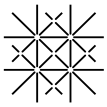


University of Basel

Heuristics for TopSpin

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence Research Group
https://ai.dmi.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Tanja Schindler

Marco Dreher
marco.dreher@stud.unibas.ch
2021-057-088

13.07.2025

Acknowledgments

I would like to thank my supervisor Tanja Schindler for always being patient with me, helping me through her feedback and discussing ideas with me. Her support has been invaluable to me and has made writing the thesis fun and engaging. I also would like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my thesis in the Artificial Intelligence Research Group. Most of all, I would like to thank my family for always supporting me through my studies and motivating me to keep going.

Abstract

Puzzles serve as important benchmarks in artificial intelligence, challenging search algorithms with large and complex state spaces. This thesis focuses on the TopSpin puzzle, a complex permutation puzzle featuring an oval track filled with numbered tokens and a turnstile mechanism that reverses a fixed-length segment of tokens. As TopSpin provides us with a large state space, it is necessary to use efficient heuristics to guide heuristic search algorithms efficiently through the state space. We will introduce four heuristics, custom-tailored to the structure of TopSpin and evaluate them both theoretically and experimentally.

Table of Contents

1
Introduction

Most people have encountered puzzles at some point in their lives, with the Rubik's Cube being one of the most iconic examples. Solving puzzles in general is entertaining, but they can also be frustrating as it takes a lot of creativity, logic and patience to solve them. Beyond being entertainment, they also serve as a good benchmark in the field of artificial intelligence, as they provide good problem settings that can be solved with search algorithms. One of the more complex puzzles is TopSpin [8], a puzzle that consists of a circular track that contains numbered tokens, with a turnstile that can reverse the order of a fixed-length window. In the original problem of TopSpin we have 20 tokens with a reversal window of size 4, resulting in a state space that contains 20! states, which is complex and poses computational challenges. Solving a state space of that size with uninformed search algorithms like Breadth-First Search is infeasible, despite the optimality guarantee it provides.

For a complex puzzle like TopSpin the use of heuristic search algorithms is more efficient and can help reduce the state exploration during search significantly. Unlike uninformed search algorithms, heuristic search makes use of heuristic functions to estimate how far a given state is from the goal. These heuristic functions guide the search algorithm into more promising regions of the state space. The choice and design of heuristics is crucial as a poorly designed heuristic can decrease performance and may lead to suboptimal solution paths.

Most of the heuristics designed for TopSpin are based on Pattern Databases [13], but we want to explore other approaches. We propose four different heuristics, custom-tailored to the structure of TopSpin. Before discussing them individually, we first introduce the formal foundations necessary to understand heuristics and heuristic search. Then, we present the four heuristics: gap, distance-based, abstraction, and the breakpoint graph heuristic. Finally, we compare their performance through a series of experiments and summarize the insights gained.

2
Background

This chapter provides the necessary fundamentals to understand heuristic search in the context of TopSpin. This includes formal definitions of state spaces, heuristics, an explanation of the TopSpin puzzle and an overview of the search algorithms we will use to solve TopSpin.

2.1 State Spaces

State spaces are crucial in describing the problem environment, as they provide us with all relevant aspects of a given problem. They not only represent the current state, but also formalize key elements such as actions, the transitions between actions and states and the definition of goal states. This formalization enables us to use search algorithms to systematically solve them.

Definition 1 (State Space)**.** *Formally a state space is defined as a 6-Tuple* $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$ *where:*

- S*: nonempty set of all possible states in* $\mathcal{S}$
- A*: set of all possible actions*
- $cost$*: cost function that maps the action to nonnegative number*
- T*: deterministic state transition function, which maps a state and an action to its successor state.*
- s_I*: initial state* $s_I \in S$
- S_G*: the nonempty set of goal states* $S_G \subseteq S$

Note that technically the set of actions A is allowed to be empty, but practically this never occurs as this results in having no transitions in the state space, which is meaningless. S and S_G are not allowed to be empty, as without states or goal states the state space is not solvable.

2.2 Heuristics

In order to effectively compare different states in the TopSpin puzzle, we require a metric that estimates how close a given state is to the goal. The comparison of different states with heuristics is crucial, because for certain problems with large state spaces it is infeasible to just search through the entire state space. This introduces the concept of heuristic functions, estimate the cost to reach the goal from a given state and guides the search algorithm towards more promising regions of the state space.

Definition 2 (Heuristic)**.** *Let S be the set of states from state space $\mathcal{S}$, then:*

$$h : S \to \mathbb{R}_0^+ \cup \{\infty\} \tag{2.1}$$

The heuristic function maps each state to a nonnegative number or to infinity for unsolvable states.

The design and choice of such heuristics significantly impact the efficiency of heuristic search methods such as A* or Greedy Best-First Search, the latter relying exclusively on heuristic evaluation. These heuristics have a few properties, that give us an idea on how optimal they will be, when used with a search algorithm. Specifically we will consider these properties: **goal-awareness**, **safety**, **consistency** and **admissibility**.

Definition 3 (Properties of Heuristics [6])**.** *Let $\mathcal{S}$ be a state space with S being the set of states and S_G being the set of goal states. The following definitions formalize key properties of heuristics commonly considered:*

perfect heuristic h^**: The perfect heuristic h^* maps the state to the true distance to the goal state.*

goal-aware*: A heuristic is goal-aware if the heuristic value of a goal state is zero*

$$h(s) = 0 \quad \textit{for all } s \in S_G \tag{2.2}$$

safe*: A heuristic is safe if it does not incorrectly assume a state is unsolvable*

$$h^*(s) = \infty \quad \textit{for all } s \in S \textit{ with } h(s) = \infty \tag{2.3}$$

consistent*: A heuristic is consistent if the heuristic value never decreases by more than the cost of the action, when moving from a state to its successor state.*

$$h(s) \leq cost(a) + h(s') \quad \textit{for all transitions } s \xrightarrow{a} s' \tag{2.4}$$

admissible*: A heuristic is admissible if it never overestimates the cost to reach a goal state.*

$$h(s) \leq h^*(s) \quad \textit{for all } s \in S \tag{2.5}$$

Theorem 1 (Relations Between Heuristic Properties)**.** Let h be a heuristic function defined on a state space. Then the following implications hold:

$$\text{If } h \text{ is consistent and goal-aware, then } h \text{ is admissible.} \tag{2.6}$$

$$\text{If } h \text{ is admissible, then } h \text{ is safe and goal-aware.} \tag{2.7}$$

2.3 TopSpin Puzzle

TopSpin [8] is a circular puzzle that was invented by Ferdinand Lammertink and patented in 1989. The original physical puzzle consists of an oval track, that holds 20 uniquely numbered tokens, labeled from 1 to 20. Attached to this oval track is a reversible window, also referred to as a turnstile of fixed-length 4, which reverses the order of four adjacent tokens. This puzzle can be generalized into any combination of (N, k), where $N \in \mathbb{N}^+$ is the number of tokens and $k \in \mathbb{N}$ is the size of the turnstile, with $2 \leq k \leq N$, thus the original puzzle would be a (20, 4)-TopSpin instance. The objective of TopSpin is to bring any arbitrarily shuffled sequence of tokens into a configuration, where these tokens are sorted in ascending order. This can be achieved with two fundamental operations. To visualize these operations we consider an example TopSpin(12, 4) state:

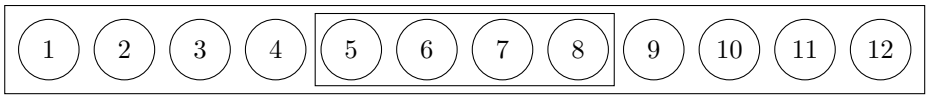


Figure 2.1: Example state for a (12, 4)-TopSpin instance

Rotation: The tokens can be rotated around the oval track by an arbitrary amount, either to the left or to the right. The following example shows a rotation on the initial state shown in Figure 2.1 by four to the right.

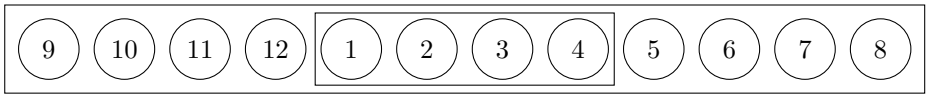


Figure 2.2: Rotate tokens four to the right

Reverse: The four adjacent nodes in the turnstile can be reversed. In the example shown in Figure 2.1, the turnstile spans positions 5 to 8, containing the elements 5, 6, 7, 8. Applying the reverse action to this window inverts the order of these elements, resulting in 8, 7, 6, 5. The rest of the state remains unchanged.

