Evaluating Search Methods using Blocksworld Tile Puzzle

COMP6231: Foundations of Artificial Intelligence

Vasin Srisupavanich
ID: 31300162
vs2n19@soton.ac.uk

1. Approach

First, I started this project by creating the underlying mechanics of the Blockworld tile puzzle. This includes creating the re-sizable NxN grid, initializing the positions of agent and blocks, and defining valid movement of agent and blocks. Second, I created a single player game where I can play and test the mechanics of the puzzle. Then, I created all the search methods, including breadth-first search, depth-first search, depth-limited search, iterative deepening depth-first search, and A* search using total Manhattan distance between the block and the goal state as the heuristic.

This project is implemented in Python. There are a total of 9 python files: BlocksWorld.py, BlocksWorldProblem.py, BlocksWorldState.py, SinglePlayer.py, Search.py, TreeNode.py, Solution.py, util.py, and SearchPlot.py. The descriptions and methods of each file is described below.

1. BlocksWorld.py: This file contain a **BlocksWorld** class which defines the mechanics of the puzzle. This class can be initiate with any NxN grid size, and any start state. This class stores the position of the agent and the blocks, and generate legal moves of the agent. Also, this class is used to generate new state of the puzzle, check whether the state is a goal state, and print the puzzle board.

2. BlocksWorldProblem.py: This class turns the BlocksWorld puzzle into a **search problem**, and contains the methods to get the start state, check the goal state, and get the successor nodes.

3. BlocksWorldState.py: This file contains all the search problem difficulty, from start state with depth 1 to state state with depth 14 (original start state).

4. SinglePlayer.py: This file can be used to run a **single player game**. This has been useful in the debugging process when creating the mechanics of BlockWorld tile puzzle.

5. Search.py: This file contains all the search algorithms required, including both tree search and graph search algorithm. This file is used to run the search problem.

6. TreeNode.py: This class represents the node in a tree search, and contains puzzle state, parent state, depth, action and the cost of the action.

7. Solution.py: This class represents the solution of the search problem which contains the path from start state to goal state, total number of nodes generated, and depth and cost of the solution.

8. util.py: This is the utility file which contain Queue, Stack and Priority Queue data structure for easy usage. Queue is a wrapper for Python's deque data structure, Stack is a wrapper for Python's array, and Priority Queue is a wrapper for Python's heapq data structure.

9. SearchPlot.py: This file contain the code to generate the plot using matplotlib library.

1.1 Search Methods Implementation

Breadth-First Search (BFS)

Breadth-first search algorithm was implemented using a FIFO queue (internally use a python deque) as a fringe. It starts by inserting a node that contains start state into the fringe. While the fringe is not empty and the goal state is not found, new successor nodes are added to the fringe. If the node is the goal state, then the path from the start state to the goal state is return. BFS will always find the optimal solution, however, the space requirement is very high, since it needs to keep all the node in memory.

Depth-First Search (DFS)

Depth-first search algorithm implementation is similar to breadth-first search except that the fringe is implemented using a stack (internally use a python array). To avoid pointless loop in DFS, the order of node expansion is randomised. This is needed in order to find the solution for the blocksworld tile puzzle, since the agent will not keep going using the same direction. In general, DFS algorithm will not find the optimal solution, but the advantage of using DFS is that the space complexity is linear.

Iterative Deepening Search (IDS)

Iterative deepening search was implemented using recursive depth limited search. Until a solution is found, depth limited search will be called with an increasing depth. IDS algorithm combines the benefit of BFS and DFS in which it will always find the optimal solution, while keeping the space complexity linear.

A* Search

A* search was implement using a priority queue (internally use python heapq) as a fringe. The sum of Manhattan distance between each block to its corresponding goal position is used as a heuristic. This is an admissible heuristic. A* always find the optimal solution, and is the fastest of all the search methods.

2. Evidence

In this section, I will show the output from running each algorithm on two problems with different difficulty. Figure 1 shows the original start state of the problem. The depth of the optimal solution to the original problem is 14. In order to get the results for BFS and DFS search methods within reasonable time, I have created another search problem by changing the start state to be nearer to the goal state. The depth of the optimal solution for this new problem is 4. Figure 2 shows the position of the start state for this new problem. Both of the problems have the same goal state as shown in figure 3.

