

# **Vergleich von Optimierung für den A\*-Algorithmus zur Lösung des n-Puzzles: IDA\*, Pattern Database und andere Heuristiken**

## **Bachelorarbeit**

Ngoc Cuong Vu  
# 385240

22. November 2022

Betreuer: Prof. Dr. Benjamin Blankertz

Zweitbetreuer: Vera Röhr



Technische Universität Berlin  
Fakultät IV – Elektrotechnik und Informatik  
Institut für Softwaretechnik und Theoretische Informatik  
Fachgebiet Neurotechnologie

# Kurzfassung

Das **n-Puzzle** ist ein interessantes Problem, da es eine hohe Komplexität aufweist und eine gute Möglichkeit bietet, Optimierungen zu vergleichen. Das Hauptziel dieser Abschlussarbeit ist es, den **A\*-Algorithmus** zu optimieren, um dieses Problem zeitlich und räumlich effizient zu lösen, im Vergleich zu einem weiteren großartigen Algorithmus zur Behandlung von **n-Puzzle: Iterative Deepening A\***, sowie zu einigen verschiedenen **Heuristiken**, beginnend mit der zulässigen, aber effizient zu implementierenden Heuristiken: **Manhattan Distance** und **Linear Conflict**. Als nächsten wichtigen Punkt der Arbeit wird die Idee von **Additive Pattern Database Heuristic** dargestellt, ebenso wie man diese Datenbank Schritt für Schritt optimal aufbauen kann. Am Ende werden alle Methoden und Spezifikationen verglichen und analysiert, um Erkenntnisse zur Lösung größerer und komplexerer Probleme zu gewinnen.

# Abstract

The **n-puzzle** is an interesting problem because it has high complexity and offers a good way to compare optimizations. The main goal of this thesis is to optimize the **A\* search algorithm** to solve this problem efficiently in term of time and space compared to another great algorithm for handling **n-puzzles**: **Iterative Deepening A\***, as well as some different **heuristics** starting with the admissible but efficient to implement heuristics: **Manhattan Distance** and **Linear Conflict**. As the next important point of the work, the idea of **Additive Pattern Database Heuristic** will be presented, as well as how this database can be optimally built up step by step. In the end, all methods and specifications are compared and analyzed to gain insights to solve larger and more complex problems.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Spielsimulation</b>	<b>4</b>
2.1	Das Puzzle . . . . .	4
2.1.1	Das Puzzle-Brett (Puzzle Board) . . . . .	4
2.1.2	Lösbarkeit . . . . .	4
2.1.3	Moves . . . . .	6
2.2	Heuristiken . . . . .	6
2.2.1	Manhattan-Distanz-Heuristik (Manhattan Distance) . . . . .	7
2.2.2	Linear-Konflikt-Heuristik (Linear Conflict Heuristic) . . . . .	8
2.2.3	Musterdatenbank-Heuristik (Pattern Database) . . . . .	8
2.3	Algorithmen . . . . .	18
2.3.1	A* . . . . .	18
2.3.2	IDA* . . . . .	20
<b>3</b>	<b>Ergebnisse</b>	<b>23</b>
3.1	Optimierungen des A*-Algorithmus . . . . .	23
3.2	Heuristiken . . . . .	24
3.2.1	Der Aufbau von Pattern Database . . . . .	24
3.2.2	Performance mit unterschiedlichen Heuristiken . . . . .	27
3.3	A* und IDA* . . . . .	28
<b>4</b>	<b>Diskussion</b>	<b>29</b>
4.1	Optimierung des A*-Algorithmus . . . . .	29
4.2	Heuristiken . . . . .	29
4.2.1	Der Aufbau von Pattern Database . . . . .	29
4.2.2	A* mit verschiedene Heuristiken . . . . .	30
4.2.3	IDA* mit verschiedene Heuristiken . . . . .	31
4.2.4	Verbesserung von A* und IDA* im Vergleich . . . . .	31
4.3	A* und IDA* . . . . .	31
<b>5</b>	<b>Fazit</b>	<b>33</b>
5.1	Forschungsthema und Ergebnisse . . . . .	33
5.2	Praktische Implikationen . . . . .	33
5.3	Ausblick . . . . .	34

# Abbildungsverzeichnis

1.1	Einige Varianten von n-Puzzle, die Beispiele sind alle bereits sortiert (die leere Kachel im Ziel ist <b>unten rechts</b> definiert).	1
2.1	Ein Standard-Board von 15-Puzzle im Zielzustand.	4
2.2	Entscheidungsbaum zur Prüfung der Lösbarkeit des <b>Puzzle-Board</b>	5
2.3	Dieses 15-Puzzle Board hat 4 Reihe und 4 Inversionen: (2,1) (11,10) (11,9) (10,9); die leere Kachel ist in 1. Reihe von unten → lösbar	5
2.4	Mögliche Züge einer Konfiguration von 15-Puzzle und was passiert, wenn die leere Kachel nach rechts verschoben wird.	6
2.5	Die Berechnung der <b>Manhattan Distance Heuristic</b> für 15-Puzzle	7
2.6	Beispiel für <b>Linear Conflict</b>	8
2.7	Visualisierung der Funktionsweise der <b>Statically-Partitioned Additive Database</b> für die 5-5-5-Partition von 15-Puzzles	9
2.8	Spielsimulation: Die Darstellung einer Pattern-Gruppe (1,2,3,4,7) aus 5-5-5 Partition des 15-Puzzle in Zielzustand.	9
2.9	Jeweils Variante für 5-5-5, 6-6-3 und 7-8 Partition des 15-Puzzle.	10
2.10	Einige erste Schritte ausgeführt durch der BFS Algorithmus für die Erstellung von <b>Pattern Database</b> für eine Partition (6,10,11,14,15) des 15-Puzzles.	11
2.11	Umwandlung von Zustand zu Kombination-Index und Permutation-Index.	13
2.12	Umwandlung von Kombination-Index = 2747 und Permutation-Index = 19 zu Zustand.	13
2.13	Die leeren Kacheln sowie der <b>Zeros-Index</b> vor und nach Anwendung von <b>freeMove()</b> von Pattern-Gruppe: (6,10,11,14,15) im Zielzustand.	14
2.14	Die leeren Kacheln konnten nicht durch <b>freeMove()</b> manche Stelle nicht erreichen.	15
2.15	Wie die leeren Kacheln in einem Byte für eine 8-Kacheln-Partition notiert werden.	15
2.16	Pseudo-code für Erstellen der <b>Pattern Databases mit dem modifizierten BFS-Algorithmus</b>	16
2.17	Ein Zustand in <b>nextMoves</b> kann mehr als eine Null von der Zustände in <b>currentMoves</b> erhalten. Da zwei Zustände in <b>currentMoves</b> nach dem Schieben gleiche Position von der <b>Pattern-Kacheln</b> ergeben haben.	17
2.18	Pseudocode für A* in basierter Implementierung	19
2.19	Pseudocode für IDA*	21
3.1	Wie sich bessere Heuristiken beim Lösung von 15-Puzzle mit A* und IDA* bezüglich expandierte Knoten (links) und Zeit (rechts) im Prozent verbessern.	27
3.2	Speicherverbrauch von A*(links) und IDA*(rechts) - IntelliJ Profiler	28

# Tabellenverzeichnis

3.1	A*-Performance mit verschiedenen Optimierungen im Vergleich . . . . .	23
3.2	Übersicht über eine PDB 5-5-5 von 15-Puzzle. Bauzeit: 1 Minute. Speicherkapazität: 1.536 kB . . . . .	24
3.3	Übersicht über eine PDB 6-6-3 von 15-Puzzle. Bauzeit: 10 Minuten. Speicherkapazität: 11.265 kB . . . . .	25
3.4	Übersicht über eine PDB 7-8 von 15-Puzzle. Bauzeit: 10 Stunden. Speicherkapazität: 563.063 kB . . . . .	26
3.5	A*-Performance mit verschiedenen Heuristiken im Vergleich . . . . .	27
3.6	IDA*-Performance mit verschiedenen Heuristiken im Vergleich . . . . .	27
3.7	Performance vom A*- und IDA*-Algorithmus mit PDB 7-8 . . . . .	28

# 1 Einleitung

**n-Puzzle** ist ein Schiebepuzzle, das auf einem  $k \times k(n + 1)$  Raster mit  $n$  Kacheln stattfindet, die jeweils von 1 bis  $n$  nummeriert sind, und eine leere Kachel zum Schieben. Die Aufgabe besteht darin, die Kacheln in die richtigen Reihenfolgen zu bringen. Typische Spielgrößen für dieses Puzzle sind 8- und 15 Puzzles. Selbst diese kleinen Bretter sind interessante Herausforderungen, da sie eine große Komplexität aufweisen und die Optimierungen gut verglichen werden können. Mit zunehmende  $n$  wächst das **n-Puzzle** faktoriell (schneller als konstantes exponentielles Basiswachstum) sowohl in der Zeitkomplexität als auch in der Platzkomplexität.

Die Art und Weise, wie eine Person ein Puzzle spielt oder lernt, unterscheidet sich von der des Computers. Menschen werden besser, wenn sie aus ihren Fehlern oder den Erfahrungen anderer Spieler lernen. Zusammen mit dem Wissen über andere grundlegende Spiele können sie ein Puzzle lösen und ein guter Spieler werden. Andernfalls muss ein Programm oft alle Pfade des Puzzles kennen und jedes Ergebnis berechnen, um eine optimale Lösung zum Besiegen des Gegenspielers zu erhalten. Aber bedeutet das immer, dass der Computer den menschlichen Spieler schlägt, oder versucht ein Informatiker, mit seinem Wissen über Algorithmen und Datenstrukturen Kenntnisse, seine Gegner auf seine Weise zu schlagen?

1	2	3
4	5	6
7	8	

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

Abbildung 1.1: Einige Varianten von n-Puzzle, die Beispiele sind alle bereits sortiert (die leere Kachel im Ziel ist **unten rechts** definiert).

## Verwandtes Puzzle

Das **n-Puzzle** hat einige Ähnlichkeiten mit anderen bekannten Schiebepuzzles wie zum Beispiel **Atomix** und **Sokoban**. Viele der Suchmethoden, die für das 15-Puzzle entwickelt wurden, können problemlos sowohl auf **Atomix** als auch auf **Sokoban** angewendet werden. Eine größere Variante von

**n-Puzzle** ist das 24-Puzzle, das aufgrund seines kleineren Verzweigungsfaktors immer noch als einfacher als **Atomix** und **Sokoban** eingestuft wird: Die Konfiguration des **n-Puzzles** erlaubt bis zu 4 Züge, während andere Puzzles mehr Züge haben und in einigen Fällen die gespielten Züge nicht rückgängig gemacht werden können. Andernfalls kann jede Instanz von **n-Puzzle** in **Atomix** formuliert werden, aber nicht umgekehrt. [1]

## Literatur

Seit seiner Erfindung wurde das n-Puzzle ausgiebig erforscht, und das 15-Puzzle mit der trivialen Manhattan-Distanz-Heuristik ist wahrscheinlich das am gründlichsten analysierte Puzzle dieser Art, zunächst von Johnson und Story [9] in 1979. Danach wurde eine Variante von **A\*** eingeführt: **Iterative deepening A\* (IDA\*)** [10], um den Speicherbedarf zu verringern. Weiter mit der Zeitleiste kam der **Linear Conflict** [11] als Verbesserung der Heuristik auf, und mit der immer weiter verbesserten Heuristik kann dann die größere Variante: 24-Puzzle [12] gelöst werden. Zu diesem Zeitpunkt ist die Interaktion zwischen den Kacheln noch nicht ausreichend berücksichtigt. Aber dieses Problem wurde gelöst, da die Arbeit über das Konzept der **Pattern Database** [13] bewiesen hat, dass die heuristische Schätzung umso besser wird, je mehr Kacheln zusammen betrachtet werden. Nicht lange danach wurde die Technik der Pattern Database verbessert, und die Lösung kann innerhalb von Sekunden gefunden werden. [3]

## Ziel der Arbeit

Das Ziel der Arbeit ist, die Lösung von **n-Puzzle** in kurzer Zeit mit angemessenem Speicherbedarf zu erhalten und dabei die unterschiedlichen Ansätze zu vergleichen. Die Strategie besteht darin, das Puzzle mit **A\*** mit einer problemspezifischen internen Optimierung und einer einfach zu implementierenden **Manhattan Distance-Heuristik** zu lösen.

Die verschiedenen Heuristiken werden dann analysiert und verglichen, insbesondere die Musterdatenbank (**Pattern Database**), wenn die Größe des Problems zunimmt, da mit besseren Heuristiken die Anzahl der Erweiterungsknoten stark reduziert werden kann, wodurch sowohl die Anforderungen an die Programmressourcen als auch die zur Lösung erforderliche Zeit reduziert werden.

Ein weiterer Teil ist das Anpassen von **IDA\*** beim Lösen von **n-Puzzle**. Obwohl **IDA\*** theoretisch langsamer als **A\*** ist, weil im Gegensatz zu **A\*** **IDA\*** die angetroffenen Zustände nicht speichert, sodass der Knoten möglicherweise mehrere Male untersucht wird, benötigt es nicht viel Speicherverbrauch, um ausgeführt zu werden. Aus diesem Grund wird für größere Probleme von **n-Puzzle** vorzugsweise **IDA\*** verwendet.