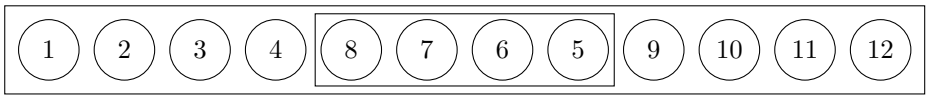


Figure 2.3: Rotate the turnstile

Even though it might seem like a relatively simple puzzle, as it only has two different actions one can take (rotate and reverse), it is quite complex as it has 20! states in the original problem or in general N!. In our formal state space, we define only the reversal action, as rotation can be simulated by shifting the reversal window's starting index. The reversal operation includes wraparound, meaning the window wraps to the beginning if it exceeds the end of the sequence.

Definition 4 (TopSpin)**.** A (N, k)-TopSpin puzzle can be formally described as a state space $\mathcal{S} = \langle S, A, cost, T, s_I, S_G \rangle$, where:

- S: All valid permutations of $\{1, 2, \ldots, n-1, n\}$
- A: Reversal of k elements starting at index i, with wraparound (mod n):

  $$A = \{\text{reverse}(i) \mid i \in \{0, 1, \ldots, n-1\}\}$$

  Precondition: None, as any reversal from position i is allowed.

  Effect: Given a state $s = \langle s_0, s_1, \ldots, s_{n-1} \rangle$, the effect of applying reverse(i) is to reverse the k-length subsequence starting at i modulo n. This results in the successor state s':

  $$s'_{(i+j) \bmod n} = s_{(i+k-1-j) \bmod n}, \quad \text{for } j = 0, 1, \ldots, k-1$$

- cost:

  $$\text{cost}(a) = 1, \quad \forall a \in A$$

- T: A deterministic function that applies an action to a state

  $$T(s, a) = s', \quad \text{where } s' \text{ is the result of applying } a \text{ to } s$$

- s_I: Any valid permutation from $\mathcal{S}$
- S_G: The set of goal states, defined as all rotations of the identity permutation {1, 2, ..., N}

2.4 Search Algorithms

Now that we have formally defined TopSpin as a state space, we can apply search algorithms to solve it. There are various algorithms available, some more efficient than others, but our focus is on those that leverage heuristics in their evaluation function. For example, Breadth-First Search (BFS) is cost-optimal. We call a search algorithm cost-optimal whenever it is guaranteed to find a least-cost solution if one exists. However, due to the large size of the state space and a branching factor of N, where N is the number of tokens and also the number of distinct actions applicable, BFS is computationally infeasible for our purposes.

The ones we are interested in are the heuristic search algorithms. Instead of blindly searching through the state space, they make use of an evaluation function to decide which states to expand next. We will denote this evaluation function as $f(s)$, where s is a state in the state space.

2.4.1 A*

A* [10] is the most common heuristic search algorithm. While algorithms like Uniform Cost Search select the nodes to expand solely by the current path cost-the g value-from the initial state to the expanded state, A* enhances this search by using the evaluation function $f(n) = g(n) + h(n)$, which not only considers the current cost g but also the heuristic value of the expanded state. As a result, the algorithm avoids exploring paths that are unlikely to lead to a solution.

The algorithm begins by inserting the initial state into a priority queue (also called open list) that stores the states as nodes with its priority being the f value. At each step it takes the node with the lowest f value out of the priority queue, checks if the state is a goal state, and if not, expands it. Expanding a node means generating all possible successor states reachable from the current state by applying all applicable actions defined in the state space. For each successor it then calculates the f value and puts the successor into the open list. To avoid revisiting explored states, A* also maintains a closed list of visited nodes. This process continues until either a goal is found or the open list is empty. An empty open list indicates that the algorithm found no solution.

A* is complete, meaning it finds a solution when one exists in the state space. It is also cost-optimal given that the heuristic is consistent. For heuristics that are not consistent there is a variant of A* that re-expands states, which is cost-optimal when used with an admissible heuristic.

The main drawback of A* is its high memory usage. Its space complexity is $\mathcal{O}(b^d)$, where b is the branching factor (i.e., the number of successors per state), and d is the depth of the state. This exponential space requirement can make A* impractical for large search spaces.

The following pseudo code outlines the A* algorithm, assuming a search variant with node reopening:

Algorithm 1 Pseudo code for A*

```
1: openList ← new MinHeap()                    ▷ priority by f-values, where f = g + h
2: if h(initialState()) < ∞ then
3:     openList.insert(rootNode())
4: closedList ← new HashMap()
5:
6: while not openList.isEmpty() do
7:     next ← openList.pop()
8:     if next.state ∉ closedList or g(next) < closedList[next.state] then
9:         closedList[next.state] ← g(next)
10:        if isGoalState(next) then
11:            return getSolutionPath(next)
12:        for succ in successors(next.state) do
13:            if h(succ.state) < ∞ then
14:                nextP ← node(next, succ.action, succ.state)
15:                openList.insert(nextP)
16: return unsolvable
```

2.4.2 IDA*

Certain problems, such as the bigger instances of TopSpin for example, require a substantial amount of memory to be solved with A*. In such harder instances, the algorithm may become impractical, as it can exhaust memory quite quickly and in consequence it fails to produce a solution. These problems require the use of Iterative Deepening A* (IDA*), which keeps the idea of evaluating on the function $f(n) = g(n) + h(n)$, but it does not keep all reached states in memory. This is more memory efficient, but comes at the cost of revisiting some states that have already been reached before.

The search algorithm performs a series of depth-first searches limited by an increasing threshold on the f-value. The first iteration uses the f-value of the initial state as the threshold, and each subsequent iteration uses the smallest f-value encountered that exceeded the previous threshold.

IDA* is complete and it is guaranteed to be cost-optimal when used with an admissible heuristic. The following pseudo code outlines the IDA* algorithm:

Algorithm 2 Pseudo code for IDA*

```
1: f_limit ← h(initialState())
2: while f_limit ≠ ∞ do
3:    ⟨f_limit, solution⟩ ← search(initialState(), 0, f_limit)
4:    if solution ≠ none then
5:       return solution
6: return unsolvable
```

Algorithm 3 Recursive search for IDA*

```
1: function search(s, g, f_limit)
2: if g + h(s) > f_limit then
3:    return ⟨g + h(s), none⟩
4: if isGoalState(s) then
5:    return ⟨none, ⟨⟩⟩
6: newLimit ← ∞
7: for ⟨a, s'⟩ ∈ successors(s) do
8:    ⟨childLimit, solution⟩ ← search(s', g + cost(a), f_limit)
9:    if solution ≠ none then
10:      solution.push_front(a)
11:      return ⟨none, solution⟩
12:   newLimit ← min(newLimit, childLimit)
13: return ⟨newLimit, none⟩
```

3
Heuristics for TopSpin

In this chapter we will describe the four custom-tailored heuristics we propose for TopSpin. We start with the gap heuristic that originally is meant for the pancake puzzle, then we take a look at a distance based heuristic that takes inspiration from the manhattan distance, after that we take a look at abstractions and try a different approach than PDBs and finally we introduce the graph based heuristic, that builds upon the idea of breakpoint graphs.

3.1 Gap Heuristic

The first heuristic we introduce is the gap heuristic, originally proposed by Helmert [5] for the pancake problem. In the pancake problem, the goal is to sort a stack of pancakes (typically represented as a sequence of numbers $\{1, ..., n+1\}$) in ascending order, by applying a reversal of the first k elements on the pancake stack. The idea of this heuristic is to count the positions that are not in consecutive order.

TopSpin shares some similarities with the pancake puzzle, as both involve transforming a sequence of numbers into ascending order. However there are some differences that have to be considered to make the heuristic fit to TopSpin:

1. **Reversals**: In TopSpin we have a fixed-length reversal of k anywhere in the permutation, instead of only reversing any amount from the top of the stack. This has a significant impact on the heuristic as each reversal can eliminate up to two gaps in the permutation, contrary to only one gap in the pancake problem. If this isn't considered the heuristic would be inconsistent.

