



L-Università ta' Malta
Faculty of Information &
Communication Technology

Department of
Computer Information
Systems

Fundamentals of Automated Planning Project

Dylan Zerafa* (348002L), Cristina Cutajar* (230802L), Gabriel Vella* (400002L)

*B.Sc. (Hons) Artificial Intelligence

Study-unit: **Fundamentals of Automated Planning**

Code: **ARI2101**

Lecturer: **Dr Josef Bajada**

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / 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.

I / We* understand that the penalties for committing a breach of the regulations include 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.

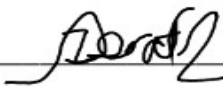
* Delete as appropriate.

(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).

Dylan Zerafa

Student Name

Signature



Cristina Cutajar

Student Name

Signature



Gabriel Vella

Student Name

Signature



Student Name

Signature

ARI2101

Course Code

Fundamentals of Automated Planning Project

Title of work submitted

11/01/2022

Date

Part 1 - Python Sliding Tile Puzzle

Implementation

For our solver, we created a node class which contains the assignment of objects, heuristics and functions to act as the building block of this program. This class initialises each node with the following values: the type of algorithm, the node's parent, the current state of the node, an empty array to later on store the node's children as well as `cost_from_start` and `cost_to_goal` for the search strategies that need a heuristic. The function **add_child** adds a child node to the list of the given node's children array as well as sets the given node as the parent of the child node. It also extends the child's cost from the start to be used by the applicable search strategies. The **get_f_val** function returns the calculated cost from the start to the goal of a particular node. This will be used by the search strategies that need a heuristic. The **traceback** function is called after the solver develops a plan. It will traverse back from the final state through all the parents until it reaches the initial state to print all the states of the plan with the count of traversals. It will then call the **validation** function. The **validation** function will loop through the list of the plan nodes and check if each move is valid by checking the indexes of the 0 in the current state and the next state and making sure that it was only moved up, down, left or right. The **print_state** function prints the given state. The **expand** function returns a list of valid expanded states from the given state. Therefore, it will try to perform the up, down, left and right moves. For each move, if it is valid, it will store the state in a temporary array and when all the moves are checked, the function will return the temporary array with the valid states. Thus, this function is used for the action of the plan where the precondition is that the index of the grid represents the tile and the effect would be the swapping of positions in a specified direction.

The Node class is very vital as it is used and called by all the separate search algorithms. One of the most important methods is the **__lt__** function. This is because f-value, which is the total estimated cost of the path through the n which represent a node, is calculated here, which is used by 3 out of the 4 search algorithms. Moreover, this is a special method that makes use of the less than operator as it compares the weights/cost of each state. However each algorithm requires a different way to calculate its f-value.

For A-star to calculate f-value we make use of $g(n)$ which represents the cost from the start node to the current node, and also take into account $h(n)$, this represents the estimated cost from n to the goal. However this cost is not known to us, hence it is a heuristic. The Greedy Best First Search calculates f-value, only using $h(n)$ and doesn't account for $g(n)$, hence why it gets the name Greedy since it doesn't take into account old knowledge ' $g(n)$ '. Furthermore, Enforced Hill Climbing much like A* search algorithm attains f-value by taking into account both the $g(n)$ and $h(n)$. Hence, this is why the algorithm is passed as one of the parameters in the Node class, so the **__lt__** function would know in which way it should calculate the f-value. As mentioned above $h(n)$ is a heuristic, something which needs to be calculated, that is why we made use of the **calc_manhattan_dist** and **calc_misplaced_tiles** functions.

The **calc_manhattan_dist** function takes the current state and goal state in its parameters and then makes use of nested for loops and two if conditions to check if the values are in a matching index as those in the goal state. If per say the value 5 has an index of index (2,2) in both the current state and goal state, then it is skipped and we continue iterating, for the value 0 we also skip. However, if we find that for a particular value the indices don't match, then we call the **find_in_sublists** to find and return the index of the passed value in the goal state. Then the 'distance' is calculated by finding the difference between the i and j index of both the states. The absolute (disregarding the sign) value returned will then be used to identify the cost of each tile transition to the goal which will help in choosing the optimal path i.e $h(n)$. This is done for all non matching values until the loop ends. Then the summed up distance is returned and is used to set the value of $h(n)$. This admissible heuristic is better suited for the eight tile puzzle problem than the euclidean distance since tiles can shift orthogonally through the 3x3 grid rather than moving in a linear line.

The **calc_misplaced_tiles** function takes the current state and goal. It is calculated by counting the number of tiles that are in the incorrect position in the grid (the empty tile is ignored). The respective lists are flattened, thus making comparison between them a lot easier. Then using the length of the goal list a loop is implemented. Then it simply checks the two lists at the same index, and if the values do not match then a variable count is incremented. This continues until the loop ends, then the value count is returned and is used to set the value of $h(n)$. This heuristic is admissible since each misplaced tile must be moved once; thus, the number of moves to place the tiles in order is at least the number of misplaced tiles.

The main function will first initialise an empty array to store the initial state of the puzzle chosen by the user as well as another array with the goal state. The **init_default_puzzle** function is then called. It will display the available puzzle options and prompt the user to choose one of them by the corresponding number. It then returns the chosen puzzle state to be stored in the `init_state` array in the `main()`. Then, the user will be prompted to choose from the available search strategies. This part has extra importance because not only is it deciding which search algorithm is being used but, as mentioned above, tells the **__it__** function in the `node` class on how to calculate the f-value. If the user chooses the Breadth First Search strategy, the **bfs** function will be called. Meanwhile, since the other search strategies require a heuristic, the **get_algorithm** function is called. This function displays the available search heuristics and returns the user's selection. Then the respective search function will be called. Once the program starts solving, time is taken in milliseconds until the program stops and solves the problem, this is done for comparison, to calculate how long each algorithm takes to solve the same problem.

