# Search: Sliding Blocks Puzzle

## Aim:

To implement Breadth First Search (BFS) and AStar Search, and effectively solve the Sliding Blocks puzzle.

## Software Used:

Python 3.7, latest version of numpy, copy, queue, os and time libraries.

## Program Design:

### State Space:

An integer state space was used, in order to reduce the memory, it consumes. The following were the representations followed for a 3x3 and 4x4 puzzle.

**3x3:**

Let us consider the following State:

| | | |
|---|---|---|
| A1 | A2 | A3 |
| A4 | A4 | A6 |
| A7 | A8 | A9 |

Step-1:

The 2D array is flattened as [A1, A2, A3, A4, A5, A6, A7, A8, A9]

Step-2:

Each element (with 0-8 range) was multiplied with a power of 9 as follows:

State = $A1*9^0 + A2*9^1 + A3*9^2 + \ldots\ldots + A9*9^8$

This representation ensured that each state occupies only an integer size of 4 bytes and saved memory compared to string representation which would have taken a total of 9 bytes.

Decoding:

The state can be converted back into the 2D array format, by applying this simple formula for each element:

$$A_n = (\text{state} / 9^n) \% 9$$

**4x4:**

Let us consider the following State:

| | | | |
|---|---|---|---|
| A1 | A2 | A3 | A4 |
| A5 | A6 | A7 | A8 |
| A9 | A10 | A11 | A12 |
| A13 | A14 | A15 | A16 |

Step-1:

Similar to 3x3, the 2D array is flattened as [A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16]

Step-2:

Each element (with 0-15 range) was multiplied with a power of 15 as follows:

State = $A1*16^0 + A2*16^1 + A3*16^2 + \ldots\ldots + A16*16^{15}$

The above representation was further optimised in the code as:

State = (A1 << 4*0) + (A2 << 4*1) + (A3<< 4*2) + . . . . .+ (A16 << 4*15)

The << (left-shift) operation with 4 is same as multiplying a number by 16, but this method saves a lot of compute time than an actual multiplication process.

This representation ensured that each state occupies only an unsigned long integer size of 8 bytes and saved memory compared to string representation which would have taken a total of 16 bytes.

Decoding:

The state can be converted back into the 2D array format, by applying this simple formula for each element:

$$A_n = (\text{state} / 16^n) \% 16$$

This operation was implemented in the code as:

$$A_n = (\text{state} >> 4*n) \& 15$$

Here the >> (right-shift) of 4*n, is used to divide a number by $16^n$ and '&' (logical-and) with 15 is used to apply modulus (%) of 16. Both >> and & operators are used to fasten the speed of the division and modulus operations.

**Action Representations:**

Since the problem state only constrained the action space to 4 directions, but not to 4 actions, a total of 8 and 12 possible actions were chosen for the 3x3 and 4x4 puzzles respectively. The action representations and their corresponding integers are as follows:

**3x3 Actions:**

| Action-ID (Integer) | String Representation | Description |
|---|---|---|
| 0 | L1 | Move left by 1 step |
| 1 | L2 | Move left by 2 steps |
| 2 | U1 | Move up by 1 step |
| 3 | U2 | Move up by 2 steps |
| 4 | R1 | Move right by 1 step |
| 5 | R2 | Move right by 2 steps |
| 6 | D1 | Move down by 1 step |
| 7 | D2 | Move down by 2 steps |

**4x4 Actions:**

| Action-ID (Integer) | String Representation | Description |
|---|---|---|
| 0 | L1 | Move left by 1 step |
| 1 | L2 | Move left by 2 steps |
| 2 | L3 | Move left by 3 steps |
| 3 | U1 | Move up by 1 step |
| 4 | U2 | Move up by 2 steps |
| 5 | U3 | Move up by 3 steps |
| 6 | R1 | Move right by 1 step |
| 7 | R2 | Move right by 2 steps |
| 8 | R3 | Move right by 3 steps |
| 9 | D1 | Move down by 1 step |
| 10 | D2 | Move down by 2 steps |
| 11 | D3 | Move down by 3 steps |

**Breath First Search (BFS):**

(i) The algorithm was implemented with a queue which stores the nodes to be explored next and a visited nodes pile to ensure that same nodes aren't visited again and to backtrack the solution once the goal node is reached.

