Foundation of AI Coursework Part 1

Introduction

This report mentions my approach to the implementation of four tree search methods (depth-first search, breadth-first search, iterative deepening search, and a* search) for the "Blocksworld Tile Puzzle". The rules of the puzzle are introduced in [1]. Evidence will be provided to show that the implementation works as intended. On the other hand, there is an analysis of the scalability of the four methods and the limitations of the implementation.

Approach

Depth-First Search

There are two approaches for the depth first search. In the first approach, the agent moves randomly to prevent the search from stuck. The agent will move indefinitely until it reaches the goal state. Then, it will return the solution.

In the second approach, a depth limit is defined to prevent the search from going endlessly. First, a node stack will be constructed. After putting the root into the stack, it will pop the node on the top of the stack. If the node is not in the goal state and it has not reached the depth limit, it will be expanded and the expanded nodes will be put on the top of the stack. Because the stack is "last-in, first-out", the most recently expanded nodes, which are also the deepest nodes will be popped first. If the node has reached the depth limit already, it will not be expanded. It repeats all the steps above until it finds a node in the goal state, or there is no more node in the stack. Then, it will return the solution or none if no solution is found.

Breath-First Search

First, it constructs a node queue then put the root into the queue. After that, it will visit all the nodes in the queue. If it finds a node in the goal state, it will stop and return the solution. If there is no node in the goal state, pop all the nodes in the queue and expand them. This will continue until all nodes in the queue are popped and expanded. The expanded nodes will then be put into the queue. Since the all the nodes are popped, the expanded nodes will all have the same depth. Therefore, the search will visit all the nodes with the same depth first, then expand all the nodes and then visit the expanded nodes in the deeper depth. It repeats all the steps above until it finds a node in the goal state, or there is no more node in the queue. Then, it will return the solution or none if no solution is found.

Iterative-Deepening Search

It is similar to the depth first search but the depth limit will increase if no solution is found after all the nodes are popped. After putting the root into the stack, it will pop the node on the top of the stack. If the node is not in the goal state and it has not reached the depth limit, it will be expanded and the expanded nodes will be put on the top of the stack. Because the stack is "last-in, first-out", the most recently expanded nodes, which are also the deepest nodes will be popped first. If the node has reached the depth limit already, it will not be expanded. If it finds a node in the goal state, it will return the solution. If there is no more node, it will increase the depth limit by 1, put the root into the stack and restart searching.

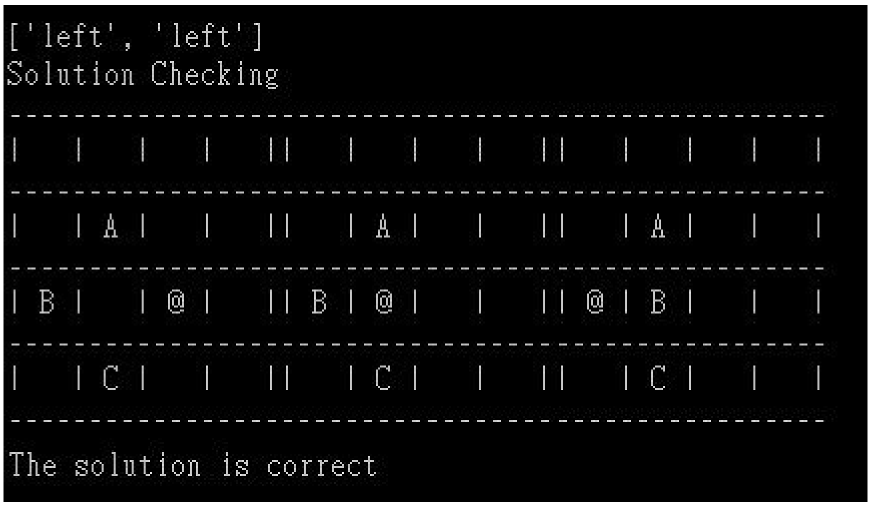
A* Search

For the A* Search, the heuristic algorithm will calculate the Manhattan distance between the position of A, B, C in the current state and the position of A, B, C in the goal state respectively. After that, it will sum up the Manhattan distance of A, B, and C and the depth of the node as the estimated cost. Like other searching algorithms, it constructs a node queue then put the root into the queue. Then, it will pop the node with the lowest estimated cost. If the popped node is not in the goal state, it will expand the popped node. The expanded node will then be put into the node queue. It keeps popping and expanding until it finds the solution or there is no more node.

Evidence

Solution Check

There is a function to check whether the solution found by the search is correct. The function checks whether it can reach the goal state from the start state by applying the solution.



```
['left', 'left']
Solution Checking
-------------------------------------------------
|   |   |   |   ||   |   |   |   ||   |   |   |   |
-------------------------------------------------
|   | A |   |   ||   | A |   |   ||   | A |   |   |
-------------------------------------------------
| B |   | @ |   || B | @ |   |   || @ | B |   |   |
-------------------------------------------------
|   | C |   |   ||   | C |   |   ||   | C |   |   |
-------------------------------------------------
The solution is correct
```

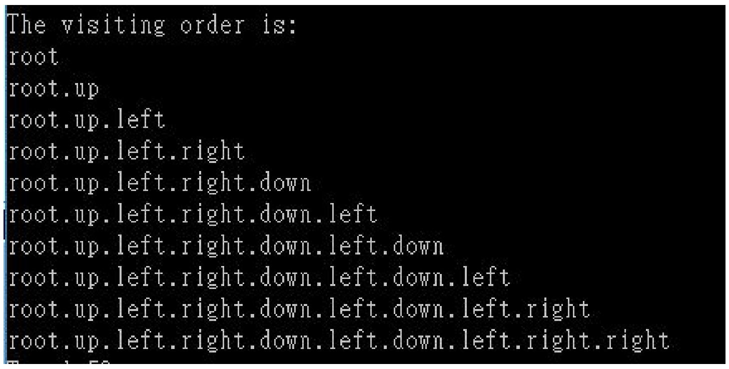
Figure 1. Solution Check Output

Visiting Order

The visiting order is recorded during the search to make sure it is performing the search wanted. Example output is provided below.

