Wissensgewinn aus Spieldatenbanken

Bachelorarbeit

Victor-Philipp Negoescu

Fachbereich Informatik

Fachgebiet Knowledge Engineering

Prof. Johannes Fürnkranz

Wissensgewinn aus Spieldatenbanken

Bachelorarbeit

Eingereicht von Victor-Philipp Negoescu

Tag der Einreichung: 29.07.2013

Technische Universität Darmstadt
Fachbereich Informatik

Fachgebiet Knowledge Engineering (KE)
Prof. Johannes Fürnkranz

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 29. Juli 2013 Victor-Philipp Negoescu

Zusammenfassung

In dieser Arbeit verwenden wir den Kompressionsalgorithmus *Krimp* [15], um Datenbanken von Schachpartien zu komprimieren. Die auf dem *Minimum Description Length* – Prinzip theoretisch fundierte Methode nutzt heuristische Algorithmen, um Itemset-Datenbanken mit Hilfe der aus ihnen extrahierten Frequent Itemsets zu kodieren und reduziert gleichzeitig die Anzahl der benötigten Frequent Itemsets um mehrere Größenordnungen [15].

Wir zeigen, wie man Schachstellungen als Itemsets repräsentiert und stellen neben quantitativen Analysen der von Krimp komprimierten Datenbanken auch ausgewählte resultierende Teilschachstellungen vor. Zudem vergleichen wir Krimp mit state-of-the-art Kompressionsalgorithmen und liefern Hinweise auf mögliche Ansatzpunkte zur Performanzverbesserung.

Abstract

In this work we use the compression algorithm *Krimp* [15] to compress a database of chess positions. While theoretically founded on the *Minimum Description Length* principle, the method uses heuristic algorithms to encode item set databases by their extracted frequent item sets and at the same time reduces the number of necessary frequent item sets by multiple orders of magnitude [15].

We show how to represent chess positions as item sets and present quantitative analysis of the compressed databases as well as hand-picked resulted partial chess positions. Moreover, we compare Krimp to state-of-the-art compression algorithms and suggest possible starting points to improve its performance.

Inhaltsverzeichnis

Zusammenfassung **iii**

1 Einleitung **1**
1.1 Motivation & Zielsetzung 1
1.2 Gliederung 2

2 Grundlagen **3**
2.1 Schach 3
2.1.1 Aktuelle Lösungsansätze 7
2.1.2 Schachdatenbanken 9
2.2 Data Mining 11
2.2.1 Informationstheorie 11
2.2.2 Frequent Itemset Mining 15
2.3 Krimp 17
2.3.1 Grundprinzip 17
2.3.2 Generierung der Codetabelle & Kompression der Datenbank 17
2.3.3 Wahl des Minimum Supports 23
2.4 Algorithmen & Methoden 24
2.4.1 Konvertierung von Schachpartien zu –stellungen 24
2.4.2 Extraktion von Abschnitten einer Partie 24
2.4.3 Extraktion der Bauernstruktur 25
2.4.4 Konvertierung von Schachstellungen zu Itemsets 26
2.4.5 Filterung nach Spielergebnis 27

3 Experimente **28**
3.1 Experiment 1.1 31
3.1.1 Konfiguration 31
3.1.2 Quantitative Analyse 31
3.1.3 Ausgewählte Teilstellungen 32
3.1.4 Analyse der Kompression 33
3.1.5 Beobachtungen 34
3.2 Experiment 1.2 35
3.2.1 Konfiguration 35
3.2.2 Quantitative Analyse 35
3.2.3 Ausgewählte Teilstellungen 36
3.2.4 Analyse der Kompression 38
3.2.5 Beobachtungen 38

3.3 Experiment 1.3 ... 39
3.3.1 Konfiguration ... 39
3.3.2 Quantitative Analyse ... 39
3.3.3 Ausgewählte Teilstellungen ... 40
3.3.4 Analyse der Kompression ... 41
3.3.5 Beobachtungen ... 41
3.4 Experiment 2.1 ... 42
3.4.1 Konfiguration ... 42
3.4.2 Quantitative Analyse ... 42
3.4.3 Ausgewählte Teilstellungen ... 43
3.4.4 Analyse der Kompression ... 44
3.4.5 Beobachtungen ... 44
3.5 Experiment 2.2 ... 45
3.5.1 Konfiguration ... 45
3.5.2 Quantitative Analyse ... 45
3.5.3 Ausgewählte Teilstellungen ... 46
3.5.4 Analyse der Kompression ... 47
3.5.5 Beobachtungen ... 47
3.6 Experiment 2.3 ... 48
3.6.1 Konfiguration ... 48
3.6.2 Quantitative Analyse ... 48
3.6.3 Ausgewählte Teilstellungen ... 49
3.6.4 Analyse der Kompression ... 52
3.6.5 Beobachtungen ... 52
3.7 Experiment 2.4 ... 53
3.7.1 Konfiguration ... 53
3.7.2 Quantitative Analyse ... 53
3.7.3 Ausgewählte Teilstellungen ... 54
3.7.4 Analyse der Kompression ... 55
3.7.5 Beobachtungen ... 55
3.8 Experiment 3.1 ... 56
3.8.1 Konfiguration ... 56
3.8.2 Quantitative Analyse ... 56
3.8.3 Ausgewählte Teilstellungen & Beobachtungen ... 58
3.8.4 Analyse der Kompression ... 62
3.9 Experiment 3.2 ... 63
3.9.1 Konfiguration ... 63
3.9.2 Quantitative Analyse ... 63
3.9.3 Ausgewählte Teilstellungen & Beobachtungen ... 65
3.9.4 Analyse der Kompression ... 68
3.10 Experiment 3.3 ... 69

3.10.1 Konfiguration 69
3.10.2 Quantitative Analyse 69
3.10.3 Ausgewählte Teilstellungen & Beobachtungen 71
3.10.4 Analyse der Kompression 75

