School of Computer Science & Informatics

Cardiff University


# Solving N-Puzzle Problem with Uninformed and Heuristic Algorithms

*Bingjie Lu*

*Module: CMT403*

*Supervisor: Yukun Lai*

*Moderator: Paul L Rosin*

*October 2022*

# Abstract

The goal of the research is to use several algorithms to solve the n-puzzle problem. We intend to use a variety of methods to address this problem, including heuristic (Simulated Annealing, Iterative Depending A Star, Genetic Algorithm), uninformed (Breadth-First Search, Depth-First Search). The results of these algorithms will then be analysed and discussed.

# Contents

# 1. Introduction

The idea of the N-puzzle issue is to identify a series of acts that, when applied to the starting arrangement, results in a sequential sequence. It is frequently employed in artificial intelligence algorithms as an example of search tactics. The n-puzzle issue is frequently used in AI research to evaluate the effectiveness of specific search algorithms, particularly heuristics, due to its huge solution search space. For each instance of a combinatorial optimisation issue to be solved, a heuristic algorithm is an approach based on intuition or experience that offers a workable solution at a reasonable cost. It's possible that the amount of departure from the ideal answer won't be known in advance. Although the cost is acceptable, it does not ensure that the solution is the one that have the greatest value. It is obvious that in heuristic search, the evaluation of node placements and search tactics are crucial, and the various algorithms employed can have a significant impact on the effectiveness and accuracy of the process.

The heuristic algorithm has a variety of forms. According to Kokash (2005), there are three main categories of heuristic algorithms: simple heuristic algorithms (such as greedy algorithms and hill climbing), meta-heuristic algorithms (such as tabu algorithms and genetic algorithms as well as simulated annealing), and hyper-heuristic algorithms.

This work focuses on the implementation of n-puzzle algorithms, including the uninformed algorithms breadth-first search, depth-first search, and various popular heuristics. In the following portions of the paper, a survey of the relevant literature is presented, followed by a definition of the n-puzzle problem. Then, the specific mechanics of the algorithms are discussed in detail within experiments. Multiple algorithms are chosen to assess the n-puzzle issue given identical initial states that

must be solvable. In the concluding section, the outcomes of the experiments are analysed and summarised.

# 2. Literature Review

Based on depth-first technique, Korf (1985) introduced depth-first iterative-deepening algorithm within 15-puzzle. This method is improved by the incorporation of fast-execution memory functions (Reinefeld and Marsland 1994). The depth-first iterative-deepening algorithm requires less space and less time than breadth-first search algorithms and depth-first search algorithms, respectively. (Reinefeld and Marsland 1994; Korf, Reid, & Edelkamp 2001). In addition, Korf and Taylor (1996) introduced a novel heuristic approach with a reduced net cost. Bauer and Bernard (1994) came up with a novel heuristic function that combines pair distance and Manhattan distance, which they then implemented in iterative deepening A*(IDA*). Manzini (1995) developed a new perimeter search technique that required less time for heuristic assessments. These based-on-previous-work algorithms have significantly enhanced the performance of the search.

Mathew, Tabassum, and Ramakrishnan (2013) compare the performance of five types of algorithms. They discovered that the greedy best-first search achieved the best performance when the answer was short, whereas A* performed best when the solution was long. However, the implemented heuristic algorithms were both simple heuristics; the research did not study any meta- heuristic or hyper-heuristic algorithms.

Bhasin and Singla constructed a genetic algorithm, a form of meta-heuristic algorithm, to solve the n-puzzle (2012). Their paper described the advantages of the Genetic Algorithm and its application. Another type of meta-heuristic method that might be used to solve the n-puzzle is simulated annealing. They offered an excellent summary of the convergence theory and evolution of simulated annealing. (1987, Henderson, Jacobson, & Johnson; 1994, Koulamas et al.)

There are other n-puzzle methods that have not been implemented in this study, and they require additional investigation. Korf and Schultze (2005) introduced many changes to the best-first search method in order to lower the algorithm's storage and time requirements during a large-scale concurrent search for an n-puzzle. In addition, Drogoul and Dubreuil (1993) proposed a distributed method for solving the n-puzzle issue that split the n-puzzle into a collection of subgoals. The improved algorithms were able to solve big n-puzzles, such as those with a size greater than 800, but with a high probability of inaccuracy due to their enormous space consumption.

Pattern database may be used to n-puzzles with a maximum of 24 puzzles (Korf & Felner 2002). The pattern database might more precisely estimate the expense of an n-puzzle (Fenler et al. 2004). It was improved by recording the starting and ending states, hence lowering storage space consumption (Zhou and Hansen, 2004). Their contributions might effectively lower the time and space requirements of algorithms like A*.

# 3. N-puzzle

Since its invention by Sam Loyd in the 1870s, when it was named the 15-puzzle, the n-puzzle problem has been a well-known problem for almost a century (Loyd, 1959). It is extensively used to evaluate the performance of heuristic algorithms based on their complexity, which makes it quite significant.

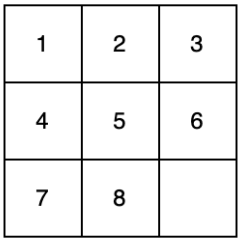


Figure 1 Goal State for 8-puzzle

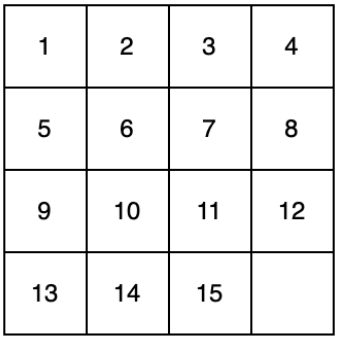


Figure 2 Goal State for 15-puzzle

The n-puzzle problem has several variants, including the 8-puzzle, the 15-puzzle, and other names such as Gem Puzzle, Boss Puzzle, Game of Fifteen, and Mystic Square, among many others. It comprises of a board with square tiles of equal size and quantity, as well as a "blank" area. The board is a square with dimensions of m × m, and the number of tiles is $N = m^2 - 1$. Initially, the tiles were arranged at random;

after a series of movements, the tiles should be in the target arrangement (Figure 1 and Figure2). During each round, only empty spaces are available for tile movement.

## 3.1 Solvability

In many starting situations, the n-puzzle cannot be solved. (1879, Johnson & Story). As Nilsson (1982) said, there are two types of tile configurations, even and odd, it is impossible to convert one to the other regardless of how many movements have been made. The solvability of the problem is explored.

There are total of $(N+1)!$ distinct permutations of tiles in the N-puzzle problem, given that the board contains a total of $N$ numbered tiles and a blank space. To represent each permutation, we expand a two-dimensional board to store all the tiles in a one-dimensional array, representing the numbered tiles as numbers and the empty spaces as zeros, and storing all the numbers representing the numbered tiles in order from top to bottom and left to right.

Now we will discuss the concept of inverted numbers. If after position $i$ in an array and there are $j$ numbers less than him, then the inversion number of location $i$ is $j$. Summarizing the inversion numbers at each place in an array yields the array's inversion number, represented by $Ivs$.

Wilbur (2001) has proved lemmas on the solvability of the puzzle, and here are the two lemmas, which apply to the two cases of n-puzzle with even and odd side lengths, respectively.

