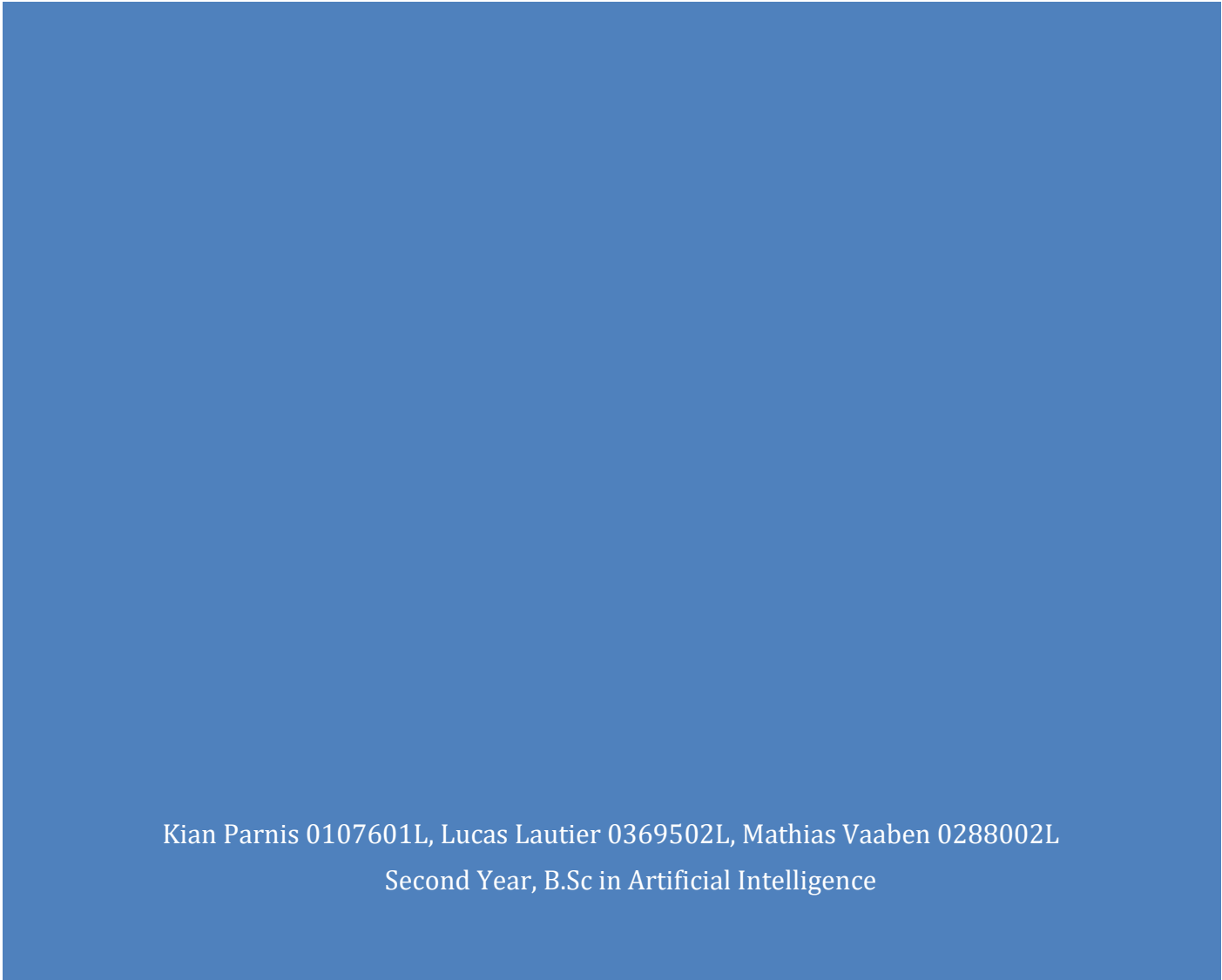




# FUNDAMENTALS OF AUTOMATED PLANNING REPORT



Kian Parnis 0107601L, Lucas Lautier 0369502L, Mathias Vaaben 0288002L  
Second Year, B.Sc in Artificial Intelligence

## Contents

Automated Planning Assignment .....	2
Deliverables.....	2
Running the code .....	2
Implementation of the Breadth First Search Solver .....	3
Breadth First Search sample tests.....	4
Implementation of the A* solver .....	5
A* Limitations .....	6
A* Misplaced tiles sample tests: .....	6
A* Manhattan Distance sample tests: .....	7
Greedy Best First Search .....	8
Testing Scenarios.....	9
Manhattan Heuristic.....	9
Misplaced Tile Heuristic.....	10
Enforced Hill Climbing.....	11
Testing Scenarios.....	11
Manhattan Heuristic.....	11
Misplaced Tile Heuristic:.....	13
Implementation of PDDL .....	14
Comparison of Results.....	17
Part 1 and Part 2 .....	17
Evaluation .....	17
Plagiarism Form.....	18
Bibliography .....	19
Distribution of Work.....	20