4 Realisierung **76**
4.1 Responsibilities 77
4.1.1 Paket *tud.chess* 77
4.1.2 Paket *tud.chess.abstraction* 77
4.1.3 Paket *tud.chess.abstractiongenerator* 77
4.1.4 Paket *tud.chess.game* 78
4.1.5 Paket *tud.chess.krimp* 78
4.1.6 Paket *tud.chess.krimp.abstractionreverter* 78
4.1.7 Paket *tud.chess.krimp.heatmap* 79
4.1.8 Paket *tud.chess.krimp.mirroredanalyzation* 79
4.1.9 Paket *tud.chess.util* 79
4.1.10 Paket *tud.chess.visualization* 79
4.2 Datenformate & -pfade 80
4.2.1 Dateiformat „Itemset-Datenbank“ 80
4.2.2 Dateiformat „Krimp-Itemset-Datenbank“ 80
4.2.3 Dateiformat „Krimp Codetabelle“ 81
4.2.4 Dateiformat „Krimp Datenbankanalyse“ 81
4.2.5 Dateiformat „Itemset-Frequenzanalyse“ 82
4.2.6 Dateiformat „Sortierte Itemset-Frequenzanalyse“ 82
4.2.7 Dateiformat „Gespiegelte Itemset-Analyse“ 83
4.2.8 Illustration des Datenpfads 84
4.3 Externe Programme & Bibliotheken 85
4.4 Einrichtung & Verwendung 86
4.4.1 Tool *PGNtoGameStateParser* 86
4.4.2 Tool *CodeTableRenderer* 87
4.4.3 Tool *ItemsetFrequencyAnalyzer* 88
4.4.4 Tool *ItemsetFrequencyRenderer* 88
4.4.5 Tool *ItemsetFrequencySorter* 89
4.4.6 Tool *MirroredChessPiecePositionAnalyzer* 89
4.4.7 Tool *HeatmapGenerator* 89

5 Resümee **90**
5.1 Frequent Itemset Mining 90
5.2 Kompression 91
5.3 Ausblick 92

Abbildungsverzeichnis **93**

Tabellenverzeichnis **97**

Listingverzeichnis **99**

Literatur- & Quellenverzeichnis **100**

Gewidmet an: *Prinzessin Rosalinde von der Knatteralm*

1 Einleitung

1.1 Motivation & Zielsetzung

Möchte man Algorithmen entwickeln, die für (Gesellschafts-)spiele, bei denen die Aktionen der Spieler das Spielergebnis maßgeblich beeinflussen, zu jeder möglichen Spielsituation die „richtige“ Aktion ausgeben, gelangt man bei komplexeren Spielen schnell an eine Grenze, an der die Anzahl der möglichen Spielsituationen so hoch ist, dass es nicht mehr möglich ist, die gewinnbringende Aktion für jede Spielsituation explizit abzuspeichern und in adäquater Zeit anzugeben. Da man gezwungen ist, den Spielbaum (siehe Abbildung 11) abzuschneiden, ist auch nicht mehr gewährleistet, dass immer die optimale Entscheidung gefunden wird.

Bei der Wahl, welche Abschnitte des Suchbaums verfolgt werden und welche verworfen werden, bedient man sich Heuristiken, die zwar kein optimales Ergebnis garantieren, jedoch bei der Auswahl der zu untersuchenden Züge hilfreich sein können. Wird im Schach beispielsweise die eigene Dame bedroht, sollte man im Suchbaum wahrscheinlich eher die möglichen Züge untersuchen, durch die die Dame gerettet werden kann, da sich ihr Verlust gewöhnlich nicht oder nur sehr schwer ausgleichen lässt.

Eine weitere Möglichkeit könnte in diesem Zusammenhang die statistische Untersuchung von vergangenen Spielen darstellen. Untersucht man eine Vielzahl an gespielten Partien, lassen sich aus ihnen womöglich Spielsituationen extrahieren, die häufig zum Sieg eines Spielers führen. Ein Algorithmus, der basierend auf der aktuellen Spielsituation die Auswahl des nächsten Spielzugs durchführt, könnte anschließend durch Heuristiken erweitert werden, die die gewonnenen Informationen in die Entscheidungsauswahl einfließen lassen.

Um Spielsituationen aus großen Datenmengen zu extrahieren, existieren bereits unterschiedliche Verfahren. Diese Bachelorarbeit befasst sich mit der Extraktion häufiger Teilstellungen aus Schachdatenbanken und verwendet dafür Krimp [15] – einen Kompressionsalgorithmus, der Itemset-Datenbanken mit Hilfe der vorkommenden Items und Itemsets komprimiert.

Im Rahmen dieser Arbeit wurden folgende Punkte bearbeitet:

1. die Extraktion von Schachpartien aus Schachpartiedatenbanken
2. die Konvertierung von Schachpartien in Schachstellungen
3. die Anwendung des Krimp-Algorithmus auf die konvertierten Schachstellungen
4. die quantitative Analyse der von Krimp ausgegebenen Teilstellungen
5. die selektive qualitative Analyse der resultierenden Teilstellungen
6. die Entwicklung von Algorithmen und Methoden zur Vor- und Nachbereitung der Daten

Auf sie wird in den folgenden Kapiteln eingegangen.

1.2 Gliederung

Kapitel 2

In Kapitel 2 stellen wir die benötigten Grundlagen und vorbereitenden Maßnahmen zur Durchführung der Experimente vor:

Abschnitt 2.1 beschreibt die Domäne des Schachs. Es werden kurz die Regeln des Schachspiels erläutert und auf die grundsätzliche Vorgehensweise beim maschinellen Lösen des Schachproblems eingegangen.

Abschnitt 2.2 umfasst wesentliche Theorien zur Informationsrepräsentation, -verarbeitung und -extraktion. Des Weiteren werden Methoden beschrieben, die häufige Muster aus Datenbanken extrahieren.

Abschnitt 2.3 beschreibt die mathematischen Grundlagen des Krimp-Algorithmus.

Abschnitt 2.4 stellt die entwickelten Methoden vor, die die Schachpartien in -stellungen umwandeln, Vorverarbeitungsschritte auf den Schachstellungen durchführen und die von Krimp gelieferten Ergebnisse analysieren und illustrieren.

Kapitel 3

Nach der Erläuterung der verwendeten Konfigurationen werden in Kapitel 3 die durchgeführten Experimente beschrieben und die quantitativen und qualitativen Analysen vorgestellt.