2. **Circular Structure**: The permutation in TopSpin is circular meaning the heuristic also has to consider the comparison of the first and last element.

3. **Special Case**: The heuristic has to include that the biggest and smallest element (in that order) are not considered a gap.

Definition 5 (h_{gap})**.** *Let $s = <s_1, ..., s_N>$ be a TopSpin state and $s' = <s_1, ..., s_N, s_{N+1}>$, with $s_{N+1} = s_1$ be an extended state that appends the first element to handle the wraparound,*

then we can formally define the gap heuristic for TopSpin as:

$$h_{gap}(s) = \left\lceil \frac{\left|\left\{i \mid i \in \{1,\ldots,N\},\ |s'_i - s'_{i+1}| > 1 \wedge \left\{s'_i, s'_{i+1}\right\} \neq \{1, N\}\right\}\right|}{2} \right\rceil \tag{3.1}$$

Example 3.1.1. Consider a random permutation of a TopSpin(12, 4) instance and the corresponding extended state:

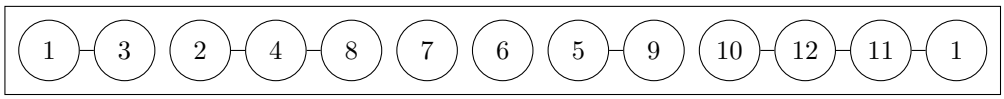


Figure 3.1: Extended state with visualized gaps

The black lines in Figure 3.1 between the tokens visualize the gaps in the permutation. Applying the heuristic to this state gives us a heuristic value of $h_{gap}(s) = \lceil \frac{7}{2} \rceil = 4$.

3.1.1 Properties of h_{gap}

As discussed in the chapter on heuristics, certain properties must be satisfied for a heuristic to be considered cost-optimal. h_{gap} fulfills all properties mentioned in Definition 3.

Goal-awareness: Any goal state corresponds to a sorted permutation and sorted permutations contain no gaps, which implies that the heuristic value assigned to all goal states is zero. Therefore h_{gap} is goal-aware (2.2).

Consistency: To demonstrate this, we can look at the effect of a reversal operation on the heuristic value. A reversal can produce one of the following changes:

- h increases by exactly one (actions that create either one or two gaps)
- h decreases by exactly one (actions that remove one or two gaps)
- h stays the same (actions that do not remove or create any gaps)

In all cases, the inequality $h(s) \leq 1 + h(s')$ is respected and we can say that h_{gap} is consistent (2.4).

Admissibility: We prove that h_{gap} is admissible by using the implication from Equation 2.6, which states that a heuristic is admissible if it is both goal-aware and consistent. Since h_{gap} satisfies both properties, it is also admissible (2.5).

Safeness: The heuristic is safe as it fulfills the implication from Equation 2.7, which states that every admissible heuristic is safe (2.3).

The heuristic h_{gap} fulfills all properties and is therefore guaranteed to find optimal solutions, when used with A*.

3.2 Manhattan Inspired Distance Heuristic

The next heuristic takes inspiration from one of the more well-known heuristics: the Manhattan distance. Russell and Norvig [10] identify it as an admissible heuristic for the 8-puzzle, providing a good foundation. The idea of the Manhattan distance is to sum the vertical and horizontal distances between each tile and its goal position. Formally the Manhattan distance [11] is defined as:

$$\sum_{i=0}^{n-1} \left(|x_{g_i} - x_i| + |y_{g_i} - y_i| \right) \tag{3.2}$$

for all tiles in the state, where n is the number of tiles, (x_i, y_i) denotes the position of the i-th tile, and (x_{g_i}, y_{g_i}) denotes its goal position.

This is a heuristic for a two-dimensional problem, which is not directly applicable on TopSpin as it has only one-dimensional states. We still want to take a look at a distance-based heuristic, as this can give us a good estimate on how far away the state is from its goal. However there are a few changes that have to be applied to make it fit to the circular one-dimensional structure:

- As TopSpin is circular and goes into both directions, the distance is not as trivial as in the original heuristic. Instead we have to look at both distances (to the left and to the right including wraparound) a token has to its goal position.
- Reversing a section has a more significant impact on the heuristic, meaning we have to divide the summed up distances by the maximum effect of the reversal. A single reversal of size k effects the distances of all k elements in the turnstile. However the impact depends on the position: the outermost elements can decrease their distance by at most $k-1$, and this maximum reduction decreases by two for each step inwards. This maximum effect any reversal of size k can have on the heuristic is exactly:

$$\Delta max = \sum_{j=0}^{k-1} |2j - (k-1)| \tag{3.3}$$

- For each state there are N different rotations, that all have a different sum of distances. This has to be considered by calculating the sum for each possible rotation and taking the minimum of those.

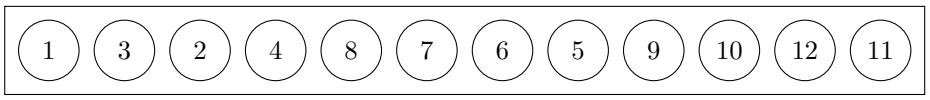
Putting all of those together, gives us a distance based heuristic tailored for TopSpin. The heuristic computes the smallest sum of the minimal distances of all tokens from their respective goal positions, and divides this total by the maximum possible reduction in distance that a single reversal of length k can achieve.

Definition 6 (h_{distance})**.** *Let $s = \langle s_1, ..., s_N \rangle$ be a TopSpin state. The rotation is represented by an offset $r \in \{0, \ldots, N-1\}$ which shifts the current state r positions to the left.*

$$dist(s, r) = \sum_{i=0}^{N-1} \min \left((i - (s_{(i+r) \bmod N} - 1)) \bmod N, ((s_{(i+r) \bmod N} - 1) - i) \bmod N \right) \tag{3.4}$$

$$h_{manhattan}(s) = \left\lceil \frac{\min\limits_{0 \le r < N} dist(s, r)}{\Delta max(k)} \right\rceil \tag{3.5}$$

Example 3.2.1. Consider a random permutation of a TopSpin(12, 4) instance:



Using Equation 3.3, we compute the maximum possible reduction in total distance that a single reversal of size $k = 4$ can achieve. This gives $\Delta max = 3 + 1 + 1 + 3 = 8$. Next we calculate all distances dist(r) for each rotation $r \in \{0, \ldots, 11\}$ using Equation 3.4:

r	Rotated State	dist(r)
0	$\langle 1, 3, 2, 4, 8, 7, 6, 5, 9, 10, 12, 11 \rangle$	12
1	$\langle 3, 2, 4, 8, 7, 6, 5, 9, 10, 12, 11, 1 \rangle$	16
2	$\langle 2, 4, 8, 7, 6, 5, 9, 10, 12, 11, 1, 3 \rangle$	26
3	$\langle 4, 8, 7, 6, 5, 9, 10, 12, 11, 1, 3, 2 \rangle$	36
4	$\langle 8, 7, 6, 5, 9, 10, 12, 11, 1, 3, 2, 4 \rangle$	46
5	$\langle 7, 6, 5, 9, 10, 12, 11, 1, 3, 2, 4, 8 \rangle$	56
6	$\langle 6, 5, 9, 10, 12, 11, 1, 3, 2, 4, 8, 7 \rangle$	60
7	$\langle 5, 9, 10, 12, 11, 1, 3, 2, 4, 8, 7, 6 \rangle$	56
8	$\langle 9, 10, 12, 11, 1, 3, 2, 4, 8, 7, 6, 5 \rangle$	46
9	$\langle 10, 12, 11, 1, 3, 2, 4, 8, 7, 6, 5, 9 \rangle$	36
10	$\langle 12, 11, 1, 3, 2, 4, 8, 7, 6, 5, 9, 10 \rangle$	26
11	$\langle 11, 1, 3, 2, 4, 8, 7, 6, 5, 9, 10, 12 \rangle$	16

Table 3.1: Distance values for all rotations of the TopSpin(12, 4) example state.

The minimal distance across all rotations is 12, resulting in $h_{\text{distance}}(s) = \lceil \frac{12}{8} \rceil = 2$.

Properties of h_{distance}

The heuristic h_{distance} fulfills all properties described in Definition 3.