(ii) The queue was implemented using the 'Queue object inside the queue library' of Python so that, the time complexity of both adding and removing an element is O(1). This was preferred over the Python's inbuilt list data structure since, although it's time complexity for adding an element is O(1), the complexity for popping an element is O(n) which makes it an inefficient choice.

(iii) The node pile containing the visited nodes was implemented using the 'Dictionary data structure inbuilt in Python' since, in each iteration the node pile has to be checked if the current node is already visited, and dictionary is the best data structure for this application, since it uses an inbuilt "hash table" and has the time complexity to access an element just as O(log(n)).

(iv) Each node contained it's state integer and the address of it's last added node in the node pile for the sake of backtracking at the end to find the solution.

**AStar Search (A*):**

(i) This method is very similar to the BFS algorithm, with and addition of a heuristic factor, based on which the Queue containing the nodes to be explored in the future is sorted in every step based on the sum of its cost and heuristic value, so that the nodes which appear close to the solutions are given more priority to explore.

(ii) Since the queue had to be sorted in every step, a priority queue was used here.

(iii) The priority queue was implemented using a linked list, since a new node can be added in the correct spot based on the order without changing the memory address of the other nodes. In addition the time complexity of removing the 1st element from this list is O(1).

(iv) For the heuristic function the Manhattan Distance of all the blocks from it's goal positions was used.

(v) Each node was assigned the state, address of previous node added to the node pile, the cost (number of steps traversed from start), the evaluation function which is the sum of cost and heuristic and the address of the next node in the priority queue since we implemented a linked list.

**AStar Search (A*) with Pattern Database Heuristic:**

(i) A Pattern Database heuristic for formed by performing a Exhaustive BFS Search from the goal node, till it runs out of nodes, and the steps taken to reach each node, was taken as the cost and the list of all the nodes (states) and their costs were stored in a hash table, which was saved in the disk.
(ii) For 3x3 Space implementing a complete Pattern Database was possible which took only 10 seconds, whereas for the 4x4 Space the 4x4 board was divided into 4 symmetrical 2x2 boards and their Pattern Database was individually calculated, each time while considering the remaining 3 2x2 elements as null.
(iii) Now the same A* Algorithm was implemented with the Pattern Database value as the heuristic function which improved the performance by a lot and gave the optimal solution for 3x3 Space everytime.

## Results:

Just as expected the A* algorithm was faster than the BFS and the A* with Pattern Database Heuristic was much faster than the A* itself.

The performances for the Easy, Medium and Hard levels can be found below:

**All times are in seconds**

**Easy:**

| **Sample no.** | **BFS Time** | **A* Time** | **A* with PDB time** | **Optimal actions** |
|---|---|---|---|---|
| **1** | 0.00099 | 0 | 0.00097 | ['D2', 'R2'] |
| **2** | 0.00299 | 0.00199 | 0.00099 | ['U1', 'L1', 'D1', 'R2'] |
| **3** | 0.00895 | 0.00299 | 0.00102 | ['L1', 'D1', 'R1', 'U1', 'R1', 'D1'] |
| **4** | 0 | 0 | 0 | Already Optimal |
| **5** | 0.00797 | 0.00199 | 0.00096 | ['D1', 'L1', 'U1', 'R1', 'D1'] |
| **6** | 0.00698 | 0.00299 | 0.00103 | ['D1', 'R1', 'U2', 'R1', 'D2'] |

|  |  |  |  |  |
|---|---|---|---|---|
| **7** | 0.00500 | 0.00299 | 0.00095 | ['L2', 'D1', 'R1', 'D1', 'R1'] |
| **8** | 0.00597 | 0.00099 | 0.00101 | ['R1', 'U1', 'L1', 'D2', 'R1'] |
| **9** | 0.00198 | 0.00099 | 0 | ['D1', 'R1', 'D1'] |
| **10** | 0.00599 | 0.00199 | 0.00099 | ['U1', 'L1', 'D1', 'R2'] |
| **Total Time** | 0.04785 | 0.01695 | 0.00797 | - |