Kapitel 4

Kapitel 4 beschreibt die Implementierung der in Abschnitt 2.4 definierten Methoden. Außerdem erfolgt dort die Definition der benötigten Dateiformate und die Einordnung der entwickelten Toolchain und des Krimp-Algorithmus in den gesamten Datenpfad. Abschließend werden Hinweise zur Verwendung der Programme gegeben.

Kapitel 5

Kapitel 5 enthält ein Resümee, abschließende quantitative Vergleiche und einen Ausblick.

2 Grundlagen

2.1 Schach

Spielregeln

Schach ist ein strategisches Brettspiel für zwei Spieler. Das quadratische Spielfeld besteht aus 64 Feldern, die in horizontaler Richtung mit den Buchstaben A bis H und in vertikaler Richtung mit den Zahlen 1 bis 8 gekennzeichnet sind. Auf dem Schachbrett befinden sich in der Grundstellung 32 Schachfiguren der zwei Spielerfarben weiß und schwarz gemäß Abbildung 1.

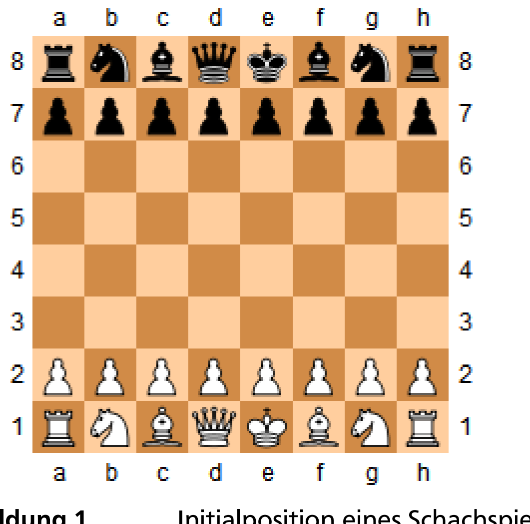
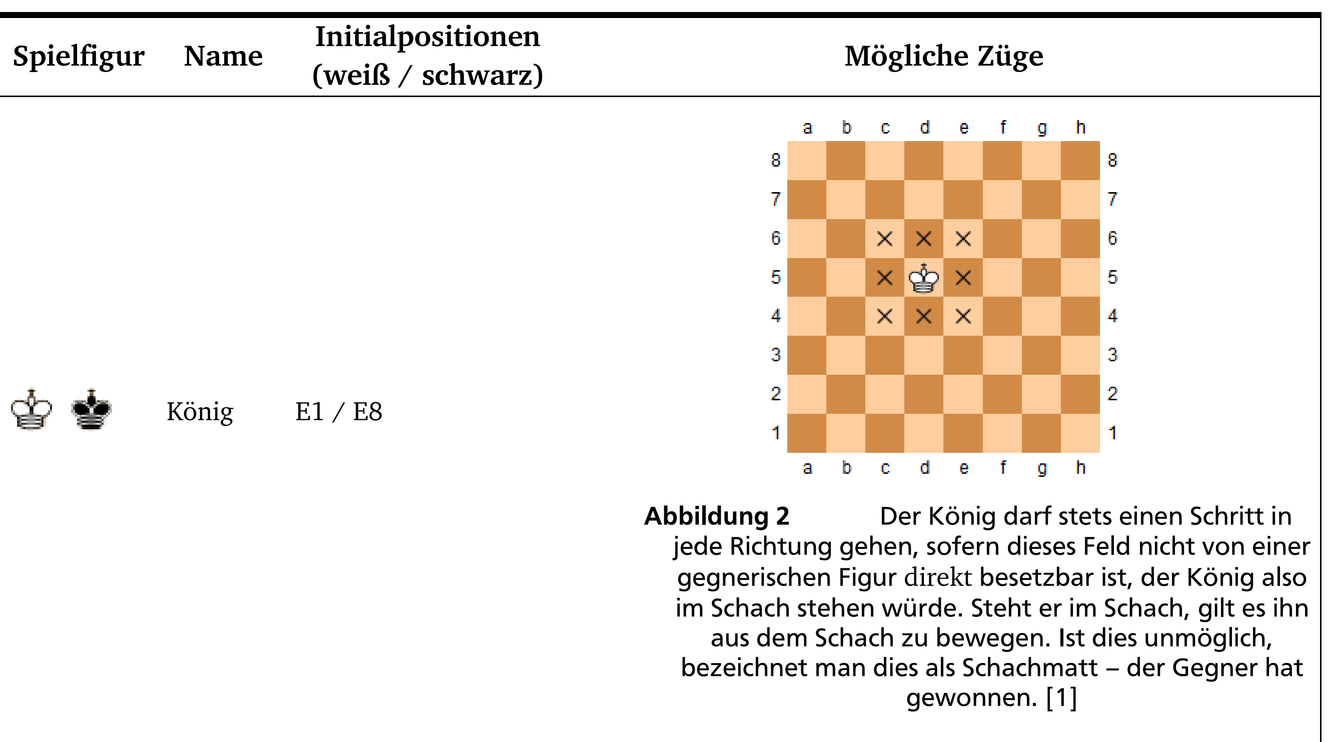
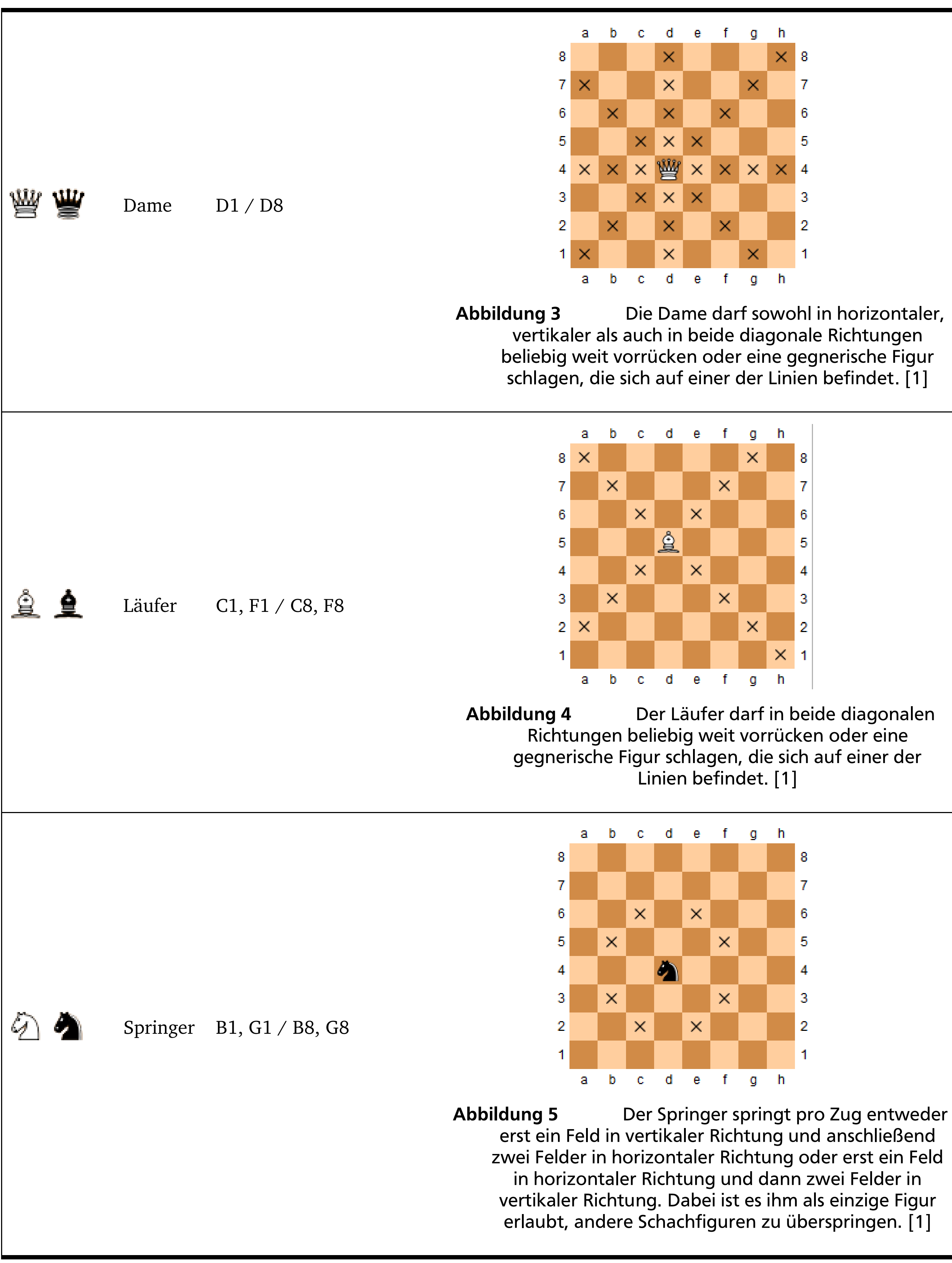


