IA Project Report
N-Sliding Puzzle

Project done by:
NourElhouda KLICH

Summary:

- User Manual and example uses
- Program design and implementation choices
- Exemples on instances
- Algorithms Comparison

• User Manual and example uses

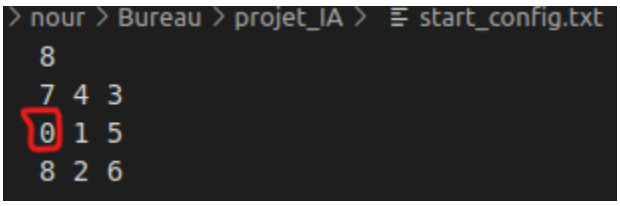
To compute the program, you should tap on the terminal:

'python3 Nsliding_puzzle.py'

Then a message will be displayed on the output screen asking you to choose the way of generating the start configuration of the problem.

```
nour@nour-virtual-machine:~/Bureau/projet_IA$ python3 Nsliding_puzzle.py
Enter the way to generate the start_configuration: 'text_file' or 'random'
```

You either write 'random' and then a random initial state will be generated or 'text_file' then the n and the initial state will be read from the file 'start_config.txt' which has this format:



n on the first line and then the **initial state**

0 represents the empty space in the puzzle!

(To test other problems you can change the content of the text file)

After choosing the start configuration choice, a message will be displayed asking you to choose which algorithm to use from the ones implemented.

```
nour@nour-virtual-machine:~/Bureau/projet_IA$ python3 Nsliding_puzzle.py
Enter the way to generate the start_configuration: 'text_file' or 'random'
text_file
************************************************
Start configuration read from the file 'text_file':
n= 8
Initial State of the game:
| 7 | 4 | 3 |
| 0 | 1 | 5 |
| 8 | 2 | 6 |
With which search algorithm you want to solve the problem?
Write 'BFS' or 'UCS' or 'A*_h1' or 'A*_h2' or 'Bi_BFS'
```

After that, the algorithm will start executing and at each iteration the size of frontier and the explored set are displayed and when it finishes, we will get the solution path with every move made.
If we choose BFS for this exemple, we get:

```
Solution with Breadth First Search

Current_state:
| 7 | 4 | 3 |
| 0 | 1 | 5 |
| 8 | 2 | 6 |

Moved Up >>>>>
Current_state:
| 0 | 4 | 3 |
| 7 | 1 | 5 |
| 8 | 2 | 6 |

Moved Right >>>>>
Current_state:
| 4 | 0 | 3 |
| 7 | 1 | 5 |
| 8 | 2 | 6 |

Moved Down  >>>>>
Current_state:
| 4 | 1 | 3 |
| 7 | 0 | 5 |
| 8 | 2 | 6 |

Moved Down  >>>>>
Current_state:
| 4 | 1 | 3 |
| 7 | 2 | 5 |
| 8 | 0 | 6 |

Moved Left >>>>
Current_state:
| 4 | 1 | 3 |
| 7 | 2 | 5 |
| 0 | 8 | 6 |
```

(1)

```
Moved Up >>>>>
Current_state:
| 4 | 1 | 3 |
| 0 | 2 | 5 |
| 7 | 8 | 6 |

Moved Up >>>>>
Current_state:
| 0 | 1 | 3 |
| 4 | 2 | 5 |
| 7 | 8 | 6 |

Moved Right >>>>>
Current_state:
| 1 | 0 | 3 |
| 4 | 2 | 5 |
| 7 | 8 | 6 |

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 |
| 4 | 0 | 5 |
| 7 | 8 | 6 |

Moved Right >>>>>
Current_state:
| 1 | 2 | 3 |
| 4 | 5 | 0 |
| 7 | 8 | 6 |

Moved Down  >>>>>
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |
SUCCESS! Goal_state Found!
```

(2)

• Program design and implementation choices

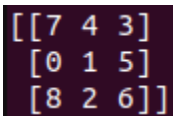
The project is structured as such: 3 files (+ the file comparison.py)

➢ **Node.py:**

It contains the class representing the node structure consisting of:
node.state: representing the state of the puzzle
node.path_cose: g, the sum of the costs of the individual step costs along the path to this node
node.total_cost: f = path_cost + heuristic
node.parent: allows to retrieve the associated path to the node.
It also contains auxiliary functions that serve to manipulate the Node structure and the class PriorityQueue that serves to store the nodes in the frontier.