Randomized Depth-first Search

The search always visits the deepest node first and the moves are selected randomly.

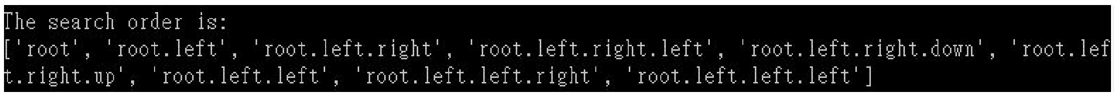


```
The visiting order is:
root
root.up
root.up.left
root.up.left.right
root.up.left.right.down
root.up.left.right.down.left
root.up.left.right.down.left.down
root.up.left.right.down.left.down.left
root.up.left.right.down.left.down.left.right
root.up.left.right.down.left.down.left.right.right
```

Figure 2. Visiting order of Randomized Depth-first Search

Depth-first Search with Depth Limit

The search visits the deepest node first. After it reaches the depth limit, it will stop expanding nodes and visit other nodes in the stack.

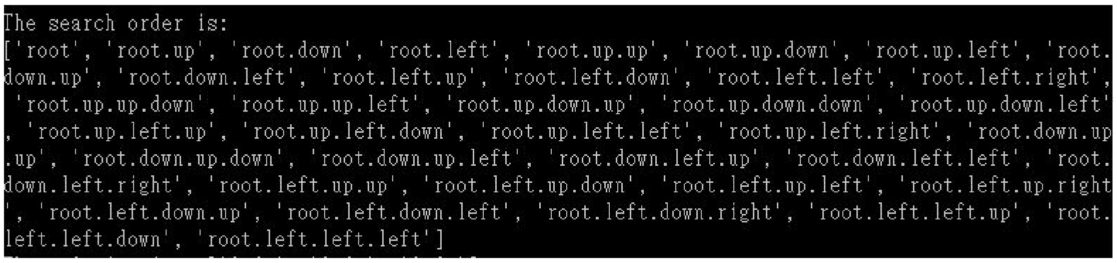


```
The search order is:
['root', 'root.left', 'root.left.right', 'root.left.right.left', 'root.left.right.down', 'root.lef
t.right.up', 'root.left.left', 'root.left.left.right', 'root.left.left.left']
```

Figure 3. Visiting order of Depth-first Search with Depth Limit

Breadth-first search

The search visits all the nodes with the same depth first, then visits the deeper nodes.

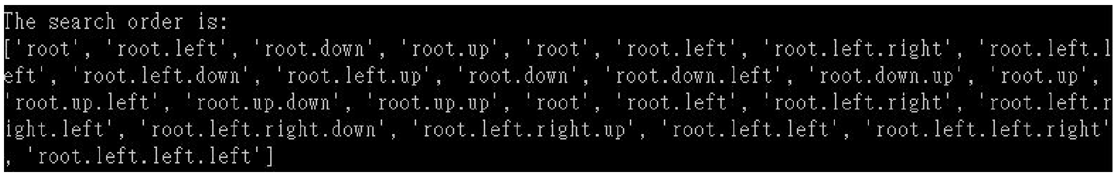


```
The search order is:
['root', 'root.up', 'root.down', 'root.left', 'root.up.up', 'root.up.down', 'root.up.left', 'root.
down.up', 'root.down.left', 'root.left.up', 'root.left.down', 'root.left.left', 'root.left.right',
 'root.up.up.down', 'root.up.up.left', 'root.up.down.up', 'root.up.down.down', 'root.up.down.left'
, 'root.up.left.up', 'root.up.left.down', 'root.up.left.left', 'root.up.left.right', 'root.down.up
.up', 'root.down.up.down', 'root.down.up.left', 'root.down.left.up', 'root.down.left.left', 'root.
down.left.right', 'root.left.up.up', 'root.left.up.down', 'root.left.up.left', 'root.left.up.right
', 'root.left.down.up', 'root.left.down.left', 'root.left.down.right', 'root.left.left.up', 'root.
left.left.down', 'root.left.left.left']
```

Figure 4. Visiting order of Breadth-first Search

Iterative Deepening Search

The search visits the deepest node first. If it reaches the depth limit, it will stop expanding nodes. If there is no more node in the stack, it will revisit the root again and the depth limit is also increased by 1.

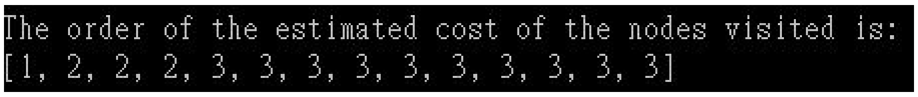


```
The search order is:
['root', 'root.left', 'root.down', 'root.up', 'root', 'root.left', 'root.left.right', 'root.left.l
eft', 'root.left.down', 'root.left.up', 'root.down', 'root.down.left', 'root.down.up', 'root.up',
'root.up.left', 'root.up.down', 'root.up.up', 'root', 'root.left', 'root.left.right', 'root.left.r
ight.left', 'root.left.right.down', 'root.left.right.up', 'root.left.left', 'root.left.left.right'
, 'root.left.left.left']
```

