

Name : Tanmay Santosh Pandit
Roll No. : B18CSE055
Course : CS323 Artificial Intelligence
Instructor : Dr. Yashaswi Verma
Topic : Solving NxM sliding tiles puzzle using AI search techniques

Introduction :

Games and puzzles have always been attracting the attention of computer scientists. The attempts to produce Artificial Intelligence to play games better than even the best human players have been made and also came to fruition in case of games like checkers, backgammon and chess. Puzzles are also a kind of games. The topic of my project is one such puzzle called sliding tiles puzzle.

In this puzzle, we are given two NxM grids of tiles, consisting of numbers from 1 to $N*M - 1$, each written on exactly one tile and a blank space. The first grid is the initial configuration or start state while the second grid is the goal state. The operation allowed is swapping any neighbouring tile (diagonal swaps are not allowed) with the blank space. Our aim is to reach the goal state by applying a sequence of operations on the start state.

This problem can be easily posed as a search problem. Its smallest variation, 3x3 puzzle, commonly known as 8-puzzle is optimally solvable by uninformed search techniques like BFS, due to the small state space of $9! < 4 \times 10^5$. However, for larger puzzles, say 4x4 or 15-puzzle, we need informed search algorithms like A* and Iterative Deepening A* (IDA*) with strong heuristics for getting an optimal solution in a decent time. As the constraints further increase, getting optimal solutions becomes more and more difficult. The best algorithms in the world also need hours to solve an average case of 5x5 or 24-puzzle.

In my project, I have compared the results of uninformed and informed search algorithms for small puzzles ($N \times M \leq 9$). I have implemented informed search algorithms with complicated heuristics to solve puzzles with $9 < N \times M \leq 16$. Also, for the same, I have suggested a novel heuristic that gives decently good sub-optimal solutions much faster than the algorithms that provide optimal solutions. For larger puzzles with $16 < N \times M \leq 36$, I have implemented greedy search with strong heuristics that gives suboptimal solutions. I also experimented with the use of a local search technique called Simulated Annealing for the same. But the solutions it provided were extremely non-optimal in comparison to greedy search.

Problem Definition :

15	2	1	12
8	5	6	11
4	9	10	7
3	14	13	

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

The sliding tiles puzzle problem can be posed as a search problem, where a two dimensional vector of the position of tiles in the grid acts as a state. The blank space is denoted by 0. The successor function is swapping any neighbouring tile with the blank space. The branching factor is 4, since the blank space could be swapped in 4 directions. The goal test is whether the current state is exactly the same as our goal state.

The cost of the solution is the total number of swaps of tiles with the blank space. Every move is assigned a cost 1. The solution/path is represented by a string consisting of 'L'(left), 'R'(right), 'U'(up) and 'D'(down), each character denoting the direction of movement of blank space.

Background Survey :

This puzzle has always attracted the attention of mathematicians and computer scientists and many solutions have been proposed for different sizes of the puzzle. Solving small puzzles e.g. 8-puzzle optimally is trivial with uninformed search algorithms like BFS, IDDFS and with A* search using Manhattan distance heuristic.

For bigger puzzles, more complicated heuristics like linear conflict heuristic, corner tiles heuristic and pattern database heuristics were proposed. IDA* with these heuristics made the 15-puzzle optimally solvable in a time frame of 5 minutes. But the best algorithm in the world that uses IDA* search taking full advantage of all these heuristics and uses Finite State Machines to prune duplicate nodes in search also needs hours to solve an average case of 24-puzzle. It took this algorithm 3 months to solve the worst case of 24-puzzle, which expanded 8×10^{12} nodes. There is no known efficient algorithm that guarantees the optimal solution for puzzles bigger than 24-puzzle.

