Comparing search algorithms for solving the sliding-block puzzle

Nishant Nayak
nn343@cornell.edu

Alan Huang
ah2294@cornell.edu

***Abstract*—We explore the speed and optimality of algorithms to solve the sliding-block puzzle game. We compare A*, breadth-first search, greedy best-first search, and a custom human-based algorithm.**

I. Introduction

In the 1870's, a puzzlemaker named Sam Loyd offered a prize of $1000 to anyone who could solve an arrangement he created for the 15-puzzle game, which is played as follows. We are given 15 numbered tiles randomly arranged in a 4x4 grid. We repeatedly slide an adjacent numbered tile to the blank location until we reach the target configuration, as shown below [1].

1	2	3
4	5	6
7	8	

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

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

Fig. 1: Target configuration of 8-puzzle, 15-puzzle, and 24-puzzle.

Loyd's challenge made the 15-puzzle very popular in the United States and Europe as thousands of people tried to solve his configuration. Unfortunately, or fortunately in Sam Loyd's case, only certain configurations of the 4×4 puzzle are solvable. Loyd's configuration was not [2].

The 8-puzzle, 15-puzzle, and 24-puzzle belong to the family of sliding-block puzzles, which are often used as test problems for new search algorithms in artificial intelligence. The 8-puzzle has 181,440 (9!/2) reachable board states and can be solved optimally almost instantly, the 15-puzzle has 1.3 trillion (16!/2) states and can be solved optimally in a few milliseconds, but the 24-puzzle has approximately 10^{25} states (25!/2) and can take several hours to solve optimally [3]. We define solving optimally as solving the puzzle with the fewest moves possible. As the board size increases, the amount of possible board states exponentially increases, and finding the optimal solution becomes more and more difficult.

We were intrigued by this game, so we decided to explore multiple algorithms which could solve the game, some optimally, some not. A* and breadth-first search were two algorithms that solve the puzzle optimally. Greedy best-first search and a human-based algorithm were two algorithms that ran much faster, but could not find solve the puzzle optimally.

II. Concepts

A. Translating the puzzle to a graph search problem

The set of all board states can be conceptualized as an immense, connected graph where each node has two, three, or four edges, reflecting the fact that any board state has two, three, or four legal moves. For example, we have node n_x that is associated with the board state x. Suppose we slide an adjacent tile left into the blank space on the board state x, and we obtain a new board state y. In our graph, there is an edge(n_x, n_y) that connects the board states to each other. Our original problem of finding the moves that convert our start state s to our goal state g (as shown in Figure 1) can be phrased as finding the shortest path of edges than connect n_s to n_g.

Now that we've phrased the problem in terms of a graph search problem, the next question which we must ask is, is this graph connected? In other words, is it possible to get from any board state to any other board state using legal moves. After all, it would be pointless trying to solve an unsolvable task. We explore this question in the next section.

B. Solvability of random start states

To check for solvability we consider the property of permutations.

We view the board state as a permutation of the numbers 1 to 16 listed row by row, where the blank space is 16 (we fix $n = 4$ for simplicity). We define an "inversion" as the swap of any two numbers. For example, 1,2,3,4 differs from 1,2,4,3 by 1 inversion. Given any 2 permutations a and b, b is reachable from a through either an odd number of inversions or an even number of inversions, but not by both. For example, 1,2,3,4 can turn into 1,2,4,3 using 1 inversion (3 with 4) or 3 inversions (2 with 4, 2 with 3, 2 with 4). No even number of swaps can get from 1,2,3,4 to 1,2,4,3. As such, we will call the distance from 1,2,3,4 to 1,2,4,3 *odd*.

Next we notice that each "move" in the puzzle consists of an inversion (or swap) involving the 16 (blank) piece. Let us assume the 16 piece is in it's proper place in the initial state. We notice that only an even number of moves would allow the blank piece to remain put (the number of up moves must equal the number of down moves, the number of right moves must equal the number of left moves). Thus, any configuration of the board with an *odd* distance from the goal is not solvable.

Now, we generalize for any positioning of the blank piece. For any positioning of the blank piece, we can count the

number of moves necessary to reach it's destination (the bottom right tile). As c_1 = right moves (r) − left moves (l) and c_2 = down moves (d) − up moves (u) must both be constant, we see that

$$r + l + u + d = c_1 + 2l + c_2 + 2d \equiv (c_1 + c_2) \mod 2$$

In other words, the parity of moves, or swaps, to the solution is constant. If the parity is odd, even distance board states (fixing the blank space) cannot be solved. If the parity is even, odd distance board states cannot be solved. Thus, every board position can be solved if and only if the distance from that board position to the goal matches up with the parity of the number of moves needed to move the blank space to the goal.

