UNIVERSITY OF Southampton

School of Electronics and Computer Science

Foundations of Artificial Intelligence

Blocksworld: Search Algorithms

Author:
Subash Poudyal
sp1g19@soton.ac.uk
28395352

Moderators:
Dr. Richard Watson
Dr. Enrico Marchioni

Contents

Nomenclature

BFS : Breadth First Search
DFS : Depth First Search
IDS : Iterative Deepening (Depth First) Search

1 Approach

After a number of design iterations focusing on the generic code structure to solve the problem along with its adaptability, I settled on the following design. Keeping separation of concern in mind, I divided my code structure intro three scripts. One class and the others two merely to make use of that main class: Node. As my favourite programming language and the fact that this problem could be solved with functional-OOP hybrid approach, my language of choosing was Python. Blocksworld is a puzzle that examines the foundations of computer science: data structures and algorithms. Therefore, I've not made use of any external libraries making the only dependency Python runtime environment itself.

1.1 Node

Programming a generic structure for the game (the world) was interesting. Although the problem seemed simple, there were a lot of nitty-gritty details to think about such as the size of the grid and starting positions of the blocks; how to make them as adaptable as possible while keeping the core stable enough for an efficient program. For this, I made a class called Node (*Node.py*). Node is the world that the puzzle is contained in. And this world is represented by a 2D array containing the blocks: 'A', 'B', 'C' and asterisk '*'.

Although my first thought was to use a single array to store all 16 blocks, I realised it would essentially make no difference in look up times. Dictionaries are the best data structures to go with when it comes to the efficiency of look up times, however, the memory occupied by the dictionary was unnecessarily big. Node class also housed methods such as find_block, check_goal_state and move_agent.

To solve this puzzle, the agent (empty space) is thought to be an element that moves around the world rather than all other blocks moving their positions. This makes the idea simpler when programming. As Python uses 'pass by reference' approach, I had to make use of inbuilt library: deepcopy. Node class was my favourite part of the system to code as it housed the integral logics of the Blocksworld. For example, what if the agent was at the bottom right corner and it was asked to move down or right? These logical tasks made designing the class enjoyable.

As an extra, I also made the world adaptable blocked nodes (represented by 'X'). i.e. blocks that cannot be moved, increasing or decreasing the difficulty of search in some cases.

1.2 Search

Search script is where all the search algorithms are implemented. Unlike Node, this is not a class but just a group of functions. In order to keep things simple yet effective for testing, I implemented search algorithms as independent functions that made use of the Node class.

All searches are implemented in a similar way; requiring parameter of the initial / start node. I've implemented 7 different searches; 4 tree searches (BFS, DFS, Iterative Deepening and A*) and 3 graph search variants of BFS, DFS and A*.

While all search methods are similar, they make use of different data structures depending on the algorithm to keep efficiency to maximum. BFS uses a queue, a first in first out (FIFO) structure in order to store the nodes and the subsequent children nodes. Although the two algorithms are very similar, DFS on the other hand uses stacks where last in first out (LIFO) is followed. Implementation of IDS was quite easy given the implementation of DFS. IDS is essentially DFS but repeated with a limit on the depth expanded on.

A* search, however, was quite different to all the other searches. While all the others are examples of uninformed searches, A* uses heuristic to navigate its way through to the goal state; better the heuristic, better the performance. To store the nodes, A* uses a priority queue where the nodes in the queue are ordered by their associated value of the heuristic. This is by far the best tree search algorithm amongst the 4 mentioned above.

Apart from Node and Search, my only other script is Test where I call functions from *Search.py* in order to solve the puzzle.

