A Pattern Database Approach for Solving the TopSpin Puzzle Problem

Pier Paolo Bortoluzzi
University of Basel

Examinar: Malte Helmert
Supervisor: Martin Wehrle

Abstract

Finding optimal solutions for general search problems is a challenging task. A powerful approach for solving such problems is based on heuristic search with pattern database heuristics. In this thesis, we present a domain specific solver for the TopSpin Puzzle problem. This solver is based on the above-mentioned pattern database approach. We investigate several pattern databases, and evaluate them on problem instances of different size.

CONTENTS

I Introduction **3**

II Preliminaries **5**

a TopSpin Puzzle Problem 5

b Heuristic Search Algorithms 6

c Heuristics 7

d Pattern Databases 9

III Implementation **11**

a Representation of States 11

b Pattern Database Implementation 11

c Investigated Patterns 12

IV Experiments **14**

a Pattern Sizes 14

b Pattern Databases 15

c Search Algorithms 17

d Discussion 18

V Conclusion **19**

Stop looking for solutions to problems and start looking for the right path.
Andy Stanley

I

INTRODUCTION

Solving a puzzle can be highly interesting or downright frustrating, but nevertheless, finding the solution is always exhilarating. When a puzzle becomes increasingly difficult to solve, most people shy away or even cheat and look up solutions. But there is a third and for computer scientists the more preferable solution: implementing a search algorithm to solve the puzzle for you, which is technically not cheating...

The *TopSpin* puzzle [6] is one of the more intriguing ones which can be put in the same shelf as the Rubik's cube [1] when comparing possible permutations. With its standard $20!$ distinct permutations it almost beats the Rubik's cube in complexity but obviously it looks less impressive and is not a feat of engineering; it never reached the same level of fame. But solving it is still quite complicated even for a computer. The *search space* for this problem is too large such that the memory needed to find a solution can exceed most standard PCs. On the other hand, finding an efficient solving strategy is equally challenging.

The TopSpin puzzle was originally designed by Ferdinand Lammertink and patented in 1988. The puzzle itself consists of an oval track containing 20 elements. The elements can be moved on the track by shifting all at once in either direction. The only way to change the order of the elements is to turn the top part of the puzzle which results in reversing the order of the 4 elements on the turntable.

In this thesis we investigated several domain-specific pattern databases [2] for the TopSpin puzzle that enhanced the solving speed of a best-first search algorithm which finds optimal solutions. The chosen search algorithm is IDA* [5] as it is space efficient and is able to use a heuristic to enhance search speed. The heuristic used was the pattern database approach with different types of problem space *abstraction*. The solving speed has been compared to different types of searching algorithms.

The thesis is organised as follows. The second chapter discusses the TopSpin puzzle itself and serves as a refresh of our knowledge of search problems and heuristics with a focus on pattern databases. The third chapter contains a more in-depth description of how we represented the search problem and what types of pattern databases have been used. In the forth chapter, we present and discuss our results. In particular, we compare our different pattern databases. As a second analysis we looked at search speed

of IDA* compared to breadth-first search and A* [3]. The fifth and last chapter contains our conclusions about this thesis and outlook about possible methods which could improve the pattern databases further.

Success is neither magical nor mysterious.
Success is the natural consequence of
consistently applying the basic fundamentals.
Jim Rohn

II

PRELIMINARIES

In this chapter we are going to introduce the very basics of search problems and how they are solved. First we provide the terminology which will be used throughout this thesis. This is followed by a short overview of what an actual search problem is. After we established what a search problem is we are going to look at standard search algorithms and how heuristics are supposed to increase search speed. To round off this section we are going to explain how pattern databases work.

a. TopSpin Puzzle Problem

A search problem usually has a starting *state* where the search for a *solution* begins. A state contains all the information to describe its position in the *search space* of the given problem. The search space is made of all possible states the problem can occupy. The search space contains the start state and the *goal* state.

The solution of search problems are usually represented as *paths* from the start state to the goal state. This path can be seen as a chain of actions. Internally the search algorithm stores these states with additional information which are then called *nodes*. These nodes usually contain information of the previous node, to allow generation of the solution, and the cost of the path so far which is called the *path-cost*. Additionally algorithms like A* store the estimated cost of the path to the goal which will be relevant when we're discussing heuristics later in this chapter.

In our case the search problem is finding the path from any scrambled state to the solved state. A state in the TopSpin puzzle is any possible arrangement of the elements and the goal state is where the elements are in ascending order. This can be achieve by using one of the three possible actions: reversing the order of the elements on the turntable and shifting all elements in either direction. By applying any action to a state we arrive at a *neighbour state*. In this search problem each state has only three neighbours.