Discussion :

There was no challenge in solving an 8-puzzle. For 15-puzzle, I implemented two optimal strategies : IDA* with Manhattan + linear conflict + corner tiles heuristic and IDA* with pattern databases. The faster of these two could solve an average case of 15-puzzle in under a minute. But for worst cases, the running time could blow up to 5 minutes or more. So, to speed it up, I made a novel modification to linear conflict heuristic. With that modification, my algorithm can now solve almost all instances of the 15-puzzle in under 5 seconds. The solution it gives is suboptimal but very close to the optimal solution. The average loss incurred in cost due to my solution as compared to the optimal solution was found to be 5, whereas in the rare worst case scenario, it could possibly go up to 15.

There's no known efficient algorithm that can solve bigger puzzles even in a time constraint of 1 hour. So, I focussed my attention on suboptimal solutions. I experimented on a local search algorithm called simulated annealing for that purpose. It could solve most of the bigger puzzles (with the exception of the worst cases of 35-puzzle) in under 2 minutes but the solution was extremely non-optimal. The cost of the solution exceeded 10^6 so I needed a better algorithm.

I shifted my attention to greedy search. I performed greedy search with Manhattan + linear conflict with my novel modification + corner tiles heuristic. The results it gave were much better. It solved all instances of 24-puzzle in under 2 seconds with an average solution length of 400, whereas it solved all instances of 35-puzzle in under 6 seconds with an average solution length of 1000.

The solution I proposed could be called an engineering based solution because it considers a trade-off between optimality and the computation power/ running time. Our computers don't have such a huge computation power. So, with the novel modification I proposed, my algorithm gives a decently suboptimal solution very swiftly with the resources that are available to us.

Algorithms and Algorithms Complexity :

The Solvability of an NxM Puzzle :

We can represent the state as a permutation of size $(N*M - 1)$, ignoring the blank space. If the relative position of two numbers is reversed in the current state w.r.t. the goal state, that is called as an inversion.

The conditions for solvability of NxM sliding tiles puzzle are :

1. If the grid width is odd, then the number of inversions in a solvable situation is even.
2. If the grid width is even, and the absolute difference between the row number of blank in start state and goal state is odd then the number of inversions in a solvable situation is odd.

3. If the grid width is even, and the absolute difference between the row number of blank in start state and goal state is even then the number of inversions in a solvable situation is even.

The idea is, if the width of the puzzle is odd, swaps always preserve the parity of inversions. In case, it's even, every vertical swap changes the parity by 1 so the parity of inversions should be the same as the parity of absolute difference between the row numbers of blank space in start and goal state.

If the given puzzle is unsolvable, we can know it with these conditions and just say that the puzzle is unsolvable and terminate the code

There are different algorithms for different sizes of the puzzle.

$N \times M \leq 9$:

1. BFS :

The state space is very small $< 4 \times 10^5$ so it is optimally solvable by BFS from the start state till we reach the goal state and then backtracking the path. Inside each iteration of BFS, we have to find the successors of the current state. The complexity of the successor function is $O(b \times N \times M)$. In the worst case, the number of iterations could be equal to the state space = $(N \times M)!$. Hence, the time complexity of this uninformed search algorithm is $O((N \times M)! \times N \times M \times b)$. Moreover, the constant factor is high due to the use of map to store the previous move for backtracking the path once we reach the goal state.

A state is a 2-dimensional $N \times M$ vector so the space required to store a unit state is $O(N \times M)$. The upper bound on the maximum number of states in the queue can be given by the size of state space. So, the space complexity of the algorithm is $O((N \times M)! \times N \times M)$.

The average time taken to solve an 8-puzzle with BFS was found to be 3 seconds whereas it took 5 seconds for the worst case of 8-puzzle. But we can do better than that.

There is no point in using UCS separately, since it would be exactly the same as BFS in this case because all the moves have the same cost.

2. A* with Manhattan Heuristic :

We can achieve better performance with informed search algorithms like A*. We need a heuristic function to predict the cost of reaching the goal state from the current state for that.

