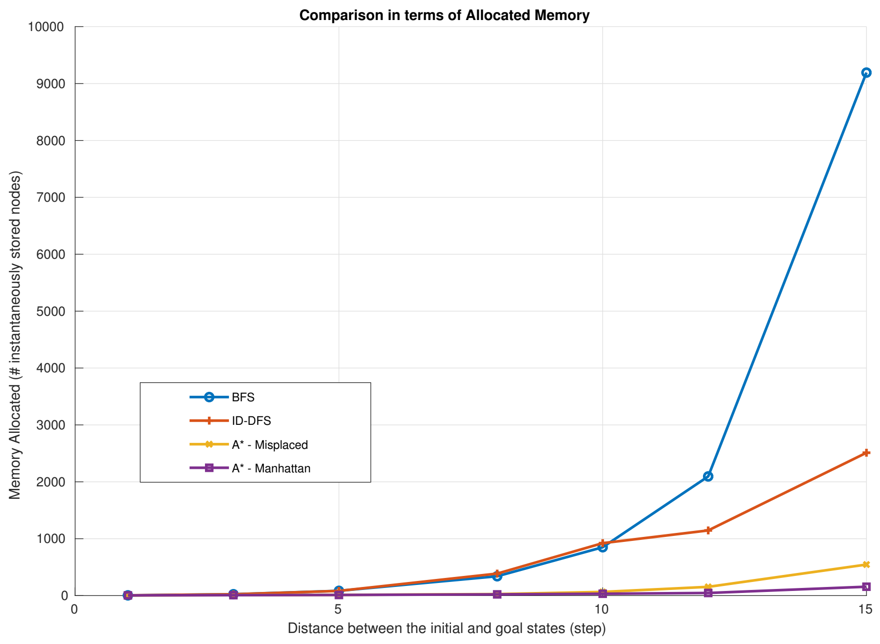


Figure 7: Comparison w.r.t. average allocated memory in terms of *Nodes*

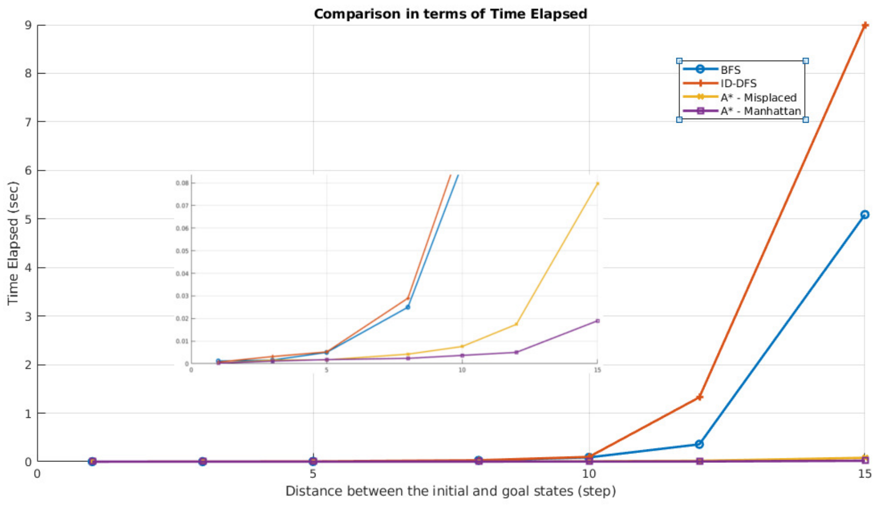