Das 15-Puzzle wurde in vielen Arbeiten zu diesem Thema am gründlichsten analysiert. Dies liegt daran, dass es schnell zu implementieren ist, eine klare Heuristik wie **Manhattan Distance** hat und sein Suchbaum nicht zu groß ist. Dies ist auch in dieser Bachelorarbeit keine Ausnahme. Das 15-Puzzle steht im Mittelpunkt aller Arbeiten zur Analyse und Lösung von **n-Puzzles** im Allgemeinen.

## Struktur der Arbeit

Diese Bachelorarbeit beginnt mit der Modellierung des Puzzles (wichtige Objekte und Funktionen) in der Programmiersprache Java. Außerdem werden alle verwendeten Heuristiken vorgestellt, wobei der Schwerpunkt auf **Additive Pattern Database Heuristic** liegt und wie man diese Datenbank effizient aufbaut. Nachfolgend wird die Leser erfahren, wie **A\*** mit seinen Optimierungen und **IDA\*** beim Lösen des Puzzles angepasst werden. Es folgt das experimentelle Ergebnis und seine Interpretation sowie einige Schlussfolgerungen. Die Arbeit endet mit einem Ausblick auf die nächsten Schritte, zukünftige Arbeit und ein Fazit.

## 2 Spielsimulation

### 2.1 Das Puzzle

Vor dem eigentlichen Lösen des Puzzles mit festgelegten Algorithmen muss das Puzzle-Brett simuliert und implementiert werden.

#### 2.1.1 Das Puzzle-Brett (Puzzle Board)

Jeder Zustand (Konfiguration) des Puzzles wird durch ein Board-Objekt dargestellt. Das Board besteht aus einem Byte-Array, dessen Elemente von 1 bis  $n$  nummeriert sind. Die leere Kachel (*blank*) des Puzzles wird durch 0 dargestellt und ihre Zielposition ist innerhalb dieser Bachelorarbeit immer **unten rechts** definiert.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

Abbildung 2.1: Ein Standard-Board von 15-Puzzle im Zielzustand.

#### 2.1.2 Lösbarkeit

Nicht alle Konfigurationen können gelöst werden. Um genauer zu sein, haben Johnson und Story 1879 [9] bewiesen, dass nur die Hälfte aller  $(n + 1)!$  Konfigurationen von **n-Puzzle** gelöst werden können. Bevor die Strategie angewendet werden, muss die Konfiguration des Boards geprüft werden, ob es lösbar ist.

**isSolvable():**

- Zunächst zählen wir die **Inversionen**, die auf dem Spielbrett vorkommen. Ein Paar von Kacheln  $a$  und  $b$  bildet eine **Inversionen**, wenn  $a$  vor  $b$  erscheint, aber  $a > b$ .
- Dann wird die Reihenposition der leeren Kachel **von unten** bestimmt.

## 2.1. DAS PUZZLE

- Da nun alle erforderlichen Informationen gesammelt sind, kann die Lösbarkeit der Konfiguration überprüft werden:

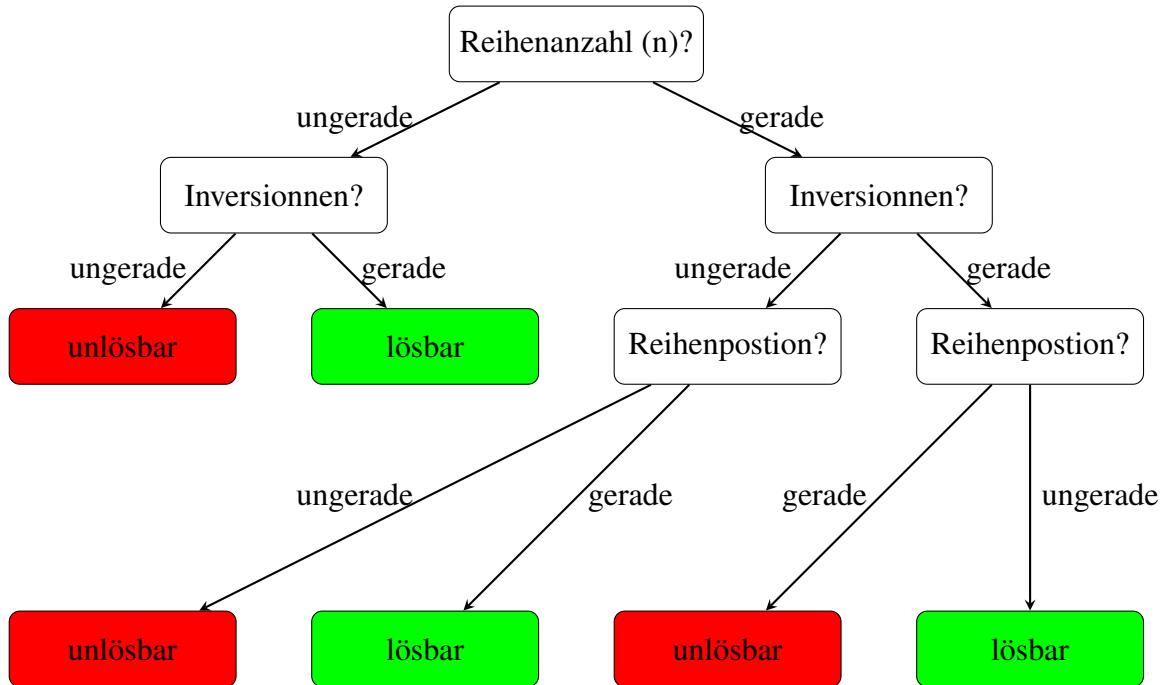


Abbildung 2.2: Entscheidungsbaum zur Prüfung der Lösbarkeit des **Puzzle-Board**

2	1	3	4
5	6	7	8
11	10	9	12
13	14	15	0

Abbildung 2.3: Dieses 15-Puzzle Board hat 4 Reihe und 4 Inversionen: (2,1) (11,10) (11,9) (10,9); die leere Kachel ist in 1. Reihe von unten → lösbar

**Anmerkung:** Wenn eine Konfiguration vom 15-Puzzle mit der leeren Kachel in der **unteren rechten** Ecke im Zielzustand nicht lösbar ist, kann es jedoch gelöst werden, wenn die leere Kachel jetzt in der **oberen linken** Ecke definiert ist. Der Grund dafür ist, dass die Reihenposition der leeren Kachel nun von oben bestimmt wird und dies immer umgekehrt sein wird wie zuvor beim 15-Puzzle (oder anderen Puzzles mit gerader Reihenzahl).

**Random Board:** Zufällige lösbare Boards können auch erstellt werden, indem zufällige Konfigurationen durch Mischen der Kacheln generiert und dann geprüft werden, ob es lösbar sind. Wenn die gerade erstellte Konfiguration mithilfe von `isSolvable()` als nicht lösbar bestimmt wird, wird der

## 2.2. HEURISTIKEN

Vorgang nochmal wiederholt, bis ein zufälliges lösbares **Puzzle-Board** gefunden wird. Eine andere Möglichkeit, eine zufällige Konfiguration von n-Puzzle zu regenerieren, besteht darin, die Kacheln aus einer bekannten lösbaren Konfiguration (normalerweise Zielzustand) zufällig zu verschieben.

### 2.1.3 Moves

Das Puzzle enthält auch einige bewegungsbezogene Funktionen. Dieser Abschnitt konzentriert sich jedoch nur auf die Standardbewegungsfunktionen (zum Schieben), da andere Bewegungsfunktionen dazu bestimmt sind, die **Additive Pattern Database Heuristic** zu unterstützen und später erklärt werden.

**validDirections():** Basierend auf der Position von der leeren Kachel bestimmt diese Funktion, in welche Richtung sie geschoben werden kann: UP, DOWN, RIGHT und LEFT. Sobald der Zug entschieden ist, wird:

**move(direction):**

- die leere Kachel und sein Nachbar in der entsprechenden Richtung (**direction**) vertauscht, das **Puzzle-Board** befindet sich in einem neuen Zustand.
- wird die Zugzahl (*move count*) um eins erhöht.
- Die Richtung selbst wird sowohl auf dem Puzzelpfad (Lösungspfad) als auch beim letzten Zug des Puzzles aufgezeichnet, sodass verhindert werden kann, dass das Puzzle bei Bedarf beim nächsten Zug in seinen vorherigen Zustand zurückkehrt.

5	6	10	15
4	1	3	7
8	11	←	→12
13	14	2	9

5	6	10	15
4	1	3	7
8	11	12	←
13	14	2	9

Abbildung 2.4: Mögliche Züge einer Konfiguration von 15-Puzzle und was passiert, wenn die leere Kachel nach rechts verschoben wird.

## 2.2 Heuristiken

Die heuristische Suche ist ein allgemeines Problemlösungsverfahren, das unter Verwendung der heuristischen Bewertungsfunktion effizient geschätzte Kosten des Knotens für seinen Zielzustand berechnen kann. Zwei wichtige Eigenschaften einer Heuristik sind **Zulässigkeit** und **Konsistenz**. [8]

- Eine heuristische Funktion wird als **zulässig** bezeichnet, wenn sie die Kosten zum Erreichen des Ziels niemals überschätzt, das heißt die geschätzten Kosten (Gewicht) zum Ziel nicht höher als

## 2.2. HEURISTIKEN

die reellen geringstmöglichen (optimal) Kosten sind. Wenn eine zulässige heuristische Funktion mit dem richtigen Algorithmus verwendet wird, ist es garantiert, eine optimale Lösung zurückzugeben (wenn das Problem lösbar ist).

- Angenommen, dass die geschätzten Kosten von der Knoten (**K**) zum Ziel (**Z**) und zu seinem Nachfolger (**N**) sind  $h_{K-Z}$  und  $h_{K-N}$ . Außerdem die geschätzten Kosten vom Nachfolger zum Ziel ist  $h_{N-Z}$ . Dann wird eine heuristische Funktion als **konsistent** oder **monoton** bezeichnet, wenn sie die Dreiecksungleichung:  $h_{K-Z} \leq h_{K-N} + h_{N-Z}$  befolgen. Eine konsistente Heuristik ist auch eine zulässige Heuristik, das Gegenteil gilt jedoch nicht immer.

Betrachten einen Zustand  $s$  des Puzzles. Wenn  $g(s)$  die Anzahl der durchgeführten Züge und  $h(s)$  die heuristische Schätzung sind, dann ist  $f(s) = g(s) + h(s)$  beim **A\*** eine Metrik, um die Priorität des Knotens zu schätzen, damit der Algorithmus entscheiden kann, welche Knoten **“vielversprechend”** (oder **“zielführend”**) ist. Allerdings funktioniert  $f(s)$  beim **IDA\*** als eine untere Grenze (**lower bound**), damit der Algorithmus schätzen kann, welche Tiefe soll der Suchbaum sein. Je genauer dieses  $f(s)$  ist, desto schneller findet **A\*** oder **IDA\*** die optimale Lösung.

### 2.2.1 Manhattan-Distanz-Heuristik (Manhattan Distance)

Wenn es darum geht, **n-Puzzle** zu lösen, würde die bekannte **Manhattan Distance** aufgrund seiner Einfachheit und anständigen Kostenschätzung wahrscheinlich zunächst in Betracht gezogen werden. Außerdem können sich Kacheln in **n-Puzzle** nur horizontal oder vertikal bewegen, wie diese Heuristik berechnet würde: Für jede Kachel des Puzzles außer der leeren Kachel wird der Mindestabstand in Rastereinheiten von dieser Kachel zu ihrem Zielort bestimmt, und die Summe dieser Werte ergibt die heuristisch geschätzten Kosten  $h(s)$  der aktuellen Konfiguration.

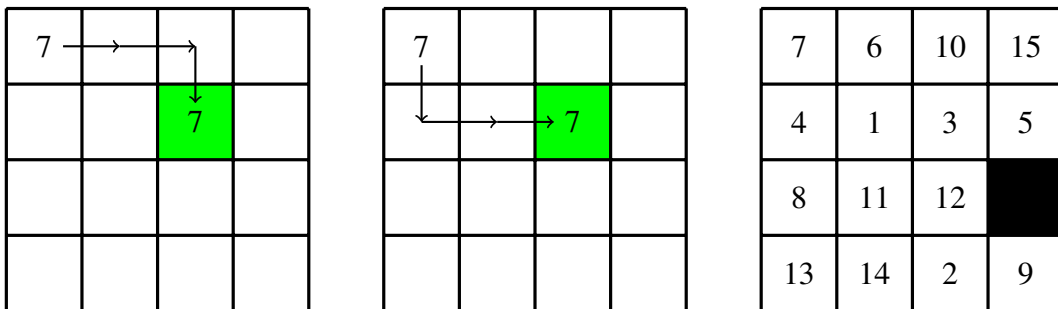


Abbildung 2.5: Die Berechnung der **Manhattan Distance Heuristic** für 15-Puzzle

Um die Zielposition zu erreichen, muss die Kachel-7 (siehe Abbildung 2.5) aus der aktuellen Position (oben links) mindestens 3 Züge machen.  $h(s) = 3$  ist also der geschätzte Wert von dieser Kachel nach der Manhattan-Distanz-Heuristik. Mit dieser Regel würde die heuristische Schätzung des am weitesten links liegenden Boards lauten:  $h_{MD} = 3 + 1 + 3 + 4 + 4 + 2 + 1 + 3 + 4 + 1 + 1 + 0 + 0 + 4 + 4 = 35$

Der heuristische geschätzte Wert eines Zustandes von **n-Puzzle** nach **Manhattan-Distance** kann wie folgt berechnet:

$$h_{MD}(s) = \sum_{k=1}^n |x_k - x_{kg}| + |y_k - y_{kg}|$$

## 2.2. HEURISTIKEN

Wobei:

- $x_k, x_{kg}$ : vertikale Achsenkoordinaten von Kachel-k in aktuellem Zustand und Zielzustand.
- $y_k, y_{kg}$ : horizontale Achsenkoordinaten von Kachel-k in aktuellem Zustand und Zielzustand.