For a moment, let's relax the constraints of the problem. Let's assume any tile could move to any neighbouring position irrespective of the position of other tiles. In this case, the number of moves required to bring any tile to its desired position would be the Manhattan distance between the initial and final position of that tile. So, the total number of moves required to solve this relaxed puzzle would be the summation of Manhattan distance for all tiles

(obviously excluding blank space because if all the rest tiles are in their desired position, blank space would automatically have its desired position.). Since it is a relaxed problem heuristic, it is consistent and admissible since it acts as a lower bound for the cost of the original puzzle.

The theoretical time and space complexity of the algorithm is same as BFS but in practice, it works much faster since it computes the expected total cost of the states with the use of heuristic and only traverses the states with the minimum total cost at every point, thus achieving a little deviation from the optimal path and hence avoiding majority of states that are not in the optimal path.

This algorithm successfully and optimally solved all instances of 8-puzzle in under 2 seconds, thus surpassing the performance of BFS.

Recommended Algorithm :

To solve any puzzle with $N \times M \leq 9$, I highly recommend A* search with Manhattan distance heuristic since it is extremely fast and gives the optimal solution.

$9 < N \times M \leq 16$:

Using A* with Manhattan doesn't suffice in case of these puzzles. Manhattan heuristic is not a good lower bound for the cost. It underestimates the value a little too much. So, we need some advancements in our heuristic function. Let's first discuss different heuristics before going to the actual algorithm.

Linear Conflict Heuristic :

15	2	1	12
8	5	6	11
4	9	10	7
3	14	13	

Start state

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Goal state

Linear conflict is said to occur when two tiles are in their goal row or column but are reversed relative to their goal position. E.g. In this case, 1 and 2 are in their goal rows but their

relative ordering is reversed. Now if you have to fix this relative ordering, one of the tiles 1 and 2 has to move up/down, let the other pass and then return back to the goal row. This would consume 2 extra moves that were not counted in Manhattan heuristic. So, we can add 2 moves without affecting the admissibility of heuristic.

However, adding 2 moves for every pair in linear conflict does not make sense. Suppose the goal row is (1 2 3 4) and the current row is (3 5 2 1), there are 3 pairs in linear conflict : (1 2), (1 3) and (2 3). But we can't add 6 moves here because if we just move aside tile 2, two conflicts : (1 2) and (2 3) are solved. Then moving aside one of tile 1 and 3 would solve the remaining conflict and we are done in 4 moves.

Here is the step by step procedure to calculate the heuristic for every row and column :

1. List the number of conflicts every tile faces.
2. Greedily move aside (i.e. erase while implementing) the tile with the maximum conflict.
3. Decrement the number of conflicts by 1, for all the tiles that were in conflict with the erased one.
4. Add 2 moves to the cost.
5. Repeat step 2, 3 and 4 until all conflicts are resolved.

Linear conflicts exist in both rows and columns but we can add them independently because the moves required to resolve row conflict are vertical whereas the other ones are horizontal.

The step by step procedure explained above has the time complexity $O(N*M)$ and since we have to repeat it for every row and column, the time complexity of this heuristic is $O((N+M)*N*M)$. The space complexity is $O(N)$ (in case of column conflict) or $O(M)$ (in case of row conflict).

Corner-Tiles Heuristic :

This heuristic focuses on the 4 corners of the grid and their 2 neighbouring tiles, each. This heuristic does not work if the corner or its neighbours have a blank space in the current state or the goal state. It's easy to notice that if we have to correctly position any tile which is not a neighbour of a blank space, from its incorrect position, any of the two neighbours must move aside, one for entry of blank space and another for the exit. We can't use the same neighbour for both entry and exit because in that case, we would reach the same state again (E.g. Up-down, Right-left) .

