IDENTIFYING OPTIMAL N-PUZZLE SOLUTIONS USING ARTIFICIAL INTELLIGENCE

AUTHOR:

ANDREW JOHN TAYLOR

DEGREE:

MSC COMPUTING

NAME OF SUPERVISOR:

YUKUN LAI

NAME OF INSTITUTION:

SCHOOL OF COMPUTER SCIENCE AND INFORMATICS, CARDIFF UNIVERSITY

DATE OF SUBMISSION:

13/09/2013

Abstract

A sliding block puzzle, commonly referred to as the N-Puzzle provides an interesting foundation on which to base research and experimentation within the field of Artificial Intelligence. Various algorithmic search approaches have been developed to computerise the solving of these puzzles, many of which utilise heuristic techniques. This document explores a range of these search techniques and heuristic concepts, and also compares the performance of each through rigorous testing and analysis of their implementation with both the 8-Puzzle and the 15-Puzzle. Not only does this document analyse the quality of existing ideas, but it also addresses the possible enhancement of existing approaches.

Acknowledgements

I would firstly like to express my appreciation to Yukun Lai. It has been a pleasure having such a valuable supervisor. His knowledge and expertise have certainly been a significant help throughout my dissertation. I would like to thank him particularly for making himself available to freely offer advice whenever it was necessary.

I would also like to thank my girlfriend, Charlotte for her patience and motivation. Without her constant encouragement this dissertation would have been a great deal harder.

I would like to express my gratitude to sister, Abigail for all of her guidance and valuable advice, which has been an amazing help to me.

Last but by no means least I would like to thank my parents, Philip and Beryl for their continuous support throughout everything I have ever done and for being such wonderful role models.

Contents

1 Introduction

An N-Puzzle is a square board which consists of $\sqrt{(n+1)}$ rows and columns and is filled with n square tiles. This leaves a gap on the board, into which adjacent tiles can be slid. Each tile either contains a numeric character, or a constituent part of a graphic, so that they can be differentiated between. A valid move consists of moving a tile that is adjacent to the space into that space. The aim of the N-Puzzle is to carry out a series of valid moves to transform the puzzle from the initial arrangement into the specified target arrangement.

The N-Puzzle forms a fascinating platform for Artificial Intelligence research for a number of reasons. The game at face value is extremely simplistic, with only a few very basic semantic rules. For such a simple problem however, due to the number of possible permutations each N-Puzzle has, these puzzles form the basis of a vast amount of both mathematical concepts and computational research. Most of the computational research that stems from this subject involves exploring numerous ways of enabling programs to make a series of informed decisions when necessary.

I will be investigating the most efficient, effective and computationally feasible ways in which these mathematical problems can be solved with a computer, by utilising a series of intelligent algorithms. I will focus my research onto the 8-Puzzle and the 15-Puzzle and aim to deduce which of the existing approaches is most effective. I also aim to apply my findings in order to implement some approaches of my own. To assist with my research I will build a puzzle solver, through which puzzles can be loaded and solved using a selected solving method from a list of options.

The knowledge I glean from this piece of research will provide me with a broad understanding of how computers can be utilised to solve problems with potentially enormous state space, such that random trial and error is not an option. My findings will be beneficial to both computer scientists who share an interest in the field of Artificial Intelligence and mathematicians who are fascinated in an algorithmic approach to problem solving.

2 Background

2.1 Solvability

As pointed out by Johnson and Story (1879), most people assume that all N-Puzzles are solvable, regardless of how the tiles are initially placed. This was certainly true on my account. Until carrying out my research, I was oblivious to the fact that N-Puzzles are not in fact always solvable. Unless I incorporate some sort of solvability check, an insolvable initial configuration would cause any algorithm I implement to run infinitely (or until the available RAM has been drained), searching for a solution that doesn't exist. Therefore it is vital that I gain an understanding of the solvability of N-Puzzles.

Sam Loyd's Puzzle

A puzzle maker named Sam Loyd created a 15-Puzzle in the 1870's, for which he offered a cash prize to anyone who could solve the puzzle (Archer 1999). Archer (1999) describes how this puzzle actually had no solution.

Archer (1999) describes the puzzle issued by Sam Loyd as looking something like the puzzle shown in Figure 2-1.

Initially the tiles in Sam Lloyd's puzzle are labelled with the numbers 1 to 15, arranged in numerical order as shown in Figure 2-1.

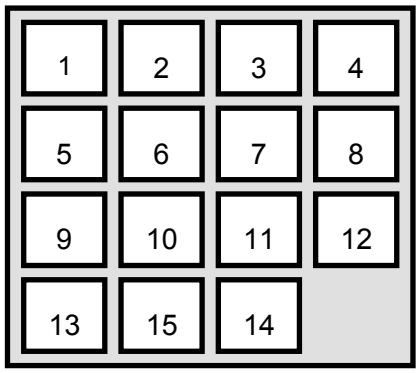


Figure 2-1: Initial configuration of Sam Loyd's puzzle

The aim is to maintain the original positions of all the tiles numbered 1 to 13, but swap positions of the tiles labelled 14 and 15 as shown in Fig 2-2.

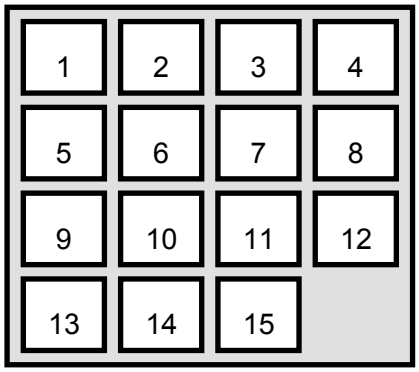


Figure 2-2: Target configuration for Sam Loyd's puzzle

Why is Sam Loyd's puzzle unsolvable?

A mathematical approach to viewing different puzzle arrangements is to consider them as being different permutations of the puzzle tiles. The overall aim is to effectively swap the tiles labelled 14 and 15, whilst preserving the positions of all other tiles and also the position of the space. The target configuration is an odd permutation of the initial configuration (Archer, 1999). However, changing the position of the space and then retaining the original position of the space requires an even number of moves, which as Archer (1999) highlights, always results in an even permutation of the initial configuration. It is the contradiction of these target conditions that makes Sam Loyd's puzzle insolvable.

Likelihood of choosing an insolvable puzzle

The number of permutations of a set of n elements is n! (Meserve, 1953), thus any given n puzzle has (n+1)! possible arrangements (considering the space to be a blank tile). Archer (1999) reveals that because the N-Puzzle can be represented as a bipartite graph, exactly half of these configurations are solvable, leaving the other half impossible to solve. Therefore Sam Loyd's puzzle is one of 16!/2 initial configurations of the 15-Puzzle which are impossible to solve.

As a result, when choosing a random starting configuration, choosing an insolvable arrangement is just as likely as choosing a solvable arrangement. Ideally, my algorithm should perform a quick check before it attempts to solve a puzzle whether it is actually solvable or not.

2.2 NP Completeness (NPC)

My initial assumption was, given how advanced modern technology is, that an algorithm would already exist, which has the ability to quickly and efficiently solve any N-Puzzle in existence. Much to my surprise, Ratner and Warmuth (1986) state that such an algorithm does not exist, due to the problem being NP-Complete. This makes it all the more fascinating a puzzle to research.

2.2.1 What is NP Completeness

Before exploring the ramifications of NP completeness, it makes logical sense to first introduce two mathematical concepts called the P class and the NP class. The P class is a complexity class, which consists only of problems that can be solved in polynomial time (Dasgupta et all. 2006, p. 258). The NP class however, which Dasgupta et all. (2006, p. 258) explain stands for “nondeterministic polynomial time”, is a complexity class that contains problems, for which a proposed solution can be verified in polynomial time Cormen et al. (2009).

Obviously, it is always easier to verify a solution than it is to find a solution, which is why Cormen et al. (2009, p. 1049) state that if a problem is in the P class, then by definition it must also be in the NP class.

The NP complete class is simply a subgroup of the NP class, whose members do not exist in P (Cormen et al. 2009, p. 1049). In other words, their solutions can be verified in polynomial time, but the puzzles themselves can not be solved in polynomial time.

Cormen et al. provide rigorous proofs supporting the notion that the N-Puzzle problem is NP-hard, and state that this makes the problem of finding an optimal solution “computationally infeasible”.

2.2.2 Implications of NP Completeness

The NP-Completeness of the N-Puzzle is what makes it such a thought-provoking topic for research, both for mathematicians and computer scientists. The impact that this will have on my research is that for some of the more complex cases of the N-Puzzle, I am not guaranteed to find an algorithm that quickly and efficiently calculates the optimal solution. Be that as it may, I still have a range of possible options available to me, enabling me to solve cases of a reasonable complexity. The NP completeness of the N-Puzzle is certainly one of its aspects that most drew me to choosing it for my dissertation topic.

2.3 Project Objectives

There already exists a fairly large amount of research in this area, particularly for the N-Puzzle. However, a lot of this information is scattered throughout a large number of sources. Many of these do not fully describe/explore the theory in its entirety and many do not provide rigorous analysis of the performance of the majority of available techniques. Through this project I will develop and analyse a fully functioning N-Puzzle solver, which implements a number of existing algorithms that I feel are best suited to the problem. I will use this as a research tool, to explore the strengths and weaknesses of each approach that I implement. I will also analyse how their performance compares to one another. In addition to this, I will carry out some further experimentation by implementing an adaptation of these approaches with some ideas of my own. Not only do I aim to make a contribution towards research in this field, but I also hope to refine my programming skills, which I have developed over the last year as part of my MSc course at Cardiff University.

2.4 Approach

Research Approach

My first aim is to develop a program that, given a specific board configuration, can find the shortest route possible to reach the target board configuration by utilising any one of a number of search techniques chosen by the user.

Skiena (2008, p. 344) and Dasgupta et all. (2006, p. 284) suggest the following 3 approaches, which can be used to tackle NPC problems:

- Development of algorithms that have been designed to solve cases of the given problem, for example by using the concepts backtracking and pruning to limit the state space.
- Development of heuristics that can be used enabling the computer system to perform informed decisions at various moments in time.
- Development of approximation algorithms, which can be used to produce solutions that are not necessarily optimal. These come with a guarantee that any solutions found are not much worse than the optimal solution.

Given that my aim is to find optimal solutions, the first two suggestions mentioned above will be of paramount importance to my research and design.

Software Development Approach

I feel that because I know the requirements of my project up-front, and that the primary use of the proposed system is to serve as a research tool to be used by myself, the waterfall approach is the best-suited development style for my project. To ensure that the development of my system is executed as effectively as possible I will be sure to define my requirements carefully and concisely in the early stages of the project, and continue to check my developments against the requirements to ensure the final product fits the initial requirements as best as possible.

I have chosen to develop my program using the programming language Java for the following reasons:

1. As this is going to be a reasonably large project, the portability and manageability of code is going to be a necessity, and therefore an object-oriented approach seems like a logical choice.
2. I have a broader knowledge in Java than any other programming language, as my development in this area formed part of my MSc course.
3. Java has a very large development community. This is very helpful because as a new developer, the availability of support if a major benefit.
4. The online documentation for Java is extremely thorough, which will be of major benefit to me should I require the use any classes of which my knowledge and experience is limited.

To ensure my program is produced to a high standard, I have tried where possible to adhere to McCall's Quality Model described by Phillips (2013), as well as following the coding conventions published by Oracle (1999).

3 Description of Algorithms

For convenience, this part of the design will be divided into 3 sections:

- Solvability
- Solving algorithms
- Heuristic functions

3.1 Solvability

Given an N-Puzzle, my program should first determine whether the puzzle is actually solvable, before attempting to solve it. There are 3 fundamental variables that I will have to consider when researching and designing solvability algorithms:

- Position of the tiles.
- Position of the space (the blank tile may not initially be in its destination position as it was in Sam Loyd's puzzle).
- Dimension of the board- puzzles with sides of an odd length often differ in the way they work to puzzles with sides of an even length. My solvability check will have to take this into consideration.

Ryan (2004) provides the following rules, which he suggests can be utilised to determine whether an N-Puzzle with a given configuration is solvable:

a. *If the grid width is odd, then the number of inversions in a solvable situation is even.*

b. *If the grid width is even, and the blank is on an even row counting from the bottom (second-last, fourth-last etc), then the number of inversions in a solvable situation is odd.*

c. *If the grid width is even, and the blank is on an odd row counting from the bottom (last, third-last, fifth-last etc) then the number of inversions in a solvable situation is even.*

In order to employ these checks, I will need to be able to determine 3 things:

- The parity of the dimension (this is straightforward).
- The parity of the number of the row in which the space is located (this is also straightforward).
- The parity of number of inversions, which Meserve (1953, p. 177) describes as being the parity of the permutation.

Knuth (1998, p11) and Pemmaraju and Skiena (2003, p. 69) convey the notion that in a permutation, an inversion consists of two elements, which are in the wrong order.

For example, Figure 3-1 shows a permutation σ that comprises 5 elements:

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 3 & 4 \end{pmatrix}$$

Figure 3-1: Example of a permutation

This permutation σ consists of 2 inversions:

1. σ(3)=5, σ(4)=3
2. σ(3)=5, σ(5)=4

Walsh (2006) describes how the sum of inversions is the number of swaps of adjacent elements it takes to transform the permutation into the identity permutation.

Walsh (2006) provides the formula shown in Figure 3-2 for finding the sum of permutation inversions,

$$I_k = \sum_{j=1}^{k-1} [\pi_j > \pi_k]$$

where π is a permutation and πx is the x^{th} value in π.

Figure 3-2: Formula for calculating number of permutation inversions

This formula simply counts the number of occurrences where an element j is positioned to the left of an element k, and the element at j is larger than the element at k.

As stated by Ryan (2004), Moving a tile left or right does not change the number of inversions, and therefore doesn't change its polarity, which is obvious considering that the tiles are labelled in numerical order, row by row.

Ryan (2004) also states that moving a tile up or down affects an even number of inversions if the width of the N-Puzzle is odd and effects an odd number of inversions if the width of the N-Puzzle is even. He shows this with an example but below I will prove that these statements hold for all puzzles of odd/even width.

Proof

Let the w be the width of a puzzle, for example, in a 4 x 4 (15) puzzle, w = 4.

Let T(x,y) be the co-ordinates of a tile T where x is the row of the tile and y is the column of the tile and $0<x, y< w$.

Moving a tile at position (x_i, y_i) in an upward direction would affect the following inversions:

- From T(x_i, y_i -1) to T(w, y_i - 1)
- From T(0, y_i) to T(x_i - 1, y_i)

It follows that the number of inversions affected (N) would be:

$N = (w - x_i) + (x_i - 1)$

(Trapa, 2004) states the below parity rules, which are certainly intuitive to the majority of people:

- *even + even = even*
- *even + odd = odd*
- *odd + odd = even*

I will utilize these for the next stage of my proof.

- ***Puzzle of odd width***

Case 1: w is odd, x_i is odd.

- ⇨ $w-x_i$ is even, x_i-1 is even
- ⇨ $N = (w-x_i)+(x_i-1)$ is even
- ⇨ Polarity is always preserved (as if each inversion switches the polarity by 1, an even number of inversions will preserve the initial polarity)

Case 2: w is odd, x_i is even

- ⇨ $w-x_i$ is odd, x_i-1 is odd
- ⇨ $N = (w-x_i)+(x_i-1)$ is even
- ⇨ Polarity is always preserved (as if each inversion switches the polarity by 1, an even number of inversions will preserve the initial polarity)

- ***Puzzle of even width***

Case 1: w is even, x_i is odd.

- ⇨ w is odd, x is even
- ⇨ w-x is even, x-1 is even
- ⇨ $N = (w-x_i)+(x_i-1)$ is odd
- ⇨ Polarity is always changed (As if each inversion switches the polarity by 1, an odd number of inversions will never preserve the initial polarity)

Case 2: w is even, x_i is even.

- ⇨ $w-x_i$ is even, x_i-1 is odd
- ⇨ $N = (w-x_i)+(x_i-1)$ is odd
- ⇨ Polarity is always changed (As if each inversion switches the polarity by 1, an odd number of inversions will never preserve the initial polarity)

Moving $T(x_i, y_i)$ downwards has the same affect on the permutation polarity and can be proven in the same way.

The above means that when dealing with an N-Puzzle of odd dimension, if the puzzle is initially in an even permutation, then it is impossible to reach an odd permutation. Knuth (1998, p. 11) states that “the only permutation with no inversions is the sorted permutation 1 2 … n”, and so by definition, the sorted state is an even permutation. Therefore regarding regular N-Puzzles where the length is of each side is odd, if the parity is odd then an illegal move has

taken place. So only permutations of an even parity can be solved, as stated by Ryan (2004).

This also means that when dealing with a puzzle of even length, every time the blank tile is shifted upwards or downwards the polarity of the permutation is switched from odd to even or vice versa. As Knuth (1998, p. 11) states, the sorted permutation contains no inversions and hence it is even. It is intuitive that every solvable configuration must be reachable from the goal permutation, because the path from the configuration to the solution can simply be retraced to revisit the initial arrangement. So logically, if the blank tile is an even number of rows from its initial position (in the puzzle's sorted state), then the parity of the permutation should also be even. If this is not the case then an illegal move has taken place.

Therefore, the above statements made by Ryan (2004) hold. However, a minor tweak is required to this logic in order for it to work for the N-Puzzle format that I am using. The reason why Ryan (2004) counts the row number of the blank tile from the bottom is because in the final configuration of the puzzle he is trying to reach, the blank is in the bottom row. In the solution state I am trying to reach however, the space is in the top row, so I will count my row number of the space from the top.

3.2 Solving Algorithms

3.2.1 Algorithm Quality Analysis

Before I begin research into specific path finding algorithms, it is necessary to know what I am looking for. As there is going to be a number of algorithms available to me, I will need to know what makes a search algorithm desirable, and what makes it unsatisfactory.

Russell and Norvig (2003, p. 71) and Coppin (2004, pp. 78-80) provide the below criteria which I can utilise to determine the quality of any proposed search algorithms:

- **Completeness**: Whether a search algorithm is guaranteed to return a correct solution (if one exists). Some search algorithms are not complete, and in many cases can search infinitely down an individual branch of a search tree, and never find a solution that exists on an alternative path.
- **Time complexity**: The amount of time the algorithm takes to locate a solution in the search tree respective of the depth of the solution in the search tree and the branching factor of the search tree.
- **Space complexity**: The amount of memory the search algorithm can be expected to require in order to find a solution. Again, this is respective of the depth of the solution in the search tree, and the branching factor of the search tree.
- **Optimality**: Whether the search algorithm in question guarantees to return the best possible solution in the case where multiple possible solutions exist.

Although the complexity calculations are based on the branching factor, for N-Puzzle search trees this is not actually constant. However, these calculations can still be useful to analyse the nature of each algorithm.

In order to limit my scope, enabling me to explore chosen approaches in greater depth, I have chosen to limit my implementation to algorithms which are guaranteed to find the optimal solution only.

Uninformed Search Algorithms

Uninformed searches are often referred to as blind searches, exhaustive searches and brute-force searches. They search for a solution with only the knowledge of what the possible options are, i.e. which operations are legal at each step (Nilsson 1998, p. 131). Therefore, they explore possible solution paths, with no consideration of how far away each node is from a solution.

Breadth-First Search (BFS)

Cormen et al. (2009, p. 594) describe how in a breadth-first search, an initial node is expanded, producing children nodes. Each of these nodes is expanded in turn to produce more children nodes (grandchildren of the initial nodes), and so on (Cormen et al. 2009, p. 594). This means nodes of the search tree will be analysed one level at a time and as a result, the path cost to reach each node within the same depth will be equal.

As the breadth-first search explores the search tree in its entirety, one level at a time, if a solution exists it will eventually be found. Also this guarantees that the solution with the smallest path will be found before solutions of a longer path. Therefore, as stated by Luger and Stubblefield (1997, p. 101) and Korf (1985), BFS is both optimal and complete.

One drawback to this search method is that as the search works depth by depth, all nodes must be saved to memory. This is highlighted by Korf (1985), where he states that this can quickly exhaust the computer's memory, because every node explored must be retained. Russel and Norvig (2003, pp. 73-74) demonstrate how in the worst possible case, the solution will be the last node to be examined in its depth, meaning that all other nodes of that depth will first be expanded, and their children will be saved to the open list before the solution node is recognised as shown in Figure 3-3.

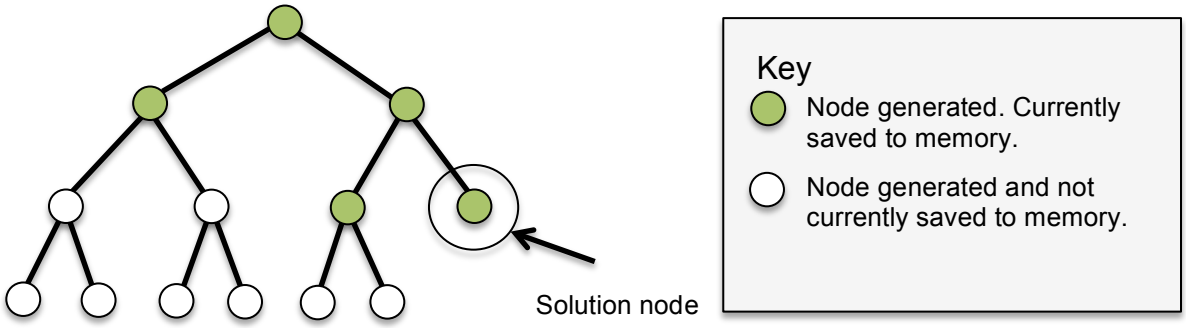


Figure 3-3: Worst-case scenario for BFS

Therefore, both the space and time complexity of the breadth-first search is $O(b^{d+1})$, where b is the branching factor and d is the depth of the shallowest solution (Russell and Norvig 2003, p. 74). Despite this rather large drawback, I have decided to include this search algorithm in my program, as I think it is an interesting approach to analyse.

Depth First Search (DFS)

Depth first searches explore the state space one path at a time. The algorithm works by always expanding the node deepest in the search tree whose children haven't been generated (Russell and Norvig 2003, p. 75). This means that the search will only change the path that it is investigating when a dead end is reached. These dead ends can occur in state spaces when either there are no possible executable operations under the current state, or as mentioned by Dasgupta et al. (2006, p. 284), when we know from a fragment of the solution that we can reject that solution. In our case, if a solution is reachable from the initial state then it is also reachable from all descendants of this state. Therefore the state space of the N-Puzzle search tree would not really contain any dead ends. This is perhaps a strong reason why a depth first search isn't the most sensible approach when the search tree is extremely deep, or worse, infinite.

As observed by Nilsson (1998, p. 135), often a very large number of nodes will have to be explored to find a relatively shallow solution. For example, the puzzle may be solvable in 1 move. If this move isn't the first move to be explored then at least 1 complete path must be generated before the 1-move solution is looked at.

A benefit of this technique however, is that the memory exhaustion that can easily occur during a breadth first search is no longer as much of an issue, because only one path needs to be stored into memory at any one time. Therefore, the memory required for executing depth first search is relatively minimalistic. In fact, Russell and Norvig (2003, p. 75) determine the space complexity for a depth-first search as being linear- O(bm) where b is the branching factor and m is the maximum depth of the search tree. This is because in the worst-case scenario, every node of the longest possible branch would have to be stored along with the children of these nodes (Russell and Norvig 2003, p. 75). Korf (1985) explains that in the worst case, the time complexity of a depth first search is $O(b^m)$ because as shown in Figure 3-4 the maximum depth of every branch would have to be explored before a solution is found.

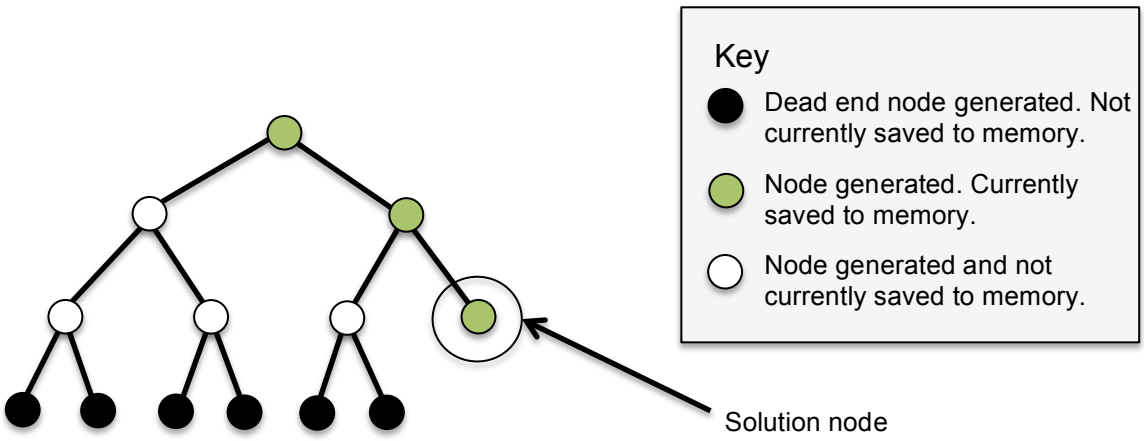


Figure 3-4: Example of worst-case time complexity for DFS

Obviously, for search trees with infinitely long paths, both the space complexity and the time complexity would be infinite.

Memory requirements can be further reduced by utilizing the concept of backtracking (Cormen et al. 2009, p. 603). Backtracking is an alternative method of dealing with dead ends in a depth-first search. A basic depth first search, as described above will search the state space one path at a time. Each time a path is examined and the nodes that form that path are saved into memory, along with their children. When a dead end is hit, the deepest unexplored child of the path can then be examined. However, Russell and Norvig (2003, p. 75) state that backtracking enables the search to reverse out of the current dead end so that other paths and branches can be explored. Therefore, when exploring a particular path, only the nodes that make up that path need to be stored (Russell and Norvig 2003, p. 76). So children of these nodes that do not form part of the path can be discarded, as backtracking can find these at a later point when required.

Backtracking has the ability to reduce the space complexity of a depth-first search to O(m) where m is the maximum depth of the state space (Russell and Norvig 2003, p. 76).

As only one branch is examined at a time, and the state space isn't explored entirely a level at a time, unlike BFS, DFS is not optimal, as confirmed by Luger and Stubblefield (1997, p. 103). Technically, neither is DFS complete, as it is possible for a never-ending branch to be explored infinitely without the algorithm finding the solution on an alternate branch. Because DFS is neither optimal, nor is it complete, I have chosen not to include it in my final designs.

Depth-Limited Depth-First Search (Depth-Limited DFS)

One major issue with depth-first search as demonstrated previously is that for puzzles that have infinitely large state space, the search could potentially execute forever and never find a solution. A depth bound can be enforced to terminate the indefinite exploration of these vast search paths (Nilsson 1998, p. 133). The pruning of these branches transforms the search technique from a depth-first search into what Russell and Norvig (2003, p. 77) refer to as a "depth-limited" search. Personally I feel the name depth-limited depth-first search is a more fitting name for this technique, as the phrase "depth-limited" offers no indication of which order the nodes in the search tree are explored.

As the state space is no longer affected by very long branches, the depth limit reduces the space complexity to O(bl) where b is the branching factor and l is the depth limit (Russell and Norvig 2003, p. 77). The time complexity is also bounded at $O(b^l)$ (Russell and Norvig 2003, p. 77). The space complexity is of the worst case situation is shown in Figure 3-5.

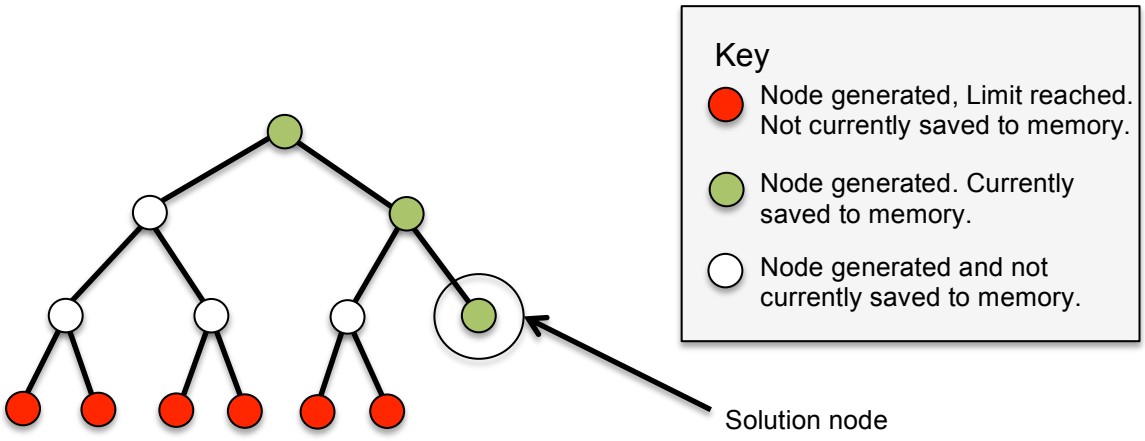


Figure 3-5: example of worst-case time complexity for depth-limited DFS

As with the regular depth-first search, backtracking has a positive effect on this search technique, reducing the space complexity to O(l) where l is the depth limit. This search is not optimal for the exact same reasons why the regular depth-first search is not optimal. As Russell and Norvig (2003, p. 77) point out, depth-limited DFS algorithms are only complete if the depth limit is large enough so that the state space contains the solution. As this search is not optimal, I have decided that it is not suited to my project.

Depth First Search with Iterative Deepening (IDS)

The regular depth first search route is neither complete, nor is it optimal, and it can potentially run indefinitely. By limiting this search technique with a bound, one can restrict the search to exploring a finite space. However, this does not make the search optimal, neither does it make it complete. Korf (1985, p. 100) introduces a clever solution- to perform a series of depth-limited depth-first searches, starting at depth 1, and each time increasing the depth limit by one every time the search completes, until the solution is found.

As the depth is increased by one every iteration, this search effectively mimics the breadth first search, without the drawback of having to save the entire explored state space into memory. We no longer have the problem of choosing a sufficient depth limit as we did with the depth-limited search. It is therefore, as expressed by Korf (1985, p. 100), both optimal and complete.

As with a regular DFS, only one branch needs to be saved at a time. Accordingly, its space complexity is linear- O(bd) (Russell and Norvig 2003, p. 78) where b is the branching factor and d is the depth of the shallowest solution. The time complexity of this search is $O(b^d)$ as it still needs to explore most of the same nodes as BFS. IDS only expands nodes up to the depth of the solution, which is why its time complexity is lower than that of BFS, which often generates children whose depth are one higher than the depth of the optimal solution (Russell and Norvig 2003, p. 78).

As this search technique is complete, optimal, and has a linear space complexity, it cleverly combines the pros of BFS and DFS, making it in my opinion superior to both methods. This is especially beneficial when the solution is deep into substantial state space. Given the many advantages of this approachI have chosen to include IDS in my program for further research.

Uniform Cost Search (Branch and Bound)

Nilsson (1998, p. 133) describes how this uses a similar concept as BFS but with one major exception. Instead of expanding nodes in order of their depth, it focuses on expanding nodes in order of the cost taken to reach that node. The uniform cost search will first explore paths, which have the lowest overall cost (Nilsson 1998, p. 133). (Russell and Norvig 2003, p. 75) and (Nilsson 1998, p. 133) concur that the uniform-cost search is both optimal and complete if no nodes have a lower cost than their parent.

Both the time complexity and the space complexity of the uniform cost search are $O(b^{1+[c^*/\varepsilon]})$, where c* is the total path cost for the shallowest solution, b is the branching factor and ε is the minimum operation cost (Russell and Norvig 2003, p. 75). This complexity expression can be derived from looking at the BFS complexity. BFS has the time and space complexity $O(b^{1+d})$ where b is the branching factor and d is the depth of the shallowest solution. With uniform-cost search, the depth of the shallowest solution is not necessarily the cost of the path to that solution. c^*/ε is a way of finding the maximum depth of a solution from knowing the cost of reaching the solution, as ε is the minimum cost of each operator required to reach the target state.

As I will be focusing my research on the N-Puzzle, the cost of every move will be equal to 1. Therefore, the cost of reaching each node will be equal to the depth of that node, and so, as pointed out by Nilsson (1998, p. 133), for this situation both BFS and the uniform cost search will yield the same results in the same manner. Russell and Norvig (2003, p. 75) confirm this, pointing out that BFS is equal to the uniform cost search if we equate the cost of a node to its depth in the search tree. Note that $O(b^{1+[c^*/\varepsilon]}) = O(b^{1+d})$ where $\varepsilon = 1$ and $c^* = d$.

Dijkstra's algorithm is very similar to the uniform-cost search. In fact, Felner (2011) describes them as being "logically equivalent". However similar these two techniques appear, they are not identical, and this seems to be a very common misunderstanding among a large number of sources. The main difference between the two is that Dijkstra's algorithm requires an explicit declaration of all vertices upfront, whereas the uniform cost search requires

only the initial node, and it unravels the state space as required (Felner 2011). As a direct result of this, not only is the time complexity greater for Dijkstra's method, but also the memory consumption of Dijkstra's algorithm at any given time will be lower than that of the Uniform search (Felner 2011).

3.2.2 Informed Search Algorithms

Unlike uninformed searches, informed searches do not choose the next node to explore blindly. Instead, informed approaches predict how far away each node is away from the solution, thus enabling us to choose more accurately which nodes are more promising, and therefore which nodes we should expand first (Coppin 2004, p. 91). If we knew exactly how far away every possible tile arrangement was from the target configuration (i.e. how many moves each tile arrangement would take to complete), then we would be able to choose the optimum path outright. As this is often impossible, heuristics can help us to estimate how promising a specific state is.

I will discuss the use of various heuristics later in this report. However, it is largely beneficial to first explore the search algorithms that can facilitate such a feature.

Hill Climbing

Hill climbing can be used to solve problems by taking a specific state, and iteratively improving it until a solution is reached (Michalewicz and Fogel 2004, p. 43). For a given state, the possible operations are assessed. If an operation has the ability to improve the state the puzzle is in, then it is executed and then the same process is carried out on the resulting node. Coppin (2004, p. 99) describes this as being similar to DFS but with a heuristic to guide the search down a more direct route.

Clearly this is not always as straightforward as it initially appears to be. For example, for a given state, there may exist more than one operation that can improve the current situation. How should the algorithm best choose which of these operations to execute?

For precisely this situation, there exist multiple variations of the hill-climbing search.

In regular hill climbing or first choice hill climbing (Russel and Norvig 2003, p. 113), the very first child that improves the expanded node is selected (Coppin 2004, p. 98). Stochastic hill climbing chooses a random child to pursue from the list of available child nodes that are better than the current node (Russel and Norvig 2003, p. 113). Steepest ascent hill climbing on the other hand expands all children of the current node and selects the child that is most promising, and therefore theoretically attempts to take an even more direct route to the goal (Coppin 2004, p. 98).

The following are some of the additional issues associated with hill climbing:

- Ridges exist in a state space when sequences of operations improve the current position, until any possible operation will worsen the situation- however the best possible state has not yet been reached (Russel and Norvig 2003, p113). This is described by Russel and Norvig (2003, p113) as being a “local maximum”. At this point, the current node is in a more desirable state than its parent or any of its children.
- Similarly, Russel and Norvig (2003, p113) describe what many computer scientists know as a plateau. This is a section of a state

space where the heuristic for each surrounding node is equal to the current node.

If these issues are not handled correctly, they can cause termination of the search.

Hill climbing searches tend to halt when they reach a local maximum and therefore do not always locate the global maximum (Michalewicz and Fogel 2004, p. 44). Therefore, it is not always complete or optimal (Russel and Norvig 2003, p. 113).

For these reasons I have chosen not to implement this search technique into my system.

Best-First Search

In many ways, the best first search uses a similar concept as the uniform-cost search and also the breadth-first search. However, instead of expanding nodes in order of their path costs from low to high, it expands them in order of their distance from the goal from low to high (Michalewicz and Fogel 2004, p. 106). This search technique is also similar to the hill-climbing technique, as it explores what it assumes to be the next best node using an estimation function. However, it has one major difference. Instead of choosing the next node to explore from the previously expanded node's children, it chooses from a list of open nodes (Michalewicz and Fogel 2004, p. 106). Open nodes are nodes that have been generated and have yet to be expanded or goal tested. Nodes are expanded in the order of their estimated distance from a solution. I.e. the smaller the heuristic value for a node, the sooner that node will be explored. As these algorithms make decisions at each step by choosing the path that appears better at the time, they are referred to by Michalewicz and Fogel (2004, p. 87) and Russell and Norvig (2003, p. 95) as "greedy algorithms". Therefore, the best-first search does not consider the cost of the path required to reach a node, it focuses purely on how far each node appears to be from a solution (Wilt et al., 2010, p. 130).

As stated by Michalewicz and Fogel (2004, p. 92), the name "best-first search" can be misleading. We can't possibly know for sure which node is the best node to expand without the use of a database. However, this method offers a way of facilitating an estimate of which node is likely to steer the search towards the goal.

Not only is the best-first search not optimal, but neither is it complete (Russel and Norvig 2003, p. 97). As with the depth first search, there is nothing stopping the best first search from exploring an infinitely long path (Russel and Norvig 2003, p. 97). For these reasons I have chosen not to include the best first search in my implementation.

A Star Search (A*)

The A* algorithm is popular in the fields of path searching, robotics and domain independent planning (Zhou and Hansen 2004). As observed by Russell and Norvig (2003, p. 97), the A* algorithm is a combination of the best-first search and the uniform-cost search. It considers both the (estimated) distance from each node to the goal, and also the cost of reaching that node from the initial state. Therefore, it acts in the same way as a breadth-first search. However, instead of expanding nodes in order of their depth in the search tree, it expands them in order of the estimated cost of the path connecting the initial state, the current node and the goal state. The estimated value of this cost can be calculated by adding the actual cost of the path from the initial node to the current node to the estimated cost of the path from the current node to the goal node (Korf 1985, p. 103).

Russell and Norvig (2003, p. 97) inform us that as long as the function which estimates the distance from any given node to the goal is admissible, then the A* algorithm will return an optimal solution (the admissibility of heuristic functions will be explained later in this document). In fact, Russell and Norvig (2003, pp. 97-101) eloquently prove both the optimality and the completeness of the A* algorithm (dependent on the admissible heuristic).

A* utilises both an open list of nodes yet to be explored, and a closed list of nodes which have already been explored. This enables the search to avoid checking the same node multiple times in cases where it can be reached via various paths (Zhou and Hansen 2004).

The major deficiency of the A* search is the sheer amount of memory it requires to perform searches of high complexity (Russell and Norvig 2003, p. 101). This is due to the number of nodes the algorithm must store as the search is executed. So although the A* search is guaranteed to expand fewer nodes than any other optimal algorithm (Russell and Norvig 2003, p. 101), it must, like the breadth first search, store every node which is generated which mean that its space and time complexities grow exponentially for harder problems.

Russel and Norvig (2003, p. 101) supply the following condition:

$$\mathbf{h(n) - h^*(n) \leq O(\log(h^*(n))}$$

Where:

- h(n) is the heuristic estimate of the distance from node n to the goal state
- h*(n) is the actual distance from node n to the goal state

Unless a heuristic with high enough accuracy such that the above condition holds, then exponential growth can not be avoided, and this results in the consumption of the computer's entire RAM (Russel and Norvig 2003, p101).

There exist two options to overcome this issue. We can either use a variation of A* which doesn't return the optimal solution, but instead returns a sub optimal solution; or we can design heuristics for which the above condition holds- however these are very rarely admissible (Russell and Norvig 2003, p. 101).

For these reasons, A* is a valuable search technique only for problems whose shallowest solution exists within a reasonable depth of the state space. However, given that the A* algorithm expands fewer nodes than any other search method algorithm (Russell and Norvig 2003, p. 101), I will implement this search in my project to investigate how it compares to other algorithms for easy to moderately difficult puzzles.