### 2.2.2 Linear-Konflikt-Heuristik (Linear Conflict Heuristic)

Obwohl **Manhattan Distance** eine zulässige Heuristik ist, wird es immer noch eine bessere Heuristik benötigt, um schwierigere Konfigurationen oder sogar größere Puzzles wie 24-Puzzle zu lösen. 1992 führten Hansson, Mayer und Yung [11] die **Linear Conflict Heuristic** ein, die erste signifikante Verbesserung der **Manhattan Distance Heuristic**. Ein **Linear Conflict** tritt auf, wenn sich 2 Kacheln in ihrer Zielreihe oder -spalte befinden, jedoch in umgekehrter Reihenfolge zueinander. Um diesen Konflikt zu lösen, muss eine Kachel aus dem Weg gehen und dann zurückkommen, damit die andere Kachel ihre Zielposition erreichen kann. Diese beiden Bewegungen können also in die **Manhattan Distance** addiert werden, ohne deren Zulässigkeit zu verletzen.

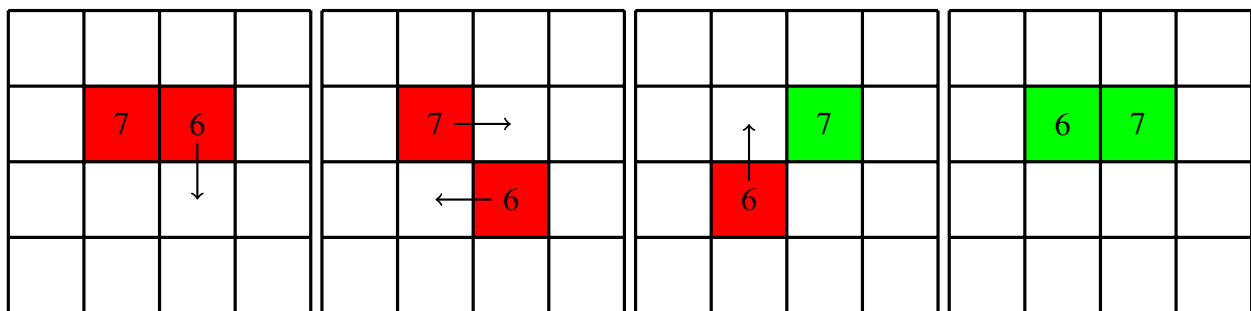


Abbildung 2.6: Beispiel für **Linear Conflict**

Die Kachel-6 und die Kachel-7 (siehe Abbildung 2.6) liegen in der richtigen Reihe, aber in der falschen Spalte, deshalb entsteht hier ein **Linear Conflict** zwischen die beiden. Entweder Kachel-6 oder Kachel-7 muss aus der Reihe weichen, damit die Andere das Ziel erreichen kann. Die Abbildung hat gezeigt, dass es insgesamt 4 Züge für den gesamten Prozess gibt. Aber 2 Züge davon gehören zur **Manhattan Distance**, weil sie auf den Zielzustand abzielen. Daher werden für **ein** Konflikt nur **zwei** Züge angerechnet. Aus diesem Grund wird **Linear Conflict** nicht als heuristisches Verfahren allein verwendet, sondern genauer als **Manhattan Distance with Linear Conflict** genannt wird.

### 2.2.3 Musterdatenbank-Heuristik (Pattern Database)

**Pattern Database** ist die beste bekannte Heuristik zur Lösung von **n-Puzzle** und auch andere Problems [4]. Die Interaktionen von Kacheln auf dem Weg zum Ziel werden berücksichtigt und es liefert einen **lower bound**, die oft besser ist als **Manhattan Distance** mit **Linear Conflict**. Aber das hat natürlich seinen Preis: Wir müssen die Datenbank aus dem Zielzustand aufbauen, basierend auf einer bestimmten Partition des Boards. Dieser Vorgang dauert einige Zeit und beansprucht Speicherplatz auf der Festplatte. Die Vorteile sind es jedoch wert, da die Zeit, die zum Lösen des Puzzles benötigt

## 2.2. HEURISTIKEN

wird, um bis zu mehrere tausend Male reduziert werden kann. Diese Bachelorarbeit beschäftigt sich mit **Statically-Partitioned Additive Database**.

### Statically-Partitioned Additive Database

Die Kacheln werden in disjunkte Gruppen von der **Pattern-Kacheln** unterteilt, dann werden die minimale Anzahl von Zügen als heuristischer Wert der **Pattern-Kacheln** in jeder Gruppe vorberechnet, die erforderlich sind, um diese **Pattern-Kacheln** zu ihrer Zielpositionen zu bringen. Diese Werte werden danach in Datenbank unter einem (oder mehreren) Index, der von dem Zustand dieser Pattern-Gruppe abhängt, gespeichert. Dann wird bei der Suche ein bestimmter Zustand angegeben, jede indexbasierte Pattern-Gruppe sucht nach den entsprechenden Werten in der Datenbank (**look-up**), diese Werte werden dann summiert, um eine insgesamt zulässige Heuristik für den gegebenen Zustand zu berechnen.

1	3	7	11
2	8	13	5
9	6	4	10
0	15	14	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

Abbildung 2.7: Visualisierung der Funktionsweise der **Statically-Partitioned Additive Database** für die 5-5-5-Partition von 15-Puzzles

Für jede Pattern-Gruppe: Basierend auf der Position der **Pattern-Kacheln** wird die minimale Zuganzahl zu ihrem Zielzustand (mit derselben Farbe - siehe Abbildung 2.7) nachgeschlagen. Dann werden diese Zugzahlen jeder Gruppe summiert, um den allgemeinen heuristischen Wert des gesamten Bretts zu bilden.

### Partition

1	2	3	4
-1	-1	7	-1
-1	-1	-1	-1
-1	-1	-1	0

Abbildung 2.8: Spielsimulation: Die Darstellung einer Pattern-Gruppe (1,2,3,4,7) aus 5-5-5 Partition des 15-Puzzle in Zielzustand.

## 2.2. HEURISTIKEN

Für die Spielsimulation werden die **Pattern-Kacheln** von der aktuell in Betracht gezogene Pattern-Gruppe wie gewohnt nummeriert und die **Nicht-Pattern-Kacheln (don't care tiles)** werden alle mit **-1** nummeriert. Die leere Kachel kann entweder mit **0** wie normal (für Bewegung) oder auch mit **-1** (um in der Datenbank zu speichern) nummeriert werden.

Für das 15-Puzzle gibt es 3 Möglichkeiten, das **Puzzle-Board** in Pattern-Gruppe aufzuteilen: 5-5-5 Partition, 6-6-3 Partition und 7-8 Partition. Die Datenbankgröße von **n-Puzzle** hängt von **n** und der Größe der Pattern-Gruppe ab und kann mit dieser Formel berechnet werden:

$$\binom{n+1}{k} \times k! = \frac{(n+1)! \times k!}{k!(n+1-k)!} = \frac{(n+1)!}{(n+1-k)!}$$

Wobei  $k$  ist die Größe der Pattern-Gruppe und wie diese Formel entstand, wird später erklärt. Es ist auch so klar, dass die Datenbank direkt proportional zu der Größe der Pattern-Gruppe ist. Die Erstellung einer Datenbank für eine 8-teilige Pattern-Gruppe eines 15-Puzzles kann auf einem Standard-PC mehrere Stunden dauern, was auch davon abhängt, wie gut das Erstellungsprogramm ist. Weil es insgesamt:

$$\frac{(n+1)!}{(n+1-k)!} = \frac{(15+1)!}{(15+1-8)!} = \frac{16!}{8!} = 518.918.400$$

Kombinationen gibt, die berechnet und gespeichert werden müssen.

Aus diesem Grund sollte 24-Puzzle in 6-6-6-6 Partition (127.512.000 Kombinationen) aufgeteilt werden, und 35-Puzzle in 5-5-5-5-5-5-5-5 Partition (45.239.040 Kombinationen) aufgeteilt werden, damit es die erforderliche Speicherkapazität nicht zu groß ist.

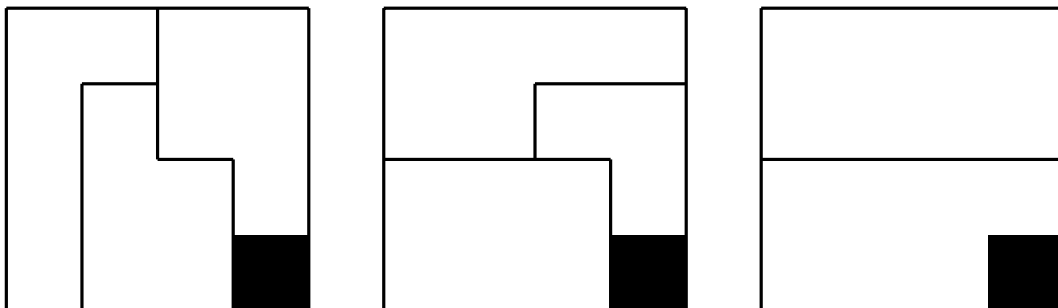


Abbildung 2.9: Jeweils Variante für 5-5-5, 6-6-3 und 7-8 Partition des 15-Puzzle.

### Erstellung der Datenbank

Nachdem die Partitionierung wie oben definiert wurde, kann der Breitensuche-Algorithmus (BFS) zur Erstellung der Datenbanken verwendet werden. Die **Pattern-Kacheln** in der Pattern-Gruppe und die leere Kachel werden wie normal nummeriert, die anderen sind irrelevant und werden mit **-1** nummeriert. Es wird mit dem Zielzustand dieser Pattern-Gruppe, während das BFS läuft, tauscht die leere Kachel den Platz mit anderen Kacheln, sodass ein neuer Zustand entsteht. Aber die Kosten erhöhen sich um eins **NUR**, wenn die leere Kachel den Platz mit einer **Pattern-Kachel** tauscht. Der Zug, mit dem der leeren Kachel ihre Position mit einer **Nicht-Pattern-Kachel** tauscht, wird als freier Zug (**free move**) bezeichnet, da er die Pattern-Gruppe nicht beeinflusst.

## 2.2. HEURISTIKEN

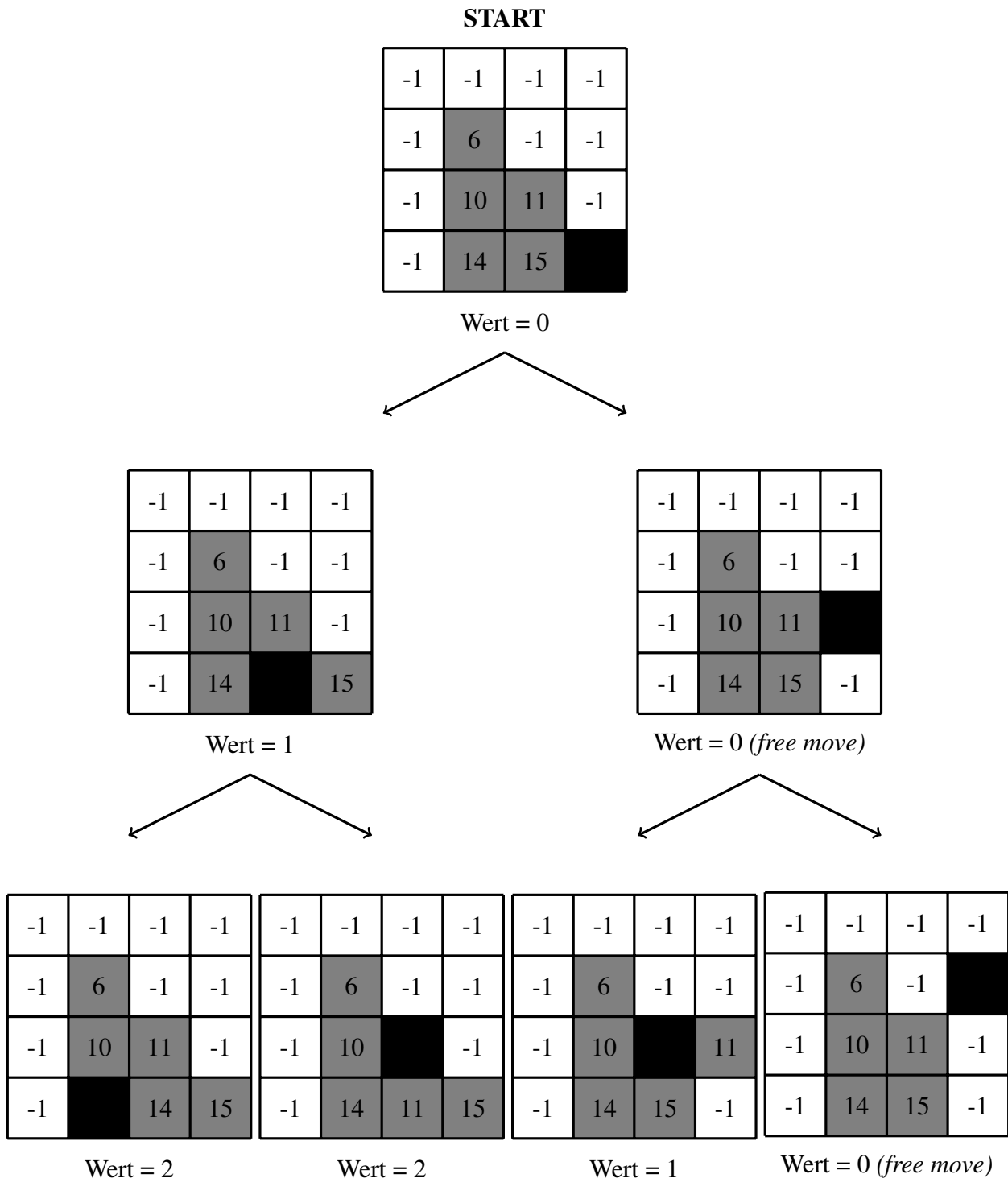


Abbildung 2.10: Einige erste Schritte ausgeführt durch der BFS Algorithmus für die Erstellung von **Pattern Database** für eine Partition (6,10,11,14,15) des 15-Puzzles.