The TopSpin puzzle has been originally proposed to have twenty elements and a turntable that reverses the order of four elements. These two parameters can be varied to change the problem size. In the future the number of total elements will be referred to as N and the size of the turntable as k. We remark that the standard TopSpin puzzle with $k = 4$ and $N = 20$ is special because every possible permutation can be reached without disassembling the puzzle[**?**].

This means that there are 20! possible states.

Note that solving the TopSpin puzzle manually isn't as simple as it seems. First: one has to solve the numbers 1 to $N - k$ which can be done quite easily. The last k elements can be solved by using solving patters to either swap two elements without changing the order of the others or cycling all k elements. Theses approaches like the solving patterns for the Rubik's Cube are rather inefficient but at least allow humans to actually solve every possible scrambled state.

b. Heuristic Search Algorithms

In this section we are going to introduce heuristic based search algorithms. There are many different search algorithms and knowing their benefits and disadvantages is important. Some algorithms try to find any solution and others aim for the *optimal solution*. Finding a *best solution* costs usually more time. A best or optimal solution is called the solution with the least path-cost. The path-cost is the combined cost of each action of the current path.

For solving our puzzle optimally we need a best-first-search algorithm which finds the best solution first. For problems with uniform costs, like ours, breadth-first-search does this but it is time and space inefficient. Best-first-search algorithms are usually implemented using a priority queue as an open queue like A*. These algorithms use the *priority* of a state which is a computed estimate of the costs from the state to the goal. This cost is calculated by a priority function f. This function f combines the current path-cost and a *heuristic* estimate of the cost to the goal state. The open queue contains all *evaluate* but not yet *explored* nodes. A node is explored when each possible neighbour has been evaluated (priority calculated). The priority queue sorts the nodes such that the next pulled node has the lowest priority value. A secondary set called the closed set contains all already explored nodes.

A* is the most famous best-first search algorithm which finds the least cost solution. A* is *complete* viz. it will always find a solution if it exists like breadth-first search. Let's take a closer look at the pseudo code of A* in Algorithm 1 [1]. First we insert the start node into the priority queue. The while loop ensures that every node from the open queue will be evaluated until a solution has been found. We pull the next

Algorithm 1: A* algorithm

```
openQueue.init();
closedSet.init();
openQueue.push(startNode);
while openSet is not empty do
    node := openSet.pull();
    if node == goal then
        return pathToSolution(node);
    closedSet.Add(node);
    for each nextNode of node do
        if
        closedSet.contains(nextNode)
        then
            continue;
        compute priority f(nextNode);
        insert nextNode into openSet;
return noSolution;
```

[1] This algorithm only works for certain classes of priority functions which will be discussed in the next section

node from the priority queue and add it to the closed set. If this node is equal to the goal we return the path to this node. Otherwise we expand its neighbours and calculate their estimated cost to the goal and insert those new nodes into the open queue as long as they are not in the closed set. The priority queue sorts the nodes by the priority value.

Another thing we notice is that while using an open queue and a closed set every single node processed will be stored, thus continuously increasing memory usage. This is not a hindrance for smaller problems but when solving large puzzles we could run out of memory very fast. If this is the case, one has to switch to a slower but more space efficient algorithm like *IDA**.

IDA* is an abbreviation for iterative deepening A* and is a mixture between iterative deepening depth first search [7] (IDDFS), and A*. IDDFS is space efficient as it only stores the path from the start node to the current node. Depth first search expands each evaluated node but only tracks one of the neighbours until the depth limit is reached. After reaching this limit it will return to the previous node exhaust every possible neighbour node and so forth. After evaluating every possible node without finding the goal it will return to the root and the depth limit will be increased by 1. Thus the total amount of searched nodes until a certain depth d is reached will be

$$\sum_{i=0}^{d}(d+1-i)b^i \quad [7],$$

where b is the number of neighbours. For the TopSpin puzzle, parameters could be $d = 15$ and $b = 3$ thus a maximum of 32285032 nodes would been searched. Instead of using an arbitrary depth limit, IDA* uses the priority value to limit its depth search. This limit will be updated when every possible node within the momentary limit is evaluated and no solution has been found. The new limit is the minimum of all newly observed priority value.

IDA* is implemented as a recursive algorithm. In Algorithm 2 we can see a pseudo code of IDA*. First the cost limit is set by the priority value. Then the recursive function is called which returns the new next cost limit and if found, the solution. The recursive function first calculates the priority value of the node if it exceeds the cost limit it will return the priority value as potential new cost limit or if it's the goal it will return the solution. If neither is the case each neighbour will be evaluated with a recursive call. The next cost limit is then the minimum of all new cost limits, which will be returned if no solution is found. Only the main loop can increase the cost limit.

