# 1.    **Approach**

## 1.1, Common components

**Language**

The 4 algorithms were developed in Python.

**Data structure**

The state in this problem was built in python dictionary with the names of blocks and agent as keys and the corresponding coordinates as the value. For example, the initial state is: {"A":[4,1],"B":[4,2],"C":[4,3],"Agent":[4,4]}

The coordinates represent the number of rows and columns, like block A is placed in row 4 column 1 in that state.

The node was also built in python dictionary, with a series of keys include ’state’, ’path\_cost’, ’ depth’,’ parent\_node’, and the corresponding values, for example the root node is: {"state":initial\_state,"path\_cost":0,"depth":0,"parent\_node":None}

**Function**

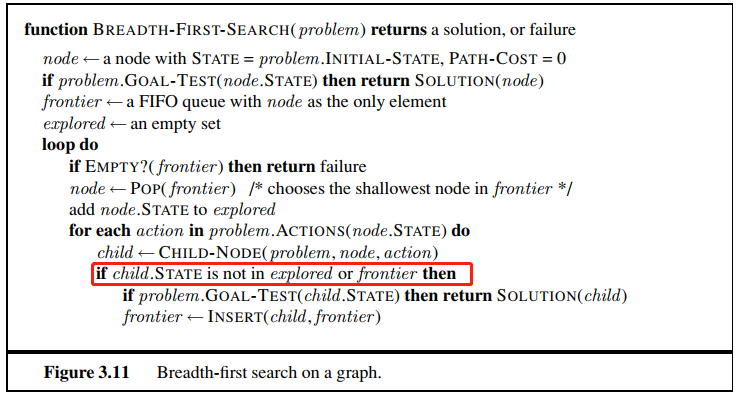
Goal test function: By comparing the positions/coordinates of the 3 blocks in current and goal state.

Action function: Consists of the movements in 4 directions. The movements were restricted within the size of grid, and when the agent meet the block, they exchange their position. The parameter of size was included in the function definition for the use of scalability.

## 1.2, Algorithms

**BFS**

The BFS was implemented according to the pseudo code(snapshot below) on page82 of the textbook *Artificial Intelligence A Modern Approach* Third Edition*,* but without the line in the red box, because it avoids exploring repeat states that belongs to the graph version:



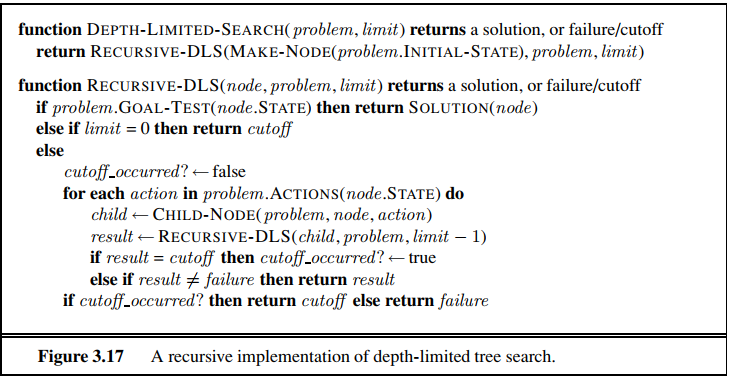
The frontier FIFO queue was implemented by building a list and poping the node with the smallest depth in it.

**DFS**

The implementation of DFS was the same as BFS except the frontier list. DFS pops the node with the largest depth in every iteration instead.

**IDS**

I implemented depth limited search first, then I iterate different depth limits to implement IDS. I followed the pseudo code on P88 *A recursive implementation of depth-limited tree search*. in the book *Artificial Intelligence A Modern Approach* Third Edition. Additionally, I added an list to record the explored nodes.



**A\***

In A\*, I built the heuristic function h(n) by using Manhattan distance. It is the sum of the Manhattan distances of every block to the final position in the goal state. It’s no greater than the steps taken by the agent to move every block to the final position e.g. it’s admissible.

The difference between the implementation of A\* and BFS was the order of poping the node in frontier list. My code pops the node with the smallest value of the sum of the path cost g(n)+the heuristic h(n) first.

# 2.    **Evidence**

I changed the initial state to make it closer to the goal state, otherwise the algorithms took too much time to get the solution in the tree search version. The initial state I took was: {"A":[2,1],"B":[3,2],"C":[4,3],"Agent":[1,1]}

## **2.1, BFS**



The solution found by BFS took 7 steps to reach the goal state. Since the step costs in this problem are all identical, it should be the optimal solution.

## **2.2, DFS**

Solution not found (took too much time and I interrupted the program)

## **2.3, IDS**

The solution found by IDS was exactly the same as BFS. Since the step costs in this problem are all identical, the solution found by IDS was also the optimal solution.