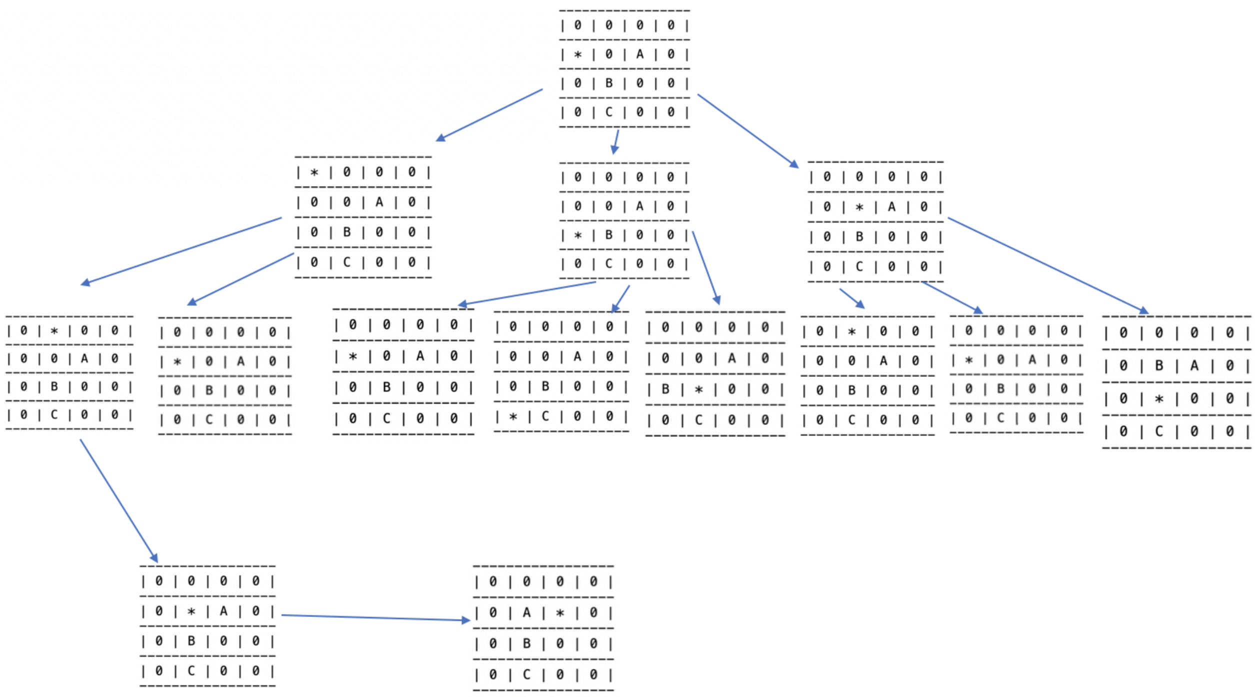
2 Evidence

In this section, I'll present working evidence of each algorithm going through how the choices were made and briefly explaining how the algorithm works as a whole. Due to some algorithms being less efficient than others in nature, I will be using example with optimal depth of 2 (specification provided initial state to goal state is of optimal depth 14). It is quite difficult to include all steps (children nodes for each node) in this report, hence, I'll only be showing few at the start and end respectively. As mentioned before, world is represented by a board containing the blocks: 'A', 'B', 'C' and asterisk '*'. Blank blocks are represented by '0' while the walls are represented by dashes '—'.

2.1 BFS and BFS Graph

In Breadth First Search, nodes are expanded in the order they're added to the fringe. On each depth, it'll expand all nodes and check for goal state, if not found, it'll move into the next depth repeating this iterative process. Evidence for depth 2 can be seen below.

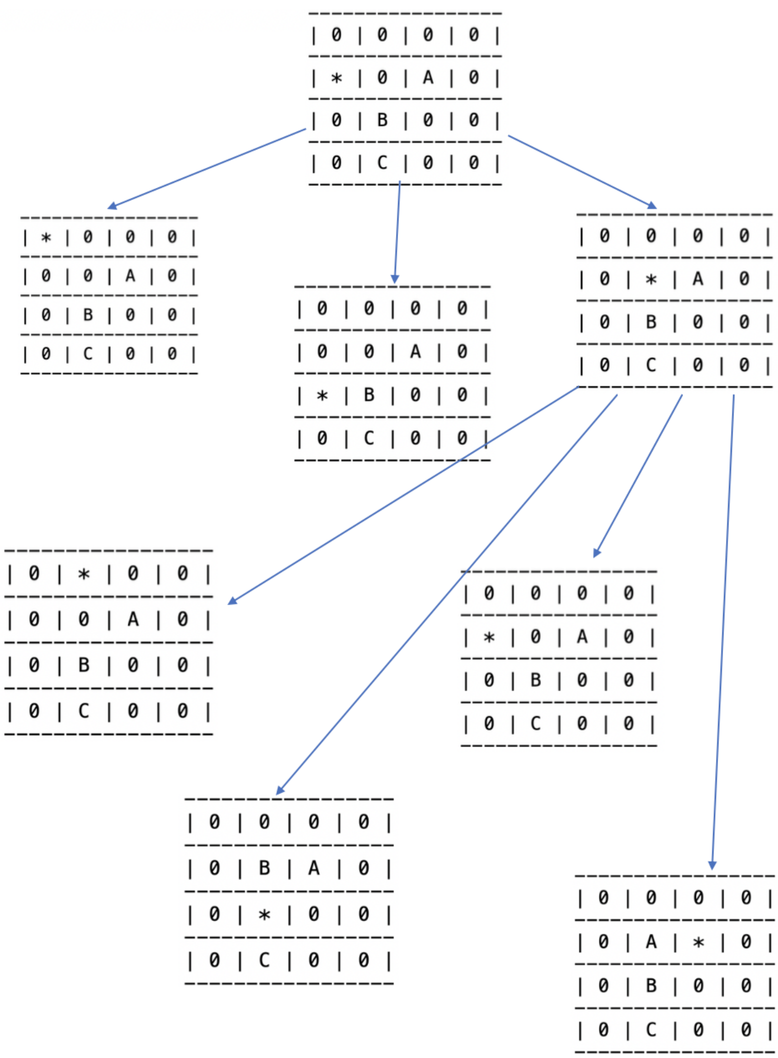
Graph search for BFS would mean that we store the visited nodes and upon each node expansion, we only add nodes to the fringe that have not been visited yet. This makes the search faster. While running for depth 14, on average **BFS nodes expanded: 5182280** and **BFS Graph nodes expanded: 3469**.