IDA-Star Search (IDA*)

As suggested by the name of this search technique, it is a combination of the Iterative deepening technique and the A* algorithm.

IDA* search performs a series of iterative-deepening searches. However, rather than applying a depth limit to each iteration, the estimated total path cost is used as a limit (Korf 1985, p. 103). The estimated path cost f(n), as mentioned earlier is:

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

Where:

- g(n) is the actual cost of the path connecting the initial node to node n.
- h(n) is the heuristic estimate of the distance of the path connecting node n to the target node.

As with A*, providing the chosen heuristic method is admissible, IDA* is optimal (Korf 1985, p. 103). Also, because IDA* uses an iterative deepening approach, it is not prone to proceeding down non-optimal paths indefinitely which means that IDA* is complete.

Korf (1985, pp. 103-105) proves that IDA* actually has a similar time complexity as A*, which means that although the iterative deepening nature of the search causes the frequent re-expansion of nodes, it does not have a huge effect on the time required to complete the search. Its space complexity however is considerably lesser than that of the A* search, given that it only requires one path to be stored in memory at any given time (Coppin 2004, p. 132).

As mentioned by Korf(1985, p. 105), IDA* does not require the retention of an open or closed list, meaning that IDA* can be implemented recursively, thus making the algorithm far more convenient to implement.

It is clear that this method could be one of the best methods available to me, so I will definitely build this search method onto my system.

3.3 Heuristic Functions

As mentioned previously, heuristics can be used within search algorithms to provide an estimate to how desirable a situation is.

It is very important to establish that a given problem is NP complete before considering heuristics (Ratner and Warmuth 1986, p. 168). This is because, if an efficient method existed with the ability of solving the problem at hand, any heuristic approaches would be undesirable, because they rely on inaccurate estimations. However, since we know that the N-Puzzle is NP complete, heuristics are a very appropriate approach.

As mentioned previously, there are two main categories for heuristics- admissible and non-admissible. As my objective is to find the optimal solution to a given N-Puzzle, I will be mostly interested in admissible heuristics.

3.3.1 Heuristic Admissibility

An admissible heuristic is what Russell and Norvig (2003, p. 97) describe as an optimistic heuristic, in that it always underestimates the distance from a given node to the target node. The cost of reaching a node is always accurate, as this is just the depth of the tree where the node exists. In order for a path to be followed all the way to the solution in A*, then the total cost of the actual path must be lower than the total estimated cost of all other options available.

If the actual costs of other non-optimal routes are potentially lower than our estimated route costs for them, then we can not guarantee that we have found the lowest possible cost from the initial state to the target state, and that there may exist a cheaper solution route which we haven't yet explored. However, by using an admissible heuristic we have put a lower bound on the actual cost- in the very worst case scenario, potential routes will cost at least their estimate, and so if the cost of our solution is below these then it must be the optimal solution.

3.3.2 Developing Admissible Heuristics

Pearl (1984, p. 119) presents a very attractive model to formalise the problem of creating heuristics to speed up the convergence of our search algorithms. He starts by defining the terminology below to represent preconditions to a valid move (x,y,z):

- ON(x,y): tile x is on cell y
- CLEAR(z): cell z is clear of tiles
- ADJ(y,z): cell y is horizontally or vertically adjacent to cell z

He then defines a valid move as a series of changes of these conditions:

Pre-move conditions (these are the constraints which must be held in order for move(x,y,z) to be an available operation):

- ON (x,y)
- CLEAR (z)
- ADJ (y,z)

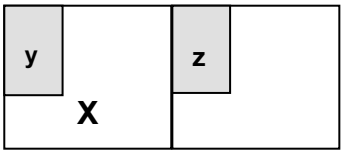


Figure 3-6: Example of pre-move conditions

Change of conditions consequence of the move (x,y,z):

- Add to condition list: ON(x,z), CLEAR(y)
- Delete from condition list: ON (x,y), CLEAR (z)

Post-move conditions:

- ON (x,z)
- CLEAR (y)
- ADJ (y,z)

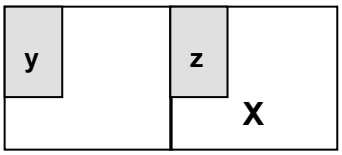


Figure 3-7: Example of post-move conditions

System for creating admissible heuristics

A major obstacle that we face when using any informed search algorithm is the development of a heuristic that is both relatively accurate and admissible. As there isn't a heuristic available which can dynamically calculate the cost of a path joining a node to the solution, the best we can do is to find a heuristic function that can get as close as possible to estimating the actual cost, without ever overestimating it. Now that a formal framework has been set for the N-Puzzle, we can relax aspects of this framework to produce less complicated versions of the N-Puzzle, which are easier to solve in every instance (Pearl 1984, p. 115). If we produce a model that is simple enough, we can invent a heuristic that has the ability to accurately determine the actual cost of the path required to complete the simplified model in every instance. As the simplified model is easier to solve in every possible case, then the heuristic value of any state of this model would be a lower bound on the cost of solving the actual model from an identical state (Pearl 1984, p. 115). Pearl (1984, pp. 118-119) demonstrates how this can be done systematically by removing a constraint from the list of preconditions for a valid MOVE(x,y,z). Relaxing alternative constraints produces different relaxed models, and each different relaxed model can be used to derive an admissible heuristic for the actual model.

3.3.3 Existing Heuristic Functions

Misplaced Tiles

If we remove the constraints ADJ(y, z) and CLEAR(z), then we are left with a heavily simplified model, in which you can move each tile directly to its target location regardless of whether the location is vacant or even adjacent to the tile being moved (Pearl 1984, p. 120). As observed by Pearl (1984, p .120), the number of moves that it will take to solve any configuration of this model will simply be the sum of the number of tiles that are not in their target location, which is why it is known as the misplaced tiles heuristic (Hansson et al. 1985, p. 6).

Relaxed Adjacency

Pearl (1984, p.120) indicates that another relaxed model can be formed by removing the precondition ADJ(y, z) alone. In this model, a tile can be moved into the space, regardless of whether the tile in question is adjacent to the space. The number of moves required to solve any configuration of tiles can again be calculated for this model using a pre-existing algorithm. A well-known swap-sort algorithm exists which does precisely this job. This algorithm is known as the “MaxSort” algorithm, and works by iteratively moving whichever tile belongs in the current location of the space into it (Gaschnig 1979, p. 26). This algorithm is essentially a sorting algorithm, the primary application of which is sorting a list of items into the correct order. However, it can in fact be used as an effective heuristic for the N-Puzzle. As this heuristic is derived from the relaxed model where tiles can move into the space regardless of whether they are adjacent to it, it is called relaxed adjacency (Hansson et al., 1985 p. 5).

For the situation demonstrated in Figure 3-8, the relaxed adjacency heuristic would return 3.

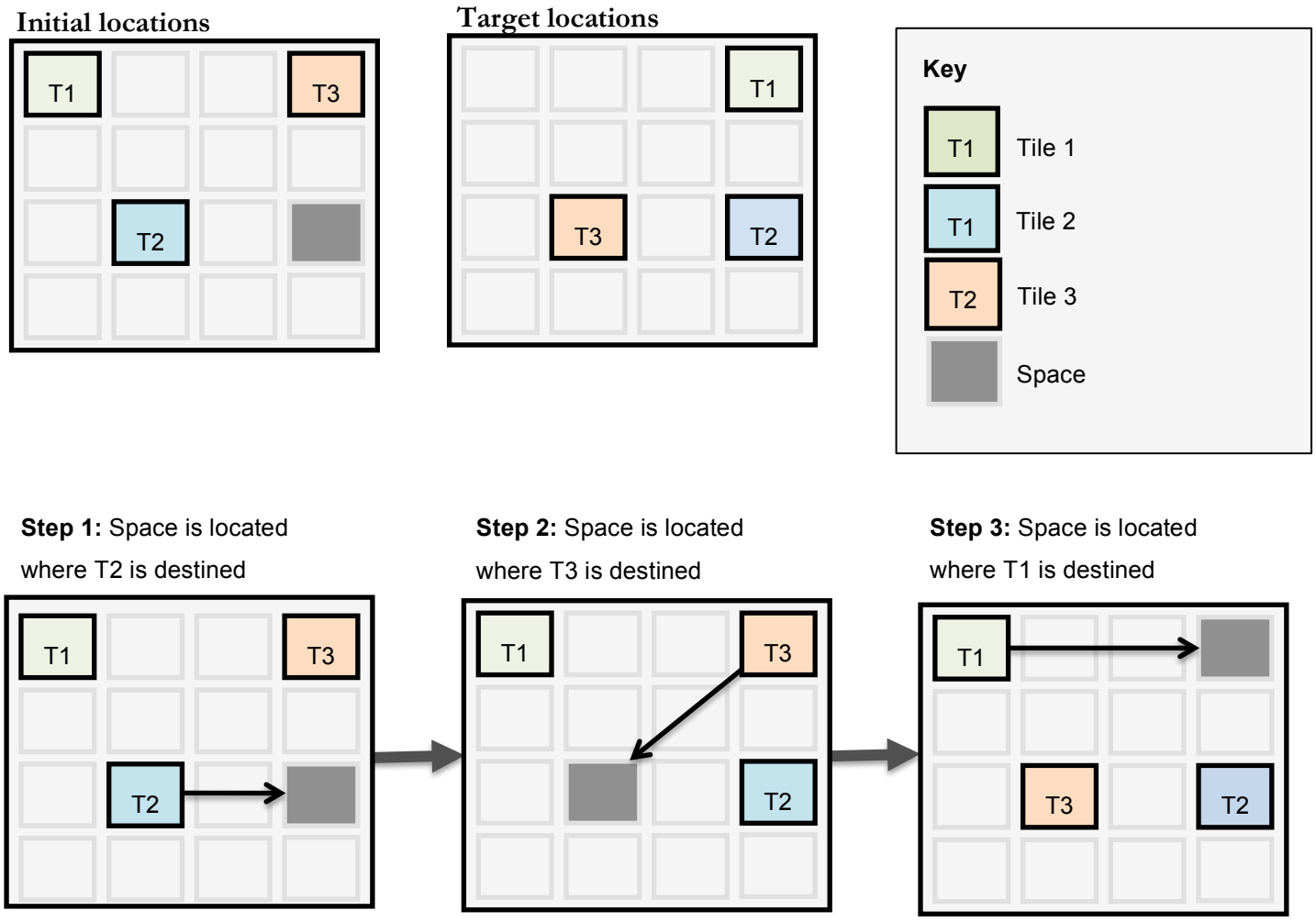


Figure 3-8: Example of relaxed adjacency estimation

Number of Tiles out of Row and Column

We don't necessarily have to remove a constraint completely to simplify the model. In fact, Mostow, J. and Prieditis (2011, p. 705) highlight a heuristic which can be derived by relaxing the ADJ(y,z) constraint of the actual model so that a tile does not have to be completely adjacent to the space in order to move into it, but that it merely needs to be in the same row or column as it. Pearl (1984, p. 120) explains how we can relax the ADJ(y,z) constraint by representing it as a product of two more primitive constraints as shown in Figure 3-9.

$$\mathbf{ADJ(y,z) \Leftrightarrow NEIGHBOUR(y,z) \wedge SAME\text{-}LINE(y,z)}$$

Figure 3-9: Representation of ADJ(y,z) as primitive constraints provided by Pearl (1984, p. 120)

If we relax ADJ(y,z) to just SAME-LINE(y,z), this new model is formed, and a new heuristic can be derived.

This heuristic effectively sums the number of tiles that are in the wrong row with those that are in the wrong column.

Figure 3-10 shows an example of tiles which would add distance onto this heuristic. In this example, tile T1 is in the correct row. With no knowledge other than this fact we know that a minimum of 1 move would be required to move this tile to its target location. Tile T2 on the other hand is in the wrong column and row. With this knowledge, it is clear that at least 2 moves are required to move this tile into the correct location. Therefore, for the above case this heuristic would return 1+2 = 3.

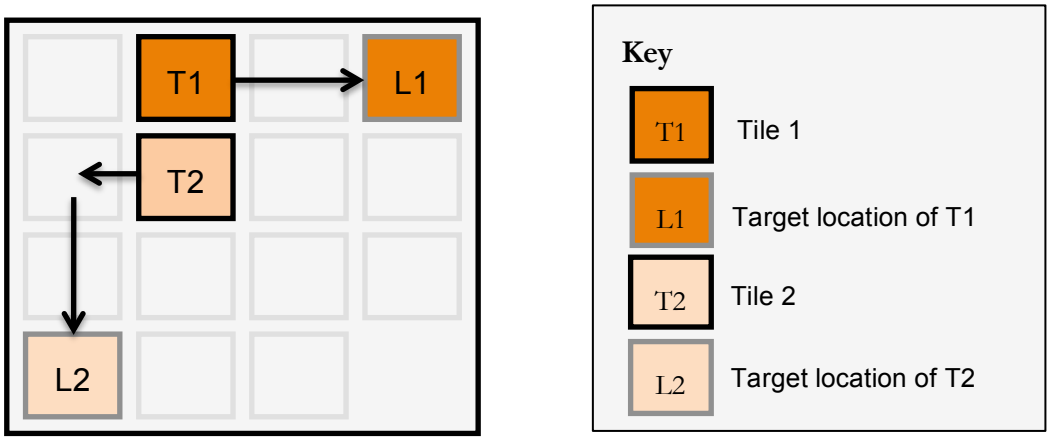


Figure 3-10: Example of a situation where multiple tiles are out of their target positions

Checkerboard Relaxed Adjacency & Checkerboard Misplaced

Instead of removing the precondition ADJ(y,z) completely, we can merely relax this criteria (Hansson et al., 1985 p. 6). The ADJ(y,z) precondition is the enforcement that tiles can only be moved into an adjacent position. Hansson et al. (1985 p. 5) describe a relaxation of this rule, where a location L1 merely has to be an odd number of moves away from tile T1 to allow T1 to move into L1. This creates what is known as the checkerboard relaxed adjacency model. To help visualise this, we can imagine the board to be checkered, creating two disjoint sets of tiles- a black half and a white half. In this model tiles can be moved into any tile in the other half of the board, provided the destination is unoccupied (Hansson et al., 1985 p. 5). For example, if the empty tile is in the black half of the board, then any tile on the white side of the board can be moved into it. This visual representation of the relaxed model is shown in Figure 3-11.

Figure 3-11: Demonstration of how a 15-Puzzle board can be split into two sides

Although this alteration produces a simplified version of the original model, Hansson et al. (1985, p. 6) point out that there does not currently exist an algorithm that solves this calculation optimally.

Consequently we must further simplify this model, by removing the precondition CLEAR(z) also (Hansson et al.1985 p. 7). In this new model, any tile can be moved to the opposite side of the board regardless of where the space is, and multiple tiles are permitted at a single location.

Manhattan Distance

If, as suggested by Pearl (1984, p. 120), we relax the model by removing the precondition CLEAR(z), we are left with a new model in which tiles can be moved adjacently regardless of whether the adjacent position is occupied or not. In this model, multiple tiles can be placed at the same location at the same time. The number of moves needed to solve any instance of this model can be calculated quickly and efficiently using a relatively simple piece of code which sums up the minimum adjacent moves required to move every tile into its target location. This algorithm can be used as an admissible heuristic for the N-Puzzle, and is called the Manhattan Distance (Hansson et al., 1985 p. 5). An example of this approach is shown in Figure 3-12.

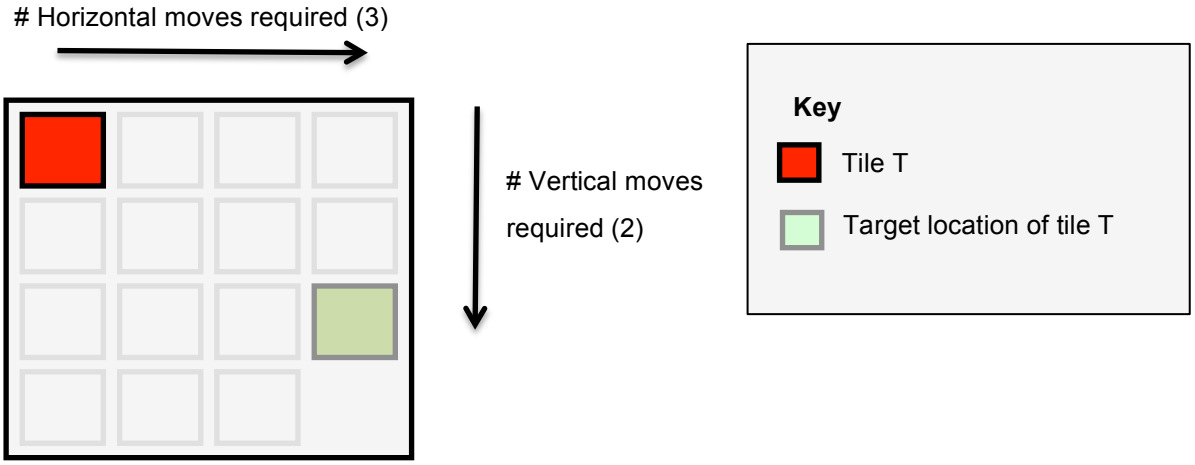


Figure 3-12: Example of the Manhattan distance calculation for a single tile

The Manhattan distance for the situation demonstrated by Figure 3-12 is 5.

Euclidean Distance

The Manhattan distance assumes each tile can move to its destination via a sequence of adjacent moves regardless of the positioning of the space. However, Patel (2013) considers the model each tile can move diagonally through the board to its location. Therefore, to find the cost of solving a given configuration, we can take the total of the straight line distances between all tiles and their target locations. This is called the Euclidean distance (Patel, 2013) and can be calculated using simple Pythagoras:

Euclidean Distance (ED) = $\sqrt{}$ ((# adjacent horizontal moves)^2 + (# adjacent vertical moves)^2)

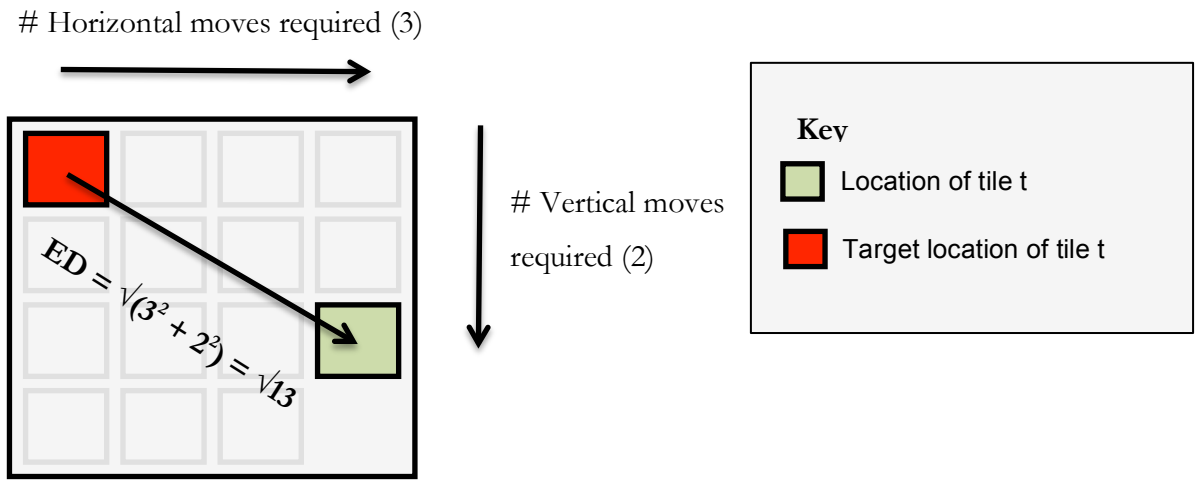


Figure 3-13: Example of the Euclidean distance between a tile and its target position

However, it is common knowledge among mathematicians that the length of one side of a triangle is always shorter than the sum of the remaining two (Tanton 2005, p. 267). Therefore the sum of the number of adjacent horizontal moves and the number of adjacent vertical moves (i.e. the Manhattan distance) will always be larger than the Euclidean distance, making it further from the original model and as a result it is a less accurate heuristic. Given that the actual calculation involved in determining the Euclidean distance of a configuration is no more efficient than that of the Manhattan distance, the Euclidean distance is not worth perusing further. Consequently I will not implement this in my system.

Manhattan Distance + Linear Conflict

There is way in which we can improve the accuracy of the Manhattan distance heuristic, whilst maintaining its strict admissibility (Hansson et al. 1985, p. 11). When two tiles are in the same row as each other, and are in the same row as their target location, but are in the wrong order, this produces a phenomenon which Hansson et al. (1985, p. 11) refer to as “linear conflict”. In order to resolve this situation, one of the tiles must be moved out of that column, before being moved back into that column (on the other side of the other conflicting tile). Therefore, every time this situation occurs it adds a minimum of 2 moves to the Manhattan distance of one of the conflicting tiles. These conflicts occur in columns as they do in rows. An example of a linear conflict is shown in Figure 3-14.

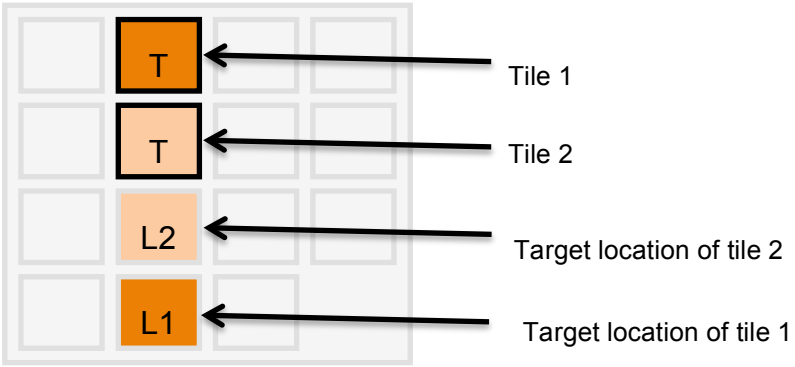


Figure 3-14: Example of tiles in a linear conflict

Tile 1 and Tile 2 are in the correct column, but the incorrect row according to their target locations. The Manhattan distances of tile 1 and tile 2 are 3 and 1 respectively. Tile 2 can be moved straight into its target location without any complications. For tile 1 to reach its location however, either tile 1 or tile 2 must be moved out of the current column to let the other pass, before being moved back into the column. Consequently, in the case of a linear conflict, a minimum of 2 extra moves can be added to the Manhattan distance of one of the two conflicting tiles. When implementing this, it will be important to assure that the 2 extra moves are added to the Manhattan distance of only one of the conflicting tiles of each linear conflict. If two extra moves are added to the Manhattan distance of both offending tiles, this heuristic will no longer be admissible. In the situation described in Figure 3-14, the Manhattan distance + linear conflict heuristic would return a total of 3 + 1 + 2 = 6.

Summary of heuristics so far

If a precondition of some model X is deleted or relaxed in order to create some model Y, then model Y is said to be a relaxation of model X (Hansson et al. 1985, p. 7).

Hansson et al. (1985, p. 8) have established a visual map of how the heuristics discussed previously were formed. Figure 3-15 shows a slightly more detailed version of the figure they produced.

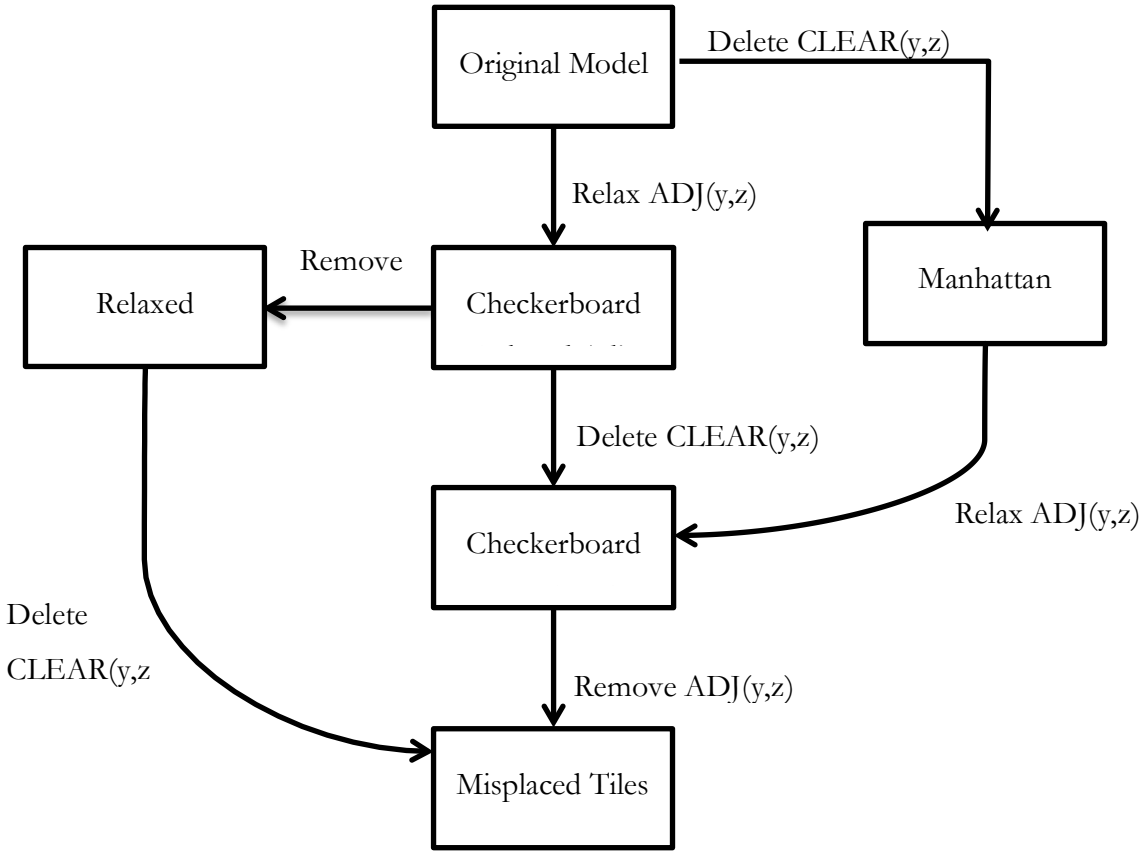


Figure 3-15: Summary of heuristics formed from a series of constraint relaxations

3.3.4 Database Heuristics

Database Lookup

N-Puzzles of width 3 have 9! = 362880 possible permutations. Half of these permutations are insolvable, and are consequently unreachable from any solvable configuration. This leaves only 9!/2 = 181440 solvable permutations of the 8-puzzle. It occurred to me that if I could develop an efficient enough means of representing possible tile arrangements, I could create a database that stores every solvable configuration, along with the minimum number of moves which they would take to solve.

As a result, given a puzzle configuration I could both quickly and efficiently look up how many moves it would take for that configuration to be solved.

The problem is however, that I don't only require the number of moves required to solve each puzzle, I also require the option of knowing the path required to solve the problem. I could theoretically store these in the database also, but would require much more memory than I have available to me.

Database Lookup Heuristic

Using the same concept as in the database lookup previously discussed, I can in fact use the stored knowledge of the optimum number of moves required to complete each configuration as a 100% accurate heuristic. In other words, I could still solve a given puzzle using a regular informed search. However, instead of using a series of guesses in order to find the optimal path to the solution, I could use the database to establish the exact cost from the each configuration to the target configuration for any possible route. As this would be more accurate than using any of the heuristics which I have discussed previously, considerably less nodes would need to be expanded and the search would therefore require less memory. One drawback of this approach is that although the search would require less memory, the database would have to be kept in memory, so the amount of RAM required to solve a solution would not actually be lower in all cases. For example, if the optimal solution was very shallow in the state space, then a regular heuristic

would locate the solution without having to expand many nodes at all, thus not a great deal of memory would be used. A database lookup heuristic however would still require the whole database to be loaded into memory.

Pattern Database Heuristics

Given that there is a limit to how many configurations we can store in a database. What this limit actually is depends entirely on the hardware available. So for larger problems, saving all possible permutations to a database isn't feasible. Be that as it may, there is a way in which larger problems can be addressed despite the limit in hardware capability.

Culberson and Schaeffer (1998, pp. 320-321) discuss a way in which larger problems can be solved despite this limit in hardware capability. Instead of saving the optimum number of moves required to manoeuvre all tiles into their target locations for every possible arrangement, we can simply save the number of moves required to manoeuvre a pre-determined subset of tiles to their target locations. In order to take this approach, first a subset of the puzzle tiles is defined, and its members are defined as being pattern tiles. All other tiles are treated as tiles with no value. Every conceivable permutation of these pattern tiles amongst the remaining blank tiles is regarded as a pattern. The resulting database a "pattern database", which contains a representation of each possible arrangement of the pattern tiles, along with the number of moves required to transform each arrangement into the target arrangement (Culberson and Schaeffer 1998, pp. 320-321).

It can be observed that the full database containing all possible permutations of an N-Puzzle can be considered to be a special case of a pattern database where every tile is a pattern tile.

Because each entry to the database is merely the minimum number of moves required to move the pattern tiles to their target locations, the pattern database can be used as an admissible heuristic for the actual model (Felner et. al. 2004, p. 282). Once a pattern database is constructed, given an N-Puzzle configuration, and a list of pattern tiles, it is possible to convert the configuration into its pattern representation, and look up the corresponding

value in the pattern database for that pattern. This is a major breakthrough in terms of path finding research as it opens the subject up to an extensive avenue of possibilities, and has the potential of avoiding (or at least postponing) problems created by hardware limitations. The number of permutations of n tiles is n!, but the number of ways to pick k unique items from a set of size n is only $P_r^n = \frac{n!}{(n-r)!}$ (Tanton 2005, p. 390). This can be directly applied to our scenario, where the number of tiles is n and the number of pattern tiles is r (counting the space as a pattern tile).

Because we are no longer constrained to saving all n! permutations in order to use an external database, we can use this method to solve puzzles of a larger dimension by limiting the size of our patterns.

There are two distinct types of pattern database- additive and non-additive (Felner, A. et al. 2004).

Non-Additive Patterns

Non- additive databases are the slightly more intuitive of the two pattern database types. Felner, A. et al. (2004) describes non-additive pattern databases as databases that simply store each pattern in the database along with how many moves it takes to return all pattern tiles to their target locations for each pattern arrangement- including movements of non-pattern-tiles.

Because this takes into consideration movements of non-pattern tiles, the movements of tiles in one pattern can interfere with tiles from another pattern. Therefore, in order to combine separate non-additive pattern databases to form a single heuristic, we must take the maximum value of two separate pattern databases (Felner, A. et al. 2004).

This at a first glance seems to be a minor issue. The pattern database can look up pre-evaluated path costs quickly and efficiently, and only enough memory to store $\frac{n!}{p!(n-p)!}$ entries is required. Yet after further analysis of non-additive databases, it quickly becomes apparent why the effectiveness of these as a chosen heuristic can depreciate rapidly for more complicated problems.

Obviously the more tiles which are incorporated into a non-additive pattern, the closer to the actual model the patterns will be and therefore the more accurate the database estimates will be to the actual costs. With relatively small puzzles, for example the 15-puzzle, pattern databases for patterns as large as 7 (8 tiles including the space) tiles can be generated, which covers half of the puzzle. The fact that each pattern can cover a substantial portion of the puzzle board makes each pattern database created reasonably accurate. For larger puzzles however, 7-tile patterns do not cover much of the puzzle board at all, making each pattern database generated inadequate. If each database becomes inaccurate, then taking the maximum of multiple database estimates will become highly ineffective reasonably quickly.

I will implement and test the effectiveness of two non-additive patterns referred to by Culberson and Shaeffer (1998, p. 320) as the “fringe pattern” and the “corner pattern”, shown in Figure 3-16 and Figure 3-17 respectively.

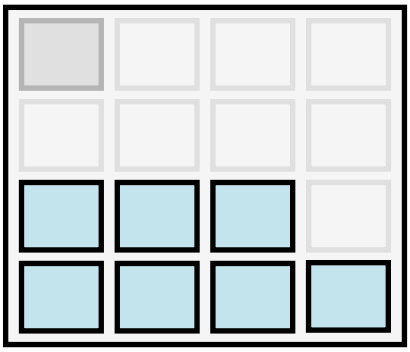


Figure 3-16: Corner pattern

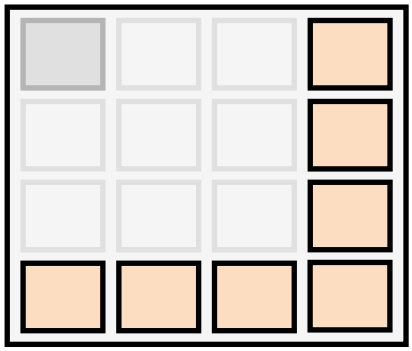


Figure 3-17: Fringe pattern

I will investigate both patterns in isolation, and then I will investigate the effectiveness of using the maximum of both pattern estimates as a heuristic.

Additive Patterns

Korf (2000) describes another kind of pattern database which can be used as an effective alternative which he calls a “disjoint pattern database”. These are also known as “additive pattern databases” (Felner et. al. 2004), the reason why will soon become apparent.

If we generate multiple pattern databases in much the same way as we did for non-additive patterns, but instead of counting the number of moves required to shift the pattern tiles to their target locations, we count the number of moves of pattern tiles only, the pattern databases formed will be disjoint (Korf, 2000). Although this is a little trickier to accomplish, it has major benefits. Unlike non-additive pattern databases, where the combination of multiple databases involved taking the maximum estimate from each, with disjoint pattern databases we can in fact add their results and the outcome will still be admissible (Korf, 2000). The beauty of this approach is that we can effectively cover the whole puzzle board with disjoint patterns and the estimated distance for every pattern can be taken into account by this calculation. Therefore, this is far more effective for larger puzzles. Using multiple additive patterns to form a single heuristic function however requires the patterns to be disjoint (Korf 2000, p. 4), meaning that they do not overlap/ do not have any tiles in common.

I will investigate the effectiveness of two additive patterns described by Felner et. al., (2005, p. 290). These are depicted in Figure 3-18 and Figure 3-19.

Figure 3-18: 5-5-5 pattern

Figure 3-19: 6-6-3 pattern

4 System Design

4.1 Requirements

4.1.1 Functional Requirements

Requirement	Classification	Acceptance Test
User able to create an N-Puzzle of a dimension and initial configuration of his/her choice, which will be loaded to the system and displayed by the GUI.	Essential	• Once the user has optioned to create a new puzzle, the system should capture the required dimension, whether or not the user wants the puzzle in a specific state or not, and if so which configuration the user would like the puzzle set to. • A puzzle will then be created and displayed by the GUI according to the user's specifications.
User able to store a currently loaded puzzle to an external file for later use.	Essential	• Providing there is a puzzle loaded into the system, the user should be able to save the currently loaded puzzle to memory. • Once saved, the puzzle should be stored in an external location as the name specified by the user.
User able to load previously saved puzzle from file to the system, which will be displayed by the GUI.	Essential	• Once the user has optioned to load a puzzle, a dialogue should be displayed displaying all previously saved puzzles. • When the user selects a puzzle to be loaded that puzzle should then be loaded to the system and displayed by the GUI on the main window.
User able to delete previously saved puzzle from file to the system, which will be displayed by the GUI.	Additional	• Once the user has optioned to load a puzzle, a dialogue should be displayed displaying all previously saved puzzles. • If the user selects a puzzle to be deleted and clicks delete, that puzzle should be removed from the external file.
User able to shuffle the configuration manually, or by using an automated shuffle function.	Additional	• Once a puzzle is displayed by the GUI, the user should be able to move puzzle tiles around the puzzle board by clicking on them. • The user should also be able to specify a number of random shuffles for the system to perform. Once the number of shuffles has been specified, the system should randomly shuffle the tiles around the board for the specified number of moves.

User able to solve the loaded puzzle by selecting a search method of his or her choice. Run details should be displayed on the screen.	Essential	• When the user creates a puzzle, specifies the chosen search method and chosen heuristic and clicks solve, the puzzle solver should attempt to solve the puzzle using the methods chosen by the user. The following details of the run should be displayed on screen: o The tile configuration of the puzzle solved o The optimal solution o The number iterations required o The search time o The average time taken per iteration o The time required to load any DB's used o The maximum number of nodes saved to memory at any one time during the search
User able to create and run a batch process which will solve a number of different input puzzles a number of specified times using search algorithms chosen by the user.	Essential	• Once the user has optioned to create a batch, a batch menu should be displayed. • The user should then be able to create and run a batch of his/her chosen size by entering o Details of the chosen puzzles o The way in which he/she would like them to be solved o The number of times each puzzle should be solved • Once the batch has run, the average run details for each puzzle should be saved to an Excel spreadsheet and saved externally.
User able to add time restriction to puzzle solver	Additional	• If the time the puzzle solver has taken calculating the solution exceeds the limit set by the user the solver should terminate and the user should be notified.
User able to cancel the current search being run by the puzzle solver	Additional	• If the user options for the solver currently running to stop, the puzzle solver should terminate and the user should be notified.

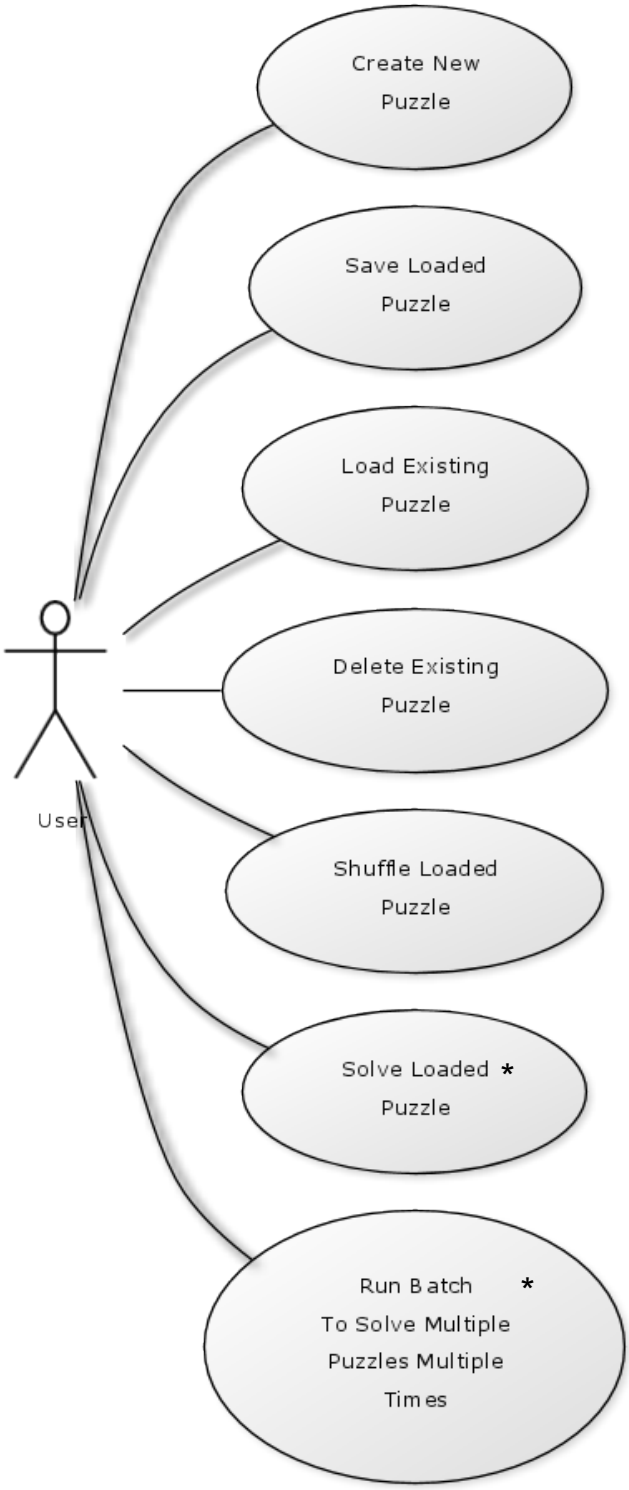
4.1.2 Non-Functional Requirements

Requirement	Description	Classification
Optimal solution should be found 100% of the time (if one exists)	For the purposes of my research I am only interested in search methods that are guaranteed to find the optimal solution to a given puzzle.	Essential
Program memory usage should not exceed memory limitations	I will be using a Macbook Pro with 8GB RAM to run my analysis, so it is wise to keep the memory usage of the program below 5GB. If the memory exceeds this then the situation should be dealt with in the appropriate way.	Essential
Easy to use	As I will be using this program primarily as a research tool, the system does not have to be completely self-explanatory. However, the easier the system is to use, the quicker and easier I will be able to carry out my research.	Additional
Reliable	System should work as expected 100% of the time in order to secure the reliability of any research findings that are based on system output.	Essential
Robust	The majority of obvious input errors should be dealt with appropriately by the system in order to prevent system failures created by user error.	Additional

4.2 Functional Design

4.2.1 Use Case Diagram

Figure 4-1 demonstrates a high level overview of the core functionality of the system from the user's point of view.