IDA* increases search time on the one hand but on the other hand reduces memory usage which is usually preferable for larger problems.

c. Heuristics

In the last section, we described heuristic search algorithms. Preferably we want to explore states with better priority values. In this section, we describe in more detail how the priority values are computed. A common way of computing these values is based on *heuristics*. Heuristics predict the possible cost to the goal

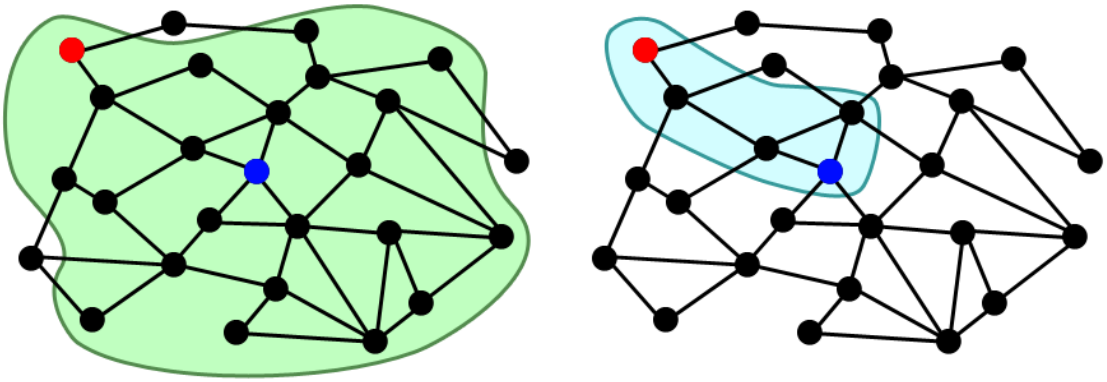

Figure 1: The green area on the right contains all searched nodes by a BFS. The cyan area on the right represents the nodes searched by an algorithm using a heuristic. Where the start node is in blue and the goal node in red.

Algorithm 2: IDA* algorithm designed for uniform search problems

```
costlimit = f(startNode);
while costLimit ≠ ∞ do
    costLimit, solution = depthSearch(startNode, costLimit);
    if solution exists then
        return pathToSolution(solution);

depthSearch(node, costLimit){
    minimumCost = node.cost() + h(node);
    if minimumCost > costLimit then
        return minimumCost, noSolution;
    if node is goal then
        return costLimit, node;
    nextCostLimit = ∞;
    for all neighbours of node do
        neighbour.setCost(node.Cost() + 1);
        newCostLimit, solution = depthSearch(neighbour, costLimit);
        if solution exists then
            return newCostLimit, solution;
        nextCostLimit = min(nextCostLimit,newCostLimit);
    return nextCostLimit, noSolution;
}
```

state. This information is usually based on a specifically designed function for the state space.

A good example to demonstrate heuristics is the shortest path problem. Imagine a city where crossings are the states and the streets describe each possible next neighbour. What is the shortest path from point A to point B? Searching for the shortest path can be quite costly if there are many streets to chose from at each crossing. The breadth-first search algorithm would expand its search into every possible direction and waste time on searching in the wrong direction. An easy solution to know if the search expands into the right direction would be to calculate the beeline from the current searching position to B. We know the cost to the position we are at g and we can predict that the distance to the goal B is at least the calculated beeline h. Thus the total estimated cost to the goal is at least $f = g + h$. In this example, sorting all nodes by f and picking the smallest value makes sure that the search algorithm expands into the right direction and reduces the time spent searching. Figure 1 shows how many states are never explored because their f-score is too high to be considered.

As mentioned in the previous section the A* Algorithm **??** is only valid for certain priority classes. They are called consistent heuristics. A heuristic is consistent when

$$h(s) \leq h(s') + 1 \quad \text{and} \quad h(s^*) = 0\,,$$

where s' is the previous node and s^* is the goal state. This means that the algorithm will never arrive at a higher priority value when the heuristic predicted a lower value. In this case the action cost is uniform.

In the context of optimal search, another important property of heuristics is the knowledge that the predicted cost to the goal is smaller or equal the actual cost. Formally, this means that

$$h(n) \leq C(n) \text{for all nodes } n, \qquad (1)$$

where n is a node and $C(n)$ the actual cost for the node to the goal. If this equation does not hold the search might end up in the wrong direction, be stuck in a dead-end or the solution found does not have to be the best solution there is. A heuristic that fulfils the condition (1) is called an *admissible* heuristic. Every consistent heuristic is also admissible but not the other way around.

