Luke Garrigan

Registration number 100086495

2017

AI search algorithms for the solution of puzzles

Supervised by Dr Pierre Chardaire

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Acknowledgements

FIrstly, I would like to express my sincere gratitude to my advisor Dr. Pierre Chardaire for the patience, guidance, wisdom and brilliance he has provided throughout my final year. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

Contents

1 Introduction

In computer science a search algorithm is a series of steps that can be used to find a desired state or a path to a particular state. In most scenarios there will be additional constraints that will need to be fulfilled such as the time taken to reach the desired state, memory availability, maximum number of moves. The main difference between informed search and uninformed search is the use of heuristic functions that guide the search towards the goal while avoiding non-promising paths Felner (2015). Informed search algorithms such as A* Hart et al. (1968), IDA* Korf (1985) are analysed in this paper with various heuristics including Manhattan distance, linear conflict Hansson et al. (1985) and pattern databases (PDB) Culberson and Schaeffer (1996). A comparison between informed and uninformed search is made along with an in-depth analysis of different heuristics for specific problem domains such as the sliding-tile puzzle, Towers of Hanoi and the Rubik's cube.

A classic example in the AI literature of pathfinding problems are the sliding-tiles puzzles such as the 3×3 8-puzzle, the 4×4 15-puzzle and the 5×5 24-puzzle. The 8-puzzle consists of a 3×3 grid with eight numbered square tiles and one blank. The blank is used to slide other tiles in which are horizontally or vertically adjacent into that position in an attempt to reach the goal state. The objective is to rearrange the tiles from some random configuration to a specified goal configuration. The number of possible solvable states for the 8-puzzle is $9!/2 = 181440$ so can be solved by means of brute-force search. However for the 15-puzzle with $16!/2 \approx 1.05 \times 10^{13}$ and 25-puzzle with $25!/2 \approx 7.76 \times 10^{24}$ a more sophisticated informed search is required.

The Towers of Hanoi puzzle consists of 3 pegs and n discs all of varying sizes. The discs are initially stacked on the leftmost peg in decreasing order of size with the largest on the bottom. The task is to move all discs from the initial peg to the goal peg without violating two constraints: Only the top disc of any peg can be moved and a larger disc can never be placed on top of a smaller disc. For the 3 peg problem a simple recursive algorithm can be used to find the solution in the minimum number of moves, however, for the 4 peg problem (TOH4) the recursive algorithm doesn't find the goal state optimally. The recursive algorithm works by initially moving the $n-1$ smallest discs to the intermediate peg, then moving the $n-1$ largest disc to the goal peg, then move the $n-1$ smallest discs from the intermediate peg to the goal peg. Finding shortest path with this algorithm is not possible with TOH4 because there are two intermediate pegs rather than one and without priori there is no way to determine which peg should be used over the other. This implies the use of systematic search to find the optimal path.

2 Relevant Literature And Context

The following review discusses a complete definition and overview of the search algorithms implemented in the project, discussing their strengths and weaknesses and their applicability to the sliding-tile puzzle.

2.1 Uninformed Search

Uninformed or brute-force search is a general problem-solving technique that consists of systematically enumerating all the possible states for a given solution and checking to see whether that given state satisfies the problem's statement. All that is required to execute a brute-force is some legal operators, an initial state and an acknowledged goal state. Uninformed search generates the search tree without using any domain specific knowledge

2.1.1 Completeness and Optimality

Often in search the input may be an implicit representation of an infinite graph. Given these conditions, a search algorithm is characterised as being complete if it is guaranteed to find a goal state provided one exists. Breadth-first search is complete and when applied to infinite graphs it will eventually find the solution. Depth-first search is not complete and may get lost in parts of the graph that do not contain a goal state.

2.1.2 Breadth-First Search

Breadth-first search expands the nodes in a tree in the order of their given distance from the root, so it expands all the neighbouring nodes before going to the next level of the tree. The algorithm doesn't trawl to the deeper levels of the tree without first expanding the lower levels thus ensures the finding of the shortest path. The amount of time used by breadth-first search is linear to the number of nodes expanded, since each node can be generated in constant time, and is a function of the branching factor b and the solution depth d. Since the number of nodes at level d is b^d, the total number of nodes generated in the worst case is $O(b^d)$. (Korf, 1995). The space requirement of breadth-first search is its largest deficiency. The 8-tile has a search space of $9!/2 = 181,400$ states with a maximum number of 31 moves to solve. In terms of practicality, with larger problem states such as the 15-tile puzzle a breadth-first search will exhaust the available memory rather quickly with its $16!/2 = 10,461,394,944,000$ states and a maximum number of 80 moves to solve.

Cormen (2009)

Operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to these operations is $O(V)$

The operations of enqueuing and dequeuing take O.1/ time, and so the total time devoted to queue operations is O.V /. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is âĂŽ.E/, the total time spent in scanning adjacency lists is O.E/. The overhead for initialization is O.V /, and thus the total running time of the BFS procedure is O.V C E/. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

2.1.3 Depth-First Search

Depth-first search (DFS) addresses the limitations of breadth-first by always generating next a child of the deepest unexpanded node. Breadth-first search manages the list as a first-in first-out queue, whereas depth-first search treats the list as a last-in first-out stack. Depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack. Given a starting state, depth-first search stores all the unvisited children of that state in a stack. It then removes the top state from the stack and adds all of the removed states children to the stack. Depth first search generates a long sequence of moves, only ever reconsidering a move when it reaches the end of a stack, this can become a serious problem given a graph of significant size and there's only one solution, as it may end up exploring the entire graph on a single DFS path only to find the solution after looking at each node. Worse, if the graph is infinite the search might not terminate.

2.1.4 Depth-Limited Search

Depth-Limited search is DFS with a cut-off depth, this prevents the search from running forever provided the limit isn't set to infinite. The algorithm is not complete as the solution depth could be greater than the cut-off depth and it is not optimal. The algorithm is second procedure shown in Algorithm 5.

2.1.5 Depth-First Search Iterative Deepening

Depth-First Iterative-Deepening (DFID) is an extension of depth-first search, it combines breadth-first search's completeness and depth-first search's space efficiency. DFID has a maximum depth by using DLS; searching all possibilities up to the specific depth and if it doesn't find the goal state it increases the depth increases. DFID performs depth-first search to depth one, then starts over and executes a depth-first search to depth two and continues deeper and deeper until a solution is found. The complexity of DFID is only O(d) where d is the depth, this is because at a given point it is executing only a depth-first search and saving only a stack of nodes. DFID ensures that the shortest path to the goal state will be found as does breadth-first search (Meissner and Brzykcy, 2011).

2.2 Informed Search

Uninformed search often expands states that pursue a direction alternative to the goal path, which can lead to searches taking an extensive amount of time and/or space. Informed search attempts to minimise this by producing intelligent choices for each selected state. This implies the use of a heuristic function which evaluates the likelihood that a given nodes is on the solution path. A heuristic is a function that ranks possible moves at each branching step to decide which branch to follow. The goal of a heuristic is to produce a fast estimation of the cost from the current state to the desired state, the closer the estimation is to the actual cost the more accurate the heuristic function. In the context of the sliding-tile puzzle, to find best move from a set configuration the heuristic function is executed on each of the child states, the child state with the smallest heuristic value is chosen.

2.2.1 Admissible Heuristics

A heuristic function is said to be admissible if it never overestimates the cost of reaching the goal, i.e the estimated cost to the goal from the current node is not higher than the lowest possible cost from the current node. The lowest possible cost $h^*(n)$ is the cost of the optimal path from n to a goal node G. The heuristic function $h(n)$ is admissible if $0 \leq h(n) \leq h^*(n)$; admissible heuristic functions are optimistic meaning they are lower bounds on the actual cost.

2.2.2 Consistent Heuristics

A heuristic function is said to be consistent if the cost from the current node n to a successor node p, plus the estimated cost from the successor node to the goal state is less than or equal to the estimated cost from the current node to the goal node $h(n) \leq c(n,p) + h(p)$, where $c(n,p)$ is the cost of reaching node p from n and the heuristic value of the goal G is zero $h(G) = 0$ (Figure 2).