* There will be a number of different ways in which a puzzle can be solved. These will be addressed in the "Algorithm design" section of this chapter.

Figure 4-1: Use Case Diagram

4.2.2 GUI Designs and Activity Diagrams

Main Frame of the GUI

Figure 4-2 is a design of the main frame of the GUI. All functionality of the system will be accessible through this frame.

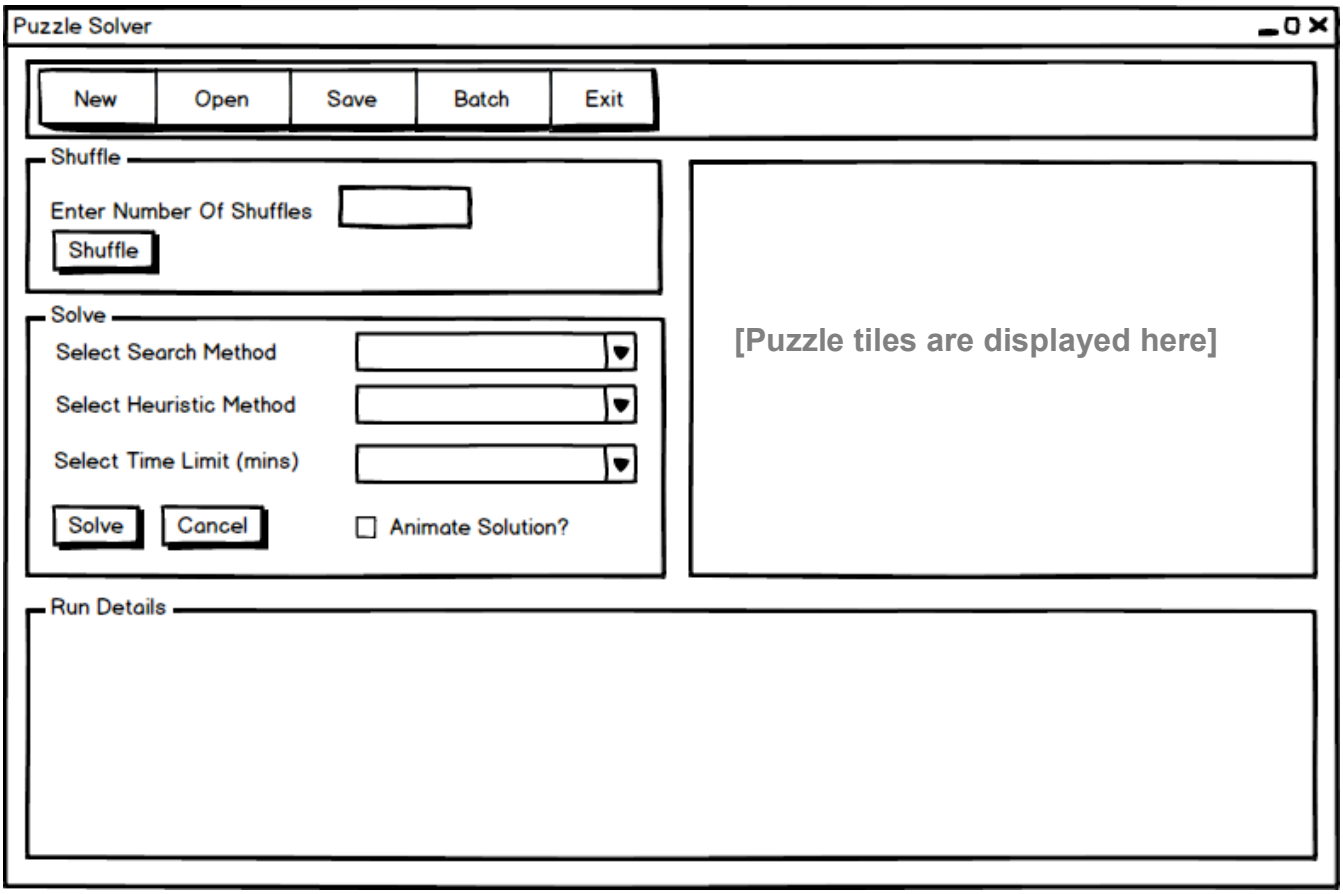


Figure 4-2: GUI design for main frame

Creating a Puzzle

The user must be able to create a puzzle of a specified dimension quickly and efficiently. In various cases the initial configuration of the tiles will need to be pre-set by the user as manually shuffling the tiles in order to reach a desired configuration is not feasible in most situations, therefore the program should cater for this functionality.

Figure 4-3 is an activity diagram showing how the user can interact with the system in order to create a new puzzle.

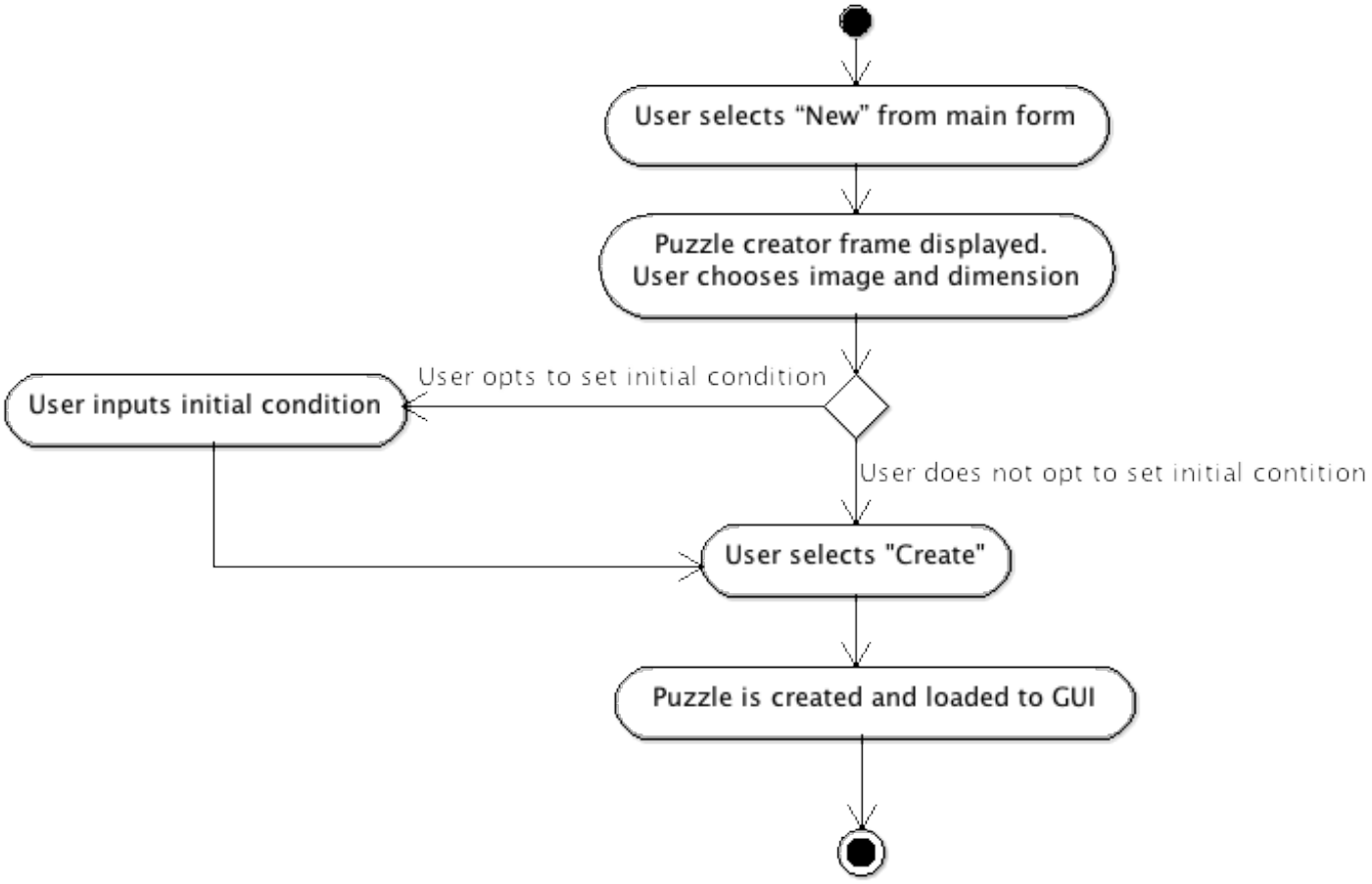


Figure 4-3: Activity diagram for creating a new puzzle

Figure 4-4 is a design of the interface that the user will use to create a new puzzle.

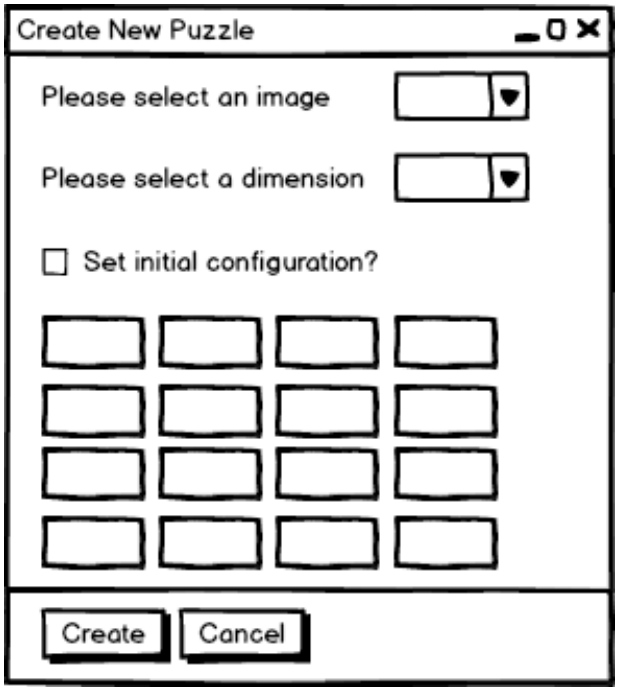


Figure 4-4: GUI design for creating a new puzzle

Loading/ Deleting an Existing Puzzle

The user will be able to load previously saved puzzles, and also delete previously saved puzzles from the same screen. Figure 4-5 shows the way the system can be used to perform either of these functions.

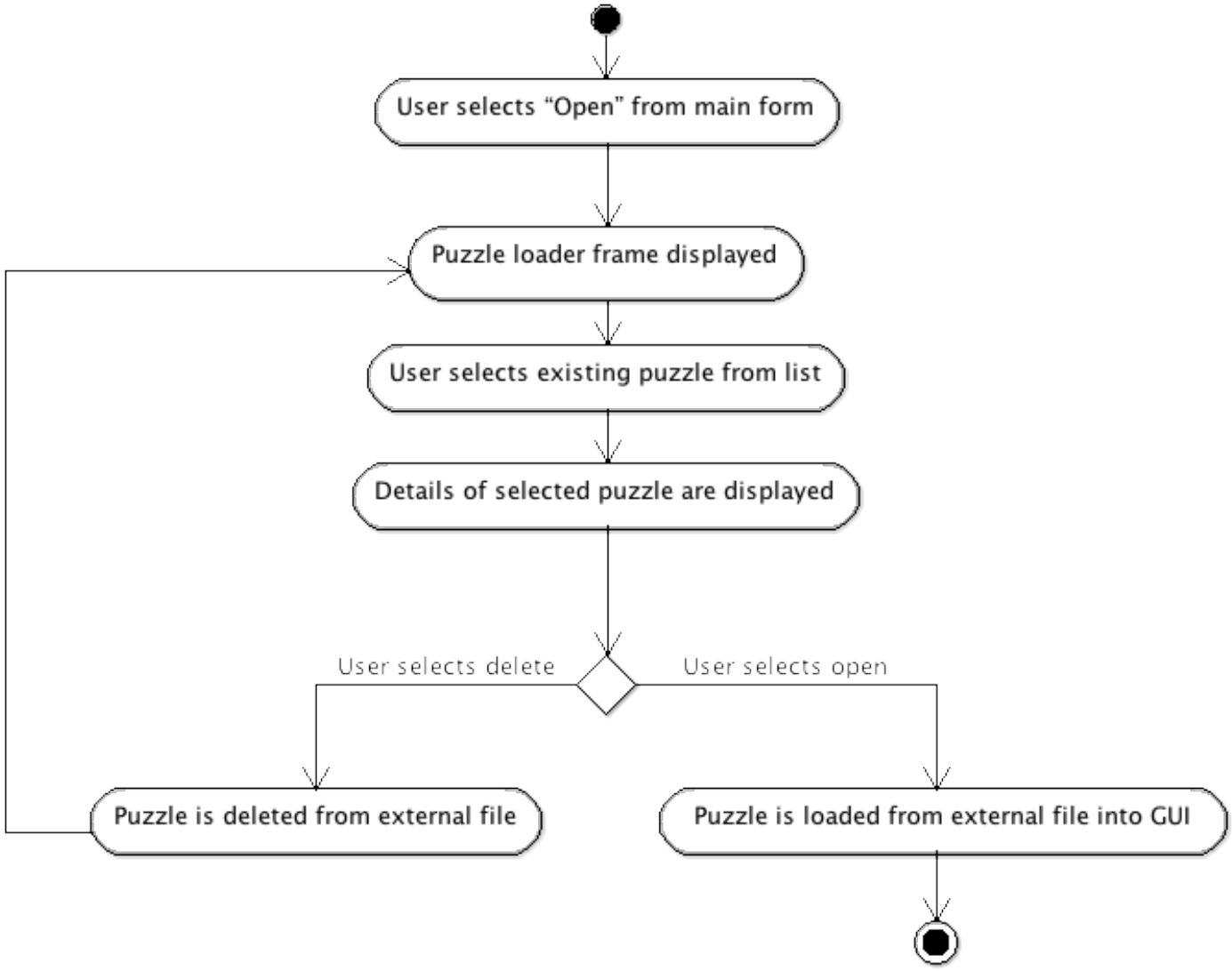


Figure 4-5: Activity diagram for opening/ deleting a previously saved puzzle

Figure 4-6 shows a design of what the window that will allow the user to load and delete previously saved puzzles will look like on-screen.

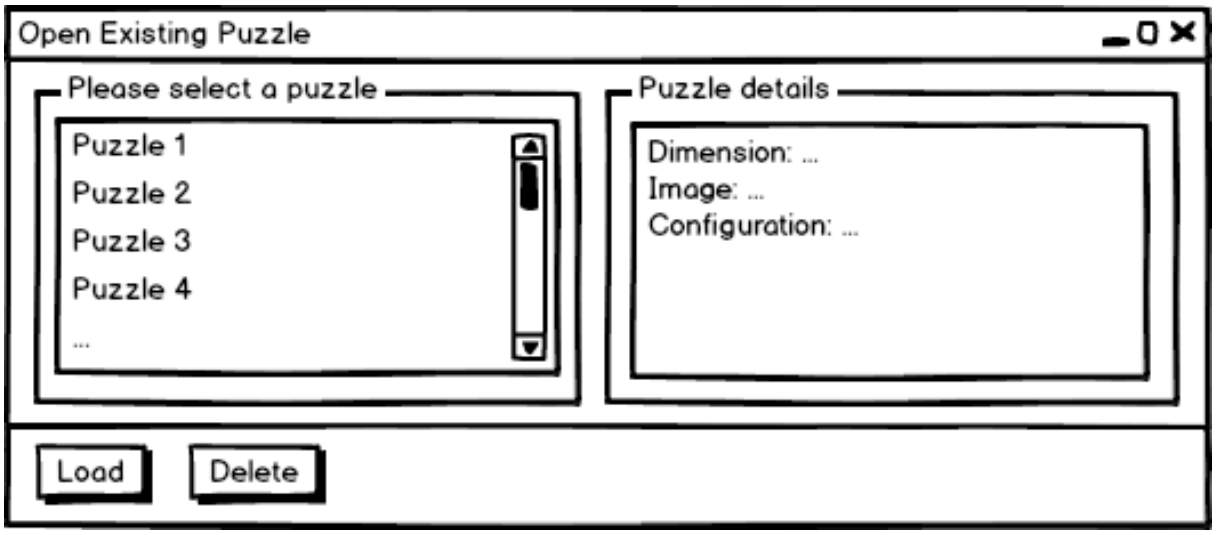


Figure 4-6: GUI design for opening/ deleting a previously saved puzzle

Saving a Loaded Puzzle

Once a puzzle has been created, the user may need to save the puzzle externally for later use. An example of when this may be necessary is when the user needs to carry out multiple tests on the same puzzle. Figure 4-7 demonstrates how a puzzle will be saved.

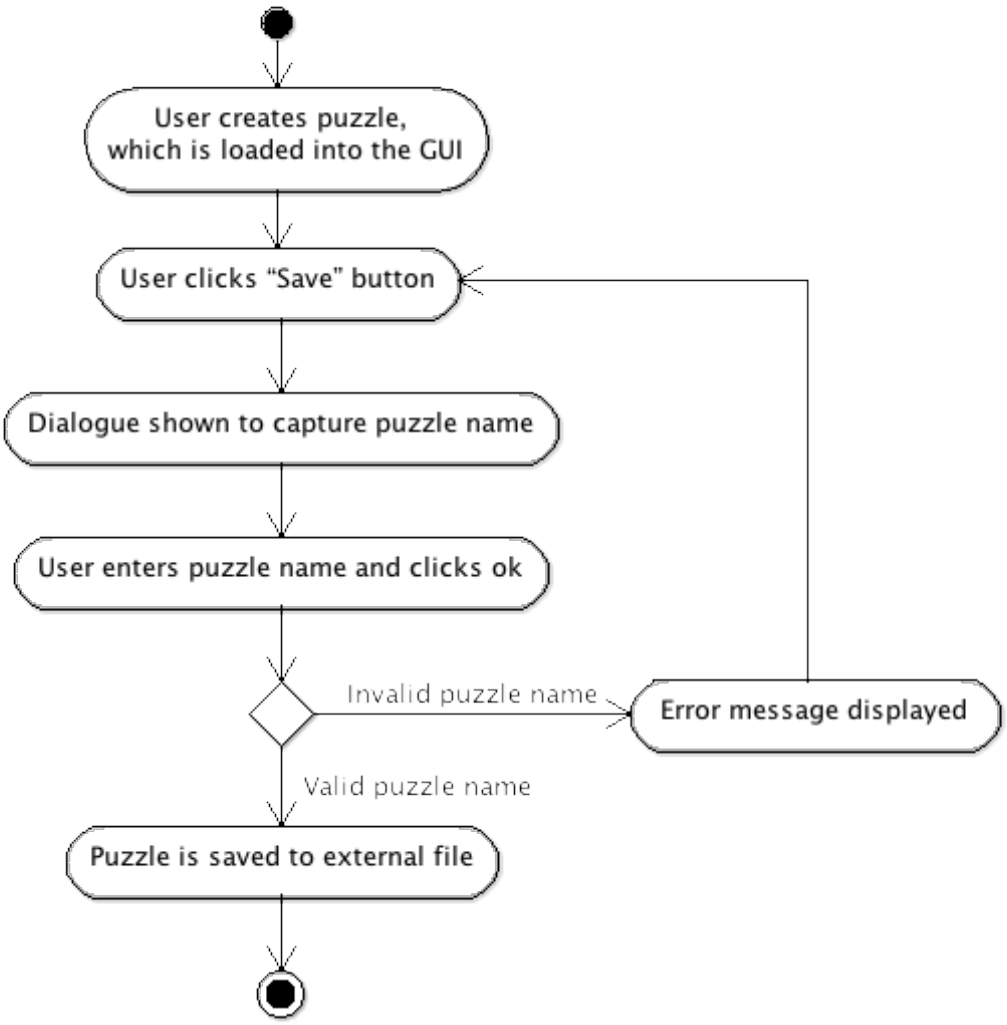


Figure 4-7: Activity diagram for saving a loaded puzzle

Saving a puzzle will be carried out directly from the main screen. Figure 4-8 shows a design of the dialogue box from which the user chooses what the puzzle will be saved as.

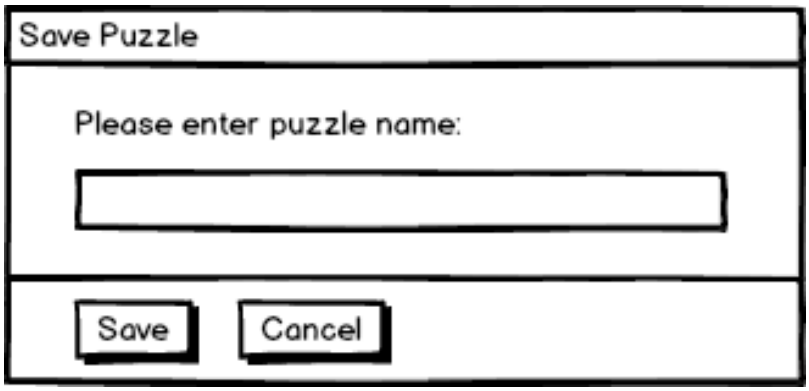


Figure 4-8: GUI design for saving a loaded puzzle

Solving a Loaded Puzzle

The program must be able to solve a puzzle when necessary and display the run details through the GUI. These details include:

- The tile configuration of the puzzle solved
- The optimal solution
- The number of iterations required
- The search time
- The average time taken per node expansion
- The time required to load any database(s) used
- The maximum number of nodes saved to memory at any one time during the search

Figure 4-9 shows how the high level process of solving a puzzle should work from the user's point of view:

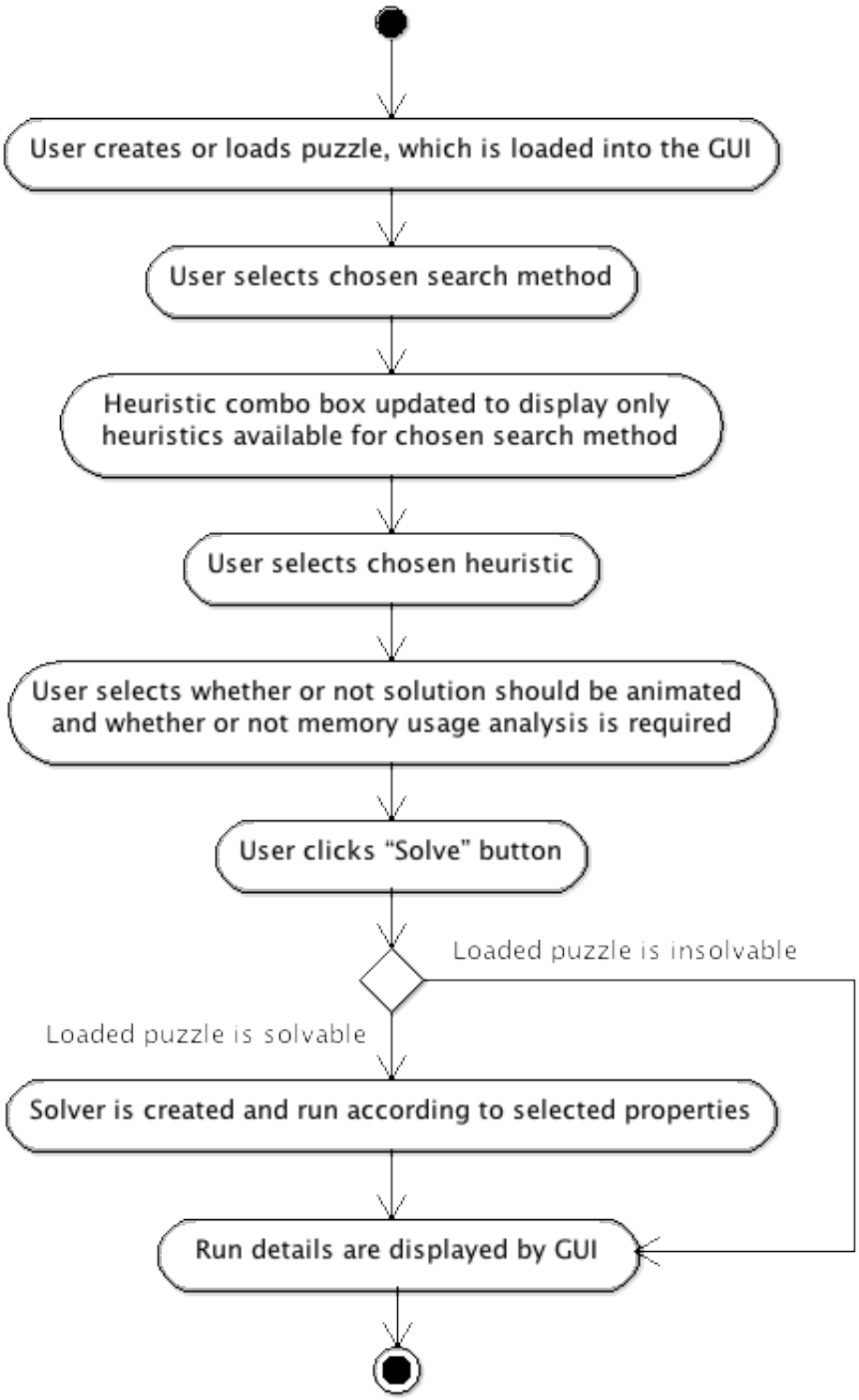


Figure 4-9: Activity diagram for puzzle solving

Running a Batch

Some of the less complex cases of the N-Puzzle will only take a very small amount of time to solve- some less than a millisecond. As a result, testing these puzzles and producing a set of accurate and dependable set of results will not necessarily be straightforward in all cases. Therefore, it would be largely beneficial to the testing of any search algorithms that I add some functionality to the system that allows the user to set up and execute a batch process. Each batch will consist of a number of puzzles. For each puzzle the user must enter the puzzle dimension and configuration, the search method and heuristic that the solver should use, along with the number of runs required for that puzzle. For each puzzle, the average results from all of its runs will be calculated. On completion, these results will be saved to an Excel file.

Figure 4-10 shows the activities involved in running a batch.

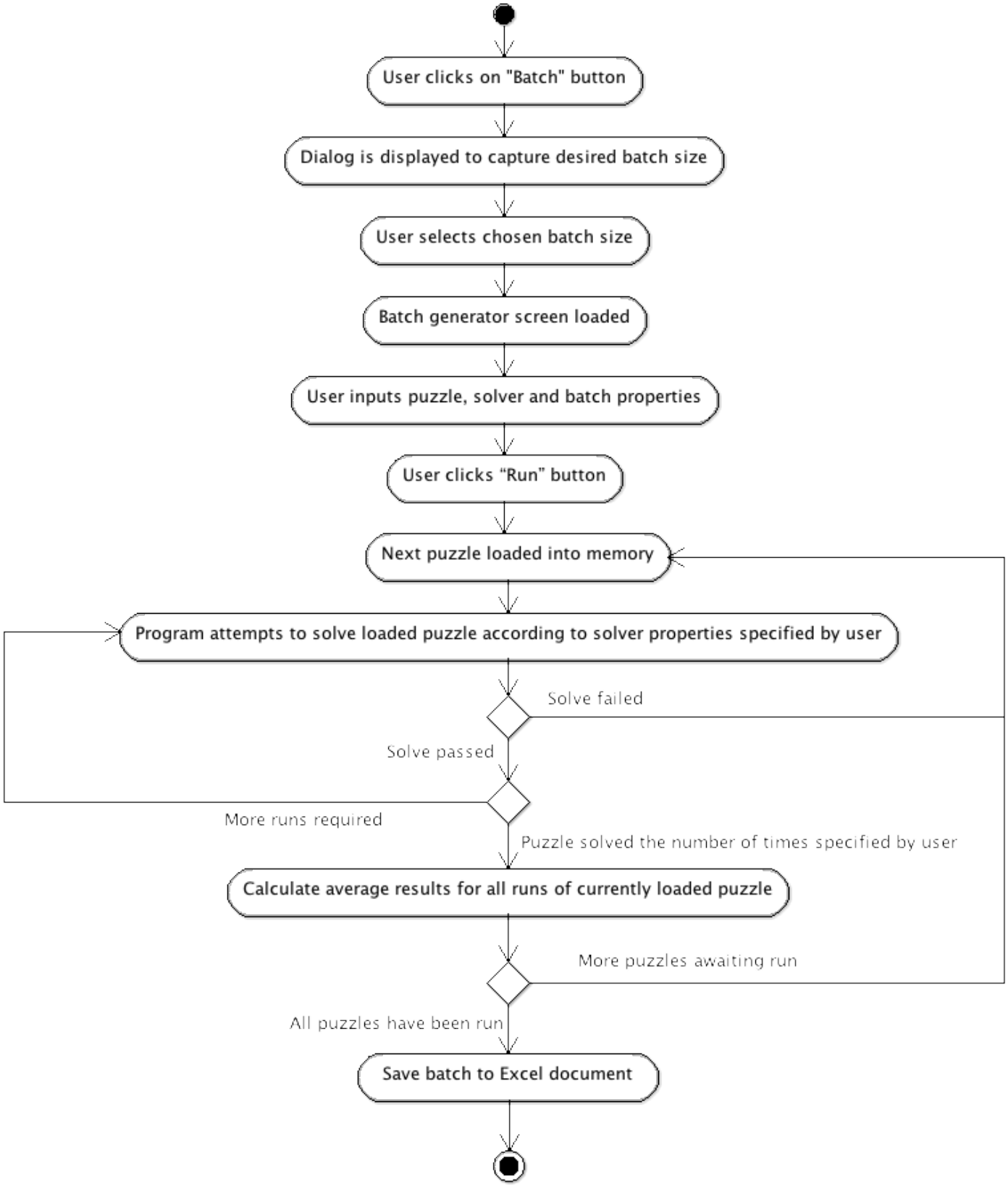


Figure 4-10: Activity diagram for running a batch

Figure 4-11 is a design of the screen, which will be displayed prior to the batch generator screen to capture the number of puzzles the user requires to test.

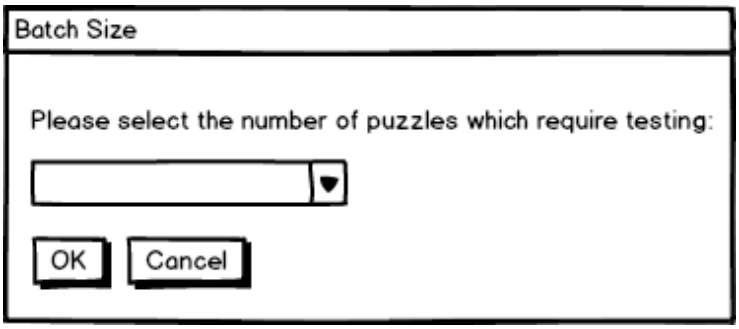


Figure 4-11: GUI design for batch size selection

Figure 4-12 is a design of the batch generator interface:

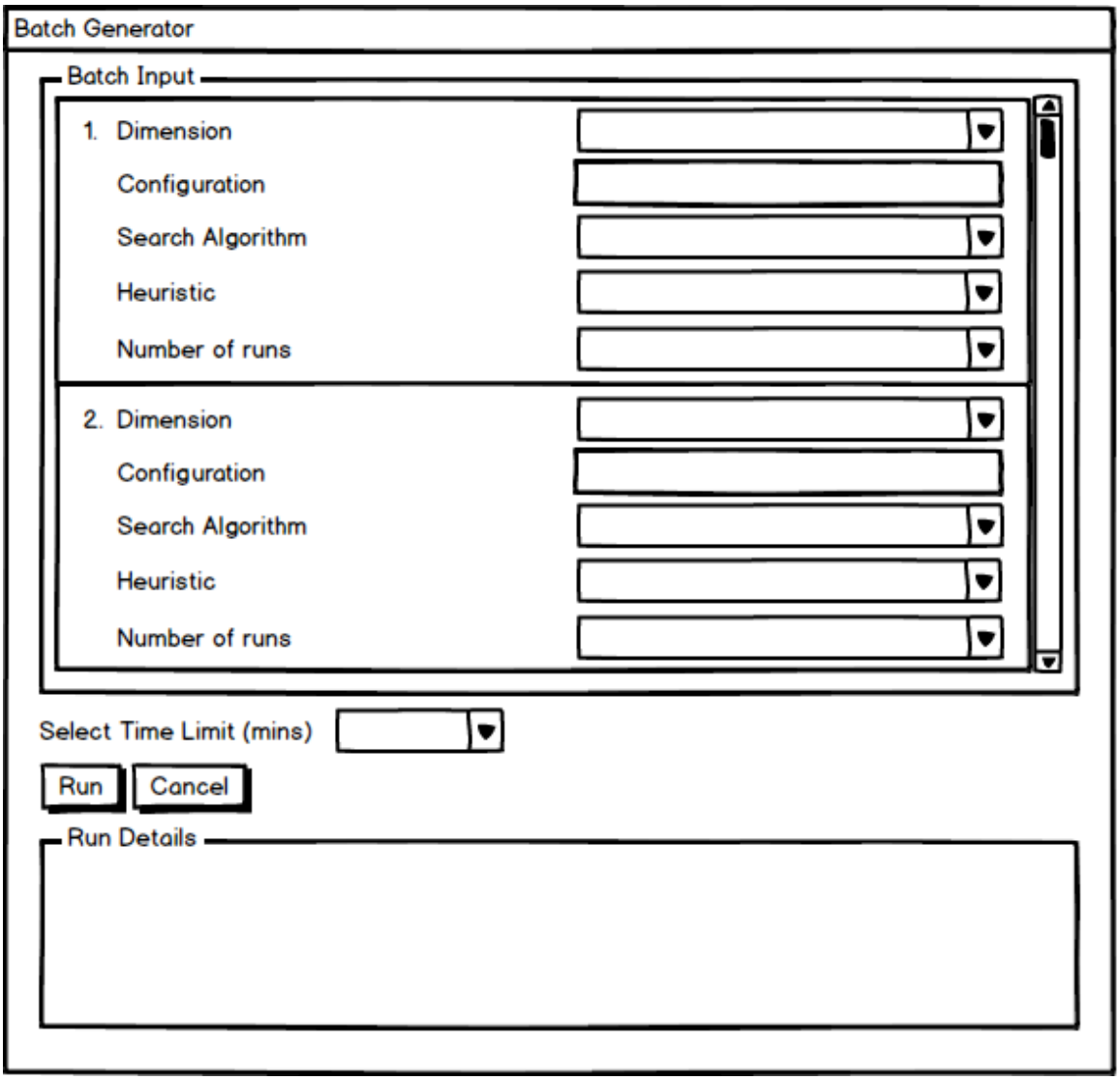
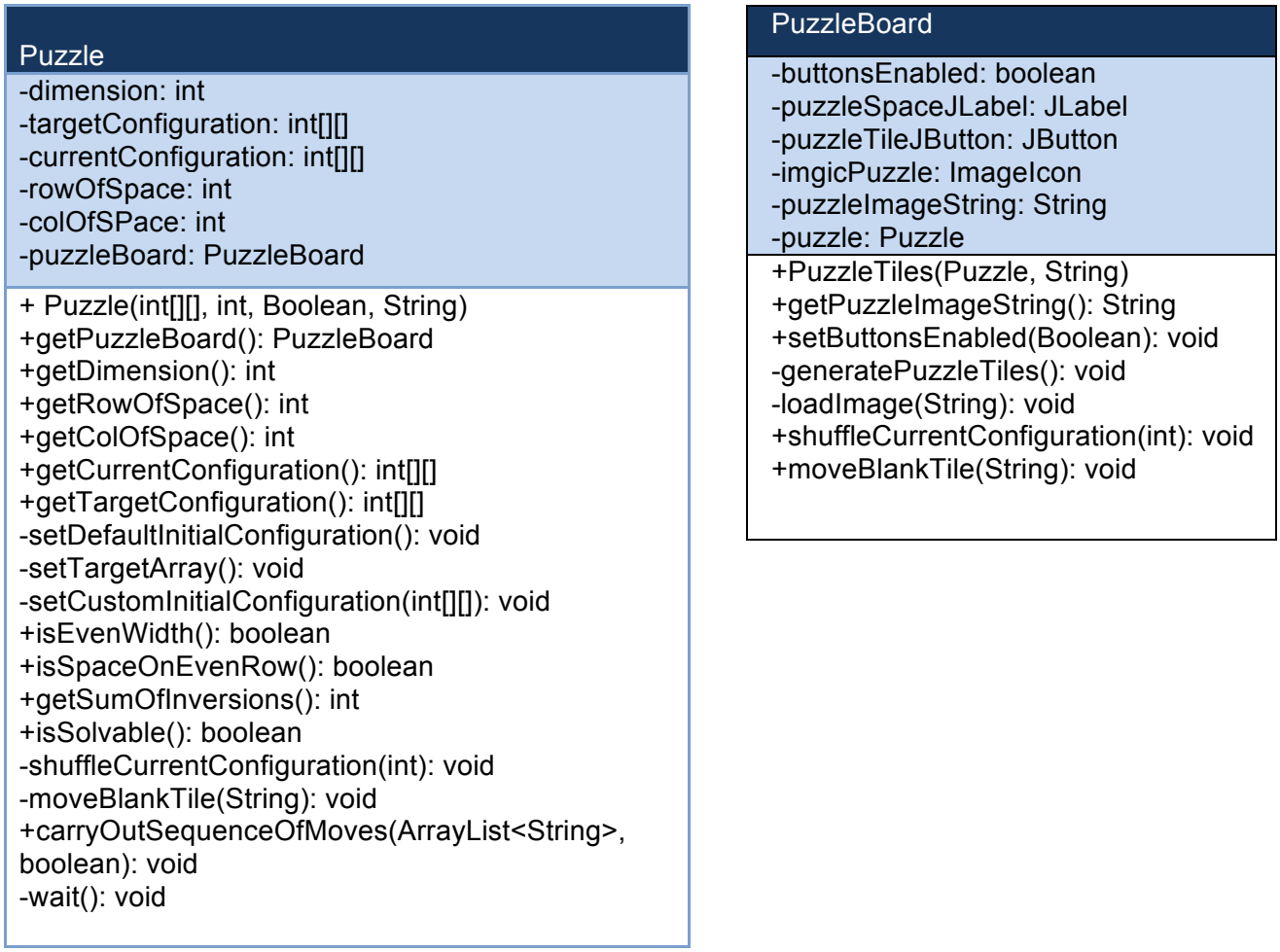


Figure 4-12: GUI design for scheduling a batch

4.2.3 Class Diagrams

In this subsection of the design, I will outline the classes that my program will require.

The puzzle class will be utilised to represent a specific puzzle. It will store information of the current and target states of the tiles in the puzzle. When necessary, a graphical representation of the tiles will be displayed using the puzzle board class.



Once a puzzle has been created of loaded, the PuzzleSolver class can be utilised to solve the loaded puzzle. The chosen search method will perform by creating and analysing a number of configurations. Each of these configurations will be an instance of the Node class.