Figure 8: Comparison w.r.t. time spent for simulations

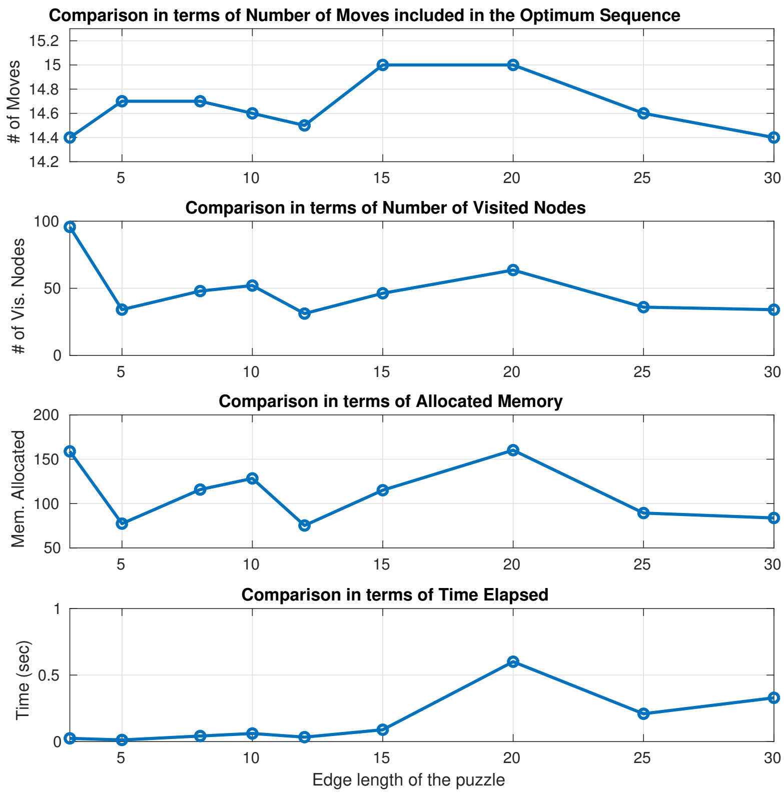
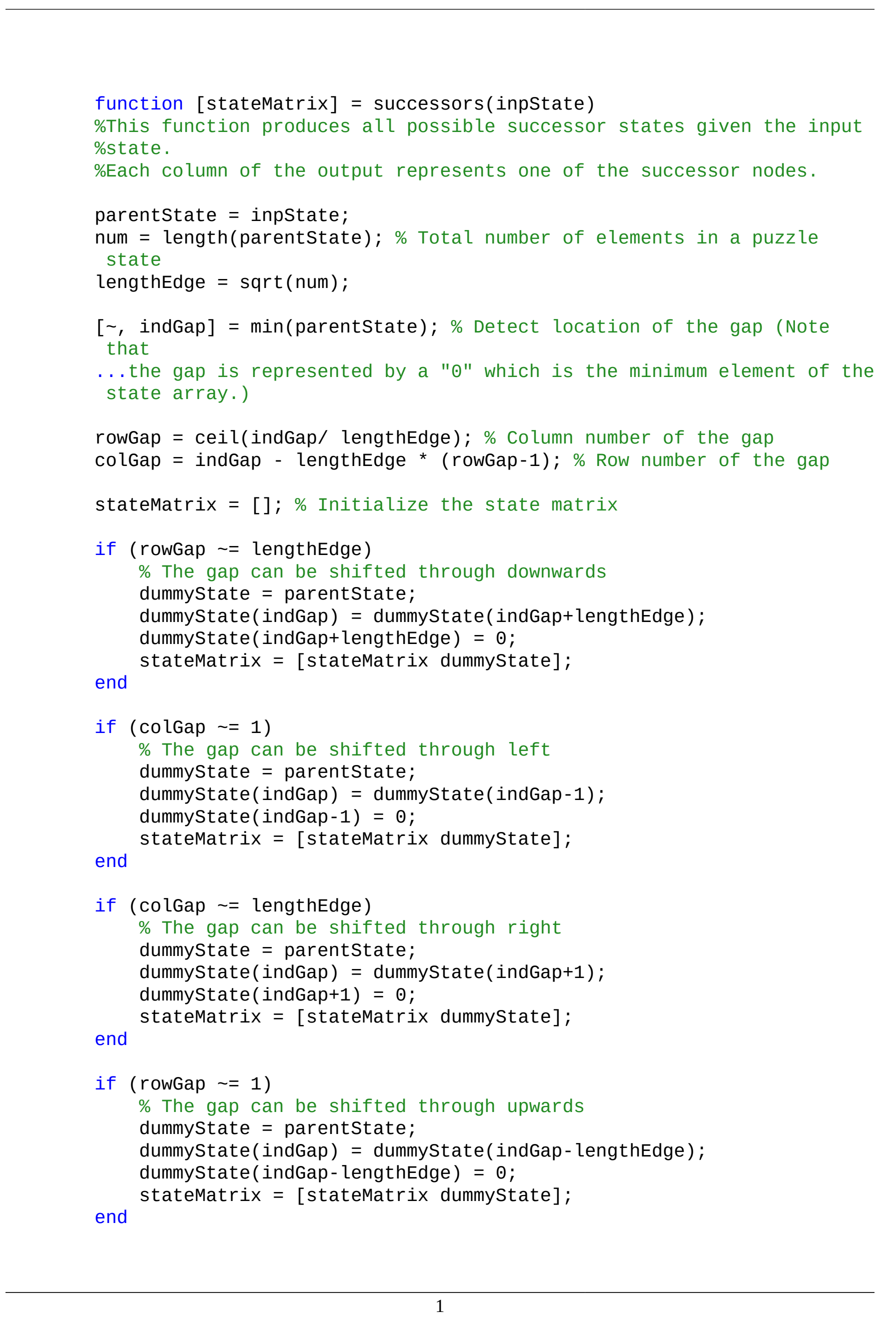