15	2	1	12
8	5	6	11
4	9	10	7
3	14	13	

9	6	11	13
14	10	8	5
4	15	1	2
12	3	7	

Suppose the first one is the current state and the second is the goal state. Focus on the bottom left corner. Tile 4 is currently in its goal location. We need to position tile 3 elsewhere and get 12 in the corner. Can we do it without moving tile 4? No! To change the value in the corner, the desired value (12, in this case) needs to be brought to one of the neighbours of the corner. Since we don't want to move tile 4, with a certain sequence of operations, let's bring tile 12 in place of tile 14. For the swap to take place, blank space has to be brought to the corner. Since tile 12 can't be moved anymore, the only way is force tile 4 to move aside. Afterwards, we need another operation to restore tile 4 back to its position. These are the 2 moves that are not counted in Manhattan distance.

However, care must be taken when we are using this heuristic along with the linear conflict heuristic. In the above case, if tile 4 was in column conflict with some tile, these 2 extra moves would be counted twice, affecting the admissibility of the heuristic.

Here is the step by step procedure to calculate the heuristic for every corner :

1. Check if there is a blank space in the corner under consideration or its neighbours in the current state or the goal state.
2. Check if the corner tile is in its goal position.
3. If answer to (1) or (2) is yes, return 0
4. Check if the first neighbour of the corner tile is in its goal position.
5. If no, jump to step 9.
6. Check if the neighbour under consideration is under any linear conflict.
7. If yes, jump to step 9.
8. Add 2 moves to the cost.
9. Repeat steps from (4) to (8) for the second neighbour.

As you can see, all these steps are cheap. The most expensive step is checking if the neighbouring tile is in linear conflict which can be done $O(N+M)$. Since this procedure has to be carried out for every corner, the time complexity is $O(4*(N+M))=O(N+M)$. Similar to linear conflict heuristic, the space complexity is $O(N)$ or $O(M)$.

Pattern Database Heuristic :

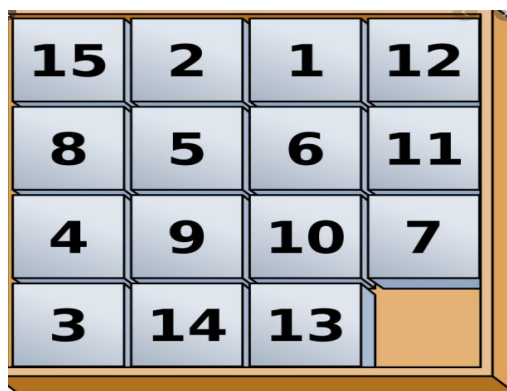
This heuristic is totally different from the heuristics discussed above. It can't be used in alliance with any of the heuristics discussed above. It has to be used independently. The idea behind this heuristic, however, comes from the Manhattan distance heuristic.

As we already discussed, the idea behind Manhattan distance heuristic is considering every tile as an independent entity and assuming blank spaces everywhere else and summing up the number of moves it takes to reach its goal position (which are equal to Manhattan distance between current and goal position.). What if we extend this idea to a group of tiles? We can assume blank spaces everywhere except the group of tiles under consideration and then instead of Manhattan distance, here we need to compute the number of swaps needed to bring all the tiles in the group to their goal positions.

So, we have to divide the tiles in groups so that every tile (excluding blank space) belongs to exactly one group and then summation of cost for every group would give the heuristic value. Please note that the tiles in the group don't necessarily have to be adjacent.

To compute the cost, we have to apply BFS from the goal state configuration of the group of tiles under consideration. The successor function would be swapping any tile in the group with the blank spaces (any tile which is not in the group is considered as blank space). This way, we can store the cost for all the states of the group of tiles.

If we divide $K=N*M-1$ tiles in G groups, of size $S = K/G$ (or $K/G + 1$, if K is not divisible by G), the memory required for this operation would be $O(G*(N*M)^S)$. Considering the memory constraints, the best choice of G is 3. With $G=3$, for 15-puzzle (the largest in this category), the memory required would be $O(3*16^5)$, which is available to us. The vector which stores the costs for each state of these groups of tiles is called the pattern database.