➢ **Nslidingpuzzle.py:**

It is the main file representing the game, it has the main function and the class Game that represents the game and it contains :
The constructor that initializes the list of tiles, the goal_state and initial_state of the game.
Implementation choices:
STATE: is represented by an np.array (a list of lists), since it is efficient to access every element and to make moves by replacing the placement of tiles.
N.B: we add 0 to the tiles, that represent the empty space.



And the class contains the functions:
goal_test(self,state): Checks if the state in the parameter is the same as the goal state. Returns True if it is the case and False otherwise.

Actions(self,state): it generates the list of actions that can be executed in state

Its pseudocode:

actions ← empty list
empty_row, empty_col ← get the position of 0 which represents the empty tile in the state
Find each possible move from state (Up, Down, right or left)
Add each possible move to actions
return actions

Results(self,node): expand the given node by applying state transitions(possible actions) and returns the list of successors (nodes) of the given node

Its pseudocode:

```
results ← empty list
empty_row, empty_col ← get the position of 0 which represents the empty tile in the state
actions ← Actions(node.state)
for action in actions:
    temp_state ← deepcopy(node.state)
    apply action to temp_state to get the new child state
    add Node(temp_state) to results
return results
```

random_init_state(self,difficulty): Generate the initial state randomly; it takes a random series of moves backwards from the goal state so that the initial state of the puzzle is guaranteed to be solvable, and difficulty is the number of moves to make from the goal state. (the n is 8 by default because it is not too easy and not too difficult).

The file also contains other auxiliary functions that serve to manipulate the game (read_text_file,random_game).

➢ **Algorithms.py:**

It contains the search algorithms and heuristic functions.

Implementation choices:

The algorithms use the graph search version. I preferred to use this version because it avoids exploring redundant paths and so it's more efficient in terms of time and space complexity. And since the two heuristics we use are consistent we can use graph search version.

Breadth First Search (BFS):

The data structure used to represent the frontier is a FIFO queue(with python library Queue),because we need to get the first node added to the frontier and this data structure makes it easy .

The explored states are stored in a simple list for all algorithms, to make it easy verifying if a state is already explored or not.

Uniform Cost Search(UCS), a_star: For UCS and a_star I use the same data structure for the frontier which is a PriorityQueue (class in the file Node.py) ordered by the total cost (function f).

- The total cost for UCS is just the path_cost of the node (g)
- The total cost for A* is the node path_cost + heuristic (f)

Since in each iteration we need to get the lowest_cost node, this data structure makes it easy since it stores the lowes_cost node in the beginning of the PriorityQueue.

a_star(chosen_h): It takes the chosen heuristic in parameters, and makes a call to the function heuristic(chosen_h) to calculate it; if chosen_h is' h1', then it calculates the number of displaced tiles, otherwise it is the sum of Manhattan distances.