Running BFS and DFS using the original start state, the solution could not be found within 1 hour. However, both IDS and A* search was able to find the optimal solution to both of

the problems. Modifying DFS by randomise the order of node expansion allows the solution to be found on the easy problem which have the solution in depth 4. The output below clearly shows that A* search method is the best, because it generated less nodes and run the fastest. The console outputs from running all the search methods are shown below.

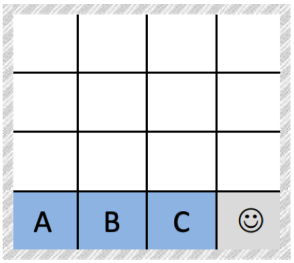

Figure 1: Original start state

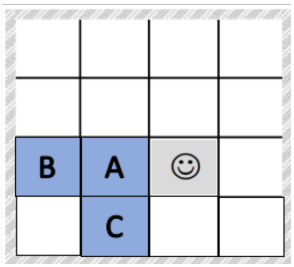

Figure 2: Easy start state

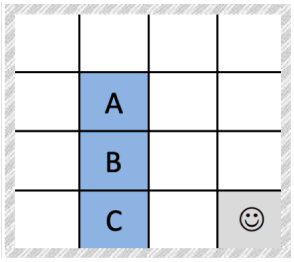

Figure 3: Goal state

Breadth-First Search (BFS)

1. **Console logs from running BFS search using easy start state:**

```
Running breadth-first search...
Found result: ['UP', 'LEFT', 'DOWN', 'LEFT']
Total elapse time in second: 0.012722015380859375
Total node generated: 306
Depth of the solution: 4
Total cost from start to goal state: 4
```

Depth-First Search (DFS)

1. **Console logs from running DFS search using easy start state:**

```
Running depth-first search...
Found result: ['UP', 'RIGHT', 'LEFT', 'UP', 'DOWN', 'RIGHT', 'DOWN', 'DOWN',
'UP', 'DOWN', 'LEFT', 'RIGHT', 'UP', 'DOWN', 'UP', 'UP',
'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'UP', 'UP', 'LEFT',
'DOWN', 'UP', 'DOWN', 'LEFT', 'UP', 'LEFT', 'RIGHT', 'DOWN',
'LEFT', 'UP', ...]  *** Omit since there are 9680 steps.
Total elapse time in second: 4.801503896713257
Total node generated: 30509
Depth of the solution: 9680
Total cost from start to goal state: 9680
```

Iterative Deepening Search (IDS)

1. **Console logs from running IDS search using original start state:**

```
Running iterative deepening search...
Found result: ['UP', 'LEFT', 'LEFT', 'DOWN', 'LEFT', 'UP',
'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP', 'LEFT', 'DOWN', 'LEFT']
Total elapse time in second: 314.89391112327576
Total node generated: 8943144
Depth of the solution: 14
Total cost from start to goal state: 14
```

2. Console logs from running IDS search using easy start state:

```
Running iterative deepening search...
Found result: ['UP', 'LEFT', 'DOWN', 'LEFT']
Total elapse time in second: 0.006056785583496094
Total node generated: 133
Depth of the solution: 4
Total cost from start to goal state: 4
```

A* Search using Manhattan Distance Heuristics

1. Console logs from running A* search using original start state:

```
Running A star heuristic search...
Found result: ['UP', 'LEFT', 'LEFT', 'DOWN', 'LEFT',
'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'UP', 'LEFT', 'DOWN', 'LEFT']
Total elapse time in second: 4.751442193984985
Total node generated: 84242
Depth of the solution: 14
Total cost from start to goal state: 14
```

2. Console logs from running A* search using easy start state:

```
Running A star heuristic search...
Found result: ['UP', 'LEFT', 'DOWN', 'LEFT']
Total elapse time in second: 0.003735065460205078
Total node generated: 59
Depth of the solution: 4
Total cost from start to goal state: 4
```

3. Scalability

To control the problem difficulty, I have created 14 start states with 1 to 14 solution depth. Each start state is chosen from the optimal path from the original problem. All the search algorithms are ran on all 14 problems, and the number of node generated are recorded. Since DFS method depends on random order of node expansion, the result of each run varies. For DFS, the average of node generated from 10 runs, when the solution is found, are recorded

(many DFS runs failed to find the solution). Figure 4 shows the result of the scalabilty of each search method, measure with the number of node generated with respect to problem difficulty.