C. Heuristics

Heuristics can be used to compute an estimated cost of the cheapest path from the state at a certain node to a goal state. They can be used to find a more informed search strategy. A heuristic function is said to be admissible if it never overestimates the cost of reaching the goal [3]. An admissible heuristic can be used to find the optimal solution of the sliding-block puzzle. We discuss the Manhattan distance heuristic and the Hamming distance heuristic.

- The Manhattan distance between the current location of a tile (x_c, y_c) and the goal location (x_f, y_f) is defined as:

  $$|x_c - x_f| + |y_c - y_f|$$

  We calculate the Manhattan distance of each tile and sum over all the values to obtain the heuristic value of the board state. The Manhattan distance is admissible because we must move each tile at least its Manhattan distance to reach the goal state.
- The Hamming distance of a board state is calculated by comparing the board state to the goal state. If the tile is not in the correct position we add 1 to our Hamming distance total, otherwise we add 0. Since every tile must move at least one time to reach its goal state, the Hamming distance is admissible.

Since the Manhattan distance gives a more accurate approximation of the moves needed to reach the goal state, we use it as our heuristic function in our greedy best-first search, A* search, and human-based algorithm.

III. Search algorithms

For this puzzle, we explore 4 algorithms: Breadth-first search, greedy best-first search, A* search, and a custom human-based algorithm. The first three algorithms are similar in that they do a direct search from the start state (s) to goal state (g). Specifically, this means each of these algorithms begins on the start node in the search graph and expands nodes in the graph. The algorithm terminates once the goal node has been found. The difference between the three algorithms lies in the method they use to determine which nodes to expand next. In the following pseudocode and sections after, we will define $f(node) = d(node) + h(node)$ where d is the depth and h is the heuristic. The priority queue (implemented as a min heap) is structured by f value.

Algorithm 1 Search algorithm for sliding-block puzzle

```
node ← node with state = initial board state, parent = none
f(node) ← depth + heuristic value of initial board state
frontier ← min heap with node as the only element
explored ← an empty set
frontier.insert(node, f(node))
loop
  if goal(node) then
    return path from start state to goal state
  node ← frontier.pop(), pop based off of node with f value f(node)
  add node state to explored
  for each possible move do
    child ← node with state = board state after new move, parent = node
    f(child) ← f value of child board state
    if child state is not in explored or frontier then
      frontier.insert(child, f(child))
```

A. Breadth-first search (BFS)

Breadth-first search is the simplest search algorithm. It decides which nodes to expand purely based on their depth in the tree. For example, all nodes of depth 3 are expanded before any nodes of depth 4. Thus, $h(node) = 0$, so $f(node) = d(node)$ where d denotes the depth (i.e the number of moves from the start node to $node$. The root node is expanded first, then all its children, then all their children, and so on.

B. Greedy best-first search

An informed search strategy where the choice of node expanded depends purely on the heuristic value. Thus, we can artificially set the depth of all nodes to 0, so $f(node) = h(node)$. The node in the tree with the lowest distance is always explored first.

C. A* search

An informed search strategy which combines the heuristic and the depth to decide which node to explore next. It uses the full formula given in the start of the section:

$$f(node) = d(node) + h(node)$$

If we choose an admissible heuristic for the A* search, the solution is always optimal (proof in next section). Since A* carefully chooses which node to expand next, its performance far exceeds that of BFS while maintaning optimality. A* search, on average, explores 25 times fewer nodes than the breadth-first search [4].

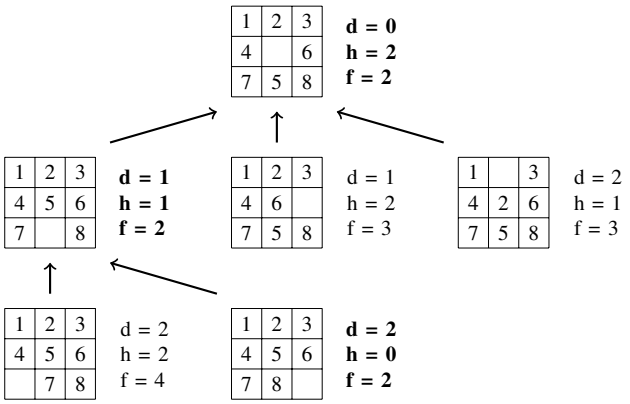

Fig. 2: Example of the A* search. Arrows connect child to parent nodes.