# Automated Planning Assignment

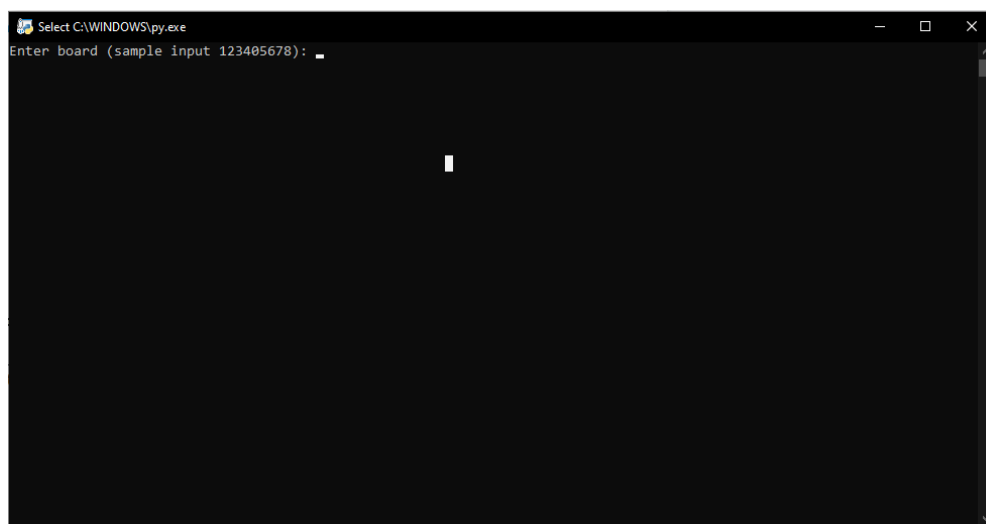
## Deliverables

### **ARI2102 - Assignment Submission**

- Search Algorithms
  - Astar Search
  - Breadth first search
  - Enforced Hill Climbing Search
  - Greedy Best First Search
- PDDL
  - domain.pddl
  - problem1.pddl
  - problem2.pddl
  - problem3.pddl
  - problem4.pddl
  - problem5-hard.pddl
  - problem6-hard.pddl

## Running the code

The PDDL files can be loaded onto the “editor.planning.domain” website and run from there. The four algorithms have been programmed using python 3.10 and are designed in a way where the user can simply load them via the python terminal and run them respectively.



# Implementation of the Breadth First Search Solver

This puzzle solver was implemented by first taking a textual input by the user for the list of numbers that will be solved by the algorithm *such as* 123405678 which would translate to

1 2 3

4 0 5. This config is passed to a function '**inversions**' [1] which counts the number of

6 7 8

inversions that happen for the configuration, if the function returns odd the puzzle is deemed unsolvable and returns 'Not Valid' if it is even then the function processed. The package 'time' is used to keep track of the time taken by the algorithm to start and end and the configuration is passed to the '**calculate\_bfs**' function. A class **Board** is used to maintain all the states via `__init__` and the initial root node is set to the current config and every time it's called a counter for the number of total states is incremented.

A simple function '**goalfound**' is used to determine if any state passed is the goal state, this is done twice, initially to check whether the initial state is the goal state and during expansion. A queue is maintained to keep track of the fringe states which is utilized by the algorithm [2] and this is implemented using the package queue. A list of explored states is maintained to avoid the risk of traversing what has already been traversed by the algorithm to avoid any potential cyclic occurrence. '**generate\_child**' is called which creates the new child nodes for all the possible actions the state can move to, and these possible actions are retrieved via the '**GetMoves**' function.

The breadth first search will now take place which will traverse all the new child nodes for the new level (checking if these new children have been previously explored or not) and will also check if there are any child states which are the goal states, and the children are put in the queue [3]. This will continue to expand level by level till a goal state is found and if so the '**getSolution**' function is called.

This function will regressively append each state to a solution list by starting at the goal node and traversing back till it can no longer find a parent to be traversed. This list is reversed and returned to the main method. The end time is taken and for the number of actions the length of the returned list is calculated. For a better visualisation for the user all the states are converted to a 3x3 matrix with the use of the numpy package and the solution is printed as well as the time taken, the number of actions and all the states generated.