From the result shown in figure 4, it is obvious that A* search using the sum of Manhattan distance between the block and its goal state perform the best. It generated significantly less number of nodes, and take significantly less amount of time to find the optimal solution. As the problem difficulty increase, the node generated from uninformed search became significantly greater than A* search in the order of exponential magnitude. For BFS and DFS, the solution was found within 1 hour only until problem with depth 12. Most of the time, DFS failed to find the solution even with problem with small depth. This is due to the fact that DFS will keep expanding deeper node first, and the solution return from DFS, most of the time will be sub-optimal. Even though, BFS will always find the optimal solution, the biggest weakness is that it is very slow and have to keep all the nodes in memory for the next expansion. Out of all the uninformed search strategies, IDS performs the best, with consistently generated less amount of nodes, and the solution in the problem with depth 14 was found without any problem. This is due to the fact that the memory usage of IDS is linear just like DFS, and it will always return the optimal solution just like BFS.

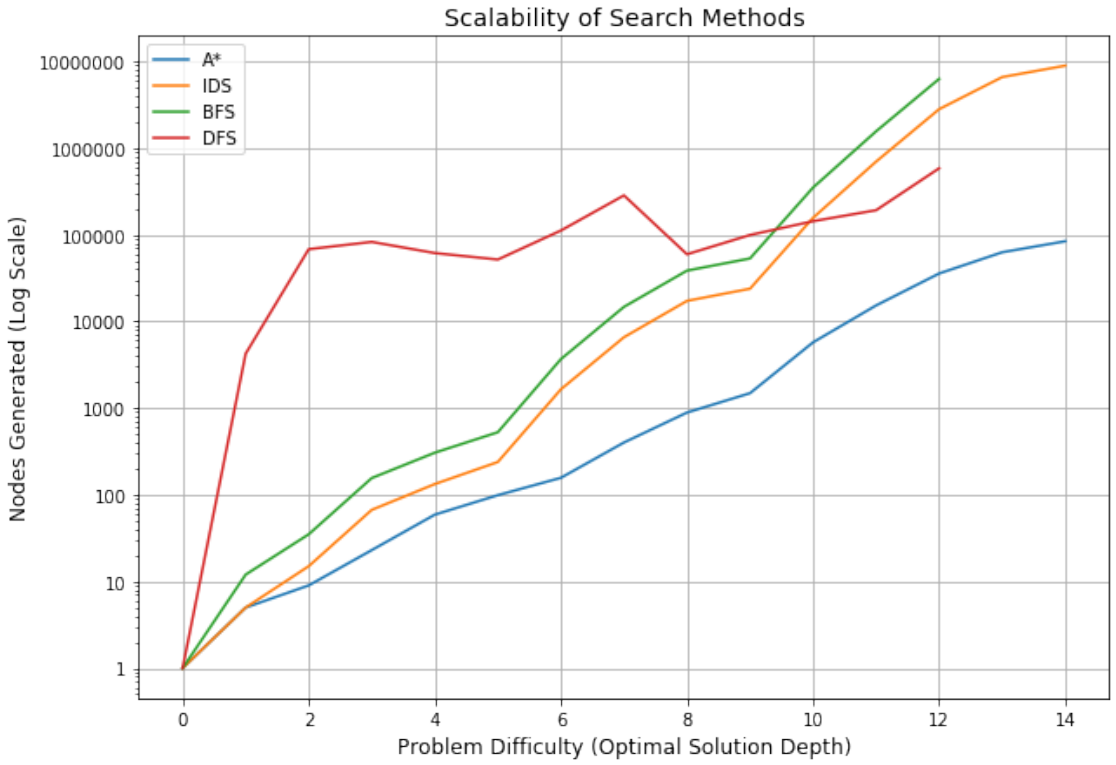

Figure 4: Number of Node Generated vs Problem Difficulty

4. Extras and limitations

4.1 Extras

I have implemented several extra features as part of this project. First, I have created a 1 player game for Blocksworld tile puzzle. This was created to help debug the puzzle mechanism. I have played Blocksworld puzzle several times, and most of the time, my solutions was sub-optimal, and I was surprised to see some of the optimal results generated from the search algorithms. Second, the implementation of the search problem is done in a generic way. This means that all the search algorithms can run on any search problem, not limiting to Blocksworld puzzle. Third, the size of the grid of the Blocksworld puzzle is adjustable. This can make the search problem more difficult or easy depends on the size of the grid. Fourth, apart from tree search, I have implemented graph search algorithms for Breadth-first, Depth-first and A* search, by adding a set to keep track of all the visited state. Since, there are a lot of duplicated states in a tree search in Blocksworld puzzle, using graph search algorithms have significantly speed up the search process. However, using graph search requires more memory, since the algorithm need to keep all the node visited in memory, and this could be a problem in a larger search problem.

4.2 Limitations

There are still some improvements that can be made in this project. For example, the goal state of Blocksworld puzzle could be made to be adjustable. Also, the number of block could be made adjustable, making the problem more difficult. Moreover, I believe there could be a better heuristics than just a sum of Manhattan distance in A* search method, which make use of the distance of the agent to the blocks.

5. Code

BlocksWorld.py

```
import copy


class BlocksWorld:
    """
       Blocksworld tile puzzle as described in the AI coursework
       The puzzle can be of any N x N size depends on the input.
       This class defines the mechanics of the puzzle.
       The following are what represent each object in the game.
       X represents an agent.
       Empty space represents empty tile.
       Alphabet letter (A, B, C) represents the block.
       The goal of this game is to arrange the block according to the rule (goal
           state).
       """
```

```
def __init__(self, size, objects_pos):
    self.size = size
    self.board = [['' for x in range(size)] for x in range(size)]
    for row in range(size):
        for col in range(size):
            self.board[row][col] = objects_pos.get(size * row + col + 1, ' ')
            if self.board[row][col] == 'X':
                self.agent_location = row, col

    self.goal = {'A': (self.size - 3, self.size - 3), 'B': (self.size - 2,