*Other steps were ignored due to space

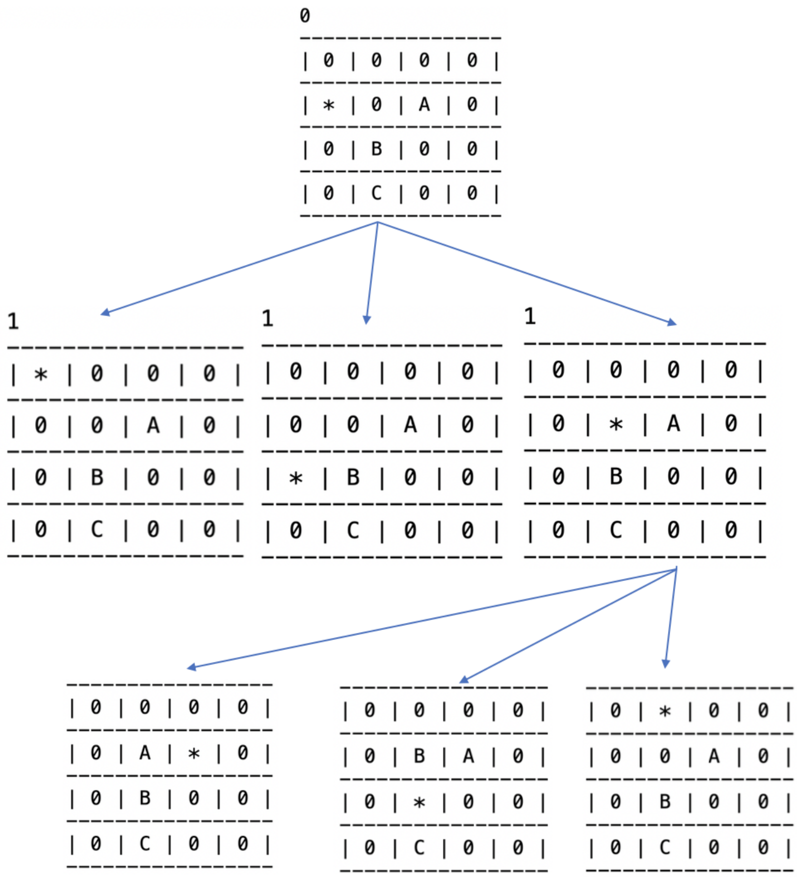
2.2 DFS and DFS Graph

Depth First Search always expands the deepest node in the fringe (last added) of the search tree. When goal state is not found, another deepest node is expanded. Similar to BFS Graph, when the visited nodes are stored, search becomes faster; reducing the nodes expanded, trading off space for time. While running for depth 14, on average **DFS nodes expanded: 29159** and **DFS Graph nodes expanded: 11607**.



2.3 IDS

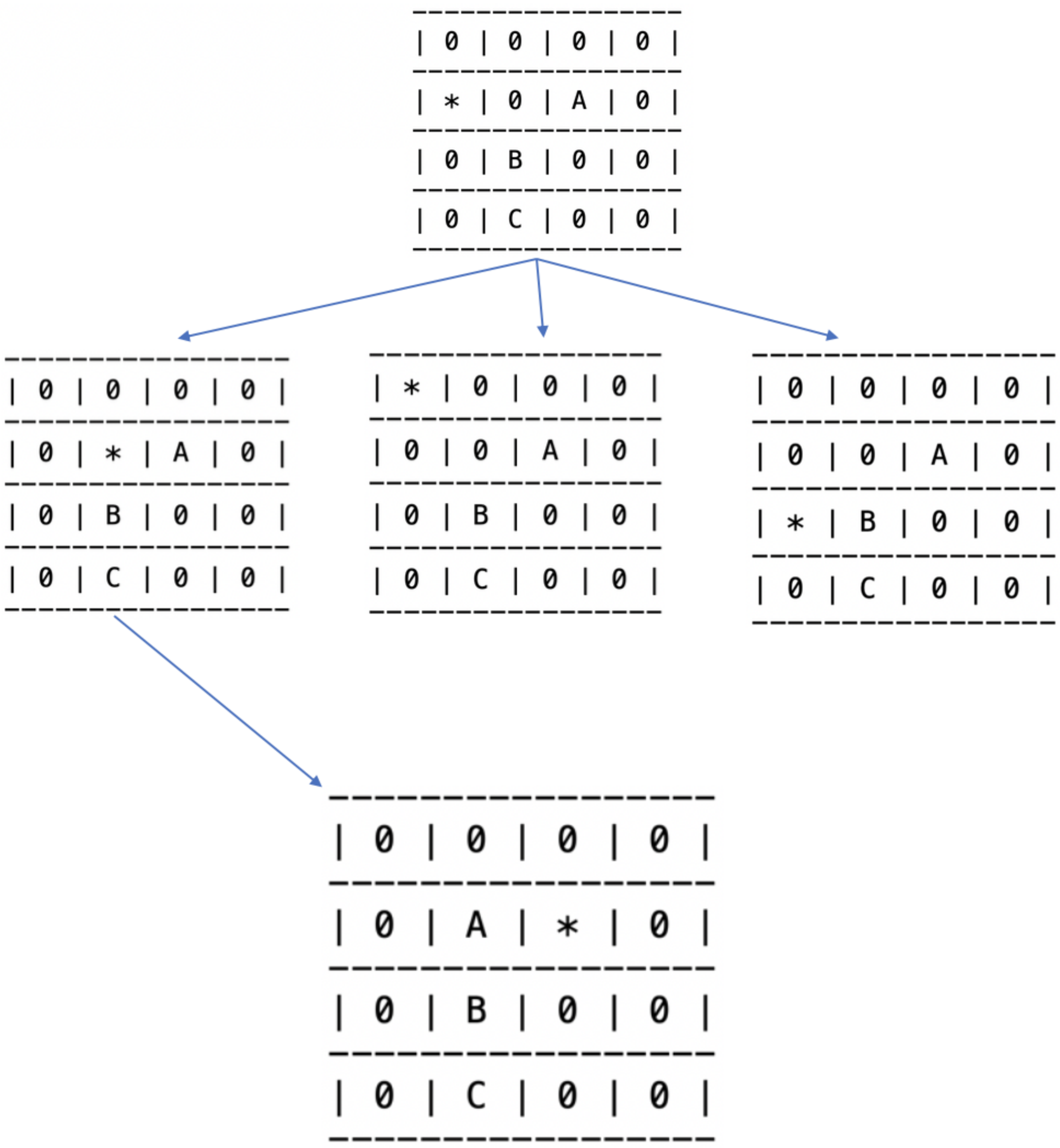
Iterative Deepening Search is often used with DFS but with a slightly restricted approach. By limiting the depth in DFS, IDS gradually performs depth-limited DFS increasing the depth until the goal is found. IDS combines the benefits of BFS and DFS [1]. While running for depth 14, on average **IDS nodes expanded: 1622898**.