**Lemma 1**: If a board in an n-puzzle has an odd number of edge lengths, then the parity of the $Ivs$ in both its beginning and end states must be the same for it to be solvable.

**Lemma 2**: If the edge length of the board in an N-puzzle is even, then the following must be true for the puzzle to be solvable:

$$(Ivs_i + \Delta\, h)\ mod\ 2 = Ivs_e\ mod\ 2$$

$Ivs_i$ is the $Ivs$ of n-puzzle's initial state, $h$ is the difference between the rows of blank space at the initial state and the target state (in this project, the row spacing of the lowest row). $Ivs_e$ is the $Ivs$ of the end-state. Both sides of the equation have identical parity.

# 4. The Algorithms and Implementations

This article implements five distinct methods to solve the n-puzzle issue. Two of the algorithms are uninformed. The other three are heuristics: iterative deepening A*, simulated annealing, and the genetic algorithm. The next section discusses these algorithms and the system.

## 4.1 The System architecture

Figure 3 depicts the overall system architecture for this project.

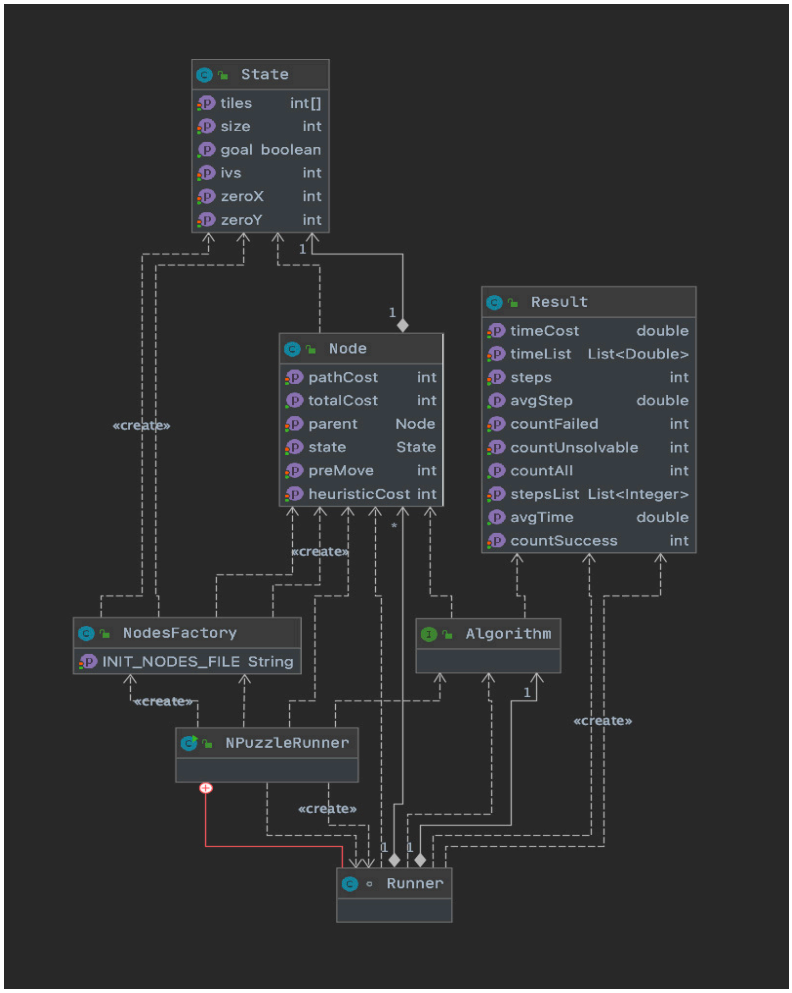


Figure 3 The structure of the system

### 4.1.1 Node, State and Result

All three classes implement the idea of encapsulation, encapsulating the nodes in the search tree, the arrangement of the tiles and the result of the search in each case respectively. The classes' structures are shown in Figure 4.

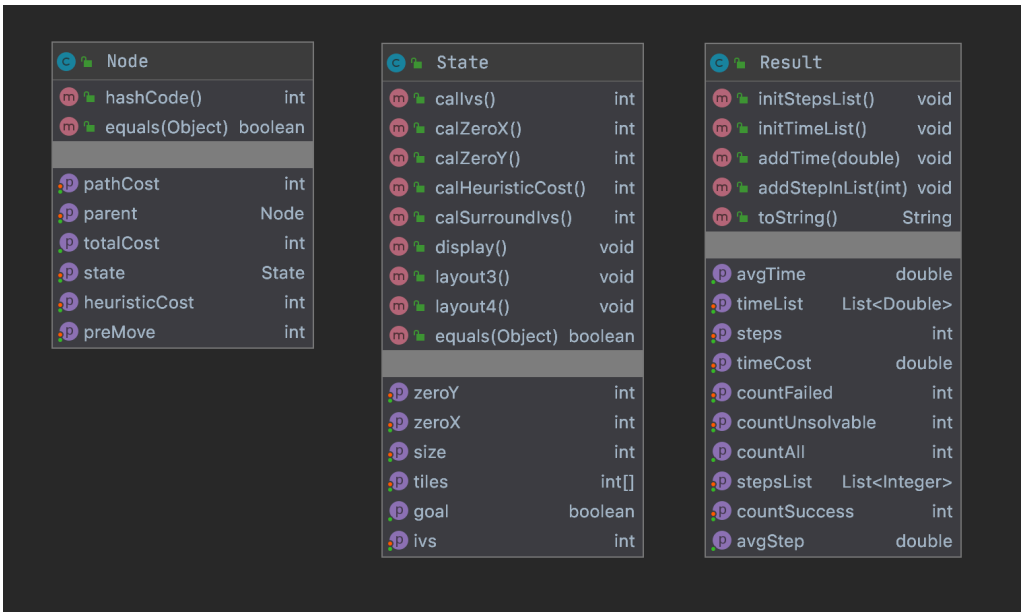


Figure 4 Node, State and Result

The $State$ represents an arrangement of tiles that stores numbered tiles in a one-dimensional array of indeterminate length. $Size$ represents the edge length. Meanwhile, $zeroX$ and $zeroY$ record the coordinates of the blank space to facilitate the movement of the blank space in the algorithm. It also has methods for calculating $ivs$ and displaying the current state.

The $Node$ is mainly used to represent the position in the search spanning tree, it has three main functions: storing and predicting cost, indicating the relationship between

parent and child nodes, and storing the direction of movement needed to move to that child node.

The $Result$ is used to store the time and step consumption of each algorithm when searching all inputs. For the input of the Nodes list, it not only statistics the success and failure rates of the list, but also the overhead of each initial node. The data in the Result is then shown on the console and recorded.