        self.size - 3),
                 'C': (self.size - 1, self.size - 3)}

def get_block_pos(self):
    block_location = {}
    for row in range(self.size):
        for col in range(self.size):
            if self.board[row][col] == 'A':
                block_location['A'] = row, col
            if self.board[row][col] == 'B':
                block_location['B'] = row, col
            if self.board[row][col] == 'C':
                block_location['C'] = row, col
    return block_location

def get_goal_state(self):
    return self.goal

def move(self, direction):
    new_agent_loc = self.get_new_agent_location(direction)
    old_value = self.board[self.agent_location[0]][self.agent_location[1]]
    self.board[self.agent_location[0]][self.agent_location[1]] =
        self.board[new_agent_loc[0]][new_agent_loc[1]]
    self.board[new_agent_loc[0]][new_agent_loc[1]] = old_value
    self.agent_location = new_agent_loc

def get_new_agent_location(self, direction):
    if direction == 'UP':
        new_location = self.agent_location[0] - 1, self.agent_location[1]
    elif direction == 'DOWN':
        new_location = self.agent_location[0] + 1, self.agent_location[1]
    elif direction == 'LEFT':
        new_location = self.agent_location[0], self.agent_location[1] - 1
    elif direction == 'RIGHT':
        new_location = self.agent_location[0], self.agent_location[1] + 1
    else:
```

```
            raise ValueError('Invalid move')
        return new_location

    def get_new_state(self, direction):
        new_agent_loc = self.get_new_agent_location(direction)
        new_board = copy.deepcopy(self.board)
        old_value = new_board[self.agent_location[0]][self.agent_location[1]]
        new_board[self.agent_location[0]][self.agent_location[1]] =
            new_board[new_agent_loc[0]][new_agent_loc[1]]
        new_board[new_agent_loc[0]][new_agent_loc[1]] = old_value
        new_obj_pos = get_objects_pos(new_board, self.size)
        return BlocksWorld(self.size, new_obj_pos)

    def get_legal_move(self):
        legal_moves = []
        row, col = self.agent_location
        if row != 0:
            legal_moves.append('UP')
        if row != self.size - 1:
            legal_moves.append('DOWN')
        if col != 0:
            legal_moves.append('LEFT')
        if col != self.size - 1:
            legal_moves.append('RIGHT')
        return legal_moves

    def get_agent_location(self):
        print(self.agent_location)

    def is_goal(self):
        if self.board[self.size - 3][self.size - 3] == 'A' and
            self.board[self.size - 2][self.size - 3] == 'B' and \
                self.board[self.size - 1][self.size - 3] == 'C':
            return True
        else:
            return False

    def create_board_string(self):
        lines = []
        vertical_line = '|'
        horizontal_line = '-' * (self.size * 4 + 1)
        for row in range(self.size):
            row_line = ''
            for col in range(self.size):
                row_line += vertical_line + ' ' + str(self.board[row][col]) + ' '
            lines.append(horizontal_line)
            lines.append(row_line + vertical_line)
```

```
        lines.append(horizontal_line)
        return '\n'.join(lines)

    def display_board(self):
        print(self.create_board_string())

    def __eq__(self, other):
        for row in range(self.size):
            if self.board[row] != other.board[row]:
                return False
        return True

    def __hash__(self):
        return hash(str(self.board))


def get_objects_pos(board, size):
    pos = {}
    for row in range(size):
        for col in range(size):
            if board[row][col] == 'A':
                pos[size * row + col + 1] = 'A'
            elif board[row][col] == 'B':
                pos[size * row + col + 1] = 'B'
            elif board[row][col] == 'C':
                pos[size * row + col + 1] = 'C'
            elif board[row][col] == 'X':
                pos[size * row + col + 1] = 'X'
    return pos
```