Figure 5. Visiting order of Iterative Deepening Search

A* Search

The search visits the node with the least estimated cost first.



```
The order of the estimated cost of the nodes visited is:
[1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

Figure 6. Visiting order of A* Search

Example Output

See Appendix.

Computational Time Scalability Study

To measure the scalability of computational time (the number of nodes expanded) of the tree search algorithms, 5 different start states will be used for the search. Each start state has different minimal distance (3 to 14) to the goal state. The longer the distance, the more moves are needed to solve the puzzle. Since the difficulty of the search will increase as the distance to the goal state increases, we can find the scalability by comparing the results.

Randomized Depth-first Search (RDFS)

Since the moves are randomized, this algorithm obtains different result each time. Therefore, an average of 100 results is used for the study. This algorithm expands the most nodes when the distance to the goal state is short. However, this algorithm scales very well, even when the distance to the goal state is 14, the number of nodes expanded is not increased by much.

Depth-first Search with Depth Limit (DFSw/DL)

The depth limit is set to 20 for the searches. It should be sufficient to solve from the start states chosen. This search algorithm does not scale well since often it expands the most nodes. However, the result depends on the depth limit selected. If the depth limit chosen is closer to the minimum depth required, the number of nodes expanded can be greatly reduced.

Breath-first Search (BFS)

The scalability of breath-first search is slightly better than depth-first search with depth limit and iterative deepening search since it expands slightly fewer nodes overall. However, it fails to find a solution when the distance to the goal state is 14. The program will raise memory error because it takes too much space.

Iterative Deepening Search

The scalability of this algorithm is similar to that of Breadth-first Search but it expands slightly more nodes since it revisits the whole tree again every time the node stack is empty. However, it succeeds to find a solution when the distance to the goal state is 14 because the space complexity of it is lower.

A* Search

The scalability of A* Search is the best overall. The slope of the line is clearly less than that of other algorithms apart from RDFS. As the distance to the goal state increases, the number of nodes expanded is more significantly fewer than other algorithms apart from RDFS.

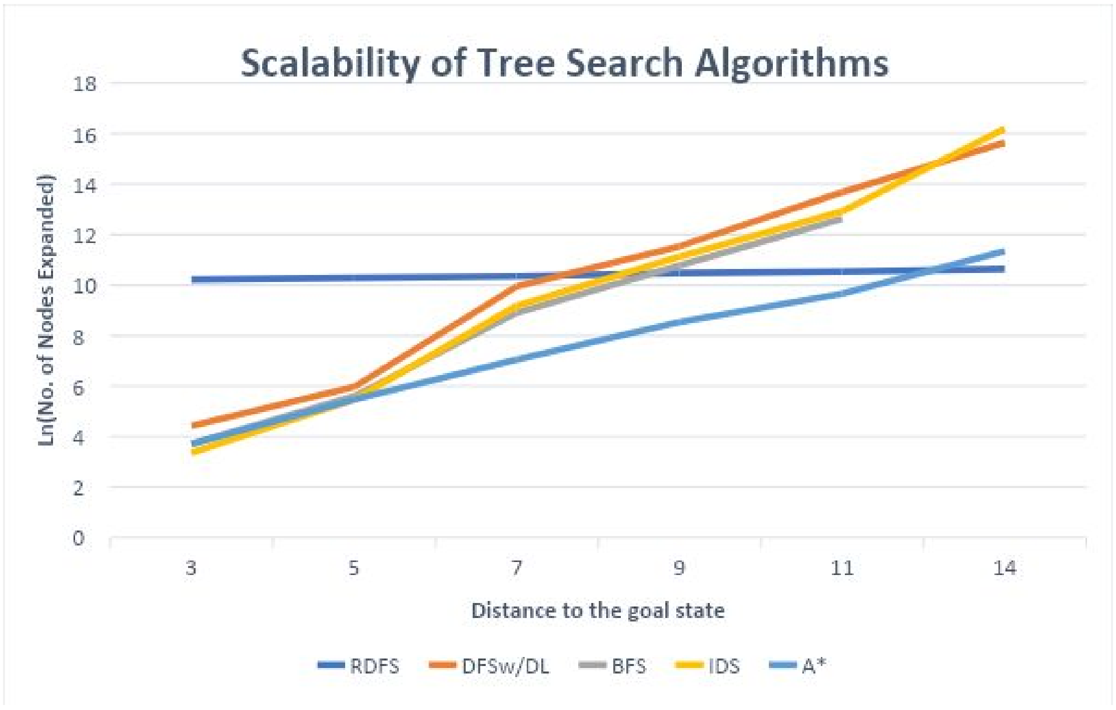


Figure 6. Scalability of Tree Search Algorithms

Limitations

Randomized Depth-first Search

This algorithm only works for puzzles with a bottomless tree. If there is a bottom in the tree, it will be stuck because it is not able to go back and visit other nodes. On the other hand, memory error will be raised if the search depth is too large since it takes too much space to save the visiting order.

Breath-first Search

Memory error will be raised if the search depth is too large since breadth-first search takes so much space to save the nodes.

References

[1] Anon, (2017). Blocksworld Tile Puzzle. [online] Available at: https://secure.ecs.soton.ac.uk/notes/comp6231/blocksworld_tile_puzzle.pdf [Accessed 6 Jan. 2017].

Appendix

Code

code.py

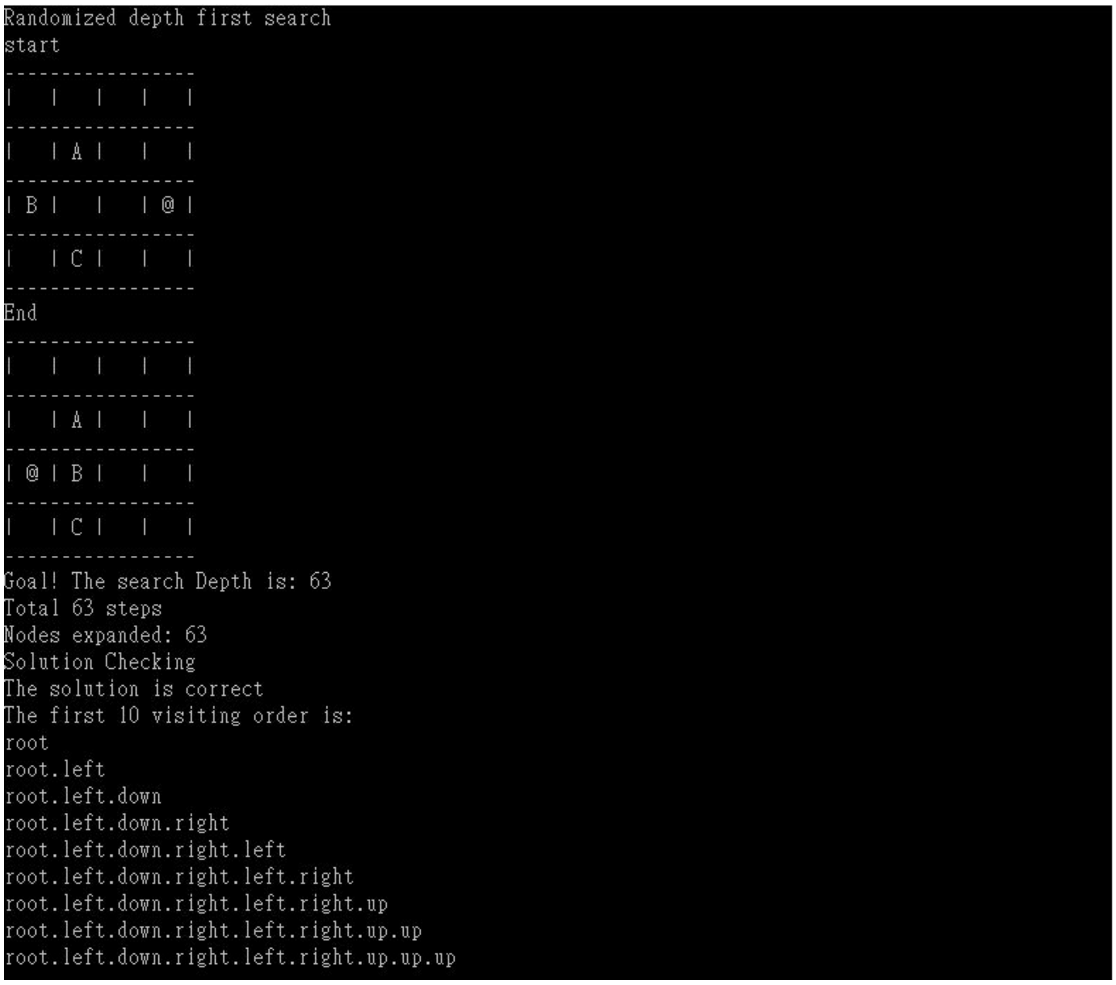
```python
import random
from operator import attrgetter