Nun kann der Zustand von der **Pattern-Kacheln** mit dem dazugehörigen heuristischen Wert in vielen Formen in einer Datenbank gespeichert werden. Zum Beispiel: Zustände können als String gespeichert werden: “- - - - 6 - - - AB - - EF - : 0” oder “- - - - 6 - - - AB - - EF : 2”. Wobei “-” für die

## 2.2. HEURISTIKEN

**Nicht-Pattern-Kacheln** (einschließlich auch die leere Kachel), und der Buchstabe, der für Kachel mit entsprechender Nummer größer als 9 dargestellt wird. (A = 10; B = 11 ...).

Selbst mit der zuvor erwähnten vernünftigen Partitionierung ist die Erstellung einer Datenbank unter Verwendung von BFS mit Warteschlange (*queue*) ungünstig, da enorme Ressourcen benötigt werden, wenn die Größe der Warteschlange zunimmt, insbesondere beim Aufbau einer Datenbank von der 8-Kacheln-Pattern-Gruppe für 15-Puzzle. Aus diesem Grund ist ein Kompromiss durch die Verwendung eines komplexeren BFS-Algorithmus ohne die traditionelle Warteschlange und einer kompakteren Version des Boards notwendig.

### Kompakteres Brett und freier Zug (free move)

**Zuordnungsfunktionen [5] (Index Functions):** Der Zustand des Boards kann durch 3 Indizes veranschaulicht werden, wenn nur **Pattern-Kacheln** von Interesse sind.

- **Kombination-Index:** Eine 16-Bit-Binärzahl kann zeigen, welche Positionen auf dem Brett von der **Pattern-Kacheln** besetzt sind, wobei die **Pattern-Kacheln** mit 1 und die andere Kacheln mit 0 kenn gezeichnet werden. Diese 16-Bit-Binärzahl wird **Kombination-Format** genannt. Für **k-Pattern-Kacheln** in **n-Puzzle** gibt es:  $\binom{n+1}{k}$  **Kombination-Formats**. Diese Binärzahlen müssen jedoch einem Array mit dem Indiz von 0 bis  $\binom{n+1}{k} - 1$  zugeordnet werden. Dieses Array wird vor dem Aufbau der **Pattern Database** erstellt und fungiert als eine Lookup-Tabelle, sodass beim Empfangen vom **Kombination-Format** des Zustands dessen Index in der Lookup-Tabelle gefunden und dieser Index als **Kombination-Index** gespeichert werden kann.
- **Permutation-Index** (bezieht sich auf die Reihenfolge von Kacheln): Dieser Index ist eine (Integer) Zahl zwischen 0 und  $k! - 1$  für **k-Pattern-Kacheln**. Mit einer Variante des Fisher-Yates [5] Shuffle kann eine Funktion erstellt werden, die eine Permutation entsprechend einem Permutation-Index liefert. Diese Permutation basiert aber immer auf der originalen Permutation von der **Pattern-Kacheln** (im Zielzustand). Danach können die Permutationen durch einem entsprechendem **Permutation-Index** in einer Lookup-Tabelle gespeichert werden, und dann anhand von diesem **Permutation-Index** kann auch die entsprechende Permutation gefunden werden.

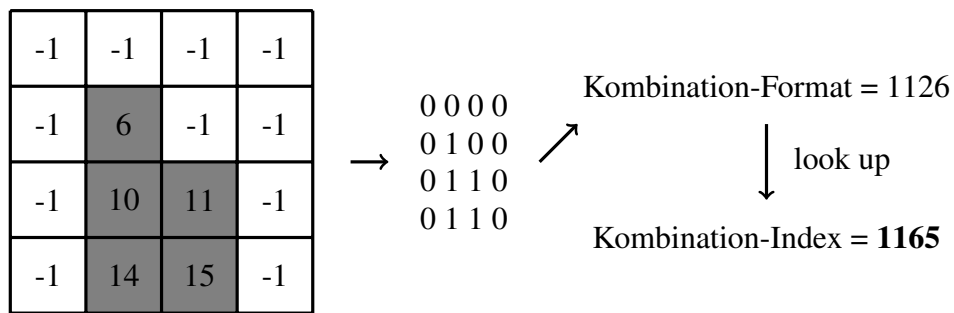
Jetzt ist klar, dass die Datenbank einer Pattern-Gruppe X Einträge haben sollte. Wo bei:

$$X = \#KombinationIndex \times \#PermutationIndex = \binom{n+1}{k} \times k! = \frac{(n+1)!}{(n+1-k)!}$$

Dies entspricht der Anzahl der möglichen Zustände der Pattern-Gruppe (ohne die leere Kachel zu berücksichtigen).

Hier ist die Beispiele:

## 2.2. HEURISTIKEN



Da die **Pattern-Kacheln** befinden sich im Zielzustand, daher Permutation-Index = 0

Abbildung 2.11: Umwandlung von Zustand zu Kombination-Index und Permutation-Index.

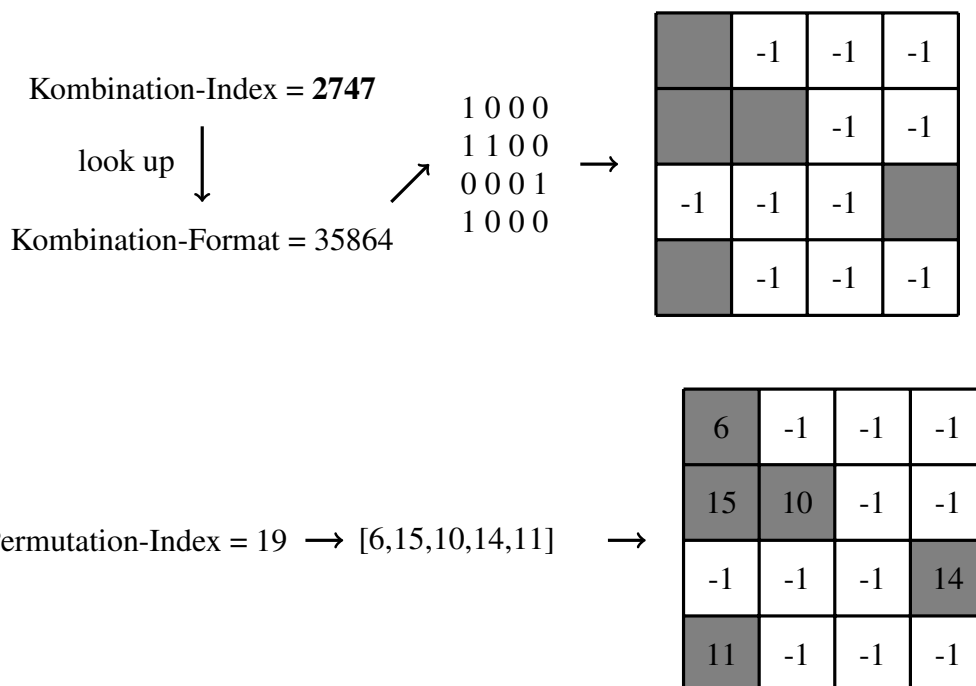


Abbildung 2.12: Umwandlung von Kombination-Index = 2747 und Permutation-Index = 19 zu Zustand.

Nun können wir die vorberechneten heuristischen Werte in der Datenbank speichern. Die Datenbank soll idealerweise ein 2-dimensionales Byte-Array mit A Zeilen und B Spalten sein, wobei A = Anzahl der **Kombination-Index** und B = Anzahl der **Permutation-Index**. Somit können wir diese Indizes verwenden, um den heuristischen Wert des Boards, das durch diese 2 Indizes dargestellt wird, nachzuschlagen.

- Der letzte ist der **Zeros-Index**. Aber für ein besseres Verständnis dieses Index sollte die Idee der Funktion **freeMove()** bekannt sein: anstatt jeweils nur einen Zug zu machen, bewegt sich jede leere Kachel als **frei** über das ganze Brett, solange die **Pattern-Kacheln** nicht berührt werden. Alle diese erreichbaren Positionen durch die leeren Kacheln werde dann geprüft, ob es ein

## 2.2. HEURISTIKEN

**trigger position** ist. **Trigger position** von der leeren Kachel ist die Position, die möglicherweise mit einer Pattern-Kachel den Platz tauschen kann. Am Ende werden aller diese **trigger positions** in ein Binärformat (16 Bit) aufgezeichnet und schließlich als **Zeros-Index** (Integer) übertragen. Das bedeutet, dass der **Zeros-Index** beschreibt, wo die leeren Kacheln im Board sind. Mit anderen Worten, **freeMove()** bringt ein Zustand zu seinem **reale bereiten** Zustand, wobei die leeren Kacheln bereits sind, ein **PDB-Zug** (tauschen mit einer Pattern-Kachel) zu machen, damit die Zustände nächster Tiefe erreicht wird. Diese Funktion spart vielen Schritte, weil es viele freien Züge der leeren Kacheln ignoriert, wobei nicht wichtig bei der Erstellung der **Pattern Database**.

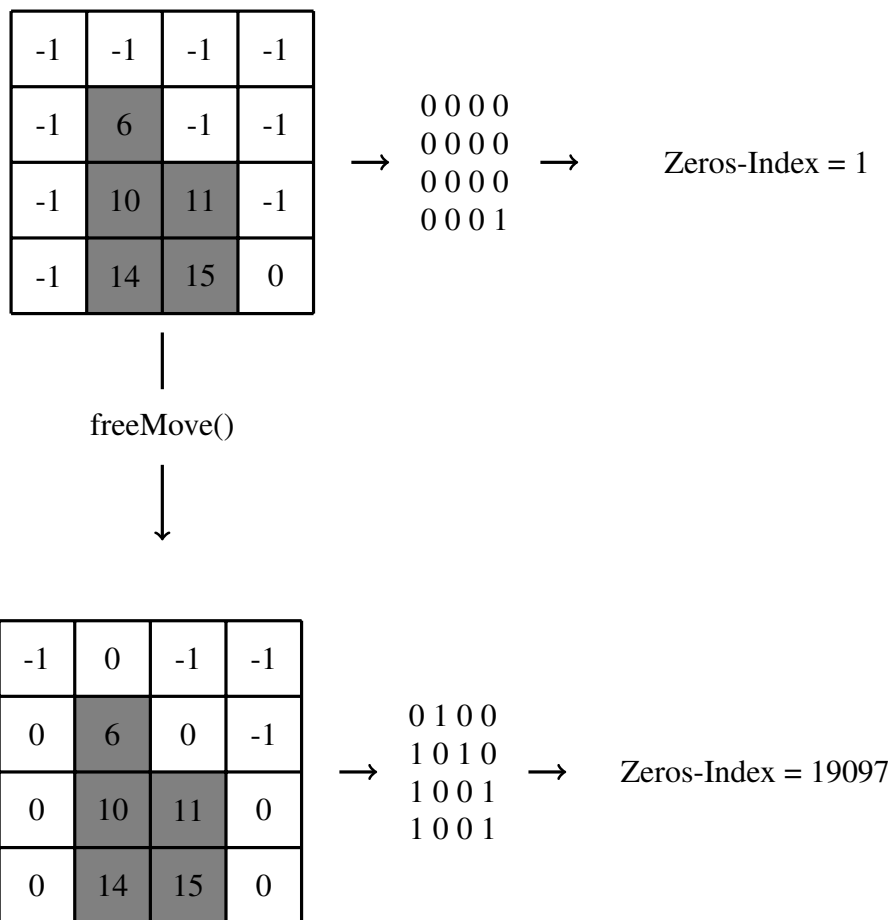


Abbildung 2.13: Die leeren Kacheln sowie der **Zeros-Index** vor und nach Anwendung von **freeMove()** von Pattern-Gruppe: (6,10,11,14,15) im Zielzustand.