BlocksWorldProblem.py

```
from TreeNode import TreeNode


class BlocksWorldProblem:

    def __init__(self, blocks_world):
        self.start_state = blocks_world

    def get_start_state(self):
        return self.start_state

    def is_goal_state(self, blocks_world):
        return blocks_world.is_goal()

    def get_successors(self, node):
        successors = []
        for direction in node.state.get_legal_move():
            new_state = node.state.get_new_state(direction)
            new_node = TreeNode(new_state, node, node.depth + 1, node.action +
                [direction], node.cost + 1)
            successors.append(new_node)
        return successors
```

BlocksWorldStates.py

```
start_state_depth_1 = {
    6: "A",
    9: "B",
    14: "C",
    10: "X"
}
start_state_depth_2 = {
    10: "A",
    9: "B",
    14: "C",
    6: "X"
}
start_state_depth_3 = {
    10: "A",
    9: "B",
    14: "C",
    7: "X"
}
start_state_depth_4 = {
    10: "A",
    9: "B",
    14: "C",
    11: "X"
}
start_state_depth_5 = {
    10: "A",
    9: "B",
    14: "C",
    15: "X"
}
start_state_depth_6 = {
    10: "A",
    9: "B",
    15: "C",
    14: "X"
}
start_state_depth_7 = {
    14: "A",
    9: "B",
    15: "C",
    10: "X"
}
start_state_depth_8 = {
    14: "A",
    10: "B",
    15: "C",
```

```
    9: "X"
}
start_state_depth_9 = {
    14: "A",
    10: "B",
    15: "C",
    13: "X"
}
start_state_depth_10 = {
    13: "A",
    10: "B",
    15: "C",
    14: "X"
}
start_state_depth_11 = {
    13: "A",
    14: "B",
    15: "C",
    10: "X"
}
start_state_depth_12 = {
    13: "A",
    14: "B",
    15: "C",
    11: "X"
}
start_state_depth_13 = {
    13: "A",
    14: "B",
    15: "C",
    12: "X"
}
start_state = {
    13: "A",
    14: "B",
    15: "C",
    16: "X"
}  # depth 14

all_start_states = [start_state_depth_1, start_state_depth_2,
    start_state_depth_3, start_state_depth_4,
                    start_state_depth_5, start_state_depth_6, start_state_depth_7,
                        start_state_depth_8,
                    start_state_depth_9, start_state_depth_10,
                        start_state_depth_11, start_state_depth_12,
                    start_state_depth_13, start_state]
```