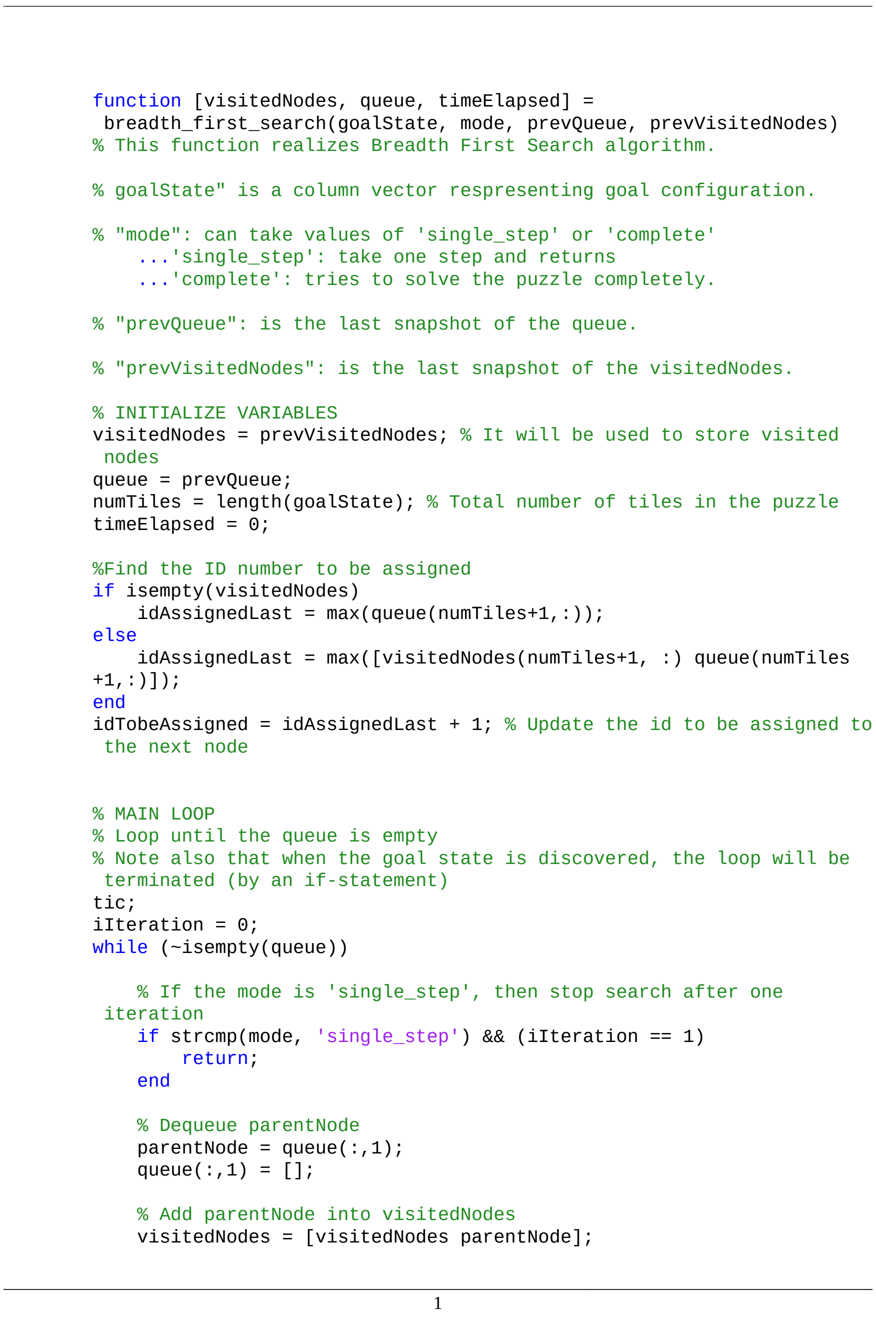
Figure 9: Performance analysis of A* for various puzzle dimensions



```
function [stateMatrix] = successors(inpState)
%This function produces all possible successor states given the input
%state.
%Each column of the output represents one of the successor nodes.

parentState = inpState;
num = length(parentState); % Total number of elements in a puzzle
 state
lengthEdge = sqrt(num);

[~, indGap] = min(parentState); % Detect location of the gap (Note
 that
...the gap is represented by a "0" which is the minimum element of the
 state array.)

rowGap = ceil(indGap/ lengthEdge); % Column number of the gap
colGap = indGap - lengthEdge * (rowGap-1); % Row number of the gap

stateMatrix = []; % Initialize the state matrix

if (rowGap ~= lengthEdge)
    % The gap can be shifted through downwards
    dummyState = parentState;
    dummyState(indGap) = dummyState(indGap+lengthEdge);
    dummyState(indGap+lengthEdge) = 0;
    stateMatrix = [stateMatrix dummyState];
end

if (colGap ~= 1)
    % The gap can be shifted through left
    dummyState = parentState;
    dummyState(indGap) = dummyState(indGap-1);
    dummyState(indGap-1) = 0;
    stateMatrix = [stateMatrix dummyState];
end

if (colGap ~= lengthEdge)
    % The gap can be shifted through right
    dummyState = parentState;
    dummyState(indGap) = dummyState(indGap+1);
    dummyState(indGap+1) = 0;
    stateMatrix = [stateMatrix dummyState];
end

if (rowGap ~= 1)
    % The gap can be shifted through upwards
    dummyState = parentState;
    dummyState(indGap) = dummyState(indGap-lengthEdge);
    dummyState(indGap-lengthEdge) = 0;
    stateMatrix = [stateMatrix dummyState];
end
```