## 2.2. HEURISTIKEN

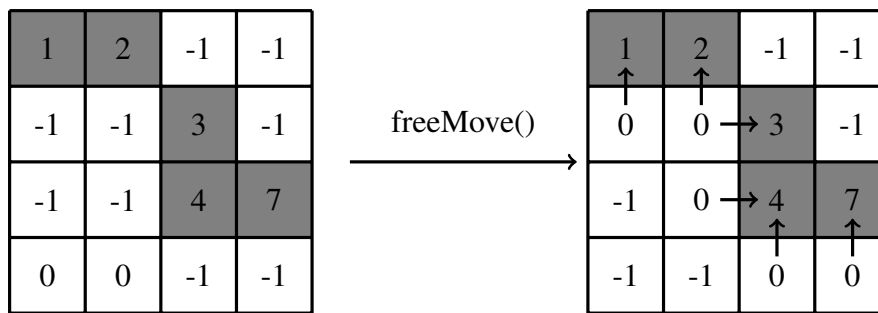


Abbildung 2.14: Die leeren Kacheln konnten nicht durch **freeMove()** manche Stelle nicht erreichen.

Die Abbildung 2.14 beschreibt ein weiteres Beispiel für **freeMove()**: die leeren Kacheln können die obere Ecke nicht erreichen, da die **Pattern-Kacheln** den Weg blockieren. Außerdem gibt es bei aller 5 leere Kacheln (nach dem **freeMove()**) insgesamt 6 mögliche Züge, die das Board in unterschiedliche Zustände bringen können. Das bedeutet auch, dass von diesem Zustand 6 weitere Zustände erweitert werden können, um die Datenbank aufzubauen. Es bedeutet auch, dass die Methode **freeMove()** einige **nicht so wichtige** Züge (freie Züge) vernachlässigt, die den geschätzten heuristischen Wert der aktuellen Positionen der **Pattern-Kacheln** nicht beeinflussen.

- Es gibt allerdings noch eine 8-Bit Version von **Zeros-Index: Zeros-ByteIndex**. Besonders nur für die Erstellung der Datenbank für 8-Kacheln-Partition von 15-Puzzle. Dies liegt daran, dass ein Zustand beim 8-Kachel-Partition höchstens 8 leere Kacheln hat. Wobei es perfekt ist, wenn es im 8-Bit-Datentyp (**byte** in Java) gespeichert ist. Es reduziert auch den benötigten Arbeitsspeicher, der für den Aufbau dieser Datenbank notwendig ist.

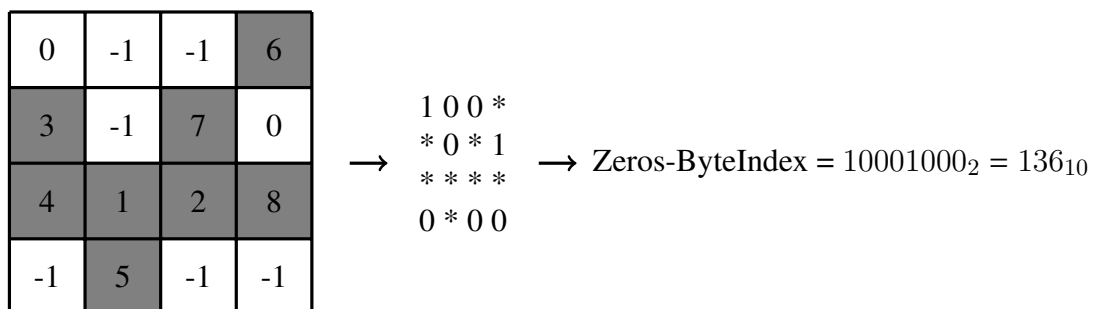


Abbildung 2.15: Wie die leeren Kacheln in einem Byte für eine 8-Kacheln-Partition notiert werden.

**Note:** In Java ist eine Zahl größer als 127 als byte negativ, aber das hat immer noch das richtige Bit, also spielt das keine Rolle.

### Array-basierte Warteschlange für BFS

Diese Optimierung ist sehr praktisch beim Erstellen einer **Pattern Database** für große Pattern-Gruppe (7-8 Partition), da die Warteschlangendatenstruktur nicht mehr für eine so große Menge von Elementen geeignet ist. Darüber hinaus wissen wir, wie in der vorherigen Analyse, genau, wie viele Zustände

## 2.2. HEURISTIKEN

vorhanden sind, und durch die Verwendung eines Arrays können wir die erforderlichen Daten in den entsprechenden Indizes speichern, die von jedem Zustand aus der **Zuordnungsfunktionen** erzeugt werden können.

```
void pdbBuilder(pattern_tiles) {
    Board goal = getGoalBoard(pattern_tiles);
    byte[C][P] database;
    int[C][P] currentMoves;
    currentMoves[goal.c][goal.p] = goal.z;
    depth = 1;
    while(database not filled) {
        int[C][P] nextMoves;
        forall(c < C) {
            forall(p < P) {
                z = currentMove[c][p];
                if(z == 0)
                    continue;
                Board current = recreate(c,p,z);
                Board pdb = current.freeMove();
                forall(direction in pdb.expand()) {
                    Board child = pdb.move(direction);
                    nextMoves[c][p] |= child.z;
                    if(database[c][p] == 0)
                        database[c][p] = depth;
                }
            }
        }
        depth++;
        currentMoves = nextMoves;
    }
}
```

Abbildung 2.16: Pseudo-code für Erstellen der **Pattern Databases mit dem modifizierten BFS-Algorithmus**

Drei 2-dimensional Array[C][P] werden benötigt. Wobei C = Anzahl der Kombination-Index und P = Anzahl der Permutation-Index. Das sind:

- **database[C][P]** als Byte Array: Speichern heuristische Werte der Zustände
- **currentMoves[C][P]** als Integer-Array (Byte-Array für 8-Kacheln-Pattern-Gruppe): Speichern den aktuellen **Zeros-Index** des Zustands, damit er erweitert werden kann.
- **nextMoves[C][P]** als Integer-Array (Byte-Array für 8-Kacheln-Pattern-Gruppe): der **Zeros-Index** werden hier auch gespeichert, für die Zustände, die aus **currentMoves** erweitert wurden.

## 2.2. HEURISTIKEN

Am Anfang wird der **Zeros-Index** von originalem Board (**Pattern-Kacheln** im Zielzustand) mit entsprechendem **Kombination-Index** und **Permutation-Index** ins **currentMoves** - Array eingetragen.

Für jede Schleife werden alle Elemente vom Arrays **currentMoves** durchsucht. Ist der Wert immer noch 0, bedeutet dies, dass der Knoten noch nicht erweitert wurde und die Suche fortgesetzt wird. Wenn ein Wert ungleich Null gefunden wird, wird das **Puzzle-Board** aus diesen 3 Indizes rekonstruiert, und **freeMove()** wird angewendet, sodass der Knoten bereit ist, verschoben/erweitert zu werden.

Nach dem Verschieben hat der Knoten einen neuen Zustand mit entsprechenden Indizes, sodass die aktuelle Tiefe (gleichbedeutend mit dem aktuellen heuristischen Wert der Ebene) des Suchbaums in die Datenbank eingetragen wird (nur wenn dies noch nie zuvor geschehen ist). Außerdem wird die Null, die für diesen Zug in das Array **nextMoves** (mit entsprechenden Indizes) verantwortlich, übertragen. Das bedeutet auch, sowohl jeder Zustand mehrere Änderungen annehmen kann als auch mehrere Änderungen im selben Zustand enden können. Der übermittelte Zustand kann also mehr als eine Null enthalten.

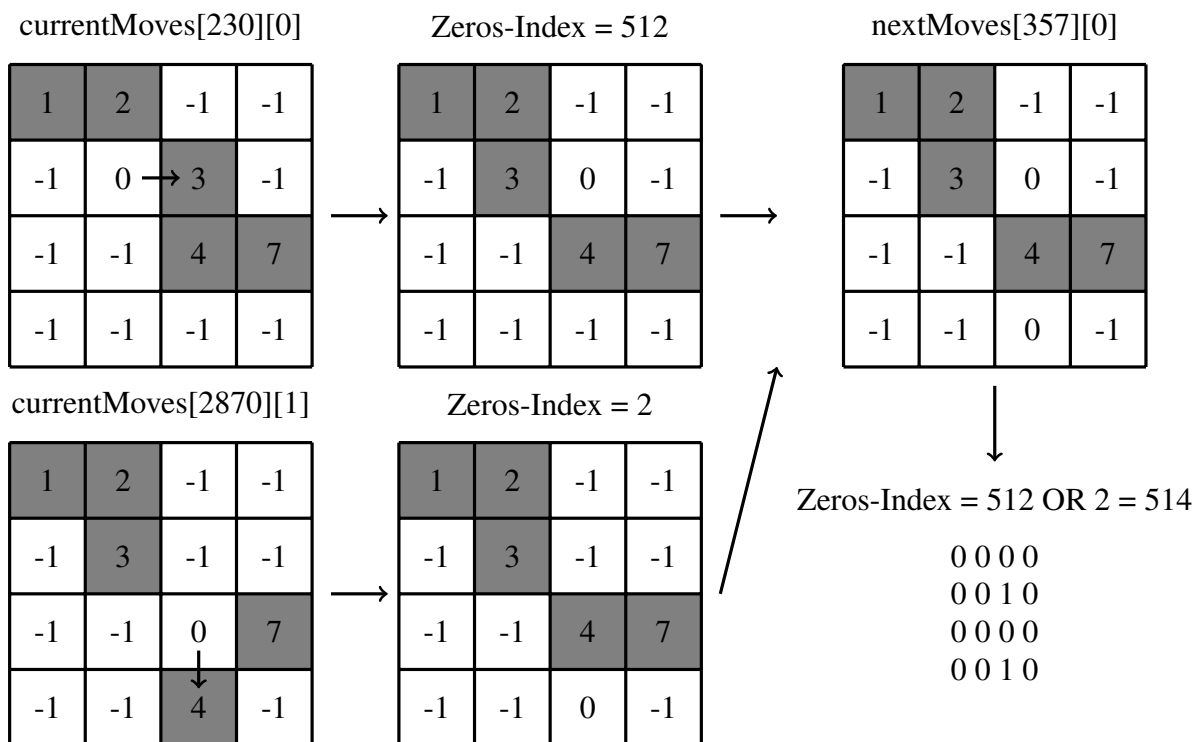


Abbildung 2.17: Ein Zustand in **nextMoves** kann mehr als eine Null von der Zustände in **current-Moves** erhalten. Da zwei Zustände in **currentMoves** nach dem Schieben gleiche Position von der **Pattern-Kacheln** ergeben haben.

Der Vorgang wird wiederholt, bis das Array **database** mit dem Wert gefüllt ist. Dann wird **database** mithilfe von **Java ByteBuffer** in einer Datei zur Verfügung gespeichert.

## 2.3 Algorithmen

### 2.3.1 A\*

A\* ist ein klassischer Suchalgorithmus. Es ist sehr zeiteffizient, aber der benötigte Speicher wächst exponentiell. Da es sich an alle jemals angetroffenen Zustände erinnert. Die Kosten  $f(s) = h(s) + g(s)$  des Zustands wird verwendet, um die Suche zu leiten. Dabei werden zunächst die vielversprechenden Knoten auf der Grundlage der geschätzten heuristischen Werte des Zustands  $h(s)$  und der Anzahl der durchgeführten Züge  $g(s)$  untersucht.

Die heuristische Funktion  $h(s)$  hat zentrale Rolle für A\*. Mit der Schätzung des Abstands zum Ziel wird der Algorithmus in jedem Schritt geführt, um den nächstbesten Knoten zu finden.

Zwei weitere wesentliche Teile von A\* sind die offene und geschlossene Liste. Die geschlossene Liste (**closed list**) ist eine Hash-Tabelle, wobei Zustände als Schlüssel und die Anzahl der durchgeführten Züge  $g(s)$  des Zustands als Werte, damit kann es verhindern, dass der alte Zustand erneut erweitert zu werden, oder die  $g$ -Werte eines Zustands können aktualisiert werden. Die offene Liste (**open list**) ist eine Prioritätswarteschlange, sie enthält alle Zustände, die noch nicht erweitert wurden. Die Position des Zustands in der **open list** basiert auf dem  $f$ -Wert =  $g + h$ . Zustände mit kleinerem  $f$ -Wert werden zuerst untersucht. Dann gibt es 3 folgende Fälle für ihre Nachfolger:

- Fall 1: Wenn der Zustand zum ersten Mal erweitert ist (der Zustand befindet sich noch nicht in der **closed list**), daher wird der Zustand sowohl in der **open list** (mit der Priorität nach  $f$ -Wert) als auch in der **closed list** hinzugefügt.
- Fall 2: Der Zustand wurde untersucht (es liegt schon in der **closed list**), aber seine durchgeführte Züge  $g(s)$  (**move count**) sind kleiner als der gespeicherte Wert, also wird sein Wert aktualisiert in die **closed list** und noch mal in der **open list** hinzugefügt wird. Dieser Prozess dient dazu, die Algorithmen zu beschleunigen, da der Zustand in der offenen Liste mit besseren  $g(s)$  und somit besseren  $f(s)$  aktualisiert wird, indem es einfach wieder in die Warteschlange gestellt wird, aber es wird jetzt eine neue Position (höhere Priorität) haben.
- Fall 3: Der Zustand wurde untersucht und die  $g(s)$  ist nicht besser als der gespeicherte Wert, also verwerfen wir ihn.