SinglePlayer.py

```
import os

from BlocksWorld import BlocksWorld


def cls():
    os.system('cls' if os.name == 'nt' else 'clear')


start_state = {
    13: "A",
    14: "B",
    15: "C",
    16: "X"
}

size = int(input('Please select board size: '))
blocks_world = BlocksWorld(size, start_state)

while not blocks_world.is_goal():
    blocks_world.display_board()
    print('Movable direction: ' + ', '.join(blocks_world.get_legal_move()))
    user_input = input('Please input direction (w = up, a = left, s = down, d =
        right)\n')
    direction = ''
    if user_input == 'w':
        direction = 'UP'
    elif user_input == 'a':
        direction = 'LEFT'
    elif user_input == 's':
        direction = 'DOWN'
    elif user_input == 'd':
        direction = 'RIGHT'

    if direction != '':
        blocks_world.move(direction)
    cls()

blocks_world.display_board()
print("You have won")
```

Search.py

```
import random
import time

import util
from BlocksWorld import BlocksWorld
from BlocksWorldProblem import BlocksWorldProblem
from BlocksWorldStates import start_state, all_start_states
from Solution import Solution
from TreeNode import TreeNode


def breadth_first_search(problem):
    print('Running breadth-first search...')
    node_expanded = 0
    fringe = util.Queue()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node)
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        for new_node in problem.get_successors(node):
            fringe.push(new_node)


def depth_first_search(problem):
    print('Running depth-first search...')
    node_expanded = 0
    fringe = util.Stack()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node)
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        successors = problem.get_successors(node)
        random.shuffle(successors) # Randomise the order of expansion in DFS
        for new_node in successors:
```

```
            fringe.push(new_node)


def iterative_deepening_search(problem):
    print('Running iterative deepening search...')
    node_expanded = 0
    depth = 0
    while True:
        result = depth_limited_search(problem, depth)
        node_expanded += result.total_nodes
        if result.path != "cutoff":
            return Solution(result.path, node_expanded, result.depth, result.cost)
        depth += 1


def depth_limited_search(problem, limit):
    node_expanded = 0

    def recursive_dls(node):
        nonlocal node_expanded
        node_expanded += 1
        cutoff_occurred = False
        cutoff = "cutoff"
        failure = "fail"
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded, node.depth, node.cost)
        elif node.depth == limit:
            return Solution(cutoff, node_expanded, node.depth, node.cost)
        else:
            for new_node in problem.get_successors(node):
                result = recursive_dls(new_node)
                if result.path == cutoff:
                    cutoff_occurred = True
                elif result.path != failure:
                    return result
        if cutoff_occurred:
            return Solution(cutoff, node_expanded, node.depth, node.cost)
        else:
            return Solution(failure, node_expanded, node.depth, node.cost)

    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    return recursive_dls(initial_node)


def a_star_search(problem, heuristic):
    print('Running A star heuristic search...')
    node_expanded = 0
```

```
    fringe = util.PriorityQueue()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node, heuristic(problem.get_start_state()))
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        for new_node in problem.get_successors(node):
            fringe.push(new_node, new_node.cost + heuristic(new_node.state))


def manhattan_heuristic(board):
    distance = 0
    block_pos = board.get_block_pos()
    for key, value in block_pos.items():
        distance += manhattan_distance(value, board.get_goal_state()[key])
    return distance


def manhattan_distance(xy1, xy2):
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])


def breadth_first_search_graph(problem):
    print('Running breadth-first graph search...')
    node_expanded = 0
    closed = set()
    fringe = util.Queue()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node)
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        if node.state not in closed:
            closed.add(node.state)
            for new_node in problem.get_successors(node):
                fringe.push(new_node)
```

```
def depth_first_search_graph(problem):
    print('Running depth-first graph search...')
    node_expanded = 0
    closed = set()
    fringe = util.Stack()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node)
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        if node.state not in closed:
            closed.add(node.state)
            successors = problem.get_successors(node)
            random.shuffle(successors) # Randomise the order of expansion in DFS
            for new_node in successors:
                fringe.push(new_node)


def a_star_search_graph(problem, heuristic):
    print('Running A star heuristic graph search...')
    node_expanded = 0
    closed = set()
    fringe = util.PriorityQueue()
    initial_node = TreeNode(problem.get_start_state(), None, 0, [], 0)
    fringe.push(initial_node, heuristic(problem.get_start_state()))
    while True:
        if fringe.is_empty():
            return Solution([], node_expanded, -1, -1) # No solution found
        node = fringe.pop()
        node_expanded += 1
        if problem.is_goal_state(node.state):
            return Solution(node.action, node_expanded + fringe.size(),
                node.depth, node.cost)
        if node.state not in closed:
            closed.add(node.state)
            for new_node in problem.get_successors(node):
                fringe.push(new_node, new_node.cost + heuristic(new_node.state))


def print_path_grid(path):
    for p in path:
```

```
        print(p)
        blocks_world.move(p)
        blocks_world.display_board()


# Code to run the search problem
grid_size = 4
blocks_world = BlocksWorld(grid_size, start_state)
blocks_world.display_board()
search_problem = BlocksWorldProblem(blocks_world)
start = time.time()
solution = a_star_search(search_problem, manhattan_heuristic)
# solution = depth_first_search(search_problem)
end = time.time()
print('Found result:', solution.path)
print('Total elapse time in second:', end - start)
print('Total node generated:', solution.total_nodes)
print('Depth of the solution:', solution.depth)
print('Total cost from start to goal state:', solution.cost)
print_path_grid(solution.path)


# # Code to get the number of node generated with increasing problem difficulty
# nodes_generated = []
# for start in all_start_states:
#     blocks_world = BlocksWorld(grid_size, start)
#     search_problem = BlocksWorldProblem(blocks_world)
#     # blocks_world.display_board()
#     solution = depth_first_search(search_problem)
#     print('Found result:', solution.path)
#     print('Total node generated:', solution.total_nodes)
#     nodes_generated.append(solution.total_nodes)
#     print(nodes_generated)
```