## Breadth First Search sample tests

Input: 123405678

Input: 102345678

Input: 812043765

```
[6 7 8]]
[1 2 3]
[4 5 0]
[6 7 8]]

[1 2 3]
[4 5 8]
[6 7 0]]

[1 2 3]
[4 5 8]
[6 0 7]]

[1 2 3]
[4 5 8]
[0 6 7]]

[1 2 3]
[0 5 8]
[4 6 7]]

[1 2 3]
[5 0 8]
[4 6 7]]

[1 2 3]
[5 6 8]
[4 0 7]]

[1 2 3]
[5 6 8]
[4 7 0]]

[1 2 3]
[5 6 0]
[4 7 8]]

[1 2 3]
[5 0 6]
[4 7 8]]

[1 2 3]
[0 5 6]
[4 7 8]]

[1 2 3]
[4 5 6]
[0 7 8]]

[1 2 3]
[4 5 6]
[7 0 8]]

[1 2 3]
[4 5 6]
[7 8 0]]

Valid
Runtime of algorithm is 0.5589947700500488
Number of actions: 15
The total number of unique states generated 11606
Press enter to exit
```

```
[6 0 3]
[7 8 5]]

[1 4 2]
[0 6 3]
[7 8 5]]

[1 4 2]
[7 6 3]
[0 8 5]]

[1 4 2]
[7 6 3]
[8 0 5]]

[1 4 2]
[7 0 3]
[8 6 5]]

[1 0 2]
[7 4 3]
[8 6 5]]

[1 2 0]
[7 4 3]
[8 6 5]]

[1 2 3]
[7 4 0]
[8 6 5]]

[1 2 3]
[7 4 5]
[8 6 0]]

[1 2 3]
[7 4 5]
[8 0 6]]

[1 2 3]
[7 4 5]
[0 8 6]]

[1 2 3]
[0 4 5]
[7 8 6]]

[1 2 3]
[4 0 5]
[7 8 6]]

[1 2 3]
[4 5 0]
[7 8 6]]

[1 2 3]
[4 5 6]
[7 8 0]]

Valid
Runtime of algorithm is 178.90601921081543
Number of actions: 22
The total number of unique states generated 191155
```

```
Enter board (sample input 123405678): 812043765
Not Valid
Press enter to exit_
```

# Implementation of the A\* solver

This puzzle solver was implemented by first taking a textual input by the user for the list of numbers that will be solved by the algorithm such as 123405678 which would translate to

1 2 3

4 0 5. This config is passed to a function '**inversions**' [1] which counts the number of

6 7 8

inversions that happen for the configuration, if the function returns odd the puzzle is deemed unsolvable and returns 'Not Valid' if it's even then the user is asked to choose between two heuristics, Manhattan Distance or Misplaced Tiles. The package 'time' is used to keep track of the time taken by the algorithm from start to end and a Boolean is passed depending on the heuristic chosen as well as the initial state and the goal state to the '**calculate\_A**' function. Once the function is called the function '**getMoves**' is called which returns the possible actions each state can take as well as a list of key value pairs which contain a list of the state's current board, a list of its parent board config and its goal cost and heuristic estimate. Depending on the heuristic chosen the respective function '**manhattan**' or '**misplaced tiles**' is called [4][5]. The Manhattan Distance is calculated by returning the sum of each of the values Manhattan Distance while the misplaced tiles returns a counter which counts the number of values which are equal in positioning to the goal states values.

The initial state is set and since it's the root node, parent is set to -1 and the goal cost is set to 0. A queue is initialised which will keep all the visited states within it. The initial state is checked to see whether it is the goal state and if it isn't an infinite loop starts the A\* algorithm. For every new layer a function '**Priority\_Queue**' is called which takes the current queue and sorts it using the NumPy's package `np.sort()` function according to the lowest cost estimate [2]. The state which has the lowest cost is retrieved and is popped off the queue. Its data is retrieved, and the children of the state are generated. Each child is first checked to see whether or it has already been traversed in order to avoid any potential cycles and if they haven't been explored there cost estimate are calculated in respect to which heuristic is chosen in addition to the new level (more formally  $f_n = g_n + h_n$ ) [2] and are appended to the queue and the next iteration starts, if at any point the goal state is found the state and length of the priority queue are returned and the function '**evaulate\_path**' is called.

This function will regressively append each state to an array of paths by starting at the goal node and traversing back till it can no longer find a parent to be traversed. For a better visualisation for the user all the states are converted to a 3x3 matrix with the use of the NumPy package and the solution is printed as well as the time taken, the number of actions and all the states generated and all the states that have been traversed.

## A\* Limitations

A limitation is present when trying to solve a board using the manhattan distance. While a solution is found, the time taken to achieve this solution is not time optimally efficient, given more time optimizations could be made to better improve the running time of this heuristic search.