PuzzleSolver
-openList: ArrayList<Node> -closedList: ArrayList<Node> -currentNode: Node -solutionNode: Node -heuristicMethod: String -searchMethod: String -puzzle: Puzzle -initialConfiguration1D: int[] -patternDatabase1: byte[] -patternDatabase2: byte[] -patternDatabase3: byte[] -patternDatabase4: byte[] -patternDatabase5: byte[] -cancelFlagStatus: boolean -databaseLoadTimeNano: long -searchInitiationTime: long -searchTimeNano: long -numExpansions: long -numberOfNodesInMemory: long -maxNumberOfNodesInMemory: long
+PuzzleSolver(String, String, Puzzle) +getMaxNumberOfNodesInMemory(): long +getHeuristicMethod(): String +getPuzzle(): Puzzle +getInitialConfiguration1D(): int[][] +getSolutionNode(): Node +getSearchInitiationTime +get searchTimeNano (): long +getDatabaseLoadTimeNano(): long +getNumberExpansions: long +getPatternDatabase1(): byte[] +getPatternDatabase2(): byte[] +getPatternDatabase3(): byte[] +getPatternDatabase4(): byte[] +getPatternDatabase5(): byte[] +getPathToSolutionNode(): ArrayList<String> +getCancelFlagStatus(): boolean +setCancelFlagStatus(boolean): void -runRequiredSearch(): void -loadRequiredDatabases():void -aStarSearch(): void -idaStarSearch(): void - fetchSolutionOrNextLimit(Node, double): double -getBestNode(): Node +run(): void

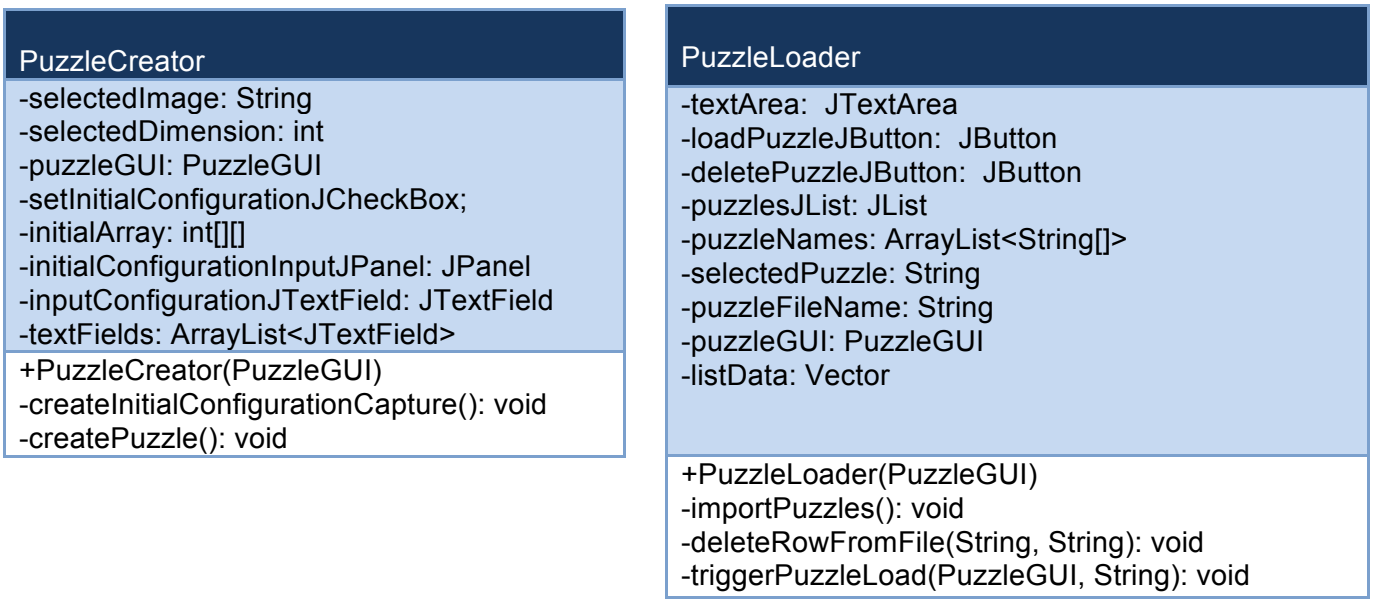
Node
-parentNode: Node -puzzleSolver: PuzzleSolver -nodeConfiguration: int[][] -rowOfSpace: int -colOfSpace: int -fValue: int -gValue: int -hValue: int -subjectOfMove: String
+Node(PuzzleSolver, Node, int[][], int, int, String) +getNodeConfiguration(): int[][] +getFValue(): int +getParentNode(): Node +getSubjectOfMove(): String -getManhattanDistance(): int -getNumberofRowsAndColsOut(): int -getMisplacedDistance(): int -getCheckerboardMisplaced(): int -getLinearConflict(): int -getRelaxedAdjacency(): int -getPatternDatabaseDistance(byte[], int[]): int -configurationToPatternArray(int[][], int[]): int[] -patternArrayToPatternInt(int[],int[]): int -multiplyNumbersBetween(int, int): long +getChildren(): ArrayList<Node>

The PuzzleGUI class will provide the main interface. Through this, individual puzzles can be created and solved, and all other functionality can be accessed. The BatchGenerator class will be responsible for allowing the user to program a schedule of puzzles to be solved successively.

PuzzleGUI
-controlsDisplayJSplitPane: JSplitPane -shuffleJButton: JButton -solveJButton: JButton -cancelJButton: JButton -numberShufflesJTextField: JTextField -animateJCheckBox: JCheckBox -memoryJCheckBox: JCheckBox -outputJTextArea: JTextArea -solvePuzzleDJPanel: JPanel -heuristicsJComboBox: JComboBox -searchMethodJComboBox: JComboBox -timeLimitJComboBoxL JComboBox -puzzleSolver : PuzzleSolver -loadedPuzzle: Puzzle -heuristicMethod: String -searchMethod: String -timeStarted: String -timeEnded: String -solvePuzzleDJLabel -initialConfiguration: int[][] -searchMethodList -heuristicsList0: String[] -heuristicsList1: String[] -heuristicsList2: String[] -heuristicsList3: String[] -selectedTimeLimit: int
+PuzzleGUI() +createPuzzle(String, int[][], int): void +loadExistingPuzzle(String): void -saveLoadedPuzzle(): void -solveLoadedPuzzle(): void -displayRunDetails(): void -resetSelections(): void -lockButtons(): void -unlockButtons: void -updateOutputJTextArea(String, String, Color) -nanoToFormattedTime(long): String -roundDoubleTo3SF(double): double -configurationStringToArray(String, int): int[][]

BatchGenerator
-inputPanel: JPanel -puzzlesJPanel: JPanel -buttonJPanel: JPanel -outputJPanel: JPanel -timeLimitJPanel: JPanel -batchRunJPanel: JPanel -inputOutputJSplitPane: JSplitPane -puzzleGUI: PuzzleGUI -runJButton: JButton -cancelJButton: JButton -timeLimitJComboBox: JComboBox -dimensionJComboBoxes: JComboBox[] -heuristicJComboBoxes: JComboBox[] -searchJComboBoxes: JComboBox[] -numberOfRunsJComboBoxes: JComboBox[] -configurationJTextAreas: JTextArea[] -outputJTextArea: JTextArea -outputJScrollPane: JScrollPane -puzzlesJScrollPane: JScrollPane -selectedDimensions: int[] -selectedNumberOfRuns: int[] -busy: boolean -cancelFlag: Boolean -selectedConfigurations: String[] -selectedHeuristics: String[] -selectedSearches: String[] -batchSize: int -timeLimitJComboBox: int -currentSolver: PuzzleSolver -searchMethodList, -heuristicsList0: String[] -heuristicsList1: String[] -heuristicsList2: String[] -heuristicsList3: String[]
+BatchGenerator(PuzzleGUI, int) -nanoToFormattedTime(long): String -isConfigurationValid(String, int): Boolean -roundDoubleTo3SF(double): double -configurationStringToArray(String, int): int[][] -solvePuzzles(): void -lockButtons(): void -unlockButtons(): void -isValidInput(): Boolean -saveToFile(ArrayList<String[]>)

The PuzzleCreator class and the PuzzleLoader class will provide the user the functionality of creating and loading a new puzzle, as well as loading a previously saved puzzle.



4.2.4 Class Association Diagrams

Figure 4-13 and Figure 4-14 demonstrate the way in which the classes within the proposed system will interact with one another.

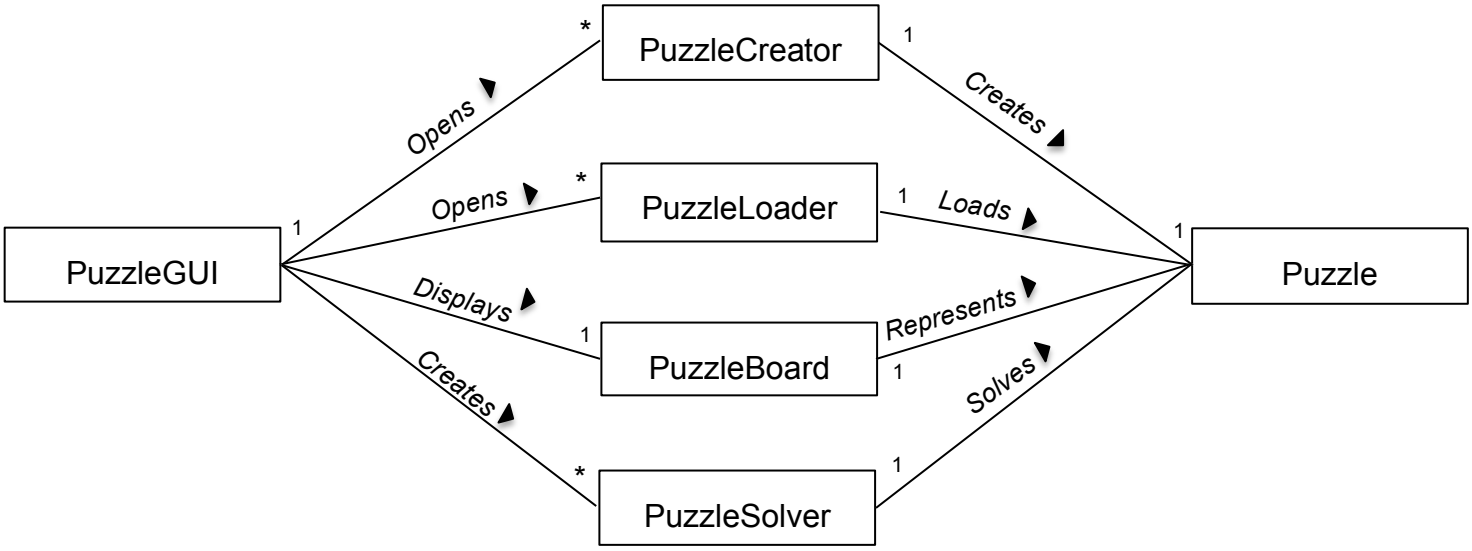


Figure 4-13: Class association diagram A

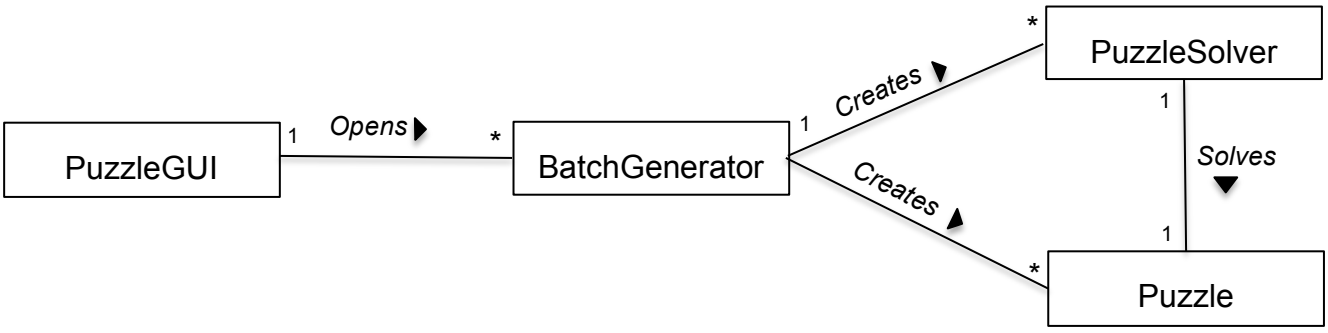


Figure 4-14: Class association diagram B

5 Implementation

5.1 Board Representation

As I will be investigating the solving of N-Puzzles of various dimensions, it will be useful to first develop a generic model, which I will use to define the problem. Any algorithms that I develop which can be used to find solutions for this general model can therefore theoretically be used to find a solution for N-Puzzles of any possible size. In other words, this generalisation will theoretically allow me to use the same core algorithms to solve N-Puzzles of all sizes.

As I will be using computational algorithms to investigate these puzzles, I must be able to represent a puzzle board and the placement of the tiles on that board as mathematical data, which can be represented efficiently within the computer.

A sensible approach to representing a 2-dimensional board of tiles in Java would be the use of a 2-dimensional array.

I will represent each tile with a unique integer between 1 and n, and the space with the integer 0. The target configuration will consist of a board with √(n+1) rows and √(n+1) columns, with the tiles numerically ordered row by row as shown in Figures 5-1 and 5-2.

(0, 0)	. . .	(0, j)	. . .	(0, √(n+1)- 1)
. . .	. . .	. . .	. . .	. . .
(i, 0)	. . .	(i, j)	. . .	(i, √(n+1) -1)
. . .	. . .	. . .	. . .	. . .
(√(n+1) -1, 0)	. . .	(√(n+1) -1, j)	. . .	(√(n+1) -1, √(n+1) -1)

Figure 5-1: Generic N-Puzzle board representation. The row and column of each possible tile position is shown in the format (row, column).

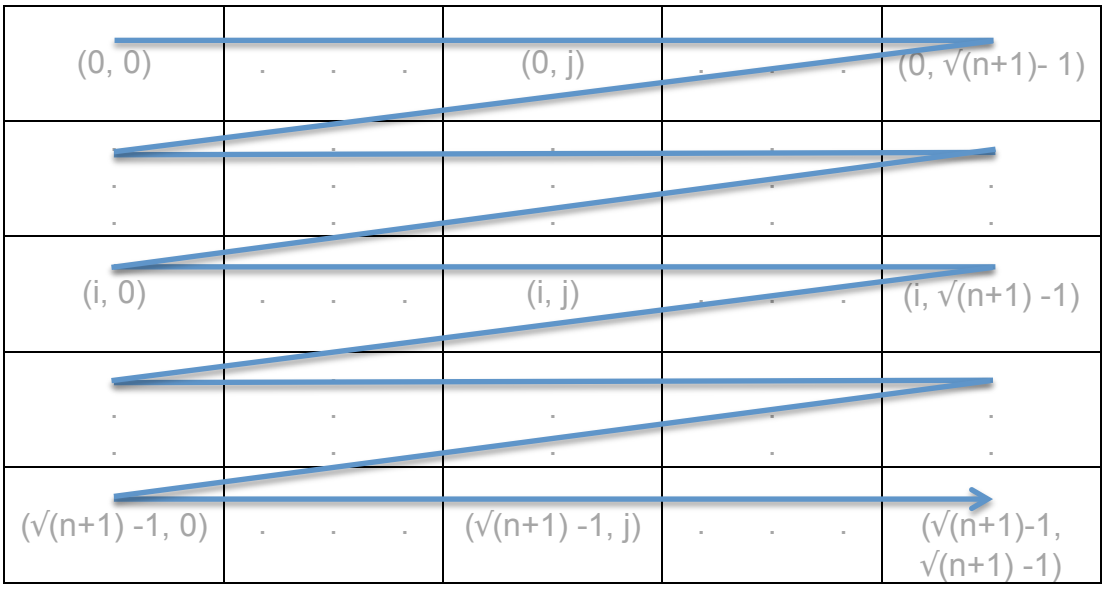


Figure 5-2: Generic representation of tile numbering. The row and column of each possible tile position is shown in the format (row, column).

5.2 Solvability Check

Using theory from section 3.1 and the findings of Ryan (2004), I have produced the pseudocode shown in Figure 5-3 to determine the solvability of a given puzzle.

```
Create a 1-dimensional array representing initial puzzle configuration
Calculate number of inversions in this array
Determine row number of space (counting from the top) taking the first row to be row 1
Determine dimension of puzzle

If puzzle dimension is even
        If row of space is even and sum of inversions is odd
                Solvable
        If row of space is odd and sum of inversions is even
                Solvable
        Else
                Insolvable
Else
        If sum of inversions is even
                Solvable
        Else
                Insolvable
```

Figure 5-3: Solvability check pseudocode

5.3 Search Algorithm Implementation

A-Star Implementation

I have chosen to use this implementation for both the breadth-first search and also the A* search. The only difference being that for the A* search the f-value will consist of the sum of the g-value and the h-value, whereas for the breadth first search, the f-value will consist of the g-value only, where:

- f-value: overall cost
- g-value: depth of node in search space
- h-value: heuristic estimate of distance from node to target node

The reason why I have chosen to use the same core algorithm for both searches, is because both searches work in the same manner, other than the fact that the A* algorithm uses a heuristic. Thus, by using the same algorithm for both, I can accurately analyse the effect the use of a heuristic has by keeping every other aspect of the search the same, thus neatly isolating the heuristic. The pseudocode for this approach is shown in Figure 5-4 and is my adaptation of the pseudocode presented by Russel and Norvig (1995, pp.93) and Russel and Norvig (1995, pp.97).

```
Create initial node
Add initial node to open list
While open list not empty
        Current node = node with lowest f-value
        If current node = target node
                Solution found: Terminate search
        Generate children of current node and store in array
        For each child in array
                If node with same tile configuration exists in closed list
                        If node in closed list has larger f-value than child node
                                Remove node from closed node and add child node to open list
                Else If node with same tile configuration exists in open list
                        If node in open list has larger f-value than child node
                                Remove node from open node and add child node to open list
                Else add child node to open list
```

Figure 5-4: A-star search pseudocode

IDA-Star Implementation

For the same reasons I used the same core algorithm for the breadth first search and the A* search, I have chosen to use the same algorithm for both the iterative-deepening depth first search and the IDA* search. As with the breadth first search and A* search, the only difference will be that for the IDA* search, the f-value will consist of the sum of the g-value and the h-value, whereas for the iterative-deepening depth-first search search, the f-value will consist of the g-value only.

I will use an iterative approach, the pseudocode for which is shown in Figure 5-5 and Figure 5-6, and is my adaption of that provided by Russell and Norvig (1995, p. 107).

```
Create initial node
Set f-limit = 0
While(true)
        fetchSolutionOrNextLimit(root,fLimit)
        If(solution found)
                Solution found: terminate search
```

Figure 5-5: IDA* search pseudocode

```
fetchSolutionOrNextLimit(root, limit)

Set next f-limit to a large number
If f-value of node is larger than the f-limit
        Return f-value of node
If tile configuration of node is equal to the target configuration
        Solution is found
        Return f-value of node
Generate children of node and store in array
For each child node
        New f-limit = fetchSolutionOrNextLimit(child node, flimit)
        If new f-limit is smaller than next f-limit
        Set next f-limit to new f-limit
Return next f-limit
```

Figure 5-6: fetchSolutionOrNextLimit pseudocode

5.4 Heuristic Implementation

Misplaced Tiles

The pseudocode I have designed for the misplaced tiles heuristic is shown in Figure 5-7.

```
Distance = 0
For each tile position of the puzzle board
        If the tile occupying the position is not in its target position (and not a space)
                Distance ++
Return Distance
```

Figure 5-7: Misplaced tiles pseudocode

Relaxed Adjacency

Figure 5-8 shows my adaptation of the pseudocode for relaxed adjacency, provided by Hansson et al. (1985, p. 6)

```
Distance = 0
While tiles are not in the correct order
        If space is in its target location
                Swap space with the tile of largest value that isn't in its target location
        Else
                Move tile, the target location of which is the location of the space, into the space
        Distance ++
Return Distance
```

Figure 5-8: Relaxed adjacency pseudocode

Number of Tiles out of Row and Column

The pseudocode I have developed to calculate the number of tiles out of row and column is shown in Figure 5-9

```
Distance = 0
For each tile position of the puzzle board
        If the tile occupying the current tile position is in the incorrect row
                Distance++
        If the tile occupying the current tile position is in the incorrect column
                Distance++
Return distance
```

Figure 5-9: Number of tiles out of row and column pseudocode

Checkerboard Misplaced

Figure 5-10 shows the pseudocode, which I have designed to calculate the checkerboard misplaced distance.

```
Distance = 0
For each tile position of the puzzle board

        If tile occupying the current tile position is not its target position
                If position is on the even side of the board, and it contains a tile that belongs on the even side
                        Distance += 2
                Else if position is on the odd side of the board, and it contains a tile that belongs on the odd side
                        Distance += 2
                Else
                        Distance += 1
Return distance
```

Figure 5-10: Checkerboard misplaced pseudocode

In order to implement this method I also need to design some code, which will dynamically calculate which side of the board a position is on, and which side of the board the tile occupying that position belongs on. The pseudocode for this is shown in Figure 5-11 and this will be used for each iteration of the pseudocode shown in Figure 5-10.

```
If puzzle board is of an even width
        If (position number + row number) is even
                Position is even
        Else
                Position is odd
        If tile number + (tile number-(tile number % board width))/board width) is even
                Tile is even
        Else
                Tile is odd
Else
        If position number is even
                Position is even
        Else
                Position is odd
        If tile number is even
                Tile is even
        Else
                Tile is odd
```

Figure 5-11: Side of tile and position calculation pseudocode

Manhattan Distance

I have developed the pseudocode shown in Figure 5-12 to calculate the Manhattan distance of an arrangement of tiles.

```
Distance = 0
For each tile position of puzzle board
        If tile occupying position is not in its target position
                Distance += |row of tile's target position- row of tiles current position|
                Distance += |column of tile's target position- column of tiles current position|
Return Distance
```

Figure 5-12: Manhattan distance pseudocode

Manhattan Distance + Linear Conflict

The pseudocode for calculating the linear conflict and adding this to the manhattan distance for a tile arrangement is shown in figure 5-13.

```
Distance = 0
For each row of puzzle board
        Set maximumTileInCorrectRow = -1
        For each tile in row
                If tile is in correct row
                        If tile value > maximumTileInCorrectRow
                                maximumTileInCorrectRow = tile value
                        else
                                Distance +=2
For each column of puzzle board
        Set maximumTileInCorrect column = -1
        For each tile in column
                If tile is in correct column
                        If tile value > maximumTileInCorrect column
                                maximumTileInCorrect column = tile value
                        else
                                Distance +=2
Distance += manhattan distance

Return Distance
```

Figure 5-13: Manhattan distance + linear conflict pseudocode

5.5 Database implementation

5.5.1 Representation of Information in Database

In order for the use of pattern databases to remain effective, I will have to store a substantial amount of data to an external file. When the database is required for use, its contents must be loaded into memory so that they can be accessed quickly and frequently. For these reasons, the efficiency of the way in which data is saved to and loaded from these databases will have a direct impact on the efficiency of any searches that require their use.

The following data will be saved to the database for each pattern arrangement:

- Position of pattern tiles
- Cost of moving position tiles to target locations

My initial idea was to hold these in a key: value data structure. The key for each entry would be a 2 dimensional integer array, storing the positions of all pattern tiles in a specific arrangement. The value for each entry would be an integer representing the number of moves required to return the pattern tiles to their target locations, from the arrangement represented by the key.

Example to Determine Feasibility of proposed Approach

The diagram below shows a simple pattern on a 4 x 4 puzzle board (pattern tiles highlighted in yellow). Our aim is to store the number of moves it will take to return any possible arrangement of these tiles to an arrangement where the pattern tiles and the space are in their target positions as shown in Figure 5-14.

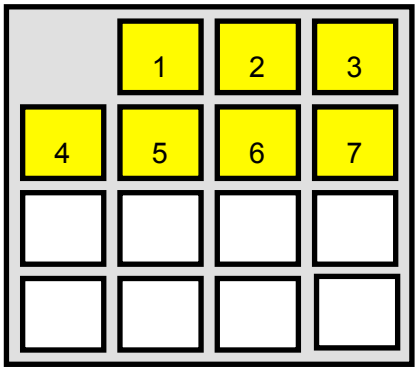


Figure 5-14: Example of a 7-tile pattern

Let us consider a pattern database storing moves required to return any possible arrangement of these tiles, to an arrangement where the pattern tiles and the space are in their target locations. The calculation shown in Figure 5-15 demonstrates how many database entries would be needed, and subsequently how much memory would be required to store this database.

- Number of possible permutations (number of records required in database)= $\frac{16!}{(16-8)!}$ = 518,918,400 permutations.
- Minimum memory requirement to store each key (int[][]) = 16 integers + 5 array addresses = (16 x 32) + (5 x 64) = 832 bits
- Minimum memory requirement to store each value (int) = 32 bits

Therefore the memory requirements would be larger than 52.1943GB.

Figure 5-15: Calculation of memory requirements to store proposed pattern database

Although I have adequate space to save this amount of data onto the hard drive of my computer, access will be extremely slow unless I have the option of storing the entire contents to RAM allowing me to view/edit it in an efficient manner. It is for this reason that the above way of representing the data is not feasible. The computer I am working with has 8GB of RAM available, so

unless I can find a means of saving pattern databases in under 8GB their implementation will not be of any benefit.

After extensive reading around this issue I discovered a concept which Pemmaraju and Skiena (2003, p. 59) call “ranking”. Lexicographical ranking in particular, is a way in which I can use a mathematical algorithm that makes use of permutation theory, in order to represent any given permutation of a set of elements as a single integer, based on where its position would be if all possible permutations were ordered in lexicographical order (Pemmaraju and Skiena 2003, p. 59).

Once stored as an integer, it becomes logical that a ranked value can then be un-ranked using an inverse of the ranking function, transforming the integer back into the original pattern permutation. This is a very effective way of compacting an extensive number of pattern arrangements into a relatively small amount of memory. This will allow me to store a significantly higher number of pattern arrangements and their costs into memory, making the utilisation of pattern databases not only possible, but also highly efficient.

Given a permutation π, Pemmaraju and Skiena (2003, p.59) provide the following ranking function r:

$$r(\pi) = d_0(n-1)! + d_1(n-2)! + \ldots + d_{n-1}(0)!$$ *Where d_i is the location of tile i relative to its preceding tiles.*

Figure 5-16: Ranking function provided by Pemmaraju and Skiena (2003, p. 59)

Pemmaraju and Skiena (2003, p.59) take into consideration the position of each tile relative to its preceding elements (d_i). As I am investigating permutations based on their pattern tiles only, I will ignore non-pattern tiles when constructing the ranking function.

With a bit of work I will be able to adapt this equation to fit my requirements.

Derivation of a Ranking Function

- Let pattern tile array p = {p_0, p_1, … p_{n-1}}
- Let π be permutation of pattern tiles= $\begin{pmatrix} p_0 & \cdots & p_{n-1} \\ \pi_{p_0} & \cdots & \pi_{p_{n-1}} \end{pmatrix}$
- Let n = number of pattern tiles in pattern
- Let N = number of pattern tiles + number of non- pattern tiles + 1

  = dimension^2
- Let l_i = location l_i of each pattern tile p_i relative to tiles p_j where j<i

  = π_i – (sum of **pattern tiles** j in π where j<i and $\pi_j < \pi_i$)

In each term of their equation, Pemmaraju and Skiena (2003, p. 59) multiply d_i by (n-(i+1))!. However, as blank tiles are not individually regarded as being unique, the number of permutations of a pattern arrangement will be far less.

Therefore, from the formula for r(π) supplied by Pemmaraju and Skiena (2003, p.59), I need only multiply l_i by (N-(i+1))!/n!, because this is the number of permutations available when i tiles have been fixed.

Hence their formula can be easily adapted to the below:

r(π) = l_0[(N-1) x (N-2) x … x (N-(n-1))]
+ l_1[(N-2) x (N-3) x … x (N-(n-1))]
+ …
+ l_{n-2}[(N-(n-1))]
+ l_{n-1}[1]

which reduces to the following formula:

$$r(\pi) = l_{n-1} + \sum_{i=0}^{i<n-1} l_i \, [(N-i-1)(N-i-2) \ldots (N-(n-1))]$$

Using this formula I can represent any position of pattern tiles by an integer, and this integer can be used as an index of an array. The cost of moving pattern tiles to their target locations from each pattern configuration can be saved in an array at the calculated index.

Derivation of an Unranking Function

As discussed previously, the ranking function I formulated can be described in the below form:

$r(\pi) = l_0[(N-1) \times (N-2) \times \dots \times (N-(n-1))]$
$+ l_1[(N-2) \times (N-3) \times \dots \times (N-(n-1))]$
$+ \dots$
$+ l_{n-2}[(N-(n-1))]$
$+ l_{n-1}[1]$

Each term in this equation consists of the product of l_i and $(n-i-1) \times (n-i-2)\dots(N-(n-1))$.

If we could determine the l_i of every pattern tile from the ranked integer, it would be very easy to recreate the pattern configuration using these values.

Let $x = r(\pi)$

Using modular theory to reverse the ranking function the following statements must hold:

$l_{n-1} = x \bmod (N-(n-1))$

$l_{n-2} = (x - l_{n-1}) \bmod ((N-(n-1)) \times ((N-(n-1)+1)$

$l_{n-3} = (x - l_{n-1} - l_{n-2}) \bmod ((N-(n-1)) \times ((N-(n-1)+1) \times ((N-(n-1)+2)$

.

.

.

$l_0 = (x - l_{n-1} - l_{n-2} - \dots - l_1) \bmod ((N-(n-1)) \times (N-(n-1)+1) \times (N-(n-1)+2) \times \dots \times (N))$

From this we can derive the following:

➔ $l_i = (x - l_{n-1} - l_{n-1} - \dots - l_{i-1}) \bmod \frac{(N-(n-1)) \times (N-(n-1)+1) \times \dots \times (N-i-1) \times (N-i)}{(N-(n-1)) \times (N-(n-1)+2) \times \dots \times (N-i-1)}$

➔ $l_i = (x - l_{n-1} - l_{n-1} - \dots - l_{i-1}) \bmod \frac{\cancel{(N-(n-1)) \times (N-(n-1)+1) \times \dots \times (N-i-1) \times} (N-i)}{\cancel{(N-(n-1)) \times (N-(n-1)+2) \times \dots \times (N-i-1)}}$

➔ $l_i = (x - l_{n-1} - l_{n-1} - \dots - l_{i-1}) \bmod (N-i)$

This formula can be used to construct an array of the l_i for each tile i, and can therefore be used to reconstruct a pattern configuration from a given integer-unranking it.

Below shows the memory consumption of the pattern database required for the situation described figure 5-14, this time using ranking and unranking:

- Number of possible permutations (number of records required in database)= $\frac{16!}{(16-8)!}$ = 518,918,400 permutations
- Minimum memory requirement of each record = 8 bits (each entry can be represented by a single byte).

Therefore the minimum memory requirements for this database is reduced to approximately 495 MB. This is a substantial improvement.

5.5.2 Constructing the database

A pattern database can be constructed using retrograde analysis (Culberson and Schaeffer 1998, p. 321). Baba and Jain (2001, p. 63) describe how retrograde analysis is an approach to investigating a given problem by starting at the solution and working backwards, in an effort to establish an idea of how the solution can be reached.

I can therefore use a reverse breadth-first search to utilise retrograde analysis for a given N-Puzzle, to enable me to find the number of moves required to transform a permutation of the pattern tiles into the target permutation.

Non-Additive

As previously discussed, non additive pattern databases hold the number of moves required to take the positions of the pattern tiles from any possible permutation to their target permutation. Therefore generating a non-additive pattern database is very straightforward.

Figure 5-17 shows the pseudocode that I have designed for generating a non-additive pattern database.

```
Create database of length n!/(n−p)! (n = dimension^2, p = length of pattern tiles (incl. space))
Int numberOfPatterns = 0
Int currentLevel = 0
Byte initialPattern = rank(goalStateArray)
Database[initialPattern] = 0
numberOfPatterns++

while(numberOfPatterns<Database.length)
        for every entry of Database which is equal to the current level
                temporaryArray = unrank the entry
                generate all possible children of the array and store in an array list
                for each child
                        rank child
                        set Database[ranked child] equal to currentLevel+1
                        numberOfPatterns++
        currentLevel++
```

Figure 5-17: Pseudocode for creating a non-additive database

Additive

Additive patterns however are different:

- They store the required number of moves of pattern tiles only.
- Individual patterns can not have overlapping pattern tiles.

Although as a concept this is rather simple to grasp, when it comes to implementation it causes a couple of issues:

Problem 1

The number of moves required to transform a given pattern configuration the target configuration is no longer the depth of the configuration's parent plus one. If it is the product of moving a non-pattern tile in its parents configuration, then this value will in fact be equal to the depth of its parent. This causes an issue with breadth-first search as once we have iterated over the entire database for entries of a certain depth, the children produced may technically be of the same depth as their parents, and so we can't commit ourselves to moving onto an incremented depth.

Solution to Problem 1

Instead of iterating through the database, searching for pattern arrangements of one depth at a time, we can hold two lists of configurations. One list will contain pattern configurations of the depth which we are currently exploring. The other will contain pattern configurations which exist at the next depth (the current depth plus one). Until the first array is empty, we can continue to iterate through each of the pattern configurations contained inside it, generating the children of each pattern arrangement and adding these children to the appropriate list, before removing the configuration from the current list. This ensures that each depth is fully explored before the algorithm proceeds to the next.

Problem 2

As disjoint patterns can not have overlapping pattern tiles, our final pattern databases can not all specify the position of the space in their pattern. If they did then they would technically have an overlapping pattern tile. However, when generating the database we iterate through previously ranked patterns, unrank them and generate their children. If we ignore the position of the space when ranking the pattern arrangements in the first place, then when they are unranked the position of the space will no longer be held. This creates a problem in that if we do not know the position of the space, we can not determine its possible children.

Solution to Problem 2

A way of avoiding this problem is to create a pattern database that takes into consideration the position of the space, and once this database is constructed, we can use it to create a pattern database which does not consider the position of the space.

This would work by iterating through the original pattern database, each time unranking the index, and then re-ranking it to obtain an updated index which ignores the position of the space. The distance stored in the original database can then be saved to the new database at the updated index. In cases where multiple entries will be mapped to the same index of the new pattern databases, the largest cost will be used.

The resulting database will therefore be an additive pattern database.

Figure 5-18 shows the pseudocode I have written for creating an additive database. Here the space is regarded as a pattern tile, this must be dealt with before the resulting database can be utilised.

```
Create DB array of length n!/(n−p)! (n = dimension^2, p = length of pattern tiles (incl. space))
Set all values in DB array to -1
Create current open list to hold ranked pattern configurations at current depth
Create next open list to hold ranked pattern configurations at current depth + 1
Set depth count= 0
Rank the initial pattern configuration and add to the open list
While current list is not empty
    While current list is not empty
        If the database array entry with the index of 1st element in current open list=-1
            Set the DB array entry to the current depth
            Unrank 1st value in current open list to get pattern configuration
            Generate its children
            Rank all children and calculate the cost of generating each child  (0/1)
            For each child
                If cost of generating child = 1
                    If (DB array entry indexed at rank of child=-1 or DB array entry
                    indexed at rank of child> depth+1)
                        Add rank of child to next open list
                Else if cost of generating child = 0
                    If (DB array entry indexed at rank of child= -1 or DB array
                    entry indexed at rank of child > depth)
                        Add rank of child to current open list
        Remove 1st element of the current open list
    Swap the current open list with the next open list
    Increment depth count
Return DB array
```

Figure 5-18: Pseudocode for generating additive database which accounts for the position of the space

Figure 5-19 shows the pseudocode that I have developed to map the database entries to a new database, disregarding position of space.

```
Create new DB array of length n!/(n−p−1)! (n = dimension^2, p = length of pattern tiles (incl. space))
Set all values in DB array to -1
For each entry in old DB array
    Unrank entry
    Re-rank entry regardless of position of space
    If(If (new DB array entry indexed at rank re-rank value =-1 or new DB array entry indexed at
    rank re-rank value> old DB array entry)
        Set new DB array entry indexed at re-rank value equal to old DB array entry
Return new DB array
```

Figure 5-19: Pseudocode for creating a new database from by disregarding the position of the space

5.5.3 Memory Restrictions for Database Creation

Whether creating an additive or a non-additive database, I must initially take into consideration the position of the space. This means that to construct either kind of pattern database for a puzzle with N tiles, using patterns of size n, at some point of the algorithm, an array holding $\frac{N+1!}{((N+1)-(n+1))!}$ elements will be necessary. This will restrict my choices regarding which database methods I can implement.

8-Puzzle

To create a database that stores the number of moves required to transpose any possible permutation to the target permutation for an 8-puzzle, I would need to store the following number of bytes in memory:

$$\frac{9!}{2} = 181440$$

An approximation of the amount of memory this would require is 177KB.

The reason why the number of entries is 9!/2 and not 9! is because since I am using retrograde analysis, and all tiles are taken into consideration, only half of the possible permutations of the N-Puzzle will be reachable from the goal state. When implemented, the solvability of a given configuration will be determined prior to the search being initiated; so only valid starting configurations need to be stored in the database.

Because 177KB is a relatively small amount of memory, the use of pattern databases will not be necessary for solving the 8-Puzzle because we can look up the actual configuration in this database.

15-Puzzle

To generate a database that stores the number of moves required to transpose any reachable permutation to the target permutation for the 15-Puzzle, I would need to store the following number of bytes in memory:

$$\frac{16!}{2} = 1.0461395e + 13$$

The amount of memory it would take to generate such a database would be approximately 9743GB. This is a vast amount of memory, and therefore when dealing with the 15-Puzzle, we must make use of pattern databases.

In order to create a pattern database for a 7-tile pattern, the following number of bytes would need to be stored within my laptop's memory:

$$\frac{16!}{(16-8)!} = 518{,}918{,}400$$

Therefore the minimum amount of memory this would require is approximately 494MB.

To construct a pattern database for an 8-tile pattern however, the following number of bytes would be stored in RAM:

$$\frac{16!}{(16-9)!} = 4{,}151{,}347{,}200$$

And so the minimum amount of memory this would require is approximately 4GB.

Given the fact that arrays in java can only store a maximum of 2,147,483,647 elements (which is the maximum size an integer can be inside java), I would need to adapt this method to divide each pattern database into two separate arrays. One array would hold the cost of solving pattern configurations, whose ranks are below 2,147,483,647 and the other would hold the cost of solving pattern configurations whose ranks are greater than or equal to 2,147,483,647. After multiple attempts to implement this alteration, the unfortunate result was that my laptop did not have sufficient RAM to generate a pattern database with this volume of entries.

6 Experimental Design

8-Puzzle

I will begin by testing the implemented algorithms on a number of 8-Puzzles. I have developed a number of test cases of various complexities, which can be found in Appendix A1. I will use these test cases to test the following algorithms:

- BFS
- IDS
- A*
- IDA*

I will test A* and IDA* with the following heuristics:

- Misplaced Tiles
- Checkerboard misplaced
- Relaxed Adjacency (N-Max Swap)
- Number of Tiles out of Row and Column
- Manhattan Distance
- Manhattan Distance + Linear Conflict
- Database Lookup

15-Puzzle

I will then move on to test the advanced algorithms with a number of 15-Puzzles. The test cases for the 15-Puzzle testing can be found at Appendix A2.

The search algorithms I will test with the 15-Puzzle are:

- A*
- IDA*

I will test each of these with the following heuristics:

- Misplaced Tiles
- Checkerboard misplaced
- Relaxed Adjacency (N-Max Swap)
- Number of Tiles out of Row and Column
- Manhattan Distance
- Manhattan Distance + Linear Conflict
- Fringe Pattern Database
- Corner Pattern Database
- Max(Fringe, Corner) Pattern Database
- Additive Database 5-5-5
- Additive Database 6-6-3

For each test I run I will record the following results:

- Number of iterations required
- Search time (hh:mm:ss.000)
- Time per iteration (nanoseconds)
- DB load time (hh:mm:ss.000)
- Maximum number of nodes stored in memory at one time
- Number of runs completed

I will run simpler puzzles up to 1000 times and record their average results (utilising the batch processing functionality). The harder the puzzles become, the less times I will test them due to project time constraints.