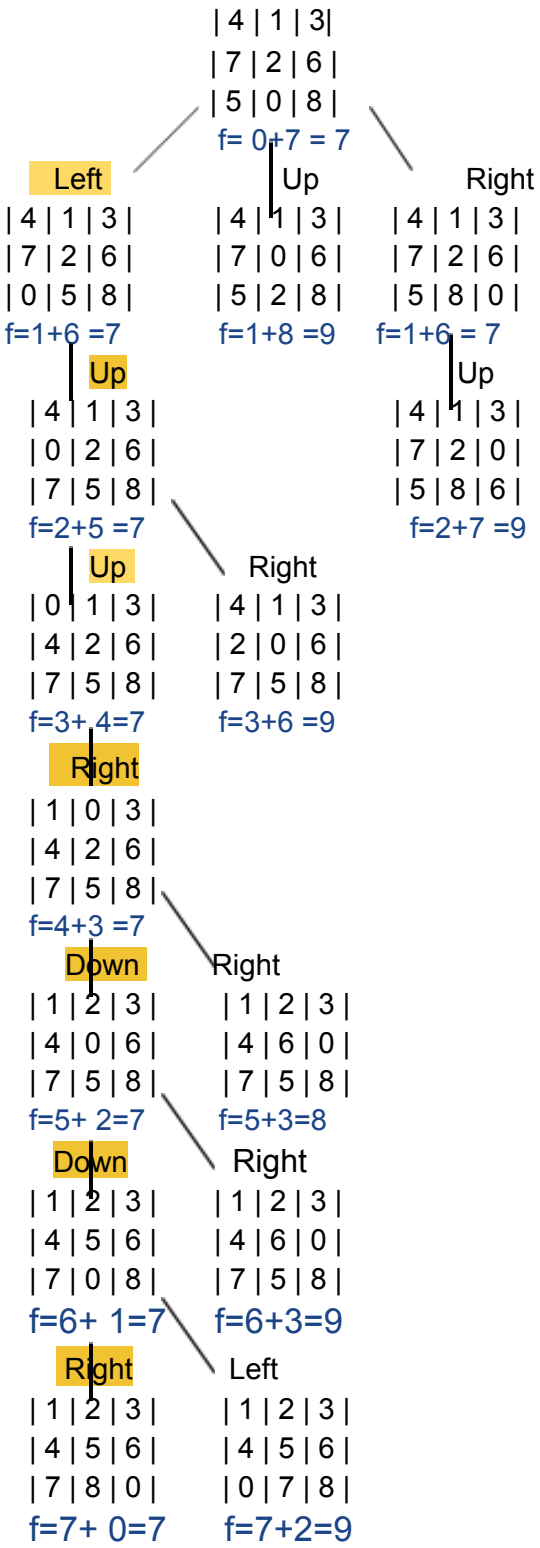
Bidirectional_BFS: to verify if we have reached the same value from the start and the end, we use an auxiliary function lists_have_same_value(list1,list2).

For all algorithms, when the goal state is found, the path of the solution is displayed using a function print_path that uses the structure node.parent to retrieve the path to each node.

• Examples on instances

We run an instance with the A*_h2 (Manhattan heuristic):

Initial state, n=8




```
size of the frontier: 7
size of the explered set: 8
****************************
Solution with Algo A* h2 :

Current_state:
| 4 | 1 | 3 |
| 7 | 2 | 6 |
| 5 | 0 | 8 |
path_cost(g)=  0
total_cost(f=g+h)=  7

Moved Left >>>>
Current_state:
| 4 | 1 | 3 |
| 7 | 2 | 6 |
| 0 | 5 | 8 |
path_cost(g)=  1
total_cost(f=g+h)=  7

Moved Up >>>>>
Current_state:
| 4 | 1 | 3 |
| 0 | 2 | 6 |
| 7 | 5 | 8 |
path_cost(g)=  2
total_cost(f=g+h)=  7

Moved Up >>>>>
Current_state:
| 0 | 1 | 3 |
| 4 | 2 | 6 |
| 7 | 5 | 8 |
path_cost(g)=  3
total_cost(f=g+h)=  7

Moved Right >>>>>
Current_state:
| 1 | 0 | 3 |
| 4 | 2 | 6 |
| 7 | 5 | 8 |
path_cost(g)=  4
total_cost(f=g+h)=  7

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 |
| 4 | 0 | 6 |
| 7 | 5 | 8 |
path_cost(g)=  5
total_cost(f=g+h)=  7

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 0 | 8 |
path_cost(g)=  6
total_cost(f=g+h)=  7

Moved Right >>>>>
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |
SUCCESS! Goal_state Found!
```