15	2	1	12
8	5	6	11
4	9	10	7
3	14	13	

Consider the above 15-puzzle. We divide it in 3 groups of size 5 : tiles 1-5, tiles 6-10 and tiles 11-15. The position of tile 1 is 2, that of tile 3 is 12 and so on. So, the state for group 1 is (2 1 12 8 5), that for group 2 is (6 11 4 9 10) and the state for group 3 is (7 3 14 13 0).

The successor function requires $O(b*S)$ time to compute all the successors ($b=4$). So, the time complexity of computing the pattern database is $O(G*(N*M)^{S*b*S})$.

In case of 15 puzzle, this works out as : $3 \cdot 16^5 \cdot 4 \cdot 5 \sim 6 \cdot 10^7$. Moreover, the constant factor is high due to hashing of states to save the memory. E.g. A state (3 4 11 8 2) is hashed to its hexadecimal equivalent i.e. $3 \cdot 16^4 + 4 \cdot 16^3 + 11 \cdot 16^2 + 8 \cdot 16 + 2$. However, my implementation computes the pattern database in under 3 seconds.

Here is the step by step procedure to compute the heuristic value:

1. Find the configuration of tiles in each group in the current state.
2. Compute the hash of positions of tiles in each group.
3. Look up their cost in the pattern database.
4. Take a summation over the costs of all the groups.

This procedure requires $O(N \cdot M)$ time and constant space, since everything is stored in the pattern database.

1. A* search with Manhattan + Linear Conflict + Corner Tiles Heuristic:

The most commonly used informed search algorithm is A* search. But when I ran A* search with Manhattan distance + linear conflict + corner tiles heuristic on randomly generated cases of 15-puzzle, on most of them, it didn't give any output till more than 5 minutes (then I had to force stop the code since the laptop started to hang). The reason is, to reach any node in A* search, all the cheaper nodes must be traversed first. But most of the times, goal states are buried deep in the search graph so in order to reach them, it's compulsory for A* to traverse on all the cheaper nodes. This shortcoming of A* search is overcome by Iterative Deepening A* search or IDA* search.

Clubbing the time complexities of all these heuristics as discussed above, the theoretical time complexity of this algorithm works out as $O((N \cdot M)! \cdot N \cdot M \cdot (N + M))$. Please note that it does not actually visit so many nodes, it's just a theoretical time complexity!! The space complexity is $O(N \cdot M \cdot V)$, where V is the number of visited nodes. There's no good theoretical upper bound on the value of V . But V goes on increasing as the algorithm runs for more and more time. This algorithm is not good for memory constraints either.

2. IDA* search with Manhattan + Linear Conflict + Corner Tiles Heuristic:

In IDA*, we repeatedly apply cost-limited DFS starting from the heuristic value of the start state. After every DFS, the new cost bound is set to the minimum of the costs of boundary nodes which were pruned by the last DFS (because their costs exceeded the bound). Needless to say, when we reach the goal state, we terminate the algorithm and return the path. The path given by IDA* is guaranteed to be optimal because if there existed a shorter path to the goal, it would have been reached in any of the previous DFSs.

When I ran IDA* search with these heuristics, it took around 3 minutes on average for the randomly generated instances of 15-puzzle but the running time still blew up above 5 minutes for the worst cases.

The theoretical time complexity of this algorithm is exactly the same as A* but in practice, it runs faster due to the reasons mentioned above. The space complexity of the algorithm, however, is extremely better than A* due to DFS. The only things to be stored are the current state and its path from the root. So, the space complexity is $O(d)$, where d is the maximum depth of the graph. In case of a 15-puzzle, it does not exceed 100 so it's safe to conclude that the space complexity is $O(1)$.