Goal-awareness: In all possible goal states, h_{distance} will have a value of 0 as all tokens have distance 0 i.e. they are in their goal position, which makes h_{distance} goal-aware (2.2).

Consistency: Each reversal move in TopSpin can reduce the total sum of distances by at most the factor Δmax (see Equation 3.3). Since the heuristic h_{distance} divides the summed distances by Δmax, each reversal can decrease the heuristic value by at most 1. Consequently, for any successor state s' reached from s by a single reversal, the heuristic satisfies $h_{\text{distance}}(s) \leq 1 + h_{\text{distance}}(s')$ and is therefore consistent (2.4).

Admissibility: For admissibility we can again use the implication from Equation 2.6, as we already showed that $h_{\text{manhattan}}$ is both goal-aware and consistent, hence it is also admissible (2.5).

Safeness: The heuristic is admissible, which implies (2.7) that it is also a safe (2.3) heuristic.

As $h_{\text{manhattan}}$ fulfills all given properties, it guarantees the generation of cost-optimal solutions when used with A*.

3.3 Domain Abstraction Heuristic

The next heuristic is based on domain abstractions, where the state is simplified into an abstract state that considers only a subset of the full information. Generally a domain abstraction heuristic partitions the original state into equivalence classes. These equivalence classes contain fewer values and are therefore easier to handle. The key idea here is that similar or irrelevant tokens can be treated as the same, which reduces the complexity. The heuristic can then be calculated by solving the abstract version of the problem, which is simpler and computationally cheaper to do in comparison to the original state.

Definition 7 (Domain Abstraction [7])**.** *Let $s = \langle s_1, \ldots, s_N \rangle$ be a TopSpin state, where each token s_i takes a value from {1, ..., N}. A* domain abstraction *is a tuple $\alpha = \langle \alpha_1, \ldots, \alpha_N \rangle$, where each α_i is a* component abstraction*, mapping individual tokens to abstract values:*

$$\alpha_i : \textit{values} \rightarrow \textit{equivalence class}. \tag{3.6}$$

We denote $\alpha_i(s_i)$ as the equivalence class (abstract value) of s_i under some fixed partitioning. Applying a domain abstraction to a state s yields an abstract state:

$$\alpha(s) = \alpha_1(s_1) \times \cdots \times \alpha_N(s_N). \tag{3.7}$$

To illustrate how Definition 7 gives us an abstract state we can consider the state $s = \langle 8, 5, 6, 4, 2, 1, 7, 3 \rangle$. We define a domain abstraction $\alpha = \langle \alpha_1, \ldots, \alpha_8 \rangle$, where each component abstraction α_i maps the token values to "5" if they are greater than 4 and otherwise keeps the value as is:

$$\alpha_i(s_i) = \begin{cases} \mathrm{s_i}, & \text{if } x \leq 4 \\ 5, & \text{if } x > 4 \end{cases}$$

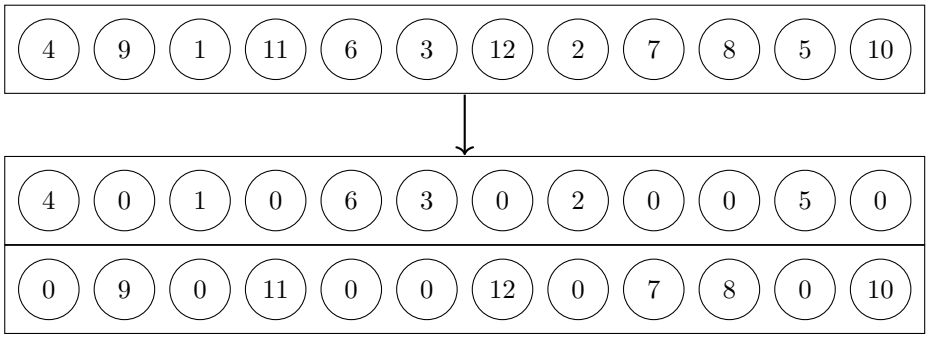
Applying the domain abstraction s, yields the abstract state $\alpha(s) = \langle 5, 5, 5, 4, 2, 1, 5, 3 \rangle$.

3.3.1 Approach 1: Maximizing Over Multiple Abstract States

To represent these abstraction mappings, we will adopt and extend the notation by Yang et al.[13]. They denote their abstractions with tuples written as a_1–a_2–...–a_M indicating a set containing M abstractions. The abstraction a_1, then contains tokens 1 to a_1, the abstraction a_2 tokens $(a_1 + 1)$ to $(a_1 + a_2)$ and so on. Tokens not present in the abstraction are transformed in so called "don't care" values (we will represent these as 0).

As we want to look at different mappings, other than just having adjacent tokens (e.g. $\langle 1, 2, 3, 4 \rangle$), we need to extend their representation. The original a_1–a_2–...–a_M abstraction, is now called a a_1–a_2–...–a_M − Group abstraction. We additionally define the a_1–a_2–...–a_M − Distance abstraction, which instead encodes the relative distances between selected tokens. For example, the a_1-a_2-Distance abstraction distinguishes odd and even tokens in the state.

Example 3.3.1. Consider a random permutation of a TopSpin(12, 4) instance, we can create a 6-6-Group abstraction that results in these abstract states:



One way to leverage these abstractions is the use of Pattern Databases (PDBs)[3]. A PDB is a abstraction heuristic that stores the optimal solution cost of an abstract state in a precomputed database. The solution cost is used as heuristic and this has the advantage that the heuristic value of a given state can be looked up in the runtime, without having to compute it. This heuristic has been explored by Yang et al. [13] on a number of different problems and they have shown that some abstractions are effective for TopSpin.

Instead of precomputing and storing heuristic values in a PDB based on projection abstractions[7], this approach computes the heuristic values during the search by applying a domain abstraction and solving the abstract states with Breadth-First Search (BFS). We chose domain abstractions instead of projections as this allows us to flexibly chose the abstraction mapping. While this is not important for this approach, it will be necessary for the second approach we will describe later. The BFS, just like the values stored in the PDB, provides us the optimal solution length, which serves as a useful estimate. While this approach may be less efficient in terms of runtime, since each heuristic value requires solving the abstraction during the search, it has the advantage of being less memory-intensive, as it does not require storing precomputed values externally. This approach is not fundamentally different to PDBs, but we still want to compare it, as this trade-off might still be interesting to look at for smaller instances especially.

When using multiple abstractions, it is often better to combine their individual heuristic values, as this results in a stronger heuristic. A common approach is to sum the solution cost from each abstraction, which is effective when the abstractions and especially their actions are disjoint. However, this is not applicable for TopSpin, because the actions applied when solving the abstractions overlap. In other words, a single reversal may affect tokens that are not considered in another abstraction (e.g. reversing tokens 4, 0, 1, 0 in the Example 3.3.1). This might lead to over-counting some actions, when adding the individual solution costs, which can lead to inadmissibility. To avoid this, we instead take the maximum of all abstract solution costs.

Definition 8 ($h_{abstract}$)**.** *Let $s = \langle s_1, ..., s_N \rangle$ be a TopSpin state and $a_1 - ... - a_M$ the domain abstraction tuple (any of the two types) of s, then:*

$$h_{abstract}(s, a_1\text{-}\ldots\text{-}a_M) = \max_{a \in a_1\text{-}\ldots\text{-}a_M} bfs\_solutionLength(a) \tag{3.8}$$

Properties of h_{abstract}

The heuristic h_{abstract} fulfills all properties described in Definition 3.

Goal-awareness: As h_{abstract} is the solution length of the abstract state, it implies that the heuristic is goal-aware (2.2), because the solution length of a goal state is 0. This holds because applying the domain abstraction to a goal state yields the abstract goal state and the BFS returns a solution length of 0.

Consistency: As described before, we take the maximum solution cost over all abstractions. For every successor state s' of a given state s, the heuristic always satisfies the inequality from Equation 2.4, since any reversal reduces the solution cost of any abstract state by at most one. This means that the individual abstractions are consistent, as their solution length does never change by more than one, because any action can either increase it by one, does not change the solution length or decrease it by one. Since h_{abstract} is defined as the maximum over these consistent abstractions, it is also consistent (2.4).