Nun wird der Algorithmus ausgeführt, bis der Zielzustand erreicht ist oder kein Zustand mehr in der **open list** ist (die ursprüngliche Konfiguration in diesem Fall ist nicht lösbar). Wie schon vorher erwähnt, mit einer zulässigen Heuristik liefert A\* immer eine optimale Lösung. Um jedoch eine Aussage über die Laufzeit treffen zu können, sollte die Konsistenz der Heuristik erwähnt werden. Denn bei einer zulässigen, aber nicht konsistenten heuristischen Funktion könnte der Knoten mehrfach neu expandiert werden, wodurch sich die Laufzeit erhöht. [7]

## 2.3. ALGORITHMEN

```
Board aStar(Board start, Heuristic h) {
    start.calculateHeuristic(h);
    OPEN = [start]
    CLOSED = []
    while OPEN not empty {
        Board best = OPEN.poll();
        forall(direction in best.expand()) {
            Board child = best.move(direction);
            child.calculateHeuristic(h);
            child.moveCount++;
            if(child.isSolved())
                return child;
            Board visited = CLOSED.get(child)
            if(visited == null or visited.g > child.g) {
                CLOSED.add(child, moves);
                OPEN.add(child); //Priorität nach f-Wert
            }
        }
    }
    return null;
}
```

Abbildung 2.18: Pseudocode für A\* in basierter Implementierung

Wenn neuer Zustand durch einen Zug erstellt, sein Zugzähler um eins erhöht. Außerdem wird die heuristische Schätzung erneut bestimmt. Nach dem Prüfen für **closed list** wird es in **open list** (anhand aktuelle g-Wert und h-Wert) hinzugefügt oder verworfen.

### Optimierungen des A\*-Algorithmus

Weitere Untersuchungen ergaben mehrere Optimierungen, um den Algorithmus sowohl zeitlich als auch räumlich zu verbessern. Diese Optimierungen basieren auf der spezifischen Verwendung von A\*, [2] um n-Puzzle zu lösen, und sind sehr notwendig, da A\* oft Schwierigkeiten hat, das Puzzle zu lösen, wenn seine Größe zunimmt.

- **Bessere Positionierung** (in Prioritätswarteschlange): Wenn es mindesten zwei Zustände mit demselben f-Wert gibt, wird der Zustand mit höherem g-Wert bevorzugt, und in der Warteschlange priorisiert. Der Grund dafür ist, dass das Ziel in der letzten Schicht mit f-Wert der Suche schneller gefunden werden kann.
- **Array-Basierte Prioritätswarteschlange (Array-Queue)**: Die Tatsache, dass die Konfigurationen von 15-Puzzle einen kleinen bekannten Bereich von f-Werten haben, wird als Vorteil genutzt, um die binäre Heap-Prioritätswarteschlange durch eine effizientere Datenstruktur zu ersetzen: eine 1-stufige Bucket-Prioritätswarteschlange. Diese Prioritätswarteschlange ist ein einfaches Array, das durch f-Werte indiziert ist, jeder Eintrag in dem Array ist eine Liste der

## 2.3. ALGORITHMEN

Knoten in der Prioritätswarteschlange mit dem entsprechenden f-Wert. Mit dieser Datenstruktur kann das Einfügen (**add**), Entfernen (**peek** oder **poll**) und Aktualisieren (**update**) in konstanter Zeit erfolgen. Allerdings ist es besser, wenn diese Warteschlange allerdings aus einem Array von Prioritätswarteschlange konstruiert wird, damit der Vorteil von vorherige Verbesserung nicht verloren geht. Jeder Eintrag wird in dem entsprechenden f-Wert platziert, dann wird hier g-Wert weiter verglichen, damit der Eintrag mit höherem g-Wert hohe Priorität hat. Das bedeutet, mit dieser Optimierung können die Knoten auch schon bessere positioniert (siehe **Bessere Positionierung** Abschnitt) werden.

- **Kompaktere Darstellung der Zustände des Puzzles (Kompakter Zustand):** Eine weitere Möglichkeit, den Speicherbedarf für  $A^*$  zu verringern, besteht darin, die Größe des Zustands zu reduzieren, der bei der derzeitigen Implementierung ein Byte-Array ist. Für das 15-Puzzle reichen die Kacheln von 0 (leere Kachel) bis 15, daher kann jede Kachel durch 4 Bits (ein **Nibble**) dargestellt werden, und dann kostet der Zustand nur 8 Bytes (50 % weniger). Die Benutzung eines Nibble-Arrays kann zu einem gewissen **overhead** bei Array-Operationen führen. Aber mit dieser Optimierung wird der Hashing-Prozess von **closed list** schneller, zusammen mit der Tatsache, dass  $A^*$  viel Zeit damit verbringt, **open list** und **closed list** zu bearbeiten, sollte dies für diesen **overhead** kompensiert werden. Ansonsten ist es manchmal keine schlechte Idee, Zeit gegen Speicher zu tauschen, vor allem, wenn der Speicher kritisch und ein sensibles Thema ist wie in  $A^*$ .

Für 8-Puzzle gibt es insgesamt  $9!/2 = 181440$  mögliche lösbare Konfigurationen des Boards, sodass es für moderne Computer kein Problem sein sollte, alle diese Konfigurationen zu speichern. Aber wenn es um 15-Puzzles geht, kann die Anzahl der zu speichernden Konfigurationen bis zu 10 Billionen ( $16!/2 \approx 10.000.000.000.000$ ) betragen. Das ist eine riesige Zahl, die unsere Computer verarbeiten muss.

Aus diesem Grund ist  $A^*$  keine gute Wahl (selbst wenn es auch sehr stark optimiert ist), um schwierige Konfigurationen (für diejenigen, die mehr als 70 Züge als optimale Lösung haben) von 15-Puzzles sowie größere **n-Puzzle** zu lösen.

### 2.3.2 IDA\*

Der **IDA\*** Algorithmus wurde erstmals 1985 von Richard Korf [10] beschrieben. Dies war tatsächlich der erste Algorithmus, der es ermöglichte, optimale Lösungen für das 15-Puzzle zu finden [1]. Es ist eine Variante der iterativen Tiefensuche, die sich die Idee von  $A^*$  aneignet, eine heuristische Funktion zu verwenden, um die verbleibenden Kosten zum Ziel zu bewerten. Was dieser Algorithmus bei der Lösung von **n-Puzzle**-Problemen praktikabler als  $A^*$  macht, ist, dass es nicht viel Speicher benötigt, um zu funktionieren. Die Strategie dieses Algorithmus ist anders als  $A^*$ , er verwendet immer noch den f-Wert, aber um die "**nicht-vielversprechende**" Zustände zu vermeiden, sodass sie überhaupt nicht untersucht werden.

Eine Reihe von Tiefensuchen mit steigendem **bound** wird ausgeführt. Dieser **bound** beginnt bei der Schätzung der Kosten im Ausgangszustand (Basis der verwendeten Heuristik). Wenn der f-Wert den **bound** überschreitet, wird der Teilbaum entfernt und der für die nächste Iteration verwendeter **bound** ist dem Minimum aller Werte, die dem aktuellen Zügeslimit überschritten haben. Allerdings terminiert der Algorithmus nicht, wenn kein Weg von Start zum Ziel existiert. In diesem Fall gerät es in eine

### 2.3. ALGORITHMEN

Endlosschleife. Um solche Situation zu verhindern, soll man die Grenze durch eine Multiplikation von dem ersten **bound** mit festem Faktor (in diesem Programm auf 10 gesetzt) setzen.

Für die Implementierung wird die Tiefensuche durch die Methode **dfs(board, bound)** durchgeführt. Diese Methode geben *null* zurück, wenn der f-Wert des Boards größer als aktueller **bound** ist, damit neue Suche mit einem neuem **bound** ausgeführt wird. Außerdem mithilfe einer globalen Variablen **nextBound** ist der beste geschätzte heuristische Wert des laufenden Suchbaums bekannt. Damit der **bound** für die nächste Tiefensuche ein Minimum aus diesen überschrittenen Werte ist. Wenn der Zielzustand gefunden ist, wird es direkt zurückgegeben.

```
int nextBound;
Board idaStar(Board start) {
    currentBound = start.f;
    max_bound = currentBound * 10;
    while (bound < max_bound) {
        Board search = dfs(start, currentBound);
        if (search != null)
            return search;
        currentBound = nextBound;
    }
    return null;
}
Board dfs(Board board, int currentBound) {
    if (board.isSolved())
        return board;
    if (board.f > currentBound) {
        nextBound = f;
        return null;
    }
    min = MAX_VALUE;
    forall (direction in board.expand()) {
        Board search = dfs(board.move(direction), currentBound);
        if (search != null)
            return search;
        if (nextBound < min)
            min = nextBound;
    }
    nextBound = min;
    return null;
}
```

Abbildung 2.19: Pseudocode für IDA\*

Wie bereits erwähnt, verwendet **IDA\*** keinen Speicher außer einem Stack (linear mit der Tiefe des Baums) für den aktuellen Teilbaum. Dadurch wurde das Programm auch sehr effizient umgesetzt.

### 2.3. ALGORITHMEN

Aber diese Vorteile haben ihren Preis: Der angetroffene Zustand kann wieder expandiert werden. Das kann dazu führen, dass ein großer Teilbaum erneut neu bewertet wird, somit erhöht sich die zeitliche Komplexität. Um dieses Problem in den Griff zu bekommen, wird ein Mechanismus zusammen mit dieser Logik festgelegt, der verhindert, dass die Kachel sofort an die Position ihres Vorgängers zurückkehrt. Dies ist eine offensichtliche Optimierung für ungerichtete zugrunde liegende Graphen wie das **n-Puzzle**.

# 3 Ergebnisse

Der Programminhalt dieser Arbeit ist vollständig mit Java (JDK 16) implementiert. Die Experimente wurden auf einem Computer mit einem Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz 3.19 GHz Prozessor, 16GB RAM, unter Windows Version 11 durchgeführt.

Die Analyse mit dem 15-Puzzle werden durchgeführt, wobei die Manhattan-Distanz als Standardheuristik verwendet wird, um die Optimierungen für A\* zu vergleichen und sowohl A\* als auch IDA\* zu verwenden, um verschiedene Heuristiken miteinander zu vergleichen. Die Konfiguration des 15-Puzzles wird in 3 Kategorien unterteilt:

- **Easy Board** hat die optimale Lösung von nicht mehr als 55 Zügen.
- **Medium Board** hat die optimale Lösung von 56 bis 70 Zügen.
- **Hard Board** hat die optimale Lösung von mehr als 70 Zügen

## 3.1 Optimierungen des A\*-Algorithmus

In dieser Sitzung werden 100 zufällig erstellte lösbare **Easy Board** von 15-Puzzle als Benchmark verwendet, um die Verbesserung von A\* durch die oben genannten Optimierungen zu sehen. **Manhattan Distance** wird als heuristisches Kriterium verwendet. Im Zielzustand bleibt die leere Kachel in der unteren rechten Ecke.

- **B** = Basisimplementierung.
- **BP** = Bessere Positionierung.
- **AQ** = Array-Queue.
- **KZ** = Kompaktere Zustände.

	<b>Zeit (s)</b>	<b>Knoten</b>	<b>Knoten per Sekunde</b>
B	757	567.804.109	750.071
KZ	659	567.804.109	861.614
BP	329	258.695.171	786.307
AQ	318	258.477.383	812.821
KZ & BP	321	258.695.171	805.904
KZ & AQ	273	258.477.383	946.803

Tabelle 3.1: A\*-Performance mit verschiedenen Optimierungen im Vergleich

## 3.2 Heuristiken

### 3.2.1 Der Aufbau von Pattern Database

Pattern Database 5-5-5					
[1,2,5,9,13]		[3,4,7,8,12]		[6,10,11,14,15]	
h-Wert	Anzahl	h-Wert	Anzahl	h-Wert	Anzahl
0	1	0	1	0	1
1	5	1	5	1	8
2	15	2	21	2	36
3	42	3	73	3	146
4	129	4	215	4	506
5	399	5	587	5	1549
6	1061	6	1503	6	4145
7	2564	7	3526	7	9825
8	5565	8	7593	8	20629
9	11010	9	14815	9	38274
10	19807	10	26452	10	60970
11	32246	11	42809	11	81428
12	47422	12	62546	12	90601
13	62693	13	79882	13	83675
14	73845	14	86719	14	63741
15	76507	15	78371	15	39428
16	68804	16	58200	16	19271
17	53198	17	35106	17	7291
18	35118	18	16923	18	2116
19	19617	19	6481	19	452
20	9149	20	1879	20	64
21	3547	21	425	21	4
22	1124	22	28		
23	252				
24	40				

Tabelle 3.2: Übersicht über eine PDB 5-5-5 von 15-Puzzle. Bauzeit: 1 Minute. Speicherkapazität: 1.536 kB

### 3.2. HEURISTIKEN

Pattern Database 6-6-3					
[1,2,3,4,5,6]		[9,10,11,13,14,15]		[7,8,12]	
h-Wert	Anzahl	h-Wert	Anzahl	h-Wert	Anzahl
0	1	0	1	0	1
1	5	1	5	1	6
2	18	2	19	2	22
3	60	3	75	3	71
4	173	4	259	4	173
5	484	5	765	5	360
6	1341	6	2083	6	574
7	3571	7	5201	7	685
8	8937	8	12685	8	630
9	20229	9	28404	9	452
10	43503	10	60214	10	252
11	84096	11	115423	11	101
12	153735	12	208055	12	28
13	251917	13	336460	13	5
14	388486	14	508892		
15	536310	15	680101		
16	690471	16	829094		
17	781524	17	862068		
18	807109	18	791892		
19	713733	19	601359		
20	563673	20	394276		
21	368607	21	204052		
22	209336	22	88509		
23	93249	23	27786		
24	34574	24	6959		
25	8807	25	1015		
26	1683	26	102		
27	128	27	6		