D. Human-based algorithm

Our motivating question was, how would a human solve the puzzle? The solution which the previous three algorithms gave certainly did not resemble how people think – many times the solution would seemingly randomly shuffle pieces around and would "magically" find the goal state in a few, careful last few moves.

We predicted that the average person would solve the puzzle row by row, solving 1 to 4, then 5 to 8. Someone with more experience might solve the top row and the right-most column, turning an $n \times n$ board into an $n-1 \times n-1$ board, and continuing recursively. We ended up creating an algorithm that could do either.

Specifically, our algorithm took as its input, a list of intermediate goal states. For example, the first row could be seen as the first goal state and the first column could be the second goal state. The final goal state would be the solved puzzle. The high-level pseudocode of this idea is given below.

Algorithm 2 Simple pseudocode for human-based algorithm

```
n, where we are trying to solve an n x n puzzle
while n >= 3 do
  if n = 3 then
    solve 8-puzzle with greedy algorithm
    return path from start state to goal state
  else
    use greedy algorithm to put the first row and first column in place
    set n = n-1 to solve the remaining rows and columns
```

This algorithm differs from the other search algorithms in its heuristic value and in the way it checks the goal state. While the other search algorithms have only one goal state, the human-based algorithm has multiple, intermediate goal states that get updated as tiles end up in their desired locations. The heuristic for the other search algorithms is the Manhattan distance from the current node to the goal node, but the heuristic for the human-based algorithm is the distance from the current node to the current goal state that it is trying to reach. This ensures that the problem is restricted, and solves one goal state at a time. By reducing the goal state and creating simpler sub-problems, we can solve large sliding-block puzzles faster than the other search algorithms.

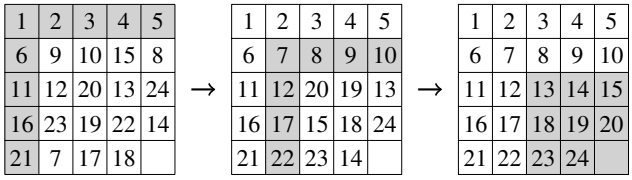

Fig. 3: How the human-based algorithm proceeds to solves the puzzle.

We can make the human-based algorithm follow the same structure as the other search algorithms by adding a few important but minor changes shown below.

Algorithm 3 Human-based algorithm for sliding-block puzzle

```
node ← node with state = initial board state, parent = none
f(node) ← depth + heuristic value of initial board state
frontier ← min heap with node as the only element
explored ← an empty set
frontier.insert(node, f(node))
loop
  if intermediate_goal(node) then
    if intermediate_goal(node) = goal(node) then
      return path from start state to goal state
    else
      intermediate_goal.update()
      node ← node with state = current board state, parent = previous node
      f(node) ← f value of current board state
      frontier ← min heap with node as the only element
      frontier.insert(node, f(node))
  node ← frontier.pop(), pop based off node with best heuristic function value f(node)
  add node state to explored
  for each possible move do
    child ← node with state = board state after new move, parent = node
    f(child) ← heuristic value of child board state
    if child state is not in explored or frontier then
      frontier.insert(child, f(child))
```

IV. Optimality

We define the optimal solution as the solution which uses the fewest moves possible. If multiple such solutions exist, all of them are considered optimal. BFS and A* solve the puzzle optimally, while greedy best-first search and the human-based algorithm do not.

A. BFS is optimal

Proof: Let p be the solution path returned by BFS having distance d from start node n_s to goal node n_g. Assume there

exists a solution path p' with distance $d' < d$ from n_s to n_g. By definition of BFS, all paths of distance d' are explored before paths of distance d and BFS would have found the solution path p'. Thus, p' cannot exist.