2.2.3 Manhattan Distance

Manhattan distance is the classic heuristic function for the sliding-tile puzzles, for each tile in the puzzle the Manhattan distance counts the number of grid units between its current location and its goal location and summing these values for all tiles. Manhattan distance is a lower bound on actual solution length, because every tile must move at least its Manhattan distance, and each move only moves one tile. The Manhattan distance is a lower bound for the number of moves required to solve an occurrence of the sliding-tile puzzle, since every tile must move at least as many times as its distance to its goal position. (Linnert et al., 2014).

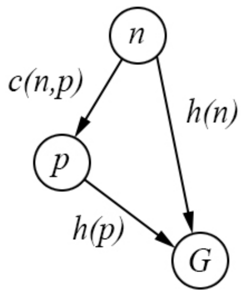


Figure 1: Consistent Heuristics Diagram

$$h(n) = \sum_{all\ tiles} distance(tile, goal\ position)$$

2.2.4 Linear-Conflict Heuristic

Linear-conflict heuristic is an improvement to the Manhattan distance, it applies when two tiles are positioned in their desired row or column but are reversed relative to their goal positions, meaning a given tile must move out of the row or column in order to let the other pass. If there is a linear conflict then an extra two moves will be added to the total sum as the tile which must move, has to transition out of the row or column and then back into its desired position. These two extra moves are not included in the Manhattan distance so can be added to the total without violating admissibility see Algorithm 1

2.2.5 A* Algorithm

A* search is a combination of lowest-cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand. For each path on the frontier, A* uses a heuristic function. This allows A* search to ensure the prioritisation of states in which are more likely to result in a low-cost goal state. The calculation of the heuristic value is dependent on the problem that is being solved. For example, for problems which aim to reach a location the Euclidean distance between

Algorithm 1 Linear Conflict

Hansson et al. (1992).

```
1:  C(tj, ri) is the number of tiles in row ri with which tj is in conflict
2:  lc(s, rj) is the number of tiles that must be removed from row rj to resolve linear
    conflicts
3:  procedure LINEAR CONFLICT($state$)                ▷ The current puzzle configuration
4:      for each row ri in the state s do
5:          $lc(s, ri) \leftarrow 0$
6:          for each tile in row do
7:              determine $C(tj, ri)$
8:              while there is a non-zero C(tj, ri) value do
9:                  find tk such that there is no
10:                     C(tj, ri) > C(tk, ri)
11:                 C(tk, ri) = 0
12:                 for every tile tj which had been in conflict with tk do
13:                     C(tj, ri) = C(tj, ri) - 1
14:                 lc(s, ri) = lc(s, ri) +1
15:     return $total$                                 ▷ The heuristic is the total
```

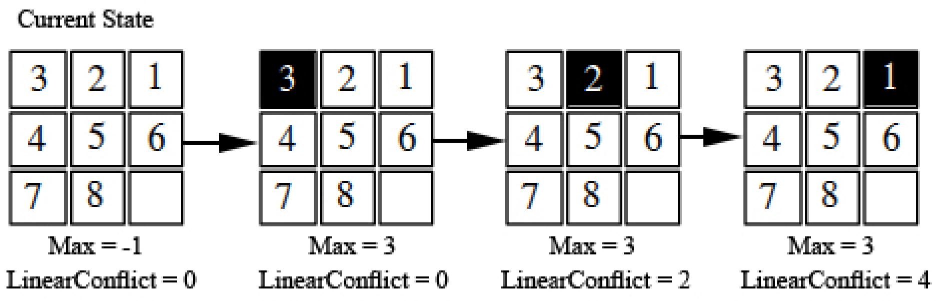


Figure 2: Calculating Linear Conflict

the state and the goal is often used as the heuristic.

The A* algorithm uses $cost(p)$, the cost of the path found, as well as the heuristic function $h(p)$, the estimated path cost from the end of p to the goal. As A* uses admissible heuristic it will always find solutions in the order of their cost. (Brewka, 1996). A* uses breadth-first traversal thus uses large chunks of memory in comparison to applications that use depth-first traversal. The performance of A* is also heavily reliant on the accuracy of the heuristic.

2.2.6 Iterative Deepening A*

Iterative-deepening A* (IDA*) eradicates the memory complication of A* by using depth-first traversal, without sacrificing solution optimality at the cost of increasing the time taken. Each iteration of A* is a complete depth-first search that keeps track of the cost, $f(n) = g(n) = h(n)$, of each node generated. If an expanded node's cost generated exceeds a threshold for that iteration, its path is cut off and the search backtracks before continuing. IDA* ranks states the same way as A*: The expected cost of a state is the cost of reaching said state plus the heuristic's estimate of the cost of reaching the nearest goal state. The maximum expected cost is assigned to the heuristic cost of the starting state, states which have a lower cost than the starting state are traversed and states with a higher expected cost are discarded.

If IDA* has searched all states lower than the current maximum without finding the goal state then it increases the maximum to the lowest value of the discarded states and begins the search again. The goal state is admissible, the current maximum cost will never be higher than the lowest cost solution, thus IDA* with an admissible heuristic will always find the lowest cost solution. IDA* searches many nodes multiple times thus is slower than A*. Hence, A* should be used on smaller applications with lower memory requirements.

The time complexity of IDA* is measured by the number of nodes expanded, provided each node can be expanded and its successors evaluated in constant time, the asymptotic time complexity of IDA* is the total nodes expanded. Otherwise it is the product of the total number of nodes and the time taken to expand each. Provided the heuristic is consistent (see 4.2), IDA* must expand all nodes with the f value less than or equal to c which is the cost of an optimal solution $f(n) \leq c$. On the final iteration of IDA* the cost threshold is equal to c, the cost of an optimal solution.(Korf et al., 2001).

2.3 Pattern Databases

A Pattern Database (PDB) stores a collection of solutions to sub-problems that must be achieved to solve the overall problem. They then used as admissible heuristics implemented as lookup tables that store the lengths of optimal solutions for sub-problem instances. (Felner et al., 2007).

2.3.1 Non-Additive Pattern Databases

For any configuration of the sliding tile puzzle, the minimum number of moves required to solve a subset of the tiles is evidently a lower bound on the number of moves needed to solve the puzzle Felner et al. (2004).

The initial PDB applied to the fifteen tile as shown in Figure 3 is the fringe pattern. The heuristic is estimated dependent on the current positions of the fringe tiles and the blank, however is independent of the positions of the other tiles. These values are precomputed and stored in a lookup table and are called upon when a heuristic estimate is needed. The total number of possible permutations of fringe tiles and blank is $16!/(16-8)! = 518,918,400$.

To retrieve all possible permutations of the Fringe tiles a single breadth-first search must be executed, commencing at the goal state. Tiles which are not fringe are all equivalent, in the database the positioning of the fringe tiles and the blank are stored along with the number of moves taken to get to that state. Using IDA* an optimal solution can be found for the 15-puzzle as the heuristic estimate is optimistic, meaning admissible. To retrieve the value from the lookup table an index must be made from the positioning of the Fringe tiles and the blank, this value is then used as the heuristic estimate for the given state.

Culberson and Schaeffer (1998a) show that it is possible to use the Fringe PDB and the Corner PDB to determine the heuristic estimate by taking the maximum of the two values, Figure 3 displays both the Fringe and the Corner pattern. They explain how there are other viable patterns but intuitively the Fringe and Corner pattern are likely the best at providing the lower bounds for the search. Their reasoning for this assumption was by analysing the work left once the subset had been solved, after solving the fringe tiles only an 8-puzzle remains. Their experiments showed that the best results were from pattern databases that reduced the problem to a simpler one whose solution requires little or no interference with the placing of the pattern, both Fringe and Corner satisfy this.