### 4.1.2 Nodes Factory

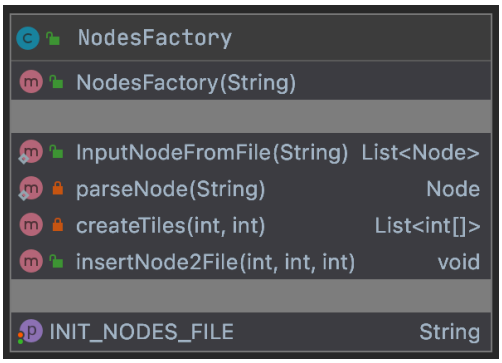


Figure 5 The Structure of NodeFactory

The Node factory is an implementation of the factory pattern in design patterns, its architecture is shown in Figure 5. First, the factory class uses the $createTiles()$ method, with input parameters the number of nodes and the size of the puzzle, to create a list of one-dimensional arrays, and then a series of arrays as a permutation of tiles, which is stored in a text file by shuffling through an array of target states. Secondly, the node factory can also read a list of tiles by line from a text file using the method $inputNodeFromFile()$. It is worth noting that a Factory corresponds to a file

address, which is determined as an input to the constructor when the Factory is initialised. The contents of the text file are shown in Figure 6.

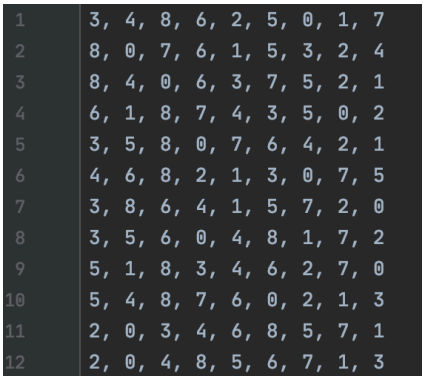


Figure 6 The Content of Text File

### 4.1.3 N-Puzzle Runner

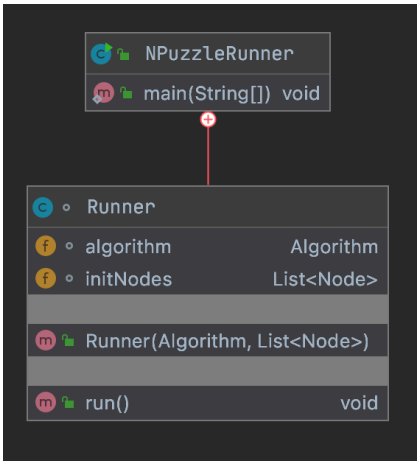


Figure 7 The N-Puzzle Runner

The Figure 7 shows the relationship between $NpuzzleRunner$ and its internal class $Runner$, as well as the parameters needed to start it. After each launch of the $NPuzzleRunner$, the $NodesFactory$ generates a list of nodes as roots based on the numbers in the file, instantiates the $Algorithm$, instantiates the $Result$, and takes the three as input to run the algorithm until it completes or time limit exceeded.

## 4.2 Breadth First Search (BFS)

The breadth-first algorithm is a well-known tree search algorithm. It is a kind of uninformed algorithm. This means that it expands without knowing any information other than the target state, or the state space. It is also a brute force algorithm for solving n-puzzle problems, which is an exhaustive method that begins with an initial state and tries all possible states: all possible states that can be generated by moving one step from the initial state, states that can be generated by moving two steps from the initial state, states that can be generated by moving three steps from the initial state, and so on.

The BFS keeps track of the outcome of each move, and each Node in the BFS has a single state. Figure 8 depicts all the states that may be formed by travelling two steps in each of the four directions (ignoring the duplication problem; each node represents a state); the BFS is extended in the following order: 1->2->3->4->5...->21.

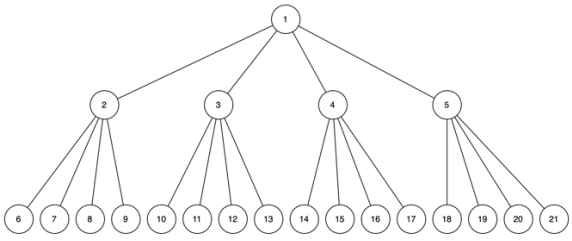


Figure 8 The Search Formed Tree

When extending nodes, the exhaustive enumeration technique encounters an issue with the duplication of states. Given instance, for an initial state root, after a left slider action develops a child node, a right slider operation is conducted on the child node such that the state of the enlarged child node is the same as the state of the root and

the state returns to its initial state. Even though the desired state may be reached in the long run, such a two-step move should not be present in the optimal solution.

There are two main solutions to the duplication problem produced by expanding child nodes: first, while moving expanding child nodes, record the way of movement (up and down, left and right) for each Node, and do not permit the following expansion to execute the opposite action. For a node obtained via an upward move operation, for instance, additional downward movements should be banned. Second, using the open dequeue and closed list, the expanded child node and expanded parent node are recorded, and all expanded nodes undergo a legality check to verify whether they have been expanded previously; if so, they cannot be added to the open list. This guarantees that every node in the open and closed tables is unique.

The BFS in this project will use a combination of an open dequeue and a closed list to remove duplicates, and the BFS implementation flow is shown in Figure 9.

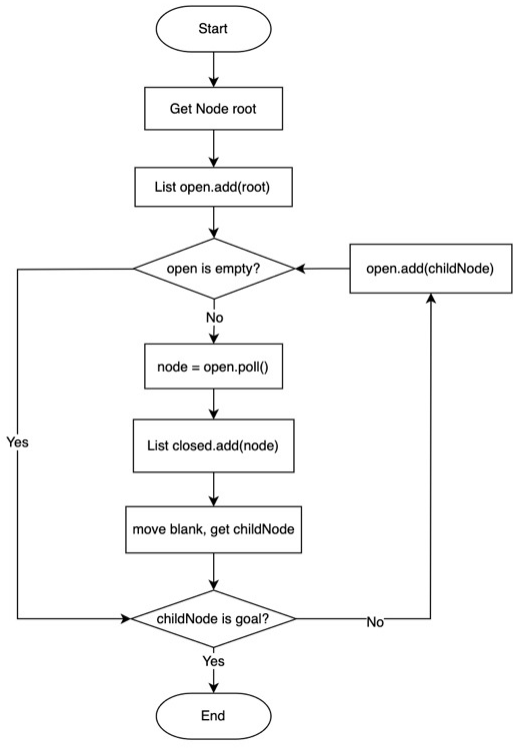


Figure 9 Breadth-First Search

As seen in the picture, BFS is implemented using an open queue and a closed list, and the BFS algorithm is a loop. When an expansion occurs, the route cost of the child node is larger than that of the parent node, hence the number of steps used during the search does not decrease. When a state with a step cost of $n$ is dequeued from open, all nodes with a step cost of $n-1$ have previously been dequeued, that is, all states with a step cost less than $n$ have been enumerated and searched. Therefore, when the BFS algorithm determines a path to the goal state, it will be the one that requires the fewest steps.

The basic steps of BFS are shown below：

```
1  open.add(root);
2  while (! open.isEmpty() && ! findTarget){
3        Node parent = open. poll();
4        closed.add(parent);
5        moveAndAddChildToOpen(parent);
6  }
```

## 4.3 Depth First Search (DFS)

DFS is another uninformed algorithm and a method of exhaustive search. For each search, DFS explores as deep as possible into each path until it cannot go to the next Node that has not been visited, and then starts trying a second path until all the pathways have been taken. For a search-formed tree, the expansion order of DFS is very different from that of BFS. As in Figure 8, the expansion order of DFS is

1->2->6->7->8->9->3->10->11->12->13->4...->21. After searching all of node 2's offspring, that is, nodes 6, 7, 8, and 9, it is time to search node 2's sibling, node 3.