Abbildung 1 Initialposition eines Schachspiels [1]

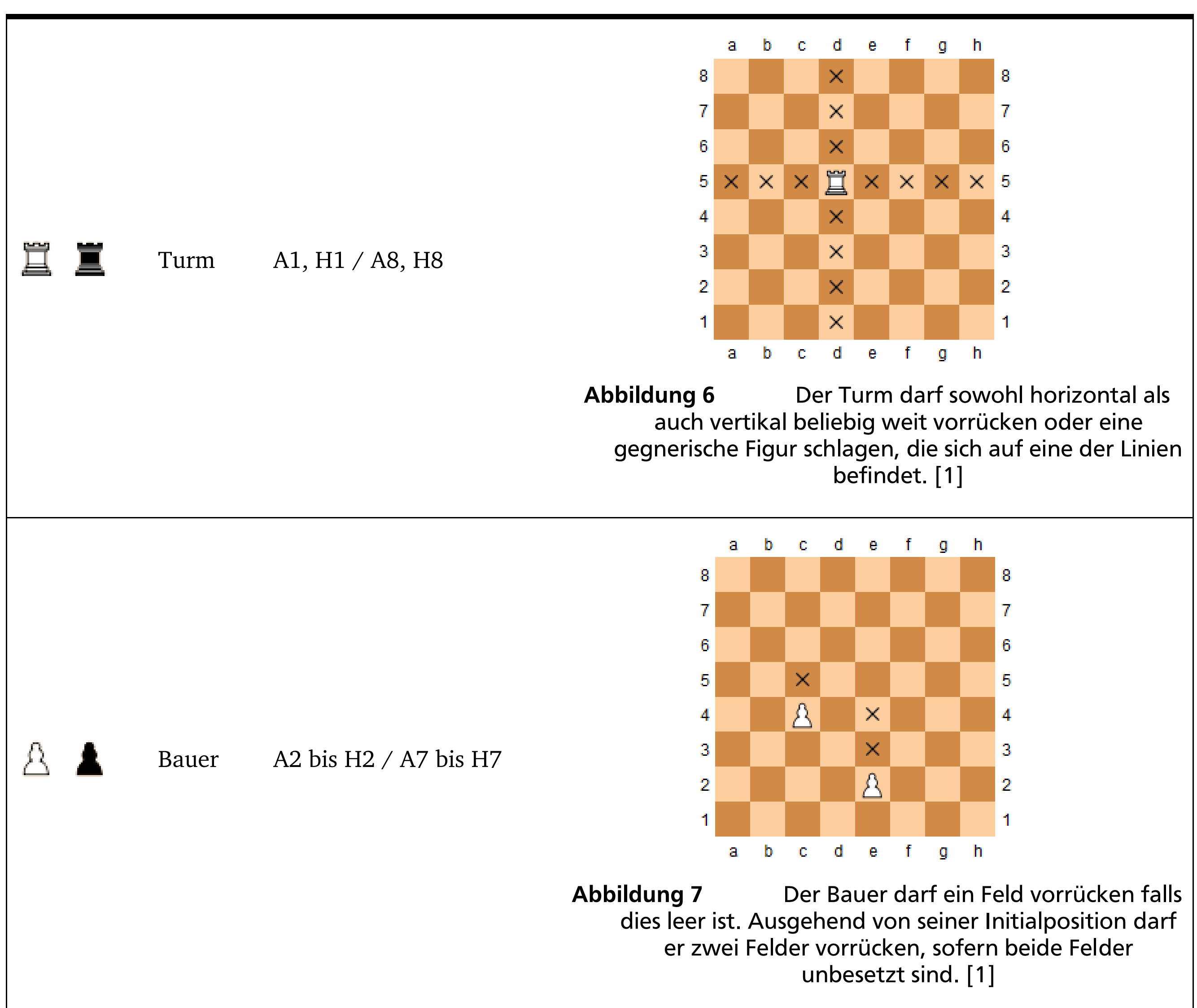
Jede Figur besitzt einen Satz an möglichen Zügen, die sie ausführen kann, wobei nur der Springer ♘ andere Figuren überspringen darf.