2.4 A* Tree Search and A* Graph Search

Unlike other searches mentioned above, A* search is an informed search that combines the cost to reach the node and the cost to get from the node to the goal. A* search is known to be both complete and optimal. For the heuristic, I implemented manhattan distance, this is used to calculate the cost function. Manhattan distance is simply the shortest distance from one tile to the other; used to find the misplaced tiles. Alternatively, Euclidean distance could be used, however, as our agent cannot move in diagonal directions, it wouldn't be practical.

To make A* even better, like before, I implemented graph search, storing the nodes that were visited. While running for depth 14, on average **A* nodes expanded: 20292** and **A* Graph nodes expanded: 374**.



3 Scalibility

In order to study how the algorithms scale as the complexity of the problem grows, I created 14 different initial state; each with different optimal depths ranging from 1 to 14 depths away from the goal state. In order to create these states, I used A* search as it is both optimal and complete, storing a state at each depth until goal state was reached.

```
Depth:  1
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | * | A | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| 0 | C | 0 | 0 |
-----------------

Depth:  2
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| * | 0 | A | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| 0 | C | 0 | 0 |
-----------------

Depth:  3
-----------------
| * | 0 | 0 | 0 |
-----------------
| 0 | 0 | A | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| 0 | C | 0 | 0 |
-----------------

Depth:  4
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | 0 | A | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| * | C | 0 | 0 |
-----------------

Depth:  5
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| A | 0 | 0 | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| 0 | C | 0 | * |
-----------------

Depth:  6
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | 0 | A | 0 |
-----------------
| 0 | B | 0 | 0 |
-----------------
| C | 0 | * | 0 |
-----------------

Depth:  7
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | 0 | A | 0 |
-----------------
| B | 0 | 0 | 0 |
-----------------
| 0 | C | 0 | * |
-----------------

Depth:  8
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | A | 0 | 0 |
-----------------
| 0 | C | 0 | B |
-----------------
| 0 | 0 | 0 | * |
-----------------

Depth:  9
-----------------
| * | 0 | 0 | 0 |
-----------------
| 0 | 0 | A | 0 |
-----------------
| B | 0 | 0 | 0 |
-----------------
| 0 | 0 | C | 0 |
-----------------

Depth:  10
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | A | 0 | 0 |
-----------------
| 0 | 0 | 0 | B |
-----------------
| C | 0 | 0 | * |
-----------------

Depth:  11
-----------------
| 0 | A | 0 | 0 |
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | B | C | 0 |
-----------------
| 0 | 0 | 0 | * |
-----------------

Depth:  12
-----------------
| A | 0 | 0 | 0 |
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| B | 0 | 0 | * |
-----------------
| 0 | 0 | C | 0 |
-----------------

Depth:  13
-----------------
| A | 0 | 0 | 0 |
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | B | C | 0 |
-----------------
| 0 | 0 | 0 | * |
-----------------

Depth:  14
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| 0 | 0 | 0 | 0 |
-----------------
| A | B | C | * |
-----------------
```

As most of the search algorithms implemented are random and uninformed, I created a test (Test.py) where all uninformed searches were run 10 times per depth, resulting in 140 runs of each search. Shown below are the states for each depth and in this section, I present results of how increase in depth affects the number of nodes expanded (averaged over the 10 runs).