TreeNode.py

```
class TreeNode:

    def __init__(self, state, parent, depth, action, cost):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.action = action
        self.cost = cost
```

Solution.py

```
class Solution:

    def __init__(self, path, total_nodes, depth, cost):
        self.path = path
        self.total_nodes = total_nodes
        self.depth = depth
        self.cost = cost
```

util.py

```
import heapq
from collections import deque


class Queue:

    def __init__(self):
        self.queue = deque()

    def push(self, item):
        self.queue.append(item)

    def pop(self):
        return self.queue.popleft()

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

class Stack:

    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        return self.stack.pop()

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)


class PriorityQueue:

    def __init__(self):
        self.queue = []
        self.count = 0

    def push(self, item, priority):
```

```
        entry = (priority, self.count, item)
        heapq.heappush(self.queue, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.queue)
        return item

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)
```

SearchPlot.py

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.ticker import ScalarFormatter

a_star_nodes_generated = [1, 5, 9, 23, 59, 99, 157, 401, 889, 1487, 5743, 15311,
    35764, 62869, 84242]
ids_nodes_generated = [1, 5, 15, 67, 133, 239, 1648, 6574, 17269, 23877, 156943,
    699291, 2816096, 6602826, 8943144]
bfs_nodes_generated = [1, 12, 35, 156, 306, 527, 3676, 14730, 38595, 53343,
    350915, 1563733, 6297070, np.nan, np.nan]
dfs_nodes_generated = [1, 4250, 68237, 82798, 159894, 51820, 111950, 284777,
    159572, 99462, 143527, 192137, 585214,
                    np.nan, np.nan]
x = range(0, 15)
fig, ax = plt.subplots(figsize=(10, 7))
ax.plot(x, a_star_nodes_generated, label='A*')
ax.plot(x, ids_nodes_generated, label='IDS')
ax.plot(x, bfs_nodes_generated, label='BFS')
ax.plot(x, dfs_nodes_generated, label='DFS')
ax.yaxis.set_major_formatter(ScalarFormatter())
ax.grid(True)
ax.set_yscale('log')
for axis in [ax.xaxis, ax.yaxis]:
    formatter = ScalarFormatter()
    formatter.set_scientific(False)
    axis.set_major_formatter(formatter)
ax.legend()
ax.set_title("Scalability of Search Methods", fontsize=14)
ax.set_xlabel("Problem Difficulty (Optimal Solution Depth)", fontsize=12)
ax.set_ylabel("Nodes Generated (Log Scale)", fontsize=12)

fig.show()
```