Spielfigur	Name	Initialpositionen (weiß / schwarz)	Mögliche Züge
♔ ♚	König	E1 / E8	**Abbildung 2** Der König darf stets einen Schritt in jede Richtung gehen, sofern dieses Feld nicht von einer gegnerischen Figur direkt besetzbar ist, der König also im Schach stehen würde. Steht er im Schach, gilt es ihn aus dem Schach zu bewegen. Ist dies unmöglich, bezeichnet man dies als Schachmatt – der Gegner hat gewonnen. [1]



Figur	Name	Startfelder	Zugmöglichkeiten
	Dame	D1 / D8	**Abbildung 3** Die Dame darf sowohl in horizontaler, vertikaler als auch in beide diagonale Richtungen beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf einer der Linien befindet. [1]
	Läufer	C1, F1 / C8, F8	**Abbildung 4** Der Läufer darf in beide diagonalen Richtungen beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf einer der Linien befindet. [1]
	Springer	B1, G1 / B8, G8	**Abbildung 5** Der Springer springt pro Zug entweder erst ein Feld in vertikaler Richtung und anschließend zwei Felder in horizontaler Richtung oder erst ein Feld in horizontaler Richtung und dann zwei Felder in vertikaler Richtung. Dabei ist es ihm als einzige Figur erlaubt, andere Schachfiguren zu überspringen. [1]



♖ ♜	Turm	A1, H1 / A8, H8	**Abbildung 6** Der Turm darf sowohl horizontal als auch vertikal beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf eine der Linien befindet. [1]
♙ ♟	Bauer	A2 bis H2 / A7 bis H7	**Abbildung 7** Der Bauer darf ein Feld vorrücken falls dies leer ist. Ausgehend von seiner Initialposition darf er zwei Felder vorrücken, sofern beide Felder unbesetzt sind. [1]

Tabelle 1 Auflistung aller Schachfiguren und möglicher Züge

Im Gegensatz zu anderen Figuren können Bauern keine gegnerischen Figuren schlagen, die sich auf den Bewegungslinien des Bauern befinden. Möchte ein Bauer eine Figur schlagen, so ist dies nur in zwei Fällen möglich:

1. Die gegnerische Figur befindet sich auf einem der Nachbarfelder des in Bewegungsrichtung des Bauern angrenzenden Felds (siehe Abbildung 8). Der eigene Bauer schlägt die gegnerische Figur und nimmt den Platz auf ihrem Feld ein.
2. Die gegnerische Figur ist ebenfalls ein Bauer und ist im vorausgehenden Zug ausgehend von seiner Initialposition um zwei Felder vorgerückt. Der gegnerische Bauer muss hierbei auf einem horizontalen Nachbarfeld des eigenen Bauers stehen. Der eigene Bauer schlägt den gegnerischen Bauer „en passant“, indem er gemäß Abbildung 9 einen Schritt in Bewegungsrichtung und einen horizontalen Schritt zur Seite des gegnerischen Bauers durchführt.

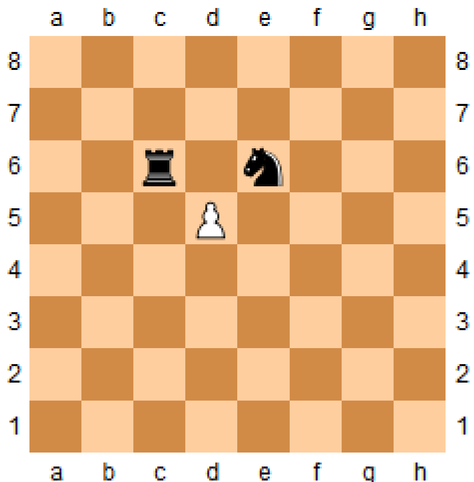


Abbildung 8 Der weiße Bauer darf den schwarzen Turm oder Springer schlagen. [1]

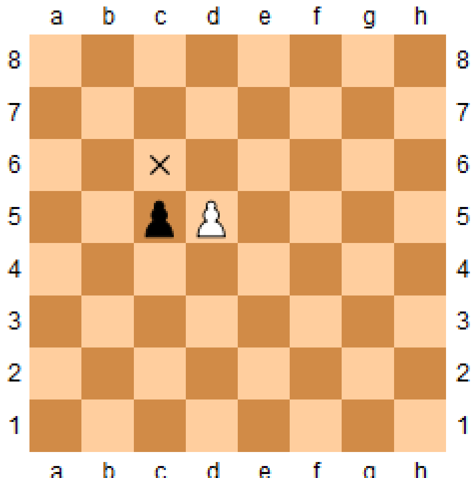


Abbildung 9 Der weiße Bauer schlägt den schwarzen Bauern „en passant“, in dem er auf das gekennzeichnete Feld vorrückt. [1]

Der weiße Spieler beginnt stets eine Partie, die entweder damit endet, dass

- ein Spieler seinen König nicht mehr bewegen kann, um einer Schachsituation zu entkommen (sog. „Schachmatt“) oder
- kein Spieler mehr einen gültigen Zug ausführen kann, ohne den eigenen König in Schach zu setzen („Patt“) oder
- kein Spieler mehr den anderen schachmatt stellen kann („Remis“).

Spielphasen

Ein Schachspiel kann grob in drei Spielabschnitte unterteilt werden: Eröffnung, Mittelspiel und Endspiel.