"""
This code solves the "Blocksworld Tiles Puzzle"
Goal State
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
|   | B |   |   |
-----------------
|   | C |   |   |  (agent can be anywhere)
-----------------
"""

class Node:
    def __init__(self, name, state, parent, move, depth, heuristic):
        self.name = name
        self.state = state
        self.parent = parent
        self.move = move
        self.depth = depth
        self.heuristic = heuristic # Estimated cost
    def __repr__(self):
        return repr((self.name, self.state, self.parent, self.move,
self.depth, self.heuristic))
```

```
def expand(node, goal):
    expanded_nodes = []
    up = move(node.state, 'up')
    down = move(node.state, 'down')
    left = move(node.state, 'left')
    right = move(node.state, 'right')
    new_depth = node.depth + 1
    new_name = node.name

    if up:
        expanded_nodes.append(Node(new_name+'.up', up, node, "up",
new_depth, new_depth + heuristic( up, goal)))
    if down:
        expanded_nodes.append(Node(new_name+'.down', down, node,
"down", new_depth, new_depth + heuristic( down, goal)))
    if left:
        expanded_nodes.append(Node(new_name+'.left', left, node,
"left", new_depth, new_depth + heuristic( left, goal)))
    if right:
        expanded_nodes.append(Node(new_name+'.right', right, node,
"right", new_depth, new_depth + heuristic( right, goal)))
    return expanded_nodes

def display(state):
    print "-----------------"
    print "| {} | {} | {} | {} |".format(state[0], state[1], state[2],
state[3])
    print "-----------------"
    print "| {} | {} | {} | {} |".format(state[4], state[5], state[6],
state[7])
    print "-----------------"
    print "| {} | {} | {} | {} |".format(state[8], state[9],
state[10], state[11])
    print "-----------------"
    print "| {} | {} | {} | {} |".format(state[12], state[13],
state[14], state[15])
    print "-----------------"
    return True

def getNextMove(agent):
    moves={0:"up",1:"down",2:"left",3:"right"}
    rng=random.randint(0,3)
    while (rng==0 and (agent in [0, 1, 2, 3])) or (rng==1 and (agent
in [12, 13, 14, 15])) or (rng==2 and (agent in [0, 4, 8, 12])) or
(rng==3 and (agent in [3, 7, 11, 15])) :#Only moves within the
boundaries
        rng=random.randint(0,3)
    return(moves[rng])

def move(state, direction):
    new_state = state[:]
    index = new_state.index( '@' )
    if direction == 'up':
        if index not in [0, 1, 2, 3]: # Can't go up
            temp = new_state[index - 4]
            new_state[index - 4] = new_state[index]
            new_state[index] = temp
            return new_state
        else:
            return None
    if direction == 'down':
        if index not in [12, 13, 14, 15]: # Can't go down
            temp = new_state[index + 4]
            new_state[index + 4] = new_state[index]
            new_state[index] = temp
            return new_state
        else:
            return None
    if direction == 'left':
        if index not in [0, 4, 8, 12]: # Can't go left
            temp = new_state[index - 1]
            new_state[index - 1] = new_state[index]
            new_state[index] = temp
            return new_state
        else:
            return None
    if direction == 'right':
        if index not in [3, 7, 11, 15]: # Can't go right
            temp = new_state[index + 1]
            new_state[index + 1] = new_state[index]
            new_state[index] = temp
            return new_state
        else:
            return None

def is_goal(state, goal):
    if state[goal['A']]=='A' and state[goal['B']]=='B' and
state[goal['C']]=='C':
        return True
    else:
        return False

def solution(node):
    moves = []
    temp = node
    while temp.move:
            moves.insert(0, temp.move)
            if temp.depth == 1:
                break
            temp = temp.parent
    return moves