## A\* Misplaced tiles sample tests:

Input: 123405678

Input: 102345678

Input: 812043765

<pre>[6 7 8]] [[1 2 3] [4 5 0] [6 7 8]]  [[1 2 3] [4 5 8] [6 7 0]]  [[1 2 3] [4 5 8] [6 0 7]]  [[1 2 3] [4 5 8] [0 6 7]]  [[1 2 3] [0 5 8] [4 6 7]]  [[1 2 3] [5 0 8] [4 6 7]]  [[1 2 3] [5 6 8] [4 0 7]]  [[1 2 3] [5 6 8] [4 7 0]]  [[1 2 3] [5 6 0] [4 7 8]]  [[1 2 3] [5 0 6] [4 7 8]]  [[1 2 3] [0 5 6] [4 7 8]]  [[1 2 3] [4 5 6] [0 7 8]]  [[1 2 3] [4 5 6] [7 0 8]]  [[1 2 3] [4 5 6] [7 8 0]]]  Valid Runtime of algorithm is 0.17806386947631836 Number of actions: 14 The total number of unique states generated 554 Total number of unique states visited: 334</pre>	<pre>[6 0 3] [7 8 5]]  [[1 4 2] [0 6 3] [7 8 5]]  [[1 4 2] [7 6 3] [0 8 5]]  [[1 4 2] [7 6 3] [8 0 5]]  [[1 4 2] [7 0 3] [8 6 5]]  [[1 0 2] [7 4 3] [8 6 5]]  [[1 2 0] [7 4 3] [8 6 5]]  [[1 2 3] [7 4 0] [8 6 5]]  [[1 2 3] [7 4 5] [8 6 0]]  [[1 2 3] [7 4 5] [8 0 6]]  [[1 2 3] [7 4 5] [0 8 6]]  [[1 2 3] [0 4 5] [7 8 6]]  [[1 2 3] [4 0 5] [7 8 6]]  [[1 2 3] [4 5 0] [7 8 6]]  [[1 2 3] [4 5 6] [7 8 0]]]  Valid Runtime of algorithm is 41.000922441482544 Number of actions: 21 The total number of unique states generated 9461 Total number of unique states visited: 6037</pre>	<pre>Enter board (sample input 123405678): 812043765 Not Valid Press enter to exit_</pre>
---	---	---

## A\* Manhattan Distance sample tests:

Input: 123405678

Input: 102345678

Input: 812043765

```
[6 7 8]]
[[1 2 3]
[4 5 0]
[6 7 8]]

[[1 2 3]
[4 5 8]
[6 7 0]]

[[1 2 3]
[4 5 8]
[6 0 7]]

[[1 2 3]
[4 5 8]
[0 6 7]]

[[1 2 3]
[0 5 8]
[4 6 7]]

[[1 2 3]
[5 0 8]
[4 6 7]]

[[1 2 3]
[5 6 8]
[4 0 7]]

[[1 2 3]
[5 6 8]
[4 7 0]]

[[1 2 3]
[5 6 0]
[4 7 8]]

[[1 2 3]
[5 0 6]
[4 7 8]]

[[1 2 3]
[0 5 6]
[4 7 8]]

[[1 2 3]
[4 5 6]
[0 7 8]]

[[1 2 3]
[4 5 6]
[7 0 8]]

[[1 2 3]
[4 5 6]
[7 8 0]]]
Valid
Runtime of algorithm is 16.090759992599487
Number of actions: 14
The total number of unique states generated 6027
Total number of unique states visited: 3671

C:\WINDOWS\py.exe
[[1 4 2]
[0 6 3]
[7 8 5]]

[[1 4 2]
[7 6 3]
[0 8 5]]

[[1 4 2]
[7 6 3]
[8 0 5]]

[[1 4 2]
[7 0 3]
[8 6 5]]

[[1 0 2]
[7 4 3]
[8 6 5]]

[[1 2 0]
[7 4 3]
[8 6 5]]

[[1 2 3]
[7 4 0]
[8 6 5]]

[[1 2 3]
[7 4 5]
[8 6 0]]

[[1 2 3]
[7 4 5]
[8 0 6]]

[[1 2 3]
[7 4 5]
[0 8 6]]

[[1 2 3]
[0 4 5]
[7 8 6]]

[[1 2 3]
[4 0 5]
[7 8 6]]

[[1 2 3]
[4 5 0]
[7 8 6]]

[[1 2 3]
[4 5 6]
[7 8 0]]]
Valid
Runtime of algorithm is 5498.730169773102
Number of actions: 21
The total number of unique states generated 66221
Total number of unique states visited: 48483
Press enter to exit

Enter board (sample input 123405678): 812043765
Not Valid
Press enter to exit_
```



# Greedy Best First Search

The greedy best first search algorithm explores the node that is closest to the goal. It does this by using a heuristic function. It is called greedy as it always takes the lowest heuristic possible, without thinking of any future states. This makes this algorithm very dependent on the heuristic function, as an inaccurate heuristic will lead the GBFS into states far away from the goal.[6]

My implementation starts from a main that asks the user whether he would like the solver to use a Manhattan or Misplaced Tiles Heuristic. It takes the input and according to that calls my '**greedybfs()**' method with a different parameter, which acts as a switch between the heuristics. The start and goal state are saved as 2D lists, so that the program had access to any element.