Finding an admissible heuristic can be quite easy given a simple problem e.g. the *sliding puzzles* [4]. In this example counting all the misplaced tiles is an admissible heuristics. However, for more complex problems finding, a good and admissible heuristic can become more complicated. This is where pattern databases are useful.

d. Pattern Databases

Pattern databases are basically lookup tables where for each state in the search space there is a pre calculated cost of the path to the goal. Though instead of using the search space of the problem pattern databases map the problem space onto an *abstract* space, which is usually a lot smaller. Important is that each state in the search space maps onto a corresponding abstract state. This reduction allows the pattern database to contain each abstract state and store their cost to the abstract goal. Consider In Figure

2 as an example. The search space is represented by the black circles. Let the centre state be the goal state. The coloured areas which combine several states together are the abstract states. The colour of the areas represent the path-cost to the abstract goal, which grow by one with each step away from the abstract goal.

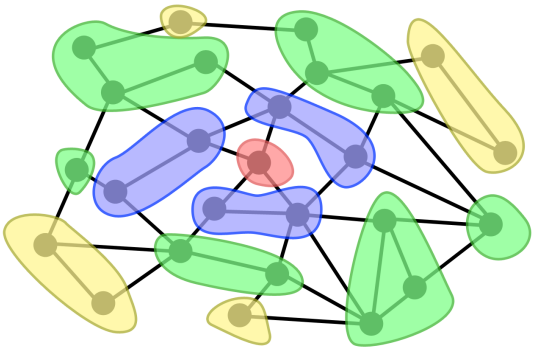

Figure 2: Visualisation of the abstract space. The different areas combine certain states into abstract states and the cost to the solution is represented by the colour. Red is the goal, blue equals cost of 1, green is 2 and yellow is cost of 3

The *pattern* describes which parts of the original problem are kept and which become interchangeable. The pattern defines a mapping function from the search space into the abstract space. The less information we keep in the abstract state the smaller the abstract space becomes. A small abstract space is less expressive as the path-costs will be smaller as well but it is processed a lot faster. This is the trade-off pattern databases must be balanced around.

In Figure 3 the 15-puzzle is shown. On the left side is the puzzle in the search space and is mapped into an abstract state on the left. The red marked area represents the pattern. Even though the blue part is still scrambled the heuristic will

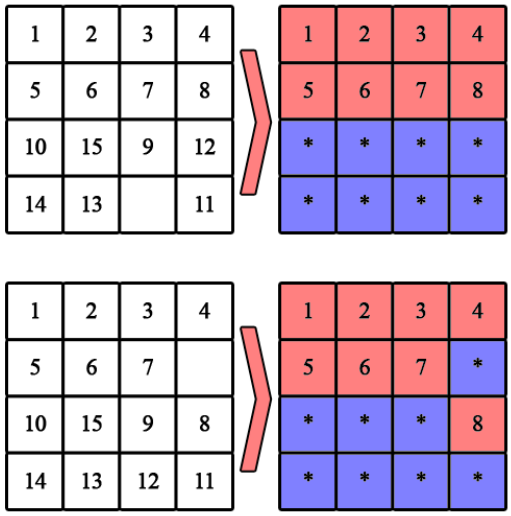

Figure 3: The 15 puzzle in a given state is mapped into the abstract space. The top represents a state which has $h(s) = 0$ and the bottom state has $h(s) = 1$

return $h(s) = 0$ as the mapped state is the abstract goal state. The bottom state mapped into the abstract space is only one action away from the goal state and thus return $h(s') = 1$.

The advantage of pattern databases, compared to other heuristics, is that they can be precalculated and stored for later use. Another positive aspect of pattern databases is that because of their design they are most of the time admissible.

Problems that are too complicated to find a good admissible heuristic for or problems which will be solved many times over are predestined for pattern databases. We remark that in case of repetitive usage a more complex pattern could be chosen. It may take longer to build the database but once it stands and many different searches have been completed the time consuming process of building the database becomes less important.

The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
Sir William Bragg

III

IMPLEMENTATION

In this section we will take a closer look at our implementation of the search problem and because we implemented the search algorithms and pattern databases ourselves we will also describe how we are building the pattern databases. We will also discuss the different patterns we chose to evaluate in our experiments.

a. Representation of States

The search problem at hand has states which will be represented by a tuple. The tuple contains the numbers in the order in which they appear in the puzzle. The length of the tuple is N (k will be stored separately as it won't change during the search). In Figure 4 we can see a representation of such a tuple; note that the line between the 4th and 5th is purely a visualisation of $k = 4$ in this example.

Each state has three neighbours: one with the reversed order of the first k elements, one with all elements shifted to the right and one to the left. When the search algorithm asks for the next neighbours these three new states will be returned.

1	2	3	4	5	6	7	8	9	10	11

Figure 4: Tuple representation of the TopSpin puzzle with $N = 11$ and $k = 4$.