def dfs_random(start, goal, show_board, show_order):
    state = start
    count = 0
    moveList = list()
    print 'start'
    if show_board:
        display(state)
    order = ['root'] # visiting order
    while not is_goal(state,goal):
        nextMove=getNextMove(state.index('@'))
        if show_order:
            order.append(order[len(order)-1]+'.'+nextMove)
        moveList.append(nextMove)
        state=move(state,nextMove)
        count+=1
    print('End')
    if show_board:
        display(state)
    print ('Goal! The search Depth is: '+str(count))
    print('Total '+str(count)+' steps')
    print('Nodes expanded: '+str(count))
    print ('Solution Checking')
    if solution_check(start, goal, moveList, show_board):
        if show_order:
            print('The first 10 visiting order is: ')
            for i in xrange(10):
                print(order[i])
        return count # Return the number of nodes expanded

def dfs_limit(start,goal, depth_limit, show_board, show_order):
    nodes = [] # Node stack
    count = 0 # Number of nodes expanded
    visited = 0 # Number of nodes visited
    order = [] # pop order
    nodes.append(Node('root', start, None, None, 0, heuristic(start,
goal))) # Root
    print 'Start'
    if show_board:
        display(start)
    while True:
        if len(nodes) == 0:
            print('No solution! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            if show_order:
                print('The search order is:')
                print(order)
            return None #No solution
        node = nodes.pop() #Pop the top node
        if show_order:
            order.append(node.name) # Record the visiting order
        visited += 1
        if is_goal(node.state, goal):
            print('End')
            if show_board:
                display(node.state)
            print('Goal! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            if show_order:
                print('The search order is:')
                print(order)
            moves=solution(node)
            print ('Solution Checking')
            if solution_check(start, goal, moves, show_board):
                return moves
        expanded_nodes = []
        if node.depth < depth_limit: #Expand only if the node is less
than the depth limit
            expanded_nodes = (expand(node, goal))# expand
            nodes.extend(expanded_nodes) #put the expanded node at the
top of the stack
```

```
            count += len(expanded_nodes)

def bfs(start, goal, show_board, show_order):
    nodes = [] # Node queue
    count = 0 # Number of nodes expanded
    visited = 0 # Number of nodes visited
    order = [] # pop order
    nodes.append(Node('root', start, None, None, 0, heuristic(start,
goal))) # Root
    print 'Start'
    if show_board:
        display(start)
    while True:
        if len(nodes) == 0:
            print('No solution! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            print('The search order is:')
            if show_order:
                print(order)
            return None #No solution
        for i in xrange(len(nodes)): # Check if there is solution in
the nodes
            if show_order:
                order.append(nodes[i].name) # Record the visiting
order
            visited += 1
            if is_goal(nodes[i].state, goal):
                print('End')
                if show_board:
                    display(nodes[i].state)
                print('Goal! The search depth is
'+str(nodes[i].depth))
                print('Nodes expanded:' + str(count))
                print('Nodes visited: ' + str(visited))
                if show_order:
                    print('The search order is:')
                    print(order)
                moves=solution(nodes[i])
                print ('Solution Checking')
                if solution_check(start, goal, moves, show_board):
                    return moves
        expanded_nodes = []
        while len(nodes) > 0: # Pop and expand until there is no nodes
            node = nodes.pop(0)
            expanded_nodes.extend(expand( node, goal ))
        count += len(expanded_nodes)
        nodes.extend( expanded_nodes )

def ids(start, goal, show_board, show_order):
    nodes = [] # Node stack
    count = 0 # Number of nodes expanded
    visited = 0 # Number of nodes visited
    depth_limit = 1
    order = [] # pop order
    nodes.append(Node('root', start, None, None, 0, heuristic(start,
goal))) # Root
    print 'Start'
    if show_board:
        display(start)
    while True:
        node = nodes.pop() #Pop the top node
        if show_order:
            order.append(node.name) # Record the visiting order
        visited += 1
        if is_goal(node.state, goal):
            print('End')
            if show_board:
                display(node.state)
            print('Goal! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            if show_order:
                print('The search order is:')
                print(order)
            moves=solution(node)
            print ('Solution Checking')
            if solution_check(start, goal, moves, show_board):
                return moves
        expanded_nodes = []
        if node.depth < depth_limit: #Expand only if the node is less
than the depth limit
            expanded_nodes = (expand(node, goal))# expand
            nodes.extend(expanded_nodes) #put the expanded node at the
top of the stack
            count += len(expanded_nodes)
        if len(nodes) == 0:  # No solution. Increse depth_limit
            depth_limit += 1
            nodes.append(Node('root', start, None, None, 0,
heuristic(start, goal))) # Root



def astar(start, goal, show_board, show_order):
    nodes = [] # Node queue
    count = 0 # Number of nodes expanded
    visited = 0 # Number of nodes visited
    order = [] # pop order
    nodes.append(Node('root', start, None, None, 0, heuristic(start,
goal))) # Root
    print 'Start'
    if show_board:
        display(start)
    while True:
        if len(nodes) == 0:
            print('No solution! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            if show_order:
                print('The search order is:')
                print(order)
            return None #No solution
        node = nodes.pop(nodes.index(min(nodes, key =
attrgetter('heuristic')))) # Pop and expand the node with the least
estimated cost
        if show_order:
            order.append(node.heuristic) # Record the visiting order
        visited += 1
        if is_goal(node.state, goal): #check if the node poped is the
goal
            print('End')
            if show_board:
                display(node.state)
            print('Goal! The search depth is '+str(node.depth))
            print('Nodes expanded:' + str(count))
            print('Nodes visited: ' + str(visited))
            if show_order:
                print('The order of the estimated cost of the nodes
visited is:')
                print(order)
            moves=solution(node)
            print ('Solution Checking')
            if solution_check(start, goal, moves, show_board):
                return moves
        expanded_nodes = []
        expanded_nodes.extend(expand(node, goal)) #expand
        count += len(expanded_nodes)
        nodes.extend(expanded_nodes)# Add the expanded node at the
back of the queue

