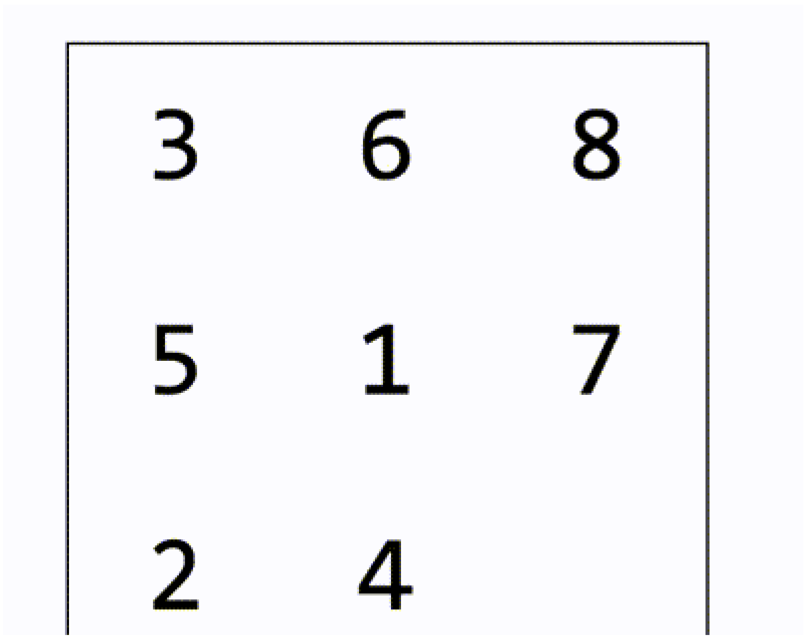
Solve the Slide Puzzle with Hill Climbing Search Algorithm



Hill climbing search algorithm is one of the simplest algorithms which falls under local search and optimization techniques. Here's how it's defined in 'An Introduction to Machine Learning' book by Miroslav Kubat:

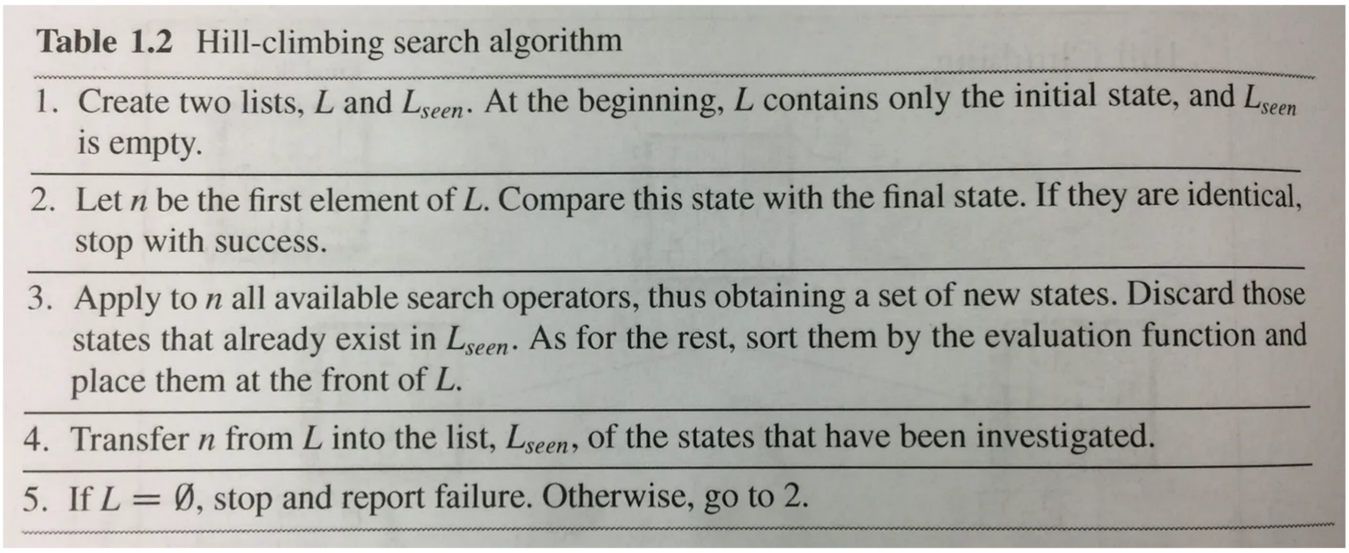


Table 1.2 Hill-climbing search algorithm

1. Create two lists, L and L_{seen}. At the beginning, L contains only the initial state, and L_{seen} is empty.
2. Let n be the first element of L. Compare this state with the final state. If they are identical, stop with success.
3. Apply to n all available search operators, thus obtaining a set of new states. Discard those states that already exist in L_{seen}. As for the rest, sort them by the evaluation function and place them at the front of L.
4. Transfer n from L into the list, L_{seen}, of the states that have been investigated.
5. If $L = \emptyset$, stop and report failure. Otherwise, go to 2.

Hill Climbing Algorithm Steps

Evaluation function at step 3 calculates the distance of the current state from the final state. So in case of 3x3 Slide Puzzle we have:

```
Final State:
1 2 3
4 5 6
7 8

Consider Current State:
1 2 3
4 5 6
7   8
```

Evaluation Function dF calculates the sum of the moves required for each tile to reach its final state. Since tiles 1 to 7 are already in its correct position, they don’t need to be moved. However, tile 8 is 1 move away from its final position.

```
dF(8) = m(1)+m(2)+m(3)+m(4)+m(5)+m(6)+m(7)+m(8)
= 1
```

Hill climbing evaluates the possible next moves and picks the one which has the least distance. It also checks if the new state after the move was already observed. If true, then it skips the move and picks the next best move. As the vacant tile can only be filled by its neighbors, Hill climbing sometimes gets locked and couldn’t find any solution. It’s one of the major drawbacks of this algorithm.

Another drawback which is highly documented is *local optima*. The algorithm decides the next move(state) based on immediate distance(cost), assuming that the small improvement now is the best way to reach the final state. However, the path chosen may lead to higher cost(more steps) later.

> *Analogues to entering a valley after climbing a small hill.*

In order to get around the local optima, I propose the usage of depth-first approach.

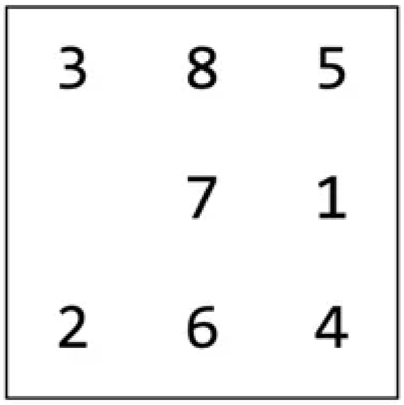
Hill Climbing with the depth-first approach

Idea is to traverse a path for a defined number of steps(depth) to confirm that it's the best move.

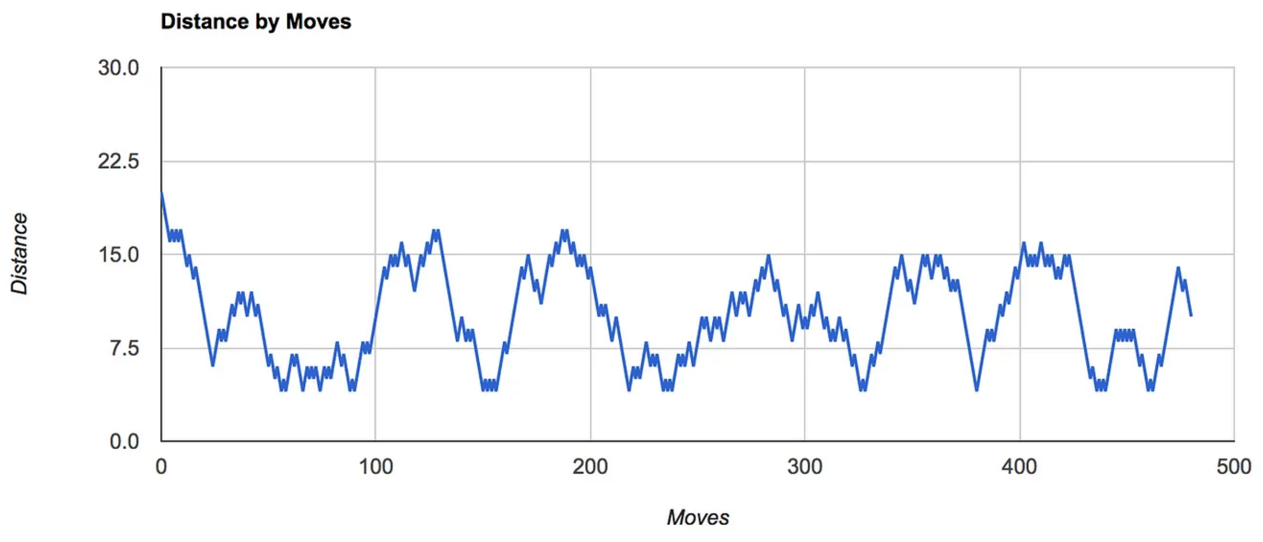
1. Loop over all the possible next moves(states) for the current state.
2. Call step 1 until depth d is reached. This generates a tree of height d.
3. Pick the move(state) with minimum cost(dF)
4. Return dF so that evaluation can be done at `depth-1` level.