DFS also requires two tables to verify that there are no duplicate states to prevent dead loops. Unlike BFS, however, the open table is a last-in-first-out stack, ensuring that every search is conducted in the same direction and that every expansion is as deep as feasible. Figure 10 illustrates the DFS execution flow.

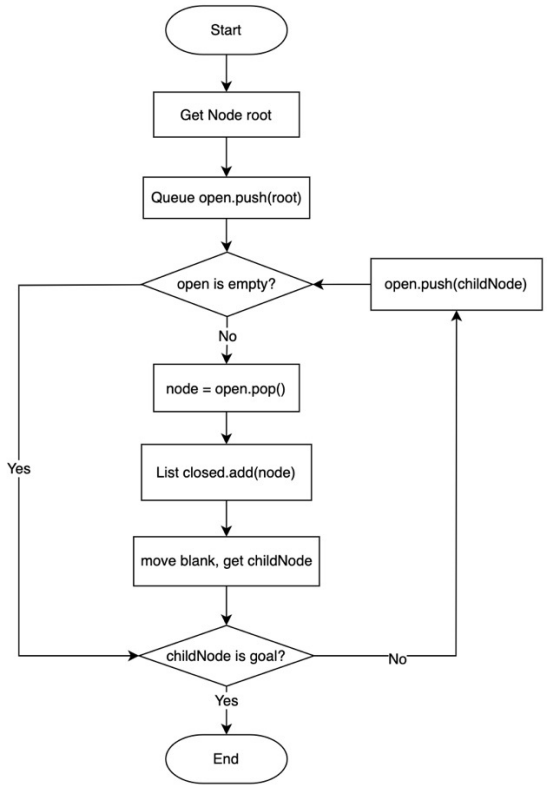


Figure 10 Depth-First Search Flowchart

DFS provides a plausible option for the next search after each search fails to reach the next unvisited node, thus the term backtracking. BFS is implemented via looping, whereas DFS may alternatively be accomplished by recursion.

The basic steps of DFS are shown below：

```
1  open.push(root);
2  while (! open.isEmpty() && ! findTarget){
3        Node parent = open. pop();
4        closed.add(parent);
5        moveAndPushChildToOpen(parent);
6  }
```

## 4.4 Iterative Deepening A* ( IDA* )

Iterative deepening A* (IDA*) is an iterative deepening depth-first search variation. (Korf, Reid, & Edelkamp 2001) A depth-first search is employed. DFS eliminates a significant amount of child-node growth and reduces space complexity, however it has a flaw. If the depth of the current path has no upper limit, DFS will be unable to initiate backtracking and the algorithm will encounter a deadlock. To tackle this issue, Korf (1985) suggested the iterative deepening depth-first search (IDDFS), in which the DFS is given a parameter to record the depth (or the number of levels of the answer tree) each time and the backtracking process is started once the route reaches a specific depth. The method constantly explores the top levels of the search-formed tree from the root node at each search, resulting in repeated expansions of the upper layers, which wastes a considerable deal of search time and considerably increases the time complexity. IDA* eliminates this deficiency.

Unlike IDDFS, IDA* utilises a recursive method to node expansion by adding a heuristic evaluation function to each stage of the process to get a 'heuristic

evaluation,' where the path is abandoned if it is anticipated that it would not reach the end even if it continues. Invoking the heuristic evaluation function is hence also known as "pruning." Following the completion of the prediction, the state is extended, and each time it is expanded to a new state, the heuristic is re-evaluated until a certain number of cycles have been completed.

In this study, the heuristic technique employs the Manhattan distance, where the heuristic evaluation is the minimal number of predicted steps f(n) to reach the end state from the present state. The typical iterative deepening depth-first search employs the search depth as a restriction on the number of searches per recursion, whereas IDA* does not. Rather, IDA* employs f(n):

$$f(n) = h(n) + g(n)$$

where g(n) is the path cost for this loop to the current one, and h(n) is a heuristic indicator for the predicted value of the current state from the target state.

The IDA*'s DFS triggers termination under two circumstances: 1, the search reaches the final state, i.e., the optimal path is found; 2, the sum of the number of steps the algorithm has searched, and the number of heuristic evaluation steps is greater than the limit, which is initially the predicted number of steps for the root state. However, when the n-puzzle size is large and the number of moving steps is excessive, the heuristic evaluation of the initial state is frequently less than the actual number of steps moved, and at this time, under the limitation of the predicted number of steps for the initial state, all DFSs are unable to search for the target state. Therefore, the algorithm must raise the step limit bound after each termination of the DFS, and the increase strategy used in this project is to update the limit after each termination to the lowest heuristic evaluations of the DFS that are larger than the bound value. The procedure is illustrated in Figure 11.

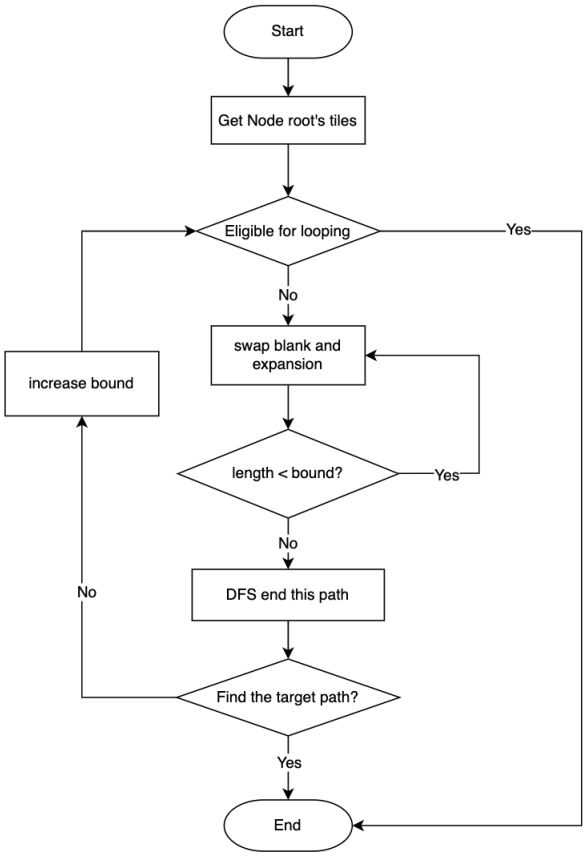


Figure 11 Iterative Deepening A*

The core steps of IDA* are shown blew:

```
1       for (bound = heuristicCost; bound <= BOUND; bound = dfs(0,
    heuristicCost, -1)) {
2                   if (pathGot) {
3                       return ;
4                   }
5               }
```

The $dfs()$ method represents a DFS search, where the parent node of the initial state is the root node and the last move of the root node is -1, as distinct from a general move.

## 4.5 Simulated Annealing

The algorithm simulated annealing (SA) is a greedy algorithm. A greedy algorithm selects the optimal option for each iteration in the current state to create the optimal outcome. It focuses on a limited part of the search space, as opposed to the entire space. In the n-puzzle problem, the parent node at each expansion only considers the state of its surrounding neighbours, and only the neighbour with the best heuristic assessment is chosen for swap.