Because the pattern database will need to solve abstract states of the puzzle we simply replace the values which the pattern maps as interchangeable with a wild card and modify the goal to match the abstract state.

b. Pattern Database Implementation

The computation of the pattern databases usually starts at the abstract goal state. Then each neighbour gets evaluated and stored in the database. This is repeated until all abstract states have been evaluated. However this forces the pattern database to keep track of each evaluated abstract state which means it has to store the complete abstract space.

In our implementation we started from each possible abstract state by creating a lexicographical list of all permutations. Then we compute for each state the path-cost to the abstract goal state and insert this value into a hash table. The hash value is calculated by using *zobrist hashing* [8]. Zobrist hashing calculates hash values with very low collision rates. In a rare case of a collision we pick the lower cost value to ensure that our pattern database stays admissible. As

an advantage, we don't have to store the abstract states but the heuristic probably won't be *consistent* any more. A heuristic is consistent when $h(N) \leq c(N, P) + h(P)$ and $h(G) = 0$ where h is the heuristic and c the cost of N to P. Our experiments have shown that no collisions have been detected which means our pattern database is consistent.

Zobrist hashing uses a single $N \times N$ table with random values. One dimension represents the N positions available in the puzzle and the other dimensions represents the N element. Such that each element at every position has a unique value within the table. To calculate the hash value we bitwise XOR all values from the table which represent the state. For example if the numeral 5 is at position 1, 6 at position 2 and 8 at position 3 etc. the first three operations would be ((((0 XOR table(5,1)) XOR table(6,2)) XOR table(8,3)). The order in which we XOR these values is irrelevant as XOR is associative. This means effectively we simply hash the pattern as it appears in the abstract state. Our database is therefore a lot smaller when not storing any actual state. For each possible state we only store 2 values instead of $N + 1$. As a consequence at the possible cost of slower building speed.

The solver is capable of using multiple pattern databases at once. It then picks the highest cost value from the pattern databases. This allows us to combine several patterns to enhance the search speed. This is because we always pick the higher cost value which means we have a better estimate.

c. Investigated Patterns

Our pattern database implementation allows us to modify the pattern on the fly with passing either one or multiple pattern files to the solver. This makes it easier to experiment with different patterns and pattern combinations to find out which have the best potential. We will introduce the four different patterns we used for our benchmarking.

Please note that each pattern will use 0 as the wild card.

i Linear

The first pattern represent the first n elements of the problem. This approach was practical as it can be used with every problem with $N \geq n$. We are going to look at two different sets of this pattern, one as seen in Figure 5 on the turntable and a second one off the turntable to find out if there is a significant difference.

1	2	3	4	0	0	0	0	0	0	0

Figure 5: Standard linear pattern

ii Stride

The second pattern is a superset of the linear pattern. Here we simply add a gap between each element. Using a gap size of one results in an even/odd pattern as seen in Figure 6 the top tuple. The bottom tuple uses a gap size of two. Increasing the gap value could lead to a pattern that can span the entire problem.

1	0	3	0	5	0	7	0	0	0	0
1	0	0	4	0	0	7	0	0	10	0

Figure 6: Pattern with gaps in-between the elements with gap-size one and two.

iii Multiple Pattern Databases

Last but not least we used multiple pattern databases to estimate the distance to the goal state. We created out of each of the two shown pattern a mixtures of pattern database to find out if the combinations increase the search speed as much or even more than the increase in building time of the pattern databases. In Figure 7 we combined two stride patterns one representing the first $n = 4$ odd elements and the second pattern the first $n = 4$ even elements.

1	2	3	4	5	6	7	8	0	0	0

Figure 7: Each colour represents a different pattern which are evaluated separately but shown together.

Insanity: doing the same thing over and over again and expecting different results.
Albert Einstein

IV

EXPERIMENTS

In the first part we compare the efficiency of different pattern sizes of each previously introduced pattern to find out at which point the patterns are most efficient. Then we compare the different patterns with each other to find out which of them is the most effective for different problems sizes. Afterwards we used the fastest of these patterns to compare them to the different search algorithms: A* and breadth-first search (which of courses doesn't use the pattern database).

We used for all problems $N = 9$ and $k = 4$ unless otherwise noted. We averaged all values over 20 different random problems. The first column describes the explored nodes. We ignored the evaluated nodes as IDA* has usually almost as many evaluated as explored nodes. The second and third column represents the search time and pattern database building time (short PDB time). The last column shows the total time. All time values are represented in seconds. The second row in the tables of each pattern entry shows the standard deviation which is important to properly analyse the results. All benchmarks have been performed on the same computer with an Intel Core 2 Quad CPU Q8400 2.66GHz.