Two very important functions are the **tree_search** and **bfs** function. These functions go hand in hand with the Node class. The data structure used for the **tree_search** function was a heapq since any object created in it has global access and garbage collection runs easily, thus it was efficient to store node objects in a heapq. Heapq is used to make use of append and pop operations in **tree_search**. The **bfs** function uses deque (Double Ended Queue) since it exhibits First In First Out behaviour. Deque was used as it provides quicker append and pop operations when compared to lists. Both these functions will first initialise the initial state as a node as well as initialising an empty array to later on store the explored nodes in it. The functions will initialise the heapq /deque, and append the initial states. The function will loop the following while the deque/heapq is not empty or until a solution is found. The function will pop the first element from the heapq/deque and it will check if the state corresponds to the final state. If the current state equals the goal state then the search stops and the solution is reached, hence, the traceback function is called and the functions return the number of expanded states as well as the number of maximum nodes in the deque. If the current state is not equal to the final state, the current state is appended to the heapq/deque and the expand function is called to get the possible states from the current state. Each expanded state is initialised as a node.

In the **tree_search function** a check is made to see if the new node is present in the heapq or in the explored state, if it is then the loop skips it since this means that it has already been seen. Here is also where the **calc_manhattan_dist** and **calc_misplaced_tiles** functions and the cost to goal of the new node is calculated and set. Then the **add_child** function is called to add the node as a child of the current node. The node will also be appended to the heapq. In the **bfs** function a check is made to see if a node is present in the deque or in the explored array, if not the **add_child** function is called to add the node as a child of the current node. The node will also be appended to the deque.

The Enforced Hill Climbing search algorithm works by using Breadth-First Search to expand and traverse each state or node in order to compare each heuristic level by level and decides which one is the best (cheapest). The heuristics are both provided by Manhattan distance and Misplaced tiles which are initialised for each new_node per iteration of the traversal. Once the first node which costs lower than the best heuristic is found, it is added as a child of the current node then the remaining list of nodes to search is cleared. This is one problem of enforced hill climbing as if one node has a lower cost than the first lower-cost node encountered it is not chosen by the algorithm thus sacrificing completeness and optimality. This process is repeated until the goal state is found.

Results of Evaluation

Graph Set-Up

A 2-D bar chart is used to represent how many expanded states each search algorithm uses depending on the heuristic. From the results provided we can then compare how each algorithm executes time and node-wise.

Manhattan Distance Heuristic

Figure 3 - Manhattan Distance Heuristic

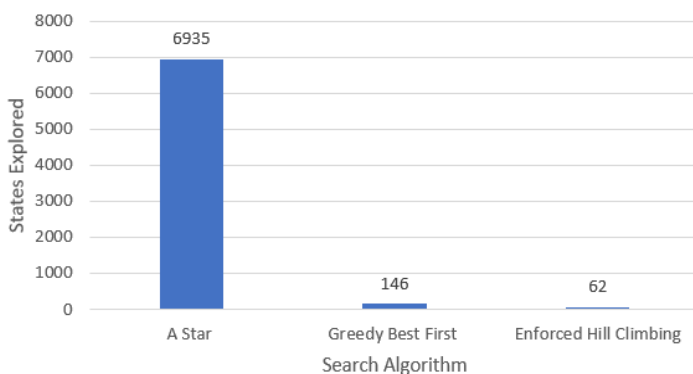
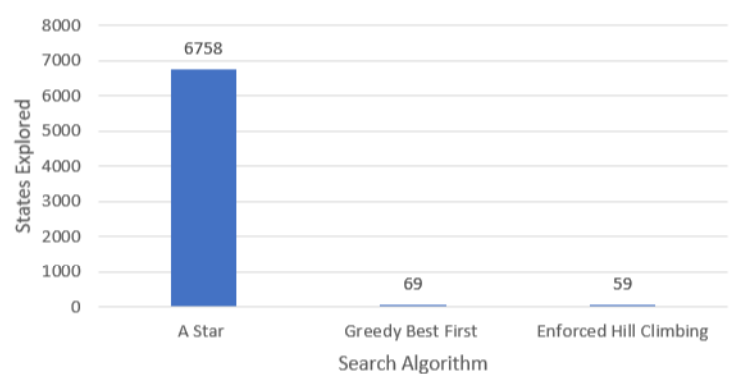


Figure 4 - Manhattan Distance



As can be seen from the graphs above A* explores the most states, followed by Greedy Best First Search and Enforced Hill Climbing. Interesting to note the vast difference in states explored between A* star and Greedy Best First Search, despite them both being Best first Searches. The only difference between them is the way they calculate the f-value. Since A* uses the formula $f(n) = g(n) + h(n)$, while Greedy utilises the formula $f(n) = h(n)$. So by discarding $g(n)$ which represents the cost so far to reach the current node, it is able to reach the goal in fewer states explored and much faster in terms of seconds. However that is to be expected since A* is complete while Greedy is not. A* has more processing to do since it is optimal. This means it will try to find the best path possible, which in this case hinders its speed. Greedy, on the other hand, does not concern itself with the optimal path, as long as it finds a path that leads to the goal, then its job is accomplished. Enforced hill climbing uses the same formula as A* to get its $f(n)$ value. The reason it is so much faster is because it is not optimal. This means that it does not keep track of all nodes it has visited, but empties the list and only keeps the current node, and expands from that node onward. It chooses the next node to continue on a first come first serve basis. This means that upon finding a child node, from the current node, with a lower $f(n)$ value, the child node is set as the new current node and the list is emptied. The time for Enforced Hill climbing Figure 3 was 0.03761739999981728 seconds and for Figure 4: 0.02825989999973899 seconds.