Während die Spieler in der Eröffnung versuchen, ihre Figuren in das Zentrum des Schachfelds hinein zu bewegen und infolgedessen eine dominanten Position einzunehmen, werden im Mittelspiel Schwächen beim Gegner provoziert, um ihn im Endspiel möglichst schachmatt zu setzen. [1]

Bauernstrukturen

Entfernt man von einer Schachstellung alle Figuren bis auf die Bauern und Könige, erhält man als Ergebnis die charakteristische Bauernstruktur dieser Stellung (siehe Abbildung 10). Je nach auftretender Bauernstruktur – so

die Annahme – positionieren die Spieler ihre restlichen Figuren in typischen Mustern. Im Rahmen dieser Arbeit wurde unter anderem nach ebendiesen Mustern gesucht.

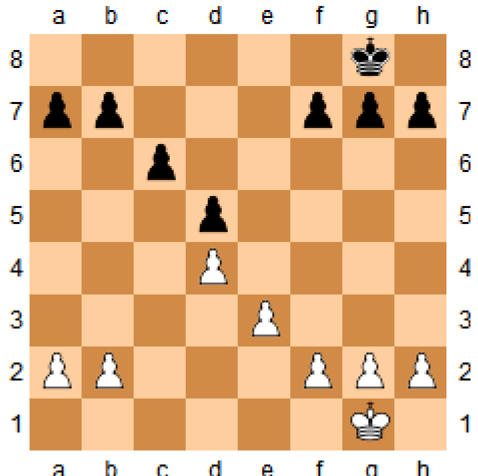


Abbildung 10 Charakteristische Bauernstruktur einer Schachstellung

Menschliche Rezeption von Schachstellungen

In seiner Studie von 1965 untersuchte De Groot, warum erfahrene Spieler sich Schachstellungen, die sie nur einige wenige Sekunden zu Gesicht bekamen, mit größerer Wahrscheinlichkeit und Genauigkeit merken konnten als unerfahrene oder Gelegenheitsspieler. Er vermutete, dass professionelle Schachspieler sich nicht einzelne Figuren einprägten, sondern sich oft auftretende Konstellationen von Schachfiguren – sogenannte *Chunks* – gemerkt hatten, mit deren Hilfe sich die gezeigten Schachstellungen wesentlich schneller einprägen und diese anschließend wiedergeben ließen. Vergleicht man dieses Bild mit der menschlichen Sprachrezeption, merken erfahrene Spieler sich im Gegensatz zu unerfahrenen Spielern die Wörter und Sätze eines Textes und nicht die einzelnen Buchstaben, da sie sich bereits häufige Redewendungen und Wortkombinationen im Vorfeld bereits eingeprägt haben und auf diese später nur noch zugreifen müssen, um einen neuen Text wiederzugeben.

Diese Vermutung unterstrich De Groot mit einem weiteren Experiment, in dem er den Probanden willkürliche Schachstellungen präsentierte, die nicht im Zuge eines Schachspiels zustande gekommen waren. Durch die „Unnatürlichkeit“ der Stellungen hatten die erfahrenen Spieler plötzlich keinen Vorteil mehr gegenüber den unerfahrenen Spieler, ihre Werte für Wiedergabewahrscheinlichkeit und Präzision näherten sich stark denen der unerfahrenen Spieler an.

Ausgehend von der Vermutung, dass Schachspieler sich mit zunehmender Spielstärke gemäß den Annahmen der *Chunkingtheorie* verhalten und dies möglicherweise zur Folge hat, dass diese Spieler selbst in Partien bekannte Chunks durch entsprechende Zugfolgen bilden, könnten diese im Rahmen einer Untersuchung über große Mengen von Schachpartien zum Vorschein kommen. [2]

ECO-Schlüssel

Eröffnungen können durch sogenannte ECO-Schlüssel (*Encyclopedia of Chess Openings*) kategorisiert werden. Jede der fünf ECO-Hauptgruppen (Buchstaben A bis E) ist dabei in einhundert Nebengruppen (00-99) unterteilt, wobei jede Nebengruppe eine spezifische Variante der durch die Hauptgruppe spezifizierten Eröffnung darstellt. [3]

2.1.1 Aktuelle Lösungsansätze

Betrachtet man Schach als informationstheoretisches Problem, stellt sich die Frage, wie ein Algorithmus aufgebaut sein müsste, um Erfolg versprechend Schach spielen zu können. Ausgehend von aktuellen Annahmen existieren ungefähr $2{,}28 * 10^{46}$ erreichbare Zustände (durch reguläres Spiel) [4]. Der naive Ansatz, ausgehend von der aktuellen Stellung alle möglichen nachfolgenden Züge sowohl des Gegners als auch der eigenen als Spielbaum

darzustellen und genau den Zug auszuwählen, der den Erfolg des Gegners gemäß der Menge an erreichbaren Endknoten (Stellungen nach Beendigung der Partie durch Schachmatt oder Patt) minimiert, scheitert an der Größe dieser Zustandsmenge.

Suche auf Mehrwegbäumen

Aktuelle Lösungsansätze (Löser) verwenden meist spezialisierte Suchverfahren auf Mehrwegbäumen. Jeder Baumknoten repräsentiert wie beim naiven Ansatz eine Schachstellung, die durch den entsprechenden Schachzug des Spielers, der unmittelbar vor dem Erreichen dieser Stellung am Zug war, hergestellt werden kann. Als Wurzelknoten wird die Schachstellung eingesetzt, von der ausgehend der nächste (beste) Zug ermittelt werden soll. Zusammengefasst besagt diese Konstruktionsregel, dass die einzelnen Knotenebenen alle Schachstellungen beinhalten, die abwechselnd jeweils vom Gegner und vom Löser durch einen gültigen Schachzug erreicht werden können und dass die Ebene unmittelbar unter dem Wurzelknoten alle nächsten erzeugbaren Schachstellungen des Lösers enthält. Entgegen dem naiven Ansatz werden bei aktuellen Verfahren nicht alle Wege bis zu den Endknoten durchsucht, sondern nur durch Teile des Suchbaums iteriert.