Admissibility: The way the heuristic is built automatically implies that it is admissible. The reason for that is that the solution for an abstract state can never be more expensive than the solution for the original state, as it is a much simpler problem that will never require more actions to solve. In other words, the solution cost of any abstraction is always less than or equal to the true solution cost. Since each individual abstraction yields an admissible heuristic value and it is taking the maximum, it never overestimates the actual distance. Therefore, the heuristic is admissible (2.5).

Safeness: The heuristic is admissible, which implies (2.7) that it is also a safe (2.3) heuristic.

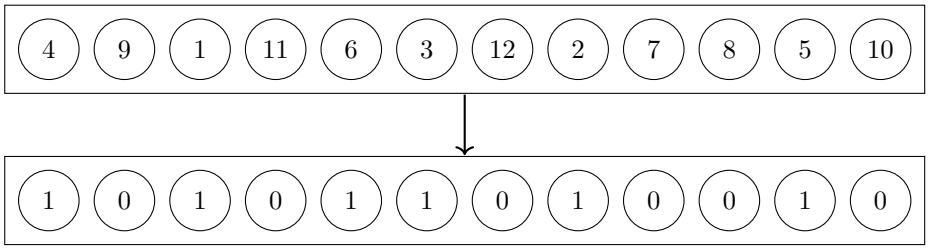
As it fulfills all given properties, it guarantees us cost-optimal solutions when using it with A^* or IDA^*.

3.3.2 Approach 2: Single State Abstraction

Another way to apply the abstraction mapping is to directly transform the original state into a single abstract state, rather than preserving certain values and replacing some with a "don't care" value. This approach has the advantage that we have a single abstraction and therefore do not have to calculate the solution for each individual abstraction in the abstract state. The trade-off is that we have a greater loss of information, as in contrary to before each group from the mapping is represented as the same value.

Any abstraction of this approach will be denoted as an M-Type abstraction, where M indicates the number of groups in the abstract state, and Type $\in$ GroupS, DistanceS specifies the type of abstraction applied. Just like before "Group" refers to elements that are close in value (e.g. 1, 2, 3 are seen as equivalent) and "Distance" refers to values that have a distance of M between them (e.g. 1, 3, 5 are seen as equivalent). The "S" appended at the end of the type signifies that this is a single-state abstraction and exists to distinguish this approach from the earlier method.

Example 3.3.2. Consider a random permutation of a TopSpin(12, 4) instance, we can create a 2-GroupS abstraction that results in this abstract state:



As this approach does not involve multiple abstractions, there is no need to combine them and the heuristic is the solution of the abstract state.

Definition 9 ($h_{abstractS}$)**.** *Let $s = \langle s_1, ..., s_N \rangle$ be a TopSpin state and a_s the abstract state derived by applying a 2-GroupS abstraction, then:*

$$h_{abstractS}(s, a_s) = bfs\_solutionLength(a_s) \tag{3.9}$$

Properties of $h_{\text{abstractS}}$

The heuristic $h_{\text{abstractS}}$ fulfills all properties described in Definition 3.

Goal-awareness: As $h_{\text{abstractS}}$ is the solution length of the abstract state, it implies that the heuristic is goal-aware, because the solution length of a goal state is 0.

Consistency: This approach behaves similarly to the other abstraction method described earlier. Any reversal of size k can change the heuristic value by at most one. This is because the actual solution cost changes by no more than one step, even with this simpler abstraction.

Admissibility: For h_{abstract} we already described that we are solving a much simpler version of the original state and the solution of this simplified state can never be more expensive as the true distance the original state has to its goal. Therefore the heuristic value of this approach will never exceed the true cost, making it admissible (2.5).

Safeness: Using Implication 2.7 we can derive that $h_{\text{abstractS}}$ is also safe, as every admissible heuristic is also safe (2.3).

Just like the other approach this fulfills all given properties and it therefore guarantees cost-optimal solutions, when using it with A* or IDA*.

3.4 Breakpoint Graph Heuristic

The final heuristic we introduce is the breakpoint graph heuristic $h_{\text{breakpoint}}$. The idea to use this as a heuristic for TopSpin was inspired by a work of Hannenhalli and Pevzner [4], who used it in an algorithm for sorting signed permutations. Although TopSpin is not a signed permutation problem, the concept is still applicable, as it also is a permutation problem.

The heuristic builds upon the idea of the gap heuristic, but extends it by considering the relationships between tokens, rather than simply counting discontinuities. To do this, we first extend the initial state by prepending 0 and appending $N + 1$.

Definition 10 (Breakpoint Graph)**.** *Let $s = <s_1, ..., s_N>$ be a TopSpin state and $s\_e = <s_0, s_1, ..., s_N, s_{N+1}>$, with $s_0 = 0$ and $s_{N+1} = N + 1$ be the extended state, then we can construct the breakpoint graph by applying the following rules:*

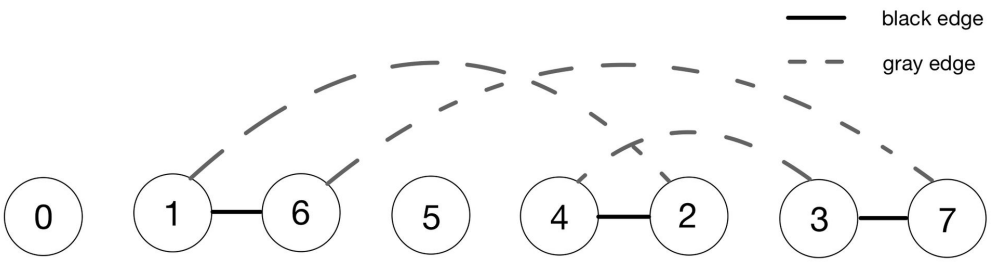
1. *We connect two tokens s_i, s_{i+1} with a black edge when:*

$$| s_i - s_{i+1} | \ > \ 1, \quad i \in \{0, \ldots, N\} \tag{3.10}$$

   We say that there is a breakpoint between s_i and s_{i+1} iff they are connected with a black edge in the breakpoint graph.

2. *Whenever a token s_i with $i \in \{0, \ldots, N+1\}$ is contained in any breakpoint, we connect s_i and its actual neighbors ($s_i - 1$ and $s_i + 1$) that are not already adjacent with a gray edge.*

Example 3.4.1. Consider the (6, 4)-TopSpin state $\langle 1, 6, 5, 4, 2, 3 \rangle$. Applying the rules from Defintion 10 results in the following breakpoint graph:



The first component of the heuristic $h_{\text{breakpoint}}$ is the number of breakpoints (black edges) in the breakpoint graph, which is very similar to the gaps previously mentioned in h_{gap}.

The second component is the maximum number of alternating cycles in the breakpoint graph. An alternating cycle is defined as a cycle in the graph where edges strictly alternate between black and gray, starting and ending at the same vertex. Finding this number can be done with a maximum alternating cycle decomposition (MAX-ACD). However a MAX-ACD is not trivial and has been proven to be NP-Hard [2]. What this means for the heuristic is that for larger instances this would be impractical as it takes too much time to calculate the exact decomposition. Instead we will rely on an approximation for the MAX-ACD that took inspiration from the approach by Pinheiro et al. [9], but this will be explained further in the Implementation chapter.

These two components give us a lower bound [4] for the reversal distance of a given permutation:

$$d(\pi) \geq b(\pi) - c(\pi) \tag{3.11}$$

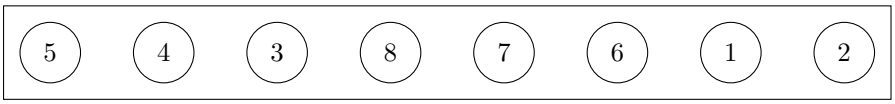
where $d(\pi)$ is the reversal distance to the goal, $b(\pi)$ is the number of breakpoints (black edges) in the breakpoint graph of π and $c(\pi)$ is the maximum number of alternating cycles.

To apply this to TopSpin we have to use a slightly different approach, as we have to consider the circular structure and the fact that there are different rotations for each state. The underlying problem is that with TopSpins structure The easiest way to do this is to take the state and look at the rotations, where 1 is in the first position. This allows us to use the heuristic without changing the logic, as without any significant change the heuristic would not see a goal state as such.