Misplaced Tiles Heuristic

Figure 3 - Misplaced Tiles Heuristic

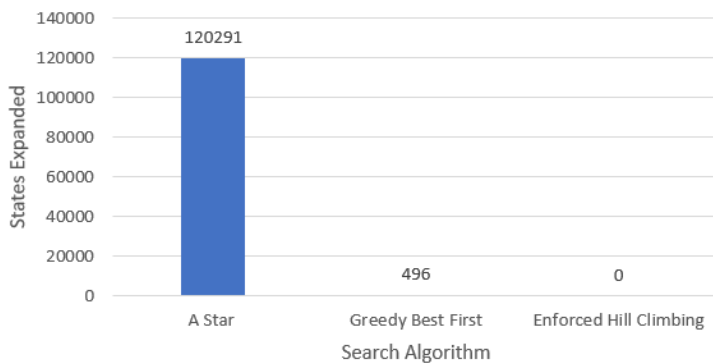
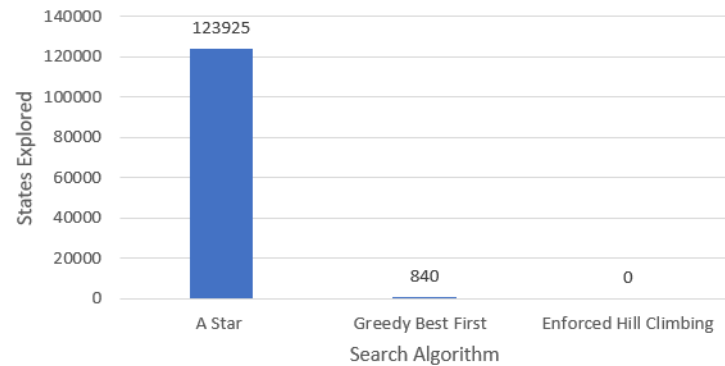


Figure 4 - Misplaced Tiles Heuristic



From the graphs shown in both figures, the A* search prevails in having the largest number of expanded states when compared to Greedy Best First Search. This may be due to both puzzles having a certain complexity to them thus the heuristic will calculate numerous misplaced tiles, thus, with the combination of A* which uses both $g(n)$ and $h(n)$, it will take into account more states, leading to a longer execution time coming at 6447.7504939 seconds for Figure 3 and 10475.9322708 seconds for Figure 4. Moreover, A* provides a complete and optimal solution thus increasing the computational load during the process as it is trying to find the best possible solution. For Greedy Best First Search this is not the case since it makes use of $h(n)$ only and is not optimal, thus reaching the goal is its main priority. This will prevent workload on the process as it does not make use of $g(n)$ which is the cost from the starting node to any other node. Thus, Greedy Best First search solves the puzzle much faster than A* search algorithm, having a total of 0.9493099999999686 seconds in Figure 3 and 1.9997683000000028 seconds in Figure 4, but sacrifices optimality. Since the misplaced tiles heuristic for Enforced Hill Climbing does not work we think that due to the algorithm choosing the cheapest cost for the path together with the summation of misplaced tiles, it will be faster than both A* and Greedy but less than the Enforced Hill Climbing using the Manhattan distance.

Breadth First Search

Figure 3 - Breadth First Search

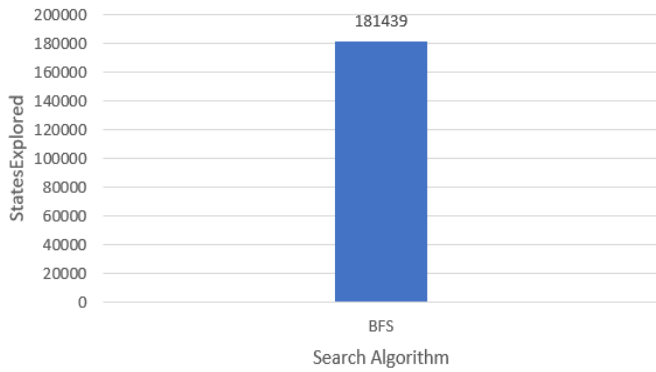
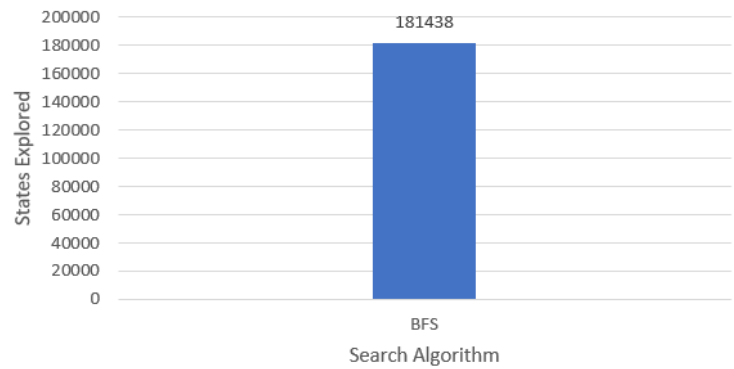


Figure 4 - Breadth First Search



As can be seen from the above figures, the Breadth First Search strategy was very expensive especially when compared to most of the other search strategies. This is due to the fact that the Breadth First Search strategy starts by traversing the graph from the initial state and explores all its neighbouring states level by level. Unlike most of the other search strategies, it had to traverse through a lot of nodes in order to reach the goal state. Hence, due to such complexity, the execution times for Figure 3 and 4 ended up being 14829.0611001 and 5959.7885446 seconds respectively. From all the implemented search strategies, the Breadth First Search strategy ended up being the one that expanded the most nodes in order to execute a valid plan. Despite this, surprisingly it was not the one that took the most time to generate a valid plan for Figure 4 as the A* Misplaced Tiles took more time.