Tabelle 3.3: Übersicht über eine PDB 6-6-3 von 15-Puzzle. Bauzeit: 10 Minuten. Speicherkapazität: 11.265 kB

### 3.2. HEURISTIKEN

Pattern Database 7-8			
[1,2,3,4,5,6,7,8]		[9,10,11,12,13,14,15]	
h-Wert	Anzahl	h-Wert	Anzahl
0	1	0	1
1	4	1	2
2	10	2	6
3	44	3	24
4	155	4	95
5	498	5	311
6	1454	6	904
7	3856	7	2392
8	9753	8	6056
9	23740	9	15091
10	57104	10	36696
11	131720	11	84265
12	294927	12	181968
13	624860	13	368821
14	1278710	14	699428
15	2456938	15	1236038
16	4529270	16	2030914
17	7768314	17	3100164
18	12755050	18	4385247
19	19446958	19	5727394
20	28359574	20	6865358
21	38145480	21	7483204
22	48739135	22	7345188
23	56731544	23	6424533
24	62044160	24	4955682
25	60881276	25	3330386
26	55472562	26	1922268
27	44521972	27	935172
28	32781777	28	371883
29	20794252	29	116971
30	11928080	30	26821
31	5692908	31	4025
32	2396445	32	285
33	783878	33	7
34	216751		
35	39436		
36	5465		
37	322		
38	17		

Tabelle 3.4: Übersicht über eine PDB 7-8 von 15-Puzzle. Bauzeit: 10 Stunden. Speicherkapazität: 563.063 kB

### 3.2. HEURISTIKEN

#### 3.2.2 Performance mit unterschiedlichen Heuristiken

Für diese Session wird der aktueller optimierter **A\*-Algorithmus** (mit **Array Queue** und **Kompakter Zustand**) und 100 zufällig erstellte lösbare **Easy Board** als Benchmark verwendet,

- **MD** = Manhattan Distance
- **LC** = Linear Conflict
- **PDB** = Pattern Database (mit Pattern-Partitionen)
- **Speicherbedarf** = maximale benötigte **Heap Memory** laut IntelliJ Profiler (ähnlich wie Abbildung 3.2), um die 100 Boards zu lösen.

	Zeit (s)	Knoten	Knoten/Sekunde	Speicherbedarf (MB)
MD	299	258.477.383	864.473	3.903
MD + LC	51	47.445.625	930.306	1.618
PDB 5-5-5	61	21.087.434	345.696	1.084 (1,5)
PDB 6-6-3	36	12.855.984	357.111	915 (11,3)
PDB 7-8	3	1.114.540	371.513	730 (563)

Tabelle 3.5: A\*-Performance mit verschiedenen Heuristiken im Vergleich

	Zeit (s)	Knoten	Knoten/Sekunde	Speicherbedarf (MB)
MD	892	2.916.584.730	3.269.714	221
MD + LC	239	377.228.431	1.578.362	148
PDB 5-5-5	281	143.074.698	509.162	155 (1,5)
PDB 6-6-3	185	92.975.462	501.597	165 (11,3)
PDB 7-8	10	6.130.964	613.096	720 (563)

Tabelle 3.6: IDA\*-Performance mit verschiedenen Heuristiken im Vergleich

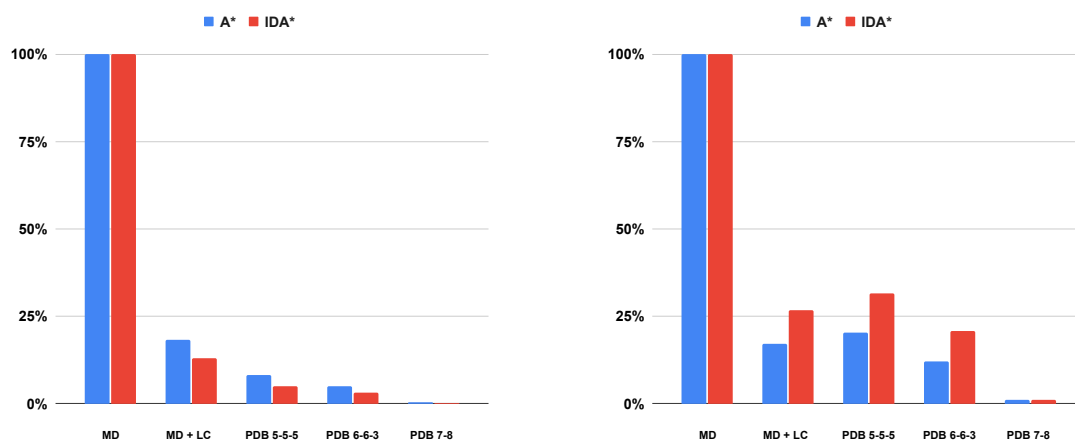


Abbildung 3.1: Wie sich bessere Heuristiken beim Lösung von 15-Puzzle mit **A\*** und **IDA\*** bezüglich expandierte Knoten (links) und Zeit (rechts) im Prozent verbessern.

### 3.3 A\* und IDA\*

Für diese Session werden der aktueller optimierter **A\*-Algorithmus** (mit **Array Queue** und **Kompakter Zustand**) und vorgestellter **IDA\*-Algorithmus** verglichen. Benchmark ist 10 zufällig erstellte lösbare **Hard Boards**, wobei da drin gibt es 6 schwierigsten Konfigurationen mit 80 (maximaler Wert beim 15-Puzzle [5]) Züge als optimale Lösung. Außerdem wird die bisher beste Heuristik: PDB 7-8 für 15-Puzzle verwendet.

	Zeit (s)	Knoten	Knoten/Sekunde
A*	504	14.689.2774	291.454
IDA*	957	473.662.241	494.945

Tabelle 3.7: Performance vom A\*- und IDA\*-Algorithmus mit PDB 7-8

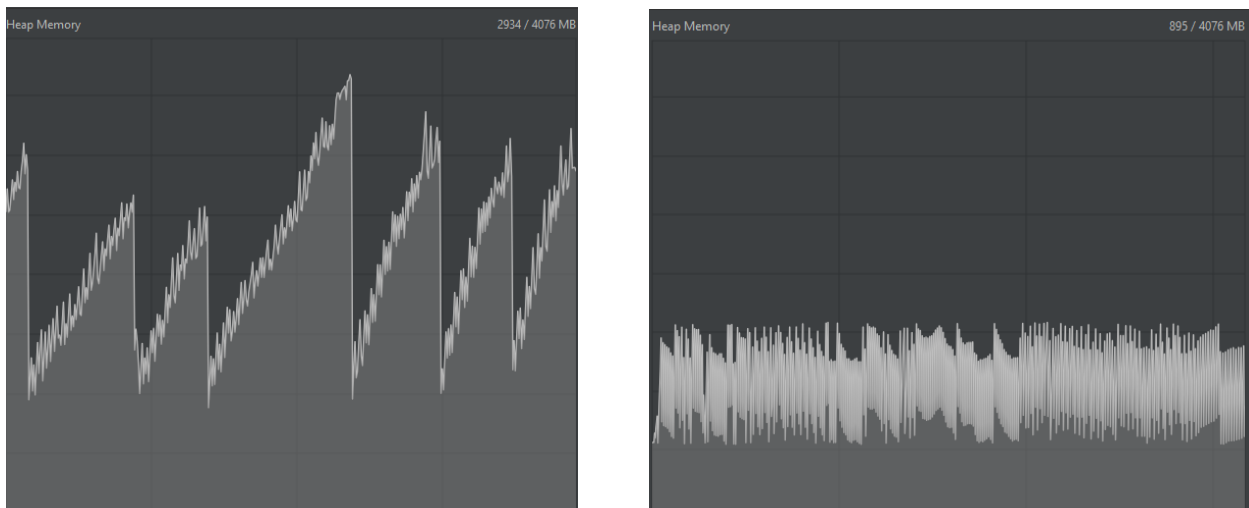


Abbildung 3.2: Speicherverbrauch von A\*(links) und IDA\*(rechts) - IntelliJ Profiler

# 4 Diskussion

## 4.1 Optimierung des A\*-Algorithmus

Durch die Verwendung der kompakteren Darstellung der Zustände (**KZ**) wird nicht nur der Speicherbedarf des Zustands um 50% reduziert, sondern auch der Algorithmus um etwa 15% gegenüber der Basisimplementierung (**B**) beschleunigt. Das bedeutet, dass der Hashing-Prozess schneller abläuft, da anstelle eines Byte-Arrays mit 16 Elementen nur noch 8 Elemente gehasht werden müssen.

Wenn der g-Wert der Knoten auch als Entscheidungspunkt (**BP**) für die Knoten mit dem gleichen f-Wert in der Prioritätswarteschlange berücksichtigt wird, verringert sich die Anzahl der Knoten (54%), die erweitert werden müssen, um das Ziel zu finden, erheblich, wodurch sich die Zeitkosten im Vergleich zur Basisimplementierung (**B**) um das 2,3-fache verringern.

Wie bereits erwähnt, hat die Verwendung von **Array Queue** den gleichen Effekt wie **BP**, da sowohl der g-Wert als auch der f-Wert des Knotens für seine Platzierung in der Prioritätswarteschlange berücksichtigt werden. Das Ergebnis zeigt jedoch, dass die Verwendung von **Array Queue** nicht wirklich schneller ist als die binäre **Heap-Prioritätswarteschlange** in der Basisimplementierung, tatsächlich beschleunigt sie den Prozess nur um 3%. Der Grund dafür ist, dass die Anzahl der erweiterten Knoten nicht groß genug ist, damit die konstante Zeitkomplexität von **Array Queue**:  $O(1)$  besser als **Priority Queue**:  $O(\log_2 n)$  [6] übertrifft.

Wenn **Array Queue** jedoch mit **Kompakter Zustand** kombiniert wird, bringt es eine weitere Verbesserung der Zeitkomplexität (bis zum 1,2-fachen) im Vergleich zu nur **KZ** ohne **AQ**. Dies beweist erneut, dass der **A\*-Algorithmus** einen Großteil seiner Zeit mit der Bearbeitung seiner offenen und geschlossenen Listen verbringt, sodass diese Optimierungen eine gewisse Wirkung haben.

Zusammenfassend lässt sich sagen, dass die Verwendung von **KZ** und **AQ** die Zeitkosten von **A\*** von 757 Sekunden auf 273 Sekunden minimiert hat - eine Verbesserung um das **2,8-fache**.

## 4.2 Heuristiken

### 4.2.1 Der Aufbau von Pattern Database

Abhängig von der Aufteilung (Partition) des Boards, selbst bei gleicher Größe von der Pattern-Gruppe, ist das Ergebnis jeder Aufteilung unterschiedlich.

**PDB 5-5-5** kann sehr schnell erstellt werden und nimmt nur ein kleiner Teil vom Speicherplatz in Anspruch. Der Suchbaum ist mithilfe der Datenbank seine Größe mehr als 2-mal im Vergleich mit **Manhattan Distance with Linear Conflict** reduziert.

## 4.2. HEURISTIKEN

Wenn sich die Größe der Pattern-Gruppe nur um eins erhöht, erhöht sich die mögliche Kombination für 6-Kacheln-Pattern-Gruppe um ungefähr 10-mal im Vergleich mit 5-Kacheln-Pattern-Gruppe. Deshalb braucht es auch 10-mal mehr Zeit und Speicherkapazität, um **PDB 6-6-3** aufzubauen. Es lohnt sich aber definitiv, da die Knoten des Suchbaums weiter reduziert wird, deshalb nimmt der Algorithmen weniger Zeit, um die Konfigurationen zu lösen.

Der Aufbau dieser **PDB 7-8** ist die größte Herausforderung in dieser Arbeit. Der Grund dafür ist, dass es zu viele mögliche Konfigurationen (berechnet in 2.2.3) gibt, die aus der Pattern-Gruppe mit 8 Kacheln erstellt werden können. Das Speichern und Betreiben dieser Konfigurationen in einer Warteschlange wie die normale Version von BFS ist nicht praktikabel, da die Verwendung von **Linked List** (als Queue) oder **Priority Queue** in Java mehr Speicher für Zeiger (**pointer**) verbraucht als ein statisches Array mit fester Größe. Mit anderen Worten kann ein statisches Array mit demselben bereitgestellten Speicher mehr Elemente enthalten als eine Warteschlange. Außerdem wäre für jede Iteration nur 1 Knoten der verarbeitet wird, wenn er aus der Warteschlange entfernt wird, aber wahrscheinlich wird mehr als 1 Knoten zur Warteschlange hinzugefügt, sodass der Speicher erschöpft ist, bevor alle Knoten behandelt werden. Darüber hinaus können die den Zustand darstellenden Indizes mit dem Array verwendet werden, sodass das Element tatsächlich nur eine ganze Zahl (**byte** oder **integer**) ist, die für die aktuelle leere Kachel dieses Zustands dargestellt wird, wodurch das Array noch weniger Speicher verbraucht. Trotz aller unternommenen Anstrengungen schätzt die **PDB 7-8** den Heuristikwert genauer als alle anderen Heuristiken, da mehr Interaktionen der Kacheln berücksichtigt werden, wenn sich das Brett während des Herstellungsprozesses bewegt.