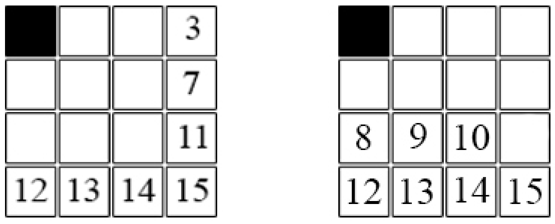


Figure 3: Fringe and Corner patterns for 15

The major issue is space required to store all possible states. When considering the 15 tile: provided each row in the database is 1 byte and there are 519,914,400 rows that's a total of 519Mb, and the time taken to perform the breadth-first search is extensive. Scalability is another issue regarding non-additive pattern databases, if scaled to the twenty-five tile puzzle and performed the same calculations for taking the fringe: $25!/(25-10)! = 1.1861676e+13$, provided each row in the database is 1 byte as previously stated, the total is roughly 12 Terabytes worth of storage.

2.3.2 Statically-Partitioned Additive Database Heuristics

Rather than taking the maximum of the two pattern databases a better option would be to sum the values without violating admissibility and tightening the lower bound. There are two methods to do this: statically-partitioned additive pattern databases and dynamically-partitioned additive pattern databases. In order to create a statically-partitioned additive database for the sliding tile puzzle, the tiles were partitioned into disjoint groups, meaning each tile is placed in a specified group and only appears in that group. The same method as explained in 2.3.1 for finding all the states was computed for each subset; a breadth-first search from the goal state with all possible configurations of each tile in the disjoint groups and the number of moves required to get to the goal state. For a particular state in the search, for each position of the tiles an index was computed for the corresponding row in the database, then retrieved the number of moves required to solve the tiles in that group and added this value to each value of the other disjoint groups. The summed total of these values will always be equal to or larger than the Manhattan distance of the state as it considers the interactions between tiles within a given subset.

Non-additive pattern databases include all the moves required to solve a pattern, including non-labelled tiles. Therefore, two PDB values cannot be added together as an admissible heuristic as moves counted in one PDB could move tiles in the other, meaning the same move would be counted twice. Whereas only the moves in a particular group are counted in additive databases, so the sum can be taken off the pattern values.

2.4 Summary

2.4.1 Evaluation Of Uninformed Searches

Althought DFS memory requirement is linear and it has less time and space complexity to BFS, often DFS induces the possibility of traversing down the left-most path forever, it is also not guaranteed to find the solution.

Depth-first iterative deepening is beneficial as it avoids infinite cycling, it obtains the same result as BFS whilst saving memory, however it is very slow as it has to repeatedly expand nodes it has already visited. DFID is only really beneficial when the solution

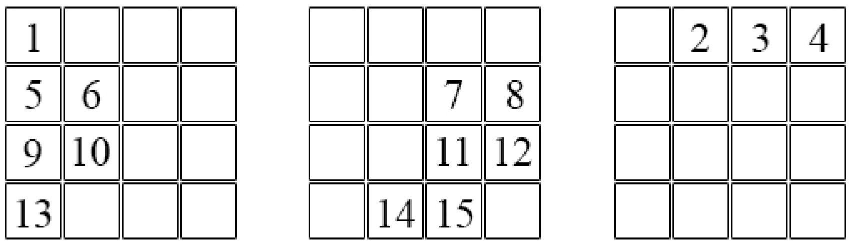


Figure 4: 6-6-3 Partioning

depth is known, else it would have to trawl through all possibilities up to the solution depth and then through all possibilities at the solution depth until it reaches the goal. It is unlikely for the sliding-tile puzzle that the user knows the solution depth, thus implies ample execution time to find the goal state. Theoretically DFID could be used on much more difficult problems which have a larger search space, such as the 15 or 25 tile, but the time taken to find the solution outweighs any benefits.

Breadth-first search is very fast when considering the 8-tile puzzle as the search space is small, however it uses too much memory to be useful with larger problems such as the 15-tile puzzle. If there are multiple solutions, it will find the solution with the least steps.

2.4.2 Evaluation of Informed Searches

Provided the heuristic function is admissible both informed search algorithms will find the optimal path to the goal state. The main drawback of the A* algorithm is its memory requirements; it must store all the visited states in the open set. Although the algorithm runs successfully on small problems such as the eight puzzle, it exhausts available memory rather quickly for the fifteen puzzle, Section 5 displays the testing results of this memory usage for random states.

IDA* on the other hand eliminates the memory constraints of A*, by performing a series of depth-first searches and not needing to store the open set. IDA* was chosen for solving larger search spaces, tested using a number of different heuristics.

3 Design

This section describes the design of the program and specifically the various search algorithms and the appropriate steps required based on the literature review. Pseudo code for each algorithm has also been constructed to form a structure for the implementation.

3.1 Class Diagram

Figure 8 shows a shortened version of the implementation, other algorithms also included in the program are DFS and DFID. Other classes used to construct testing states were not included in the diagram for simplicity, these classes include *RandomStates* which given a particular puzzle returns random solvable states and *Testing* which is used primarily for testing algorithms and determining the number of nodes they expand, the time taken to solve the problem, number of moves and many other statistical values, used extensively in Section 5

3.2 Representing the state

The first step to solving the sliding tile puzzle is determining the best way to store a particular state. There were three elements that needed to be considered: the order of the tiles, the dimensions of the puzzle and the position of the blank. Puzzle configurations were stored using an array, the blank tile was represented by a 0 and the dimensions of the puzzle can be found from the length of the array (square root of the length, provided the state has an equal height and width). The positioning of the blank is important as it is needed for finding legal moves; every tile must push into the blank and computing the position of the blank each iteration would be costly.

Additional fields including the parent state were stored for the ability to traverse the path the algorithm took back to the initial state, the direction that each state took is also stored. Each state contains a heuristic array which is the same length as the puzzle state, each position in the heuristic array represents the number of estimated moves for that given tile to reach its goal state. It would be costly to continually compute the heuristic estimate of every tile in the current configuration: the heuristic array is filled by an initial heuristic calculation and then each state after only computes the heuristic for the previously moved tiles. Figure 8 shows the class diagram of project.

The State class also contains the number of moves from the starting state g, the heuristic value h and f or $(h+g)$.

3.3 Finding legal moves

In order to find legal moves for the sliding puzzle, two variables must be known: the position of the blank and the dimensions of the puzzle. As shown above by representing the state, the zero value for a given state is stored and the dimensions can be easily computed. For a given state in the sliding-tile puzzle, there are 2-4 legal moves: when the blank is in a corner there are two legal moves, if the blank is against an edge but not in the corner there are three legal moves, and if the blank is surrounded by tiles in all directions there are four legal moves. The blank can move left, right, up or down, there are a elements to consider here: the blank cannot go up if its in the top row, cannot go left if its in the left-most column, right if its in the right-most column and down if its in the bottom row. In order to construct this algorithm it helped to visualise the puzzles based by their index rather than their labelled value. Figure 5 displays the 8 and 15-puzzle with their index representation. Algorithm 2 shows a procedure finding the

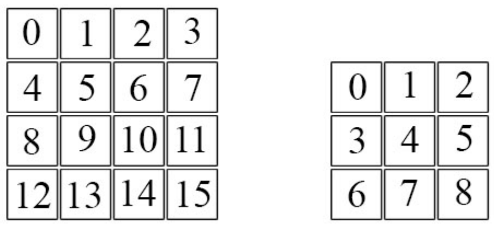


Figure 5: Sliding Tile Index Representation

neighbouring nodes of a particular state. Firstly, an empty list is initialised as we know there are a minimum of two possible moves from every state in the puzzle. Each possible move is tested, checking whether the empty tiles positioning satisfies the constraints, if so then a new state is built with the updated values and added to the list. Once all possible moves have been tried the list of neighbouring nodes is returned to the search algorithm.