-we run an instance with Bidirectional_BFS , | 1 | 3 | 0 | 4 |
with n=15 and the initial state: | 6 | 2 | 7 | 8 |
| 5 | 14| 10 | 11|
| 9 | 13| 15 | 12|

```
Solution with Algo Bidirectional_BFS* :

-----------------Intersection state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 14 | 10 | 11 |
| 0 | 13 | 15 | 12 |
-----------------Path from start:
Current_state:
| 1 | 3 | 0 | 4 |
| 6 | 2 | 7 | 8 |
| 5 | 14 | 10 | 11 |
| 9 | 13 | 15 | 12 |

Moved Left >>>>
Current_state:
| 1 | 0 | 3 | 4 |
| 6 | 2 | 7 | 8 |
| 5 | 14 | 10 | 11 |
| 9 | 13 | 15 | 12 |

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 6 | 0 | 7 | 8 |
| 5 | 14 | 10 | 11 |
| 9 | 13 | 15 | 12 |

Moved Left >>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 0 | 6 | 7 | 8 |
| 5 | 14 | 10 | 11 |
| 9 | 13 | 15 | 12 |

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 0 | 14 | 10 | 11 |
| 9 | 13 | 15 | 12 |

Moved Down  >>>>>
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 14 | 10 | 11 |
| 0 | 13 | 15 | 12 |
```

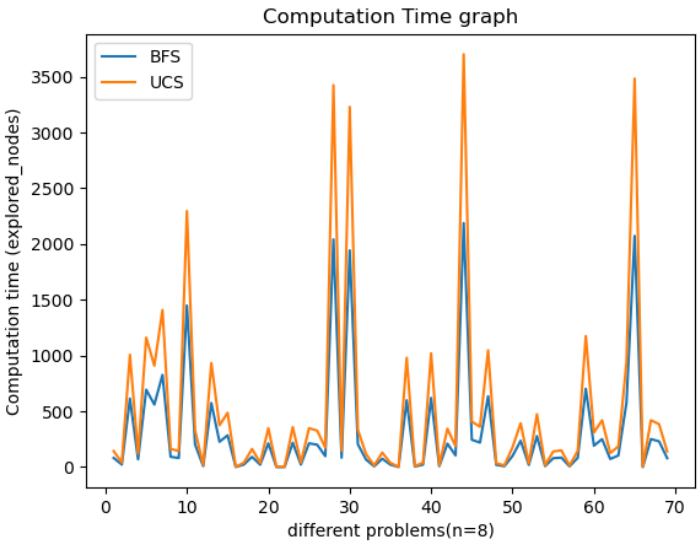
```
-----------------Path from end:
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 0 |

Moved Up >>>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 0 |
| 13 | 14 | 15 | 12 |

Moved Left >>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 0 | 11 |
| 13 | 14 | 15 | 12 |

Moved Left >>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 0 | 10 | 11 |
| 13 | 14 | 15 | 12 |

Moved Down  >>>>>
Current_state:
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 14 | 10 | 11 |
| 13 | 0 | 15 | 12 |

Moved Left >>>>
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 14 | 10 | 11 |
| 0 | 13 | 15 | 12 |
```

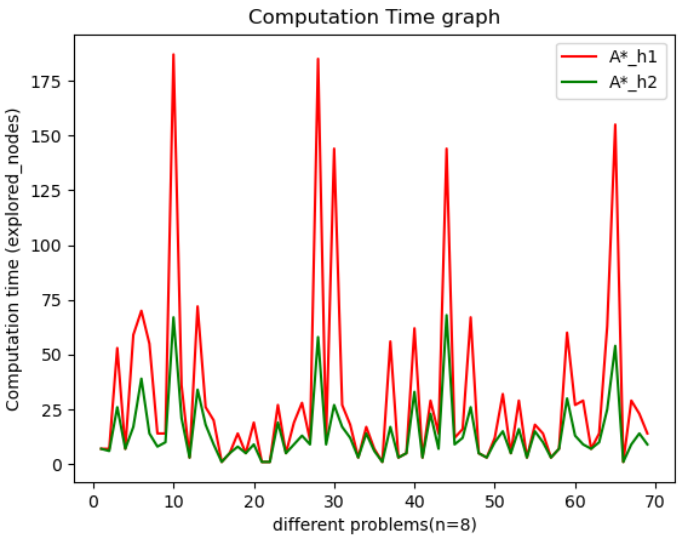
• Algorithms comparison

➢ Time complexity comparison:

Different problems for the same n=8 :

We take randomly 70 generated different problems with n=8 and we calculate the computation time(in terms of numbers of explored nodes) for each algorithm. We get these graphs:





As we can see on the graphs, for different problems, the computation time for each algorithms is ordered as follows:
A*_h2 (Manhattan) < A*_h1 (displaced tiles) << BFS <UCS