**Medium:**

| **Sample no.** | **BFS Time** | **A* Time** | **A* with PDB time** | **Optimal actions** |
|---|---|---|---|---|
| **1** | 0.26230 | 0.03492 | 0.00101 | ['R1', 'D2', 'R1', 'U1', 'L1', 'U1', 'R1', 'D1', 'L2', 'D1', 'R2'] |
| **2** | 0.23625 | 0.03492 | 0.00199 | ['L1', 'D2', 'R1', 'U1', 'R1', 'D1', 'L1', 'U1', 'L1', 'D1', 'R2'] |
| **3** | 1.70448 | 0.14163 | 0.00299 | ['R1', 'U1', 'R1', 'D1', 'L2', 'U1', 'R2', 'U1', 'L1', 'D2', 'L1', 'U1', 'R2', 'D1'] |
| **4** | 0.00498 | 0.00100 | 0 | ['U1', 'L1', 'D1', 'R2', 'D1'] |
| **5** | 0.01994 | 0.00298 | 0.00099 | ['U1', 'L1', 'U1', 'R1', 'D1', 'R1', 'D1'] |
| **6** | 0.32711 | 0.02590 | 0.00199 | ['D1', 'R2', 'D1', 'L1', 'U1', 'L1', 'D1', 'R1', 'U2', 'R1', 'D2'] |
| **7** | 0.00299 | 0.00101 | 0.00099 | ['L2', 'U1', 'R2', 'D1'] |

| | | | | |
|---|---|---|---|---|
| **8** | 0.01392 | 0.00197 | 0.00099 | ['D2', 'L1', 'U1', 'R1', 'D1', 'R1'] |
| **9** | 0.16059 | 0.01196 | 0.00099 | ['D1', 'L1', 'D1', 'R1', 'U1', 'L1', 'U1', 'R1', 'D2', 'R1'] |
| **10** | 0.03599 | 0.00496 | 0.00197 | ['L2', 'U1', 'R1', 'U1', 'L1', 'D1', 'R2', 'D1'] |
| **Total Time** | 2.76962 | 0.26130 | 0.01396 | - |

**Difficult:**

| **Sample no.** | **BFS Time** | **A* Time** | **A* with PDB time** | **Optimal actions** |
|---|---|---|---|---|
| **1** | 1.04318 | 0.12468 | 0.00199 | ['D1', 'R1', 'D1', 'L1', 'U1', 'R1', 'U1', 'L2', 'D1', 'R1', 'U1', 'R1', 'D2'] |
| **2** | 9.78183 | 18.47485 | 0.00299 | ['D1', 'R1', 'U2', 'R1', 'D1', 'L1', 'U1', 'L1', 'D1', 'R2', 'D1', 'L2', 'U1', 'R2', 'D1', 'L1', 'U2', 'R1', 'D2'] |
| **3** | 8.08594 | 3.89285 | 0.00399 | ['D1', 'L1', 'D1', 'R1', 'U1', 'L1', 'U1', 'R1', 'D1', 'L2', 'U1', 'R1', 'D2', 'L1', 'U2', 'R1', 'D2', 'R1'] |
| **4** | 7.98465 | 8.90158 | 0.00199 | ['R1', 'D2', 'L2', 'U2', 'R2', 'D2', 'L1', 'U1', 'R1', 'D1', 'L2', 'U2', |