3. IDA* search with Pattern Database Heuristic :

I have already explained in detail how the pattern databases are computed and the heuristic values are calculated using them. As I already explained, with optimised implementation, the pattern database was computed in under 3 seconds. IDA* search with pattern database heuristics gave better results than the previous one. After running around 30 randomly generated instances of 15-puzzle, the average running time was found to be around 40-45 seconds, much better than the previous one. Although in the absolute worst case of 15-puzzle, the running time was found to go up to 5 minutes.

The theoretical time complexity of the algorithm is $O((N*M)! * N*M)$, whereas the space complexity is the same as that of pattern database i.e. $O(G*(N*M)^S)$, where G : the number of groups and S : the size of the largest group.

Novel Modification to Linear Conflict Heuristic :

As we can see, the algorithms discussed above give the optimal solution for the puzzle. But even the best of them takes 1 minute on average and 5 minutes in the worst case of 15-puzzle. Can we make some modification in any of the above algorithms which would speed it up considerably at the expense of its optimality so that the utility gained with the speed is more than the utility lost due to sub-optimality? The answer is yes.

When I observed the heuristic values computed by Manhattan + linear conflict + corner tiles heuristic, I found that they were much smaller than the actual cost. But there is no any known admissible and consistent heuristic that fills this gap, which in turn would improve the performance of IDA* even further. Working out such heuristic might be a long-term and extremely ambitious research project.

What if we made the heuristic theoretically non-admissible but in most of the states, the overestimation caused by non-admissibility actually took the heuristic value closer to the actual value because the original heuristic value was way too underestimated ? That does not sound so bad. I exploited this observation.

As we already discussed in detail, linear conflicts occur when two tiles are in their goal row or column but their relative positions are reversed. As an experiment, I relaxed the condition that both tiles should be in their goal row or column. This heuristic, after modification, should not be called “linear conflict heuristic” anymore because the conflicts are not “linear” anymore due to relaxation of the condition of goal row/column. Let’s just call it “novel heuristic” hereafter. After the relaxation, I compared the new heuristic values with the actual values (computed using IDA* with pattern database heuristic) on randomly generated 30 states. The heuristic value was not overestimated even once. However, the costs for all these states were decently high (above 20). There would be more probability of overestimation for states nearer to the goal state. So, I ran a BFS from the goal state till the number of states reached 10^5 , stored their actual costs and compared them with their new heuristic value. 25% of the states at a depth less than 5 were found to be overestimated, whereas in total, 2780 states were overestimated (< 3%) and the rest of the states were only brought closer to the actual value.

It’s important to note that the heuristic proposed above is non-admissible and inconsistent and can be safely called an engineering-based heuristic. However, the statistics stated above don’t seem to be bad.

4. IDA* search with Manhattan + Novel + Corner Tile Heuristic :

IDA* search with these heuristics gave extremely fast results. I ran the algorithm against 30 randomly generated instances of 15-puzzle. The average running time of the algorithm was found to be 1.5 seconds and the worst running time recorded was 4 seconds. Clearly, this algorithm is faster than IDA* with pattern database heuristic by many folds but what about optimality ?

I computed the optimal solutions of those instances of 15-puzzles with pattern database heuristic to compare the optimality. The average loss of optimality was found to be around 12 moves. The worst solution, among them all, had 22 extra moves. If we consider the trade-off of increased speed against the loss of optimality, this modification can come handy when we want quick results.

The theoretical time complexity and space complexity of this algorithm is exactly the same as IDA* with Manhattan+linear conflict+corner-tiles heuristic.

Recommended Algorithms :

1. When optimality is the priority and we have a time frame of at least 6 minutes, I highly recommend IDA* search with pattern database heuristic. This algorithm is guaranteed to find the optimal solution in that time frame.
2. When optimality can be compromised for high speed i.e. decently suboptimal but quick solutions are welcome, I would recommend IDA* search with Manhattan distance + novel + corner tiles heuristic algorithm.