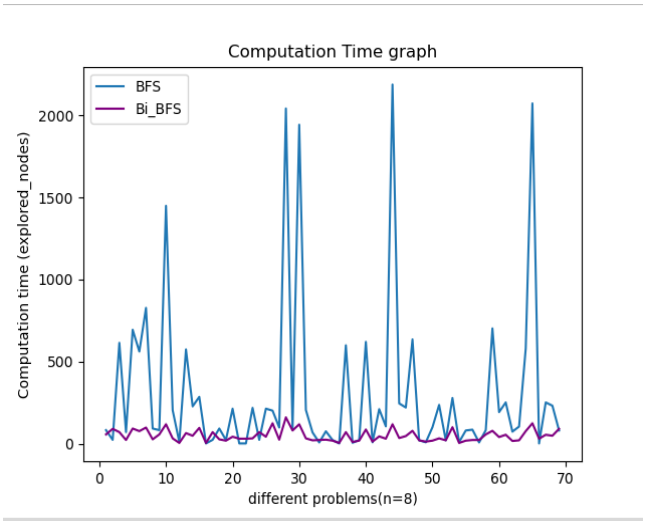
As we can see on the first graph, for uninformed search, BFS is more efficient than UCS here in terms of time complexity but not with an important difference; which makes sense since that for the n-sliding puzzle problem, all actions costs have the same value so UCS and BFS are almost the same ,but the only difference is the termination condition since the BFS returns the state as soon as it

finds it, however UCS verifies the path cost before returning the state. Thus BFS is more efficient for the problem.
As we see on the second graph, A* (for both heuristics) is way faster(in terms of time) than uniformed cost searches(the maximum number of explored nodes with A* is 175, however with UCS it's 3500), which makes sense since UCS has no information about the goal location and so it explores options in every 'direction',but with A*, when using the heuristic ,the total cost (f(n)) becomes the estimated cost of the cheapest path to a goal state through a node n and therefore A* has information about the goal location which makes it better .

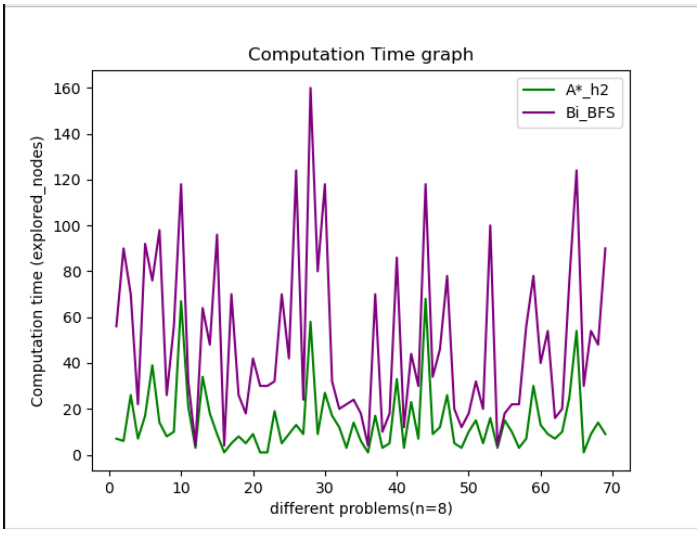
And A* with Manhattan heuristic h2 is more efficient than with the number of displaced tiles heuristic h1 (the maximum time (number of explored nodes) using h2 is 70 while using h1 it is 175) .
This makes sense since h1 only takes into account whether a tile is misplaced or not, but it does not take into account how far away that tile is far from being correct, in contrast h2 does take this information into account.
So we can conclude that A*_h2 is the best search algorithm in terms of time complexity that we can use for the n sliding problem.

- ***Other algorithm***:

Since uninformed search algorithms took a lot of time, i tried to improve it by implementing the bidirectional search using BFS, and as we can see on the graph, there is an important difference between the two algorithms computation time (maximum of 2000 for BFS and 160 for Bi_BFS).