The hill-climbing algorithm is the most well-known greedy algorithm, serving as the foundation for the simulated annealing approach. In the n-puzzle problem, the mountain climbing method is implemented by expanding from the starting point, adding the neighbour with the lowest heuristic cost to the open table at each step, and continuously updating this open table to form an iterative expansion until no neighbour with a lower heuristic cost can be found. If the heuristic cost is zero, the goal state is discovered; nonetheless, the mountain climbing approach is susceptible to the local optimality problem. Since the greedy algorithm does not examine the entire, it frequently produces a locally optimal solution, but not necessarily the global optimum one; it is referred to as "falling into local optima."

Simulated annealing is an improvement on the mountain-climbing approach, which cannot identify a globally optimum solution but instead climbs the mountain step by step. Simulated annealing was developed in 1983 (Aydin and Fogarty, 2004) because it resembles the physical mechanism of solid annealing. Each iteration of solid annealing consists of heating the solid to the appropriate temperature and then allowing it to cool gradually. As the temperature rises, the internal particles of a solid become less ordered and further from the target state, whereas as the solid cools,

the internal particles become progressively more ordered, reaching an equilibrium state (steady state) at each temperature and a ground state at room temperature when the orderliness is at its global maximum. Its overall process is comparable to that of mountain climbing, in which the expansion will choose the collocation with the optimal heuristic evaluation for expansion. However, the simulated annealing method introduces a random factor (Bertsimas and Tsitsiklis 1993), a probability $P$ called the acceptance probability, and $P$ is calculated as follows:

$$P = \begin{cases} 1\ , & E(n+1) < E(n) \\ e^{-\frac{E(n+1)-E(n)}{T}}, & E(n+1) > E(n) \end{cases}$$

Where $E(n)$ is the heuristic evaluation of the current state, $E(n+1)$ is the heuristic evaluation of the neighbor state, $T$ is the temperature in the current loop and $P$ is the acceptance probability of accepting the neighbour state.

The formula indicates that if the heuristic assessment improves, the move will be accepted for this state (acceptance probability $P$ = 1); if the heuristic evaluation worsens, it indicates that the state is further from the goal, but the algorithm will not immediately reject it. It generates a random number $\varepsilon$, if ε< $P$, then the state with the inferior value is still permitted, — in other words the slightly more expensive point is accepted; otherwise, the transfer is denied, and so on. The algorithm accepts the state with a certain probability, but as the temperature drops, the probability of acceptance $P$ diminishes. The algorithm's flowchart is presented in Figure 12.

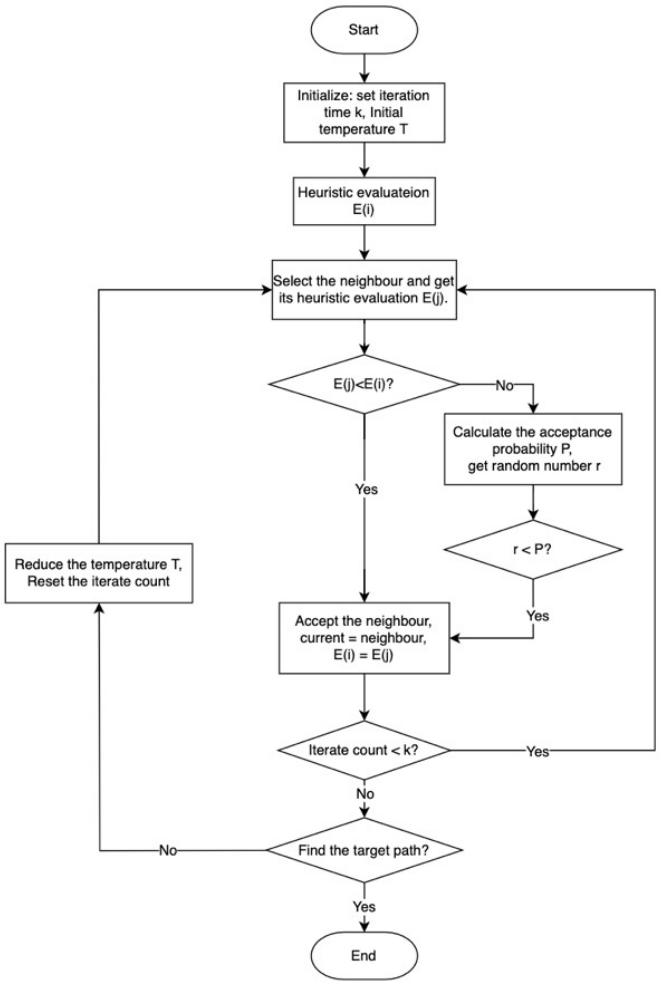


Figure 12 Simulated Annealing

The core steps of SA are shown blew:

```
1   while (temperature > temperature_min) {
2       for (int i = 0; i < LOOP_COUNT; i++) {
3           Node childNode = randomMove(cur);
4           If (SystemWouldAccept){
5               Cur = childNode;
6           } else {
7               continue ;
8           }
```

```
9         }
10  }
```

## 4.6 Genetic Algorithm

The genetic algorithm is a standard example of an evolutionary algorithm. Evolutionary algorithms arose from computer simulations of biological systems that imitated the principles of biological evolution in nature, incorporating the concepts of evolution and natural selection from biology into search algorithms (Liepins and Hilliard, 1989). In the search process, the genetic algorithm creates initial values at random each time and evolves the ideal solution to the issue through three primary operations: select, crossover, and mutation across N iterations (Davis 1991). Essentially, it is a simultaneous stochastic global search and optimisation algorithm.

In this part, the fundamental principles of the genetic algorithm are discussed, followed by an explanation of how these biological theories may be incorporated into an algorithm, and then the genetic algorithm is implemented.

### 4.6.1 Individuals, genes, and populations

Individuals, often referred to as chromosomes, represent a single search effort; an individual is composed of a sequence of genes, each of which represents a single decision variable for a move issue. Various individuals (chromosomes) constitute a population, hence the population of solutions consists of multiple solutions to the problem.

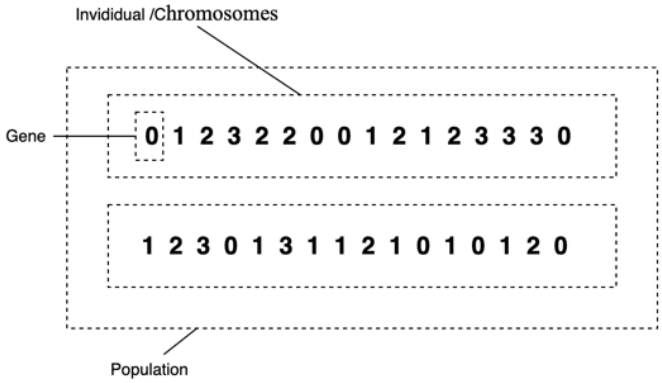


Figure 13 Genes, individuals, population

In the n-puzzle problem, each step is treated as a gene, with 0, 1, 2, and 3 signifying up, down, left, and right, respectively. An individual (chromosome) is the set of movements constituting a single attempt along a path of specified length. Figure 13 depicts the various populations and individuals.