1

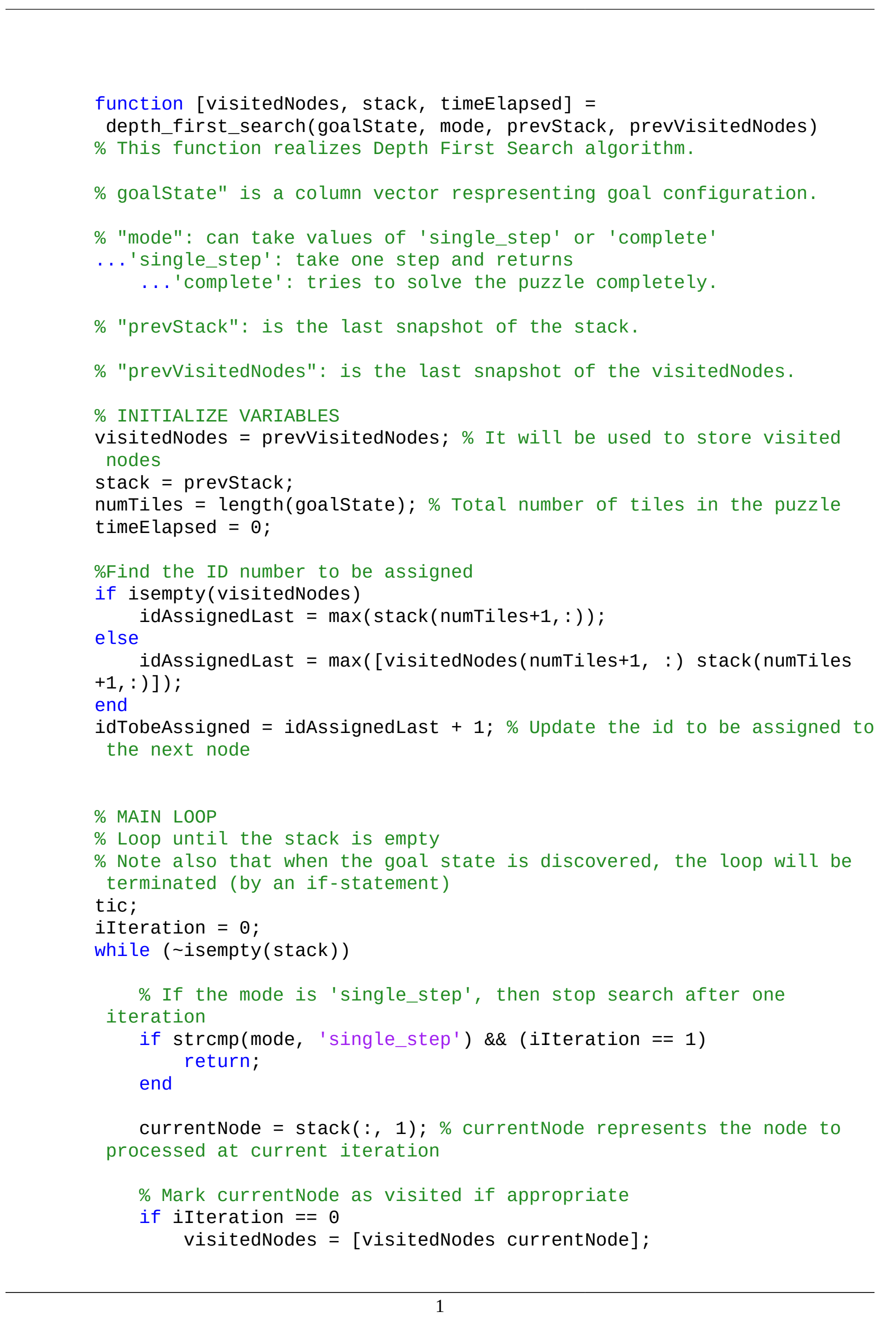
Figure 10: Implementation: *successors* function



```
function [visitedNodes, queue, timeElapsed] =
 breadth_first_search(goalState, mode, prevQueue, prevVisitedNodes)
% This function realizes Breadth First Search algorithm.

% goalState" is a column vector respresenting goal configuration.

% "mode": can take values of 'single_step' or 'complete'
    ...'single_step': take one step and returns
    ...'complete': tries to solve the puzzle completely.

% "prevQueue": is the last snapshot of the queue.

% "prevVisitedNodes": is the last snapshot of the visitedNodes.

% INITIALIZE VARIABLES
visitedNodes = prevVisitedNodes; % It will be used to store visited
 nodes
queue = prevQueue;
numTiles = length(goalState); % Total number of tiles in the puzzle
timeElapsed = 0;

%Find the ID number to be assigned
if isempty(visitedNodes)
    idAssignedLast = max(queue(numTiles+1,:));
else
    idAssignedLast = max([visitedNodes(numTiles+1, :) queue(numTiles
+1,:)]);
end
idTobeAssigned = idAssignedLast + 1; % Update the id to be assigned to
 the next node


% MAIN LOOP
% Loop until the queue is empty
% Note also that when the goal state is discovered, the loop will be
 terminated (by an if-statement)
tic;
iIteration = 0;
while (~isempty(queue))

    % If the mode is 'single_step', then stop search after one
 iteration
    if strcmp(mode, 'single_step') && (iIteration == 1)
        return;
    end

    % Dequeue parentNode
    parentNode = queue(:,1);
    queue(:,1) = [];

    % Add parentNode into visitedNodes
    visitedNodes = [visitedNodes parentNode];
```