B. A* is optimal

Proof: Assume A* returns a solution with distance $dist$. Assume there exists a better solution $(n_0, n_1, ..., n_{dist'})$ with distance $dist' < dist$. Since the A* solution did not return the better solution, there must exist some node in that solution which was added to the search tree, but not expanded by A*. Call this node n_i where $i < dist'$. The depth of n_i is i and the distance of n_i to the goal is at most $dist' - i$ (Since the better solution already achieves that distance). By admissibility of our heuristic, $h(n_i) < dist' - i$. Then,

$$f(n_i) = d(n_i) + h(n_i) < i + (dist' - i) = dist' < dist$$

But, we see that the goal node in A* has a f value of

$$f(n_g) = d(n_g) + h(n_g) = dist$$

This is greater than $f(n_i)$! Thus, A* should have expanded n_i before it expanded its solution node. Thus, there cannot exist a node along the supposed "better solution" not expanded by A*. By contradiction, A* is optimal.

C. Greedy best-first search is suboptimal

From our results, greedy best-first search took on average roughly 3 times more moves than A*, showing that it is suboptimal.

D. The human-based algorithm is suboptimal

From our results, the human-based algorithm search took on average roughly 1.5 times more moves than A*, showing that it is suboptimal.

V. Implementation

Our implementation was done in python and revolved around the node class. The node class consisted of:

- board state object
- a previous node pointer, which was used to keep track of the path after finding the goal state
- the heuristic integer, which could be chosen to be the Manhattan or the Hamming distance
- the depth integer, which was the depth of the tree

A. Search algorithms implementation

To begin, we randomized a starting state by populating a set with numbers from 0 to $n^2 - 1$ (0 represented the blank tile). Then, we randomly sampled and removed a number from the set to fill each tile of the board. We did a solvability check based on the positioning of the blank tile and the number of inversions, as explained in the solvability section. If it wasn't solvable, we inverted 2 adjacent nonzero tiles to add one more inversion, making it solvable.

We then initialized a min priority queue using python's heapq class, whose comparator was defined as a node's depth plus a node's heuristic. For BFS, we would set the heuristic to 0 so only the depth would be used. For greedy best-first search, we would set the depth to 0, so only the heuristic would be used. We then created a node, populated it's variables with the corresponding information and added it to the priority queue.

We ran a while loop which expanded the nodes and expanded the graph, or simply put, did the searching. In this while loop, we popped the node at the head of the priority queue. We added this node to our set of visited nodes and then expanded the node with a generate_children function. The function generated all the children of the node by checking which of the four directions the blank space can move and creating a new node for each legal swap of the blank space. For each child, we checked if its board state had been visited by comparing it our set of visited nodes. If it had not been visited we added it to the priority queue and also added it to the set of visited nodes.

After each pop of the priority queue, we check if the popped node was the goal state. If so, we ran a loop starting from the node of the goal state, and traversed along its previous pointer until we reached the starting state. We stored all the intermediate states, and when reversed, we obtained the solution path from the start node to the goal node.

B. Human-based algorithm implementation

The human-based algorithm took as input not only the initial board state, but also a sequence of intermediate goals to reach before reaching the end goal. For example, one might decide to try to get the 1 in place, then the 2, then the 3, and so on until the puzzle was completed. So our first step was defining the proper goals.

We decided that the easiest solution for a human to learn would be to solve the first row and first column and reducing the $n \times n$ puzzle to a $n - 1 \times n - 1$ puzzle. They would repeat this process until the entire board was solved. Our set of goals reflected this human process. The first goal was to put the tile 1 in place, then the tile 2, then 3, until the first row was completed. The next goals were $n + 1$, $2n + 1$, etc., until the first column was completed. These goal states were stored as a list of sets where the final set would contain every tile in its proper location, corresponding to the finished goal state.

To find a path from the current state to the intermediate goal we used the A* algorithm defined in the implementation above. However, instead of checking whether the popped node was the final goal state, we checked if it was the intermediate goal state. This was done by looping over all the tiles and making sure that the elements in the popped node were in the positions the intermediate goal state required. After the intermediate goal was obtained, a new A* search would begin with the starting node as the intermediate goal and the goal node as the next intermediate goal.

One additional change that was necessary to get the human-based algorithm to work properly was changing the heuristic function. In the human-based algorithm, the distance from a state to the final goal state was irrelevant. We wanted to know

how close that state was to the current intermediate goal state. Thus, the heuristic used for each iteration of the A* search was the combined Manhattan distance (from current position to desired position), restricted to the tile locations in the current intermediate goal state.