a. Pattern Sizes

i Linear

In Table 1 we can clearly see that the smaller patterns are less robust and slower than their larger counterparts. The (5 6) pattern probably outperformed (1 2) because the pattern in its goal state is off the turntable, which means the average path-cost will be larger. Another interesting fact is that the larger problems spend almost the same time in building the pattern database and the search. In contrast the smaller pattern databases spent all their time in the search. We will pick pattern (1 2 3) for further investigation as it performed best and has the lowest standard deviation.

ii Stride

The stride patterns performed almost the same compared to the larger linear patterns. Their PDB times are all smaller than the ones for the linear patterns as we can see in Table 2. The shorter PDB times lead to longer search times except for (1 3 5) and (1 4 7) which are even quite robust. It will be interesting to see how well they will scale for different problem sizes. All of the other patterns show longer search times than

Pattern	Explored	Search time	PDB time	Total time
1 2	29842550.4	20.550	0.034	20.584
sd	47939863.7	33.454	0.017	33.460
5 6	12559905.6	8.278	0.036	8.313
sd	16838833.2	11.012	0.008	11.010
1 2 3	2410149.0	1.942	2.092	4.034
sd	3326810.0	2.667	0.049	2.652
5 6 7	1815534.5	1.414	3.766	5.180
sd	4168910.4	3.208	0.055	3.235

Table 1: Results for linear patterns comparing explored nodes, search, pattern database building and total time averaged over 20 random problems. Each second row shows the standard deviation of the mean values above.

PDB times which makes them less interesting for scalability. None of them achieved a robustness as good as the ones we picked out. These results show that in general stride patterns are less robust and perform worse than linear ones.

iii Multiple Databases

The multiple database approach has shown to be the most effective in terms of smallest search time and total solving time. The small linear pattern approach has been shown to be ineffective before but when we combine several of them together we arrive at a very fast PDB time and a surprisingly good search time. The pattern combination (1 2)(3 4)(5 6)(7 8) is so far the fastest of them all. On the other hand we have the pattern (1 2 3)(5 6 7) with the highest PDB time but the fastest search time of all patterns so far. The (1 2 3)(5 6 7) pattern is a perfect example of pattern databases because in this case building the database once and using it for multiple search problems would be the most cost effective solution of them all. It is even very robust with the lowest standard deviation yet.

All the other patterns are almost indistinguishable in their solving time.

b. Pattern Databases

In this section we will examine most of the previously selected patterns on different problem sizes to find out which of the selected patterns has an overall best performance. We had to modify the patterns slightly such that they can fit into smaller sized problems. We also create new and more random problems to get a better average. Each graph shows the PDB, search and total time of the patterns across problem sizes from six to nine. The best performing pattern will then be used for our final evaluation comparing IDA* to A* and breadth-first search.

In Figure 8 we can see that the (1 2 3) pattern grew quickly with the problem size. The PDB to search time ratio stays almost the same throughout each problem size. This pattern is very robust and with it's relative low solving times, one of the best pattern we have found.

The slightly modified (1 4 6) pattern performed very badly compared to our previous results. This might be due to us modifying the pat-

Pattern	Explored	Search time	PDB time	Total time
1 3 5	2118077.0	1.769	1.909	3.678
sd	3465754.5	2.895	0.073	2.947
2 4 6	3041088.8	2.514	1.934	4.447
sd	5521276.4	4.502	0.067	4.490
1 4 7	2227634.0	1.836	1.794	3.630
sd	3340768.7	2.736	0.024	2.737
2 5 8	4989883.1	4.080	1.865	5.944
sd	11002262.0	8.937	0.081	8.951
3 6 9	8475263.9	6.517	1.839	8.355
sd	13186127.4	10.047	0.075	10.047
1 5 9	8964459.3	6.920	2.057	8.977
sd	17328509.6	13.299	0.027	13.306

Table 2: Results for stride patterns comparing explored nodes, search, pattern database building and total time averaged over 20 random problems. Each second row shows the standard deviation of the mean values above.

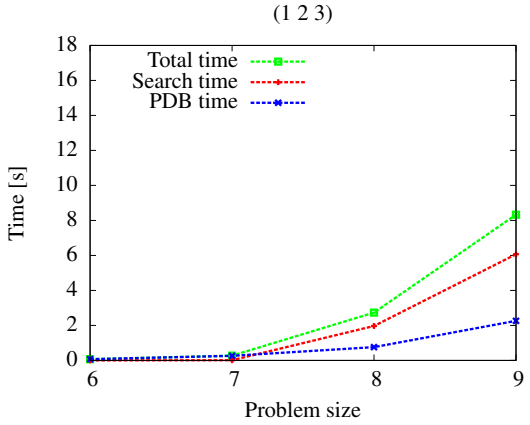

Figure 8: Results of the time measurements over four problem sizes with the (1 2 3) pattern.