### 4.6.2 Fitness Score

After obtaining a population, the merits of each individual are evaluated, and based on the results of the evaluation, some good individuals are selected in each generation, while the less well-adapted individuals are eliminated. As a result, the chromosome quality improves with each generation, and the solution approaches its optimal state. This measurement of a species' environmental adaptation is known as the fitness score.

In this project, the fitness score is the least heuristic cost of each chromosome as it advances onto slide tiles, which indicates the fewest amount of steps each individual must take to reach the target state.

### 4.6.3 Selection

In the parent population, individuals and genes with high fitness are selected for through the process of selection for replication. This study applied a tournament selection technique in which the chromosome with the least Manhattan distance created during each selection move was chosen as the best for the following generation.

### 4.6.4 Crossover

The algorithm can choose the best individuals based on fitness score throughout each iteration, but this is not the end of the process. Each repetition requires the crossover operation in order to increase the fitness score. This process of creating child chromosomes from two parent chromosomes is known as a crossover, which can be classed as a single-point crossover, two-point crossover, concordant crossover, sequential crossover, or cyclic crossover. The most popular is the single point crossover, in which the parent chromosome is severed from a randomly created identical position on each parent chromosome (this position is also called the mating point) and the severed parent chromosomes are then spliced together to form the offspring genes. This process is illustrated in Figure 14.

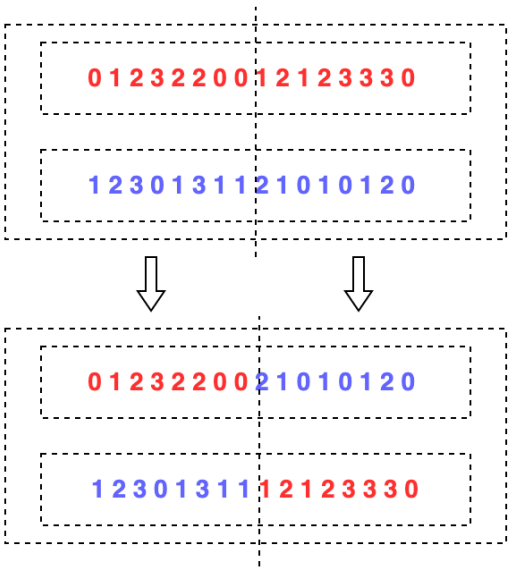

Figure 14 Crossover

### 4.6.5 Mutation

Crossover guarantees that each evolution leaves excellent genes behind, but it merely picks and reorders the same set of results; no new genes are added. This merely guarantees that after N iterations, the computational output approaches the local optimum solution, but never the global optimal solution. To address this issue, the genetic algorithm requires an operator known as the mutation.

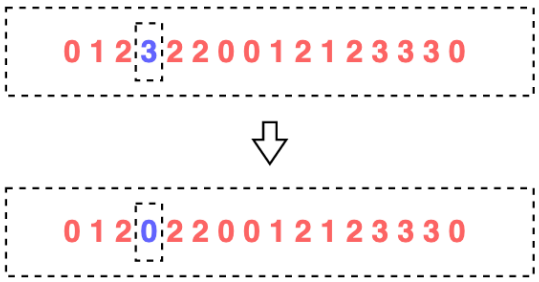


Figure 15 Mutation

As depicted in Figure 15, when an individual duplicates a gene, there is a small chance that the gene will change, and this change is completely random, with the location of the selected gene and the value of the modification being both random, allowing the global optimum to be discovered beyond the limits of the local optimum.

Following an introduction to the ideas, we may deduce the genetic algorithm's flow as shown in Figure 16.

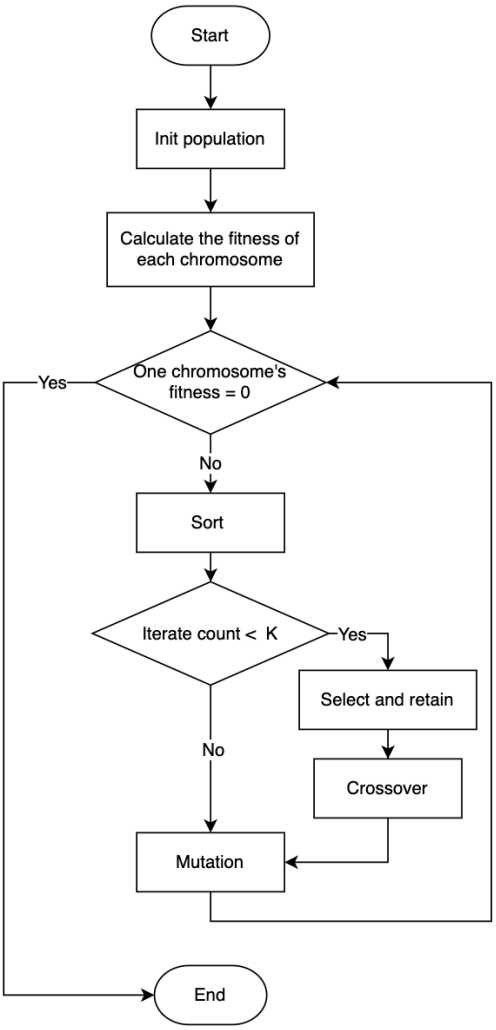


Figure 16 Genetic Algorithm Flowchart

This project also enhances the conventional genetic algorithm. Traditional genetic algorithms keep the best individuals from each iteration to achieve algorithm convergence. However, when iterations are repeated, the diversity of individuals reduces, resulting in premature convergence. This project addresses this issue by establishing a value for the number of iterations during which the best individuals do not change, as well as a constant constraint K. If the best individuals do not change during the K generation, the following expansion will eliminate the retention of the best individuals and crossover but will simply undertake the mutation operation, thereby developing. The original execution procedure will be replayed after one generation. This results in a generation of mutation only after each generation of

evolution, thereby compensating for the lack of population variety and preventing early convergence.

The genetic algorithm requires initial parameterization. In this research, we consider factors such as population size $NUMS$, chromosomal length $LENGTH$, and the evolutionary generation limit $K$ for retaining optimum individuals. The values of these parameters are provided in Table 1.

Table 1 Genetic Algorithm Parameters

| | NUMS | K | LENGTH |
|---|---|---|---|
| 8-puzzle | 10 | 10 | 50 |
| 15-puzzle | 30 | 10 | 100 |

The core steps of GA are shown blew:

```
1   while (geneCount < GENERATION) {
2       population = calFitnessInChromosome(state, population);
3       population = sort(population);
4       if (population [0] [chromosome.length - 1] == 0) {
5           return ;
6       }
7       crossover(population);
8       mutation(population);
9   }
```

# 5. Result Analysis

To ensure that the project's findings are not impacted by external influences, the identical setup of a single machine is used, and all algorithms are written in java version 14.0.2. The operating system used is macOS Big Sur, and the compiler is IntelliJ IDEA.

## 5.1 Uninformed Algorithms

We selected the iterative-dependent A star as a sample heuristic approach to compare with conventional search (breadth-first search, depth-first search). Five randomly generated identical 8-puzzle states were utilised as input for two uninformed algorithms and a heuristic algorithm before the search was began. Multiple identical 8-puzzle entries were used to evaluate three indicators: the number of nodes created by the method, the time required to search, and the greatest depth to grow the search-formed tree.