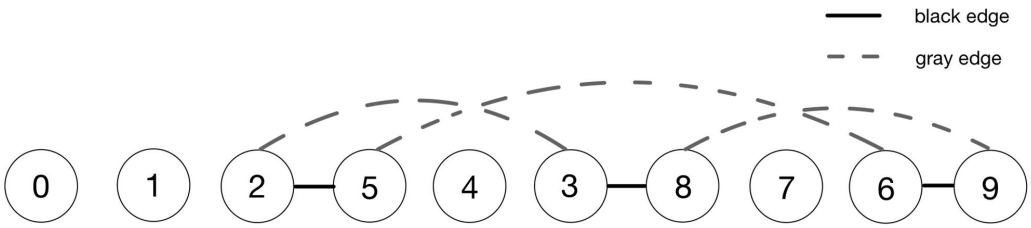
Definition 11 ($h_{\text{breakpoint}}$)**.** *Let $s = \langle s_1, ..., s_N \rangle$ be a TopSpin state, s_r be the rotation described above. Then we can apply the reversal distance of Equation 3.11 to define $h_{breakpoint}$ as:*

$$h_{breakpoint}(s, s_r) = b(s_r) - c_{approx}(s_r) \tag{3.12}$$

Example 3.4.2. Consider a random permutation of a TopSpin(8, 4):



We then rotate the state so that token 1 is the first element of the state. From the rotated state we can then create the corresponding breakpoint graph:



The breakpoint graph of state s has three breakpoints and the decomposition gives us one cycle $(2 \to 5 \to 6 \to 9 \to 8 \to 3 \to 2)$. So for state s the heuristic value is $h_{\text{breakpoint}}(s) = 3 - 1 = 2$.

Properties of $h_{\text{breakpoint}}$

As $h_{\text{breakpoint}}$ is based on an approximation of the MAX-ACD, it only fulfills the conditions for **goal-awareness** and **safeness**, while **admissibility** and **consistency** are not guaranteed.

Goal-awareness: In any goal state of TopSpin the breakpoint graph has no black edges, which also implies that there are no gray edges. This means the heuristic is 0 for all goal states of TopSpin, hence it is goal-aware (2.2)

Consistency: Bafna and Pevzner [1] showed that the distance in Equation 3.11 can not reduce by more than one for any reversal, which would make this heuristic theoretically consistent. However as we are working with an approximation for the MAX-ACD it is not guaranteed that the heuristic is consistent. For some states the approximation might miss a cycle and this might not be the case for the successor state. In those cases the

heuristic decreases by more than one, which ends up not satisfying the triangle inequality from Equation 2.4, hence consistency is not guaranteed.

Admissibility: While Equation 3.11 provides a lower bound for reversal problems, using an approximation does not guarantee the admissibility. If the approximation returns less cycles than the exact decomposition, we exceed this lower bound, hence overestimating the distance. In such cases, the heuristic is no longer admissible (2.5).

Safeness: In the worst case the heuristic does not find any cycles at all and the heuristic is the number of breakpoints in the graph. So it can never identify a state as unsolvable and is therefore safe (2.3).

As only goal-awareness and safeness are given, the heuristic does not guarantee cost-optimal solutions, when using it with A^* search for example.

4

Implementation

The entire implementation was done in `C++20` and the code can be accessed via this GitHub Repository [1]. All the source files used for experimenting and all the heuristics can be found in the **implementation** folder. This section describes all the components and some of the decisions made for the implementation.

TopSpinStateSpace.cpp: Contains the entire state space representation, just like described in Definition 4. The states themselves are implemented with vectors, as they allow easy memory management and are flexible in regard to the problem size N. In Section 2.3 we summarized the actions into a single reverse operation. This is implemented with a variable starting position of the turnstile, where the reversal starts at position i and reverses a window of size k. Goal states are checked with a simple iteration over the state, where we use modulo to ensure the wraparound and the special case of element N followed by 1 is handled as well. The transition function is represented through `TopSpinStateActionPairs` that store both the action and the successor state.

Heuristics.cpp: This is where the all the heuristics from Chapter 3 are implemented.

- h_{gap}: Iterates over the state and increments a counter for every consecutive pair, where the absolute distance is bigger than one. The special case 1, N is not considered as gap. The final result is divided by two and rounded up.
- $h_{\text{manhattan}}$: Iterates over all rotations of the given state and sums up the distances described in Section 3.2. The minimal sum is then divided by the factor described in Equation 3.3 and rounded up.
- $h_{\text{abstract}}/h_{\text{abstractC}}$: For both abstraction approaches the implementation defines a general function for both types (group, distance). The individual commands for each abstraction are defined in **TopSpinStateSpace.cpp** (e.g. "twoGroup" is the 6-6-Group abstraction described in Section 3.3).
- $h_{\text{breakpoint}}$: Creates the breakpoint graph and decomposes the graph with an approximation for the MAX-ACD. This is done for the rotation, where 1 is the first token

[1] https://github.com/drehermarco/Heuristics-for-TopSpin

and for the rotation, where N is the last token. The approximation takes a greedy approach inspired by RANDOM approach from the algorithm Pinheiro et al. [9] used. The RANDOM approach takes a random edge from the breakpoint graph and searches for a cycle. Instead of using BFS, this heuristic uses a depth-first approach to ensure a better performance. The trade-off is that the decomposition is worse, but for larger states we can still use the heuristic. To reduce the randomness it repeats the process a few times, which helps with finding a better decomposition.

Abstraction.cpp: Implements the two abstraction approaches described in Section 3.3. The normalize function takes any state and rotates it so that the smallest non-zero element is at the correct position (more specifically at position token - 1). This normalization is necessary for checking the goal states. For the first approach the abstraction function simply iterates over the state and sets all values as 0 that are not in out mapping. The second abstraction approach is created by applying the mapping directly with the `std::transform` function. Both approaches are solved with a generic BFS function, that takes the goal function as input and returns the corresponding solution length. To make the implementation a bit faster and reduce duplicate computation of solution lengths, we used a cache for already computed values. For the approach with multiple abstractions per state, only the successor states that have at least one non-zero value are considered to reduce redundant expansions.

AStarSearch.cpp: Implementation for A* without re-opening nodes. It evaluates nodes with $f = g + h$ and tie-breaking is done in favor of deeper nodes in the search space. The implementation also tracks the amount of expanded nodes and the time spent searching until a solution is found. After compilation it can be started with `./solver` N k m, where N is the amount of tokens, k the size of the turnstile and m the number of random steps done from a goal state to the initial state.

IDAStarSearch.cpp: Implementation for IDA* that starts with the initial threshold $h(s_I)$. Successors are sorted by h value in ascending order (preferring lower h values). The implementation takes some inspiration from the HOG2 [12] implementation and we added a node table to reduce node redundancies, as without it the expanded nodes would explode in number. To ensure that this does not take too much memory we limited the amount of entries. After compilation it can be started with `./solver` N k m, where N is the amount of tokens, k the size of the turnstile and m the number of random steps done from a goal state to the initial state.

5
Experiments

5.1 Setup

All experiments were conducted using our C++ implementation, available in the GitHub repository. Computations were performed on the sciCORE[2] scientific computing center at the University of Basel. Depending on the experiment, we adjusted memory usage and time limits accordingly.

For each experiment, initial states were generated by applying a random walk of length $steps \in \{25, 50, 100, 150\}$ from the goal state $\langle 1, \ldots, N \rangle$. The experiments were evenly distributed across these values to ensure a balanced range of problem difficulties.

For smaller problem sizes ($N \leq 16$), we used A* with varying memory limits depending on the state size. Specifically, we used:

- 6 GB for $N = 10$ and $N = 12$
- 12 GB for $N = 14$
- 24 GB for $N = 16$

We ran 101 instances for $N = 10, 12, 14$ and 51 instances for $N = 16$, due to the increased memory usage and longer runtimes.

For larger problem sizes ($N \geq 16$), memory requirements made A* infeasible, so we used IDA* instead. For each problem size, we tested 21 instances with a time limit of 24 hours per instance.

To evaluate the heuristics, we used the following performance metrics:

- **Average and Median** h: Measures the informativeness of the heuristic at the initial state.
- **Average and Median runtime**: Time (in seconds) required to solve the instance.
- **Average and Median solution length**: Number of moves in the optimal solution.

[2] http://scicore.unibas.ch/

- **Average and Median node expansions**: Measures how efficiently the heuristic guides the search.
- **Success rate**: This measure is used in the IDA* experiments, as some instances could not be solved in the 24 hours.