I observed that the depth-first approach improves the overall efficiency of reaching the final state. However, its memory intensive, proportional to the depth value used. This is because the system has to keep track of future states as per the depth used.

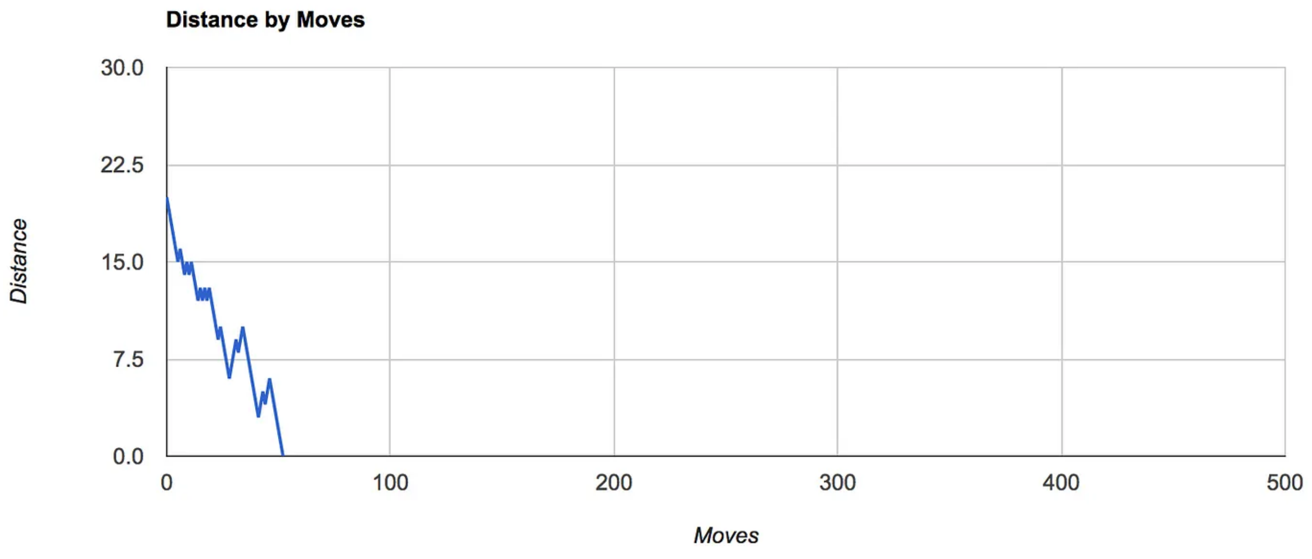
Here's a brief comparison of the Distance by Moves charts for various depths:



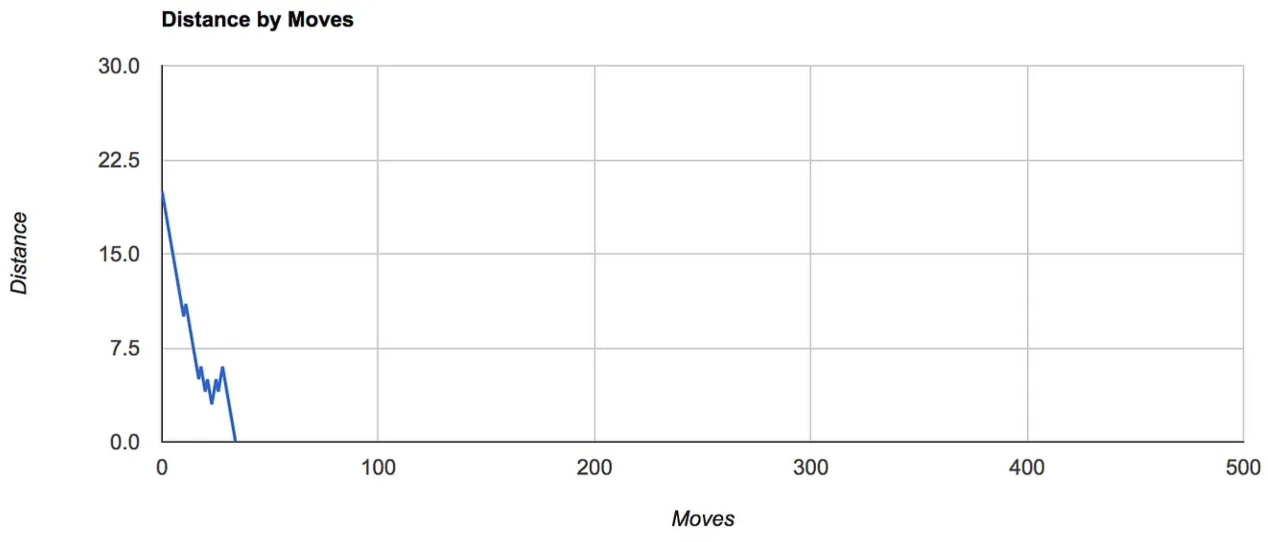
Puzzle Initial State



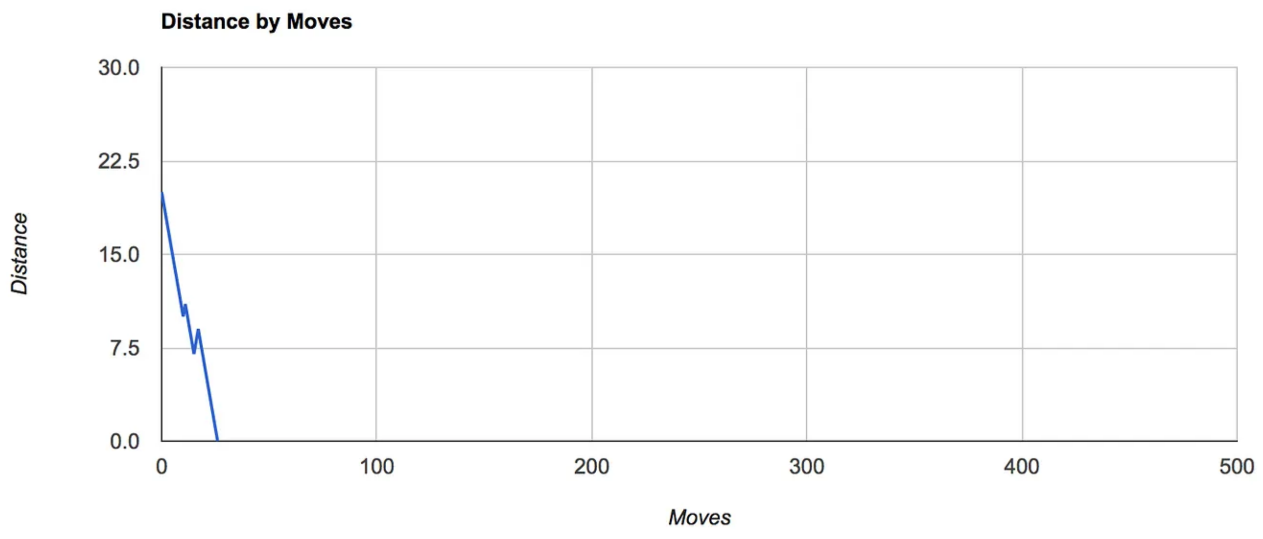
Depth=0, Puzzle Unresolved even after 480 steps



Depth=1, Puzzle Unresolved in 48 Steps



Depth=3, Puzzle Unresolved in 34Steps (Note fewer hills)



Depth=6, Puzzle Unresolved in 26 Steps (Even fewer hills)