Algorithm 2 Finding Neighbouring States

```
 1: procedure FINDNEIGHBOURS(state)                ▷ The current puzzle configuration
 2:     neighbours ← emptylist
 3:     dimensions ← dimensions of state                      ▷ 3 for 3x3, 4 for 4x4
 4:     if zeroPosition % dimensions != 0 then                           ▷ left move
 5:         left ← updated state           ▷ copy of state, changing zero and other tile
 6:         neighbours.add(left)
 7:     if zeroPosition % dimensions != dimensions -1 then              ▷ right move
 8:         right ← updated state
 9:         neighbours.add(right)
10:     if zeroPosition > dimensions -1 then                                ▷ up move
11:         up ← updated state
12:         neighbours.add(up)
13:     if zeroPosition < size(state) - dimensions then                   ▷ Down move
14:         down ← updated state
15:         neighbours.add(down)
16:     return neighbours                            ▷ Returns the list of neighbours
```

3.4 Algorithm Design

3.5 BFS

BFS begins by declaring an empty set *s* which stores all the states that have been expanded. A queue *q* is then initialised which stores states to be expanded. The initial state is added to both *q* and *s*, *q* is then repeatedly looped through until a solution is found or *q* is empty meaning there are no possible solutions. The *currentState* is dequeued from q, checked against the goal state and returned if a match. Otherwise, each possible move from $currentState$ is found and added to q to be expanded see Algorithm 3.

Algorithm 3 Breadth-First Search

```
1: procedure BFS($state$)
2:     $s \leftarrow empty\ set$
3:     $q \leftarrow empty\ queue$
4:     $s.add(state)$
5:     $q.enqueue(state)$
6:     while $q$ is not $empty$ do
7:         $currentState \leftarrow q.dequeue()$
8:         if $currentState = goal$ then
9:             return $current$
10:        for $neighbour$ in $neighbours(currentState)$ do
11:            if $neighbour$ is not in $s$ then
12:                $s.add(neighbour)$
13:                $q.enqueue(neighbour)$
```

3.6 DFS

DFS begins by declaring an empty set $visted$ which stores all the states that stores expanded states. The state is checked against the goal and returned if a match. All possible moves from the current state are looped through and recursively call the function to repeat the process see Algorithm 4.

3.7 DFID

DFID begins by looping from 0 to the maximum depth of the problem domain. A function is then called with the current state and the specified depth which returns $found$ if the goal state has been discovered or $null$ otherwise. The function depth-limited depth-first search *DLS* is called, the state is checked against the goal state provided the depth is 0 as the previous depths have already been checked. If the depth is greater than

Algorithm 4 DFS

```
1: $visited \leftarrow emptyset$
2: procedure DFS($state$)
3:     if $state = goal$ then
4:         return state
5:     for $neighbour$ in $neighbours(state)$ do
6:         if $neighbour$ not in $visited$ then
7:             $DFS(neighbour)$
8:     $visited.add(state)$
```

zero each possible move is found and used in a recursive call of *DLS* with a decremented depth see Algorithm 5.

3.8 Manhattan Distance

Algorithm 6 shows the pseudocode of the Manhattan distance for computing the heuristic for each tile in the current state. For each tile it computes the expected row and expected column (goal row/column), then the actual row and column. The absolute difference of the expected row and actual row and the absolute difference of the expected column and actual column are added together which gives the estimate for that single tile, this is then added to the total. Once all calculations have been performed on each tile, the total is then returned and used as the heuristic estimate.

As mentioned previously computing the Manhattan distance for every state is generally cheap, however it can be improved. A heuristic array can be created, where each element in the array will store its corresponding lower-bound estimate on the number of moves required to get to their goal position. Figure 6 presents an example of a heuristic array, the blank tile is not added to the heuristic function so is defined as 0. On the far left is the current state with configuration: $[1,2,15,0,9,4,10,7,14,13,3,12,8,5,6,11]$ which has a corresponding heuristic array: $[0,0,3,0,1,3,2,1,2,2,2,0,5,3,3,2]$ where each element represents the number of moves to the goal position, this example is assuming the goal state is: $[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0]$. It is clear that when a move has been made only two tiles need to change, but only one of these needs

Algorithm 5 Depth-First Iterative Deepening

```
1:  procedure DFID(state)
2:      for $depth \leftarrow 0, \infty$ do
3:          $found \leftarrow DLS(state, depth)$
4:          if $found \neq null$ then
5:              return $found$
6:  procedure DLS(state, depth)
7:      if $depth = 0$ and $state = goal$ then
8:          return $found$
9:      if $depth > 0$ then
10:         for $neighbour$ in $neighbours(state)$ do
11:             $found \leftarrow DLS(neighbour, depth - 1)$
12:             if $found \neq null$ then
13:                 return $found$
14:     return $null$
```

Algorithm 6 Manhattan Distance

```
1:  $dimensions$                ▷ The dimensions of the puzzle e.g 3 for 3x3 and 4 for 4x4
2:  procedure MANDIST($state$)                        ▷ The current puzzle configuration
3:      $total \leftarrow 0$
4:      for $i \leftarrow 1, puzzleLength$ do               ▷ Loops through each tile of the puzzle
5:          $tileValue \leftarrow state[i]$
6:          $expectedRow \leftarrow (tileValue - 1) \div dimensions$
7:          $expectedCol \leftarrow (tileValue - 1) \mod dimensions$
8:          $rowNum \leftarrow i \div dimensions$
9:          $colNum \leftarrow i \mod dimensions$
10:         $total \leftarrow total + \mid expectedRow - rowNum \mid + \mid expectedCol - colNum \mid$
11:     return $total$                                          ▷ The heuristic is the total
```

to compute the Manhattan distance. The tile that has moved into the blank must now update its heuristic estimate in the heuristic array accordingly, by computing a Manhattan distance given its new position. However, the blank tile must update the heuristic array with 0 for its new position. Figure 6 shows the blank moving down and swapping with tile value 7, the far right puzzle labelled *Updated Heuristic Array* shows the two elements that have been updated: $heuristicArray[3] = 0$ is now $updatedHeuristicArray[7]$ and $heuristicArray[7] = 1$ is now $updatedHeuristicArray[3] = 2$. To get the overall heuristic estimate all elements in the heuristic array must be added together. The same rules apply, compute the entire heuristic estimate once and store the value, and then every calculation after the heuristic of the tile being moved in the blank is taken away and the new heuristic is added. Algorithm 7 shows this process in more detail.

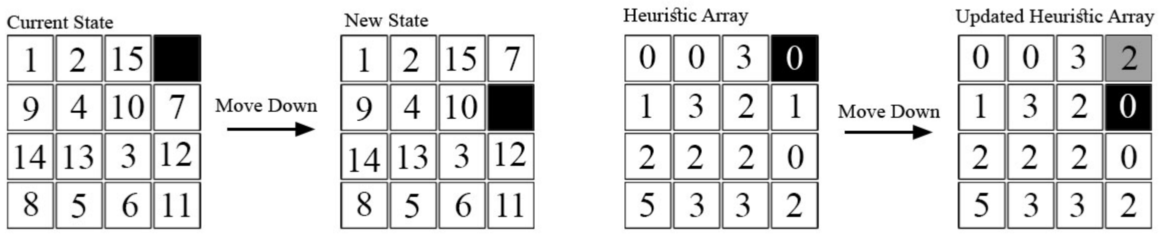


Figure 6: A representation of a heuristic array

3.9 Linear-Conflict

The Linear-Conflict heuristic was implemented by table lookup with one entry for each permutation of a row or column, storing the corresponding linear conflict. Even for the 15-puzzle, there are only 24 permutations of 4 tiles in a row or column. So the cost of the linear-conflict implementation is only a small number of table-lookups. Hansson et al. (1992) implementation showed that on average each calculation of the linear-conflict caused the search program for the fifteen puzzle to be only 5% slower in examining each node, and this was more than compensated for their dramatic decrease in the number of nodes that needed to be examined when using the linear-conflict heuristic.