1

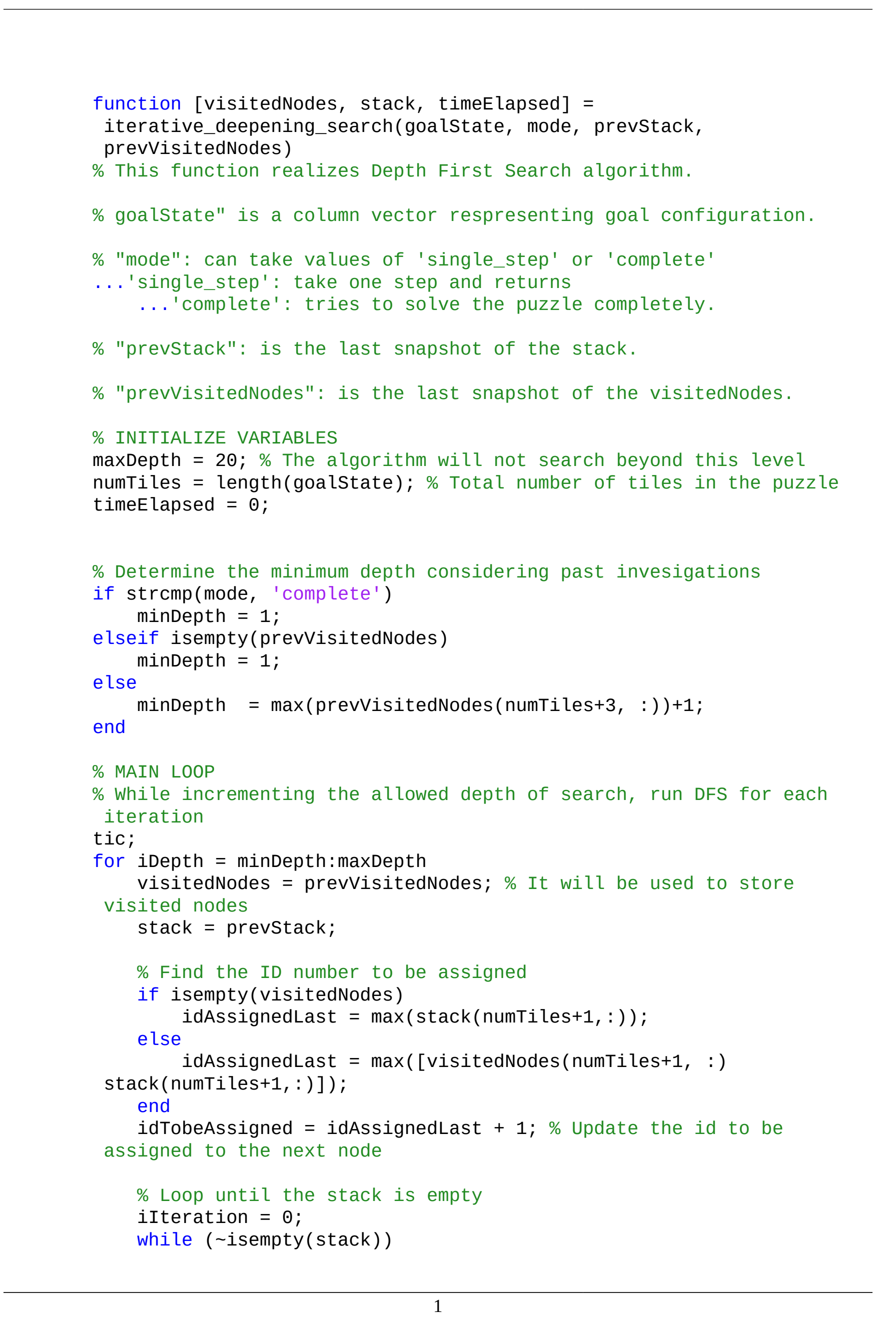
Figure 11: Implementation: *breadth_first_search* function



```
function [visitedNodes, stack, timeElapsed] =
 depth_first_search(goalState, mode, prevStack, prevVisitedNodes)
% This function realizes Depth First Search algorithm.

% goalState" is a column vector respresenting goal configuration.

% "mode": can take values of 'single_step' or 'complete'
...'single_step': take one step and returns
    ...'complete': tries to solve the puzzle completely.

% "prevStack": is the last snapshot of the stack.

% "prevVisitedNodes": is the last snapshot of the visitedNodes.

% INITIALIZE VARIABLES
visitedNodes = prevVisitedNodes; % It will be used to store visited
 nodes
stack = prevStack;
numTiles = length(goalState); % Total number of tiles in the puzzle
timeElapsed = 0;

%Find the ID number to be assigned
if isempty(visitedNodes)
    idAssignedLast = max(stack(numTiles+1,:));
else
    idAssignedLast = max([visitedNodes(numTiles+1, :) stack(numTiles
+1,:)]);
end
idTobeAssigned = idAssignedLast + 1; % Update the id to be assigned to
 the next node


% MAIN LOOP
% Loop until the stack is empty
% Note also that when the goal state is discovered, the loop will be
 terminated (by an if-statement)
tic;
iIteration = 0;
while (~isempty(stack))

    % If the mode is 'single_step', then stop search after one
 iteration
    if strcmp(mode, 'single_step') && (iIteration == 1)
        return;
    end

    currentNode = stack(:, 1); % currentNode represents the node to
 processed at current iteration

    % Mark currentNode as visited if appropriate
    if iIteration == 0
        visitedNodes = [visitedNodes currentNode];
```