7 Results and Analysis

7.1 Search Method Comparison Analysis

7.1.1 Search Time

Probably the most obvious way in which to analyse multiple search algorithms is to compare running times for a number of problems.

One way to analyse this for the A* and IDA* algorithms is to compare the running times of the BFS and the IDS search times. This is possible because I purposely implemented BFS and IDS using the same algorithms as A* and IDA* respectively. The only difference in their implementation is that BFS and IDS set the heuristic value of generated nodes to 0. Therefore, by comparing these algorithms I can completely isolate my analysis of the effectiveness of different search techniques from any heuristic interference.

Figure 7-1 demonstrates how IDS outperformed BFS for all tested cases of the 8-Puzzle, especially as the complexity of the puzzles increased. This provides us with an insight of how the performance of the A* and the IDA* algorithms will compare with harder puzzles.

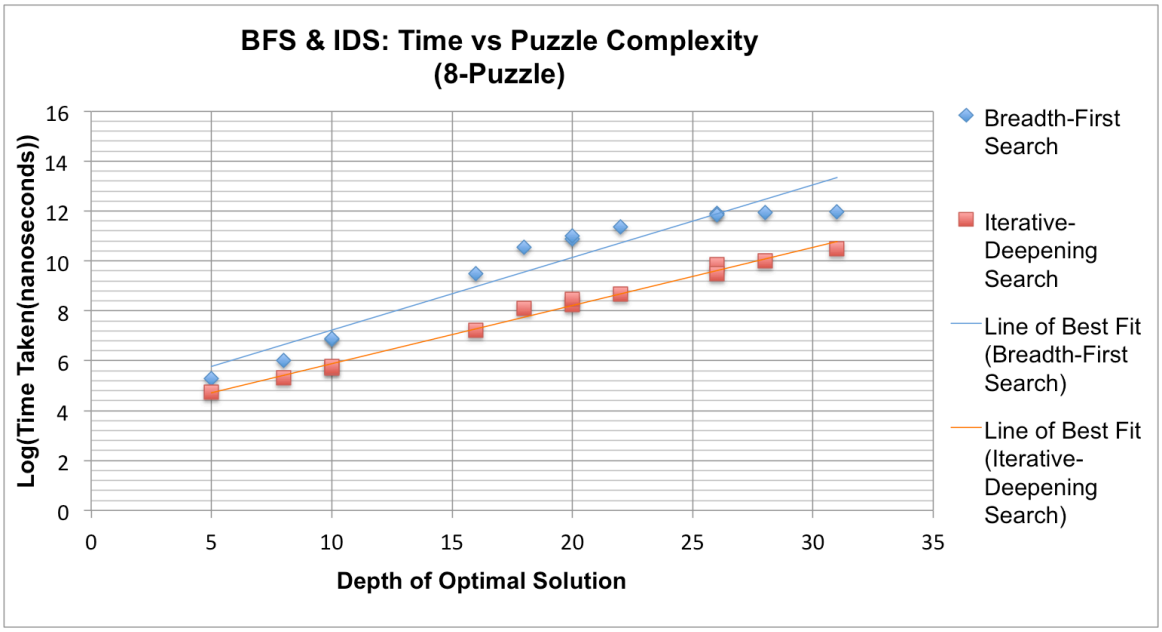


Figure 7-1: Scatter graph of solve times for 8-Puzzle test cases with BFS and IDS

Another way in which we can isolate the performance of the alternative search methods is by using a constant heuristic. Figure 7-2 and Figure 7-3 below show how the running times of A* and IDA* for both the 8-Puzzle and the 15-Puzzle are relatively close to one another for some of the less complex puzzles. However, as with the BFS vs IDS graph, as the complexity increases the performance of IDA* starts to exceed that of A* dramatically.

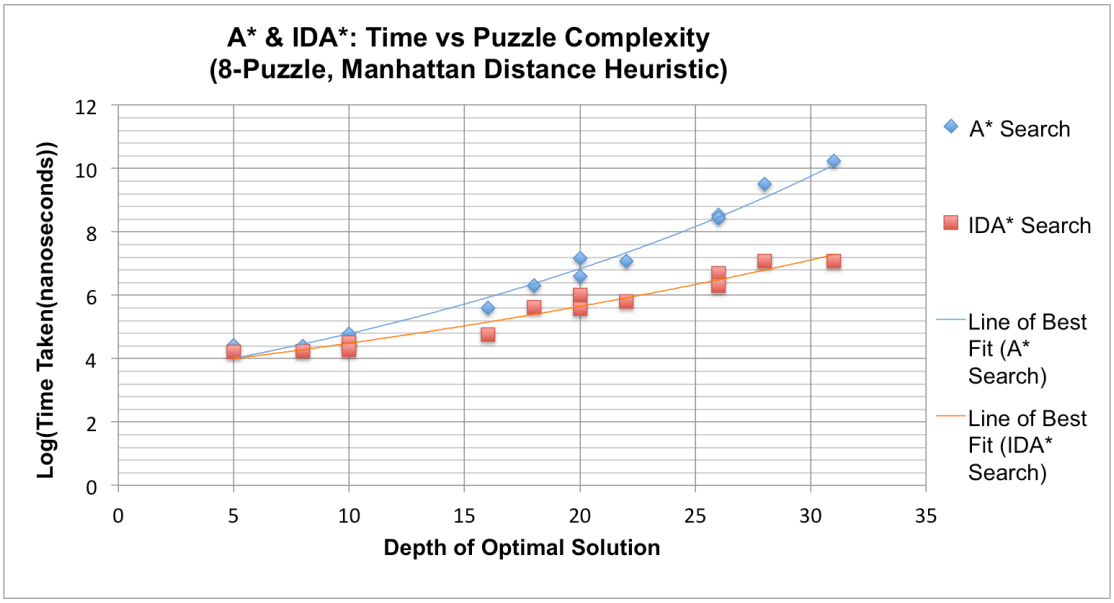


Figure 7-2: Scatter graph of solve times for 8-puzzle test cases with A* & IDA*. Manhattan distance heuristic used for all cases.

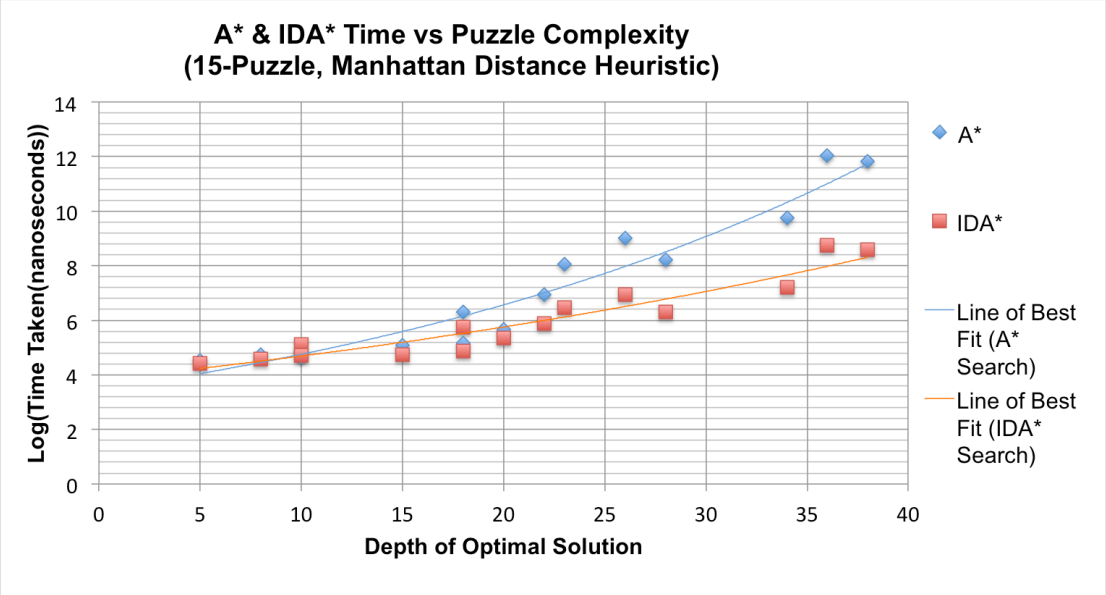


Figure 7-3: Scatter graph of solve times for 15-puzzle test cases with A* & IDA*. Manhattan distance heuristic used for all cases. Only test cases with an optimal solution at a maximum depth of 38 are in included.

Table 7-1 below demonstrates how the average search times differ as the core search algorithm is changed. For test cases with solution at a depth of 38 or below, the A* search (with the Manhattan distance heuristic) took on average approximately 2 minutes, however IDA* with the same heuristic took on average only 65 milliseconds. By observing the trend shown in Figure 7-2 and Figure 7-3 on the previous page, for more complicated instances of the N-Puzzle, this gap would surely rise.

Search Method	Average Time Taken (seconds)
A* (Manhattan Distance Heuristic)	120.5005284
IDA* (Manhattan Distance Heuristic)	0.065796277

Table 7-1: Average search time of A* and IDA* with Manhattan distance heuristic for 15-Puzzles with optimal solution at maximum depth of 38.

7.1.2 Memory Requirements

Another factor that can set algorithms apart from each other is the amount of memory they require to run. Because of the time constraints of this project, I did not have the opportunity to allow searches to run for long enough that memory became an issue. However, the amount of memory an algorithm requires is a very critical factor to consider. As a search progresses and the search depth increases, the number of nodes held in memory increases, and this can be the main source of memory issues. Figure 7-4 below demonstrates the maximum number of nodes stored in memory at any one point for BFS and IDS.

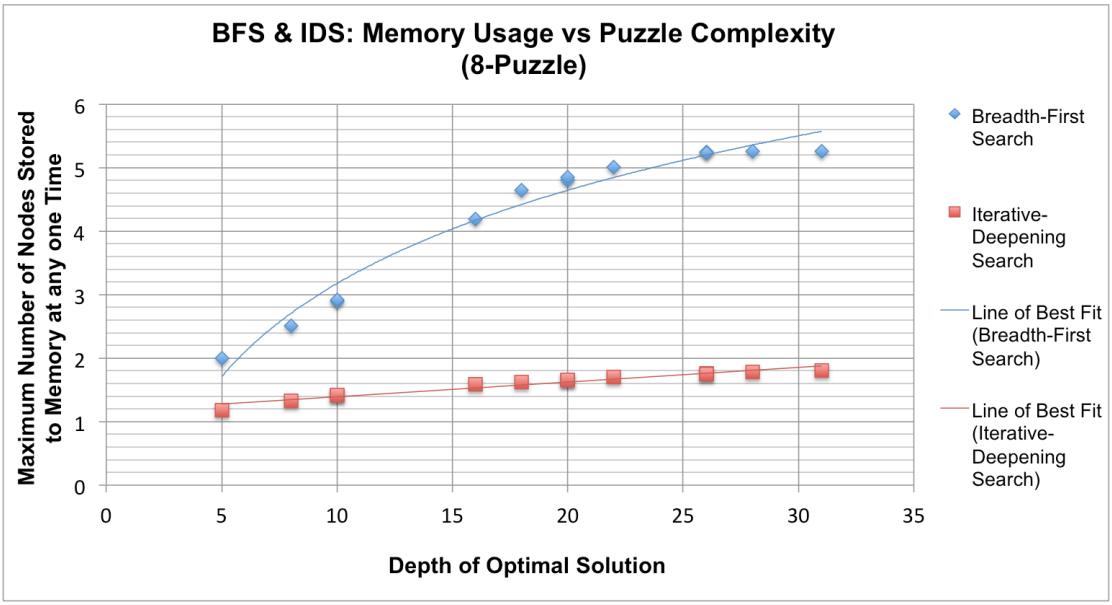


Figure 7-4: Scatter graph of maximum number of nodes stored in memory for 8-Puzzle test cases with BFS and IDS.

As expected, very similar results can be found by comparing the A* and IDA* algorithms with a fixed heuristic method Figure 7-5 and Figure 7-6.

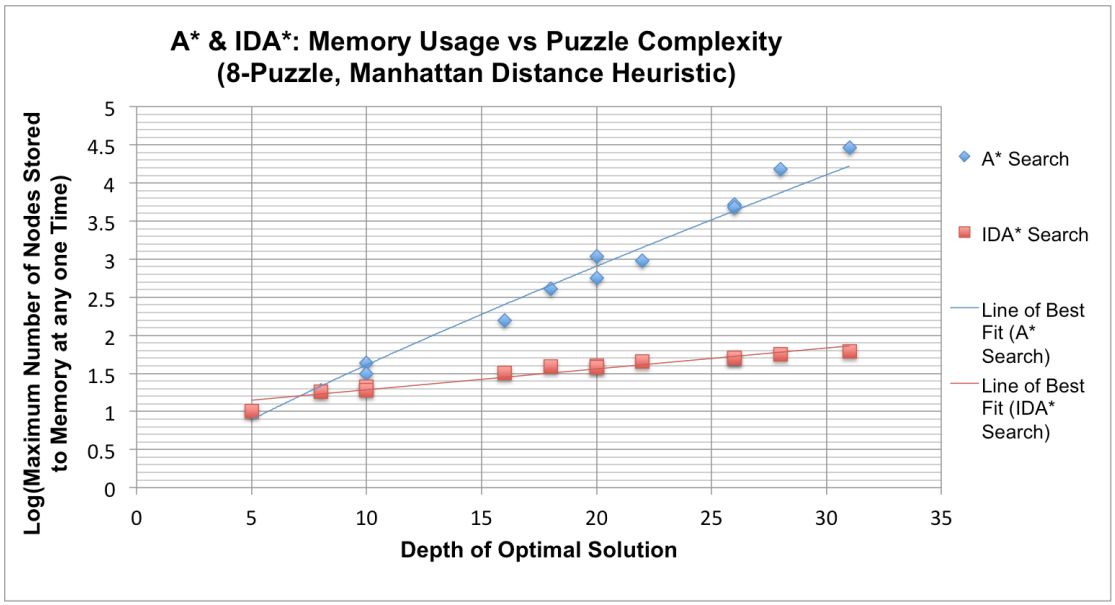


Figure 7-5: Scatter graph of maximum nodes stored in memory for 8-puzzle test cases with A* & IDA*. Manhattan distance heuristic used for all cases.

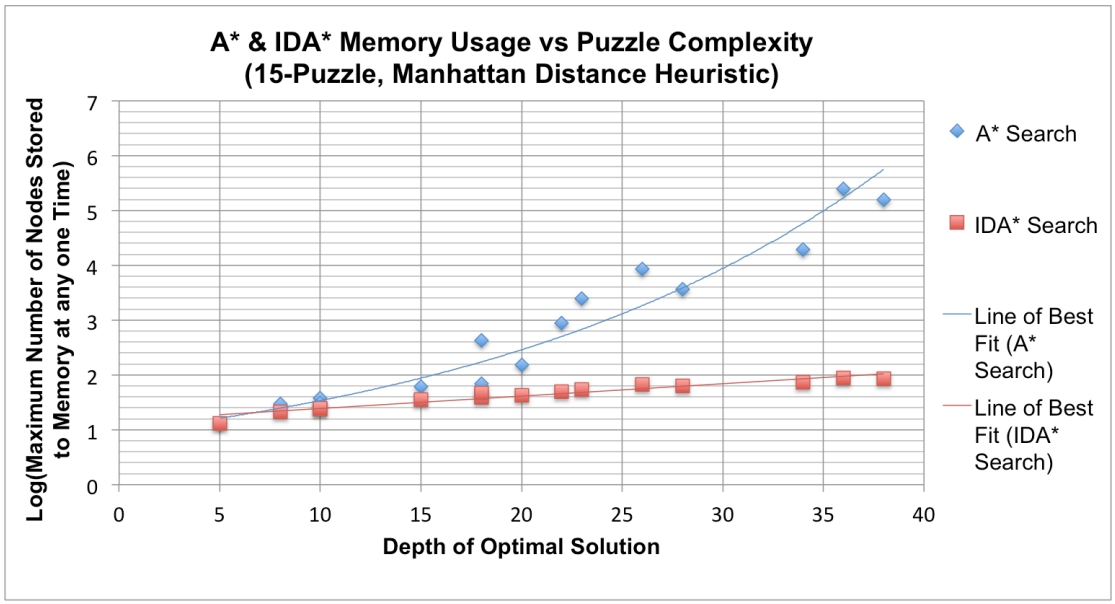


Figure 7-6: Scatter graph of maximum nodes stored in memory for 15-puzzle test cases with A* & IDA*. Manhattan distance heuristic used for all cases. Only test cases with an optimal solution at a maximum depth of 38 are in included.

It is apparent from studying Figures 7-5 and 7-6 on the previous page that the memory required by A* grows exponentially as the puzzles get harder. IDA* however shows only a steady incline. This reflects what I learned during my research, and can be explained by the fact that A* must save every node visited to memory, but IDA* only stores nodes in the current path being explored. Therefore, as shown, the more complex the problems get, the larger the difference between each method's memory requirements becomes. To provide some context around why the difference in the number of nodes saved to memory is an issue, it is worth taking a look at how much memory each node requires.

Each node contains nine class variables. Ignoring any Java overheads, I have produced a very crude calculation shown in Figure 7-7 of the minimum memory required to store each node (for a 15-Puzzle), assuming 64 bits are required for storing each address. More accurate memory requirement calculations can be problematic, as these overheads vary between JVM's.

Approximate memory required for storing Node class variables:

- Parent node (reference to Node): 64 bits
- Puzzle Solver (reference to Solver): 64 bits
- Node Configuration (int[4][4]): (32 x 16) +(5 x 64) = 832 bits
- Row of space (int): 32 bits
- Column of space (int): 32 bits
- f-value (int): 32 bits
- g-value (int): 32 bits
- h-value (int): 32 bits
- Subject of move (String, longest possible = "right"): (16 x 5) + 64 = 144 bits

Approximate Total memory required per node = 158 bytes.

Figure 7-7: Calculation of the approximate amount of memory required to store each node to memory

Studying Table 7-2 below, we can see that the average maximum number of nodes stored in memory for A* (using the Manhattan distance heuristic) when solving 15-Puzzle was is 29351.2 (considering only puzzles whose optimal solutions are no greater than 38). According to the calculations in Figure 7-7 on the previous page, this would require at least 4.42265 MB to store. The average maximum number of nodes stored by IDA* at any one time however, was only 48.6 nodes. This would only require approximately 7.49883 KB of memory. Figures 7-5 and Figure 7-6 (p.101) suggest that for puzzles of higher complexity this difference would grow rapidly.

Search Method	Average Maximum Number of Nodes Stored in Memory at a Time
A* (Manhattan Distance Heuristic)	29351.2
IDA* (Manhattan Distance Heuristic	48.6

Table 7-2: Average maximum number of nodes stored at a time by A* and IDA* with Manhattan distance heuristic for puzzles with optimal solution at maximum depth of 38 (15-Puzzle)

7.1.3 Rate of Iterations

Not all search methods work at a constant rate. To measure the rate at which an algorithm functions is relatively easy. We can simply count the number of iterations that occur and divide the time taken by the number of iterations of the most time consuming part of the algorithm. Comparing the rate of two completely different algorithms however, is not a straightforward challenge. In an effort to produce some informative results, I found I could not measure the rate of the A* algorithm in the same way as the IDA* algorithm. This is because both algorithms work in a completely different way. The most time consuming part of my A* implementation is the generation of children nodes, because each node generated must be checked against the entire open and closed lists to check that it hasn't been visited previously. This is not the case for IDA* and so it is the expansion of nodes which is most time consuming part of the algorithm. Therefore, the most informative way in which to measure the rate at which A* works in my opinion is to divide the time taken by the number of nodes generated. However, in my opinion the most informative way in which to measure the rate at which IDA* works is to divide the time taken by the number of nodes whose children have been generated. Therefore, it is not fair to plot the results of both on the same graph, but I can view them separately to gain perspective on how both algorithms perform for deeper searches.

Figure 7-8, Figure 7-9 and Figure 7-10 on the following pages show how the time the BFS and consequently the A* algorithms take to generate each node dramatically increases as puzzles increase in complexity. This is largely due to the fact that they must check through an ever-increasing list of previously visited nodes for every node expansion. As the search deepens, this list can become extremely large. In other words, with my implementation of A*, the complexity of the iterations increases exponentially as the search deepens, which evidently has a massive impact on the efficiency of the algorithm.

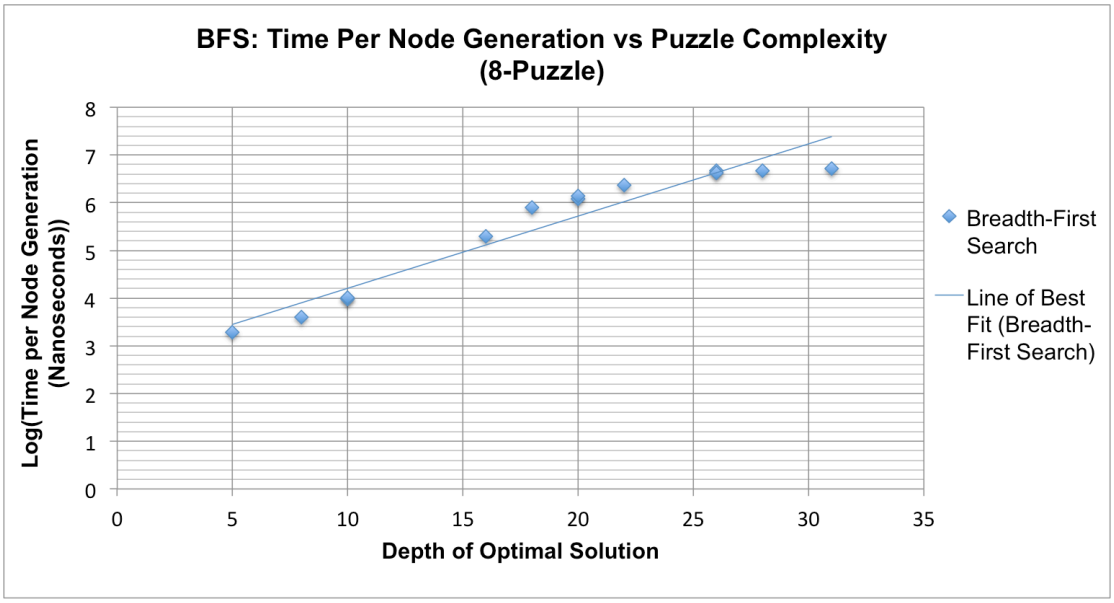


Figure 7-8: Scatter graph showing the average time taken for each node generation for the 8-Puzzle test cases with BFS.

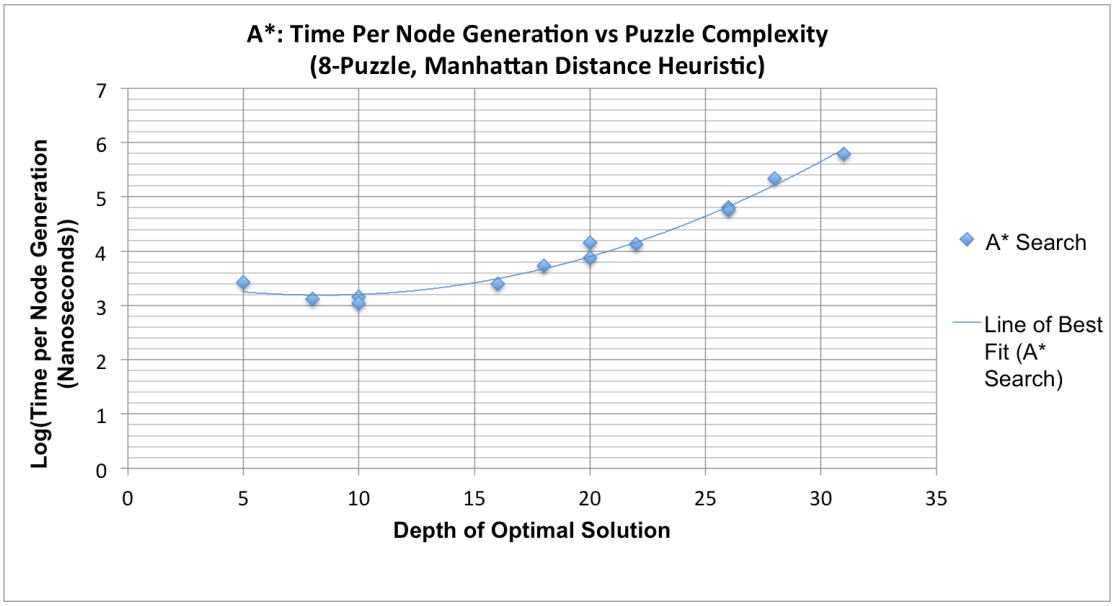


Figure 7-9: Scatter graph showing the average time taken for each node generation for the 8-Puzzle test cases with A*. Manhattan distance used for all cases.

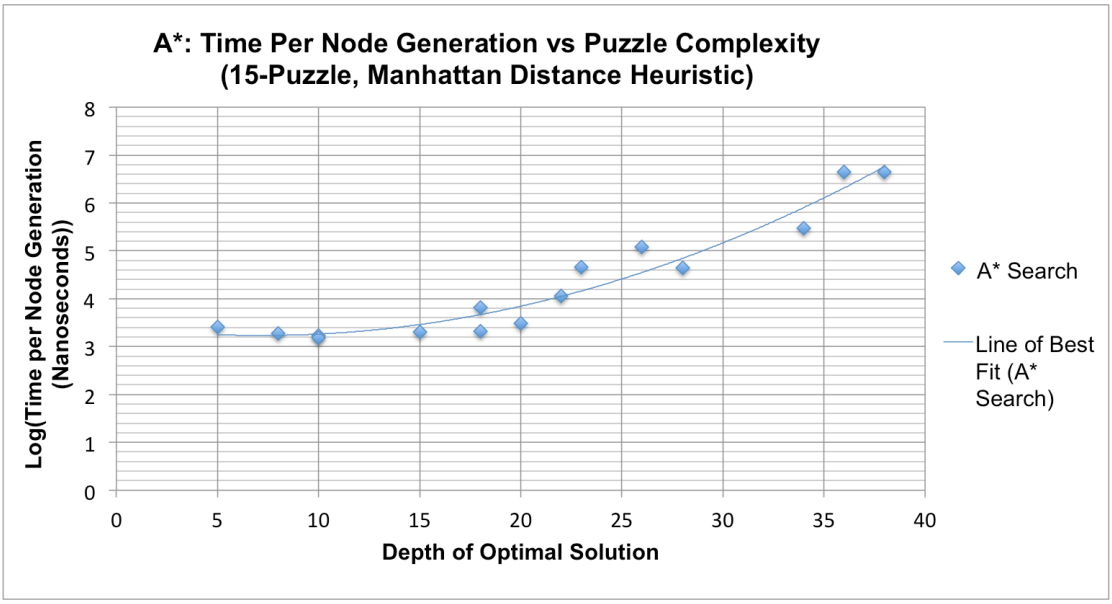


Figure 7-10: Scatter graph showing the average time taken for each node generation for the 15-Puzzle test cases with A*. Manhattan distance used for all cases. Only test cases with an optimal solution at a maximum depth of 38 are in included.

Figure 7-11, Figure 7-12 and Figure 7-13 on the following pages all demonstrate a major difference between the ways in which the two core algorithms perform. IDS and consequently IDA* perform just as efficiently with puzzles of a higher complexity as they do with the more basic puzzles. A reason for this is that IDA* does not consider whether nodes have been previously visited, and therefore, unlike A*, every iteration of the IDA* algorithm is equally quick.

One observation made from Figure 7-11, Figure 7-12 and Figure 7-13 concludes that IDS and IDA* appear to speed up slightly as the puzzles get more complex. This is probably due to the fact that the initial stage of the IDA* algorithm takes time, and here no nodes are expanded. Therefore, when the results are calculated by dividing the time taken by the number of nodes expanded, this initiation time is effectively shared among the expansions that take place. This makes it seem as if each expansion takes longer than it actually does. As the complexity increases however, this time is shared between a greater number of expansions and eventually becomes negligible. For this reason, if I were to repeat this exercise, I would initiate the timers just before the first expansion takes place for both algorithms.

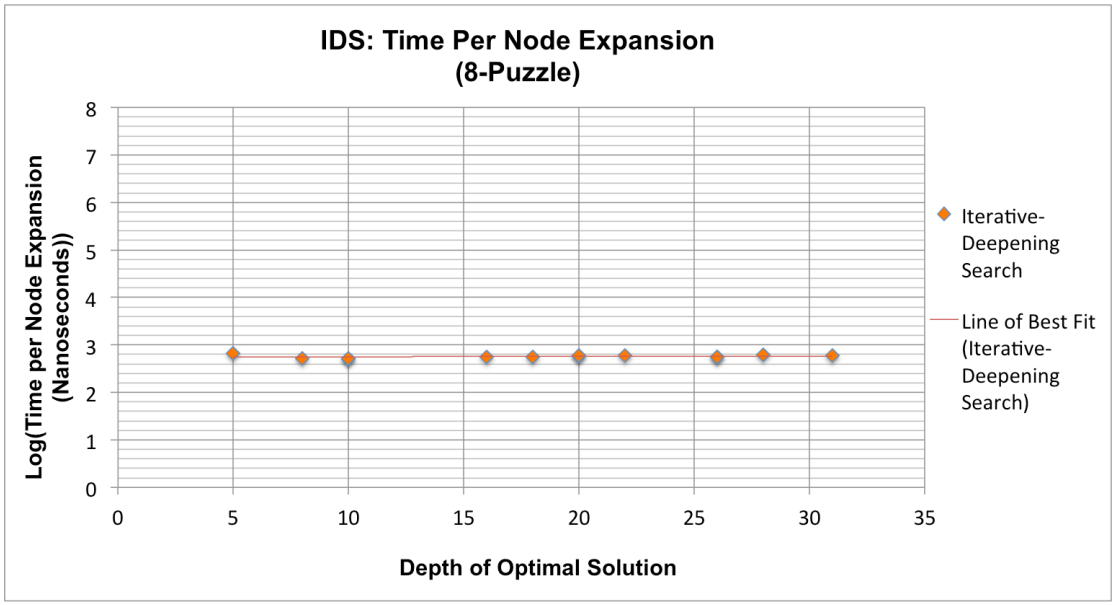


Figure 7-11: Scatter graph showing the average time taken for each node generation for the 8-Puzzle test cases with BFS.

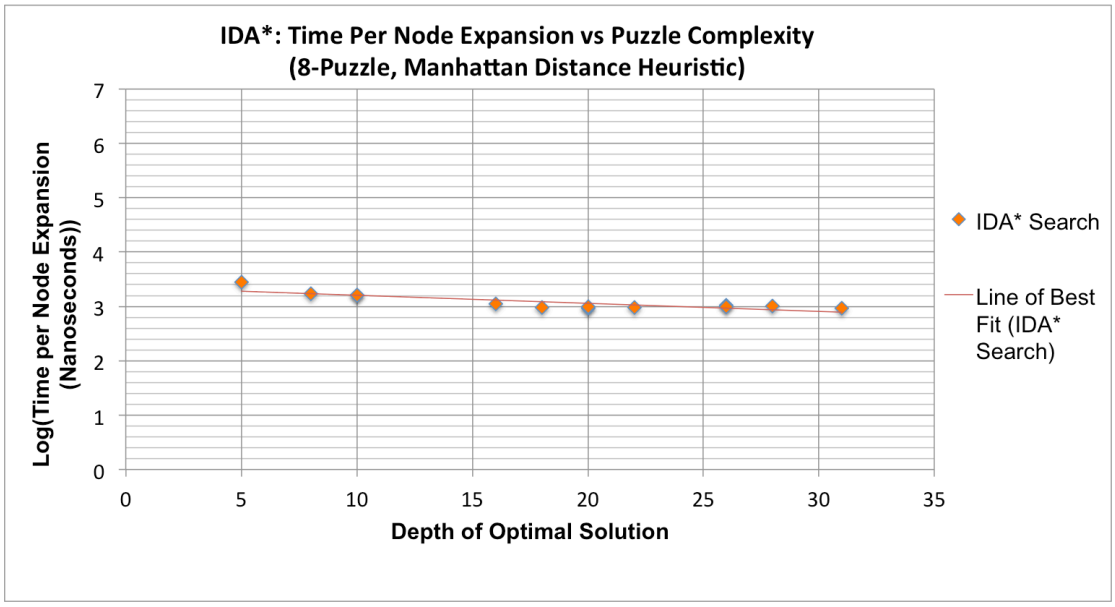


Figure 7-12: Scatter graph showing the average time taken for each node generation for the 8-Puzzle test cases with A*. Manhattan distance used for all cases.

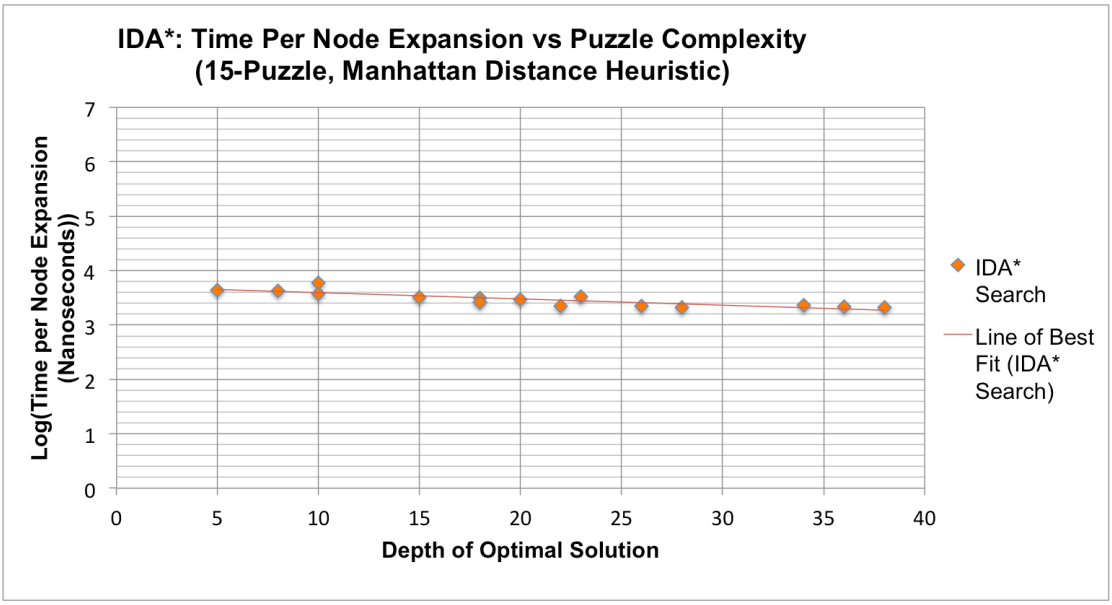


Figure 7-13: Scatter graph showing the average time taken for each node generation for the 15-Puzzle test cases with A*. Manhattan distance used for all cases. Only test cases with an optimal solution at a maximum depth of 38 are in included.

7.2 Heuristic Method Comparison Analysis

The heuristic functions I implemented all took different amounts of time to return each estimate. There are two main factors which cause this; the suitability of each heuristic approach to the specific situation, and the time complexity of the individual heuristic algorithm. However, the difference in the time complexity of my heuristic algorithms is reasonably limited. Therefore, when it comes to analysing the effectiveness of alternative heuristics, the most crucial factor to consider is the rate at which the search converges with each heuristic method. The more advanced a heuristic is, the less iterations of the informed search will be required.

Table 7-3 below shows how many nodes were generated for each tested case of the 8-Puzzle, for each heuristic. On observing Table 7-3, it becomes apparent that the database lookup heuristic was the most effective by far. This is certainly no surprise, as it provides the accurate distance from each node to the goal rather than just an estimate. The Manhattan distance with and without the linear conflict calculation also performed very well.

Depth of Optimal Solution	Misplaced Tiles	Checkerboard misplaced	Relaxed Adjacency (N-Max Swap)	Number of Tiles out of Row and Column	Manhattan Distance	Manhattan Distance + Linear Conflict	Database Lookup
5	10	10	10	10	10	10	10
8	46	19	31	25	19	19	18
10	114	58	78	66	43	43	29
10	92	37	52	37	31	28	19
16	1672	454	923	472	157	129	32
18	4234	1143	2482	1239	410	333	39
20	7886	1987	4453	1957	570	389	76
20	8792	2668	5160	2992	1092	766	38
22	17523	4192	9944	4422	958	423	45
26	82670	27164	56038	27363	5226	3207	135
26	71506	23371	47000	22647	4735	2779	94
28	115979	49543	86307	50381	15136	9180	464
31	161867	110396	155641	87834	28951	18431	879

Table 7-3: Number of iterations required to solve the 8-puzzle with A* for each heuristic

As can be seen in Table 7-4 below, the performance of heuristics varies from case to case. Due to the variety in heuristic approaches it is not the case that a *better* heuristic will always out perform a *worse* heuristic.

Test Case	Depth of Optimal Solution	Number of Iterations: Max(Fringe, Corner) Pattern Database	Number of Iterations: Additive Database 5-5-5
TC1	5	15	13
TC2	8	122	30
TC3	10	125	33
TC4	10	27	25
TC5	15	42	35
TC6	18	95	41
TC7	18	257	193
TC8	20	65	43
TC9	22	3791	312
TC10	23	98	506
TC11	26	5475	1132
TC12	28	258	509
TC13	34	2025	416
TC14	36	21711	42219
TC15	38	1460	2453
TC16	39	45560	13619
TC17	42	358186	309769
TC18	46	35261	60501
TC19	48	75175	33458

Table 7-4: Number of nodes generated for 8-Puzzle with A* test cases of maximum depth of 48.

Therefore a scatterplot of these results does not provide for any conclusive analysis.

However, taking the average number of iterations required by A* and IDA*, for the 15-puzzle test cases, with each heuristic provides us with a clear comparison of the overall performance of each heuristic. These results can be observed in Figure 7-14, Table 7-5, Figure 7-15 and Table 7-6 on the next few pages.