16 < N x M ≤ 36 :

As already stated, the known most efficient algorithm takes hours to solve an average case of 24-puzzle. It took 3 months to solve the absolute worst case of 24-puzzle, expanding around 8×10^{12} nodes. So, my main focus for this category of puzzles was to find a decently suboptimal solution in a decent time frame.

1. Simulated Annealing :

Generally, local search techniques are used to solve CSPs because we don't care about the path to the solution. But in this case, I anyway was not expecting the optimal solution so I decided to try local search techniques. The most commonly used local search technique is hill-climbing with random restart. But if we closely observe the states in this problem, it's not hard to notice there are a lot of states whose heuristic values are less but the actual cost to reach the goal is way more. In other words, there are a lot of local optima so there's a very high chance that hill climbing would be stuck in local optima. So, we need a local search technique that would allow us to escape the local optima. This is where simulated annealing comes off as a good candidate.

We can't afford fancy heuristics like linear conflict in this case because it would slow down the algorithm considerably. Euclidean distance heuristic was found to give comparatively better results than other heuristics. Euclidean distance heuristic was successful in finding some paths Manhattan distance heuristic failed to find so I finalised the use of Euclidean distance as a heuristic.

We don't have enough computation power to implement the actual simulated annealing algorithm where T is initialised to $-\infty$ and gradually decreased to zero. This algorithm is complete and guaranteed to find the solution. But it is extremely slow. So, we instead compute the value of T as $R * (2 - X)$, R : random real number in $(0, 1]$ and X is gradually increased from -2 to 2. As X approaches 2, T approaches 0 and the state becomes stable. The variant we have implemented is not complete and can possibly fail to find the path in some cases.

Here is the step by step description of the algorithm:

1. Start the simulated annealing from the start state of the puzzle.
2. Initialise X to -2, path as a string to store the path.
3. If the current state is a goal state, return path
4. If X is equal to 2, return -1 (failed)
5. Randomly select a successor of the current state.
6. If the heuristic value of the successor is less than the current state, follow step (6).
7. Go to the successor and append the respective direction to path.
8. Else follow steps from (8) - (10).

9. Compute the value of T as $R * (2 - X)$, R : random real number in $(0, 1]$.
10. Compute the probability P as $e^{-dh/T}$ where $dh = h(\text{successor}) - h(\text{current})$.
11. With probability P , follow step (5).
12. Increment X by 0.000001.
13. Repeat step (4)-(11).

The results of this algorithm were disappointing in general. The average running time for the puzzles was around 1 minute. It solved almost all the instances of 24-puzzle that I tried. But it failed on some cases of 35-puzzle. The main concern was pathetic optimality. Since it is a local search technique, the path was not even close to optimal. The average solution cost was found to be above 10^6 . We have to do better than this.

The time complexity of this algorithm is $O(I*b*N*M)$, where b : branching factor = 4 and I : number of iterations = 4×10^6 . The constant factor is also high due to randomisation. The space complexity is $O(I)$, to store the path in a string.

2. Greedy Search :

This is an informed search technique that is seldom used because it does not guarantee to give the optimal solution. But we might use it for bigger puzzles since optimality is not our concern although we want it to be decently suboptimal, unlike in the case of simulated annealing. Greedy search is exactly the same as A^* search except for the cost function. In A^* search, cost function is the sum of the actual cost of the state from the start state and the expected cost of the state to the goal computed with the help of heuristics. But in greedy search, we only consider the heuristic value in cost. In other words, we greedily go on choosing the neighbouring state that is nearest to the goal according to heuristic values.

It's easy to notice that greedy search goes directly deep in the graph looking for the goal state without the guarantee that we are on the optimal path. Since we only consider the cost to the goal, there's very little deviation from the path to the goal. So, greedy search visits very less nodes as compared to A^* search or IDA* search for that matter. So, it's much faster.