| | | | | |
|---|---|---|---|---|
| | | | | 'R1', 'D1', 'L1', 'U1', 'R2', 'D2'] |
| **5** | 1.95079 | 0.18352 | 0.00199 | ['D1', 'R1', 'U2', 'L1', 'D2', 'R2', 'U1', 'L1', 'U1', 'R1', 'D1', 'L2', 'D1', 'R2'] |
| **6** | 4.83573 | 0.91051 | 0.00199 | ['D1', 'R1', 'U1', 'L2', 'D1', 'R2', 'U2', 'L2', 'D1', 'R1', 'U1', 'R1', 'D1', 'L1', 'D1', 'R1'] |
| **7** | 0.90757 | 0.11471 | 0.00201 | ['L2', 'U1', 'R2', 'D1', 'L2', 'D1', 'R2', 'U2', 'L1', 'D1', 'L1', 'D1', 'R2'] |
| **8** | 2.45251 | 0.30616 | 0.00296 | ['U1', 'R2', 'D2', 'L2', 'U1', 'R1', 'D1', 'L1', 'U2', 'R2', 'D2', 'L1', 'U1', 'R1', 'D1'] |
| **9** | 0.41688 | 0.04390 | 0.00101 | ['R2', 'U2', 'L1', 'D2', 'L1', 'U1', 'R2', 'U1', 'L2', 'D2', 'R2'] |
| **10** | 2.71478 | 0.18548 | 0.00299 | ['R1', 'U1', 'L2', 'D1', 'R1', 'D1', 'R1', 'U2', 'L2', 'D2', 'R2', 'U2', 'L2', 'D2', 'R2'] |
| **Total Time** | 40.17786 | 33.13828 | 0.02395 | - |

**Extreme:**

I implemented the Pattern Database Heuristic for the 4x4 Space as well, by dividing the board into 4 pieces of 2x2 blocks each. But that wasn't sufficient enough to solve the Extreme cases under a minute. The Pattern Database took 10 seconds to be computed for the 3x3 world and a total of 250 seconds to be computed for the 3x3 world.

However, the 4x4 Space could have been solved faster if I just used 4 pieces of 2x3 blocks with overlap or it would have been even faster if I used 2 pieces of 2x4 blocks. But however, the Pattern Database of 2x3 blocks would have taken a few to several hours to be generated and for the 2x4 blocks it would have taken about 1-2 days to be generated. But unfortunately, I did not have the time and resources to try that out, since I have Assignment deadlines from the other subjects on 4 consecutive days which are time consuming. But since the code for generate Pattern Database for any block size is just the same, I have attached my Pattern Database codes of both 3x3 and 4x4 blocks along with this submission.

## Discussion:

- Instead of the conventional method of solving this problem by using just 4 actions, I have solved it using 8 & 12 actions for the 3x3 and 4x4 worlds respectively.
- It was interesting to note that, almost every sample had the same optimal paths, even in this new action space (except 2 cases). [For example, a 'L2' in the new action space is same as two consecutive 'L' actions in the conventional 4 action space].
- Even the 2 Samples which had different paths in this new action space, were the most optimal paths in this new action space. But still it was interesting to note that 28 out of 30 samples had the same optimal path when the problem was approached using these 2 different action states.
- The number of total steps to reach the solution was reduces since the number of actions was increased. But however, upon counting 'L2' as 2 steps and 'L3' as 3 steps (and similarly for the other 3 directions also) it was found that the total number of steps (of the optimal solution) was the same for 28 out of 30 cases.
- It was observable in A* that, upon adjusting the weights between the heuristic function and the step value in the evaluation function, the speed of execution of the A* Search was changed, but beyond a limit it ended up not returning the optimal path. So, an optimal ratio between the speed and performance in A* Search had to be adjusted for it to return the optimal path every time. This is because optimising the evaluation function allows us to adjust the breath of the A* Search.

- However, when the Pattern Database heuristic was used with A* Search the speed of execution was **improved by 700 times!** (and the optimal path was returned every time).
- And finally, the BFS, A* & A* with Pattern Database Heuristic were implemented successfully with A* with Pattern Database Heuristic being 700 times faster than A* with Manhattan distance & 900 times faster than the BFS Algorithm.