Part 2 – PDDL Sliding Tile Puzzle

Domain Code

```
1 (define (domain strips-sliding-tile)
2   (:requirements :strips)
3   (:predicates
4     (tile ?t)
5     (position ?p)
6     (at ?t ?x ?y)
7     (blank ?x ?y)
8     (inc ?to ?from)
9     (dec ?to ?from)
10  )
11
12  (:action up
13    :parameters
14      (?t
15       ?x
16       ?y
17       ?new)
18    :precondition
19      (and (tile ?t) (position ?x) (position ?y) (position ?new) (at ?t ?x ?y)
20           (blank ?x ?new) (dec ?new ?y))
21    :effect (and (not (blank ?x ?new)) (not (at ?t ?x ?y))
22               (blank ?x ?y) (at ?t ?x ?new)))
23
24  (:action down
25    :parameters
26      (?t
27       ?x
28       ?y
29       ?new)
30    :precondition
31      (and (tile ?t) (position ?x) (position ?y) (position ?new) (at ?t ?x ?y)
32           (blank ?x ?new) (inc ?new ?y))
33    :effect (and (not (blank ?x ?new)) (not (at ?t ?x ?y))
34               (blank ?x ?y) (at ?t ?x ?new)))
35
36  (:action left
37    :parameters
38      (?t
39       ?x
40       ?y
41       ?new)
42    :precondition
43      (and (tile ?t) (position ?x) (position ?y) (position ?new) (at ?t ?x ?y)
44           (blank ?new ?y) (dec ?new ?x))
45    :effect (and (not (blank ?new ?y)) (not (at ?t ?x ?y))
46               (blank ?x ?y) (at ?t ?new ?y)))
47
48  (:action right
49    :parameters
50      (?t
51       ?x
52       ?y
53       ?new)
54    :precondition
55      (and (tile ?t) (position ?x) (position ?y) (position ?new) (at ?t ?x ?y)
56           (blank ?new ?y) (inc ?new ?x))
57    :effect (and (not (blank ?new ?y)) (not (at ?t ?x ?y))
58               (blank ?x ?y) (at ?t ?new ?y)))
59 )
```

Problems Code

Problem 1

```
1 (define (problem problem1)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x2 y2) (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
12    (at t4 x1 y2) (at t5 x2 y3) (at t6 x3 y2) (at t7 x1 y3)
13    (at t8 x3 y3))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
```

Problem 2:

```
1 (define (problem problem2)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x2 y3) (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
12    (at t4 x1 y2) (at t5 x3 y2) (at t6 x3 y3) (at t7 x1 y3)
13    (at t8 x2 y2))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
```

Problem 3:

```
1 (define (problem problem3)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x1 y3) (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
12    (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2) (at t7 x2 y3)
13    (at t8 x3 y3))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
20
```

Problem 4:

```
1 (define (problem problem4)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x3 y2) (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
12    (at t4 x1 y2) (at t5 x2 y3) (at t6 x2 y2) (at t7 x1 y3)
13    (at t8 x3 y3))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
```

Problem5 - hard

```
1 (define (problem problem5-hard)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x2 y3) (at t1 x3 y3) (at t2 x1 y2) (at t3 x1 y3)
12    (at t4 x3 y2) (at t5 x2 y2) (at t6 x2 y1) (at t7 x3 y1)
13    (at t8 x1 y1))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
```

Problem6 - hard

```
1 (define (problem problem6-hard)
2   (:domain strips-sliding-tile)
3   (:objects t1 t2 t3 t4 t5 t6 t7 t8 x1 x2 x3 y1 y2 y3)
4   (:init
5     (tile t1) (tile t2) (tile t3) (tile t4) (tile t5) (tile t6)
6     (tile t7) (tile t8)
7     (position x1) (position x2) (position x3)
8     (position y1) (position y2) (position y3)
9     (inc x1 x2) (inc x2 x3) (dec x3 x2) (dec x2 x1)
10    (inc y1 y2) (inc y2 y3) (dec y3 y2) (dec y2 y1)
11    (blank x3 y2) (at t1 x3 y3) (at t2 x2 y3) (at t3 x1 y3)
12    (at t4 x2 y1) (at t5 x2 y2) (at t6 x1 y1) (at t7 x3 y1)
13    (at t8 x1 y2))
14   (:goal
15     (and (at t1 x1 y1) (at t2 x2 y1) (at t3 x3 y1)
16           (at t4 x1 y2) (at t5 x2 y2) (at t6 x3 y2)
17           (at t7 x1 y3) (at t8 x2 y3) ))
18   )
19
```

Results of Evaluation

The following are the generated plans for the problems.