def heuristic(state, goal): #Sum of the Manhattan Distance between A B
C in the current state and that in the goal state
    if state:
        A = state.index('A')
        B = state.index('B')
        C = state.index('C')
        dist =
abs(A%4-goal['A']%4)+abs(A/4-goal['A']/4)+abs(B%4-goal['B']%4)+abs(B/4
-goal['B']/4)+abs(C%4-goal['C']%4)+abs(C/4-goal['C']/4)
        return dist
    else:
        return None

def solution_check(start, goal, solution, show_board):
    if is_goal(start,goal):
        print ('The solution is correct')
        return True
    if solution:
        state = start
        if show_board:
            display(state)
        for i in xrange(len(solution)):
            state = move(state, solution[i])
            if show_board:
                display(state)
        if is_goal(state,goal):
            print ('The solution is correct')
            return True
        else:
            print ('The solution is wrong')
            return False
    else:
        print('No solution')
        return False
```

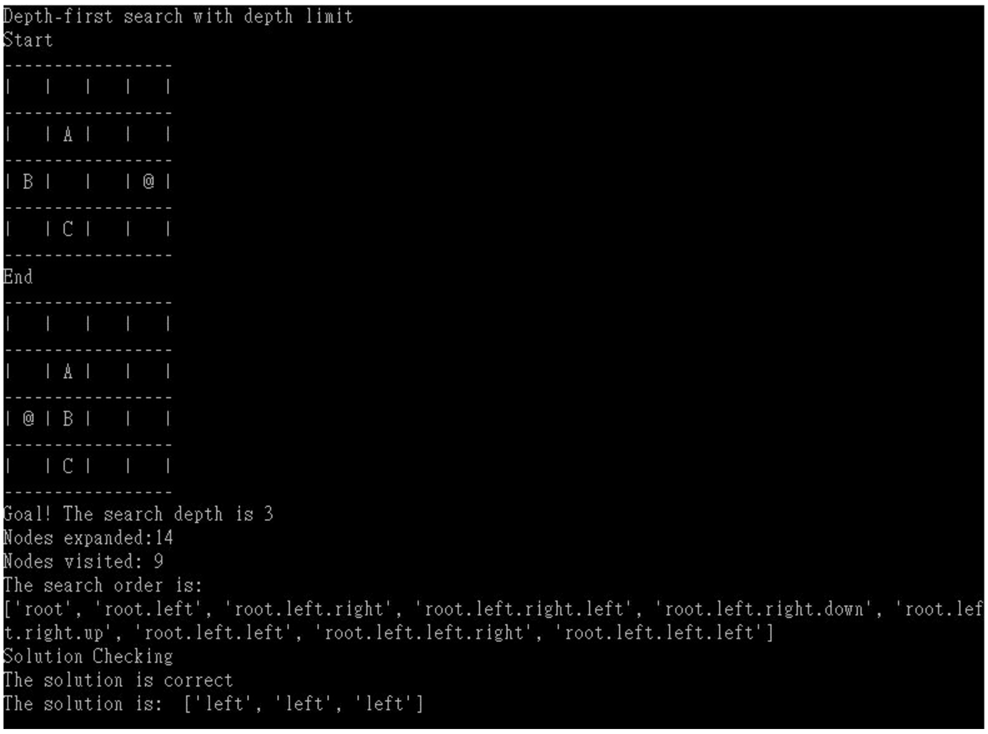
```python
def avg(start,goal, times, show_board, show_order):
    count = 0
    for i in xrange(times):
        count+=dfs_random(start,goal, show_board, show_order)
    avg = count//times
    return avg




def main():
    start = [' ',' ',' ',' ',' ','A',' ',' ','B',' ',' ','@',' ','C','
',' '] # Distance from goal is 3
    #start = [' ',' ',' ','@',' ','A',' ',' ','B',' ',' ',' ',' ','C','
',' '] # Distance from goal is 5
    #start = [' ',' ','A',' ',' ',' ','@',' ','B',' ',' ',' ',' ','C','
',' '] # Distance from goal is 7
    #start = [' ',' ','A',' ',' ',' ',' ',' ','B',' ',' ',' ','
','C','@',' '] # Distance from goal is 9
    #start = [' ',' ','A',' ',' ',' ',' ',' ',' ',' ','B',' ',' ','C','
','@'] # Distance from goal is 11
    #start = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','
','A','B','C','@'] # Distance from goal is 14
    #start = ['A','B','C',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' ','
','@'] # Distance from goal is 14
    goal = {'A':5, 'B':9, 'C':13}
    show_board = False # display the board if True
    show_order = False # show the visiting order if True

    print ('Randomized depth first search')
    average=(avg(start, goal, 100, show_board, show_order))
    print 'The average is',
    print(average)
    print('\n')

    print ('Depth-first search with depth limit')
    solution=(dfs_limit(start, goal, 20, show_board, show_order))
    print 'The solution is: ',
    print solution
    print('\n')

    print ('Performing breadth first search')
    solution=(bfs(start, goal, show_board, show_order))
    print 'The solution is: ',
    print solution
    print('\n')

    print ('Iterative deepening search')
    solution=(ids(start, goal, show_board, show_order))
    print 'The solution is: ',
    print solution
    print('\n')

    print ('A* search')
    solution=(astar(start, goal, show_board, show_order))
    print 'The solution is: ',
    print solution
    print('\n')