Everything is written in a class Node, where I implemented a node structure. A method `_init_` also acts as a constructor class giving different properties that can be used throughout the program, to make code more organized and concise.

'**greedybfs()**' is where everything comes together, as all the other methods are called here to solve the problem. A max list size variable is initialized, this will be used throughout the program to make sure that the value of explored nodes does not exceed the memory available and so preventing a crash.

The heuristic functions are called first. These have a similar implementation where a nested for loop goes through all the elements, from there different functionalities are used to find the heuristic value stored in the int variable '**hVal**'.

- **manhattan()**
  - This works by having two lists which store the goal X and Y axis for every element. Inside the nested for loop, a formula was used to determine how far away a node was from both axes. ( $\text{distance} = (i - \text{goalY}) + (j - \text{goalX})$ ), where i and j are the current positions, and goalY and goalX are the goal state. Every node's distance was summed up in hVal.
- **misplacedTiles()**
  - A much simpler function, where the for-loop checks if a specific node is in its goal state. If not, hVal is incremented. Distance is not considered.

**movementOptions()** is called after the calculation of the heuristic. This first finds out where the blank(0) is by calling a method called **findBlank()**. This method goes through the grid until the blank is found and returns its position. From here, the first method will find all the options available depending on the current position and uses a library function deepcopy to place them into a list **finalPath**, with every move possible a variable int holding all nodes explored is incremented.

When this is done, and we return back to **'greedybfs()'**, it goes through the moves, checks what will make the heuristic value lower than the previous value, and stores that move in **currPath**.

The programs final output shows the movement of the blank, the total nodes visited, and the time taken. If it fails, a message saying that the program has timed out will be outputted.

### Testing Scenarios

Scenario 1: 8,6,7,2,5,4,3,0,1

Scenario 2: 6,4,7,8,5,0,3,2,1

### Manhattan Heuristic

- Testing Scenario 1:

```
Enter board (sample input 123405678): 867254301
Greedy Best First Search
1. Manhattan Heuristic
2. Misplaced Tile Heuristic
1
-----
Moves: 45
Each move represents where the blank(0) went
['UP', 'UP', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN']
Nodes visited: 5588
Time: 0.37056446075439453
Final state: [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Press enter to exit
```

- Testing Scenario 2:

```
Enter board (sample input 123405678): 647850321
Greedy Best First Search
1. Manhattan Heuristic
2. Misplaced Tile Heuristic
1
-----
Moves: 75
Each move represents where the blank(0) went
['UP', 'LEFT', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'LEFT', 'LEFT', 'DOWN', 'RIGHT', 'UP', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'DOWN']
Nodes visited: 6044
Time: 0.46558070182800293
Final state: [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Press enter to exit
```

## Misplaced Tile Heuristic

- Testing Scenario 1

```
Greedy Best First Search
1. Manhattan Heuristic
2. Misplaced Tile Heuristic
2
-----
Moves: 63
Each move represents where the blank(0) went
['RIGHT', 'UP', 'UP', 'LEFT', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'UP', 'LEFT', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'LEFT', 'LEFT', 'DOWN', 'RIGHT', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT']

Nodes visited: 985
Time: 0.03500533103942871
Max List Size=400
```

- Testing Scenario 2

```
Greedy Best First Search
1. Manhattan Heuristic
2. Misplaced Tile Heuristic
2
-----
Moves: 63
Each move represents where the blank(0) went
['DOWN', 'LEFT', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'UP', 'RIGHT', 'DOWN', 'LEFT', 'DOWN', 'RIGHT', 'UP', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'DOWN', 'LEFT', 'UP', 'RIGHT', 'UP', 'LEFT', 'DOWN', 'DOWN', 'RIGHT', 'UP', 'LEFT', 'UP', 'RIGHT', 'DOWN', 'DOWN']

Nodes visited: 878
Time: 0.030505895614624023
Max List Size=358
```

# Enforced Hill Climbing

Enforced hill climbing tries to get to the goal state without using the space complexity that A\* search gets. It computes the heuristic value of the initial state, which is then set as the largest H value. This is because in hill climbing, the next heuristic must be smaller than the current. When a state with a lower heuristic value is found, it goes down that path without expanding other states. It does not consider future states, and it also sacrifices possible alternatives. This makes this algorithm not complete and not optimal. [7]

The hill climbing program has a main method that calls a method '**solve()**', passing the initial state and heuristic chosen as parameters to solve the puzzle given.