Problem 1:

PDDL Editor

File Session Import Solve Plugins Help

strips-sliding-tile.pddl

problem1.pddl

Plan (1)

Found Plan (output)

(down t5 x2 y3 y2)

(right t8 x3 y3 x2)

```
(:action down
:parameters (t5 x2 y3 y2)
:precondition
  (and
    (tile t5)
    (position x2)
    (position y3)
    (position y2)
    (at t5 x2 y3)
    (blank x2 y2)
    (inc y2 y3)
  )
:effect
  (and
    (not
      (blank x2 y2)
    )
    (not
      (at t5 x2 y3)
    )
    (blank x2 y3)
    (at t5 x2 y2)
  )
)
```

Statespace

Visited States: 7 Tree Height: 2

0% 100% Change Layout Download JSON

Plan length: 2

Number of states discovered: 7

Total time : -1.04308e-10

Nodes generated during search : 7

Nodes expanded during search : 2

Problem 2:

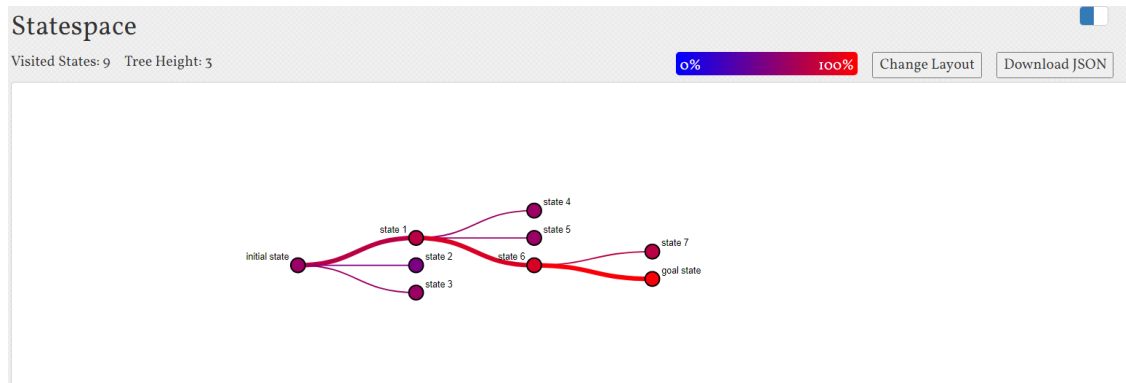
PDDL Editor File Session Import Solve Plugins Help

strips-sliding-tile.pddl
problem2.pddl
Plan (2)

Found Plan (output)

(up t8 x2 y2 y3)
(right t5 x3 y2 x2)
(down t6 x3 y3 y2)

```
(:action up
:parameters (t8 x2 y2 y3)
:precondition
  (and
    (tile t8)
    (position x2)
    (position y2)
    (position y3)
    (at t8 x2 y2)
    (blank x2 y3)
    (dec y3 y2)
  )
:effect
  (and
    (not
      (blank x2 y3)
    )
    (not
      (at t8 x2 y2)
    )
    (blank x2 y2)
    (at t8 x2 y3)
  )
)
```



Plan length: 3

Number of states discovered: 9

Total time : -1.04308e-10

Nodes generated during search : 9

Nodes expanded during search : 3

Problem3:

PDDL Editor

File Session Import Solve Plugins Help

strips-sliding-tile.pddl

problem3.pddl

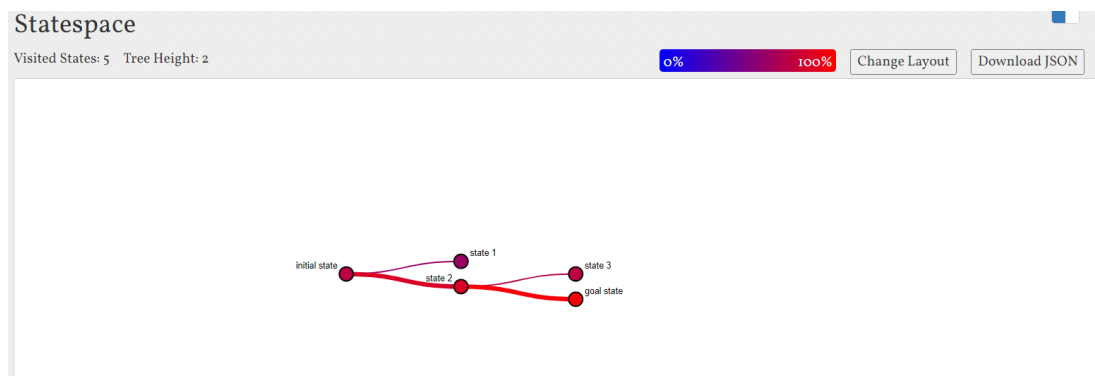
Plan (3)

Found Plan (output)

(right t7 x2 y3 x1)

(right t8 x3 y3 x2)

```
(:action right
:parameters (t7 x2 y3 x1)
:precondition
  (and
    (tile t7)
    (position x2)
    (position y3)
    (position x1)
    (at t7 x2 y3)
    (blank x1 y3)
    (inc x1 x2)
  )
:effect
  (and
    (not
      (blank x1 y3)
    )
    (not
      (at t7 x2 y3)
    )
    (blank x2 y3)
    (at t7 x1 y3)
  )
)
```



Plan length: 2

Number of states discovered: 5

Total time : -1.51992e-09

Nodes generated during search : 6

Nodes expanded during search : 2

Problem 4:

strips-sliding-tile.pddl

problem4.pddl

Plan (4)

Found Plan (output)

(left t6 x2 y2 x3)

(down t5 x2 y3 y2)

(right t8 x3 y3 x2)

```
(:action left
:parameters (t6 x2 y2 x3)
:precondition
  (and
    (tile t6)
    (position x2)
    (position y2)
    (position x3)
    (at t6 x2 y2)
    (blank x3 y2)
    (dec x3 x2)
  )
:effect
  (and
    (not
      (blank x3 y2)
    )
    (not
      (at t6 x2 y2)
    )
    (blank x2 y2)
    (at t6 x3 y2)
  )
)
```

Statespace

Visited States: 9 Tree Height: 3

0% 100% Change Layout Download JSON

```
graph LR
  initial_state((initial state)) --> state1((state 1))
  initial_state --> state2((state 2))
  initial_state --> state3((state 3))
  state3 --> state4((state 4))
  state3 --> state5((state 5))
  state3 --> state6((state 6))
  state5 --> state7((state 7))
  state5 --> goal_state((goal state))
```