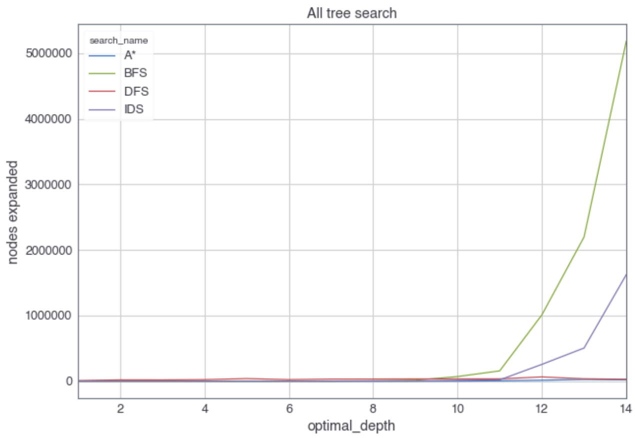


Figure 1: All tree searches compared

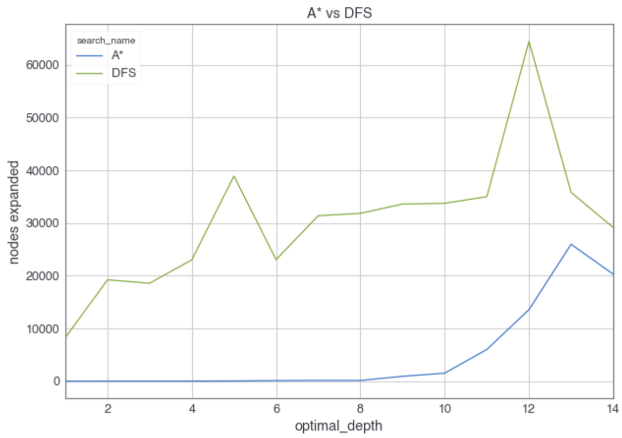


Figure 2: A* vs DFS

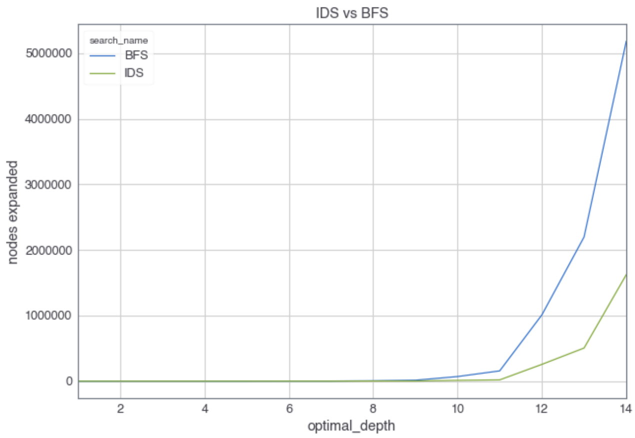


Figure 3: IDS vs BFS

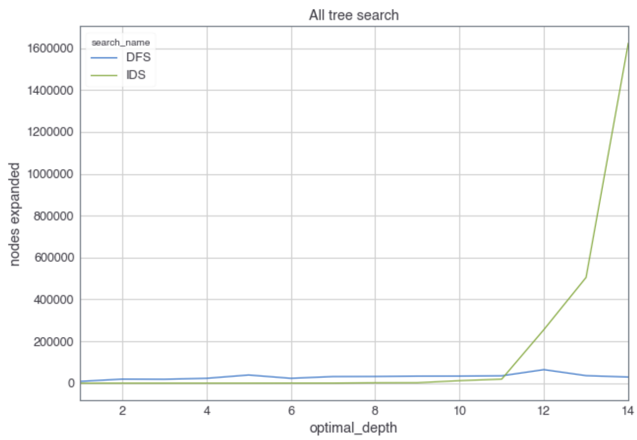


Figure 4: DFS vs IDS

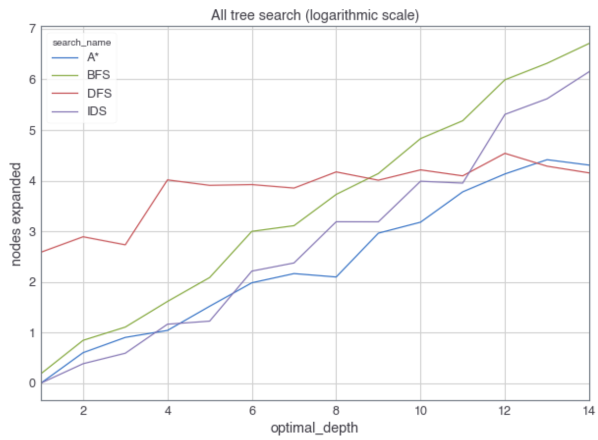


Figure 5: Comparison of all tree search

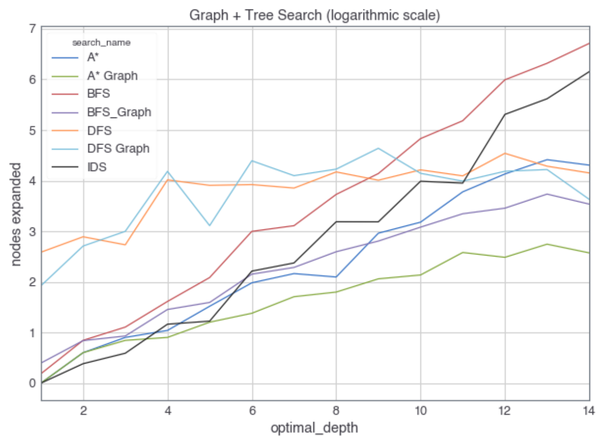


Figure 6: Comparison of all search results

It is clearly visible from figure 1 that after depth 10, BFS and IDS start getting heavily affected by the number of nodes expanded. In other words, their number of nodes expanded starts growing at a very rapid rate. As their number of nodes expanded are so large, we are unable to see the impact on other searches such as A* and DFS. Hence, figure 5 shows the comparisons in node expanded; A* vs DFS. I also created graphs for some more comparisons between graph search and tree search. To visualise the exponential expansion nature of some search methods, I've used logarithmic scale for some graphs. Below, I compare different search methods in more detail.

3.1 BFS vs IDS

From figure 1, we can see that IDS and BFS are the most dominating (in terms of nodes expanded) search methods amongst the others. BFS and IDS are quite similar in terms of performance and this can be seen from the test results as they both follow a close trend; proving that their time complexity is similar. However, although we cannot conclude from the graphs presented above, space complexity of the two algorithms are different. While BFS stores the whole tree, IDS only stores (as an estimate) the branch nodes, making IDS more memory efficient than BFS.

3.2 DFS vs A*

Comparing an informed search with uninformed search strategy isn't fair. However, in the search methods explored and implemented, A* approaches the time complexity of DFS around the depth of goal state (depth 14). A* in general has great time and space complexity, however, for this particular problem, the heuristic is not the best. Although I make use of Manhattan distance as the heuristic method, because of the fact that the end position of agent tile doesn't matter, we cannot have a "perfect" heuristic. For this reason, I believe if we were to test A* against DFS for more complex problem (higher depth), DFS may outperform A* search. However, from the results observed, A* is clearly better than DFS and for that matter, any other algorithms.

3.3 IDS vs DFS

From figure 6, we can see that IDS is superior to DFS but that only holds true until depth 11. IDS seems to exponentially grow after the depth of 11 or so. DFS also has a problem; when the search space is large and there is only one solution. These are both not the best search algorithms we have but in this case, DFS seems to be better because of the time complexity (compared to IDS, DFS looks to have constant time).