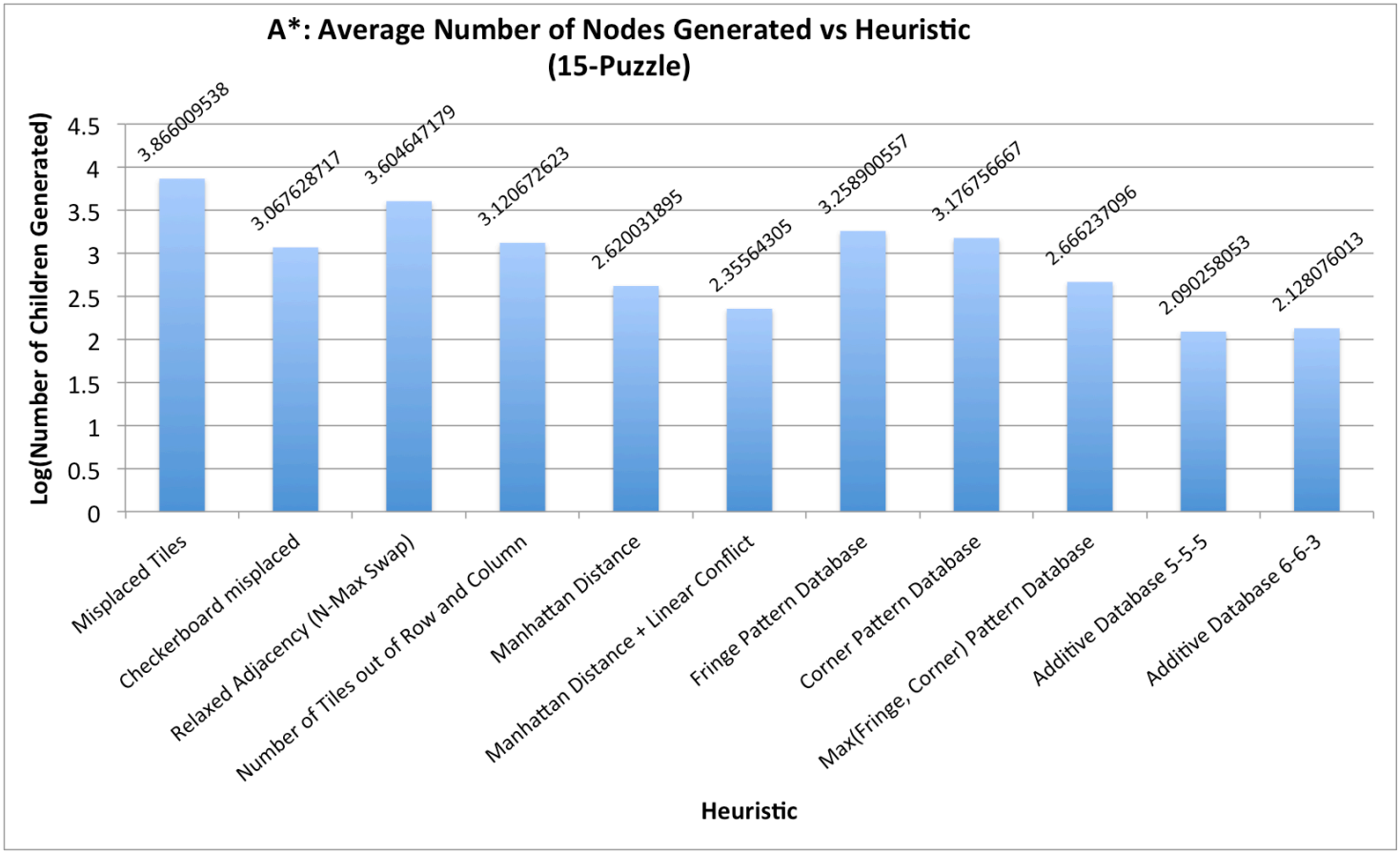


Figure 7-14: Average number of nodes generated to solve each test case of the 15-Puzzle with A* for each heuristic. Only test cases with an optimal solution at a maximum depth of 23 are in included

Heuristic	Average Number of Nodes Expanded
Misplaced Tiles	7345.3
Checkerboard misplaced	1168.5
Relaxed Adjacency (N-Max Swap)	4023.9
Number of Tiles out of Row and Column	1320.3
Manhattan Distance	416.9
Manhattan Distance + Linear Conflict	226.8
Fringe Pattern Database	1815.1
Corner Pattern Database	1502.3
Max(Fringe, Corner) Pattern Database	463.7
Additive Database 5-5-5	123.1
Additive Database 6-6-3	134.3

Table 7-5: Average number of children generated to solve each test case of the 15-Puzzle with A* for each heuristic. Only test cases with an optimal solution at a maximum depth of 23 are in included

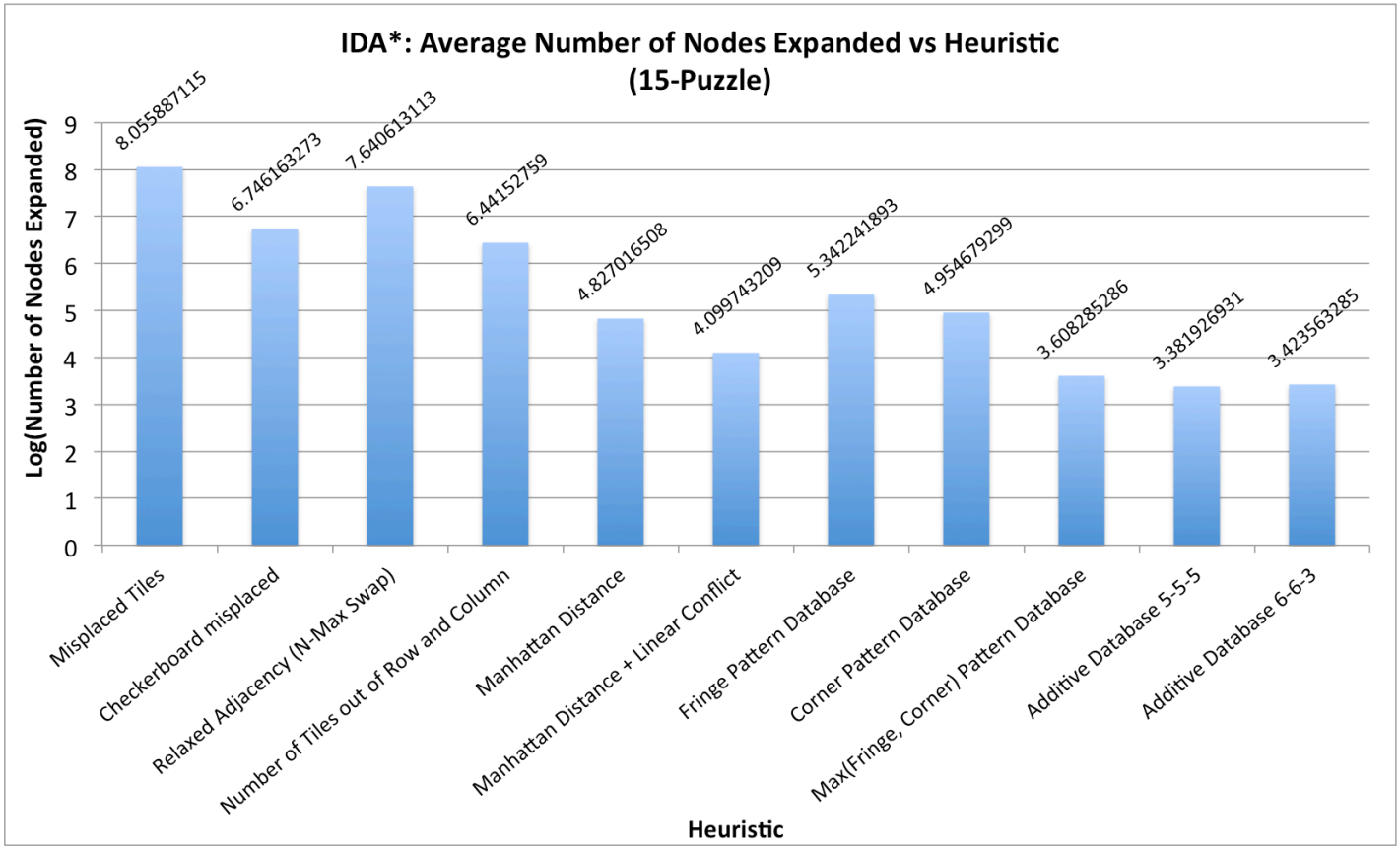


Figure 7-15: Average number of nodes generated to solve each test case of the 15-Puzzle with IDA* for each heuristic. Only test cases with an optimal solution at a maximum depth of 39 are in included

Heuristic	Average Number of Nodes Expanded
Misplaced Tiles	113733162.3
Checkerboard misplaced	5573952.625
Relaxed Adjacency (N-Max Swap)	43713251.63
Number of Tiles out of Row and Column	2763933.5
Manhattan Distance	67145.4375
Manhattan Distance + Linear Conflict	12581.8125
Fringe Pattern Database	219908.4375
Corner Pattern Database	90090.5625
Max(Fringe, Corner) Pattern Database	4057.75
Additive Database 5-5-5	2409.5
Additive Database 6-6-3	2651.9375

Table 7-6: Average number of nodes generated to solve each test case of the 15-Puzzle with IDA* for each heuristic. Only test cases with an optimal solution at a maximum depth of 39 are in included

Figure 7-14, Table 7-5, Figure 7-15 and Table 7-6, all show that on average the most accurate heuristics are (in no particular order):

- Additive databases: 6-6-3
- Additive databases: 5-5-5
- Non- Additive databases: Max(Fringe Pattern, Corner Pattern)
- Manhattan distance
- Manhattan distance and linear conflict

Although the Max(Fringe Pattern, Corner Pattern) heuristic forms a relatively accurate heuristic, the size of the databases which must be loaded into memory is rather substantial. Each of the two databases must store nearly 519 million bytes of data (which is approximately 495MB). If we compare this with the databases used in the 6-6-3 pattern, which require approximately 5.5 MB, 5.5 MB and 3KB bytes of data respectively, we can see that additive databases can be as effective- if not more effective than additive databases and can have far lower memory requirements. At times, the databases required for the Max(Fringe Pattern, Corner Pattern) heuristic caused some issues such as taking a long time to load into memory, as shown in Table 7-7 on the next page.

		Database Loading Time (Seconds)				
Test Case	Depth of Optimal Solution	Fringe	Corner	Max(Fringe, Corner)	Additive Datavases 5-5-5	Additive Databases 6-6-3
TC1	5	1.014	1.011	3.303	0.001	0.013
TC2	8	1.018	1.016	3.176	0.001	0.013
TC3	10	1.479	2.048	18.178	0.002	0.013
TC4	10	1.03	1.009	2.792	0.001	0.013
TC5	15	1.025	1.013	7.63	0.001	0.013
TC6	18	1.026	1.013	10.137	0.001	0.013
TC7	18	1.016	1.006	8.152	0.001	0.013
TC8	20	1.018	1.008	3.214	0.001	0.013
TC9	22	1.011	1.011	3.359	0.001	0.013
TC10	23	1.569	1.733	18.796	0.002	0.018
TC11	26	1.096	1.037	18.592	0.002	0.017
TC12	28	1.025	1.03	19.221	0.002	0.017
TC13	34	7.928	2.025	22.75	0.002	0.017
TC14	36	10.047	1.131	5.422	0.002	0.017
TC15	38	1.337	1.539	3.427	0.002	0.017
TC16	39	1.24	1.338	3.365	0.002	0.017
TC17	42	12.756	12.257	3.891	0.002	0.019
TC18	46	7.785	1.015	24.812	0.002	0.017
TC19	48	1.08	1.006	22.405	0.002	0.017
TC20	50	1.163	8.174	18.338	0.002	0.017
TC21	52	1.237	1.269	18.465	0.002	0.017
TC22	54	0.961	1.366	22.69	0.002	0.017
TC23	56	7.443	1.5	21.106	0.002	0.017

Table 7-7: Database loading times for 15-Puzzle test cases with IDA*. Only test cases with an optimal solution at a maximum depth of 56 are in included.

Often this problem occurred when puzzle being solved was part of a batch. In this situation, because of the size of the required databases, the garbage collector must work hard to free up enough virtual memory to allow this load, which could explain the large loading times. Having said that, this probably wouldn't be as much of an issue with a more powerful machine.

7.3 Overall Performance Analysis

As previously discussed, various search methods and heuristics perform differently under particular circumstances. It makes for interesting analysis to compare the overall performance of all available combinations, the search methods and heuristics implemented.

Figure 7-16 shows a comparison of how the average solve times compare for each combination of search algorithm and heuristic for the 8-Puzzle.

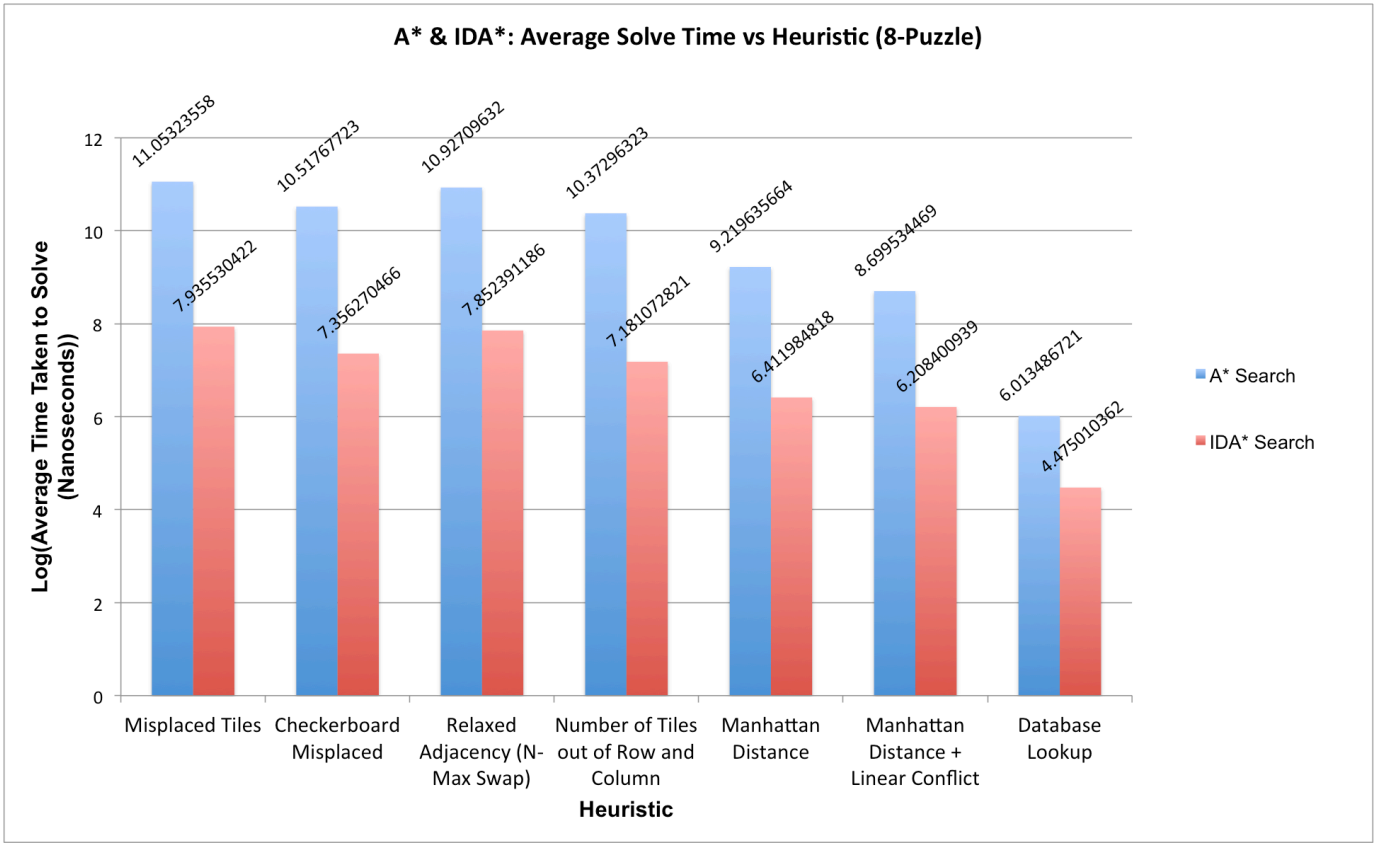


Figure 7-16: Average solve times for 8-Puzzle with A* and IDA* and each Heuristic.

Figure 7-16 demonstrates that the most effective combination in terms of time taken to solve an 8-puzzle are (in order):

- IDA* with Database lookup
- IDA* with Manhattan distance + linear conflict
- IDA* with Manhattan Distance

Figure 7-17 demonstrates that when it comes to solving the more simple cases of the 15-Puzzle, the most effective combinations in terms of time required are (in order):

- IDA* Additive database 5-5-5 database lookup
- IDA* Additive database 6-6-3 database lookup
- IDA* with Manhattan distance
- IDA* with Manhattan distance + linear conflict

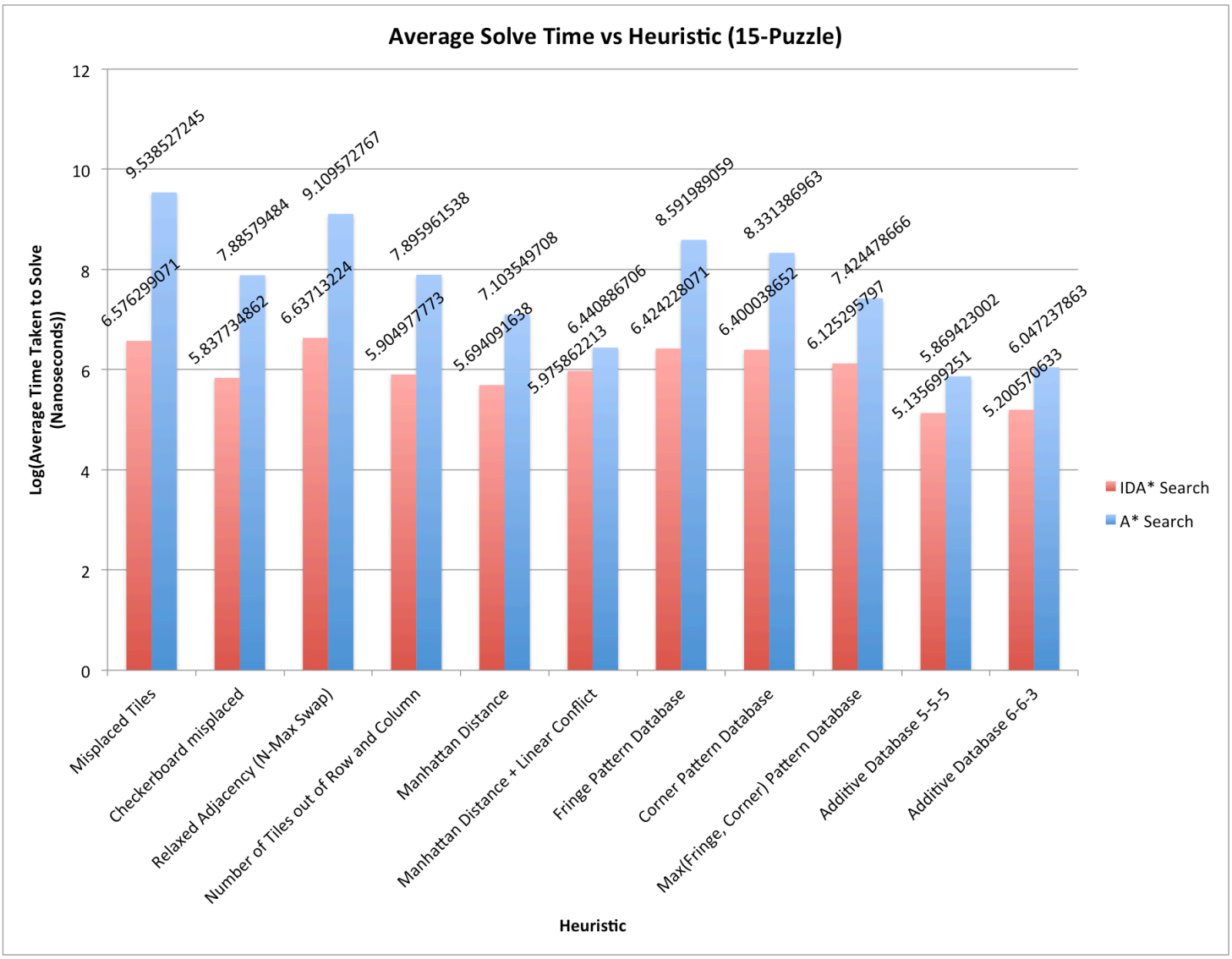


Figure 7-17: Average solve times for 15-Puzzle with A* and IDA* and each Heuristic. Only test cases with an optimal solution at a maximum depth of 23 are in included.

For simpler cases, it seems that linear conflict takes more time to calculate than the amount of time it saves by narrowing the search.

Table 7-8 shows the average solve time for each combination of search and heuristic for 15-Puzzle problems whose optimal solutions are at a maximum depth of 23 which supports the conclusions we drew from Figure 7-17.

	Average Time Taken (Milliseconds)	
	A*	IDA*
Misplaced Tiles	3455.630078	3.769633
Checkerboard misplaced	76.876719	0.688232
Relaxed Adjacency (N-Max Swap)	1286.982872	4.336429
Number of Tiles out of Row and Column	78.697609	0.803485
Manhattan Distance	12.6925741	0.494415
Manhattan Distance + Linear Conflict	2.7598578	0.945937
Fringe Pattern Database	390.83105	2.656
Corner Pattern Database	214.48008	2.51211
Max(Fringe, Corner) Pattern Database	26.57533	1.33443
Additive Database 5-5-5	0.740326	0.1366782
Additive Database 6-6-3	1.114905	0.1586977

Table 7-8: Average solve times for 15-Puzzle with A* and IDA* and each Heuristic. Only test cases with an optimal solution at a maximum depth of 23 are in included.

Something which can be observed from Table 7-8 on the previous page, is that in simpler cases, some of the more basic heuristics, for example the checkerboard-misplaced heuristic perform relatively well. However, how useful is an approach if it is only effective for simple cases? Korf, R. E and Schultze, P. (2005) represent the number of reachable configurations at each depth of the 15-Puzzle in a table. The spread of these configurations in the state space can be represented by Figure 7-18.

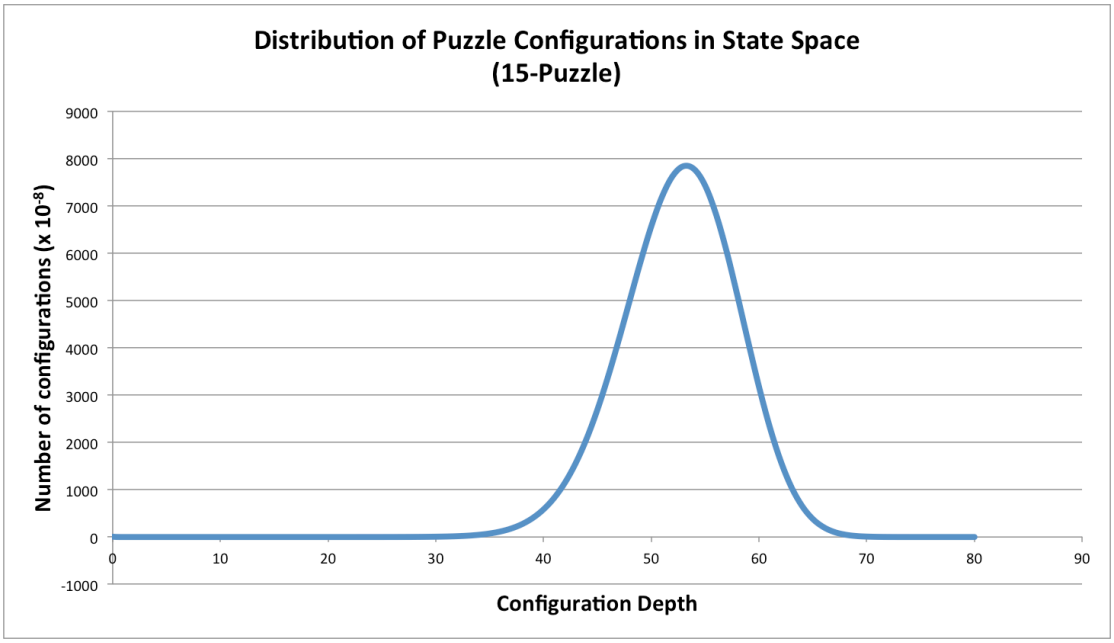


Figure 7-18: Distribution of tile arrangements for depths of the 15-Puzzle search space

A large proportion of the possible puzzle states exist at a depth greater than 50. In fact, over half of the possible configurations are at a depth of 53 or higher. Therefore, in order to find the most effective overall approach to solving the 15-Puzzle, we should consider only puzzles which can solve the majority of these difficult solutions within a reasonable time limit.

Figure 7-19 demonstrates the depth of most difficult problem, which each combination of search algorithm and heuristic found before its search times began to exceed 2 hours.

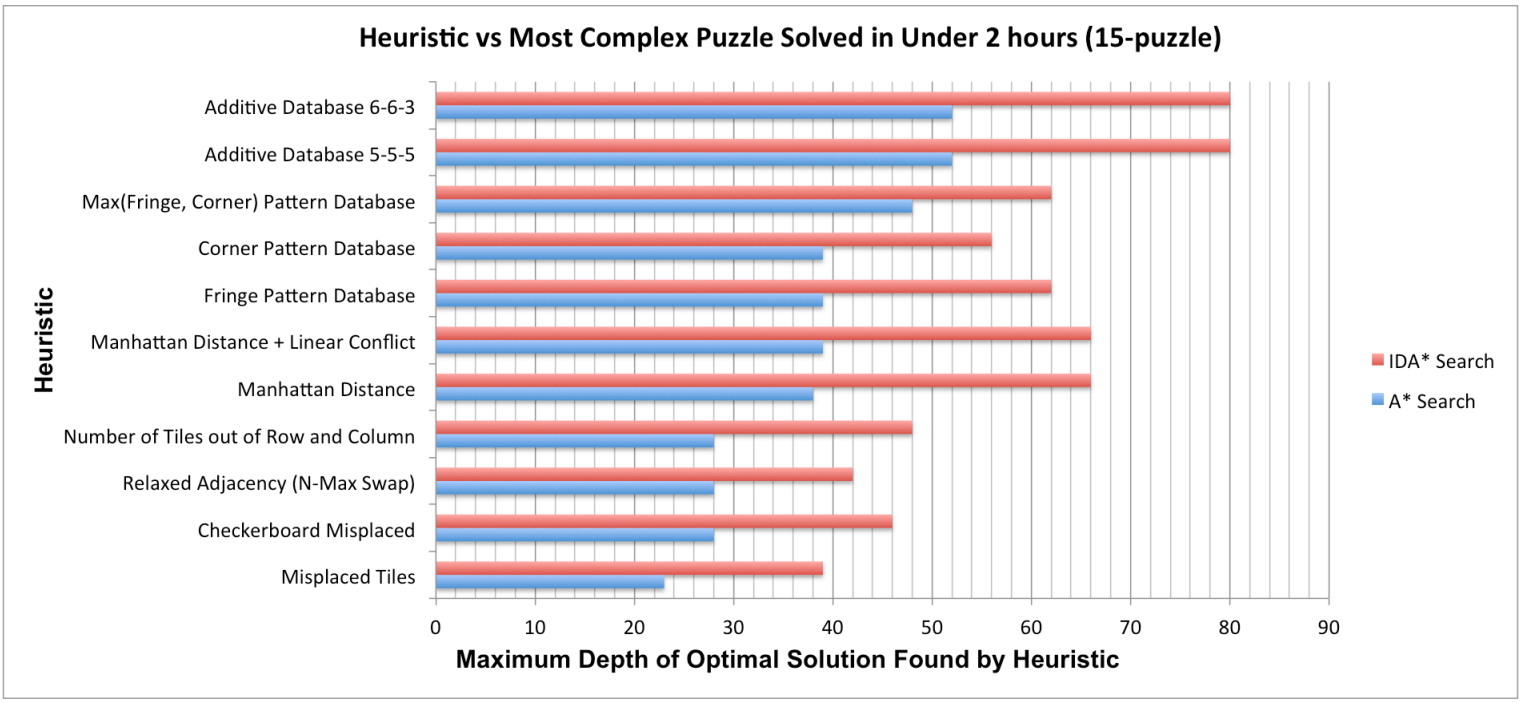


Figure 7-19: Maximum depth of optimum solution that can be found by each approach in under 2 hours.

From Figure 7-19, we can clearly see that although the Table 7-8 showed that checkerboard misplaced is pretty effective for some of the less complex puzzles, it becomes rather ineffective with puzzles where the optimal solutions are at a depth of 58, whereas even though the maximum of fringe and corner pattern databases heuristic is was slower on average for less complex puzzles, it in fact effective up until puzzles where the optimal solution is at a depth of around 65.

Figure 7-19 suggests that if we were to judge the quality of an approach by the maximum complexity of puzzles which it can generate a solution in a reasonable time frame, that the favoured approaches are the below:

- Additive database patterns 5-5-5
- Additive database patterns 6-6-3
- Manhattan distance
- Manhattan distance with linear conflict
- Fringe Pattern database
- Max(Fringe, Corner) pattern databases

As the additive databases proved to be the only heuristics which allowed a solution to be found for any 15-Puzzle that I tested, it makes sense to compare their performance for the most difficult cases tested to see which is the most effective. Table 7-9 shows that for 15-Puzzles of high complexity, the 6-6-3 pattern group outperformed the 5-5-5 pattern group every time.

	Search Time (hh:mm:ss.000)	
Depth of Optimal Solution	Additive Database 5-5-5	Additive Database 6-6-3
65	00:01:03.820	00:00:14.513
66	00:00:15.564	00:00:05.428
71	00:01:39.145	00:01:31.029
74	00:03:58.694	00:02:02.743
80	01:34:19.948	00:52:40.218
80	01:18:07.721	00:41:15.149

Table 7-9: Search time of IDA* with each additive pattern group for the 15-Puzzle configurations of highest complexity.

7.4 Evaluation

Evaluation of the Software Developed

Overall, the final piece of software has proven to be a very valuable research tool. All requirements (functional and non-functional) were met, which had a large impact on how manageable the experimentation stage was. The batch processing functionality especially proved to be a vital piece of the software, which made the running of multiple test cases a smooth and almost effortless process.

Evaluation of Quality of Results

Overall, the quality of my results is of a high standard. In most cases they have enabled me to discover patterns and reach informed conclusions that are relevant to my research. Due to the capability of the program I implemented, puzzles of a reasonable complexity were able to be solved a large number of times, which provided result averages. This increased the accuracy of my results, and helped to eliminate anomalies from my final findings. Therefore I would say that my results are very reliable.

Evaluation of Findings

Nothing in my findings contradicts anything stated in any previous studies that I have researched. In all cases, the obtained results reflect exactly what I expected to find, based on the research that I carried out earlier in the project. The fact that my findings strongly match my theory makes them dependable, as not only have they been reached through experimentation, but they have also been proven using scientific and mathematical concepts and calculations.

8 Further Work

As the additive patterns proved to be the most effective heuristic for the 15-Puzzle, I would like to perform some further investigation in this area. I would like to find out what makes a particular combination of additive pattern patterns effective. In order to explore this avenue I will design, build and implement some pattern databases of my own. I will approach this strategically, in an effort to achieve results which allow me to draw useful conclusions.

How do pattern arrangements affect performance?

I will test the following types of pattern arrangements

- Patterns consist of dispersed tiles
- Patterns arranged in rows
- Patterns arranged in blocks
- Patterns slot together

To make sure my results reflect the effectiveness of pattern shapes as much as possible I will keep the pattern sizes constant to (4-4-4-3)

Dispersed Patterns

I first tested this pattern to determine whether it matters if the blocks making up a pattern are adjacent to one another.

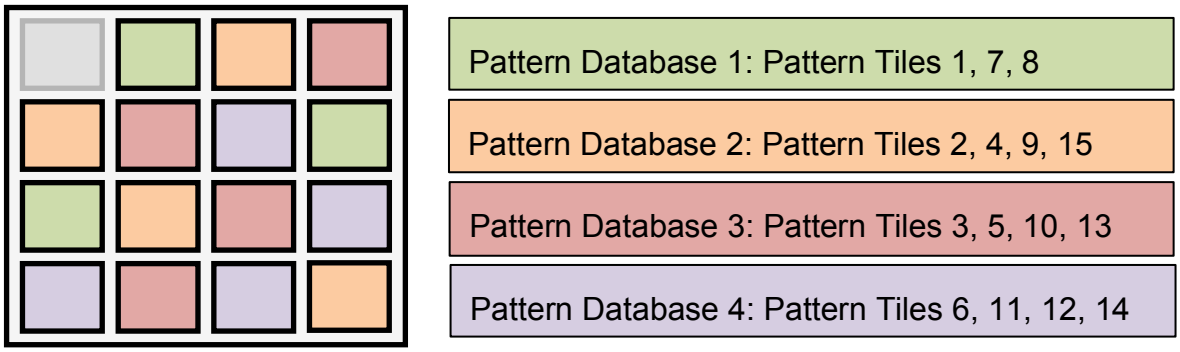


Figure 8-1: Dispersed pattern group

Patterns Arranged in Rows

The next formation I tested was to set each row of the puzzle to a separate pattern

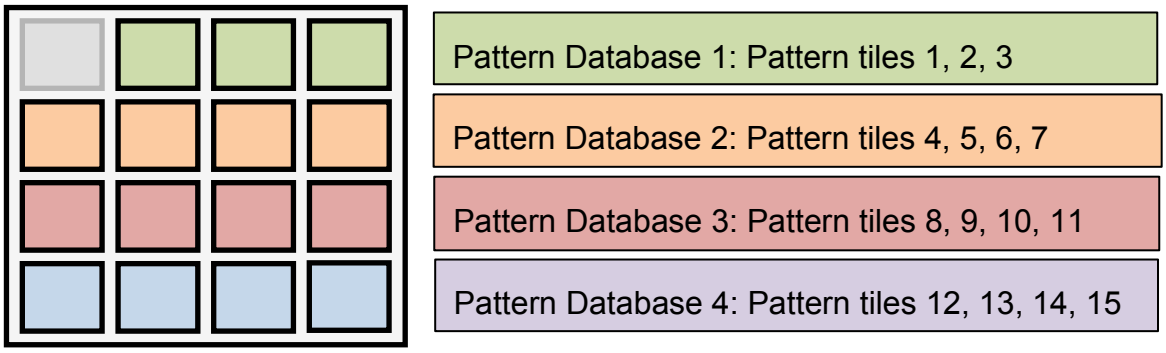


Figure 8-2: Row pattern group

Interlocked Patterns

These individual pattern designs slot together. They all contain a tile that lies between at least three tiles of other patterns.

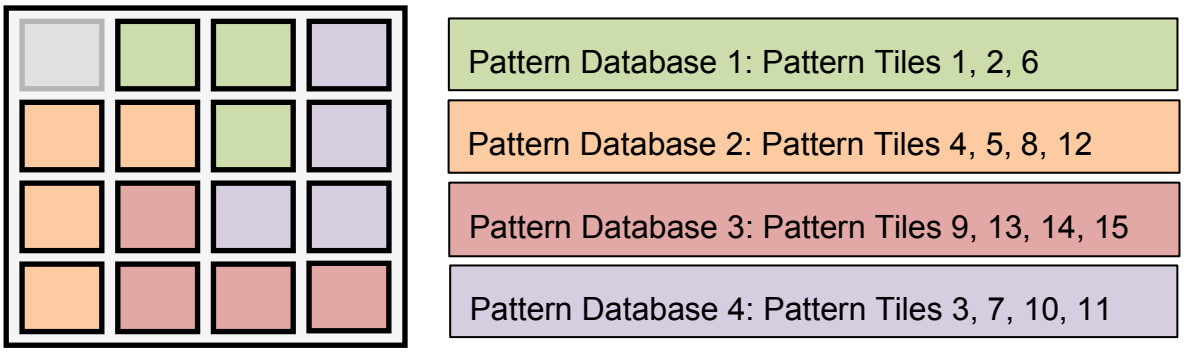


Figure 8-3: Interlocked pattern groups

Patterns Arranged in Blocks:

Using this arrangement I will test how blocked patterns affect the performance of the heuristic.

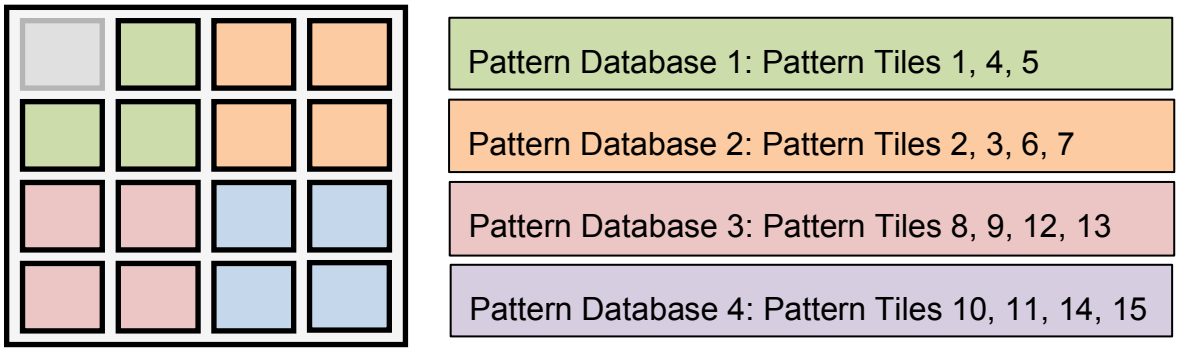


Figure 8-4: Block pattern group

Results:

	Search Time (Seconds)			
Depth of Optimal Solution	Custom Additive Database 4-4-4-3 (Disperse)	Custom Additive Database 4-4-4-3 (Rows)	Custom Additive Database 4-4-4-3 (Blocks)	Custom Additive Database 4-4-4-3 (Interlocked)
38	0.249	0.045	0.001	0.077
39	0.605	0.08	0.02	0.134
42	10.708	1.386	0.474	2.238
46	2.713	0.68	0.056	0.366
48	0.682	0.074	0.071	0.174
50	48.449	10.134	1.51	0.858
52	87.14	5.154	0.894	9.419
54	79.748	14.901	4.905	17.819
56	397.904	69.135	79.977	32.523
60	165.175	47.477	8.753	58.388
62	81.46	21.012	1.668	26.206
62	2811.385	346.445	40.305	663.721

Table 8-1: Run times of a selection of 15-Puzzle test cases with IDA* and new additive databases.

The average search times from Table 8-1 are:

- Custom Additive Database 4-4-4-3 (Dispersed)= 307.184833333333
- Custom Additive Database 4-4-4-3 (Rows)= 43.0435833333333
- Custom Additive Database 4-4-4-3 (Blocks)= 11.5528333333333
- Custom Additive Database 4-4-4-3 (Interlocked)= 67.66025

As expected, the dispersed pattern was outperformed by the more organised patterns. At first it surprised me that the interlocked pattern wasn't more effective. However, although at first this is not intuitive, each additive pattern database only considers moves of its own pattern tiles, and therefore the way in which the patterns are mixed does not seem to make much of a difference. The block pattern out performed the other patterns dramatically, so when developing a pattern database it is probably a good idea to keep tiles in blocks where possible.

Application of Findings

Unfortunately, as discussed previously, due to the memory limitations of the laptop I have access to, the construction of an 8 tile additive pattern is not an option. However, I can certainly apply my findings to create some alternative pattern database techniques using patterns up to 6 tiles.