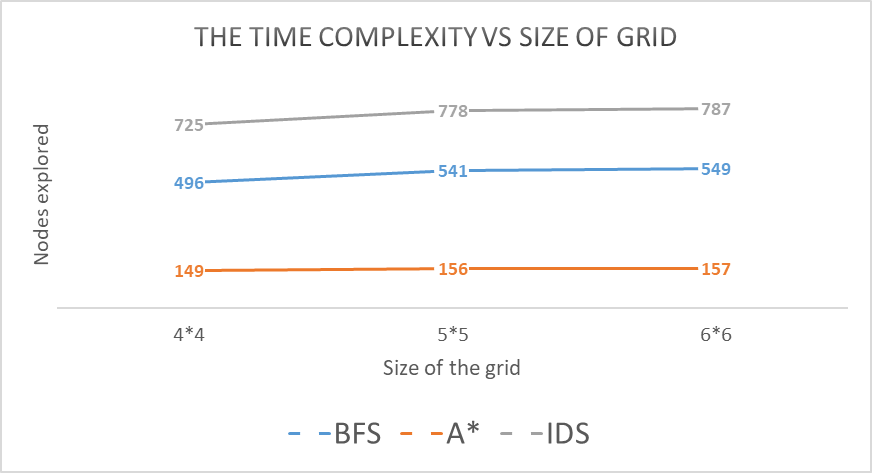
## **2.4, A∗**

The solution found by A\* was also the same as BFS. Since my heuristic function is admissible. So, it could get the optimal solution.

# 3.    **Scalability**

I controlled the problem difficulty by changing the size of the grid, I chose 3 sizes : 4\*4, 5\*5, 6\*6 to compare the time complexity.

The graph showed the relationship (No solution found in DFS):

****

Obviously, the uninformed tree search method (BFS, IDS) took much time to find the solution than the informed one(A\*).

# 4.    **Extras and limitations**

## 4.1, Extras:

### Graph search

1, I found the solution by graph search version of **BFS** with original initial state:



2, I found the solution by graph search version of **DFS** with original initial state, but with the depth of 12841 and explored nodes of 13566.

### Heuristic function

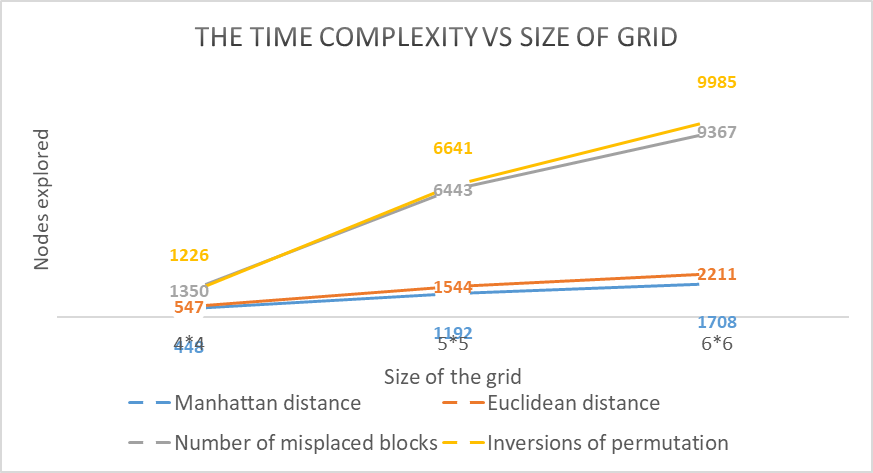
I tried another 3 kinds of heuristic functions in graph version A\* algorithms beyond Manhattan distance:

1, Euclidean distance: By calculating the Euclidean distance of the coordinates of each block between current state and goal state and sum them up.

2, Number of misplaced blocks: By counting the number of blocks with different positions compare with the goal state.

3, Inversions of permutation of the blocks' row coordinate: The row coordinates of blocks in the goal state should be in ascending order i.e. {"A":[2,2],"B":[3,2],"C":[4,2]}. By counting the number of inversions of blocks’ row coordinates in the current states is a way to quantify the distance to the goal state.

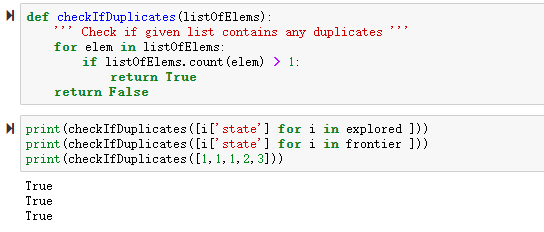
The relationship between time complexity and size of grid in different heuristic functions is like following (with original start state{"A":[4,1],"B":[4,2],"C":[4,3],"Agent":[4,4]}):



According to the result, the Manhattan distance is the best heuristic function in the graph version of A\*. I think it’s because the Manhattan distance is the most accuracy description of the movements in this problem compare to other heuristic functions.

## 4.2, Limitations:

1, I couldn’t find the reason why there were duplicate states in both explored and frontier list (graph version of BFS) when the optimal solution was found. The following code found the problem:



# 5.    **References**

1, P82 *Breadth-first search on a graph* Pseudo code in the book *Artificial Intelligence A Modern Approach Third Edition*

2, P88 *A recursive implementation of depth-limited tree search.* pseudo code in the book *Artificial Intelligence A Modern Approach Third Edition*

3, *Heuristic function* http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf

# 6.    **Code**