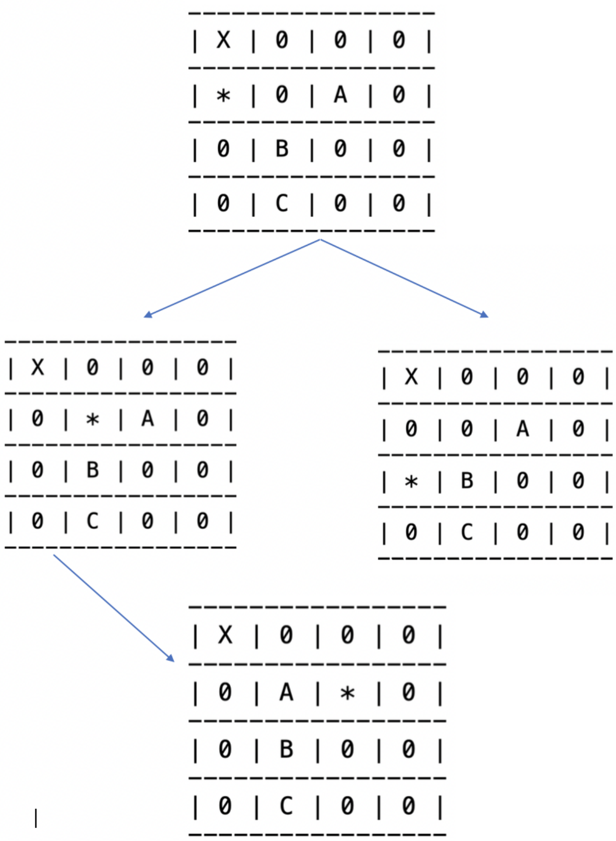
3.4 Tree vs Graph

In general, we notice that graph search is faster than tree search however, graph search requires more memory, as it keeps track of all visited states. In order to optimise memory, I stored all the visited nodes in a set which ensures no duplicates are contained.

4 Extras and Limitations

If I had more time, I'd have liked to explore the problem further, increasing the depth to more than 14 to compare different search strategies. I'd also include more searches such as bidirectional search and most interestingly, Iterative deepening A* search as it claims to "keep the memory usage lower than in A*" [2]. In terms of the Blocksworld puzzle itself, I'd have liked if the search problem wasn't limited to a 4 x 4 grid; I'd make the grid adaptable to any sizes. Although heuristics other than Manhattan distance was studied, implementation of Chebyshev distance [3] was not functional, I'd have loved to expand on that to see the difference in performance against manhattan distance.

Other than what was required by the coursework specification, I implemented graph searches to be compared against tree searches. It was helpful to visualise how space time complexity trade off would result in better performance in certain cases. I also added blocks in the world which could not be moved (shown below). This, based on the placement made the search problem sometimes more complex and easier the other times. If I had more time I'd have compared the performance with and without blocks with different algorithms.



References

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach.* Malaysia; Pearson Education Limited,, 2016.

[2] M. Nosrati, R. Karimi, and H. A. Hasanvand, "Investigation of the*(star) search algorithms: Characteristics, methods and approaches," *World Applied Programming*, vol. 2, no. 4, pp. 251–256, 2012.

[3] H. Mora, J. M. Mora-Pascual, A. Garcia-Garcia, and P. Martinez-Gonzalez, "Computational analysis of distance operators for the iterative closest point algorithm," *PloS one*, vol. 11, no. 10, p. e0164694, 2016.

5 Code

All code snippets and data analysis can be found at: https://github.com/subash774/Blocksworld

5.1 Node.py

```
import copy
import random

class Node:

    def __init__(self, board, agent_row, agent_column, parent, path):
        self.board = board
        self.agent_row = agent_row
        self.agent_column = agent_column
```

```
        self.parent = parent
        if parent is not None:
            self.depth = self.parent.depth + 1
        else:
            self.depth = 0
        self.path = path


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


    def __eq__(self, other):
        return str(self.board) == str(other.board)


    def __lt__(self, node2):
        if (self.depth < node2.depth) and (self.get_man_heuristic() < node2.get_man_he
            return True
        else:
            return False


    def find_block(self, board, block):
        for i in range(len(board)):
            if block in board[i]:
                return i, board[i].index(block)
            else:
                continue


    def check_goal(self, board):
        return board[3][1] == "C" and board[2][1] == "B" and board[1][1] == "A"


    def print_board(self, board):
        print("-----------------")
        for i in board:
            vertical_dash = "| "
            for j in i:
                vertical_dash = vertical_dash + str(j) + " | "
            print(vertical_dash)
            print("-----------------")
        print("")



    def check_moves(self, agent_row, agent_column, board):
        possible = {
            "up" : True,
            "down" : True,
            "left" : True,
            "right" : True
        }
        if agent_row - 1 < 0 or board[agent_row][agent_column] == "X":
            possible["up"] = False
        if agent_row + 1 > 3 or board[agent_row][agent_column] == "X":
            possible["down"] = False
```

```
        if agent_column - 1 < 0 or board[agent_row][agent_column] == "X":
            possible["left"] = False
        if agent_column + 1 > 3 or board[agent_row][agent_column] == "X":
            possible["right"] = False

        return possible


    def move_agent(self, new_board, direction):
        agent_row, agent_column = self.find_block(new_board, "*")
        if self.check_moves(agent_row, agent_column, new_board).get(direction):
            if direction == "up":
                # swap the agent with upper block
                new_board[agent_row][agent_column], new_board[agent_row - 1][agent_col

            if direction == "down":
                # swap the agent with lower block
                new_board[agent_row][agent_column], new_board[agent_row + 1][agent_col

            if direction == "left":
                # swap the agent with left block
                new_board[agent_row][agent_column], new_board[agent_row][agent_column

            if direction == "right":
                # swap the agent with right block
                new_board[agent_row][agent_column], new_board[agent_row][agent_column

            agent_row, agent_column = self.find_block(new_board, "*")
            return Node(new_board, agent_row, agent_column, self, direction)
        else:
            return None


    def get_children_nodes(self):
        children_nodes = []
        directions = ["up", "left", "right", "down"]
        for direction in directions:
            new_board = copy.deepcopy(self.board)
            new_node = self.move_agent(new_board, direction)
            if new_node is not None:
                children_nodes.append(new_node)
            else:
                continue
        random.shuffle(children_nodes)
        return children_nodes


    def get_man_heuristic(self):
        a = self.find_block(self.board, "A")
        b = self.find_block(self.board, "B")
        c = self.find_block(self.board, "C")



        man_dist = (abs(a[0] - 1) + abs(a[1] - 1)
                    + abs(b[0] - 2) + abs(b[1] - 1)
                    + abs(c[0] - 3) + abs(c[1] - 1))