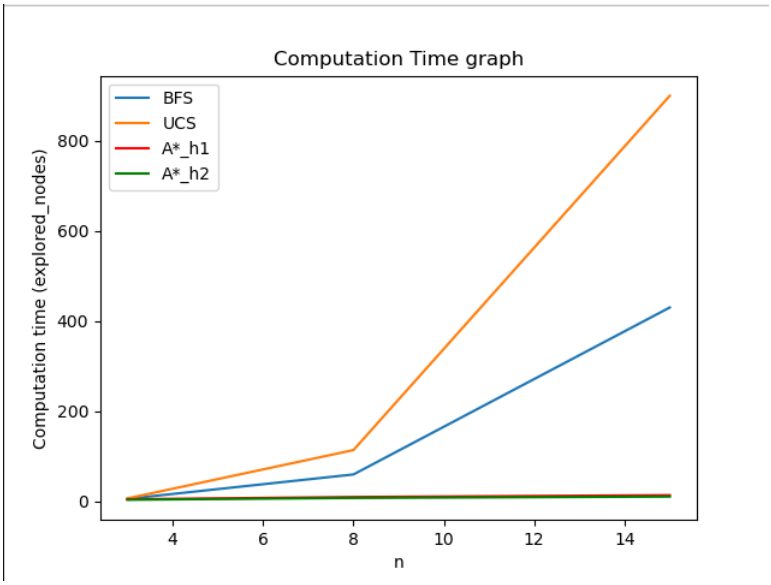


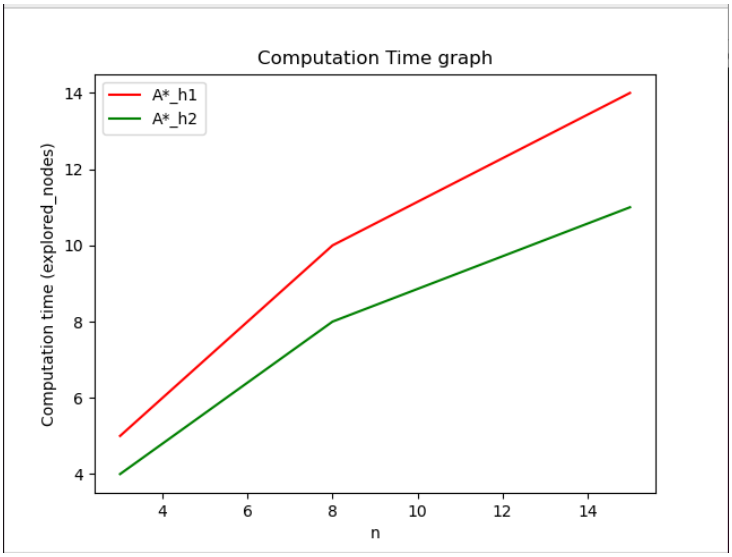
A*_h2 is certainly still the search algorithm that has minimum computation time , as we can see on this graph, but bidirectional BFS gets a little close to it compared to other uninformed searches.



Problems with different n:

We take one randomly generated problem for each n in {3,8,15} and we calculate the computation time for each algorithm:





As we can see on the graph, the larger n gets, the more time the search algorithms spend to find the solution.

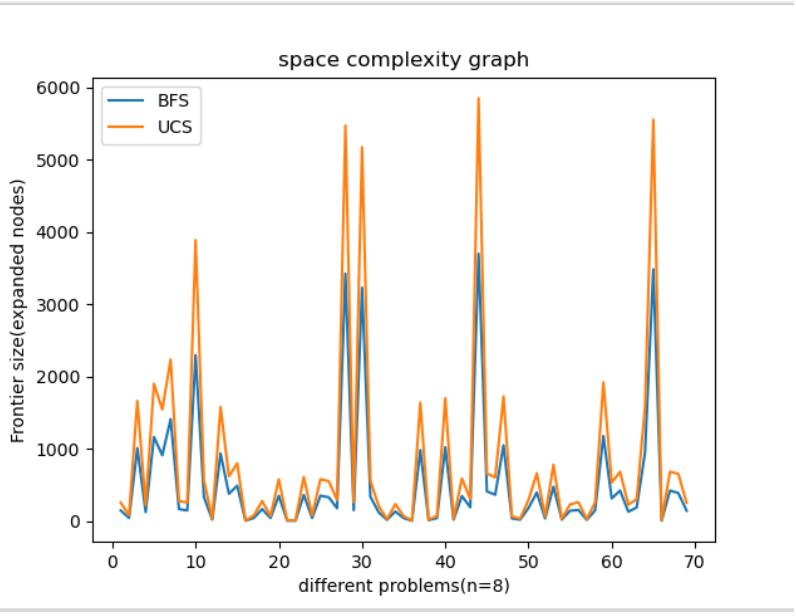
But we can see that BFS and UCS are efficient for small size problems but take too much time for the largest ones.And clearly, A*_h2 is the fastest one for large size problems (11 explored