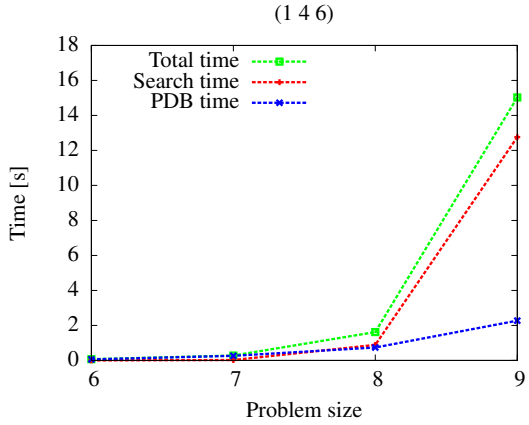

Figure 9: Results of the time measurements over four problem sizes with the (1 4 6) pattern.

tern but this is probably just caused by the larger set of problems. If we look at the graph in Figure 9 we can see that the search speed increased disproportionally compared to before. This is probably caused by an unlucky set of problems. We mentioned before that the stride patterns weren't performing that well and this confirms our previous assumption. Nonetheless it had the worst results compared to every other pattern.

The (1 2 3)(4 5 6) pattern even with the light modification solved each problem the fastest. The interesting property of this pattern database is that the PDB time is higher than the search time by a big margin. This is the only pattern where we found this occurrence. We have seen that it performed quite well in the previous results but still this outcome came as a surprise. The one with the highest PDB time still man-

Pattern	Explored	Search time	PDB time	Total time
1 2 ,5 6	4277835.9	5.440	0.070	5.509
sd	7049937.9	9.244	0.012	9.245
1 2 ,4 5 7 8	1361676.3	2.075	0.093	2.168
sd	1623383.1	2.453	0.027	2.446
1 2 ,3 4 ,5 6 ,7 8	713799.3	1.549	0.135	1.684
sd	900026.4	1.967	0.024	1.975
1 2 3 ,5 6 7	66153.2	0.106	6.763	6.869
sd	62540.3	0.094	0.406	0.399
1 3 5 ,2 4 6	219812.9	0.348	4.282	4.630
sd	383164.8	0.602	0.352	0.638
1 4 7 ,2 5 8	301834.8	0.475	4.104	4.579
sd	394720.2	0.611	0.246	0.563
1 4 7 ,3 6 9	445714.2	0.661	3.900	4.561
sd	705597.0	1.048	0.268	1.062
2 5 8 ,3 6 9	517225.1	0.786	4.135	4.921
sd	666857.2	0.997	0.329	0.973

Table 3: Results for multiple patterns comparing explored nodes, search, pattern database building and total time averaged over 20 random problems. Each second row is showing the standard deviation of the mean values above

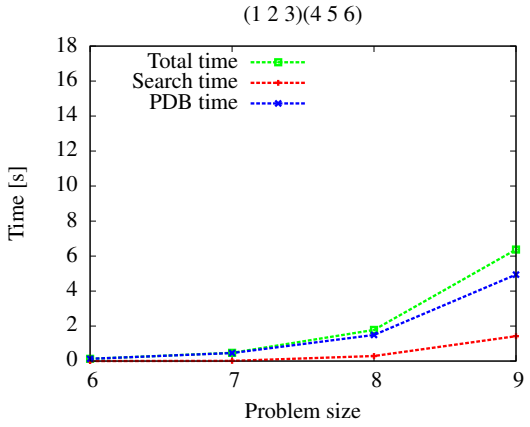

Figure 10: Results of the time measurements over four problem sizes with the (1 2 3)(4 5 6) pattern.

aged to be the fastest. In Figure 10 we can see that in the smaller problem sizes the PDB dominated the solving time until the problem became larger.

c. Search Algorithms

The pattern (1 2 3)(4 5 6) outperformed all of the other patterns and its unique property with more PDB time than search time makes it a very valuable pattern combination. In this last section we will show the difference in solving speed of A*, IDA* and breadth-first search. We will use the same problem set as the section before. Problem sizes from six to nine with each twenty problems. We only compare the overall solving time, explored and evaluated nodes.

As we can see in Figure 11 IDA* evaluates a lot more nodes than breadth-first search and A*. In Figure 12 we can also see that IDA* and breadth-first search explore almost the same amount of nodes as they evaluated but A* explored only half the nodes it evaluated. The final but most impressive plot in Figure 13 shows

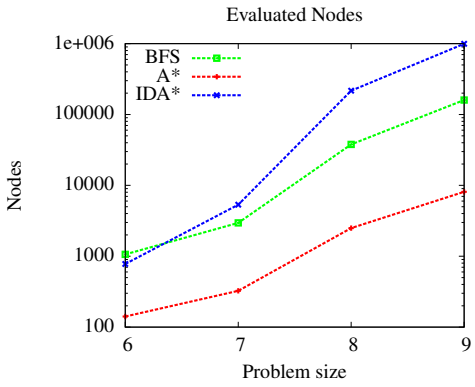

Figure 11: Comparing the number of evaluated nodes by all three algorithms. The mean value is calculated over 20 random problems. Y axis is in log scale.