The first thing the '**solve()**' does is use a nested for loop to go through the grid and find the position of the blank. It initializes variables which will be used with every loop like '**board**' and '**h**' which will hold the current state and its heuristic respectively, whilst also initializing variables that will only be outputted at the end, such as '**total\_states**' and '**move\_count**', also starting the timer.

From here a while loop is initialized, that only stops when the heuristic h is 0, meaning that all the numbers are at their correct position and hence the goal state is reached. Every time this while loop is re-run, it is a new move.

Inside the loop, four if-statements will determine a possible move that the blank can do, with respect to its current position. For example:

- If the blank is on the bottom row, it can't go down
- If the blank is on the left-most column, it can't go left

It then picks from one of the possible moves, in respect to the heuristic at the current state, and the heuristic is updated. When the heuristic is at 0, the loop ends, and the time is stopped.

As output, the program shows the total moves, the time, the total states generated, and the current state of the board.

## Testing Scenarios

Scenario 1: 8,6,7,2,5,4,3,0,1

Scenario 2: 6,4,7,8,5,0,3,2,1

## Manhattan Heuristic

Testing Scenario 1:

```
Move Down States: 2
Move Left States: 3
Move Down States: 4
Move Right States: 3
Move Up States: 2
Move Up States: 3
Move Left States: 2
Move Left States: 3
Move Down States: 2
Move Right States: 3
Move Right States: 4
Move Down States: 3

Done. Total moves: 239

Total states generated: 656
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Time: 0.02500438690185547 seconds
```

Testing Scenario 2:

```
Move Up States: 3
Move Left States: 2
Move Left States: 3
Move Down States: 2
Move Down States: 3
Move Right States: 2
Move Right States: 3
Move Up States: 2
Move Left States: 3
Move Left States: 4
Move Up States: 3
Move Right States: 2
Move Down States: 3
Move Right States: 4
Move Down States: 3

Done. Total moves: 109

Total states generated: 295
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Time: 0.010501384735107422 seconds
```

### Misplaced Tile Heuristic:

Testing Scenario 1:

```
Move Right States: 2
Move Right States: 3
Move Up States: 2
Move Up States: 3
Move Left States: 2
Move Down States: 3
Move Right States: 4
Move Down States: 3

Done. Total moves: 291

Total states generated: 790
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Time: 0.02800464630126953 seconds
```

Testing Scenario 2:

```
Move Up States: 2
Move Left States: 3
Move Left States: 4
Move Down States: 3
Move Right States: 2
Move Up States: 3
Move Right States: 4
Move Down States: 3

Done. Total moves: 179

Total states generated: 466
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Time: 0.016503334045410156 seconds
```

# Implementation of PDDL

Planning domain definition language, also known as PDDL in short, is according to Alfonso Emilio Gerevini, the go-to problem modelling language that is endorsed frequently by planning systems [8].

In this implementation, there are two objects, six predicates and four actions. The two objects are location and tile. In a three-by-three sliding tile puzzle, eight tile objects are required and nine location objects are needed.

Four of the six predicates created are simply used to represent the relationship between two tiles, these being '*leftof*', '*rightof*', '*bottomof*' and '*topof*'. They all take two parameters, both of

type location. Using the matrix,  $A = \begin{matrix} l1 & l2 & l3 \\ l4 & l5 & l6 \\ l7 & l8 & l9 \end{matrix}$ , as an example, these predicates would be

used in this manner "*leftof l4 l5*" since *l4* is on the left of *l5*, the rest of the predicates are all applied in the same method. The other two predicates are '*at*' and '*empty*'. The '*at*' predicate is used to say which tile is in which location. To say that the tile object *t1* is in the location *l2* can be done like this "*at t1 l2*". The empty predicate takes only one location as a parameter, and this is where the adjacent tiles will be able to slide to using the different actions.

The four actions are '*SLIDE-LEFT*', '*SLIDE-RIGHT*', '*SLIDE-UP*' and '*SLIDE-DOWN*'. They are nearly identical to each other only the relationship between the two locations passed are different. The parameters for all of them are a tile named '*?t*', the current location of the tile labelled '*?cLoc*' and the current empty location where the tile will be moved to is the location called '*?nLoc*'. The mutual predicates that must be satisfied are "*at ?t ?cLoc*" and "*empty ?nLoc*". These predicates make sure that the tile meant to be moved is in the current location and that the location it is going to be moved to is empty.

The different parameters for each action are the directional relationships between the two locations. That means that to do the action "*SLIDE-LEFT*", for example, the predicates "*(rightof ?cLoc ?nLoc) (leftof ?nLoc ?cLoc)*" along with the mutual predicates need to be satisfied. The other three predicates all have these two predicates but in their corresponding directions and locations.