It's impossible to make use of pattern database heuristic for these huge puzzles because of memory constraints. So, the choice of heuristic was Manhattan + novel + corner tiles heuristic.

The algorithm solved all the instances of 24-puzzle that I tried, with an average running time of 1.5 seconds. The average solution cost was found to be 400. In the case of a 24-puzzle, the cost of optimal solutions does not exceed 125. So, the solution provided is 3 to 4 times as costly as the optimal solution. But as you can see, the greedy search has improved the optimality of the solution by many folds, as compared to the simulated annealing algorithm.

For 35-puzzle as well, the algorithm solved all the tried instances with an average running time of 15 seconds. In the worst case, the algorithm took around 55 seconds to compute the solution. The average solution cost was found to be 1000, the cost of the worst solution

being around 2300. Although the costs seem to be expensive, they have improved significantly from the simulated annealing. Solving the 35-puzzle problem optimally is an unsolvable problem right now considering the computation power of computers. So, considering that, the quick suboptimal solutions provided by the greedy search under a tiny time frame might come handy.

The theoretical time complexity of this algorithm is exactly the same as A* search, $O((N*M)! * N*M * (N+M))$ but in practice, it visits very few nodes as already explained above. The space complexity is $O(V)$, where V is the number of visited nodes.

Recommended Algorithm :

For this category of puzzles, since it is impossible to find the optimal solution even in a time frame of 1 hour, we can only give suboptimal solutions. For that, I would recommend the greedy search algorithm with Manhattan + novel + corner tiles heuristic.

Summary and Conclusions :

To summarize, solving puzzles with $N*M \leq 9$ is very trivial with A* search with Manhattan distance heuristic or even BFS for that matter. For puzzles with $9 < N*M \leq 16$, there are a couple of ways to obtain the optimal solution with IDA* search, the faster of them needs a time frame of 5 minutes to guaranteedly provide the optimal solution. As of today, there is no known algorithm that can optimally solve bigger puzzles (24-puzzle, 35-puzzle or so) even in a time frame of 1 hour. For this category of puzzles, making use of greedy search with Manhattan+novel+corner-tile heuristic, we can obtain decently suboptimal solutions with average costs ranging from 400 to 1000, depending on the size of a puzzle.

Solving this problem is feasible in terms of running time or computation power or memory, if optimality is not the priority. In case, it is the priority, we can still solve 15-puzzle or smaller puzzles with feasibility. It becomes harder and harder and finally impossible, after that. The hard part about the problem was huge state spaces (except for 8-puzzle). Handling such huge state spaces, obtaining optimal solutions wherever possible with the help of different complicated heuristics and settling for suboptimal solutions with minimum loss of optimality when speed is the priority or optimality is unattainable with available resources, required insights about the sliding tiles puzzle.

For the second category of puzzles, as already mentioned, the optimal solutions can be guaranteedly obtained in a time frame of 5 minutes. To speed up the algorithm, I suggested a novel modification to the linear conflict heuristic which made it theoretically non-admissible but is guaranteed to give a suboptimal solution with loss of optimality of 12 moves on average, in a tiny time frame of 4 seconds. The future scope of the project is filling the gap between the original admissible heuristic cost and the actual cost with an admissible heuristic, instead of using a non-admissible modification. This way, the optimal solution would be guaranteed in a

tiny time frame. However, that would be a serious, long-term and extremely ambitious research project with no guarantee of success.

References :

1. Traditional linear conflict heuristic and corner-tiles heuristic :
<https://courses.cs.washington.edu/courses/csep573/10wi/korf96.pdf>
2. Pattern database heuristic :
<https://arxiv.org/pdf/1107.0050.pdf>
3. Simulated annealing :
https://kcir.pwr.edu.pl/~witold/aiarr/2009_projekty/35Puzzle/
4. IDA* search :
https://en.wikipedia.org/wiki/Iterative_deepening_A*
5. Solvability of NXM puzzle :
<https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html>