### 4.2.2 A\* mit verschiedene Heuristiken

Das Ergebnis zeigt uns sehr deutlich, dass **Manhattan Distance with Linear Conflict** eine große Verbesserung gegenüber **Manhattan Distance** allein darstellt, es reduziert die Anzahl der erweiterten Knoten um das **5,5-fache** und die Zeitkosten entsprechend um das **6-fache**.

Als die Heuristik zur **Pattern Database** geändert wurde, ist **PDB 5-5-5** immer noch langsamer als **Manhattan Distance with Linear Conflict**, obwohl sich die Knoten um mehr als die Hälfte reduzierten. Denn die Suche in der Datenbank kostet mehr Zeit als die Berechnung des heuristischen Werts nach **Manhattan Distance with Linear Conflict**: Für jeden Zustand, müssen die **Kombination-Index** und **Permutation-Index** für jede Pattern-Gruppe berechnet werden, um den heuristischen Wert jeder Pattern-Gruppe in der Datenbank nachzuschlagen und dann zu summieren.

Mit **PDB 6-6-3** löst der Algorithmus die Konfigurationen jetzt ungefähr zweimal schneller. Das ist der Wert von 10 Minuten und 11 MB Speicherplatz, um die Datenbank zu erstellen, im Vergleich mit nur 1 Minute Erstellungszeit und nur 1,5 MB Speicher von **PDB 5-5-5**.

Mit der aktuellen Implementierung werden 10 Stunden benötigt, um **PDB 7-8** zu erstellen. Es verbraucht auch mehr als ein halbes GB Speicher. Aber mit diesem einmaligen Build kann die Datenbank verwendet werden, um das Puzzle sehr schnell zu lösen. Es kostet nur 3 Sekunden, alle 100 Easy Boards zu lösen. Es ist 12-mal besser als **PDB 6-6-3** und **250-mal** besser als das klassische **Manhattan Distance** in Bezug auf erweiterte Knoten. Die Zeitkomplexität wird etwa **100-mal** verbessert.

Wenn sich die Anzahl der erweiterten Knoten verringert, verringert sich außerdem auch die maximale Anforderung für **Heap Memory** zum Lösen des Puzzles erheblich. Es ist zu beachten, dass der

### 4.3. A\* UND IDA\*

aufgezeichnete Speicherbedarf (Intellij Profiler) auch den Festplattenspeicherbedarf für die Pattern-Datenbank (in Klammern, siehe Tabeller 3.5 und 3.6) enthält, da die Datenbank in das Programm geladen wurde, als es ausgeführt wurde. Das heißt zum Beispiel, der maximalen Bedarf für A\* zum Lösen der Konfigurationen sind eigentlich nur  $730 - 563 = 197$  MB.

#### 4.2.3 IDA\* mit verschiedene Heuristiken

Die Ergebnisse sind ähnlich wie beim A\*-Algorithmus:

- Mit zusätzlichen geschätzten Zügen von **Linear Conflict** wird die benötigte Zeit(3,7-mal) sowie die Anzahl der erweiterten Knoten (7,7-mal) stark reduziert.
- **PDB 5-5-5** reduziert die Knoten weiter, ist aber aufgrund der Look-up Operation immer noch langsamer als **Manhattan Distance with Linear Conflict**.
- Nur wenn es um **PDB 6-6-3** geht, kann sich die Zeit zum Lösen von 100 Konfigurationen wieder verbessern. 1,3-mal schneller und 4-mal weniger Knoten als **MD + LC**
- Mit **PDB 7-8** wird die Lösungszeit ebenfalls um das 90-fache reduziert im Vergleich mit **MD**- nicht gut wie beim A\*. Aber die Anzahl der Knoten, die erweitert werden müssen, sinkt erheblicher (475-mal) als **PDB 7-8** im Vergleich mit **MD** beim A\*.
- Der Speicherbedarf ist deutlich geringer als A\* und es hat sich auch nicht viel geändert.

#### 4.2.4 Verbesserung von A\* und IDA\* im Vergleich

Im Allgemeinen gibt es große Verbesserungen in Zeit-und Platzkomplexität von der **Manhattan Distance** Heuristic bis zu **PDB 7-8**.

**IDA\*** hatte eine bessere Verbesserung in Bezug auf Knoten als A\*, da der Algorithmus, bevor es im richtigen Zweig mit dem richtigen **bound** landen kann, alle anderen Optionen ausprobieren musste. Es gibt keine anderen Indikatoren als der **bound**, damit der **IDA\*** erkennen kann, dass er anhalten muss, den Zweig zu untersuchen. Der **IDA\*-Algorithmus** spart also zwar Ressourcen, leistet aber im Vergleich zu A\* oft mehr als er sollte.

Aus diesem Grund ist die Verbesserung von IDA\* in der Zeit nicht besser als A\*, obwohl für diesen Algorithmus mithilfe besserer Heuristiken mehr "nutzlose" Knoten entfernt werden als A\*.

### 4.3 A\* und IDA\*

Mit dem vorherigen Ergebnis in 3.2 ist klar, dass **IDA\*** im Durchschnitt mehr Zeit benötigt, um das Puzzle zu lösen, als A\* (ungefähr 3-mal). Jetzt werden die beiden Algorithmus intensiver verglichen mit der schwierigsten Konfiguration von 15-Puzzle.

IDA hat zwar eine höhere Geschwindigkeit beim Erweitern der Knoten als A\*, muss aber aufgrund seiner Mechanik auch mehr Knoten erweitern, weil IDA\* anders als A\* keine Nachschlagetabelle hat, um zu überprüfen, ob der Knoten angetroffen wurde. Genauer gesagt, dass IDA\* die Knoten oft

### 4.3. A\* UND IDA\*

mehr als einmal untersucht, und daher kann dieser die letztendlich benötigte Zeit im Vergleich mit A\* verdoppeln oder verdreifachen.

Das Speicherverbrauchsdiagramm hat gezeigt, dass A\* zwar weniger Knoten expandierte als IDA\* (ungefähr dreimal), aber im Vergleich zu IDA\* immer noch sehr viel Speicher benötigt. Immer mehr Knoten gehen in die **open list** und **closed list** von A\*, bis das Ziel gefunden wurde (einmal fast den gesamten Heap-Speicher verbraucht hat). Wenn eine weniger fortgeschrittene Heuristik verwendet wurde oder die Konfiguration noch schwieriger war, hätte A\* sie möglicherweise nicht mit dem aktuellen maximalen Heap-Speicher gelöst.

Im Gegensatz zu A\* wurde der Speicher direkt proportional zur Tiefe des Suchbaums aufgebaut und der Speicher wurde freigegeben, wenn der Algorithmus zum nächsten Baum geht, daher sinkt der Wert im Diagramm und baut sich dann wieder auf. Dies macht IDA\* in verschiedenen Situationen robuster und zulässiger als A\*.

Mit dieser Beobachtung wird IDA\* eine bessere Wahl zum Lösen der Puzzles, die größer als 15-Puzzle ist.

# 5 Fazit

## 5.1 Forschungsthema und Ergebnisse

Es ist eindeutig, dass 15-Puzzle gelöst wurden, und die Ergebnisse durch verschiedene Optimierungen und Heuristiken sowohl in Zeit als auch in Speicherplatz verbessert werden.

- Am Ende können der optimierte A\*-Algorithmus und die leistungsstarke 7-8-Partition Pattern-Datenbank beliebige Konfigurationen von 15-Puzzle in kurzer Zeit ohne Speicherprobleme auf dem aktuellen Computer lösen.
- Auch der IDA\*-Algorithmus zeigt seine Robustheit und Vorteile, wenn er an die Problemstellung angepasst wird. Obwohl die zum Lösen des Puzzles benötigte Zeit immer noch viel hinter A\* zurückbleibt, wird es mit weiterer Optimierung ein großartiger Kandidat sein, um ein größeres Problem wie 24- oder 35-Puzzle zu lösen.
- **Manhattan Distance** wurde sehr effizient umgesetzt. Obwohl es das schlechteste experimentelle Ergebnis hat, ist es immer noch eine zulässige Heuristik und ein großartiger Vergleichspunkt für die anderen. Aber wenn **Manhattan Distance** mit **Linear Conflict** kombiniert wird, entsteht die beste Verbesserung, die in Abbildung 3.1 leicht realisiert werden kann. Es zeigt, dass die Interaktionen zwischen Kacheln in **n-Puzzle** sehr wichtig sind, um die beste Heuristik zu finden. Dies wird erneut durch die **Additive Pattern Database Heuristics** bewiesen.
- Als weiterer Schwerpunkt der Arbeit wurde das Konzept der **Additive Pattern Database Heuristics** vorgestellt, sowie wie die Datenbanken effizient erstellt werden können, trotz der Herausforderungen, denen man begegnen könnte, um das genaueste heuristische Schätzwerkzeug für das Problem zu werden. Die Verwendung dieser Heuristik mit der größtmöglichen Partitionierung brachte das beste Ergebnis, da die meisten Boards in Sekunden oder höchstens mehreren Minuten gelöst werden konnten.
- Zum Schluss wurden alle Methoden mit verschiedenen Optimierungen verglichen und analysiert. Damit die Strategie (genutzte Algorithmen und Datenstrukturen) bewertet werden kann. Darüber hinaus kann es eine Perspektive auf noch größere Probleme bieten.

## 5.2 Praktische Implikationen

Der gesamte Prozess der Bearbeitung dieser Bachelorarbeit ist auch eine Lernkurve für die Recherche und Bezugnahme auf andere Studien und Arbeiten (Dissertationen) zum Thema. Zusammen mit dem gesammelten Wissen während des Studiums, um das Problem aus eigener Perspektive anzugehen. Dies vertieft unser Verständnis der engen Beziehung zwischen Algorithmen und Datenstrukturen und

### 5.3. AUSBLICK

der Kreativität, die erforderlich ist, um sie anzupassen, um ein bestimmtes Problem so effizient wie möglich zu lösen.

## 5.3 Ausblick

Das Ergebnis hatte die Vorteile des vorberechneten heuristischen Werts nach dem Konzept der **Pattern Database** sowie das unterschiedliche Verhalten der **A\***- und **IDA\***-Algorithmen während des Tests gezeigt. Obwohl **A\*** **IDA\*** in der Leistung übertrifft, gibt es einige Verbesserungen (wie zum Beispiel die Symmetrie des Bretts auszunutzen), die für **IDA\*** vorgenommen werden können, um es wettbewerbsfähiger zu machen.

Es kann davon ausgegangen werden, dass zukünftig die größeren 24- und 35-Puzzle mit **IDA\*** und **Pattern Database Heuristics** gelöst werden. Deshalb ist es wichtig, den **IDA\*** bzw. **Pattern Database Heuristics** zu optimieren, um die größeren Boards von n-Puzzle herauszufordern. Die möglichen sinnvollen Ansätze sind: **Symmetry reduction** [13], **Finite State Machine Pruning** [5] [1] oder **Optimal Partitioning for Pattern Database** [5]

# Literaturverzeichnis

- [1] Hüffner, Falk & Edelkamp, Stefan & Fernau, Henning & Niedermeier, Rolf. (2001). Finding Optimal Solutions to Atomix. 2174. 229-243. 10.1007/3-540-45422-5\_17.
- [2] Burns, Ethan & Hatem, Matthew & Leighton, M.J. & Ruml, Wheeler. (2012). Implementing Fast Heuristic Search Code. Proceedings of the 5th Annual Symposium on Combinatorial Search, SoCS 2012. 3. 25-32. 10.1609/socs.v3i1.18245.
- [3] Korf, Richard & Felner, Ariel. (2002). Disjoint pattern database heuristics. Artificial Intelligence. 134. 9-22. 10.1016/S0004-3702(01)00092-3.
- [4] Felner, Ariel & Korf, Richard & Hanan, Sarit. (2004). Additive Pattern Database Heuristics. J. Artif. Intell. Res. (JAIR). 22. 279-318. 10.1613/jair.1480.
- [5] Robert Clausecker. (2017). Notes on the Construction of Pattern Databases. ZIB-Report (17-59). 1438-0064.
- [6] Java™ Platform, Standard Edition 7 API Specification, (2022). PriorityQueue<E>. [online] Available at: <https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html> [Accessed Nov 1, 2022].
- [7] Felner, Ariel & Zahavi, Uzi & Holte, Robert & Schaeffer, Jonathan & Sturtevant, Nathan & Zhang, Zhifu. (2011). Inconsistent heuristics in theory and practice. Artif. Intell.. 175. 1570-1603. 10.1016/j.artint.2011.02.001.
- [8] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Third Edition.. Artif. Intell.. 175. 93-99. 10.1016/j.artint.2011.01.005.
- [9] Wm. Woolsey Johnson & W. E. Storey. Notes on the 15-puzzle. American Journal of Mathematics, 2:397–404, 1879.
- [10] Korf, R. E. (1985). Depth-First Iterative-Deepening: An Optimal Admissible Tree Search.. Artif. Intell., 27, 97-109.
- [11] Hansson, Othar & Mayer, Andrew & Yung, Moti. (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. Information Sciences. 63. 207-227. 10.1016/0020-0255(92)90070-O.
- [12] Korf, Richard & Taylor, Larry. (1999). Finding Optimal Solutions to the Twenty-Four Puzzle. Proceedings of the National Conference on Artificial Intelligence. 2.
- [13] Culberson, Joseph & Schaeffer, Jonathan. (1996). Searching with Pattern Databases. Advances in Artificial Intelligence. 402-416. 10.1007/3-540-61291-2\_68.