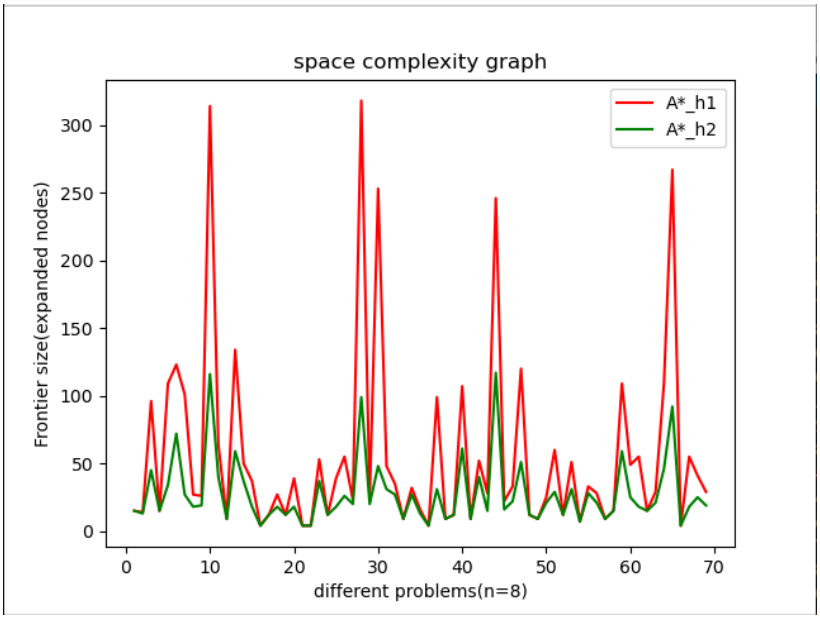
nodes for the n=15 while UCS explores more than 800 for the same problem).
If we take time limit= 15 seconds and a ‘difficult’ board is passed in, then BFS and obviously UCS will not be able to solve problems for n greater than 15, and even A* will timeout within 15 seconds due to using all memory if the problem is too complicated.
(A better solution would be bidirectional_A* algorithm).

➢ **Space complexity comparison:**

Different problems for the same n=8 :

We take the same previously randomly generated different problems with n=8 and we calculate the number of expanded nodes(frontier size) during each algorithm to compare space complexity. We get these graphs:





As we can see in the graphs,the space complexity is ordered in the following way:
A*_h2 (Manhattan) < A*_h1 (displaced tiles) << BFS <UCS

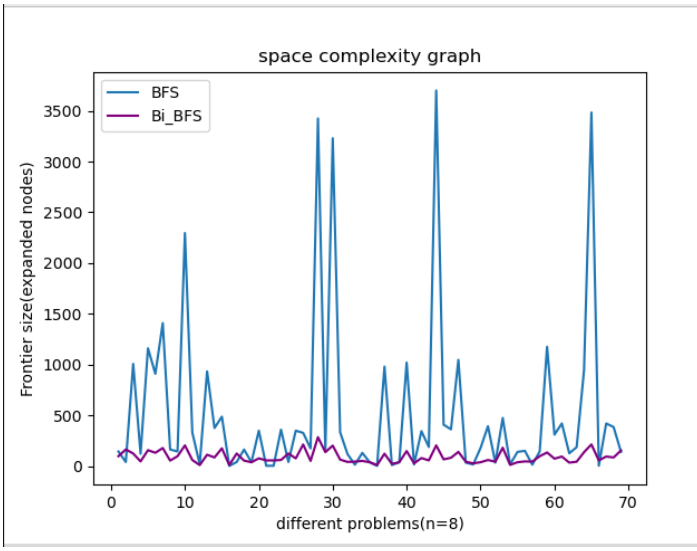
For uninformed search, BFS is more efficient than UCS(maximum expanded nodes by UCS is 6000, while with BFS it is 4000) which

makes sense since BFS expands less nodes but it still has not a good performance in terms of memory use.