Algorithm 7 Single Manhattan Distance

```
1: $heuristicArray$  ▷ Array where each element has its respected tiles Heuristic value
2: $heuristicTotal$  ▷ The sum of all elements in the heuristic array
3: $dimensions$  ▷ The dimensions of the puzzle e.g 3 for 3x3 and 4 for 4x4
4: $movedPosition$  ▷ The position of the tile that previously swapped with the blank
5: $zeroPosition$  ▷ The current position of the blank tile
6: procedure SINGLEMANDIST($state$)  ▷ The current puzzle configuration
7:     $tileValue \leftarrow state[i]$
8:     $expectedRow \leftarrow (tileValue - 1) \div dimensions$
9:     $expectedCol \leftarrow (tileValue - 1) \mod dimensions$
10:    $rowNum \leftarrow i \div dimensions$
11:    $colNum \leftarrow i \mod dimensions$
12:    $tempTotal \leftarrow heuristicTotal - heuristics[zeroPosition]$
13:    $newHeuristicValue \leftarrow | expectedRow - rowNum | + | expectedCol - colNum |$
14:    $heuristics[movedPosition] \leftarrow newHeuristicValue$
15:    $heuristics[zeroPosition] \leftarrow 0$
16:    $total \leftarrow tempTotal + newHeuristicValue$
17:    return $total$  ▷ The heuristic estimate of the state is the total
```

3.10 A*

A* begins by declaring two empty sets: *closedSet* stores states that have already been visited; *openSet* contains states to be visited. The initial state is added to the *openSet*, similar to BFS the *openSet* is iterated through until it finds a goal state or there are no more states to explore, implying no solutions. The most promising state is retrieved from the *openSet* (the state with the lowest *F* score) and assigned to *currentState*. This state is then removed from the *openSet* and added to the *closedSet*. The *currentState* is checked to see whether it matches the goal state and returned if so. From the *currentState* all the possible moves are found and denoted as *neighbours*, each *neighbour* is checked against the *closedSet* to make sure the state hasn't already been expanded. The *neighbour* is then checked against the *openSet* and if the *openSet* contains *neighbour* the *G* values are compared to check which version of the state has the best route from the starting position see Algorithm 8.

3.11 IDA*

The IDAStar algorithm is very similar to to DFID; IDAStar iteratively deepens dependent on the *F* value rather than a fixed depth. The initial *bound* is assigned to the *F* score of state, which is also just the *H* value as *G* is inherently 0 as no moves have been made to get to this state. A depth-first limited search is then executed to the specified threshold. If the *F* value of the current state is greater than the *bound* this is returned and assigned to the new *bound*. If the *H* estimate of the state is 0, no more moves are required to get to the goal state thus the solution is found. Similar to A* the neighbour with the smallest *F* value is prioritised see Algorithm 9.

3.12 Design of Java Application

The Java application will run by initially providing the user with two possible inputs: initial/starting state and goal state (optional). The user will then be able to choose which algorithm they would like to use to solve the problem and which heuristic (provided an informed algorithm). Upon completion, the path from the initial state to the goal state will be displayed along with various statistics: time taken, nodes expanded, no. moves

Algorithm 8 A*

```
1:  $g(state)$                                   ▷ The cost to reach the current state
2:  $h(state)$                                   ▷ Estimated cost of the cheapest path from state to goal
3:  $f(state) \leftarrow g(state) + h(state)$
4:  $neighbours(state)$                          ▷ Expands possible moves from current state ordered by g + h
5:  procedure ASTAR($startState$)
6:      $closedSet \leftarrow empty\ set$
7:      $openSet \leftarrow empty\ set$
8:      $openSet.add(startState)$
9:      while $openSet$ is not $empty$ do
10:         $currentState \leftarrow state$ in $openSet\ with\ lowest\ f\ value$
11:         $openSet.remove(currentState)$
12:         $closedSet.add(currentState)$
13:         if $currentState = goal$ then
14:             return $currentState$
15:         for $neighbour$ in $neighbours(currentState)$ do
16:             if $neighbour$ not in $closedSet$ then
17:                 if $neighbour$ in $openSet$ then
18:                     if $g(currentState) < g(neighbour)$ then
19:                         $g(neighbour) \leftarrow g(currentState)$
20:                 else
21:                     $openSet.add(neighbour)$
```

Algorithm 9 Iterative Deepening A Star

```
 1: $state$                                        ▷ The current puzzle configuration
 2: $g(state)$                                     ▷ The cost to reach the current state
 3: $h(state)$                     ▷ Estimated cost of the cheapest path from state to goal
 4: $f(state) \leftarrow g(state) + h(state)$
 5: $neighbours(state)$   ▷ Expands possible moves from current state ordered by g + h

 6: procedure IDASTAR($state$)
 7:     $bound \leftarrow f(state)$
 8:     while not found do                         ▷ Loops until a solution is found
 9:         $bound \leftarrow DLS(state, bound)$         ▷ Performs a bounded depth-first search

10: procedure DLS($state, bound$)
11:     if $f(state) > bound$ then
12:         return $f(state)$
13:     if $h(state) = 0$ then                    ▷ No more moves needed to reach goal state
14:         return $found$
15:     $min \leftarrow \infty$
16:     for $neighbour$ in $neighbours(state)$ do
17:         $temp \leftarrow DLS(neighbour, bound)$
18:         if $temp < min$ then
19:             $min \leftarrow temp$
20:     return $min$                              ▷ Returns the smallest of the neighbours
```

and memory used. Initial preprocessing will be done to ensure that the states provided are actually solvable by checking the parity and the length of each state. The program is generic, so can be used to solve any puzzle provided the initial state, goal state and legal moves have been issued.

3.12.1 Swing

Swing is a set of program components for Java that enable the possibility of creating graphical user interfaces (GUI) components. Swing was used to allow for a more user-friendly program and display a frame for the results to placed. Also a visual representation of how the program solved the puzzle state will be displayed.