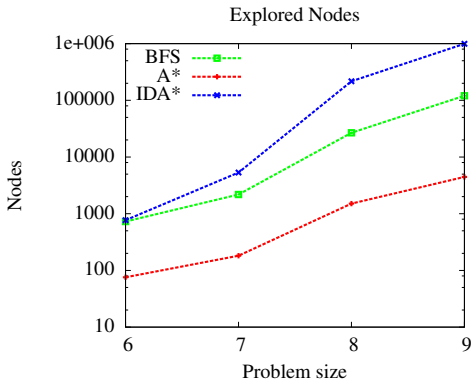

Figure 12: Comparing the number of explored nodes by all three algorithms. The mean value is calculated over 20 random problems. Y axis is in log scale.

that breadth first search is faster for smaller problems but between problem size seven and eight breadth-first search literally explodes. At problem size nine breadth-first search is 100 times slower than A* and IDA*. Surprisingly IDA* could keep up with A* but is slower by a fifth.

d. Discussion

The experiments have shown that the chosen pattern can have a huge impact on both the pattern database building time and the search time. Patterns with the size of three have shown the best results everything above will increase the database build time exponentially and everything below increases the search time.

The comparison between the different search algorithms was pretty much what has been expected breadth-first search was the slowest for larger problem sizes. A* was faster than IDA* which was clear from the beginning but the small margin was actually surprising. The only reason this could be is because A* and breadth-first

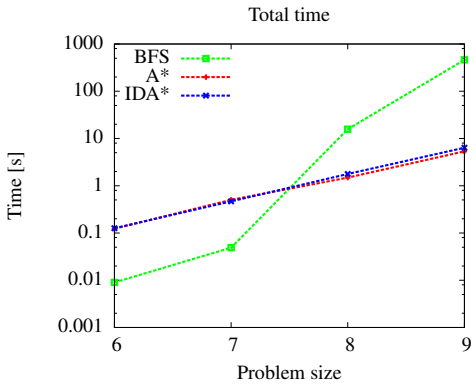

Figure 13: Comparing the time used to solve the problem. The mean value is calculated over 20 random problems. Y axis is in log scale.

search need a lot more space and thus have to allocate more memory which is probably the bottleneck and allows IDA* to keep up. The TopSpin puzzle itself scaled very well.

If . . . the past may be no rule for the future, all experience becomes useless and can give rise to no inference or conclusion.
David Hume

V

CONCLUSION

We created our own solver for the TopSpin puzzle problem with a domain specific pattern databases. We have shown that the choice of patterns is important for solving speed and knowing which will be better is not something that can be done intuitively. The experiments have also shown that even for a large set of problems the variance can be very high. Solving speeds from 2 seconds to 100 seconds isn't unusual for the same problem size and same pattern database. We have also used a different approach to build a pattern database and it worked quite well. Zobrist hashing has also proved how simple one can calculate hash values with very low collision rates. In our case we haven't encountered a single collision. Which makes this implementation even for A* consistent.

Using pattern database to solve a search problem has been shown to be quite successful and without a lot of tweaking the solving speed was way ahead of an exhaustive search.

For future projects it would be interesting to evaluate the patterns for larger problem sizes and find out if they scale well with size or if the patterns have to be scaled as well. Comparing the solver to a universal solver would be interesting as well to see how much faster our solver is. And last but not least, implementing additive pattern database heuristics.

REFERENCES

[1] Hana M. Bizek. *Mathematics of the Rubik's Cube Design*. Pittsburgh, Pa.: Dorrance Pub. Co., 1997.

[2] Schaeffer J. Culberson, J. Pattern databases. *Computational Intelligence*, 14 (3):318–334, 1998.

[3] N. J.; Raphael B. Hart, P. E.; Nilsson. Correction to a formal basis for the heuristic determination of minimum cost paths. *SIGART Newsletter*, 37:2829, 1972.

[4] Wm. Woolsey Johnson. Notes on the 15-puzzle (1). *American Journal Mathematics*, pages 397–399, 1879.

[5] Richard Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97109, 1985.

[6] Ferdinand Lammertink. Puzzle or game having token filled track and turntable, 1988. United States Patent.

[7] Peter Russell, Stuart J.; Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Upper Saddle River, New Jersey: Prentice Hall, 2003.

[8] Albert L. Zobrist. A new hashing method with application for game playing. *Technical Report*, 88, April 1970.