```

```
        return man_dist


    def get_cheb_heuristic(self):
        a = self.find_block(self.board, "A")
        b = self.find_block(self.board, "B")
        c = self.find_block(self.board, "C")

        cheb_dist = (max(abs(a[1] - b[1]),
                        abs(b[1] - c[1]),
                        abs(a[1] - c[1]))
                        *
                        max(abs(a[0] - b[0]),
                        abs(b[0] - c[0]),
                        abs(a[0] - c[0])))

        return (cheb_dist + self.get_man_heuristic())/2
```

5.2 Search.py

```
from Node import Node
from collections import deque
from queue import PriorityQueue as Q
import time

def dfs(start_node):
    fringe_nodes = [start_node] # Stack of nodes
    nodes_expanded = 0
    while True:
        if len(fringe_nodes) == 0:
            print("Solution not found")
            return None

        node = fringe_nodes.pop()

        if node.check_goal(node.board):
            node.print_board(node.board)
            # Search name, depth, nodes expanded
            return ["DFS", node.depth, nodes_expanded]

        node.print_board(node.board)
        children_nodes = node.get_children_nodes()
        for child in children_nodes:
            fringe_nodes.append(child)
        nodes_expanded += 1


def dfs_graph(start_node):
    fringe_nodes = [start_node] # Stack of nodes
    nodes_expanded = 0
    visited_nodes = {}

    while True:
        if len(fringe_nodes) == 0:
            print("Solution not found")
            return None

        node = fringe_nodes.pop()

```

```
        if node.check_goal(node.board):
            # Search name, depth, nodes expanded
            return ["DFS Graph", node.depth, nodes_expanded]

        node.print_board(node.board)
        children_nodes = node.get_children_nodes()
        for child in children_nodes:
            if visited_nodes.get(str(child.board)) is None:
                fringe_nodes.append(child)

        nodes_expanded += 1

def bfs(start_node):
    nodes_expanded = 0
    fringe_nodes = deque([]) # Queue of nodes
    fringe_nodes.append(start_node)
    t_end = time.time() + 60 * 15 # 15 min time limit
    while True:
        if time.time() < t_end:
            if len(fringe_nodes) == 0:
                print("Solution not found")
                return None

            node = fringe_nodes.popleft()

            if node.check_goal(node.board):
                # Search name, depth, nodes expanded
                node.print_board(node.board)
                return ["BFS", node.depth, nodes_expanded]

            node.print_board(node.board)
            for child in node.get_children_nodes():
                fringe_nodes.append(child)

            nodes_expanded += 1


        else:
            return None


def bfs_graph(start_node):
    fringe_nodes = deque([]) # Queue of nodes
    visited_nodes = {}
    fringe_nodes.append(start_node)
    nodes_expanded = 0

    while True:
        if len(fringe_nodes) == 0:
            print("Solution not found")
            return None

        node = fringe_nodes.popleft()
        visited_nodes[str(node.board)] = 1

        if node.check_goal(node.board):
            # Search name, depth, nodes expanded
            node.print_board(node.board)
```

```
                return ["BFS_Graph", node.depth, nodes_expanded]

            node.print_board(node.board)

            for child in node.get_children_nodes():
                if visited_nodes.get(str(child.board)) is None:
                    fringe_nodes.append(child)

            nodes_expanded += 1


def depth_limited(start_node, depth):
    fringe_nodes = [start_node] # Stack of nodes
    nodes_expanded = 0

    if len(fringe_nodes) == 0:
        print("Solution not found")
        return None

    while len(fringe_nodes) > 0:
        node = fringe_nodes.pop()
        if node.check_goal(node.board):
            # Search name, depth, nodes expanded
            node.print_board(node.board)
            return node, nodes_expanded, True

        if node.depth < depth:
            node.print_board(node.board)
            children_nodes = node.get_children_nodes()
            for child in children_nodes:
                fringe_nodes.append(child)

            nodes_expanded += 1


def iterative_deepening(start_node, limit):
    for i in range(limit):
        print(i)
        node, nodes_expanded, goal = depth_limited(start_node, limit)
        if goal:
            break
    return ["IDS", node.depth, nodes_expanded]


def a_star_graph(start_node, h):
    visited_nodes = {}
    fringe_nodes = Q()
    nodes_expanded = 0

    fringe_nodes.put((start_node.get_man_heuristic(), start_node))

    if fringe_nodes.qsize() == 0:
        print("Solution not found")
        return None

    while fringe_nodes.qsize()  > 0:
        node = fringe_nodes.get()[1]
        visited_nodes[str(node.board)] = 1
```

```

        node.print_board(node.board)

        if node.check_goal(node.board):
            # Search name, depth, nodes expanded
            node.print_board(node.board)
            return ["A* Graph", node.depth, nodes_expanded]

        node.print_board(node.board)
        children = node.get_children_nodes()
        nodes_expanded += 1

        for child in children:
            if visited_nodes.get(str(child.board)) is not None:
                continue
            if h == "m":
                fringe_nodes.put((child.depth + child.get_man_heuristic(), child))
            if h == "c":
                fringe_nodes.put((child.depth + child.get_cheb_heuristic(), child))


def a_star(start_node, h):
    fringe_nodes = Q()
    nodes_expanded = 0
    fringe_nodes.put((start_node.get_man_heuristic(), start_node))

    if fringe_nodes.qsize() == 0:
        print("Solution not found")
        return None

    while fringe_nodes.qsize() > 0:
        node = fringe_nodes.get()[1]

        if node.check_goal(node.board):
            node.print_board(node.board)
            return ["A*", node.depth, nodes_expanded]

        node.print_board(node.board)
        children = node.get_children_nodes()
        nodes_expanded += 1

        for child in children:
            if h == "m":
                fringe_nodes.put((child.depth + child.get_man_heuristic(), child))
            if h == "c":
                fringe_nodes.put((child.depth + child.get_cheb_heuristic(), child))
```