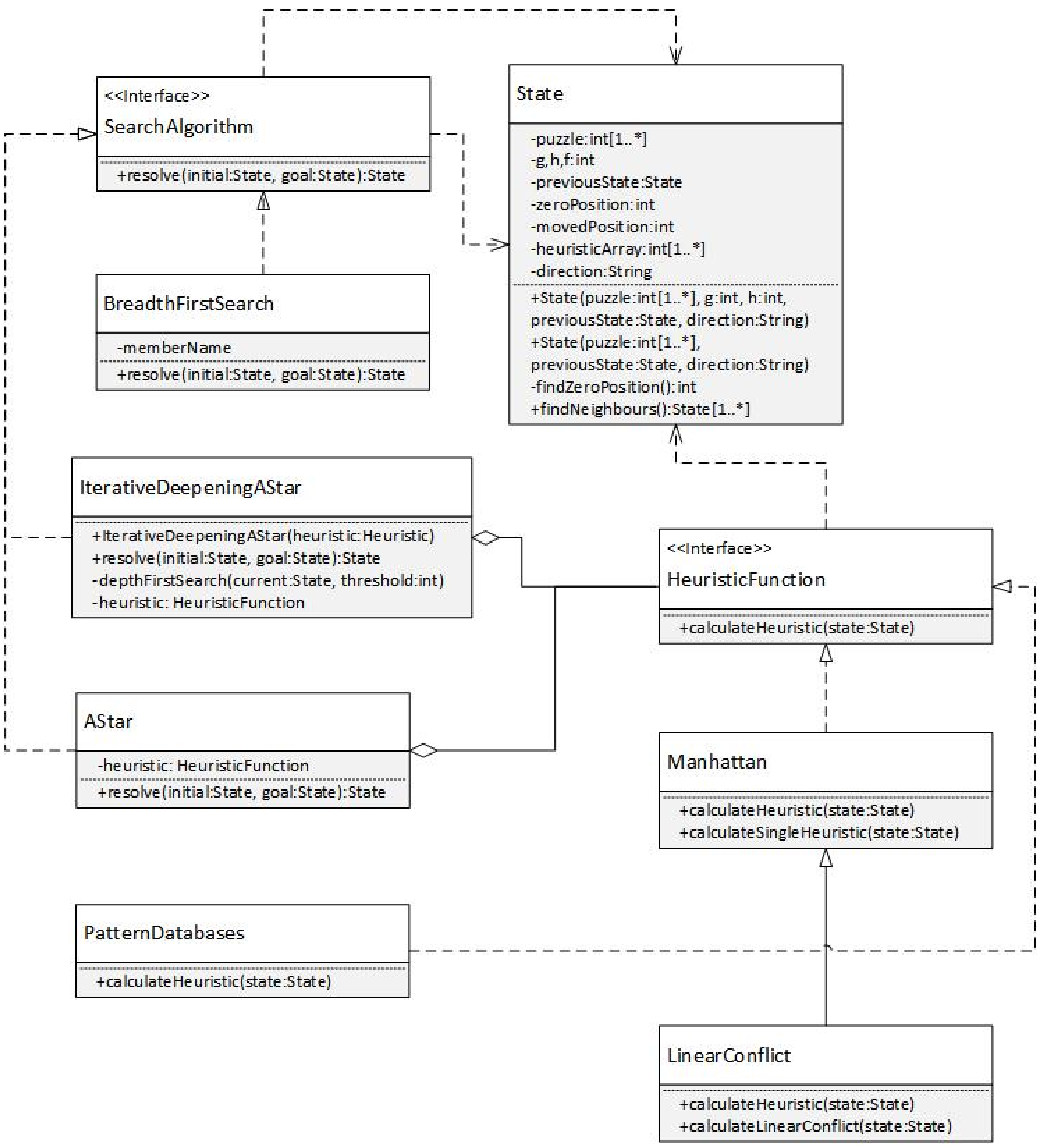
4 Implementation

This section covers the main aspects of the implementation of the project, icluding details on how the algorithms were implemented, as well as the methods used and the implementation details of these methods.

4.1 Structure

When a node is being expanded it is necessary to find its neighbours. Neighbours for the sliding-tile puzzle represent the possible moves from the current configuration, each state has a minimum of 2 neighbours and a maximum of 4. Figure 1 shows the possible moves from current state and how the direction variable would be stored ready for output; movements are relative to the blank meaning the variable is represented by the direction the blank moves. The *findNeighbours()* method returns a list of the current state's child nodes.

The strategy or policy pattern is a behavioural design pattern that enables behaviour to be selected at runtime. The strategy pattern was used in the implementation as shown by Figure 8. The family of algorithms was defined by using the *SearchAlgorithm* interface, which defines the *resolve(initialState:State, goalState:State)* which is then further implemented by the specific search classes. Strategy ensures encapsulation of each al-



<<Interface>>
SearchAlgorithm
+resolve(initial:State, goal:State):State
State
-puzzle:int[1..*]
-g,h,f:int
-previousState:State
-zeroPosition:int
-movedPosition:int
-heuristicArray:int[1..*]
-direction:String
+State(puzzle:int[1..*], g:int, h:int, previousState:State, direction:String)
+State(puzzle:int[1..*], previousState:State, direction:String)
-findZeroPosition():int
+findNeighbours():State[1..*]
BreadthFirstSearch
-memberName
+resolve(initial:State, goal:State):State
IterativeDeepeningAStar
+IterativeDeepeningAStar(heuristic:Heuristic)
+resolve(initial:State, goal:State):State
-depthFirstSearch(current:State, threshold:int)
-heuristic: HeuristicFunction
<<Interface>>
HeuristicFunction
+calculateHeuristic(state:State)
AStar
-heuristic: HeuristicFunction
+resolve(initial:State, goal:State):State
Manhattan
+calculateHeuristic(state:State)
+calculateSingleHeuristic(state:State)
PatternDatabases
+calculateHeuristic(state:State)
LinearConflict
+calculateHeuristic(state:State)
+calculateLinearConflict(state:State)

Figure 7: Class Diagram

gorithm and makes the family of algorithms interchangeable Gamma (1995). Similar to *SearchAlgorithm*, *HeuristicFunction* encapsulates the heuristic functions *Manhattan*, *LinearConflict* and *PatternDatabases*. The *LinearConflict* class is a child to *Manhattan* and performs a super method call to retrieve the Manhattan heuristic estimate for the state, then adds the linear-conflict on top.

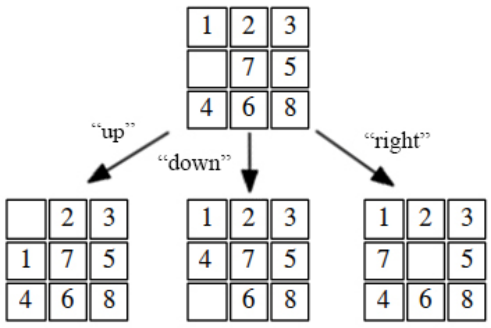


Figure 8: Finding possible moves

4.2 Experimental Plan

The experiments in this paper vary dependent on the size of the problem domain; a complete experiment was performed on the 8-puzzle, testing each and every possible state in terms of time, expanded nodes, and number of moves taken to reach the goal state.

4.2.1 Preliminaries

In order to generate testing states it is important to ensure that the state is solvable. Problem domains may have constraints that can preclude parts of the search space. For example, although the eight puzzle has 9! possible positions, only one half of them can be reached from the goal. (Culberson and Schaeffer, 1998b). To determine whether a sliding-tile puzzle is solvable we must calculate the sum of the inversions. An inversion is when a tile precedes another tile with a smaller number. Algorithm 10 was used to create testing states for the sliding-tile puzzle, states with randomised permutations were constructed and then checked to see if they were solvable, once a total of 10000 solvable test cases were created the search algorithms were then experimented with.

Algorithm 10 Is Current State Solvable

```
procedure IsSolvable(state)
    $puzzleLength \leftarrow state.size()$
    $gridWidth \leftarrow \sqrt{puzzleLength}$
    $blankRowEven \leftarrow true$
    for $i \leftarrow 1, puzzleLength$ do
        if $state[i] = 0$ then
            $blankRowEven \leftarrow (i/gridWidth) \bmod 2 = 0$
            continue
            for $j \leftarrow i+1, puzzleLength$ do
                if $state[i] > state[j]$ and $state[j] \neq 0$ then
                    $parity \leftarrow !parity$
    if $gridWidth \bmod 2 = 0$ and $blankRowEven$ then
        return $!parity$
    return $parity$
```

5 Experimental Results

To further explore the algorithms discussed a series of experiments were executed to analyse their performance to make an empirical comparison.

5.1 8-Puzzle

The initial puzzle tested was the 8-puzzle as its small number of solvable states allows for the use of unintelligent search and gives rise to the opportunity of comparison to intelligent search; the 15-puzzle has too large a search space to make a credible comparison.

5.1.1 BFS

Figure 9 displays the distribution of states in the 8-puzzle: the minimum number of moves to solve the state and the number of existing alternate states which require the same amount of moves. It also displays the results of the summed time taken to solve all of these particular states using BFS.

Table 2 displays the results of the 8-puzzle using breadth-first search. The tests were performed on all $9!/2 = 181440$ states, an initial BFS was executed from the goal configuration and each state along with the number of moves required was stored. States were split up into testing lists dependent on the number of moves as shown by column 1 and 2. Tests were then executed using breadth-first search on each state, column 3 displays the average number of nodes expanded before finding the goal state, column 4 is the average time taken to solve a given state and column 5 shows the total time taken to solve all states in the specific testing list.