if __name__ == "__main__":
    main();
```

Example Output



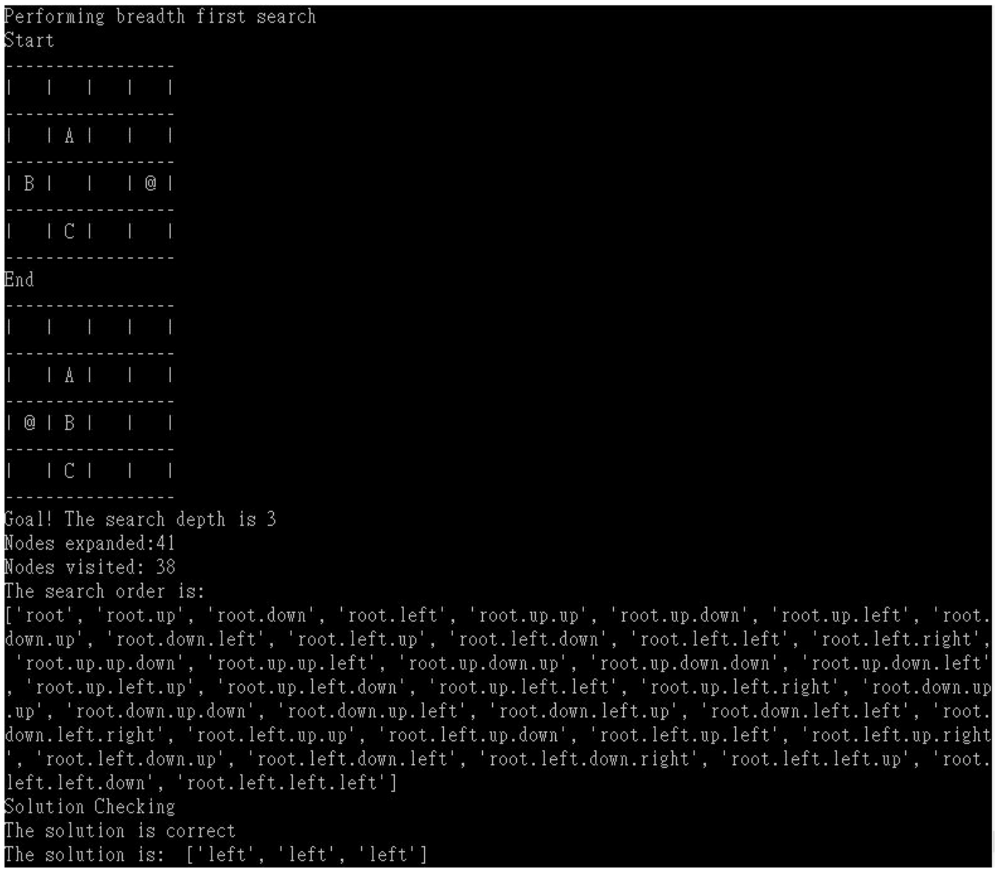
```
Randomized depth first search
start
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| B |   |   | @ |
-----------------
|   | C |   |   |
-----------------
End
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| @ | B |   |   |
-----------------
|   | C |   |   |
-----------------
Goal! The search Depth is: 63
Total 63 steps
Nodes expanded: 63
Solution Checking
The solution is correct
The first 10 visiting order is:
root
root.left
root.left.down
root.left.down.right
root.left.down.right.left
root.left.down.right.left.right
root.left.down.right.left.right.up
root.left.down.right.left.right.up.up
root.left.down.right.left.right.up.up.up
```

Figure 7. Example Output of Randomized Depth-first Search



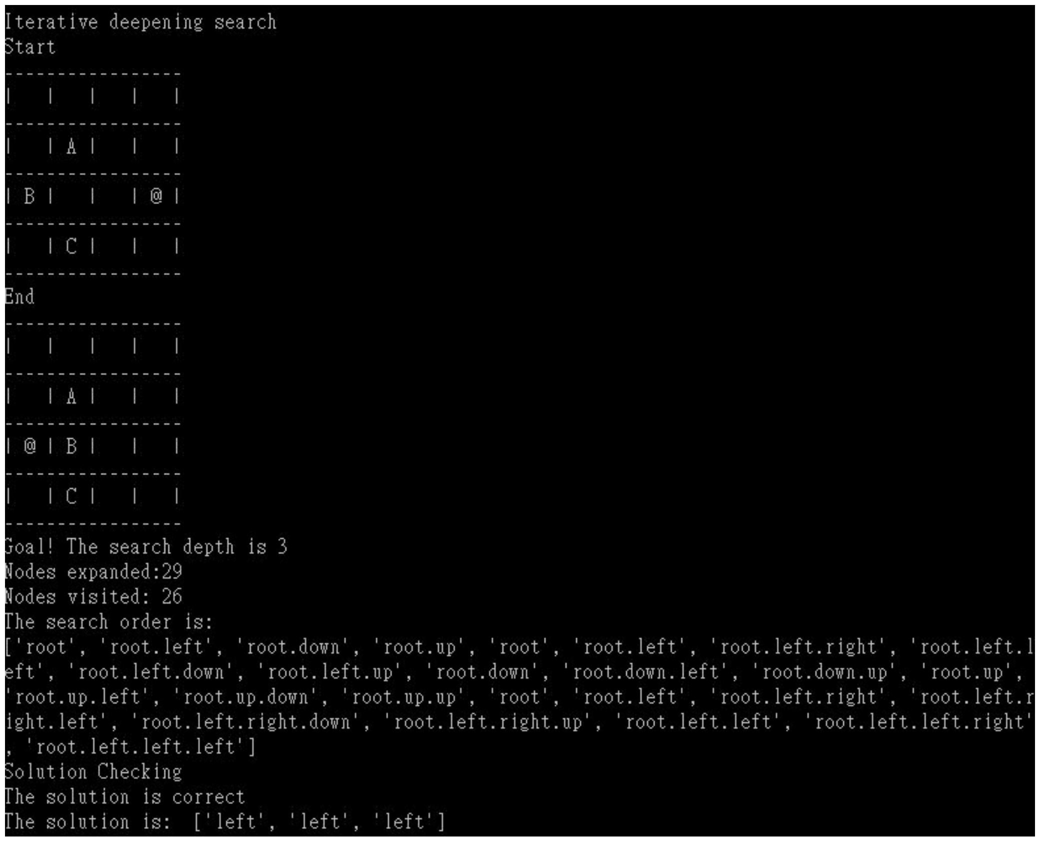
```
Depth-first search with depth limit
Start
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| B |   |   | @ |
-----------------
|   | C |   |   |
-----------------
End
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| @ | B |   |   |
-----------------
|   | C |   |   |
-----------------
Goal! The search depth is 3
Nodes expanded:14
Nodes visited: 9
The search order is:
['root', 'root.left', 'root.left.right', 'root.left.right.left', 'root.left.right.down', 'root.lef
t.right.up', 'root.left.left', 'root.left.left.right', 'root.left.left.left']
Solution Checking
The solution is correct
The solution is:  ['left', 'left', 'left']
```

Figure 8. Example Output of Depth-first Search with Depth Limit



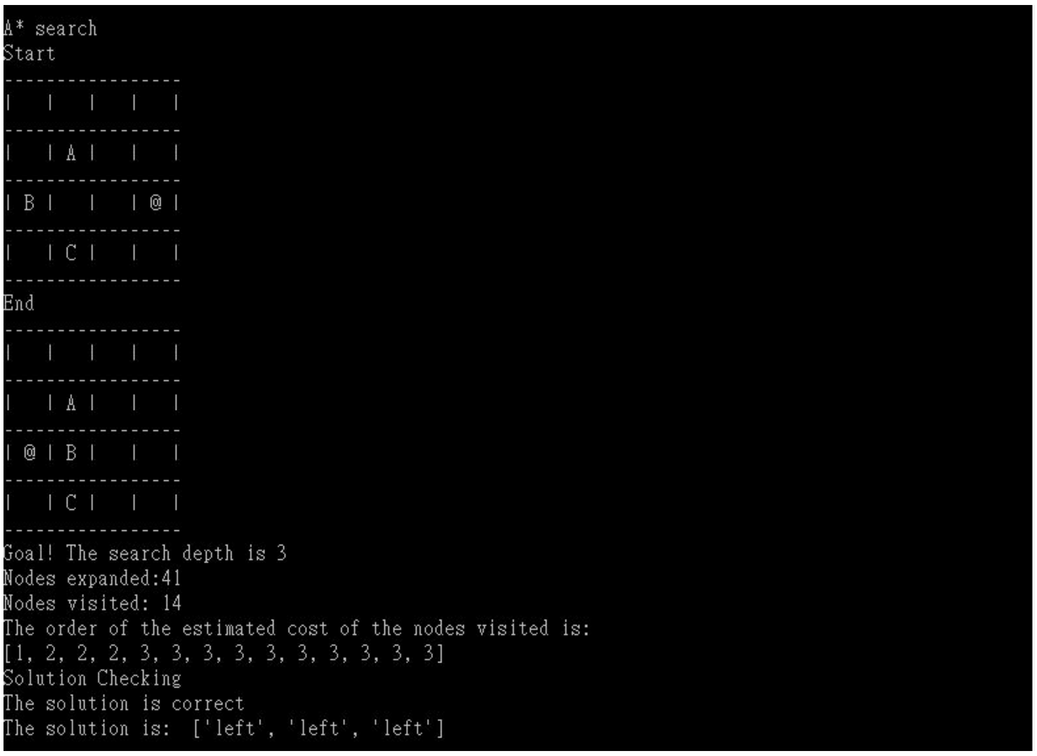
```
Performing breadth first search
Start
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| B |   |   | @ |
-----------------
|   | C |   |   |
-----------------
End
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| @ | B |   |   |
-----------------
|   | C |   |   |
-----------------
Goal! The search depth is 3
Nodes expanded:41
Nodes visited: 38
The search order is:
['root', 'root.up', 'root.down', 'root.left', 'root.up.up', 'root.up.down', 'root.up.left', 'root.
down.up', 'root.down.left', 'root.left.up', 'root.left.down', 'root.left.left', 'root.left.right',
 'root.up.up.down', 'root.up.up.left', 'root.up.down.up', 'root.up.down.down', 'root.up.down.left'
, 'root.up.left.up', 'root.up.left.down', 'root.up.left.left', 'root.up.left.right', 'root.down.up
.up', 'root.down.up.down', 'root.down.up.left', 'root.down.left.up', 'root.down.left.left', 'root.
down.left.right', 'root.left.up.up', 'root.left.up.down', 'root.left.up.left', 'root.left.up.right
', 'root.left.down.up', 'root.left.down.left', 'root.left.down.right', 'root.left.left.up', 'root.
left.left.down', 'root.left.left.left']
Solution Checking
The solution is correct
The solution is:  ['left', 'left', 'left']
```

Figure 9. Example Output of Breadth-first Search