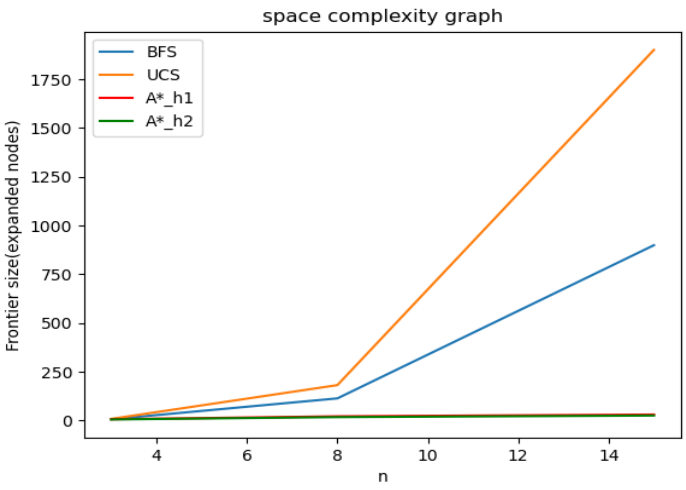
And for time complexity, A* with both heuristics is way more efficient in terms of space complexity than uninformed search algorithms since it expands less nodes : 300 expanded nodes at maximum with A* ,but 4000 with BFS.
And A* using Manhattan heuristic h2 is more efficient in terms of space complexity than with the number of displaced tiles heuristics h1 (the maximum number of expanded nodes using h2 is 100 while using h1 it is 300).Which makes sense, for the same reason as explained in time complexity comparison.
Thus, A*_h2 is the best search algorithm in terms of space complexity as well.

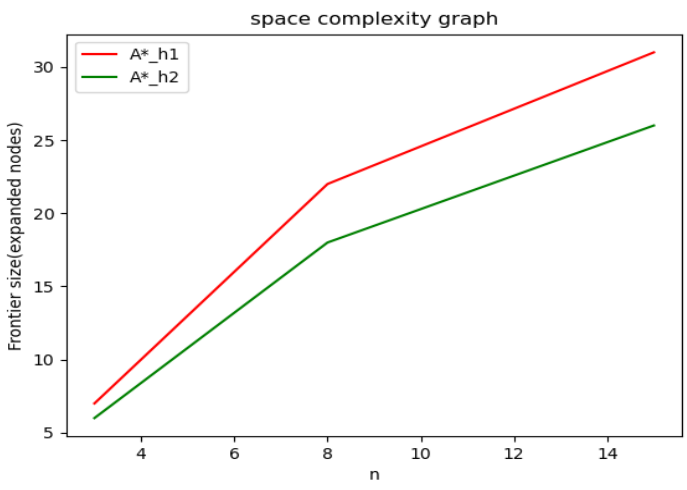
- ***Other algorithm:***

Like for time complexity, uninformed search algorithms take a lot of space in the memory as well. To improve it, I implemented bidirectional search using BFS, and as we can see on the graph, there is an important difference between the two algorithms' space complexity (maximum of 3500 for BFS and 300 for Bi_BFS).



<u>Problems with different n:</u>
For the same randomly generated problems in the computation time comparison, for each n in {3,8,15} we calculate the number of expanded nodes(frontier size) during each algorithm to compare space complexity and we get these graphs:





As we can see on the graph, the larger n gets, the more space the search algorithms use to find the solution.
But we can see that BFS and UCS are efficient for small size problems but they use more space for the largest ones.
And, A*_h2 is the most efficient one in terms of space complexity (25 expanded nodes for the n=15 while UCS explores more than 1750 for the same problem).

But even that A*_h2 Manhattan distance heuristic search manages to find a solution with moderate memory compared to other search algorithms, it still usually runs out of space long before it runs out of time and it is not practical for many large-scale problems (for n>=24 for example).