Table 1: Breadth-First Search Analysis: 8-puzzle (10000 Random States)

Average Moves	Avg Nodes Expanded	Avg. Time(ms)
21.7142±0.065	89201.4297	29.7373

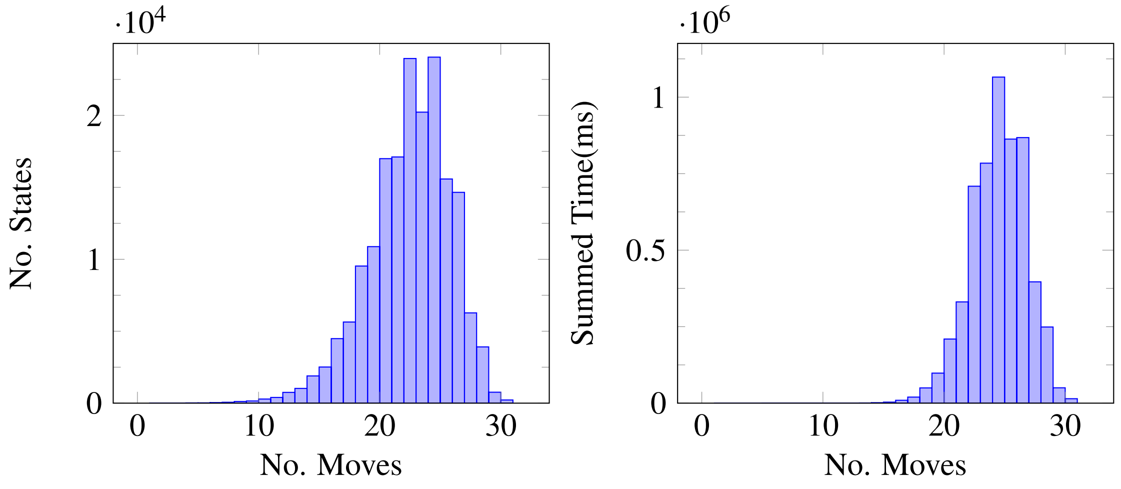


Figure 9: A representation of the number of states that require a specific amount of moves to be solved optimally in the 8-tile and the summed time taken for all these states to be solved

Table 2: Breadth-First Search Analysis: 8-puzzle

Min Moves	No. States	Avg Nodes Expanded	Avg. Time(ms)	Total Time(ms)
0-4	31	747.19±435.76	0.5483	18
5-9	389	4249.35±4277.99	1.0668	432.0
10-14	4347	23257.75±20745.99	5.7888	25164
15-19	33042	58446.06±35484.49	17.7823	587563
20-24	102326	114724.84±36410.64	41.0058	4195959
25-29	41082	162772.14±15654.31	60.0550	2467181
30+	223	180524.70±548.90	67.3542	15006

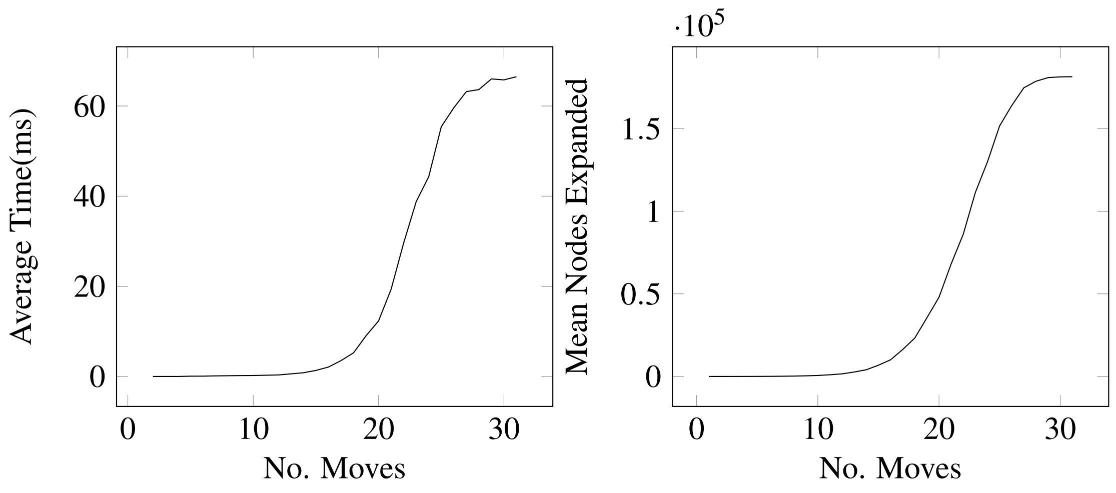


Figure 10: 8-tile analysis using breadth-first search to display how the minimum number of moves required to reach the goal state affects the time(ms) and the number of nodes expanded

5.1.2 DFID

Although DFID maintains the advantage of requiring a comparable supply of memory as depth-first search and doesn't get caught in infinite loops, DFID's time taken to find a solution is vast and generally longer than both depth-first search and breadth-first search, this is because it has to expand the same states multiple times. Figure 11 shows the exponential growth of time taken as the number of moves to solve the problem increases.

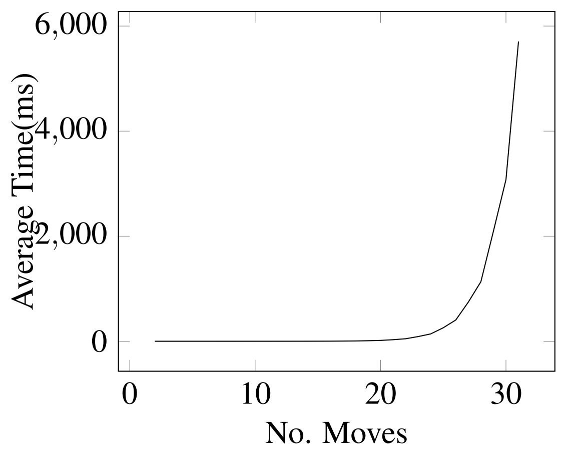


Figure 11: 8-tile analysis using DFID to display how the minimum number of moves required to reach the goal state affects the time(ms) and the number of nodes expanded

Table 3: Depth-First Iterative Deepening: 8-Tile

Min Moves	Moves	Nodes Expanded	Time(ms)
5	5	56	0.9521
10	10	6392	16.777216
15	15	3350884	2902.4583

5.1.3 A*

A* was the first informed search to be tested using both the Manhattan distance heuristic and the linear conflict. Every possible permutation of the puzzle was tested using the A* with Manhattan distance and A* with Manhattan distance + Linear Conflicts, to compare the results and analyse how much linear conflict tightens the lower bound of the estimate. Figure 12 displays the results of A* with an increasing number of moves using Manhattan distance as the heuristic estimator, the graphs show a clear exponential growth on the y axis, both in the same taken to solve the state and the average number of nodes expanded. When the number of moves required to solve the state reached 31 (The maximum number of moves to solve any sliding tile state) there is a decline in the average time taken and the average number of nodes expanded. But as there are only two states in the puzzle which require 31 moves to solve no empirical conclusion can be made until tests are performed on puzzles with a larger domain.

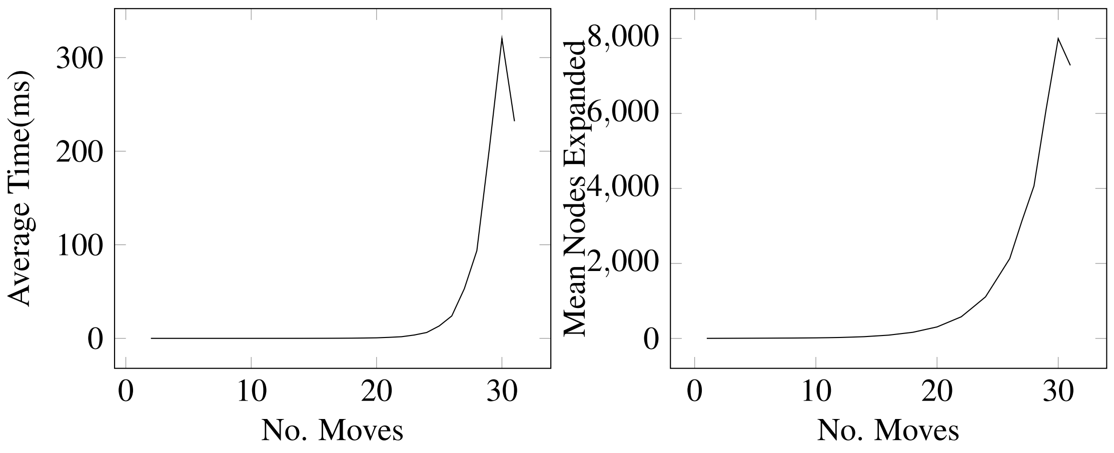


Figure 12: 8-tile analysis using A star search to display how the minimum number of moves required to reach the goal state affects the time(ms) and the number of nodes expanded