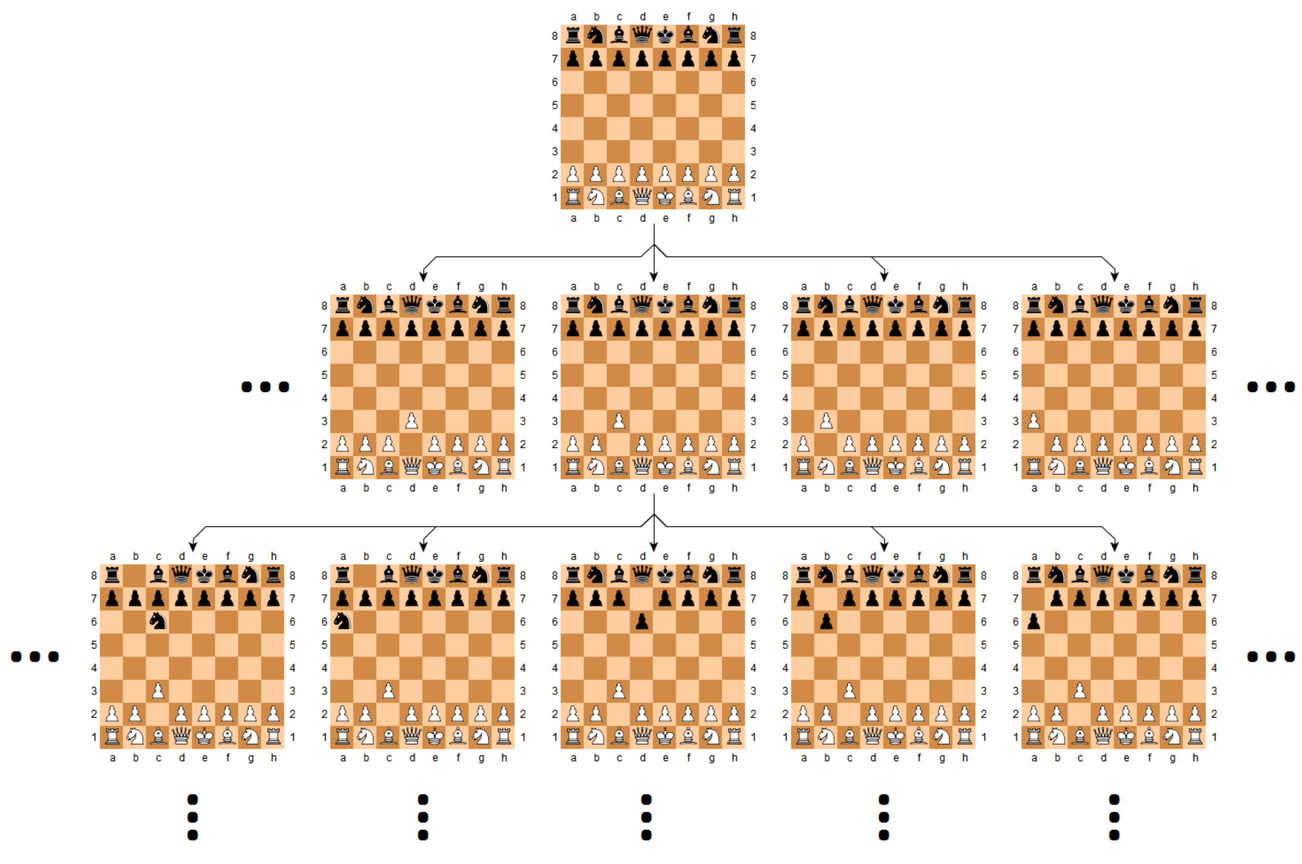


Abbildung 11 Auszug aus einem Suchbaum von Schachstellungen

Bewertungsfunktion

Eine Bewertungsfunktion klassifiziert jede Schachstellung des Suchbaums an Hand verschiedener Faktoren. Hierbei werden unter anderem materielle (wie viele Figuren befinden sich noch auf dem Brett?) als auch positionelle (welche Figuren werden bedroht oder bedrohen gegnerische Figuren?) Komponenten bewertet, die sich in einer Gesamtbewertung für die jeweilige Stellung gewichtet wiederfinden. Anschließend wird das Vorzeichen dieser Gesamtbewertung, je nachdem ob es sich bei dem Zug um einen gegnerischen oder um einen eigenen handelt, umgekehrt. So erhalten Stellungen, die für den Gegner besonders vorteilhaft erscheinen eine kleinere Bewertung als Stellungen, die für das Spiel des Lösungsalgorithmus von Vorteil sind. [4]

Pfadreduktion

Durch den exponentiellen Anstieg der Knoten mit zunehmender Tiefe, bedient man sich im Gegensatz zum naiven Ansatz verschiedener Verfahren sowie Heuristiken, um die Anzahl der zu untersuchenden Pfade einzuschränken. Beispiele sind die Alpha-Beta- oder die Monte-Carlo-Suche [5].

2.1.2 Schachdatenbanken

Der voranschreitende Einsatz von Informationstechnik und Kommunikationstechnologien wie dem Internet ermöglichen es, gespielte Schachpartien zu digitalisieren und einer breiten Interessengemeinschaft zur Verfügung zu stellen. Im Laufe der Jahre etablierte sich eine Vielzahl an Webseiten, die ihren Besuchern Schachdatenbanken zur Verfügung stellen. Selbst Partien von Schachgroßmeistern, die weit vor der Erfindung des Internets gespielt worden sind, sind dort zu finden und herunterzuladen.

Neben ihrem Verwendungszweck als Anschauungsmaterial für Schachinteressierte bieten sich die in digitaler Form abgespeicherten Partien auch als Datengrundlage für Untersuchungen innerhalb des Forschungsgebiets *Data Mining* an.