5.2 Results

Table 5.1 shows the performance of the gap heuristic h_{gap} with A*. While this heuristic is simple and fast, it scales poorly with increasing problem size. Although it solves small instances quickly, node expansions grow rapidly with N, making it impractical for larger problems. This happens because we divide the number of gaps by two, to ensure h_{gap} is both admissible and consistent, which overall makes the heuristic search less informed and leads to more node expansions.

N	k	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes**	**Med Nodes**
10	4	0.277	0.002	6.9	7	588	263
12	4	0.129	0.024	9.2	9	11604	3284
14	4	2.739	0.910	11.4	11	174274	60790
16	4	58.279	26.013	13.7	14	3651111	1745276

Table 5.1: Performance of h_{gap} with A*

The next table shows the performance of $h_{\text{manhattan}}$, which turned out to performed worse than anticipated. It scales incredibly poorly with increasingly larger N, to the point that we could not run A* with the described memory constraints for $N = 16$.

N	k	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes**	**Med Nodes**
10	4	0.127	0.043	6.9	7	7615	3121
12	4	2.685	1.023	9.0	9	94858	38164
14	4	60.033	43.727	11.6	12	1538303	1039649

Table 5.2: Performance of $h_{\text{manhattan}}$ with A*

Table 5.3 shows the results of approach one of the domain abstraction heuristic. Generally we can say that for this approach the group based abstraction type worked better than the distance abstraction type. The results also show that the amount of abstractions play a crucial role in regards to performance, which can also be seen in the analysis Yang et al. did in their paper about additive state space abstractions with PDBs. Generally the higher the amount of abstractions is, the shorter the abstract solutions are, which leads to a less informed heuristic and ultimately to a higher number of node expansions.

N	k	**Abstraction**	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes**	**Med Nodes**
10	4	5-5-Group	2.482	1.208	6.9	7	136	63
10	4	5-5-Distance	2.112	1.237	6.8	7	327	134
12	4	6-6-Group	736.862	256.496	9.2	9	896	449
12	4	4-4-4-Group	5.417	6.046	9.0	9	13535	3633
12	4	4-4-4-Distance	4.216	3.608	9.3	9	50244	22100
12	4	3-3-3-3-Group	2.260	0.570	9.2	9	77919	17619
12	4	3-3-3-3-Distance	14.689	7.523	9.1	9	493343	235092
14	4	5-5-4-Group	1603.174	1744.300	11.4	12	197346	106430
14	4	5-5-4-Distance	823.144	877.068	11.5	12	740293	433766
16	4	4-4-4-4-Group	208.218	221.978	12.8	13	3561281	3986085

Table 5.3: Performance of h_{abstract} with A*

Table 5.4 presents results for the single-abstraction variant $h_{\text{abstractS}}$. These abstractions are simpler and faster to evaluate, but come with the disadvantage that this simplified the original state too much. We were able to observe that this simplification led to overall poor scalability and performance compared to approach one, as the heuristic ended up being less informative. For smaller problem sizes, such as $N = 10$ and $N = 12$, they still work and complete within somewhat reasonable time limits. However, for $N = 14$, the runtimes exceed 4500 seconds on average. Attempts for $N = 16$ were not executed as the disadvantage compared to approach one was already clear with the other test instances.

N	k	*Abstraction*	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes**	**Med Nodes**
10	4	2-GroupS	0.488	0.231	6.9	7	40551	22811
10	4	2-DistanceS	0.170	0.107	7.0	7	15095	10504
12	4	3-GroupS	44.754	37.286	9.1	9	106085	47870
12	4	3-DistanceS	73.367	55.790	9.2	9	76499	33641
14	4	3-GroupS	4764.672	4605.740	11.1	11	1535956	953739
14	4	3-DistanceS	3573.832	3806.820	11.2	11	2807351	2342687

Table 5.4: Performance of $h_{\text{abstractS}}$ with A*

While being theoretically weak compared to the other heuristics, as $h_{\text{breakpoint}}$ does not guarantee cost-optimal solutions, it still performed well in practice for the tests we conducted. Compared to the other heuristics we could observe that using $h_{\text{breakpoint}}$ resulted in less node expansions across all instances. This of course came with the trade-off in time needed to solve the instances, which is natural given that the computation of $h_{\text{breakpoint}}$ is more complex compared to the simpler heuristics like the gap heuristic. It has a similar average and median solution length indicating that in practice, even with no guarantee in cost-optimal solutions, we still get mostly optimal results.

N	k	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes**	**Med Nodes**
10	4	0.228	0.124	7.2	7	309	170
12	4	4.713	2.490	9.4	10	3136	1628
14	4	53.949	23.434	11.8	12	27187	11793
16	4	1036.191	376.794	14.1	14	304633	112091

Table 5.5: Performance of $h_{\text{breakpoint}}$ with A*

Across all heuristics, the best performance was achieved by the breakpoint heuristic regarding the number of nodes expanded. It navigated the search space efficiently.

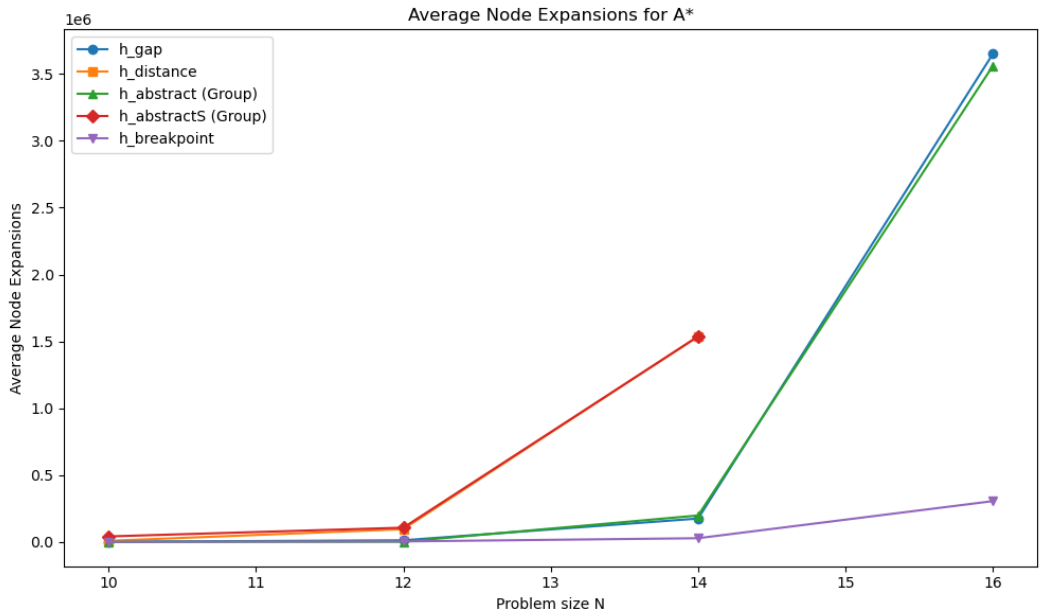


Figure 5.1: Average amount of nodes expanded

Table 5.6 demonstrates the results of running bigger instances with the proposed heuristics. We only include h_{gap}, h_{distance} and $h_{\text{breakpoint}}$ as the domain abstraction approach did not scale well and ended up breaking the memory constraints, because of the solution cache. We included a success rate in our experiments, as not every instance was solved in the 24 hour time limit. The best performance overall was achieved with the breakpoint graph heuristic, as it expanded significantly less nodes in comparison to the other heuristics. For h_{distance} we only had a success rate of 23.81%, where most of the problems solved were the 25 or 50 random walk instances, which also carries over to the other metrics. It might look like it performed better than gap, but this is only because it was able to complete the easier instances, while the gap heuristic was able to solve some of the harder ones.