Plan length: 3

Number of states discovered: 9

Total time : -8.64267e-10

Nodes generated during search : 9

Nodes expanded during search : 3

Problem5 - hard:

PDDL Editor

File
Session
Import
Solve
Plugins
Help

strips-sliding-tile.pddl
problem5-hard.pddl
Plan (5)

Found Plan (output)

```

(:action left
:parameters (t6 x1 y1 x2)
:precondition
  (and
    (tile t6)
    (position x1)
    (position y1)
    (position x2)
    (at t6 x1 y1)
    (blank x2 y1)
    (dec x2 x1)
  )
:effect
  (and
    (not
      (blank x2 y1)
    )
    (not
      (at t6 x1 y1)
    )
    (blank x1 y1)
    (at t6 x2 y1)
  )
)

```

(up t5 x2 y2 y3)

(up t6 x2 y1 y2)

(left t8 x1 y1 x2)

(down t2 x1 y2 y1)

(right t6 x2 y2 x1)

(down t5 x2 y3 y2)

(right t1 x3 y3 x2)

(up t4 x3 y2 y3)

(up t7 x3 y1 y2)

(left t8 x2 y1 x3)

(left t2 x1 y1 x2)

(down t6 x1 y2 y1)

(down t3 x1 y3 y2)

(right t1 x2 y3 x1)

(right t1 x2 y3 x1)

(right t4 x3 y3 x2)

(up t7 x3 y2 y3)

(up t8 x3 y1 y2)

(left t2 x2 y1 x3)

(left t6 x1 y1 x2)

(down t3 x1 y2 y1)

(down t1 x1 y3 y2)

(right t4 x2 y3 x1)

(right t7 x3 y3 x2)

(up t8 x3 y2 y3)

(up t2 x3 y1 y2)

(left t6 x2 y1 x3)

(left t3 x1 y1 x2)

(down t1 x1 y2 y1)

(down t4 x1 y3 y2)

(right t7 x2 y3 x1)

(right t7 x2 y3 x1)
(up t5 x2 y2 y3)
(right t2 x3 y2 x2)
(up t6 x3 y1 y2)
(left t3 x2 y1 x3)
(down t2 x2 y2 y1)
(down t5 x2 y3 y2)
(right t8 x3 y3 x2)

Statespace

Visited States: 557

Tree Height: 65

0%100%

Change Layout

Download JSON

Plan length: 37

Number of states discovered: 557

Total time : 0.004

Nodes generated during search : 190

Nodes expanded during search : 67

Problem6 - hard:

PDDL Editor

File
Session
Import
Solve
Plugins
Help

strips-sliding-tile.pddl
problem6-hard.pddl
Plan (6)

Found Plan (output)

(left t5 x2 y2 x3)
(left t8 x1 y2 x2)
(down t3 x1 y3 y2)
(right t2 x2 y3 x1)
(up t8 x2 y2 y3)
(right t5 x3 y2 x2)
(up t7 x3 y1 y2)
(left t4 x2 y1 x3)
(left t6 x1 y1 x2)
(down t3 x1 y2 y1)
(right t5 x2 y2 x1)
(right t7 x3 y2 x2)
(down t1 x3 y3 y2)
(left t8 x2 y3 x3)

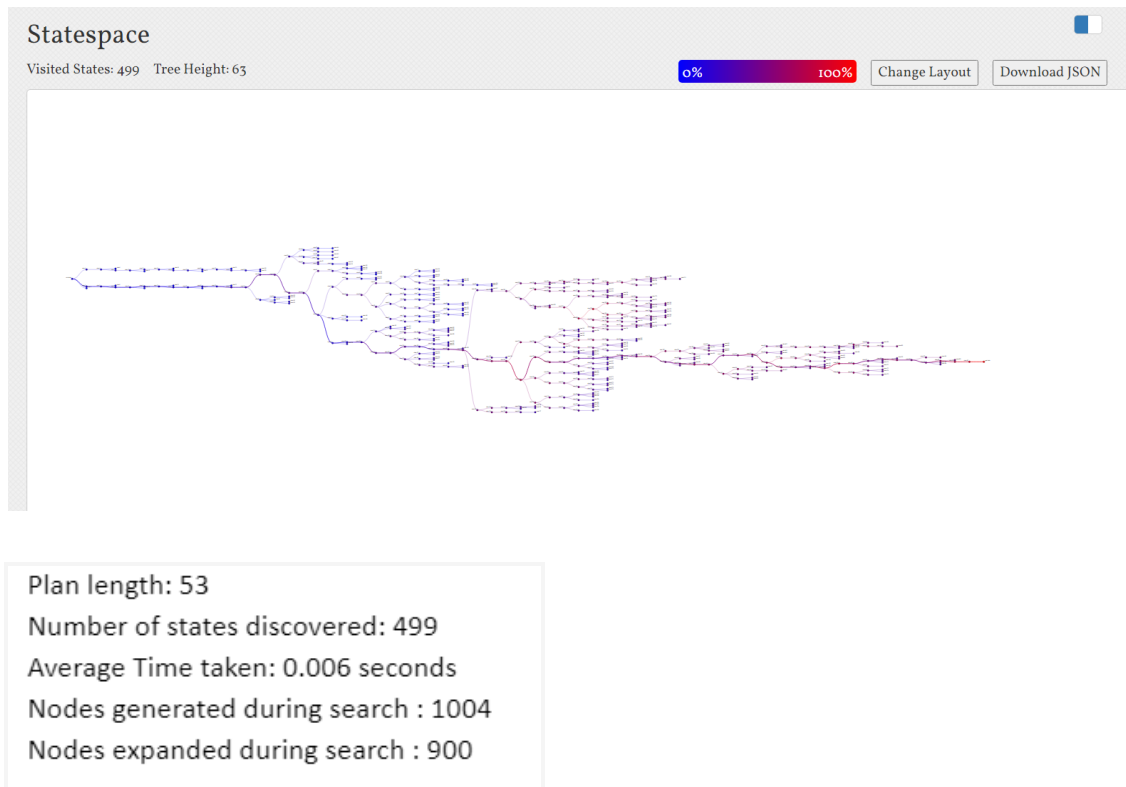
```

(:action left
:parameters (t5 x2 y2 x3)
:precondition
  (and
    (tile t5)
    (position x2)
    (position y2)
    (position x3)
    (at t5 x2 y2)
    (blank x3 y2)
    (dec x3 x2)
  )
:effect
  (and
    (not
      (blank x3 y2)
    )
    (not
      (at t5 x2 y2)
    )
    (blank x2 y2)
    (at t5 x3 y2)
  )
)

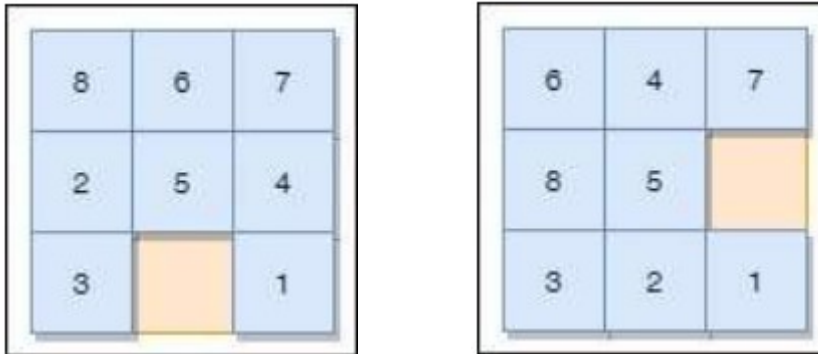
```