C. Visualizer

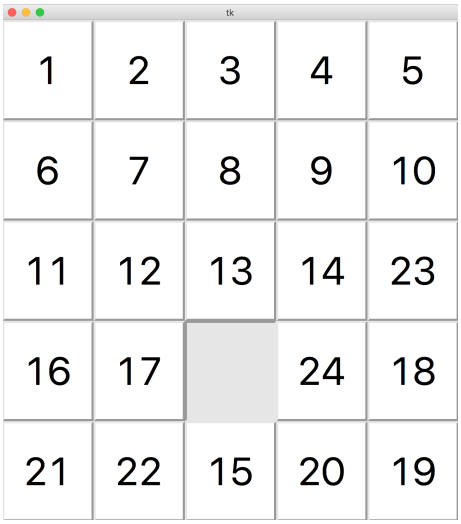

Fig. 4: Visualization of the 24-puzzle game using Python's tkinter library

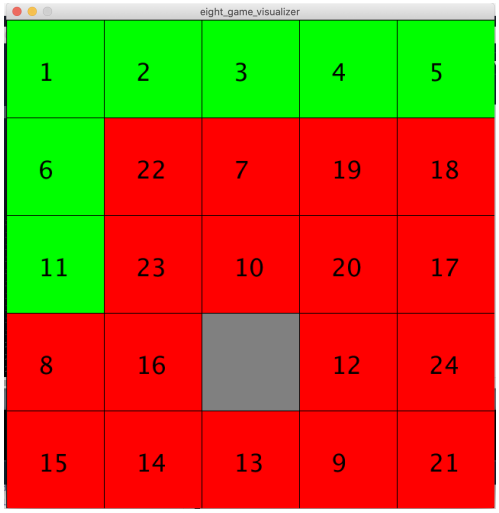

Fig. 5: Java visualizer made using the Processing library. Shows human-based algorithm solving row/column/row/column. Green shows the tiles already in place

VI. Performance

A. Setup

To measure the performance of the search algorithms we ran each algorithm on 100 randomized board states on the 8-puzzle, 15-puzzle, and 24-puzzle. We could only run the breadth-first search (BFS) when it was feasible. BFS was not feasible for the 15-puzzle or the 24-puzzle. The 15-puzzle can be solved between 0 to 80 moves [5], and if we assumed BFS solved the puzzle in 25 moves, had a branching factor of 3, and our program searched 1 millon nodes per second, the search algorithm would take 10 days.

$$\frac{3^{25} \text{ nodes explored}}{25 \text{ moves}} * \frac{1 \text{ second}}{1 \text{ million nodes}} * \frac{1 \text{ day}}{86400 \text{ seconds}} \approx 10 \text{ days}$$

Therefore, BFS was not feasible for the 15-puzzle or 24-puzzle. We could only run the A* algorithm on the 15-puzzle 10 times because of time and storage restraints. Though iterative-deeping A* can solve the 15-puzzle in reasonable time [6], A* cannot. In Google Colab (where we ran our experiments), A* search on the 15-puzzle explored over 18 million nodes and used 25 gigabytes of RAM when the estimated path was greater than 50 moves. The Colab notebook would shutdown before we were able to find the A* solution in these examples.

B. Results

8-puzzle (3x3 grid)			
	Average nodes expanded	*Average nodes explored*	*Moves to solution*
Breadth-first search	137,255	102,631	22
A* search	1,446	911	22
Greedy best-first search	420	243	64
Human-based search	478	288	30

15-puzzle (4x4 grid)			
	Average nodes expanded	*Average nodes explored*	*Moves to solution*
Breadth-first search	unfeasible	unfeasible	unfeasible
A* search	3,523,100	2,272,498	47
Greedy best-first search	5409	2594	259
Human-based search	40,935	19,308	81

24-puzzle (5x5 grid)			
	Average nodes expanded	*Average nodes explored*	*Moves to solution*
Breadth-first search	unfeasible	unfeasible	unfeasible
A* search	unfeasible	unfeasible	unfeasible
Greedy best-first search	38,780	17,043	618
Human-based search	860,126	355,714	189

In the 8-puzzle, the average optimal number of moves came out to 22 moves. The human-based and greedy best-first search had the best performance in terms of nodes expanded and explored, but could not find the optimal solution. The greedy best-first search performed three times worse than the A* search in terms of moves taken. The breadth-first search performed exponentially worse than the other algorithms when it came to nodes expanded and explored. From our results, for the 8-puzzle, the A* search algorithm seems like the best choice.