Alternative 5-5-5 Pattern Group

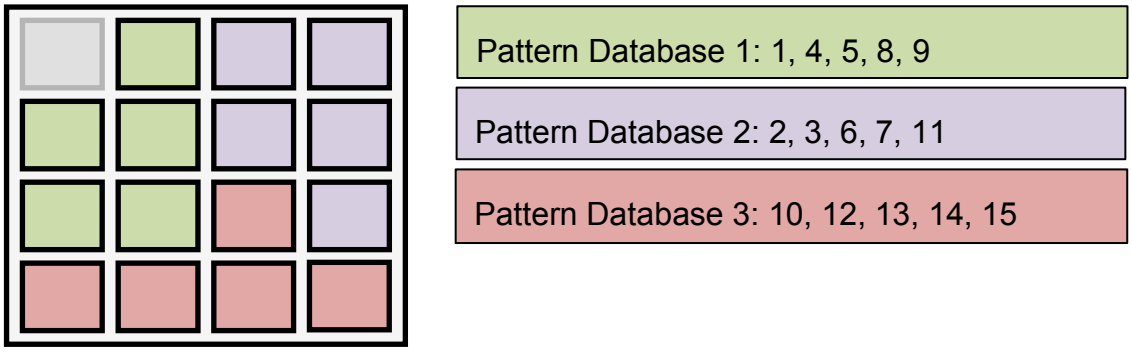


Figure 8-5: Additive 5-5-5 Pattern group

6-5-4 Pattern Group

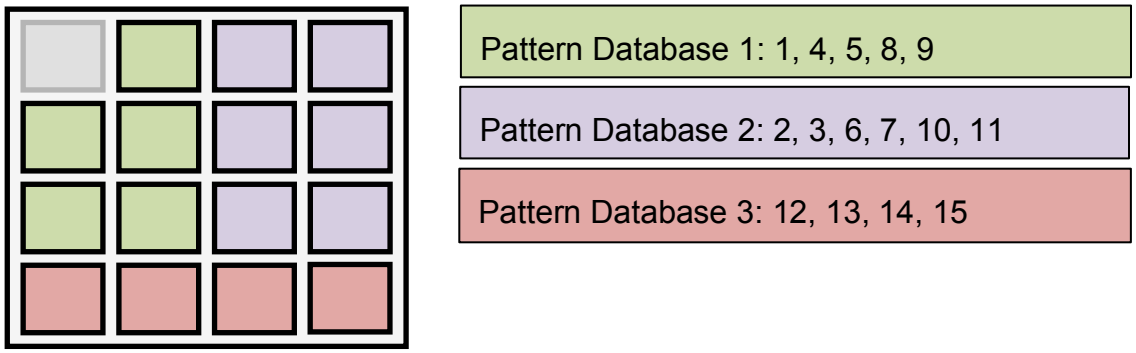


Figure 8-6: Additive 6-4-3 pattern group

7-7-1 Pattern Group

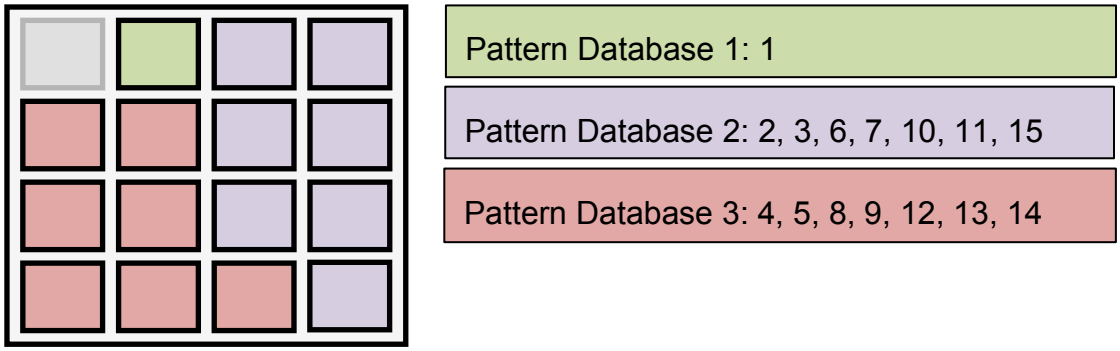


Figure 8-7: Additive 7-7-1 pattern group

Database Combination 1

Not only will I test the performance of the 3 pattern arrangements above, but I will also investigate an alternative use of pattern databases. Rather than only using one group of pattern tile databases, if these patterns are small enough then using multiple groups of pattern databases is an option. I will test this theory by importing into memory both the 5-5-5 and the 6-6-3 databases provided by Felner et. al., 2005, p. 290). I will then use calculation shown in Figure 8-8 as a heuristic.

```
Max(Sum(5-5-5 pattern DB estimates), Sum(6-6-3 pattern DB estimates))
```

Figure 8-8: Heuristic that takes the maximum of two groups of pattern databases

Database Combination 2

An alternative approach that is also worth considering is to use the same technique discussed above, but to compromise its performance to conserve memory. This can be achieved by choosing two groups of pattern databases, which contain a common pattern. For example, my customisation of the 5-5-5 pattern arrangement and 6-5-4 pattern arrangement I designed contain the green pattern in figure 8-9.

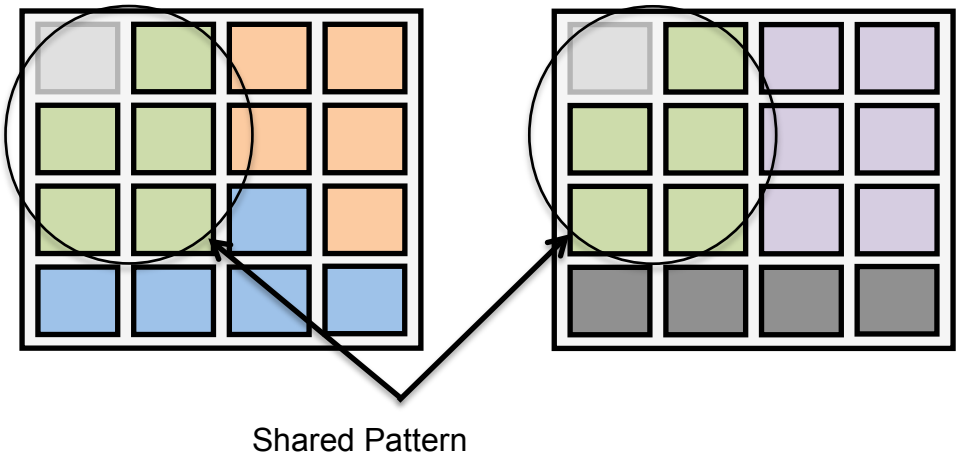


Figure 8-9: Two groups of pattern arrangements that share a common pattern

This common pattern will allow us to conserve form a heuristic by using the calculation shown in Figure 8-10 (where the colours represent the database estimate for the pattern shown in figure 8-9 of that colour):

Green + Max(Orange + Blue, Purple + Grey)

Figure 8-10: Heuristic that takes maximum of two groups of pattern databases which share a common pattern

Results

	Search Time (seconds)						
Depth of Optimal Solution	Additive Database 5-5-5	Additive Database 6-6-3	Alternative Additive Database 5-5-5	Custom Additive Database 6-5-4	Custom Additive Database 7-7-1	Database combination 1	Database combination 2
54	2.134	1.699	2.466	1.151	0.939	1.517	1.232
56	9.99	11.285	7.587	5.781	1.18	6.231	7.94
60	1.844	1.664	1.096	1.448	4.517	1.482	1.155
62	1.272	0.412	0.422	0.842	1.455	0.53	0.501
62	10.434	4.385	9.606	8.041	29.821	7.042	4.723
65	63.82	14.513	45.299	44.301	100.545	42.86	19.588
66	15.564	5.428	8.595	13.078	17.95	9.628	6.274
71	99.145	91.029	138.886	104.597	59.11	95.384	66.979

Table 8-2: Search times for a selection of 15-Puzzle test cases with IDA* and the additive databases which I have designed. I have included the 5-5-5 and 6-6-3 patterns from my initial designs (described by Felner et. al. (2005, p. 290)) for comparison.

- Additive Database 5-5-5: 25.525375
- Additive Database 6-6-3: 16.301875
- Custom Additive Database 5-5-5: 26.744625
- Custom Additive Database 6-5-4: 22.404875
- Custom Additive Database 7-7-1: 26.939625
- Database combination 1: 20.58425
- Database combination 2: 13.549

In many cases as shown in Table 8-2, the 5-5-5 and 6-5-4 patterns which I designed out performed the 5-5-5 and 6-6-3 databases described by Felner et. al., 2005, p. 290). On average however they were not as effective over the range of test cases I chose. In fact, my findings suggest that it is not really possible to rate which of these pattern databases is the most effective, as none of the tested pattern techniques in Table 8-2 outperforms the others for every test case. Therefore it clearly depends on the specific problem at hand. However, what my results do show is that the combination of more than one group of pattern databases by taking their maximum estimate proves to be a very effective heuristic.

Avenues of Possible Future Investigation

If I had more time I would like to investigate possible improvements I could make to my A* algorithm. As shown in my results and analysis, the time taken to expand a node increases at a large rate as the depth of the search space increases. This is due to the fact that I do not use a priority queue. My reasoning for this is because the breadth first search can be susceptible to the exploration of infinite loops, where tiles are moved in a cycle and thus the current state does not move towards the goal. I counteracted this issue by implementing an open list and a closed list, which are checked at every expansion to see whether the node of the expanded node has already been investigated. As these lists grow quickly as the search progresses, the time taken to scan them increases dramatically. If more time were available, I would like to explore the impact of using a priority queue for the open list, and utilise a concept called hash mapping to store previously explored states. This would decrease the time required to check if a node has been visited previously, as hash mapping enables us to perform very quick and efficient lookups to data stored in a priority queue. However, this would compromise our memory usage. I would be very interested in determining whether this would speed up my implementation of the A* algorithm and if so, by how much.

I could perhaps address this increase in memory requirements by reducing the memory required to store each node. This could be achieved by representing each node's tile configuration as an integer rather than an integer array, utilising the ranking and unranking functions, which I developed for my pattern database construction.

I would also like to acquire access to a more powerful computer than my own, which would enable me to investigate additive patterns of over 7 tiles, and even N-Puzzles of a higher dimension than the 15-Puzzle.

9 Conclusion

My original aim was to investigate the most efficient computational methods for solving the N-Puzzle. I set out with the intent of exploring possible avenues and implementing any approaches that I deemed to be the most suitable for the task at hand. I discovered that the two most fitting candidates were the A* and the IDA* search algorithms, which are basically improved versions of the breadth-first and iterative deepening search algorithms respectively which have been adapted to facilitate heuristic guidance. I established that IDA* is a far more effective way of finding an optimal solution within a vast search space, although I am aware, as mentioned in Section 8 that there are substantial improvements I could make to my implementation to the A* algorithm.

Another concept which I have discovered is that when using an informed search, for example the A* of the IDA* search, the quality of the heuristic is vital. I learnt that the Manhattan distance and linear conflict calculations are the most accurate heuristics without the use of an external database. When utilising databases, my results demonstrated that the implementation of additive pattern databases was the best approach, due to their accuracy and scalability, making them the best heuristics available for solving N-Puzzles out of the heuristics I have experimented with. Non-additive databases also performed rather well. However, they proved to less effective for complex situations and nowhere near as scalable as additive pattern databases.

I also wanted to develop and implement some ideas and approaches of my own in an effort to improve my implementations of existing algorithms that I had learnt about during my research. I first learned that patterns with bunched tiles are the most effective. I used my findings to create some patterns of my own. In some cases these outperformed the additive patterns previously researched, but on average they were not as effective. I then experimented with the use of multiple additive pattern groups, taking the maximum estimate from each group. This proved to be the most effective heuristic in every case. However, all databases from each pattern group needed to be imported into memory at each time, which for larger problems could use excessive memory.

In an attempt to compromise some performance in order to conserve memory I experimented with implementing the same concept but with pattern groups, which contained a common pattern, therefore reducing the number of pattern databases that needed to be loaded into memory and therefore slightly reducing the memory requirements. This was less effective, as can be expected, however it may be a technique to consider with larger database where the number of required databases begins to cause problems.

10 Reflection

Given that this was the first piece of formal research I have performed, I am very pleased with what I have accomplished. I feel I have worked to the best of my ability and that I have achieved all that I set out to achieve. That being said, as with any piece of work, there are many lessons that I have learned, and there are many improvements that I could make if I were to perform this exercise again.

One mistake I feel I made was in spending too much time writing my program. Although I still managed to reach my overall targets and meet my deadline, it required me to work extremely hard later on in the project to ensure my written work was also of a high standard. Had I managed my time more efficiently, this would have averted a lot of the stress that I faced in the final stages of the dissertation. In future I will try my best to set deadlines for each stage of the project and ensure I keep to these deadlines so that any weaknesses I have in one area will not have a direct impact on other facets of my project.

During my research, I purposely covered a very wide range of theory. Since this is my first piece of formal research, I worked industriously to develop a broad understanding of the subject matter in its entirety. However, in future projects, I feel I should limit my scope a little more, allowing me to cover a smaller area in far more depth and therefore giving me more of an opportunity to develop and implement more ideas of my own.

An important lesson learned from this project is the importance of careful planning at each step of the way. Although at major project milestones I made sure I had planned for the next stages, I found that on times I fell into the temptation of writing code based on an idea that I had formed in my head. I found that often, this approach resulted in me having to revert to previous versions. This was usually due to issues that probably could have been anticipated, had I formally completed all designs prior to attempting to implement them. I have learnt from this not only for future research projects, but also for any future developing projects in the “real world”.

When I was initially briefed about this dissertation, and was told to decide on a topic, which I wanted to research I was a little sceptical about finding a problem which would maintain my interest for the duration of the course. As soon as I discovered this topic I instantly knew it was the one for me. As a mathematician I find the mathematical aspect of the problem very fascinating and found myself reading around the subject even during my breaks, The programming side of the project was even more stimulating, At many points during my dissertation I found myself thinking about how privileged I was to be researching a subject which I find truly fascinating. The subject matter certainly kept my interest right up to the submission date, so much so that I intend learning far more about Artificial Intelligence in the future.

During this exercise I have not only developed a deeper understanding of the subject matter of my dissertation, but I have also learned a great deal about key skills such as time management, organisation, problem solving and the ability to work under pressure. Additionally, I have gained a significant grasp on a number of research methods, which I will no-doubt utilise throughout my career.

11 Appendix

Appendix A1- 8 Puzzle Test Cases

Difficulty group	Test Case	Depth of Optimal Solution	Tile Arrangement
0-10	TC1	5	[[3, 1, 2], [6, 4, 0], [7, 8, 5]]
	TC2	8	[[3, 1, 2], [7, 5, 8], [4, 6, 0]]
	TC3	10	[[0, 6, 2], [1, 3, 5], [7, 4, 8]]
	TC4	10	[[0, 4, 1], [3, 6, 2], [7, 8, 5]]
11-20	TC5	16	[[2, 3, 4], [1, 0, 6], [7, 8, 5]]
	TC6	18	[[6, 2, 5], [4, 0, 3], [7, 8, 1]]
	TC7	20	[[0, 8, 3], [2, 4, 1], [6, 5, 7]]
	TC8	20	[[3, 8, 2], [1, 5, 4], [0, 6, 7]]
21-30	TC9	22	[[5, 6, 1], [8, 3, 4], [0, 2, 7]]
	TC10	26	[[2, 6, 7], [5, 0, 8], [1, 4, 3]]
	TC11	26	[[6, 4, 7], [5, 1, 3], [0, 2, 8]]
	TC12	28	[[8, 2, 4], [3, 7, 1], [0, 5, 6]]
31-40	TC13	31	[[8, 0, 6], [5, 4, 7], [2, 3, 1]]

Appendix A2- 15 Puzzle Test Cases

Difficulty Group	Test Case	Depth of Optimal Solution	Tile Arrangement
0-10	TC1	5	[[1, 5, 2, 3], [4, 9, 6, 7], [12, 8, 10, 11], [0, 13, 14, 15]]
	TC2	8	[[5, 4, 2, 3], [1, 6, 10, 7], [8, 9, 0, 11], [12, 13, 14, 15]]
	TC3	10	[[0, 5, 3, 7], [1, 4, 6, 2], [8, 9, 10, 11], [12, 13, 14, 15]]
	TC4	10	[[1, 5, 2, 3], [4, 9, 6, 7], [13, 10, 14, 11], [8, 0, 12, 15]]
11-20	TC5	15	[[1, 6, 5, 2], [4, 9, 7, 3], [8, 13, 15, 10], [12, 14, 0, 11]]
	TC6	18	[[0, 1, 7, 2], [4, 6, 10, 3], [8, 5, 11, 15], [9, 12, 13, 14]]
	TC7	18	[[1, 2, 9, 3], [4, 0, 5, 7], [8, 10, 14, 11], [12, 13, 15, 6]]
	TC8	20	[[4, 1, 6, 2], [8, 9, 5, 3], [13, 12, 0, 11], [14, 10, 15, 7]]
21-30	TC9	22	[[5, 2, 3, 7], [1, 0, 4, 11], [6, 9, 10, 15], [8, 12, 13, 14]]
	TC10	23	[[1, 2, 6, 3], [8, 4, 5, 7], [12, 15, 10, 0], [9, 14, 13, 11]]
	TC11	26	[[4, 1, 3, 7], [2, 0, 10, 11], [8, 9, 13, 5], [12, 6, 14, 15]]
	TC12	28	[[5, 4, 6, 7], [12, 0, 1, 2], [8, 3, 9, 11], [13, 14, 15, 10]]
31-40	TC13	34	[[6, 8, 3, 11], [2, 7, 9, 15], [4, 1, 14, 10], [12, 5, 13, 0]]
	TC14	36	[[5, 4, 7, 2], [10, 1, 3, 0], [12, 8, 9, 14], [13, 11, 6, 15]]
	TC15	38	[[3, 8, 6, 2], [4, 9, 1, 7], [13, 15, 0, 5], [14, 11, 12, 10]]
	TC16	39	[[5, 7, 9, 10], [6, 1, 2, 3], [4, 12, 13, 0], [8, 15, 14, 11]]
41-50	TC17	42	[[4, 6, 5, 7], [13, 3, 14, 1], [9, 8, 0, 15], [12, 2, 10, 11]]
	TC18	46	[[4, 3, 12, 6], [14, 9, 7, 11], [5, 8, 2, 13], [1, 0, 15, 10]]
	TC19	48	[[9, 4, 8, 2], [13, 14, 3, 1], [15, 7, 10, 5], [11, 0, 12, 6]]
	TC20	50	[[2, 8, 0, 11], [13, 9, 5, 14], [6, 1, 10, 7], [4, 15, 12, 3]]
51-60	TC21	52	[[3, 15, 11, 6], [9, 8, 2, 14], [1, 13, 5, 10], [4, 0, 7, 12]]
	TC22	54	[[11, 7, 10, 3], [4, 13, 12, 1], [9, 2, 5, 8], [6, 0, 15, 14]]
	TC23	56	[[1, 9, 8, 13], [4, 0, 5, 11], [6, 10, 14, 2], [3, 15, 12, 7]]
	TC24	60	[[11, 9, 7, 15], [12, 14, 8, 0], [3, 5, 6, 2], [10, 13, 1, 4]]
61-70	TC25	62	[[9, 14, 6, 5], [15, 7, 13, 0], [11, 8, 4, 1], [2, 10, 3, 12]]
	TC26	62	[[14, 7, 0, 1], [15, 6, 5, 10], [9, 3, 13, 12], [8, 11, 2, 4]]
	TC27	65	[[15, 9, 3, 14], [7, 13, 0, 6], [11, 8, 5, 4], [2, 10, 12, 1]]
	TC28	66	[[15, 14, 6, 5], [7, 9, 13, 0], [11, 8, 4, 1], [2, 10, 3, 12]]
71-80	TC29	71	[[15, 11, 10, 12], [14, 2, 5, 9], [6, 7, 13, 1], [3, 8, 0, 4]]
	TC30	74	[[15, 11, 13, 12], [14, 6, 10, 5], [3, 2, 9, 1], [7, 0, 8, 4]]
	TC31	80	[[15, 11, 13, 12], [14, 10, 9, 5], [2, 6, 8, 1], [3, 7, 4, 0]]
	TC32	80	[[15, 14, 13, 12], [10, 11, 8, 9], [2, 6, 5, 1], [3, 7, 4, 0]]

Appendix B1- Breadth-First Search Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	99	192200 nanoseconds	1940	N/A	99	100/100
TC2	8	324	00:00:00.001	4000	N/A	324	100/100
TC3	10	790	00:00:00.007	9520	N/A	790	100/100
TC4	10	820	00:00:00.008	10200	N/A	820	100/100
TC5	16	15456	00:00:03.068	199000	N/A	15456	10/10
TC6	18	44166	00:00:35.343	800000	N/A	44166	10/10
TC7	20	63155	00:01:16.614	1210000	N/A	63155	10/10
TC8	20	70935	00:01:39.918	1410000	N/A	70935	10/10
TC9	22	102670	00:03:55.831	2300000	N/A	102670	1/1
TC10	26	177668	00:13:46.846	4650000	N/A	177668	1/1
TC11	26	171946	00:11:57.877	4180000	N/A	171946	1/1
TC12	28	180737	00:14:05.237	4680000	N/A	180737	1/1
TC13	31	181440	00:15:48.685	5230000	N/A	181440	1/1

Appendix B3- Iterative-Deepening Search Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	83	54370 nanoseconds	655	N/A	15	100/100
TC2	8	401	212020 nanoseconds	528	N/A	21	100/100
TC3	10	1007	513540 nanoseconds	509	N/A	26	100/100
TC4	10	1112	577890 nanoseconds	519	N/A	26	100/100
TC5	16	31561	00:00:00.017	549	N/A	39	100/100
TC6	18	226032	00:00:00.127	564	N/A	42	100/100
TC7	20	320180	00:00:00.183	573	N/A	45	100/100
TC8	20	501330	00:00:00.297	594	N/A	45	100/100
TC9	22	767401	00:00:00.464	604	N/A	50	100/100
TC10	26	12408496	00:00:07.038	567	N/A	58	10/10
TC11	26	5625570	00:00:03.188	566	N/A	56	10/10
TC12	28	16308629	00:00:09.941	609	N/A	61	10/10
TC13	31	51806217	00:00:30.635	591	N/A	64	10/10

Appendix C1- A* with Misplaced Tiles Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	83	54370 nanoseconds	655	N/A	15	100/100
TC2	8	401	212020 nanoseconds	528	N/A	21	100/100
TC3	10	1007	513540 nanoseconds	509	N/A	26	100/100
TC4	10	1112	577890 nanoseconds	519	N/A	26	100/100
TC5	16	31561	00:00:00.017	549	N/A	39	100/100
TC6	18	226032	00:00:00.127	564	N/A	42	100/100
TC7	20	320180	00:00:00.183	573	N/A	45	100/100
TC8	20	501330	00:00:00.297	594	N/A	45	100/100
TC9	22	767401	00:00:00.464	604	N/A	50	100/100
TC10	26	12408496	00:00:07.038	567	N/A	58	10/10
TC11	26	5625570	00:00:03.188	566	N/A	56	10/10
TC12	28	16308629	00:00:09.941	609	N/A	61	10/10
TC13	31	51806217	00:00:30.635	591	N/A	64	10/10

Appendix C2- A* with Checkerboard Misplaced Tiles Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	**5**	10	49809 nanoseconds	4980.0	N/A	10	1000/1000
TC2	**8**	19	37648 nanoseconds	1980.0	N/A	19	1000/1000
TC3	**10**	58	132984 nanoseconds	2290.0	N/A	58	1000/1000
TC4	**10**	37	64343 nanoseconds	1740.0	N/A	37	1000/1000
TC5	**16**	454	00:00:00.002	5800.0	N/A	454	1000/1000
TC6	**18**	1143	00:00:00.015	13900.0	N/A	1143	1000/1000
TC7	**20**	1987	00:00:00.046	23400.0	N/A	1987	1000/1000
TC8	**20**	2668	00:00:00.090	33900.0	N/A	2668	1000/1000
TC9	**22**	4192	00:00:00.231	55200.0	N/A	4192	10/10
TC10	**26**	27164	00:00:15.319	564000.0	N/A	27164	10/10
TC11	**26**	23371	00:00:10.314	441000.0	N/A	23371	10/10
TC12	**28**	49543	00:01:00.100	1210000.0	N/A	49543	10/10
TC13	**31**	110396	00:05:42.057	3100000.0	N/A	110396	1/1

Appendix C3- A* with Relaxed Adjacency Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	10	19343 nanoseconds	1930.0	N/A	10	1000/1000
TC2	8	31	39559 nanoseconds	1280.0	N/A	31	1000/1000
TC3	10	78	111681 nanoseconds	1430.0	N/A	78	1000/1000
TC4	10	52	62634 nanoseconds	1200.0	N/A	52	1000/1000
TC5	16	923	00:00:00.011	12800.0	N/A	923	1000/1000
TC6	18	2482	00:00:00.083	33800.0	N/A	2482	1000/1000
TC7	20	4453	00:00:00.269	60600.0	N/A	4453	1000/1000
TC8	20	5160	00:00:00.355	69000.0	N/A	5160	1000/1000
TC9	22	9944	00:00:01.351	136000.0	N/A	9944	10/10
TC10	26	56038	00:01:10.238	1250000.0	N/A	56038	10/10
TC11	26	47000	00:00:45.798	974000.0	N/A	47000	10/10
TC12	28	86307	00:03:18.472	2300000.0	N/A	86307	10/10
TC13	31	155641	00:13:02.529	5030000.0	N/A	155641	1/1

Appendix C4- A* with Number of Tiles out of Row and Column Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	10	22556 nanoseconds	2260.0	N/A	10	1000/1000
TC2	8	25	29516 nanoseconds	1180.0	N/A	25	1000/1000
TC3	10	66	92475 nanoseconds	1400.0	N/A	66	1000/1000
TC4	10	37	45660 nanoseconds	1230.0	N/A	37	1000/1000
TC5	16	472	00:00:00.002	5910.0	N/A	472	1000/1000
TC6	18	1239	00:00:00.019	15300.0	N/A	1239	1000/1000
TC7	20	1957	00:00:00.047	24400.0	N/A	1957	1000/1000
TC8	20	2992	00:00:00.108	36300.0	N/A	2992	1000/1000
TC9	22	4422	00:00:00.243	55100.0	N/A	4422	10/10
TC10	26	27363	00:00:14.440	528000.0	N/A	27363	10/10
TC11	26	22647	00:00:09.033	399000.0	N/A	22647	10/10
TC12	28	50381	00:01:03.169	1250000.0	N/A	50381	10/10
TC13	31	87834	00:03:39.775	2500000.0	N/A	87834	10/10

Appendix C5- A* with Manhattan Distance Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	10	26756 nanoseconds	2680.0	N/A	10	1000/1000
TC2	8	19	24637 nanoseconds	1300.0	N/A	19	1000/1000
TC3	10	43	62072 nanoseconds	1440.0	N/A	43	1000/1000
TC4	10	31	33561 nanoseconds	1080.0	N/A	31	1000/1000
TC5	16	157	391064 nanoseconds	2490.0	N/A	157	1000/1000
TC6	18	410	00:00:00.002	5330.0	N/A	410	1000/1000
TC7	20	570	00:00:00.004	7580.0	N/A	570	1000/1000
TC8	20	1092	00:00:00.015	14100.0	N/A	1092	1000/1000
TC9	22	958	00:00:00.012	13400.0	N/A	958	10/10
TC10	26	5226	00:00:00.337	64600.0	N/A	5226	10/10
TC11	26	4735	00:00:00.277	58600.0	N/A	4735	10/10
TC12	28	15136	00:00:03.311	219000.0	N/A	15136	10/10
TC13	31	28951	00:00:17.598	608000.0	N/A	28951	10/10

Appendix C6- A* with Manhattan Distance and Linear Conflict Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	10	33499 nanoseconds	3350.0	N/A	10	1000/1000
TC2	8	19	27735 nanoseconds	1460.0	N/A	19	1000/1000
TC3	10	43	60636 nanoseconds	1410.0	N/A	43	1000/1000
TC4	10	28	38521 nanoseconds	1380.0	N/A	28	1000/1000
TC5	16	129	293334 nanoseconds	2270.0	N/A	129	1000/1000
TC6	18	333	00:00:00.001	4480.0	N/A	333	1000/1000
TC7	20	389	00:00:00.001	5130.0	N/A	389	1000/1000
TC8	20	766	00:00:00.007	10200.0	N/A	766	1000/1000
TC9	22	423	00:00:00.004	10200.0	N/A	423	10/10
TC10	26	3207	00:00:00.146	45600.0	N/A	3207	10/10
TC11	26	2779	00:00:00.098	35400.0	N/A	2779	10/10
TC12	28	9180	00:00:01.130	123000.0	N/A	9180	10/10
TC13	31	18431	00:00:05.121	278000.0	N/A	18431	10/10

Appendix C7- A* with Database Lookup Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	10	19930 nanoseconds	1990.0	739631 nanoseconds	10	1000/1000
TC2	8	18	23825 nanoseconds	1320.0	705262 nanoseconds	18	1000/1000
TC3	10	29	39190 nanoseconds	1350.0	714728 nanoseconds	29	1000/1000
TC4	10	19	23536 nanoseconds	1240.0	613192 nanoseconds	19	1000/1000
TC5	16	32	39134 nanoseconds	1220.0	609210 nanoseconds	32	1000/1000
TC6	18	39	54657 nanoseconds	1400.0	575123 nanoseconds	39	1000/1000
TC7	20	76	117264 nanoseconds	1540.0	542364 nanoseconds	76	1000/1000
TC8	20	38	47504 nanoseconds	1250.0	566001 nanoseconds	38	1000/1000
TC9	22	45	394100 nanoseconds	8760.0	00:00:00.001	45	10/10
TC10	26	135	478000 nanoseconds	3540.0	692600 nanoseconds	135	10/10
TC11	26	94	172900 nanoseconds	1840.0	517700 nanoseconds	94	10/10
TC12	28	464	00:00:00.002	5850.0	536300 nanoseconds	464	10/10
TC13	31	879	00:00:00.010	11900.0	550700 nanoseconds	879	10/10

Appendix D1- IDA* with Misplaced Tiles Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	27081 nanoseconds	4510.0	N/A	10	1000/1000
TC2	8	21	42899 nanoseconds	2040.0	N/A	18	1000/1000
TC3	10	65	64705 nanoseconds	995.0	N/A	21	1000/1000
TC4	10	26	44170 nanoseconds	1700.0	N/A	19	1000/1000
TC5	16	804	480895 nanoseconds	598.0	N/A	32	1000/1000
TC6	18	5863	00:00:00.003	589.0	N/A	39	1000/1000
TC7	20	7948	00:00:00.004	601.0	N/A	40	1000/1000
TC8	20	15087	00:00:00.009	624.0	N/A	40	1000/1000
TC9	22	17076	00:00:00.010	608.0	N/A	45	1000/1000
TC10	26	261749	00:00:00.153	586.0	N/A	53	1000/1000
TC11	26	124878	00:00:00.072	578.0	N/A	51	1000/1000
TC12	28	379722	00:00:00.235	619.0	N/A	56	1000/1000
TC13	31	971420	00:00:00.634	653.0	N/A	61	1000/1000

Appendix D2- IDA* with Checkerboard Misplaced Tiles Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	49001 nanoseconds	8170.0	N/A	10	1000/1000
TC2	8	10	27074 nanoseconds	2710.0	N/A	18	1000/1000
TC3	10	30	40103 nanoseconds	1340.0	N/A	21	1000/1000
TC4	10	12	22216 nanoseconds	1850.0	N/A	19	1000/1000
TC5	16	165	161417 nanoseconds	978.0	N/A	32	1000/1000
TC6	18	1223	966446 nanoseconds	790.0	N/A	39	1000/1000
TC7	20	1669	00:00:00.001	755.0	N/A	40	1000/1000
TC8	20	3723	00:00:00.002	749.0	N/A	40	1000/1000
TC9	22	2979	00:00:00.002	755.0	N/A	45	1000/1000
TC10	26	44479	00:00:00.032	728.0	N/A	53	1000/1000
TC11	26	22886	00:00:00.017	780.0	N/A	51	1000/1000
TC12	28	73935	00:00:00.052	714.0	N/A	56	1000/1000
TC13	31	267266	00:00:00.188	706.0	N/A	61	1000/1000

Appendix D3- IDA* with Relaxed Adjacency Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	21711 nanoseconds	3620.0	N/A	10	1000/1000
TC2	8	15	33473 nanoseconds	2230.0	N/A	18	1000/1000
TC3	10	42	44558 nanoseconds	1060.0	N/A	21	1000/1000
TC4	10	19	26921 nanoseconds	1420.0	N/A	19	1000/1000
TC5	16	377	284084 nanoseconds	753.0	N/A	32	1000/1000
TC6	18	3055	00:00:00.002	757.0	N/A	39	1000/1000
TC7	20	4005	00:00:00.003	752.0	N/A	40	1000/1000
TC8	20	7952	00:00:00.006	789.0	N/A	40	1000/1000
TC9	22	8144	00:00:00.006	793.0	N/A	45	1000/1000
TC10	26	125980	00:00:00.098	781.0	N/A	53	1000/1000
TC11	26	59330	00:00:00.046	787.0	N/A	51	1000/1000
TC12	28	185717	00:00:00.142	766.0	N/A	56	1000/1000
TC13	31	798517	00:00:00.622	779.0	N/A	61	1000/1000

Appendix D4- IDA* with Number of Tiles out of Row and Column Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	14353 nanoseconds	2390.0	N/A	10	1000/1000
TC2	8	10	32801 nanoseconds	3280.0	N/A	18	1000/1000
TC3	10	36	45639 nanoseconds	1270.0	N/A	21	1000/1000
TC4	10	12	25465 nanoseconds	2120.0	N/A	19	1000/1000
TC5	16	179	131361 nanoseconds	733.0	N/A	32	1000/1000
TC6	18	1590	00:00:00.001	700.0	N/A	39	1000/1000
TC7	20	1786	00:00:00.001	701.0	N/A	40	1000/1000
TC8	20	4261	00:00:00.003	726.0	N/A	40	1000/1000
TC9	22	3560	00:00:00.002	700.0	N/A	45	1000/1000
TC10	26	44631	00:00:00.033	749.0	N/A	53	1000/1000
TC11	26	21301	00:00:00.016	759.0	N/A	51	1000/1000
TC12	28	79301	00:00:00.056	716.0	N/A	56	1000/1000
TC13	31	122673	00:00:00.085	694.0	N/A	61	1000/1000

Appendix D5- IDA* with Manhattan Distance Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	16734 nanoseconds	2790.0	N/A	10	1000/1000
TC2	8	10	16998 nanoseconds	1700.0	N/A	18	1000/1000
TC3	10	20	32175 nanoseconds	1610.0	N/A	21	1000/1000
TC4	10	12	19095 nanoseconds	1590.0	N/A	19	1000/1000
TC5	16	50	56131 nanoseconds	1120.0	N/A	32	1000/1000
TC6	18	436	411154 nanoseconds	943.0	N/A	39	1000/1000
TC7	20	407	379279 nanoseconds	931.0	N/A	40	1000/1000
TC8	20	1414	00:00:00.001	976.0	N/A	38	1000/1000
TC9	22	661	636643 nanoseconds	963.0	N/A	45	1000/1000
TC10	26	5430	00:00:00.005	1040.0	N/A	51	1000/1000
TC11	26	2803	00:00:00.002	976.0	N/A	51	1000/1000
TC12	28	12277	00:00:00.012	1030.0	N/A	56	1000/1000
TC13	31	13582	00:00:00.012	941.0	N/A	61	1000/1000

Appendix D6- IDA* with Manhattan Distance and Linear Conflict Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	13846 nanoseconds	2310.0	N/A	10	1000/1000
TC2	8	10	24065 nanoseconds	2410.0	N/A	18	1000/1000
TC3	10	20	36713 nanoseconds	1840.0	N/A	21	1000/1000
TC4	10	12	21241 nanoseconds	1770.0	N/A	19	1000/1000
TC5	16	42	47616 nanoseconds	1130.0	N/A	32	1000/1000
TC6	18	336	364367 nanoseconds	1080.0	N/A	39	1000/1000
TC7	20	224	244586 nanoseconds	1090.0	N/A	40	1000/1000
TC8	20	995	00:00:00.001	1130.0	N/A	38	1000/1000
TC9	22	238	253611 nanoseconds	1070.0	N/A	45	1000/1000
TC10	26	3382	00:00:00.003	1160.0	N/A	51	1000/1000
TC11	26	1451	00:00:00.001	1150.0	N/A	51	1000/1000
TC12	28	6234	00:00:00.007	1200.0	N/A	56	1000/1000
TC13	31	7355	00:00:00.008	1180.0	N/A	61	1000/1000

Appendix D7- IDA* with Database Lookup Heuristic Results for 8-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	16325 nanoseconds	2720.0	883828 nanoseconds	10	1000/1000
TC2	8	9	19704 nanoseconds	2190.0	799622 nanoseconds	18	1000/1000
TC3	10	11	26092 nanoseconds	2370.0	807124 nanoseconds	21	1000/1000
TC4	10	11	19946 nanoseconds	1810.0	703327 nanoseconds	19	1000/1000
TC5	16	17	24456 nanoseconds	1440.0	646865 nanoseconds	32	1000/1000
TC6	18	19	29466 nanoseconds	1550.0	617761 nanoseconds	39	1000/1000
TC7	20	21	29825 nanoseconds	1420.0	606087 nanoseconds	40	1000/1000
TC8	20	21	28598 nanoseconds	1360.0	591450 nanoseconds	38	1000/1000
TC9	22	23	37322 nanoseconds	1620.0	583880 nanoseconds	45	1000/1000
TC10	26	27	35119 nanoseconds	1300.0	585527 nanoseconds	51	1000/1000
TC11	26	27	39833 nanoseconds	1480.0	558101 nanoseconds	51	1000/1000
TC12	28	29	39753 nanoseconds	1370.0	543529 nanoseconds	56	1000/1000
TC13	31	32	41670 nanoseconds	1300.0	535771 nanoseconds	61	1000/1000

Appendix D1- A* with Misplaced Tiles Heuristic Results for 15–Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	10300 nanoseconds	792	N/A	13	100/100
TC2	8	85	125990 nanoseconds	1480	N/A	85	100/100
TC3	10	116	243390 nanoseconds	2100	N/A	116	100/100
TC4	10	98	183310 nanoseconds	1870	N/A	98	100/100
TC5	15	224	737790 nanoseconds	3290	N/A	224	100/100
TC6	18	1409	00:00:00.026	18600	N/A	1409	100/100
TC7	18	9054	00:00:01.039	115000	N/A	9054	100/100
TC8	20	5721	00:00:00.449	78500	N/A	5721	100/100
TC9	22	23143	00:00:09.896	428000	N/A	23143	100/100
TC10	23	33590	00:00:23.145	689000	N/A	33590	10/10
TC11	26	N/A	>2 hours	N/A	N/A	N/A	0/1

Appendix D2- A* with Checkerboard Misplaced Tiles Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	91740 nanoseconds	7060	N/A	13	100/100
TC2	8	30	107380 nanoseconds	3580	N/A	30	100/100
TC3	10	38	70120 nanoseconds	1850	N/A	38	100/100
TC4	10	31	62930 nanoseconds	2030	N/A	31	100/100
TC5	15	73	170870 nanoseconds	2340	N/A	73	100/100
TC6	18	103	264150 nanoseconds	2560	N/A	103	100/100
TC7	18	1957	00:00:00.052	26800	N/A	1957	100/100
TC8	20	368	00:00:00.002	6360	N/A	368	100/100
TC9	22	2476	00:00:00.075	30500	N/A	2476	100/100
TC10	23	6596	00:00:00.639	96900	N/A	6596	1/1
TC11	26	97963	00:03:45.989	2310000	N/A	97963	1/1
TC12	28	44854	00:00:46.294	1030000	N/A	44854	1/1
TC13	34	N/A	N/A	N/A	N/A	N/A	0/1