(left t8 x2 y3 x3)
(left t2 x1 y3 x2)
(up t5 x1 y2 y3)
(right t7 x2 y2 x1)
(up t6 x2 y1 y2)
(right t4 x3 y1 x2)
(down t1 x3 y2 y1)
(left t6 x2 y2 x3)
(down t2 x2 y3 y2)
(left t5 x1 y3 x2)
(up t7 x1 y2 y3)
(up t3 x1 y1 y2)
(right t4 x2 y1 x1)
(down t2 x2 y2 y1)
(down t5 x2 y3 y2)
(right t8 x3 y3 x2)
(up t6 x3 y2 y3)

	(up t6 x3 y2 y3)	
	(left t5 x2 y2 x3)	
	(left t3 x1 y2 x2)	
	(up t4 x1 y1 y2)	
	(right t2 x2 y1 x1)	
	(down t3 x2 y2 y1)	
	(right t5 x3 y2 x2)	
	(down t6 x3 y3 y2)	
	(up t6 x3 y2 y3)	
	(up t1 x3 y1 y2)	
	(left t3 x2 y1 x3)	
	(left t2 x1 y1 x2)	
	(down t4 x1 y2 y1)	
	(right t5 x2 y2 x1)	
	(right t1 x3 y2 x2)	
	(up t3 x3 y1 y2)	
	(left t2 x2 y1 x3)	
	(left t2 x2 y1 x3)	
	(down t1 x2 y2 y1)	
	(left t5 x1 y2 x2)	
	(up t4 x1 y1 y2)	
	(right t1 x2 y1 x1)	
	(right t2 x3 y1 x2)	
	(down t3 x3 y2 y1)	
	(down t6 x3 y3 y2)	



Comparison of results of Part 1 with Part 2



Taking the board states of the figures above as an example :

In our implementation the biggest difference between domain-specific solver and the domain-independent planner is consistency. By this we are referring to the time taken to find and return a good plan for different initial states. When using the initial states of the images above in the domain-independent planner, the number of states explored, time taken and plan length are all similar.

This is in contrast to the results we got from domain-specific solver, where the time taken, states explored and plan length differs by a considerable amount between each heuristic. The Manhattan heuristic is faster and more efficient than the Misplaced tiles heuristics. For example using the same search algorithm but different heuristics we get this :

Based on the initial state of Figure 3 :

Greedy: Manhattan: 146 states explored : plan length 46: 0.12685749999991458 seconds

Greedy: Misplaced: 496 states explored : plan length 116: 0.9493099999999686 seconds

Based on the initial state of Figure 4 :

Greedy: Manhattan: 69 states explored : 48 plan states : 0.04368990000011763 seconds

Greedy: Misplaced: 840 states explored : 144 plan states: 1.999768300000028 seconds

As can be seen there is a noticeable difference. This also applies to the other search algorithms.

From the results we obtained we have concluded that Greedy Best First is the fastest , followed by A*, Enforced Hill Climbing and Breadth First Search .

When comparing the two planners, we have come to the conclusion that the domain-specific solver utilising the Manhattan Distance heuristic is most optimal, in terms of speed, and obtaining the plan and states explored. Of course since Greedy Best First Search is not complete, this means that it runs the risk of taking a path that does not reach the goal state and may or not be the optimal path. While A* is complete and optimal. So there is a tradeoff while Greedy, might return results faster, A* will return the most optimal plan. As can be seen below:

A*: Manhattan: 6935 states explored : 84.29556179999986 seconds : plan length 32
Greedy: Manhattan: 146 states explored : 0.12685749999991458 seconds : plan length 46

A* clearly takes more time and explores more states, however returns a shorter plan.