N	k	**Heuristic**	**Avg Time (s)**	**Med Time (s)**	**Avg Sol**	**Med Sol**	**Avg Nodes (mil)**	**Med Nodes (mil)**	**Success Rate**
18	4	$h_{\text{breakpoint}}$	29399.559	24651.750	16.4	16	103.69	84.68	85.71%
18	4	h_{gap}	26235.485	15083.930	15.5	16	74935.59	40916.77	57.14%
18	4	h_{distance}	15908.277	1111.440	13.4	15	4328.28	282.21	23.81%
20	4	$h_{\text{breakpoint}}$	20272.150	14398.585	16.5	17	53.46	37.84	19%
20	4	h_{gap}	27.869	27.869	14.0	14	66.97	66.97	9.5%

Table 5.6: Performance of heuristics with IDA*

For the next table, we examine the initial h values at the start of the search. These values provide insight into how informed a heuristic is before the search. In general, a higher initial h indicates a more informed heuristic, which can help guide the search more efficiently and reduce the number of node expansions.

As shown in Figure 5.2, the breakpoint heuristic consistently yields the highest initial h

Heuristic	$N = 10$	$N = 12$	$N = 14$	$N = 16$
h_{gap}	4.16	5.13	6.03	6.92
h_{distance}	2.76	3.79	5.33	-
h_{abstract} (Group)	4.92	4.95	5.93	5.57
h_{abstract} (Distance)	4.57	4.47	5.66	-
$h_{\text{abstractS}}$ (Group)	1.89	4.39	5.45	-
$h_{\text{abstractS}}$ (Distance)	3.02	5.03	5.20	-
$h_{\text{breakpoint}}$	5.48	6.78	8.32	9.73

Table 5.7: Average initial h on different TopSpin sizes with A*

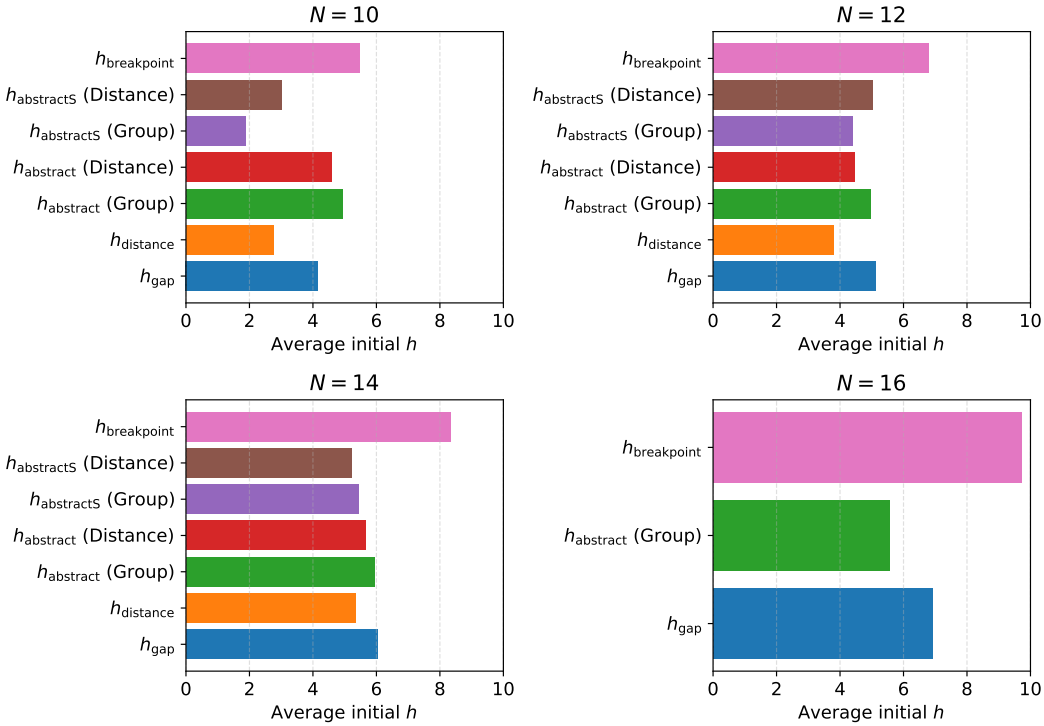


Figure 5.2: Initial h values across different instance sizes N

values across all tested instance sizes. This indicates that it is generally the most informed heuristic at the start of the search, which helps reduce search effort by better guiding the exploration toward the goal.

5.2.1 Negative Results

With a (12, 4)-TopSpin instance the 6-6-Group abstraction resulted in 12 minute search time on average, which is not feasible for even bigger instances as this does not scale well regarding time. The more complex it gets to solve the abstraction, the longer the search will take, as in contrary to PDBs our approach is calculating the solution length during the search and this applies to both approaches.

Our approach for the abstraction heuristic unfortunately did not scale well, when used on bigger instances with IDA*. The amount of cached solutions exploded quickly making the memory usage exceed the limit. For those bigger instances PDBs are more feasible and faster than our approach.

6
Conclusion

In this thesis, we explored several heuristic approaches tailored to solving the TopSpin puzzle. While traditional solutions often rely on precomputed pattern databases, we aimed to investigate lightweight, domain-specific alternatives that can be computed at runtime.

We presented four heuristics: the gap heuristic, a distance-based heuristic inspired by the Manhattan distance, two variations of domain abstraction heuristics, and a breakpoint graph heuristic. Each heuristic was formally defined, analyzed in terms of heuristic properties (goal-awareness, admissibility, consistency, and safeness), and evaluated using A* and IDA* search on instances of increasing complexity.

The gap heuristic proved simple yet effective for smaller problem sizes, but it exhibited poor scalability. In contrast to the Pancake problem, we must divide the heuristic value by two to ensure admissibility and consistency in TopSpin. This significantly reduces its informativeness and hinders performance on larger instances.

The distance-based heuristic has the worst performance, as the amount of node expansions exploded quickly, making it infeasible for bigger instances and while it is quick to compute this leads to long search times.

The domain abstraction heuristics, particularly the approach using multiple abstractions, showed promise on smaller instances, but it failed to scale to larger instances due to runtime and memory constraints. The approach of using a distance based abstraction (e.g. odd and even) did not work well for the approach with multiple abstractions, but showed an increase in performance for the second approach. This is because approach two simplifies the state too much and using a distance based abstraction counteracts this.

Finally, the breakpoint graph heuristic, despite relying on an approximation of MAX-ACD, performed surprisingly well, even without admissibility or consistency guarantees. While in theory it is rather weak without these guarantees, it showed that it is very close to the optimal solution returned by the gap heuristic for example.

From our experiments, we conclude that there is no universally optimal heuristic for TopSpin: each has trade-offs in terms of informativeness, computational cost, memory usage and

optimality. However, the breakpoint graph heuristic, when optimized correctly can be very promising for TopSpin.

Bibliography

[1] Vineet Bafna and Pavel A Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on computing*, 25(2):272–289, 1996.

[2] Alberto Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM journal on discrete mathematics*, 12(1):91–110, 1999.

[3] Joseph C Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[4] Sridhar Hannenhalli and Pavel A Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM (JACM)*, 46(1):1–27, 1999.

[5] Malte Helmert. Landmark heuristics for the pancake problem. In *Proceedings of the International Symposium on Combinatorial Search*, volume 1, pages 109–110, 2010.

[6] Malte Helmert. State-space search: Analysis of heuristics. March 2025. URL https://ai.dmi.unibas.ch/_files/teaching/fs25/ai/slides/ai-b10.pdf.

[7] Raphael Kreft, Clemens Büchner, Silvan Sievers, and Malte Helmert. Computing domain abstractions for optimal classical planning with counterexample-guided abstraction refinement. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 221–226, 2023.

[8] Ferdinand Lammertink. Puzzle or game having token filled track and turntable, October 3 1989. US Patent 4,871,173.

[9] Pedro Olímpio Pinheiro, Alexsandro Oliveira Alexandrino, Andre Rodrigues Oliveira, Cid Carvalho de Souza, and Zanoni Dias. Heuristics for breakpoint graph decomposition with applications in genome rearrangement problems. In *Brazilian Symposium on Bioinformatics*, pages 129–140. Springer, 2020.

[10] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Pearson, 2016.

[11] Shrawan Kumar Sharma and Shiv Kumar. Comparative analysis of manhattan and euclidean distance metrics using a* algorithm. *J. Res. Eng. Appl. Sci*, 1(4):196–198, 2016.

[12] Nathan Sturtevant. Hierarchical Open Graph 2 (HOG2). https://github.com/nathansttt/hog2, 2025. Last Accessed: 18-07-2025.

[13] Fan Yang, Joseph Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662, 2008.