The experimental results are shown in Table 2, where we expand the states into linear arrays in the order from top to bottom from left to right to represent.

Table 2 Results of 8-puzzle

| No | Initial State | Nodes Generated | | | Time Consumption | | | Depth of Search-Formed Tree | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | DFS | BFS | IDA* | DFS | BFS | IDA* | DFS | BFS | IDA* |
| 1 | 6, 4, 7, 8, 5, 0, 3, 2, 1 | 73827 | 181437 | 16660 | 40.911s | 721.553s | 0.016s | 40755 | 31 | 31 |
| 2 | 5, 6, 3, 4, 7, 8, 1, 0, 2 | 26920 | 100387 | 4099 | 3.234s | 151.598s | 0.015s | 14825 | 23 | 23 |
| 3 | 2, 4, 3, 0, 8, 7, 5, 1, 6 | 14085 | 13617 | 192 | 0.694s | 1.173s | 0.012s | 7741 | 17 | 17 |
| 4 | 3, 5, 4, 6, 2, 8, 1, 7, 0 | 4108 | 81321 | 771 | 0.075s | 91.395s | 0.016s | 2256 | 22 | 22 |
| 5 | 8, 4, 6, 2, 7, 0, 5, 1, 3 | 46784 | 155257 | 721 | 14.901s | 452.588s | 0.013s | 25853 | 25 | 25 |

As seen in the table, the depth of the search-formed trees created by depth-first searches is significantly larger than the depth of those generated by breadth-first searches. As previously described, breadth-first search grows the tree layer by layer, ensuring that the shortest path, with the fewest number of steps, is always sought. In contrast, depth-first search tends to grow in a single direction, and each time a parent node extends a child node, the search-formed tree deepens by one layer. Hence, the depth of expansion is sometimes many times that of breadth-first search.

However, as can clearly be observed from the table, the greatest issue with the breadth-first search approach is that the number of nodes to be searched is excessively enormous, as every node in each layer is looked for. As the depth of the search tree's nodes rises, the number of nodes to be searched at each level grows swiftly and exponentially, and as a result, the storage space required and the time spent searching both increase exponentially.

The heuristic algorithm IDA* addresses each of these shortcomings. As shown in the table, IDA* offers benefits over both BFS and DFS since it adds a bound to the search process to ensure that the depth of the search-formed tree corresponds to the depth of the shortest path. It may also be used to solve the 15-puzzle, which is a strength of the heuristic method since it prevents the proliferation of superfluous nodes and lowers the number of pointless pathways to be examined.

## 5.2 Heuristic Algorithms

As heuristic algorithms, IDA*, simulated annealing, and genetic algorithms can more efficiently solve n-puzzle problems. To verify this, we will generate 1000 solvable 8-puzzle states and 100 solvable 15-puzzle states by the Nodes Factory as initial states and then count the number of successes, number of failures, success rate, average number of steps spent on success, and time consumed by the three heuristics to determine the efficiency of the three algorithms in solving n-puzzles.

### 5.2.1 8-puzzle

First, we used the 1000 8-puzzle states generated by the Nodes Factory for the three heuristic algorithms to search. The parameters for the simulated annealing were

attenuation rate of 0.99, loop count of 150, initial temperature of 5, and minimum temperature of 0.001; the results of these three searches are shown in Table 3.

Table 3 Result of Heuristic Algorithm for 8-puzzle

| Algorithm Type | Average Steps | Total Time Consumed | Success Rate |
|---|---|---|---|
| IDA* | 2208.499 | 0.26s | 100% |
| SA | 20778.4 | 4.907s | 99.80% |
| GA | 6.26 | 31.39s | 100% |

Where average steps denote the average number of times the algorithm grows. In IDA* and SA, it signifies one blank motion, but in GA, it signifies one population evolution. In addition, GA has the longest running time, IDA* has the lowest running time, and SA has the second-longest running time after IDA. Nevertheless, compared to other heuristics, SA cannot guarantee a 100% success rate. This is due to SA limiting the number of searches and finishing the process when it anneals to the lowest temperature. This restriction will affect the success rate while decreasing the likelihood of dead loops. The total time used by the three heuristics to solve an 8-puzzle with 1000 puzzles may be even less than the time spent by the uninformed search to solve a single 8-puzzle. Therefore, we may infer those heuristics solve 8-puzzle problems significantly better than the uninformed algorithm.

### 5.2.2 15-puzzle

Second, we utilised 100 15-puzzle beginning states, which were generated by the Nodes Factory as explained before, for the three heuristic algorithms to search, with SA adjusted differently based on the attribute attenuation rate: 0.99, 0.999, and 0.9999, respectively, while the remaining parameters remained the same as before.

The outcomes of these three 15-puzzle-solving algorithms are displayed in Table 4 below.

Table 4 The Result of Heuristic Algorithm for 15-puzzle

| Algorithm Type | Average Number of Steps | Total Time Consumption | Successes Rate |
|---|---|---|---|
| IDA* | 6320771.696 | 589.366s | 100% |
| SA (0.99) | 38419.0 | 3.412 | 11% |
| SA (0.999) | 39845.442 | 30．845s | 52% |
| SA (0.9999) | 3457833.47 | 170.534 | 100% |
| GA | 4200.29 | 3523.648s | 100% |

As shown in the table, IDA* can solve the 15-puzzle faster and with fewer searches than the genetic algorithm. However, the genetic algorithm does not offer a considerable advantage in solving the issue. The reason for this is that the crossover operator employed by the genetic algorithm is a global search. It builds a population of several individuals to initiate a parallel search operation. This enormous number of global searches precludes the solution from being quickly retrieved within a short range (8-puzzle and 15-puzzle).

The 15-puzzle sheds further light on SA's lack of success. The effectiveness of simulated annealing is extremely parameter-dependent. While the attenuation rate is 0.99, the same as when searching for an 8-puzzle, the success rate for solving a 15-puzzle is just 11%; even when the temperature decay rate is decreased and the attenuation rate is increased to 0.999, the success rate is still only 52%. Only by raising the attenuation rate to 0.999 would it be possible to attain a success rate of one hundred percent, but the time required will also rise dramatically. Consequently,

when solving the n-puzzle issue, SA must adjust the attenuation rate to a number that strikes a balance between the success rate and the time required. If the settings are appropriately adjusted, the simulated annealing process is far more effective than IDA*.

# 6. Conclusion and Future Work

1）Traditional algorithms such as BFS and DFS perform much worse than the heuristic algorithm IDA* for solving n-puzzles. Both forms of uninformed search require a high initial state. During repeatedly attempting the way, they will expand numerous meaningless nodes, using a great deal of time and space. In addition, we were aware of the obstacles that standard algorithms would face while attempting to solve the 15-puzzle issue. In contrast, IDA* is expanded using a depth-first expansion strategy and a valuation function that works better in both 8-puzzle and 15-puzzle problems.