The effect of these four actions are equivalent, as all four actions will result in the predicates "*(not (at ?t ?cLoc)) (at ?t ?nLoc) (empty ?cLoc) (not (empty ?nLoc))*". These will tell the algorithm that the tile moved is no longer at the old location, '*?cLoc*', but now at '*?nLoc*'. It will also say that '*?cLoc*' is now the empty location.

To solve a problem the objects required are all the locations and all the tiles. The predicates that need to be entered are then all the different relationships between the tiles, where all the tiles are at, and which location is empty. The goal then needs to be specified by saying where each tile needs to be at, and which tile must be the empty one. Figures 1, 2 and 3 below depict all the data required to represent the first puzzle while the latter figures 4 and 5

contain the results for both matrices. These assortments being

	8	6	7		6	4	7
2	2	5	4	and	8	5	0,
3	3	0	1		3	2	1

0 representing the empty location.

```
(:objects  
l1 - location  
l2 - location  
l3 - location  
l4 - location  
l5 - location  
l6 - location  
l7 - location  
l8 - location  
l9 - location  
t1 - tile  
t2 - tile  
t3 - tile  
t4 - tile  
t5 - tile  
t6 - tile  
t7 - tile  
t8 - tile  
)
```

Figure 1

```
(:goal (and  
(at t1 l1)  
(at t2 l2)  
(at t3 l3)  
(at t4 l4)  
(at t5 l5)  
(at t6 l6)  
(at t7 l7)  
(at t8 l8)  
(empty l9)  
)
```

Figure 2



```

(:init
  (leftof 11 12)
  (topof 11 14)
  (leftof 12 13)
  (topof 12 15)
  (rightof 12 11)
  (topof 13 16)
  (rightof 13 12)
  (bottomof 14 11)
  (leftof 14 15)
  (topof 14 17)
  (bottomof 15 12)
  (leftof 15 16)
  (topof 15 18)
  (rightof 15 14)
  (bottomof 16 13)
  (topof 16 19)
  (rightof 16 15)
  (bottomof 17 14)
  (leftof 17 18)
  (bottomof 18 15)
  (leftof 18 19)
  (rightof 18 17)
  (bottomof 19 16)
  (rightof 19 18)
  (at t8 11)
  (at t6 12)
  (at t7 13)
  (at t2 14)
  (at t5 15)
  (at t4 16)
  (at t3 17)
  (empty 18)
  (at t1 19)
)

```

Figure 3

```

Total time: 0.004
Nodes generated during search: 228
Nodes expanded during search: 81
Plan found with cost: 37
BFS search completed

```

Figure 4

```

Total time: 0.004
Nodes generated during search: 232
Nodes expanded during search: 82
Plan found with cost: 37
BFS search completed

```

Figure 5

# Comparison of Results

## Part 1 and Part 2

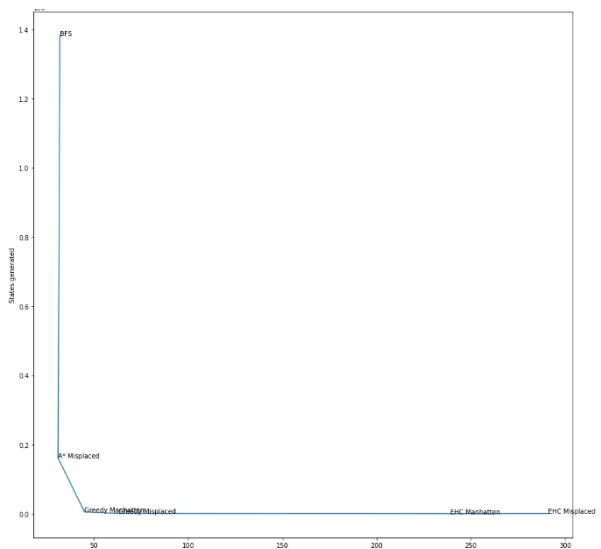
The example,  $\begin{matrix} 8 & 6 & 7 \\ 2 & 5 & 4 \\ 3 & 0 & 1 \end{matrix}$ , provided in the specification shall be used to compare the different solver and planners.

The first noticeable detail is that the PDDL, the domain-independent planner, solves consistently faster than the rest. The number of tiles generated and expanded is also noticeably lower than the other search strategies used in the domain-specific solver.