The results we got from the domain-independent planner are :

problem5-hard:
Plan length: 37
Number of states discovered: 557
Time taken: 0.004 seconds
Nodes generated during search : 190
Nodes expanded during search : 67

problem6-hard
Plan length: 53
Number of states discovered: 499
Time taken: 0.006 seconds
Nodes generated during search : 1004
Nodes expanded during search : 900

After closely comparing the results we obtained, we believe that the domain-independent planner is generally better, since it consistently returns results in a short amount of time with a short plan length, and generally when compared to domain-specific solver, it explores less states when taking into account all the search algorithms and heuristics.

References

Greedy Best First Search :

- [1] nbro. "What are the differences between A* and greedy best-first search?" StackExchange.
<https://ai.stackexchange.com/questions/8902/what-are-the-differences-between-a-and-greedy-best-first-search> (Accessed Dec 13, 2021)
- [2] "Informed Search Algorithms." javatpoint.com.
<https://www.javatpoint.com/ai-informed-search-algorithms> (Accessed Dec 13, 2021)
- [3] Great Learning Team. " Best First Search Algorithm in AI | Concept, Implementation, Advantages, Disadvantages." mygreatlearning.com
<https://www.mygreatlearning.com/blog/best-first-search-bfs/> (Accessed Dec 13, 2021)

Breadth First Search:

- [4] "Breadth-first search" en.wikipedia.org.
https://en.wikipedia.org/wiki/Breadth-first_search#Example (Accessed Dec. 13, 2021)
- [5] J. D. Canty. "Graph Traversal: solving the 8-puzzle with basic A.I." jackcanty.com.
<https://jackcanty.com/solving-8-puzzle-with-artificial-intelligence.html> (Accessed Dec. 16, 2021)
- [6] GeeksforGeeks. "Breadth First Search or BFS for a Graph" geeksforgeeks.org.
<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/> (Accessed Dec. 17, 2021)
- [7] "8 Puzzle problem in AI (Artificial Intelligence)." goeduhub.com.
<https://www.goeduhub.com/8062/8-puzzle-problem-in-ai-artificial-intelligence> (Accessed Dec. 18, 2021)
- [8] R. Aware. "Solve the Slide Puzzle with Hill Climbing Search Algorithm" towardsdatascience.com.
<https://towardsdatascience.com/solve-slide-puzzle-with-hill-climbing-search-algorithm-d7fb93321325> (Accessed Dec. 18, 2021)
- [9] nikhilagarwal3. "Queue in Python" geeksforgeeks.org.
<https://www.geeksforgeeks.org/queue-in-python/> (Accessed Dec. 22, 2021)
- [10] GeeksforGeeks. "Deque in Python" geeksforgeeks.org.
<https://www.geeksforgeeks.org/deque-in-python/> (Accessed Dec. 23, 2021)

[11] S. Bhadaniya. "Breadth First Search in Python (with Code) | BFS Algorithm." Favtutor.com [Breadth First Search in Python \(with Code\) | BFS Algorithm | FavTutor](#) (Accessed Jan. 5, 2022)

Enforced Hill Climbing:

[12] J. Bajada. (2021). Lecture 7 – Planning Graph Heuristics [PowerPoint slides] (Accessed Jan. 5, 2022)

Part 1:

[13] AmirUCR. "eight_puzzle_solver_heuristic_search" github.com [AmirUCR/eight_puzzle_solver_heuristic_search: Attempts to solve the 8-Puzzle by using various heuristic search methods. Written in Python. \(github.com\)](#) (Accessed Dec. 6, 2021)

[14] Gabriel-RCastro "eight-puzzle." github.com https://github.com/nakahwra/eight-puzzle?fbclid=IwAR3Ryc_4djz1Pw5-brg93G_DTD_q40GMtv9KAqdm-m1ZfjnvQymQxTMAXGE (Accessed Dec. 6, 2021)

[15] microStationCorp. "8_puzzle" github.com [8_puzzle/puzz_8.py at master · microStationCorp/8_puzzle \(github.com\)](#) (Accessed Dec. 6, 2021)

PDDL:

[16] SoarGroup. "Domains-Planning-Domain-Definition-Language" github.com. <https://github.com/SoarGroup/Domains-Planning-Domain-Definition-Language/tree/master/pddl> (Accessed Jan. 8, 2022)

[17] "Writing Planning Domains and Problems in PDDL" <http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html> (Accessed Jan. 8, 2002)

Distribution of Work

Dylan Zerafa:

- Source Code Part 1:
 - Greedy Best First Search
 - Enforced Hill climbing
 - Combined/alterd all algorithms
- Source Code Part 2:
 - Created the PDDL problem files
 - Aided in PDDL Domain files
- Report:
 - Report writing is shared amongst the group equally
 - Co-wrote the comparison between domain-specific solver and domain-independent planner

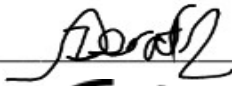
Cristina Cutajar:

- Source Code Part1:
 - Breadth First Search
 - Validation Function
 - Combined/alterd all algorithms
- Source Code Part2:
 - PDDL Domain File Code
- Report:
 - Report writing is shared amongst the group equally
 - Co-wrote the comparison between domain-specific solver and domain-independent planner

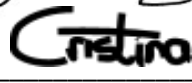
Gabriel Vella:

- Source Code Part1:
 - A Star Search
 - Enforced Hill Climbing
 - Combined/alterd all algorithms
- Source Code Part2:
 - PDDL Domain File Code
 - Aided in PDDL Problem Files
- Report
 - Graphs For Analysis with written analysis
 - Report writing shared amongst the group equally

Dylan Zerafa



Cristina Cutajar



Gabriel Vella