1

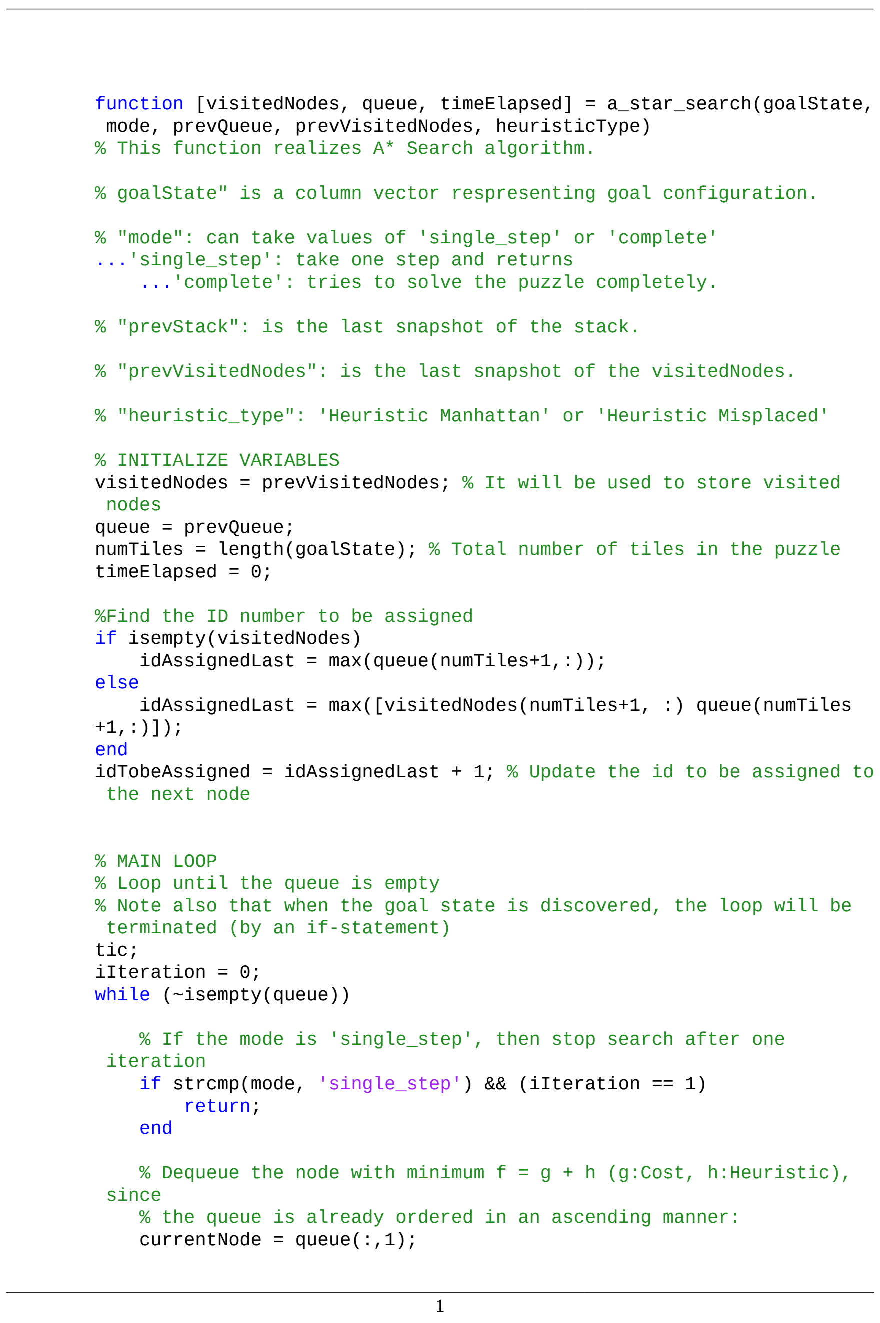
Figure 12: Implementation: *depth_first_search* function



```
function [visitedNodes, stack, timeElapsed] =
 iterative_deepening_search(goalState, mode, prevStack,
 prevVisitedNodes)
% This function realizes Depth First Search algorithm.

% goalState" is a column vector respresenting goal configuration.

% "mode": can take values of 'single_step' or 'complete'
...'single_step': take one step and returns
    ...'complete': tries to solve the puzzle completely.

% "prevStack": is the last snapshot of the stack.

% "prevVisitedNodes": is the last snapshot of the visitedNodes.

% INITIALIZE VARIABLES
maxDepth = 20; % The algorithm will not search beyond this level
numTiles = length(goalState); % Total number of tiles in the puzzle
timeElapsed = 0;


% Determine the minimum depth considering past invesigations
if strcmp(mode, 'complete')
    minDepth = 1;
elseif isempty(prevVisitedNodes)
    minDepth = 1;
else
    minDepth  = max(prevVisitedNodes(numTiles+3, :))+1;
end

% MAIN LOOP
% While incrementing the allowed depth of search, run DFS for each
 iteration
tic;
for iDepth = minDepth:maxDepth
    visitedNodes = prevVisitedNodes; % It will be used to store
 visited nodes
    stack = prevStack;

    % Find the ID number to be assigned
    if isempty(visitedNodes)
        idAssignedLast = max(stack(numTiles+1,:));
    else
        idAssignedLast = max([visitedNodes(numTiles+1, :)
 stack(numTiles+1,:)]);
    end
    idTobeAssigned = idAssignedLast + 1; % Update the id to be
 assigned to the next node

    % Loop until the stack is empty
    iIteration = 0;
    while (~isempty(stack))
```