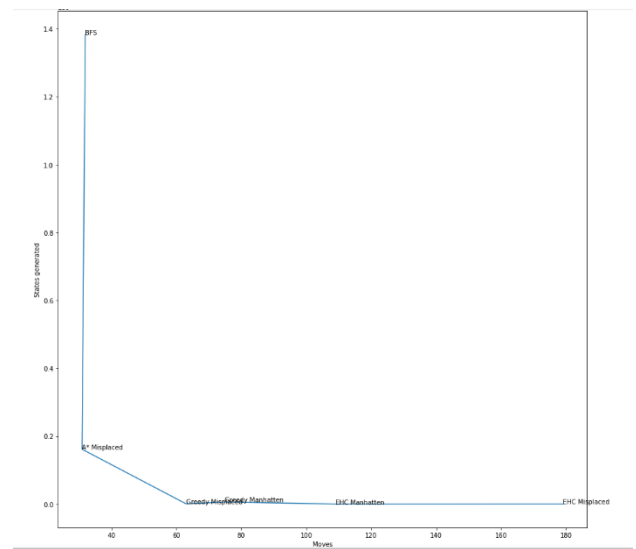
Though there is a noteworthy point about the A\* and BFS. Although they took much longer than the PDDL, both managed to find a plan within five or six less moves. The Enforced Hill Climbing and Greedy Best First Search return plans that are much longer than that returned by the PDDL but spends a significant lesser amount of time than A\* and BFS as it only takes a very small amount of time more than the PDDL.

## Evaluation

Testing Scenario 1: Results



Testing Scenario 2: Results



It is noted that Breadth First Search and A\* are the most optimally efficient when it comes to the total moves to reach the goal state, while generating the most states out of all search algorithms. On the other hand, Greedy Best First Search and Enforced Hill Climbing generated the least number of states but ended up with the most moves generated to reach the goal state. It is also noted that a heuristic implemented with the Manhattan distance will generate less moves to the goal state then one which would use the Misplaced Tiles heuristic.

# Plagiarism Form

## FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

### Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

☒ / We\*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is ~~my~~ / our\* work, except where acknowledged and referenced.

☒ / We\* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

\* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Kian Parnis  
Student Name

Kian  
Signature

Mathias Vaaben  
Student Name

Mathias  
Signature

Lucas Lautier  
Student Name

Lucas  
Signature

\_\_\_\_\_  
Student Name

\_\_\_\_\_  
Signature

AR12101  
Course Code

AR12101-Assignment Submission  
Title of work submitted

14/01/2022  
Date

# Bibliography

- [1] ceving, "How to check if a 8-puzzle is solvable?," Mathematics Stack Exchange. <https://math.stackexchange.com/questions/293527/how-to-check-if-a-8-puzzle-is-solvable> (accessed Jan. 14, 2022).
- [2] J. Bajada, "Solid-State Search." Accessed: Jan. 14, 2022. [Online]. Available: [https://www.um.edu.mt/vle/pluginfile.php/897020/mod\\_resource/content/2/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%203%20-%20Planning%20as%20State-Space%20Search.pdf](https://www.um.edu.mt/vle/pluginfile.php/897020/mod_resource/content/2/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%203%20-%20Planning%20as%20State-Space%20Search.pdf), pp. 9, 14, 15.
- [3] A. Bundy and L. Wallen, "Breadth-First Search," in Catalogue of Artificial Intelligence Tools, Springer Berlin Heidelberg, 1984, pp. 13–13. doi: 10.1007/978-3-642-96868-6.
- [4] P. E. Black, "Dictionary of Algorithms and Data Structures [online]," Accessed: Jan. 14, 2022. [Online]. Available: <https://xlinux.nist.gov/dads/HTML/manhattanDistance.html>
- [5] "Admissable Heuristic," *Wikipedia*, Jul. 19, 2021.
- [6] Lukas Beck, "A Case Study on the Search Topology of Greedy Best-First Search," 2014. Accessed: Jan. 14, 2022. [Online]. Available: <https://ai.dmi.unibas.ch/papers/theses/beck-master-14.pdf>
- [7] J. Bajada, "Planning Graph Heuristics." pp. 26–27. Accessed: Jan. 14, 2022. [Online]. Available: [https://www.um.edu.mt/vle/pluginfile.php/897034/mod\\_resource/content/3/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%207%20-%20Planning%20Graph%20Heuristics.pdf](https://www.um.edu.mt/vle/pluginfile.php/897034/mod_resource/content/3/Fundamentals%20of%20Automated%20Planning%20-%20Lecture%207%20-%20Planning%20Graph%20Heuristics.pdf)
- [8] A. E. Gerevini, "An Introduction to the Planning Domain Definition Language (PDDL): Book review," *Artificial Intelligence*, vol. 280. Elsevier B.V., Mar. 01, 2020. doi: 10.1016/j.artint.2019.103221.

# Distribution of Work

Kian Parnis (0107601L)

- Implementation of A\* and its respective documentation.
- Implementation of Breadth First Search and its respective documentation

Lucas Lautier (0369502L)

- Implementation of Greedy Best First Search and its respective documentation.
- Implementation of Enforced Hill Climbing Algorithm and its respective documentation.

Mathias Vaaben (0288002L)

- Implementation of PDDL, Domain and Problem Files its respective documentation.
- Created bibliography and comparison of results.

Signatures:

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_