Appendix D3- A* with Relaxed Adjacency Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	12560 nanoseconds	966	N/A	13	100/100
TC2	8	50	59480 nanoseconds	1190	N/A	50	100/100
TC3	10	59	82580 nanoseconds	1400	N/A	59	100/100
TC4	10	51	73660 nanoseconds	1440	N/A	51	100/100
TC5	15	194	600440 nanoseconds	3100	N/A	194	100/100
TC6	18	446	00:00:00.002	5840	N/A	446	100/100
TC7	18	4229	00:00:00.250	59300	N/A	4229	100/100
TC8	20	2373	00:00:00.087	36700	N/A	2373	100/100
TC9	22	9532	00:00:01.310	137000	N/A	9532	100/100
TC10	23	23292	00:00:11.220	482000	N/A	23292	10/10
TC11	26	307419	00:27:44.793	5420000	N/A	307419	1/1
TC12	28	351111	00:39:05.853	6680000	N/A	351111	1/1
TC13	34	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix D4- A* with Number of Tiles out of Row and Column Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	12520 nanoseconds	963	N/A	13	100/100
TC2	8	30	28280 nanoseconds	942	N/A	30	100/100
TC3	10	49	68530 nanoseconds	1400	N/A	49	100/100
TC4	10	46	60100 nanoseconds	1310	N/A	46	100/100
TC5	15	61	80550 nanoseconds	1320	N/A	61	100/100
TC6	18	220	726110 nanoseconds	3300	N/A	220	100/100
TC7	18	1720	00:00:00.040	23300	N/A	1720	100/100
TC8	20	947	00:00:00.013	14000	N/A	947	100/100
TC9	22	3977	00:00:00.192	48400	N/A	3977	100/100
TC10	23	6140	00:00:00.541	88100	N/A	6140	10/10
TC11	26	70563	00:01:56.709	1650000	N/A	70563	1/1
TC12	28	43817	00:00:45.015	1030000	N/A	43817	1/1
TC13	34	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix D5- A* with Manhattan Distance Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	32986 nanoseconds	2540	N/A	13	1000/1000
TC2	8	30	56069 nanoseconds	1870	N/A	30	1000/1000
TC3	10	38	64828 nanoseconds	1710	N/A	38	1000/1000
TC4	10	31	46700 nanoseconds	1510	N/A	31	100/100
TC5	15	61	121281 nanoseconds	1990	N/A	61	1000/1000
TC6	18	69	143665 nanoseconds	2080	N/A	69	1000/1000
TC7	18	424	00:00:00.002	6480	N/A	424	1000/1000
TC8	20	150	460212 nanoseconds	3070	N/A	150	1000/1000
TC9	22	875	00:00:00.009	11400	N/A	875	1000/1000
TC10	23	2478	00:00:00.115	46400	N/A	2478	10/10
TC11	26	8516	00:00:01.033	121000	N/A	8516	10/10
TC12	28	3617	00:00:00.161	44700	N/A	3617	10/10
TC13	34	19259	00:00:05.721	297000	N/A	19259	10/10
TC14	36	248685	00:18:33.487	4480000	N/A	248685	1/1
TC15	38	156022	00:11:26.979	4400000	N/A	156022	1/1
TC16	39	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix D6- A* with Manhattan Distance and Linear Conflict Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	27625 nanoseconds	2130	N/A	13	1000/1000
TC2	8	30	105696 nanoseconds	3520	N/A	30	1000/1000
TC3	10	38	68365 nanoseconds	1800	N/A	38	1000/1000
TC4	10	31	47870 nanoseconds	1540	N/A	31	100/100
TC5	15	61	114134 nanoseconds	1870	N/A	61	1000/1000
TC6	18	66	139103 nanoseconds	2110	N/A	66	1000/1000
TC7	18	363	00:00:00.002	5580	N/A	363	1000/1000
TC8	20	46	95785 nanoseconds	2080	N/A	46	1000/1000
TC9	22	479	00:00:00.003	7020	N/A	479	1000/1000
TC10	23	1141	00:00:00.022	19900	N/A	1141	10/10
TC11	26	4928	00:00:00.341	69400	N/A	4928	10/10
TC12	28	1205	00:00:00.019	16300	N/A	1205	10/10
TC13	34	4049	00:00:00.203	50200	N/A	4049	10/10
TC14	36	97151	00:03:54.842	2420000	N/A	97151	1/1
TC15	38	14543	00:00:03.414	235000	N/A	14543	1/1
TC16	39	128406	00:07:35.855	3550000	N/A	128406	1/1
TC17	42	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix D7- A* with Fringe Pattern Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	28	85200 nanoseconds	3040	00:00:01.022	28	10/10
TC2	8	263	00:00:00.001	6720	00:00:01.027	263	10/10
TC3	10	129	00:00:00.002	22800	00:00:01.790	129	10/10
TC4	10	31	127500 nanoseconds	4110	00:00:01.179	31	10/10
TC5	15	42	106200 nanoseconds	2530	00:00:01.024	42	10/10
TC6	18	145	00:00:00.001	7790	00:00:01.024	145	10/10
TC7	18	1627	00:00:00.036	22600	00:00:01.026	1627	10/10
TC8	20	65	235200 nanoseconds	3620	00:00:01.017	65	10/10
TC9	22	15709	00:00:03.867	246000	00:00:01.015	15709	10/10
TC10	23	112	756400 nanoseconds	6750	00:00:01.520	112	10/10
TC11	26	28039	00:00:17.212	614000	00:00:01.069	28039	10/10
TC12	28	539	00:00:00.004	8900	00:00:01.069	539	10/10
TC13	34	209959	00:17:19.147	4950000	00:00:01.077	209959	1/10
TC14	36	250325	00:25:15.531	6050000	00:00:12.147	250325	1/1
TC15	38	2836	00:00:00.114	40200	00:00:02.073	2836	1/1
TC16	39	445323	01:23:20.617	11200000	00:00:01.076	445323	1/1
TC17	42	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix D8- A* with Corner Pattern Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	15	35000 nanoseconds	2330	00:00:01.014	15	10/10
TC2	8	148	432100 nanoseconds	2920	00:00:01.008	148	10/10
TC3	10	510	00:00:00.004	8450	00:00:01.960	510	10/10
TC4	10	27	65700 nanoseconds	2430	00:00:00.999	27	10/10
TC5	15	491	00:00:00.004	8940	00:00:00.986	491	10/10
TC6	18	393	00:00:00.002	5630	00:00:01.112	393	10/10
TC7	18	540	00:00:00.004	7630	00:00:01.016	540	10/10
TC8	20	563	00:00:00.005	10600	00:00:01.013	563	10/10
TC9	22	12238	00:00:02.125	174000	00:00:01.167	12238	10/10
TC10	23	98	268000 nanoseconds	2730	00:00:09.193	98	10/10
TC11	26	23101	00:00:11.127	482000	00:00:08.828	23101	10/10
TC12	28	13657	00:00:02.833	207000	00:00:01.372	13657	10/10
TC13	34	5800	00:00:00.474	81800	00:00:08.333	5800	10/10
TC14	36	110712	00:04:48.603	2610000	00:00:08.454	110712	1/1
TC15	38	230053	00:21:51.753	5700000	00:00:10.785	230053	1/1
TC16	39	95723	00:03:30.535	2200000	00:00:09.995	95723	1/1
TC17	42	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix D9- A* with Max(Fringe Pattern, Corner Pattern) Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	15	44800 nanoseconds	2990	00:00:03.163	15	10/10
TC2	8	122	482800 nanoseconds	3960	00:00:03.161	122	10/10
TC3	10	125	493200 nanoseconds	3950	00:00:18.329	125	10/10
TC4	10	27	91500 nanoseconds	3390	00:00:04.972	27	10/10
TC5	15	42	123000 nanoseconds	2930	00:00:02.783	42	10/10
TC6	18	95	313500 nanoseconds	3300	00:00:03.575	95	10/10
TC7	18	257	00:00:00.001	6080	00:00:03.362	257	10/10
TC8	20	65	204500 nanoseconds	3150	00:00:05.912	65	10/10
TC9	22	3791	00:00:00.216	57100	00:00:03.386	3791	4/10
TC10	23	98	00:00:00.047	484000	00:01:21.731	98	1/1
TC11	26	5475	00:00:00.775	142000	00:00:45.158	5475	1/1
TC12	28	258	00:00:00.002	11200	00:00:31.427	258	1/1
TC13	34	2025	00:00:00.060	29900	00:00:31.874	2025	1/1
TC14	36	21711	00:00:09.799	451000	00:00:18.239	21711	1/1
TC15	38	1460	00:00:00.036	24900	00:00:20.250	1460	1/1
TC16	39	45560	00:00:45.520	999000	00:00:18.198	45560	1/1
TC17	42	358186	00:54:34.796	9140000	00:00:18.006	358186	1/1
TC18	46	35261	00:00:28.619	812000	00:00:48.834	35261	1/1
TC19	48	75175	00:02:13.335	1770000	00:00:21.757	75175	1/1
TC20	50	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix D10- A* with 5-5-5 Pattern Databases (Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	29650 nanoseconds	2280	00:00:00.002	13	100/100
TC2	8	30	73530 nanoseconds	2450	00:00:00.002	30	100/100
TC3	10	33	81110 nanoseconds	2460	00:00:00.003	33	100/100
TC4	10	25	67850 nanoseconds	2710	00:00:00.002	25	100/100
TC5	15	35	89780 nanoseconds	2570	00:00:00.002	35	100/100
TC6	18	41	109480 nanoseconds	2670	00:00:00.002	41	100/100
TC7	18	193	839870 nanoseconds	4350	00:00:00.002	193	100/100
TC8	20	43	111990 nanoseconds	2600	00:00:00.002	43	100/100
TC9	22	312	00:00:00.001	6190	00:00:00.002	312	100/100
TC10	23	506	00:00:00.005	11000	00:00:00.002	506	100/100
TC11	26	1132	00:00:00.021	18600	00:00:00.002	1132	100/100
TC12	28	509	00:00:00.004	8430	00:00:00.002	509	100/100
TC13	34	416	00:00:00.002	7180	00:00:00.002	416	100/100
TC14	36	42219	00:00:42.377	1000000	00:00:00.002	42219	10/10
TC15	38	2453	00:00:00.101	41300	00:00:00.002	2453	10/10
TC16	39	13619	00:00:03.235	238000	00:00:00.002	13619	10/10
TC17	42	309769	00:46:02.159	8920000	00:00:00.002	309769	1/1
TC18	46	60501	00:01:38.007	1620000	00:00:00.002	60501	1/1
TC19	48	33458	00:00:27.372	818000	00:00:00.002	33458	1/1
TC20	50	64119	00:01:52.789	1760000	00:00:00.002	64119	1/1
TC21	52	345318	00:58:11.100	1.01E+07	00:00:00.002	345318	1/1
TC22	54	N/A	> 2 hours	N/A	N/A	N/A	N/A

Appendix D11- A* with 6-6-3 Pattern Databases (Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	13	31370 nanoseconds	2410	00:00:00.017	13	100/100
TC2	8	30	79820 nanoseconds	2660	00:00:00.017	30	100/100
TC3	10	33	86710 nanoseconds	2630	00:00:00.019	33	100/100
TC4	10	28	78620 nanoseconds	2810	00:00:00.017	28	100/100
TC5	15	59	168620 nanoseconds	2860	00:00:00.017	59	100/100
TC6	18	39	106120 nanoseconds	2720	00:00:00.017	39	100/100
TC7	18	129	477200 nanoseconds	3700	00:00:00.017	129	100/100
TC8	20	43	120590 nanoseconds	2800	00:00:00.017	43	100/100
TC9	22	241	00:00:00.001	5510	00:00:00.017	241	100/100
TC10	23	728	00:00:00.009	13600	00:00:00.013	728	100/100
TC11	26	1640	00:00:00.047	29100	00:00:00.019	1640	100/100
TC12	28	437	00:00:00.002	6720	00:00:00.013	437	100/100
TC13	34	526	00:00:00.004	7860	00:00:00.013	526	100/100
TC14	36	39617	00:00:35.860	905000	00:00:00.013	39617	10/10
TC15	38	1252	00:00:00.025	20600	00:00:00.013	1252	10/10
TC16	39	31395	00:00:21.933	699000	00:00:00.013	31395	10/10
TC17	42	202234	00:17:52.976	5310000	00:00:00.015	202234	1/1
TC18	46	60501	00:01:18.027	1290000	00:00:00.046	60501	1/1
TC19	48	31429	00:00:22.131	704000	00:00:00.016	31429	1/1
TC20	50	82140	00:02:52.108	2100000	00:00:00.358	82140	1/1
TC21	52	99329	00:04:03.305	2450000	00:00:00.013	99329	1/1
TC22	54	N/A	> 2 hours	N/A	N/A	N/A	N/A

Appendix E1- IDA* with Misplaced Tiles Heuristic Results for –Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	16900 nanoseconds	2820	N/A	13	100/100
TC2	8	30	49780 nanoseconds	1660	N/A	21	100/100
TC3	10	58	190770 nanoseconds	3290	N/A	24	100/100
TC4	10	54	81060 nanoseconds	1500	N/A	25	100/100
TC5	15	54	77490 nanoseconds	1440	N/A	35	100/100
TC6	18	209	280330 nanoseconds	1340	N/A	39	100/100
TC7	18	5939	00:00:00.006	1120	N/A	46	100/100
TC8	20	4013	00:00:00.003	988	N/A	43	100/100
TC9	22	11347	00:00:00.011	995	N/A	50	100/100
TC10	23	18209	00:00:00.017	936	N/A	55	10/10
TC11	26	503688	00:00:00.479	951	N/A	67	10/10
TC12	28	683225	00:00:00.615	900	N/A	66	10/10
TC13	34	21481574	00:00:19.752	919	N/A	81	1/1
TC14	36	191473769	00:02:54.226	909	N/A	91	1/1
TC15	38	1087550355	00:16:18.313	899	N/A	92	1/1
TC16	39	517998067	00:07:40.784	889	N/A	93	1/1
TC17	42	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix E2- IDA* with Checkerboard Misplaced Tiles Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	129180 nanoseconds	21500	N/A	13	100/100
TC2	8	9	105930 nanoseconds	11800	N/A	21	100/100
TC3	10	22	143990 nanoseconds	6550	N/A	24	100/100
TC4	10	14	76450 nanoseconds	5460	N/A	25	100/100
TC5	15	17	44880 nanoseconds	2640	N/A	35	100/100
TC6	18	26	59320 nanoseconds	2280	N/A	39	100/100
TC7	18	1215	00:00:00.001	1430	N/A	46	100/100
TC8	20	207	322570 nanoseconds	1560	N/A	43	100/100
TC9	22	965	00:00:00.001	1420	N/A	50	100/100
TC10	23	3451	00:00:00.004	1380	N/A	55	41557
TC11	26	503688	00:00:00.490	972	N/A	67	41557
TC12	28	683225	00:00:00.638	934	N/A	66	41557
TC13	34	21481574	00:00:20.052	933	N/A	81	41275
TC14	36	5440989	00:00:06.908	1270	N/A	88	41275
TC15	38	30759399	00:00:41.847	1360	N/A	91	41275
TC16	39	30308435	00:00:39.086	1290	N/A	93	41275
TC17	42	423504656	00:08:48.934	1250	N/A	103	41275
TC18	46	1549061305	00:31:47.893	1230	N/A	109	41275
TC19	48	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix E3- IDA* with Relaxed Adjacency Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	19480 nanoseconds	3250	N/A	13	100/100
TC2	8	16	39690 nanoseconds	2480	N/A	21	100/100
TC3	10	29	89180 nanoseconds	3080	N/A	24	100/100
TC4	10	24	49100 nanoseconds	2050	N/A	25	100/100
TC5	15	48	85060 nanoseconds	1770	N/A	35	100/100
TC6	18	42	81780 nanoseconds	1950	N/A	39	100/100
TC7	18	2507	00:00:00.003	1230	N/A	46	100/100
TC8	20	1599	00:00:00.001	1160	N/A	43	100/100
TC9	22	4301	00:00:00.005	1260	N/A	50	100/100
TC10	23	11568	00:00:00.034	3000	N/A	55	10/10
TC11	26	185421	00:00:00.260	1400	N/A	67	10/10
TC12	28	213984	00:00:00.260	1220	N/A	64	10/10
TC13	34	6746241	00:00:08.703	1290	N/A	78	10/10
TC14	36	57874218	00:01:08.202	1180	N/A	88	1/1
TC15	38	319357153	00:06:18.695	1190	N/A	91	1/1
TC16	39	315014869	00:06:25.266	1220	N/A	93	1/1
TC17	42	3681878420	01:18:12.015	1270	N/A	103	1/1
TC18	46	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix E4- IDA* with Number of Tiles out of Row and Column Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	18630 nanoseconds	3110	N/A	13	100/100
TC2	8	9	24190 nanoseconds	2690	N/A	21	100/100
TC3	10	28	90760 nanoseconds	3240	N/A	24	100/100
TC4	10	23	47770 nanoseconds	2080	N/A	25	100/100
TC5	15	17	37240 nanoseconds	2190	N/A	35	100/100
TC6	18	33	63040 nanoseconds	1910	N/A	39	100/100
TC7	18	967	00:00:00.001	1340	N/A	46	100/100
TC8	20	584	753220 nanoseconds	1290	N/A	43	100/100
TC9	22	2002	00:00:00.002	1260	N/A	50	100/100
TC10	23	2704	00:00:00.004	1810	N/A	55	10/10
TC11	26	37164	00:00:00.055	1500	N/A	67	10/10
TC12	28	21943	00:00:00.027	1240	N/A	64	10/10
TC13	34	558218	00:00:00.691	1240	N/A	79	10/10
TC14	36	5605351	00:00:06.672	1190	N/A	88	10/10
TC15	38	26275732	00:00:31.471	1200	N/A	91	10/10
TC16	39	11718155	00:00:14.921	1270	N/A	91	10/10
TC17	42	305266436	00:06:25.039	1260	N/A	106	2/10
TC18	46	1048305175	00:20:52.598	1190	N/A	109	1/10
TC19	48	3958930123	01:22:27.656	1250	N/A	114	6/10
TC20	50	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix E5- IDA* with Manhattan Distance Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	26580 nanoseconds	4430	N/A	13	100/100
TC2	8	9	38300 nanoseconds	4260	N/A	21	100/100
TC3	10	22	130190 nanoseconds	5920	N/A	24	100/100
TC4	10	14	53170 nanoseconds	3800	N/A	25	100/100
TC5	15	17	55300 nanoseconds	3250	N/A	35	100/100
TC6	18	23	72640 nanoseconds	3160	N/A	39	100/100
TC7	18	217	564780 nanoseconds	2600	N/A	46	100/100
TC8	20	75	221960 nanoseconds	2960	N/A	43	100/100
TC9	22	349	781230 nanoseconds	2240	N/A	50	100/100
TC10	23	1178	00:00:00.003	3330	N/A	55	10/10
TC11	26	4095	00:00:00.009	2240	N/A	67	10/10
TC12	28	1136	00:00:00.002	2140	N/A	64	10/10
TC13	34	7219	00:00:00.016	2280	N/A	74	10/10
TC14	36	265958	00:00:00.571	2150	N/A	88	10/10
TC15	38	182457	00:00:00.384	2110	N/A	85	10/10
TC16	39	611552	00:00:01.312	2150	N/A	90	10/10
TC17	42	6368671	00:00:13.492	2120	N/A	102	10/10
TC18	46	1431913	00:00:02.964	2070	N/A	106	10/10
TC19	48	646996	00:00:01.368	2120	N/A	110	10/10
TC20	50	32465360	00:01:11.694	2210	N/A	116	10/10
TC21	52	57909618	00:01:59.102	2060	N/A	123	1/1
TC22	54	46578487	00:01:37.083	2080	N/A	129	1/1
TC23	56	273520091	00:09:45.031	2140	N/A	138	1/1
TC24	60	135718642	00:04:41.797	2080	N/A	139	1/1
TC25	62	73237830	00:02:42.722	2220	N/A	144	1/1
TC26	62	1724849518	01:00:16.807	2100	N/A	149	1/1
TC27	65	2606488180	01:27:49.021	2020	N/A	158	1/1
TC28	66	646564154	00:20:51.506	1940	N/A	150	1/1
TC29	71	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix E6- IDA* with Manhattan Distance and Linear Conflict Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	28350 nanoseconds	4730.0	N/A	13	100/100
TC2	8	9	43420 nanoseconds	4820.0	N/A	21	100/100
TC3	10	22	121240 nanoseconds	5510.0	N/A	24	100/100
TC4	10	14	57420 nanoseconds	4100.0	N/A	25	100/100
TC5	15	17	63580 nanoseconds	3740.0	N/A	35	100/100
TC6	18	23	83130 nanoseconds	3610.0	N/A	39	100/100
TC7	18	191	537760 nanoseconds	2820.0	N/A	46	100/100
TC8	20	23	81150 nanoseconds	3530.0	N/A	43	100/100
TC9	22	142	443320 nanoseconds	3120.0	N/A	50	100/100
TC10	23	561	00:00:00.008	15000.0	N/A	55	10/10
TC11	26	2261	00:00:00.008	3970.0	N/A	67	10/10
TC12	28	397	00:00:00.001	3730.0	N/A	64	10/10
TC13	34	1596	00:00:00.005	3670.0	N/A	74	10/10
TC14	36	94051	00:00:00.272	2900.0	N/A	88	10/10
TC15	38	13331	00:00:00.032	2450.0	N/A	82	10/10
TC16	39	88665	00:00:00.222	2500.0	N/A	89	10/10
TC17	42	1289224	00:00:03.299	2560.0	N/A	98	10/10
TC18	46	421093	00:00:01.046	2480.0	N/A	106	10/10
TC19	48	42781	00:00:00.100	2350.0	N/A	110	10/10
TC20	50	357351	00:00:00.843	2360.0	N/A	116	10/10
TC21	52	1946439	00:00:04.543	2330.0	N/A	120	1/1
TC22	54	7172835	00:00:17.830	2490.0	N/A	129	1/1
TC23	56	32215324	00:01:17.249	2400.0	N/A	138	1/1
TC24	60	37546847	00:01:33.214	2480.0	N/A	139	1/1
TC25	62	13501326	00:00:35.701	2640.0	N/A	144	1/1
TC26	62	333625056	00:13:50.797	2490.0	N/A	149	1/1
TC27	65	607642289	00:24:36.389	2430.0	N/A	158	1/1
TC28	66	129608337	00:05:15.552	2430.0	N/A	150	1/1
TC29	71	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix E7- IDA* with Fringe Pattern Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	16	38500 nanoseconds	2410	00:00:01.014	13	10/10
TC2	8	101	185300 nanoseconds	1830	00:00:01.018	23	10/10
TC3	10	61	129100 nanoseconds	2120	00:00:01.479	25	10/10
TC4	10	11	37100 nanoseconds	3370	00:00:01.030	25	10/10
TC5	15	19	54100 nanoseconds	2850	00:00:01.025	35	10/10
TC6	18	61	115100 nanoseconds	1890	00:00:01.026	39	10/10
TC7	18	1426	00:00:00.002	1730	00:00:01.016	47	10/10
TC8	20	33	69000 nanoseconds	2090	00:00:01.018	43	10/10
TC9	22	17192	00:00:00.023	1370	00:00:01.011	55	10/10
TC10	23	44	931800 nanoseconds	21200	00:00:01.569	55	10/10
TC11	26	20690	00:00:00.040	1980	00:00:01.096	69	10/10
TC12	28	267	511200 nanoseconds	1910	00:00:01.025	65	10/10
TC13	34	372551	00:00:00.850	2280	00:00:07.928	88	10/10
TC14	36	1063480	00:00:01.728	1630	00:00:10.047	95	10/10
TC15	38	2397	00:00:00.004	1690	00:00:01.337	86	10/10
TC16	39	2040186	00:00:03.219	1580	00:00:01.240	99	10/10
TC17	42	5855502	00:00:09.809	1680	00:00:12.756	110	1/1
TC18	46	196573	00:00:00.446	2270	00:00:07.785	117	1/1
TC19	48	955739	00:00:01.760	1840	00:00:01.080	120	1/1
TC20	50	24245037	00:00:39.039	1610	00:00:01.163	127	1/1
TC21	52	3488993	00:00:05.815	1670	00:00:01.237	131	1/1
TC22	54	209095212	00:05:21.041	1540	00:00:00.961	141	1/1
TC23	56	67867176	00:01:47.683	1590	00:00:07.443	141	1/1
TC24	60	2821838614	01:10:12.984	1490	00:00:27.897	151	1/1
TC25	62	2805421053	01:14:47.341	1600	00:00:03.169	160	1/1
TC26	62	989131020	00:25:29.898	1550	00:00:02.090	158	1/1
TC27	65	N/A	> 2 hours	N/A	N/A	N/A	0/1

Appendix E8- IDA* with Corner Pattern Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	22900 nanoseconds	3820	00:00:01.011	13	10/10
TC2	8	77	133600 nanoseconds	1740	00:00:01.016	23	10/10
TC3	10	276	446200 nanoseconds	1620	00:00:02.048	26	10/10
TC4	10	11	39200 nanoseconds	3560	00:00:01.009	25	10/10
TC5	15	304	587500 nanoseconds	1930	00:00:01.013	38	10/10
TC6	18	127	290500 nanoseconds	2290	00:00:01.013	43	10/10
TC7	18	531	00:00:00.001	2000	00:00:01.006	47	10/10
TC8	20	307	488800 nanoseconds	1590	00:00:01.008	48	10/10
TC9	22	14734	00:00:00.022	1540	00:00:01.011	59	10/10
TC10	23	44	112400 nanoseconds	2550	00:00:01.733	55	10/10
TC11	26	22501	00:00:00.034	1520	00:00:01.037	72	10/10
TC12	28	9533	00:00:00.015	1630	00:00:01.030	72	10/10
TC13	34	1826	00:00:00.003	1720	00:00:02.025	82	10/10
TC14	36	130400	00:00:00.405	3110	00:00:01.131	96	10/10
TC15	38	1013372	00:00:02.091	2060	00:00:01.539	98	10/10
TC16	39	247400	00:00:00.408	1650	00:00:01.338	100	10/10
TC17	42	15749973	00:00:25.209	1600	00:00:12.257	114	1/1
TC18	46	2441843	00:00:04.039	1650	00:00:01.015	121	1/1
TC19	48	29616276	00:00:45.830	1550	00:00:01.006	125	1/1
TC20	50	50477811	00:01:23.003	1640	00:00:08.174	133	1/1
TC21	52	102988050	00:02:36.564	1520	00:00:01.269	138	1/1
TC22	54	2239822970	01:02:53.873	1680	00:00:01.366	139	1/1
TC23	56	905332296	00:23:59.913	1590	00:00:01.500	145	1/1
TC24	60	N/A	>2 hours	N/A	N/A	N/A	N/A

Appendix E9- IDA* with Max(Fringe Pattern, Corner Pattern) Database (Non-Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	36000 nanoseconds	6000	00:00:03.303	13	10/10
TC2	8	64	209900 nanoseconds	3280	00:00:03.176	23	10/10
TC3	10	46	213000 nanoseconds	4630	00:00:18.178	24	10/10
TC4	10	11	50600 nanoseconds	4600	00:00:02.792	25	10/10
TC5	15	19	76900 nanoseconds	4050	00:00:07.630	35	10/10
TC6	18	34	138000 nanoseconds	4060	00:00:10.137	39	10/10
TC7	18	133	505000 nanoseconds	3800	00:00:08.152	46	10/10
TC8	20	33	114900 nanoseconds	3480	00:00:03.214	43	10/10
TC9	22	4152	00:00:00.009	2180	00:00:03.359	55	10/10
TC10	23	44	00:00:00.003	81500	00:00:18.796	55	10/10
TC11	26	2725	00:00:00.012	4700	00:00:18.592	68	10/10
TC12	28	142	467000 nanoseconds	3290	00:00:19.221	65	10/10
TC13	34	500	00:00:00.001	2990	00:00:22.750	81	10/10
TC14	36	10682	00:00:00.033	3140	00:00:05.422	91	10/10
TC15	38	844	00:00:00.002	3070	00:00:03.427	86	10/10
TC16	39	45489	00:00:00.117	2580	00:00:03.365	97	10/10
TC17	42	584759	00:00:01.970	3370	00:00:03.891	108	1/1
TC18	46	20783	00:00:00.105	5080	00:00:24.812	115	1/1
TC19	48	39378	00:00:00.099	2520	00:00:22.405	113	1/1
TC20	50	597859	00:00:01.677	2810	00:00:18.338	120	1/1
TC21	52	99434	00:00:00.305	3070	00:00:18.465	117	1/1
TC22	54	4850512	00:00:13.526	2790	00:00:22.690	135	1/1
TC23	56	915980	00:00:02.667	2910	00:00:21.106	138	1/1
TC24	60	67649415	00:02:55.438	2590	00:00:16.502	148	1/1
TC25	62	85869116	00:03:51.453	2700	00:00:29.500	152	1/1
TC26	62	71343816	00:03:10.071	2660	00:00:21.883	156	1/1
TC27	65	N/A	> 2 hours	N/A	N/A	N/A	N/A

Appendix E10- IDA* with 5-5-5 Pattern Databases (Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	26209 nanoseconds	4370	00:00:00.001	13	1000/1000
TC2	8	9	35549 nanoseconds	3950	00:00:00.001	21	1000/1000
TC3	10	17	57624 nanoseconds	3390	00:00:00.002	24	1000/1000
TC4	10	11	39579 nanoseconds	3600	00:00:00.001	25	1000/1000
TC5	15	16	51596 nanoseconds	3220	00:00:00.001	35	1000/1000
TC6	18	19	56886 nanoseconds	2990	00:00:00.001	39	1000/1000
TC7	18	79	213896 nanoseconds	2710	00:00:00.001	46	1000/1000
TC8	20	21	60629 nanoseconds	2890	00:00:00.001	43	1000/1000
TC9	22	89	226494 nanoseconds	2540	00:00:00.001	50	1000/1000
TC10	23	209	598320 nanoseconds	2860	00:00:00.002	55	100/100
TC11	26	533	00:00:00.001	2650	00:00:00.002	67	100/100
TC12	28	125	377710 nanoseconds	3020	00:00:00.002	64	100/100
TC13	34	123	388920 nanoseconds	3160	00:00:00.002	74	100/100
TC14	36	31243	00:00:00.078	2520	00:00:00.002	88	100/100
TC15	38	2203	00:00:00.005	2620	00:00:00.002	82	100/100
TC16	39	3849	00:00:00.009	2510	00:00:00.002	89	100/100
TC17	42	352767	00:00:00.908	2580	00:00:00.002	98	100/100
TC18	46	20256	00:00:00.050	2500	00:00:00.002	106	100/100
TC19	48	11194	00:00:00.027	2440	00:00:00.002	110	100/100
TC20	50	51133	00:00:00.117	2300	00:00:00.002	116	100/100
TC21	52	100100	00:00:00.243	2430	00:00:00.002	117	10/10
TC22	54	854543	00:00:02.134	2500	00:00:00.002	129	10/10
TC23	56	4115400	00:00:09.990	2430	00:00:00.002	138	10/10
TC24	60	773717	00:00:01.844	2380	00:00:00.002	139	10/10
TC25	62	556052	00:00:01.272	2290	00:00:00.002	138	1/1
TC26	62	4268227	00:00:10.434	2440	00:00:00.002	141	1/1
TC27	65	26620512	00:01:03.820	2400	00:00:00.002	150	1/1
TC28	66	6581810	00:00:15.564	2360	00:00:00.002	144	1/1
TC29	71	44348477	00:01:39.145	2240	00:00:00.006	154	1/1
TC30	74	110441995	00:03:58.694	2160	00:00:00.002	162	1/1
TC31	80	2661801414	01:34:19.948	2130	00:00:00.011	170	1/1
TC32	80	2195545439	01:18:07.721	2140	00:00:00.109	168	1/1

Appendix E11- IDA* with 6-6-3 Pattern Databases (Additive) Heuristic Results for 15-Puzzle

Test Case	Depth of Optimal Solution	Number of Iterations	Search Time (hh:mm:ss.000)	Time per Iteration (nanoseconds)	DB Load Time (hh:mm:ss.000)	Maximum Nodes Stored at a Time	Number of Runs Completed
TC1	5	6	24872 nanoseconds	4150.0	00:00:00.013	13	1000/1000
TC2	8	9	34258 nanoseconds	3810.0	00:00:00.013	21	1000/1000
TC3	10	17	53787 nanoseconds	3160.0	00:00:00.013	24	1000/1000
TC4	10	12	41596 nanoseconds	3470.0	00:00:00.013	25	1000/1000
TC5	15	16	50072 nanoseconds	3130.0	00:00:00.013	35	1000/1000
TC6	18	19	58505 nanoseconds	3080.0	00:00:00.013	39	1000/1000
TC7	18	36	108941 nanoseconds	3030.0	00:00:00.013	46	1000/1000
TC8	20	21	59648 nanoseconds	2840.0	00:00:00.013	43	1000/1000
TC9	22	60	155298 nanoseconds	2590.0	00:00:00.013	50	1000/1000
TC10	23	366	00:00:00.001	4870.0	00:00:00.018	55	100/100
TC11	26	726	00:00:00.002	2780.0	00:00:00.017	67	100/100
TC12	28	111	373190 nanoseconds	3360.0	00:00:00.017	64	100/100
TC13	34	217	696480 nanoseconds	3210.0	00:00:00.017	74	100/100
TC14	36	26974	00:00:00.077	2890.0	00:00:00.017	88	100/100
TC15	38	654	00:00:00.002	3240.0	00:00:00.017	82	100/100
TC16	39	13187	00:00:00.036	2810.0	00:00:00.017	89	100/100
TC17	42	137107	00:00:00.458	3340.0	00:00:00.019	98	100/100
TC18	46	16848	00:00:00.044	2670.0	00:00:00.017	106	100/100
TC19	48	14448	00:00:00.039	2770.0	00:00:00.017	110	100/100
TC20	50	65334	00:00:00.167	2570.0	00:00:00.017	116	100/100
TC21	52	23261	00:00:00.057	2480.0	00:00:00.017	117	10/10
TC22	54	594923	00:00:01.699	2860.0	00:00:00.017	129	10/10
TC23	56	4633074	00:00:11.285	2440.0	00:00:00.017	138	10/10
TC24	60	695314	00:00:01.664	2390.0	00:00:00.017	139	10/10
TC25	62	177825	00:00:00.412	2320.0	00:00:00.020	138	1/1
TC26	62	1986566	00:00:04.385	2210.0	00:00:00.020	143	1/1
TC27	65	6530529	00:00:14.513	2220.0	00:00:00.020	150	1/1
TC28	66	2363773	00:00:05.428	2300.0	00:00:00.021	144	1/1
TC29	71	37513959	00:01:31.029	2430.0	00:00:00.020	154	1/1
TC30	74	52564982	00:02:02.743	2340.0	00:00:00.017	162	1/1
TC31	80	1329932862	00:52:40.218	2380.0	00:00:00.021	170	1/1
TC32	80	1084339385	00:41:15.149	2280.0	00:00:00.096	167	1/1

12 References

- Archer, A. F. 1999. A Modern Treatment of the 15 Puzzle. *The American Mathematical Monthly* [Online] 106(9). Available at: http://www.jstor.org/stable/2589612 [Accessed: 5 July 2013].
- Baba, N. and Jain, L. C. 2001. *Computational Intelligence in Games*. Germany: Springer.
- Bonet, B. 2008. Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order. *Workshop on Search in Artificial Intelligence and Robotics at AAAI 2008* [Online]. Chicago, 13-14 July, 2008). Available at: http://ldc.usb.ve/~bonet/reports/AAAI08-ws10-ranking.pdf [Accessed 30 August 2013].
- Coppin, B. 2004. *Artificial Intelligence Illuminated*. USA: Jones and Bartlett Publishers
- Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. 2009. *Introduction to Algorithms*. 3rd ed. USA: The MIT Press
- Culberson, J. C. and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence- An International Journal* 14(3), pp. 318-334.
- Dasgupta, C. H. et.al. 2006. *Algorithms*. New York: McGraw-Hill.
- Edelkamp, S. and Schrödl. 2012. *Heuristic Search: Theory and Applications.* USA: Elsevier.
- Felner, A. et al. 2004. Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research* [Online] 22, pp. 279-318. Available at: http://www.jair.org/media/1480/live-1480-2332-jair.pdf [Accessed 28th August 2013].
- Felner, A. 2011. Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm. *Proceedings of the Fourth International Symposium on Combinatorial Search*. Barcelona, 15-16 July, 2011. (n.l): AAAI Press, pp. 47-51.
- Gaschnig, J. 1979. A Problem Similarity Approach To Devising Heuristics: First Results. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*. Tokyo, 20-23 August, 1979. USA:

Morgan Kaufmann Publishers, Inc., pp. 301-306.

- Hansson, O. et al. 1985. *Criticizing solutions to relaxed models yields powerful admissible heuristics* [Online]. New York: Department of Computer Science, Columbia University. Available at: http://academiccommons.columbia.edu/catalog/ac:141289. [Accessed: 24th July 2013].
- Johnson, W. W. and Story, W. E. 1879. Notes on the "15" Puzzle. *American Journal of Mathematics* [Online] 2(4). Available at: http://www.jstor.org/stable/2369492 [Accessed: 5 July 2013].
- Knuth, D. E. 1998. *The Art of Computer Programming*, vol. 3. Canada: Addison- Wesley.
- Korf, R. E. 1985. *Depth-First Iterative-Deepening: An Optimal Admissible Tree Search*. Holland: Elsevier Science Publishers B. V.
- Korf, R. E. 2000. Recent Progress in the Design and Analysis of Admissible Heuristic Functions. *Proceedings of The Seventeenth National Conference on Artificial Intelligence*. Austin, Texas, 13 August. (n.l): American Association for Artificial Intelligence.
- Korf, R. E and Schultze, P. 2005. Large-Scale Parallel Breadth-First Search. *Proceedings of the 20th National Conference on Artificial Intelligence*. Pennsylvania, 9-13 July, 2005. (n.l): American Association for Artificial Intelligence.
- Luger, G. F. and Stubblefield, W. A. 1997. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* 3rd ed. USA: Addison Wesley Longman, inc.
- Meserve, B. E. 1953. *Fundamental Concepts of Algebra*. Cambridge: Addison-Wesley.
- Michalewicz, Z. and Fogel, D. B. 2004. *How to Solve It: Modern Heuristics*. 2nd ed. Berlin: Springer.
- Mostow, J. and Prieditis, A. E. 2011. Discovering Admissible Heuristics by Abstracting and Optimizing: A Transformational Approach. *Proceedings of the 11th international joint conference on Artificial intelligence*. Barcelona, 16–22 July, 2011. California: AAAI Press, pp. 701-707.

- Nilsson, N. J. 1998. *Artificial Intelligence: A New Synthesis*. San Francisco: Morgan Kaufmann Publishers, Inc.
- Oracle. 1999. *Code Conventions for the Java TM Programming Language* [Online]. (n.l): Sun Microsystems, Inc. Available at: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html. [Accessed: 2nd August 2013].
- Patel, A. 2013. *Heuristics* [Online]. Available at: *http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html* [Accessed: 25th August 2013].
- Pearl, J. 1984. *Heuristics: intelligent search strategies for computer problem solving.* Reading, Massachusetts: Addison-Wesley Pub. Co..
- Pemmaraju, S. and Skiena, S. 2003. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. New York: Cambridge University Press.
- Phillips, H. 2013. Software Product Quality. *CMT303 Software Engineering* [Online]. Available from: Blackboard. [Accessed: 5th July 2013].
- Ratner, D. and Warmuth, M. 1986. Finding a Shortest Solution for the N × N Extension of the 15-puzzle is Intractable. *Proceedings of the fifth national conference on artificial intelligence*. Philadelphia, 11-15 August, 1986. (n.l): AAAI Press, pp. 168-172.
- Russell, S. J. and Norvig, P. 1995. *Artificial Intelligence A Modern Approach*. New Jersey: Prentice Hall.
- Russell, S. J. and Norvig, P. 2003. *Artificial Intelligence A Modern Approach*. 2nd ed. New Jersey: Prentice Hall.
- Ryan, M. 2004. *Solvability of the Tiles Game* [Online]. Available at: http://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html. [Accessed: 5th July 2013].
- Skiena. S. S. 2008. *The Algorithm Design Manual*. 2nd ed. London: Springer.
- Tanton, J. 2005. *Encyclopedia Of Mathematics.* USA: Facts On File, Inc.

- Trapa, P. 2004. Permutations and the 15-Puzzle [Online]. Available at: http://www.math.utah.edu/mathcircle/notes/permutations.pdf. [Accessed: 6th July 2013].
- Walsh, D. P. 2006. *Inversions of a Random Permutation and Kendall's Tau* [Online]. Available at: http://capone.mtsu.edu/dwalsh/4370/437KTAU1.pdf. [Accessed 5th July 2013].
- Wilt, C., Thayer, J., and Ruml, W. 2010. A Comparison of Greedy Search Algorithms. *Third Annual Symposium on Combinatorial Search*. Georgia, 8-10 July, 2010. (n.l): AAAI Press, pp. 47-51.
- Zhou, R. and Hansen, E. A. 2004. Breadth-First Heuristic Search. *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*. British Columbia, 3-7 June, 2004. (n.l): AAAI Press, pp. 325-333.