Table 4: A*: 8-puzzle (10000 Random States)

Average Moves	Avg Nodes Expanded	Avg. Time(ms)
21.7142 ± 0.065	958.989	9.8062

5.1.4 Manhattan Distance and Linear Conflict

Tests displayed in Table 4 show that A* search with the linear conflict heuristic improves the performance of A* with Manhattan distance for the 8-tile puzzle. On average A* with linear conflict would take around 1ms longer than A* with Manhattan distance, until the number of moves required to solve the puzzle were greater than 25, as this showed a substantial decline in time when comparing the linear conflict to the Manhattan heuristic. This implies that the performance of linear conflict exceeds the performance of Manhattan distance when the state space grows, so the more difficult the puzzle the greater the gap in performance between the two heuristics becomes.

As the state space grows exponentially according to the depth the growth is smaller for the A* with linear conflict, this is because the heuristic is a more accurate representation of the displacement of the tiles, thus the H value is always larger than or equal to the H value of the Manhattan Distance. Table 5 shows the difference in H value for 10 calculations of both algorithms and how the linear conflict is nearly always larger than the Manhattan distance.

Table 4 shows the exponential growth by the number of states the algorithm expanded. The average number of States for the Manhattan distance heuristic from 5 to 30 moves is 972.5, whereas linear conflict's average is 651.

5.2 15-Puzzle

Initially, the A* Manhattan distance and the A* linear conflict was implemented for the fifteen tile puzzle, the algorithm struggled due to the number of states required to be stored in the closed set.

A* uses breadth-first traversal thus the number of states that are stored increases exponentially with depth. Finding an algorithm which would require less memory but still had the benefits of a heuristic search was necessary. This is similar to the previous scenario when considering how to reduce the memory used in breadth-first by taking depth-first search's memory used as inspiration.

The memory used by Depth-first iterative deepening is linear with respect to the depth. In theory DFID could be used to solve the 15-puzzle without running out of memory, however taking the results found in Table 3, the time taken to reach the goal state for any random instance of the fifteen tile puzzle would be unreasonable.

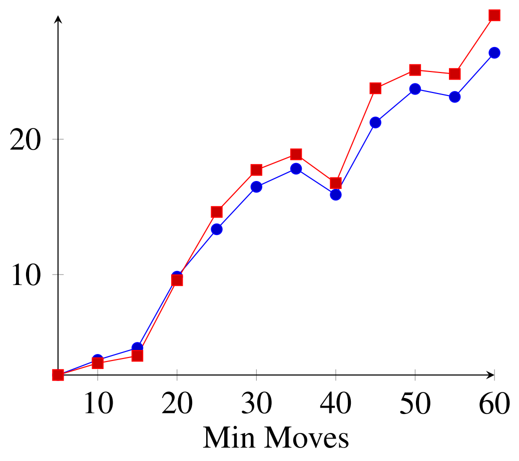
5.2.1 IDA*

Table 5: Iterative Deepening A* Results

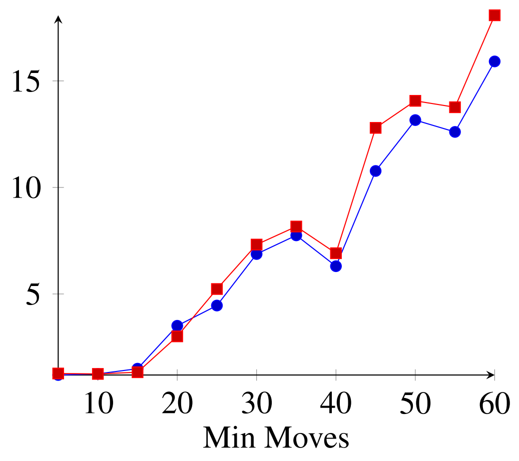
	Linear Conflict		Manhattan Distance	
Min Moves	Time(ms)	States Expanded	Time(ms)	States Expanded
5	2.2913	6	2.4060	6
10	2.3751	13	2.3739	11
15	2.8140	24	2.5081	16
20	11.3692	919	8.0932	768
25	21.9585	10420	37.5986	25272
30	117.3271	91471	159.5604	217239
35	215.4406	231508	287.0284	482893
40	78.972869	61141	120.4332	110903
45	1750.4441	2468006	7111.9494	14202684
50	9162.8202	13704525	17143.5055	36202294
55	6233.2757	9132143	13912.0609	29537720
60	61771.3520	87466771	277232.1371	596363837

5.3 Statically-Partioned Additive Database Heuristics

The 6-6-3 partitioning was initially implemented many papers state that it is the best compromise between speed and memory to other possible solutions, Figure 4 displays the used partitioning. The first two images display the fifteen tile with the subsets comprising of 6 tiles, $2 \times (16!/(16-6)!) = 11531520$ states to be stored, the final subset requires $16!/(16-3)! = 3360$ states stored that's a total of 11534880 states to that were stored. To get these disjoint groups a BFS was used once and the states was serialised into 3 separate files. When testing, a table lookup was performed and the states were stored into the data structure for testing.



(a) Log Of The States Expanded



(b) Log Of The Time

Figure 13: Logarithmic Comparison Of IDA* Fifteen Tile

5.4 Tower of Hanoi

6 Conclusion and Evaluation

References

Brewka, G. (1996). *Artificial intelligence - a modern approach* by stuart russell and peter norvig, prentice hall. series in artificial intelligence, englewood cliffs, NJ. *Knowledge Eng. Review*, 11(1):78–79.

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Culberson, J. and Schaeffer, J. (1996). Searching with pattern databases. *Advances in Artifical Intelligence*, pages 402–416.

Culberson, J. C. and Schaeffer, J. (1998a). Pattern databases. *Computational Intelligence*, 14(3):318–334.

Culberson, J. C. and Schaeffer, J. (1998b). Pattern databases. *Computational Intelligence*, 14(3):318–334.

Felner, A. (2015). Early work on optimization-based heuristics for the sliding tile puzzle. In *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Felner, A., Korf, R. E., and Hanan, S. (2004). Additive pattern database heuristics. *J. Artif. Intell. Res.(JAIR)*, 22:279–318.

Felner, A., Korf, R. E., Meshulam, R., and Holte, R. C. (2007). Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)*, 30:213–247.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Hansson, O., Mayer, A., and Yung, M. (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227.

Hansson, O., Meyer, A. E., and Yung, M. M. (1985). *Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models*. Department of Computer Science, Columbia University.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107.

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

Korf, R. E. (1995). Heuristic evaluation functions in artificial intelligence search algorithms. *Minds and Machines*, 5(4):489–498.

Korf, R. E., Reid, M., and Edelkamp, S. (2001). Time complexity of iterative-deepening-a*. *Artif. Intell.*, 129(1-2):199–218.

Linnert, B., Schneider, J., and Burchard, L. (2014). Mapping algorithms optimizing the overall manhattan distance for pre-occupied cluster computers in sla-based grid environments. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 132–140.

Meissner, A. and Brzykcy, G. (2011). Reasoning with the depth-first iterative deepening strategy in the dlog system. In Meersman, R., Dillon, T. S., and Herrero, P., editors, *On the Move to Meaningful Internet Systems: OTM 2011 Workshops - Confederated International Workshops and Posters: EI2N+NSF ICE, ICSP+INBAST, ISDE, ORM, OTMA, SWWS+MONET+SeDeS, and VADER 2011, Hersonissos, Crete, Greece, October 17-21, 2011. Proceedings*, volume 7046 of *Lecture Notes in Computer Science*, pages 504–513. Springer.