2）The attenuation rate has an impact on the performance of the simulated annealing procedure. If the attenuation rate is low and the temperature decays too quickly, then the time and step consumption will be considerably less, but the success rate will be significantly lower. When resolving a low-complexity 8-puzzle, the algorithm might be given a larger decay rate to decrease time lost. In the case of the more difficult 15-puzzle, however, if the temperature drops too quickly, the success percentage would plummet. It is impossible to correctly locate the desired path. This experiment explored three rates of 0.99, 0.999, and 0.9999. In the future, we will investigate the link between the performance of the simulated annealing technique and the temperature and decay rate, as well as optimise the annealing process under 15-puzzle and 24-puzzle conditions.

3）The genetic algorithm performs poorly on the 8-puzzle and 15-puzzle problems, obtaining neither the best nor the fastest solution. This is since that the genetic algorithm's crossover operator is a global search. Ture plans include tweaking the evolutionary algorithm's parameters, applying it to the 24-puzzle, and comparing it to other heuristic algorithms. Next, we can attempt to build a better chromosome

code, select the optimal number of populations and chromosomal lengths, create a valuation function that better matches the situation, and enhance the genetic operator to optimise genetics.

# 7. Reflection

This challenging project has taught me not only a good deal of useful technical programming abilities, but also essay writing and literary research skills.

Initially, I did not attempt to construct a comprehensive system for the project's architecture, but rather used five distinct Java files to finish the experiment. This leads to duplicate code and poor user readability. Besides, the system was necessary to segregate the status of the recording problem from the Java file used to regulate input. Based on the preceding considerations, the system must be structured as a modular project to decrease the coupling between the system's runner, input, and output values. In addition, it is important to note that the system must be scalable for future study on 24-puzzle and more complicated issues. I learnt design patterns such as the factory pattern and the singleton pattern while developing, which optimised the system to fulfil these specifications. The project was constructed with the encapsulation of node, state, and result as a primary concern, and the result is satisfactory.

I also attempted A *'s of the hill climbing method while selecting an n-puzzle solution, but this presented a formidable obstacle. In these studies, I discovered that the success rate of the hill climbing algorithm decreased too rapidly with rising n-puzzle optimum path complexity. Its success rate against the 8-puzzle was small, preventing the system from counting the number of steps and time spent, and rendering the trial findings unrepresentative. This is a compromise made by hill-climbing, a greedy algorithm, to avoid reaching a local optimal solution. Therefore, I was compelled to employ the simulated annealing technique as an improvement to hill-climbing in order to achieve the thesis's objective.

During this study, my capacity to locate and analyse literature has also been much increased. This essay took much research, reading, and paper summarization, among other tasks. In the literature review and introduction sections, I was required to look for relevant publications, extract relevant material, and summarise it for reference in the appropriate section. This was a talent that I previously lacked, and I have grown a lot after all these experiences. Besides, Due to a lack of prior studies in artificial intelligence, I had to devote a great deal of time to studying n-puzzle and heuristic algorithms, particularly genetic algorithms. In addition, writing the thesis presented me with a significant task. As a developer who focuses mostly on

the design and coding of computer systems, I am not accustomed to performing a great deal of writing. This article has provided me with essential knowledge in this area.

I have made substantial progress on this topic overall, from having no relevant basis to creating three heuristic algorithms. In addition, I am satisfied with the current system design for achieving the objectives. However, during the project's testing phase, I discovered that there are still several areas that could be improved. For instance, the construction of a new State needs a deep copy of an array to be modified afterwards, which improves the code's readability but consumes a great deal of memory. The topic of whether this would be different or even better if the expansion was performed on the original array requires additional exploration. In the future, I will attempt to optimise this system more and apply it to other challenging puzzle situations.

# References


Aydin, M. & Fogarty, T. (2004). A distributed evolutionary simulated annealing algorithm for combinatorial optimisation problems. *Journal of heuristics*, *10*(3), 269-292.

Bertsimas, D., & Tsitsiklis, J. (1993). Simulated annealing. *Statistical science*, *8*(1), 10-15.

Bhasin, H., & Singla, N. (2012). Genetic based algorithm for N-puzzle problem. *International Journal of Computer Applications*, *51*(22).

Bauer, B. (1994). The Manhattan pair distance heuristic for the 15-puzzle. *Paderborn, Germany*.

Davis, L. (1987). Genetic algorithms and simulated annealing

Davis, T. E. (1991). *Toward an extrapolation of the simulated annealing convergence theory onto the simple genetic algorithm*. University of Florida.

Drogoul, A., & Dubreuil, C. (1993, May). A distributed approach to n-puzzle solving. In *Proceedings of the Distributed Artificial Intelligence Workshop*.

Felner, A., & Adler, A. (2005, July). Solving the 24 puzzle with instance dependent pattern databases. In *International Symposium on Abstraction, Reformulation, and Approximation* (pp. 248-260). Springer, Berlin, Heidelberg.

Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, *22*, 279-318.

Henderson, D., Jacobson, S. H., & Johnson, A. W. (2003). The theory and practice of simulated annealing. In *Handbook of metaheuristics* (pp. 287-319). Springer, Boston, MA.

Johnson, W. W., & Story, W. E. (1879). Notes on the “15” puzzle. *American Journal of Mathematics*, *2*(4), 397-404.

Koulamas, C., Antony, S. R., & Jaen, R. (1994). A survey of simulated annealing applications to operations research problems. *Omega*, *22*(1), 41-56.

Kokash, N. (2005). An introduction to heuristic algorithms. *Department of Informatics and Telecommunications*, 1-8.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, *27*(1), 97-109.

Korf, R. E., & Felner, A. (2002). Disjoint pattern database heuristics. *Artificial intelligence*, *134*(1-2), 9-22.

Korf, R. E., & Schultze, P. (2005, July). Large-scale parallel breadth-first search. In *AAAI* (Vol. 5, pp. 1380-1385).

Korf, R. E., & Taylor, L. A. (1996, August). Finding optimal solutions to the twenty-four puzzle. In *Proceedings of the national conference on artificial intelligence* (pp. 1202-1207).

Korf, R. E., Reid, M., & Edelkamp, S. (2001). Time complexity of iterative-deepening-A$*$. *Artificial Intelligence*, *129*(1-2), 199-218.

Loyd, S. (1959). *Mathematical puzzles* (Vol. 1). Courier Corporation.

Liepins, G. E., & Hilliard, M. R. (1989). Genetic algorithms: Foundations and applications. *Annals of operations research*, *21*(1), 31-57.

Manzini, G. (1995). BIDA∗: an improved perimeter search algorithm. *Artificial Intelligence*, *75*(2), 347-360.

Mathew, K., Tabassum, M., & Ramakrishnan, M. (2013). Experimental comparison of uninformed and heuristic AI Algorithms for N puzzle solution. *Proceedings of the International Journal of Digital Information and Wireless Communications, Hongkong, China*, 12-14.

Nilsson, N. J. (1982). *Principles of artificial intelligence*. Springer Science & Business Media.

Reinefeld, A., & Marsland, T. A. (1994). Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(7), 701-710.

Wilbur, E. (2001). Topspin: Solvability of sliding number games. *Rose-Hulman Undergraduate Mathematics Journal*, *2*(2), 2.

Zhou, R., & Hansen, E. A. (2004, July). Space-efficient memory-based heuristics. In *AAAI* (pp. 677-682).