In the 15-puzzle the A* performed much worse when it came to nodes expanded and explored, exploring 87 times more nodes than the human-based search and 650 times more nodes than the greedy best-first search. Though the greedy best-first search explored far fewer nodes than the A* search or human-based search, it took 3 times more moves than the human-based search and 6 times more moves than the A* search to get to the solution. In the 15-puzzle, to optimize for time and space, the human-based search seems like the best option.

In the 24-puzzle, the BFS and A* search are unfeasible, so we could only compare the greedy best-first search and the human-based search. The human-based search found a solution with 3 times fewer moves than the greedy best-first search, but explored 20 times more nodes. In this scenario, if one was looking for a quick solution, the greedy best-first search would be a better option. If someone was looking for a closer to optimal, intuitive solution, the human-based search would be the better option.

VII. Discussion

A. Optimizations

As discussed in the previous sections, the exponential increase in the search space as board size increased proved to be more than the optimal algorithms could handle. We implemented an optimization by putting a weight on the heuristic. Specifically, we set a parameter $c > 1$ and let $f(v) = d(v) + c \cdot h(v)$. In this way, the A* algorithm more strongly preferred nodes that were closer to the goal state. We used this optimization with $c = 3$ for the human-based algorithm and it significantly sped up the runtime for the 5×5 board.

There were also many other possible optimizations we found specific to our implementation. Some examples are listed.

- Our visited set stored each node as a tuple (as lists are unhashable in python). This required an $O(n^2)$ (n being board size) conversion, and made the hashing take slightly longer. One way we could have optimized this was to convert the board state into an integer written in base n^2. For $n \leq 4$, this would fit into a 64-bit integer. We could then calculate each neighbor's "board hash" by looking at only the 2 numbers that were swapped. This would likely be faster than the O(n^2) conversion.
- Every node could store not only the board state, but the position of the blank tile as well. At the cost of more memory, this would allow us to expand the node (find its neighbors) without running an $O(n^2)$ loop over the board to look for the blank.
- When expanding a node, we could have calculated its children's heuristic value by looking at the parent's heuristic function as well as what changed from child to parent. This would get rid of the $O(n^2)$ runtime to calculate the heuristic.

Combining all three optimizations, it would be possible to have each iteration of the main while loop run in a few operations per child of the popped node (likely no more than 10 operations in total). From the original O(n^2) operations per iteration, this would likely be an improvement by a factor of 5 or more when $n = 4$. When $n = 5$, we would have an even bigger improvement, but might run into problems with converting the board into an integer.

B. Takeaways

Creating a human-based algorithm helped us better understand how a human would play the game and how a human could devise a plan to solve the sliding-block puzzle themselves, no matter what the size. It was amazing to see that a human could find the solution faster than a computer could! Though the human-based algorithm was sub-optimal, it was intuitive and could be used on boards where finding the optimal solution was unfeasible. It was interesting to compare the trade-offs between time, space, and optimality for the different search algorithms. Depending on one's task, different algorithms could be used. If one had no time or space restraints, A* would be the go-to solution. If all we cared for was a solution, the greedy best-first algorithm would explore the fewest number of nodes and return a solution extremely quickly. The search algorithms all inevitably led to the same, final goal state, but understanding when to use which algorithm based on our constraints could help us better understand when to use which algorithm for other graph problems.

REFERENCES

[1] I. Parberry, "A real-time algorithm for the $(n^2 - 1)$-puzzle," Information Processing Letters, vol. 56, no. 1, pp. 23–28, June 1995.

[2] A. F. Archer, "A modern treatment of the 15 puzzle," The American Mathematical Monthly, vol. 106, no. 9, pp. 793–799, 1999.

[3] S. J. Russell and P. Norving, "Solving problems by searching" in Artificial Intelligence: a Modern Approach, 3rd ed. New Jersey: Pearson, 2010, ch. 3, pp. 64-108.

[4] D. Kunkle, "Solving the 8 puzzle in a minimum number of move: An application of the A* algorithm," Introduction to Artificial Intelligence, 2001.

[5] R. E. Korf, "Linear-time disk-based implicit graph search," ACM vol. 55, no. 6, December 2008.

[6] B. Bauer, "The Manhattan pair distance heuristic for the 15-puzzle," Technical Report Paderborn Center for Parallel Computing, vol. 1, no. 94, January 1994.