```
Iterative deepening search
Start
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| B |   |   | @ |
-----------------
|   | C |   |   |
-----------------
End
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| @ | B |   |   |
-----------------
|   | C |   |   |
-----------------
Goal! The search depth is 3
Nodes expanded:29
Nodes visited: 26
The search order is:
['root', 'root.left', 'root.down', 'root.up', 'root', 'root.left', 'root.left.right', 'root.left.l
eft', 'root.left.down', 'root.left.up', 'root.down', 'root.down.left', 'root.down.up', 'root.up',
'root.up.left', 'root.up.down', 'root.up.up', 'root', 'root.left', 'root.left.right', 'root.left.r
ight.left', 'root.left.right.down', 'root.left.right.up', 'root.left.left', 'root.left.left.right'
, 'root.left.left.left']
Solution Checking
The solution is correct
The solution is:  ['left', 'left', 'left']
```

Figure 10. Example Output of Randomized Iterative Deepening Search



```
A* search
Start
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| B |   |   | @ |
-----------------
|   | C |   |   |
-----------------
End
-----------------
|   |   |   |   |
-----------------
|   | A |   |   |
-----------------
| @ | B |   |   |
-----------------
|   | C |   |   |
-----------------
Goal! The search depth is 3
Nodes expanded:41
Nodes visited: 14
The order of the estimated cost of the nodes visited is:
[1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
Solution Checking
The solution is correct
The solution is:  ['left', 'left', 'left']
```

Figure 11. Example Output of A* Search