1

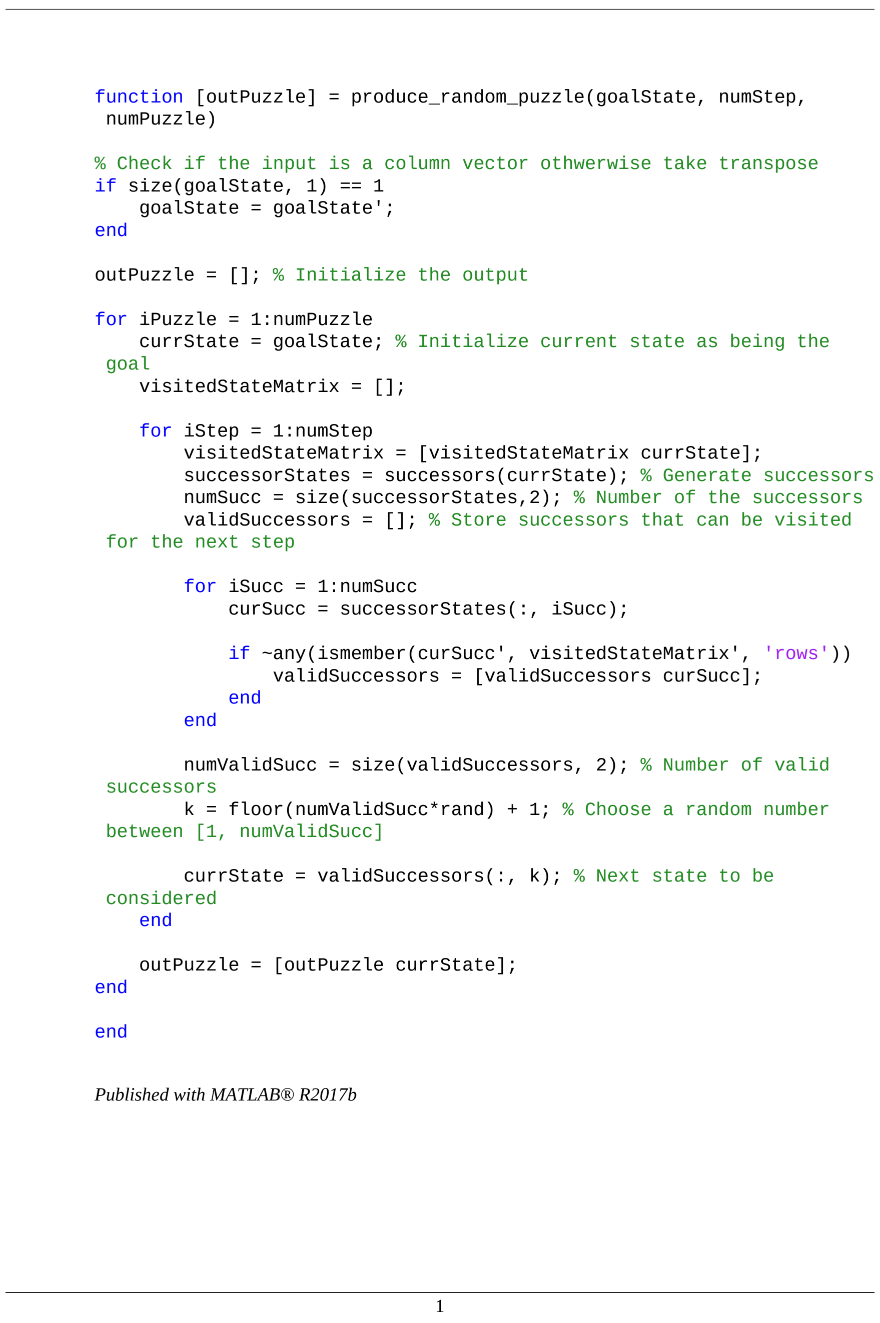
Figure 13: Implementation: *iterative_deepening_search* function



```
function [visitedNodes, queue, timeElapsed] = a_star_search(goalState,
 mode, prevQueue, prevVisitedNodes, heuristicType)
% This function realizes A* Search algorithm.

% goalState" is a column vector respresenting goal configuration.

% "mode": can take values of 'single_step' or 'complete'
...'single_step': take one step and returns
    ...'complete': tries to solve the puzzle completely.

% "prevStack": is the last snapshot of the stack.

% "prevVisitedNodes": is the last snapshot of the visitedNodes.

% "heuristic_type": 'Heuristic Manhattan' or 'Heuristic Misplaced'

% INITIALIZE VARIABLES
visitedNodes = prevVisitedNodes; % It will be used to store visited
 nodes
queue = prevQueue;
numTiles = length(goalState); % Total number of tiles in the puzzle
timeElapsed = 0;

%Find the ID number to be assigned
if isempty(visitedNodes)
    idAssignedLast = max(queue(numTiles+1,:));
else
    idAssignedLast = max([visitedNodes(numTiles+1, :) queue(numTiles
+1,:)]);
end
idTobeAssigned = idAssignedLast + 1; % Update the id to be assigned to
 the next node


% MAIN LOOP
% Loop until the queue is empty
% Note also that when the goal state is discovered, the loop will be
 terminated (by an if-statement)
tic;
iIteration = 0;
while (~isempty(queue))

    % If the mode is 'single_step', then stop search after one
 iteration
    if strcmp(mode, 'single_step') && (iIteration == 1)
        return;
    end