5.3 Test.py

```
from Node import Node
from Search import dfs, dfs_graph, bfs, bfs_graph, iterative_deepening, a_star, a_
import time


states = []

states.append([[0,0,0,0],[0,'*','A',0],[0,'B',0,0],[0,'C',0,0]]) #depth 1
states.append([[0,0,0,0],['*',0,'A',0],[0,'B',0,0],[0,'C',0,0]]) #depth 2
states.append([['*',0,0,0],[0,0,'A',0],[0,'B',0,0],[0,'C',0,0]]) #depth 3
```


UNIVERSITY OF Southampton

```
states.append([[0,0,0,0],[0,0,'A',0],[0,'B',0,0],['*','C',0,0]]) #depth 4
states.append([[0,0,0,0],['A',0,0,0],[0,'B',0,0],[0,'C',0,'*']]) #depth 5
states.append([[0,0,0,0],[0,0,'A',0],[0,'B',0,0],['C',0,'*',0]]) #depth 6
states.append([[0,0,0,0],[0,0,'A',0],['B',0,0,0],[0,'C',0,'*']]) #depth 7
states.append([[0,0,0,0],[0,'A',0,0],[0,'C',0,'B'],[0,0,0,'*']]) #depth 8
states.append([['*',0,0,0],[0,0,'A',0],['B',0,0,0],[0,0,'C',0]]) #depth 9
states.append([[0,0,0,0],[0,'A',0,0],[0,0,0,'B'],['C',0,0,'*']]) #depth 10
states.append([[0,'A',0,0],[0,0,0,0],[0,'B','C',0],[0,0,0,'*']]) #depth 11
states.append([['A',0,0,0],[0,0,0,0],['B',0,0,'*'],[0,0,'C',0]]) #depth 12
states.append([['A',0,0,0],[0,0,0,0],[0,'B','C',0],[0,0,0,'*']]) #depth 13
states.append([[0,0,0,0],[0,0,0,0],[0,0,0,0],['A','B','C','*']]) #depth 14


def find_block(board, block):
        for i in range(len(board)):
            if block in board[i]:
                return i, board[i].index(block)
            else:
                continue




for j in range(len(states)):
    row, column = find_block(states[j],"*")
    node = Node(states[j],row,column,None,None)
    res = a_star_graph(node, "m")
    print(j+1, ",", res[0], ",", res[1], ",", res[2])


for j in range(len(states)):
    row, column = find_block(states[j],"*")
    node = Node(states[j],row,column,None,None)
    res = a_star(node, "m")
    print(j+1, ",", res[0], ",", res[1], ",", res[2])


for i in range(10):
    for j in range(len(states)):
        row, column = find_block(states[j],"*")
        node = Node(states[j],row,column,None,None)
        res = dfs(node)
        print(j+1, ",", res[0], ",", res[1], ",", res[2])


for i in range(10):
    for j in range(len(states)):
        row, column = find_block(states[j],"*")
        node = Node(states[j],row,column,None,None)
        res = dfs_graph(node)
        print(j+1, ",", res[0], ",", res[1], ",", res[2])


for i in range(10):
    for j in range(len(states)):
        row, column = find_block(states[j],"*")
        node = Node(states[j],row,column,None,None)
        res = bfs_graph(node)
```

```
            print(j+1, ",", res[0], ",", res[1], ",", res[2])


    for i in range(10):
        for j in range(len(states)):
            row, column = find_block(states[j],"*")
            node = Node(states[j],row,column,None,None)
            res = iterative_deepening(node,j+1)
            print(j+1, ",", res[0], ",", res[1], ",", res[2])


    for i in range(10):
        for j in range(len(states)):
            row, column = find_block(states[j],"*")
            node = Node(states[j],row,column,None,None)
            res = bfs(node)
            if res is None:
                print(j+1, ",", None, ",", None, ",", None)
            else:
                print(j+1, ",", res[0], ",", res[1], ",", res[2])
```