    % Dequeue the node with minimum f = g + h (g:Cost, h:Heuristic),
 since
    % the queue is already ordered in an ascending manner:
    currentNode = queue(:,1);
```

1

Figure 14: Implementation: *a_star_search* function



```
function [outPuzzle] = produce_random_puzzle(goalState, numStep,
 numPuzzle)

% Check if the input is a column vector othwerwise take transpose
if size(goalState, 1) == 1
    goalState = goalState';
end

outPuzzle = []; % Initialize the output

for iPuzzle = 1:numPuzzle
    currState = goalState; % Initialize current state as being the
 goal
    visitedStateMatrix = [];

    for iStep = 1:numStep
        visitedStateMatrix = [visitedStateMatrix currState];
        successorStates = successors(currState); % Generate successors
        numSucc = size(successorStates,2); % Number of the successors
        validSuccessors = []; % Store successors that can be visited
 for the next step

        for iSucc = 1:numSucc
            curSucc = successorStates(:, iSucc);

            if ~any(ismember(curSucc', visitedStateMatrix', 'rows'))
                validSuccessors = [validSuccessors curSucc];
            end
        end

        numValidSucc = size(validSuccessors, 2); % Number of valid
 successors
        k = floor(numValidSucc*rand) + 1; % Choose a random number
 between [1, numValidSucc]

        currState = validSuccessors(:, k); % Next state to be
 considered
    end

    outPuzzle = [outPuzzle currState];
end

end
```

Published with MATLAB® R2017b

1

Figure 15: Implementation: *produce_random_puzzle* function