Die Portable Game Notation

Das gebräuchlichste – weil offene und leicht zu verarbeitende – Format ist die *Portable Game Notation* (kurz: *PGN*). Das PGN-Format ist sowohl leicht lesbar als auch durch Zusatzinformationen wie den Namen der Spieler oder dem Ort und Datum der Austragung erweiterbar und leicht zu parsen. Des Weiteren existiert eine Vielzahl an Programmen, die die Speicherung von Schachpartien im PGN-Format unterstützen.

```
[Event "IBM Kasparov vs. Deep Blue Rematch"]
[Site "New York, NY USA"]
[Date "1997.05.11"]
[Round "6"]
[White "Deep Blue"]
[Black "Kasparov, Garry"]
[Opening "Caro-Kann: 4...Nd7"]
[ECO "B17"]
[Result "1-0"]

1.e4 c6 2.d4 d5 3.Nc3 dxe4 4.Nxe4 Nd7 5.Ng5 Ngf6 6.Bd3 e6 7.N1f3 h6
8.Nxe6 Qe7 9.O-O fxe6 10.Bg6+ Kd8 {Kasparov schüttelt kurz den Kopf}
11.Bf4 b5 12.a4 Bb7 13.Re1 Nd5 14.Bg3 Kc8 15.axb5 cxb5 16.Qd3 Bc6
17.Bf5 exf5 18.Rxe7 Bxe7 19.c4 1-0
```

Listing 1 Beispiel einer Schachpartie im PGN-Format [6]

Datenquelle

Im Rahmen dieser Arbeit wurde auf die Schachdatenbank *ICOfY Base 2011.1* [7] von Ingo Schwarz zurückgegriffen. Sie beinhaltet insgesamt 4.234.538 Partien, die nach ihren spezifischen ECO-Codes in fünf Dateien eingeordnet sind.

Datenbank	Partien
ECO A	904.333
ECO B	1.355.566
ECO C	831.996
ECO D	653.882
ECO E	488.761

Tabelle 2 Enthaltene Dateien in der ICOfY Base 2011.1 – Datenbank

2.2 Data Mining

Die grundlegende Aufgabe von Data Mining liegt darin, aus einer großen Datenmenge die relevanten Informationen zu extrahieren. Was relevant ist, hängt hierbei stark vom betrachteten Kontext ab. Das an dieser Stelle gerne hervorgebrachte Zitat

> *„We are drowning in data but starving for knowledge."* [8]

von John Naisbitt aus seinem Buch „Megatrends" (1982) beschreibt passend das Problem, das das Forschungsgebiet *Data Mining* adressiert: Die Suche nach geeigneten Methoden, um diese relevante Informationen zu erhalten und entsprechend aufzubereiten. Selbst aktuelle Methoden liefern teilweise nur mäßige Ergebnisse oder sind zwar auf Modellprobleme anwendbar, versagen jedoch bei größeren Datenmengen. Probleme, die auch im Rahmen dieser Arbeit aufgetaucht sind.

2.2.1 Informationstheorie

Symbole, Wörter & Sprachen

Möchte man Informationen in einem mathematischen System repräsentieren, bietet es sich an, für jede *Informationseinheit* ein *Symbol* zu definieren, welches diese repräsentiert. Im Schach könnten so beispielsweise die Schachfiguren (König, Dame, Läufer, Springer, Turm und Bauer), die Spielerfarben (schwarz, weiß) und die Koordinaten auf dem Schachbrett (A1-H8) die Informationseinheiten darstellen und somit durch individuelle Symbole kodiert werden. Die Menge aller definierten Symbole eines abgeschlossenen Informationssystems bezeichnet man als *Alphabet* (üblicherweise durch ein $\boldsymbol{\Sigma}$ notiert).

Definition 1 Sei $\boldsymbol{P}$ das Alphabet über Symbole von Schachfiguren: $\boldsymbol{P} = \{K, Q, B, N, R, P\}$ (K = König, Q = Dame, B = Läufer, N = Springer, R = Turm, P = Bauer).

Definition 2 Sei $\boldsymbol{C}$ das Alphabet über Symbole von Spielerfarben: $\boldsymbol{C} = \{b,\ w\}$ (b = schwarz, w = weiß)

Definition 3 Sei $\boldsymbol{H}$ das Alphabet über Symbole von horizontalen Koordinaten:

$$\boldsymbol{H} = \{a, b, c, d, e, f, g, h\}$$

Definition 4 Sei $\boldsymbol{V}$ das Alphabet über Symbole von vertikalen Koordinaten:

$$\boldsymbol{V} = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

Ein Alphabet, das fähig ist zu beschreiben, dass eine Figur einer bestimmten Farbe auf einem Schachfeld steht, ist definiert durch die Vereinigung der soeben definierten Unteralphabete.

Definition 5 Sei $\boldsymbol{\Sigma}$ das Alphabet über Symbole von Schachfiguren, Spielerfarben und Koordinaten:

$$\boldsymbol{\Sigma} = \boldsymbol{P} \cup \boldsymbol{C} \cup \boldsymbol{H} \cup \boldsymbol{V}$$

Symbolfolgen, also die Konkatenation von einzelnen Symbolen, werden durch die Definition der *Kleenschen Hülle* (sowohl *Sternhülle* als auch *positive Hülle*) ermöglicht.

Definition 6 Sei Σ ein Alphabet. Sei weiterhin Σ^n definiert als das Tupel der Länge $n \in \mathbb{N}$ über dem Alphabet Σ mit $\Sigma^n = \prod_{i=0}^{n} \Sigma$ und $\Sigma^0 = \{\varepsilon\}$. ε bezeichne das leere Wort.
Dann ist die Sternhülle Σ^* wie folgt definiert:

$$\Sigma^* = \bigcup_{i \in \mathbb{N}_0} \Sigma^i$$

Die positive Hülle Σ^+ ist definiert durch

$$\Sigma^+ = \bigcup_{i \in \mathbb{N}} \Sigma^i = \Sigma^* \setminus \{\varepsilon\}$$

Im Gegensatz zur positiven Hülle enthält die Sternhülle somit zusätzlich das leere Wort ε. [9]

Definition 7 $|t|$ bezeichnet die Länge einer Symbolfolge mit $t \in \Sigma^*$.

Die Konkatenation von Symbolen beschreiben wir wie folgt:

Definition 8 Seien $u, v \in \Sigma^1$, also Symbole des Alphabets Σ oder das leere Wort. Dann ist $\circ$ der Infixoperator der Konkatenation.
Für $u \neq \varepsilon$ und $v \neq \varepsilon$ gilt: $(u \circ v) \in \Sigma^2 \backslash \Sigma^1$.
Für $u = \varepsilon$ und $v \neq \varepsilon$ gilt: $(u \circ v) = (\varepsilon \circ v) = v \in \Sigma^1$.
Für $u \neq \varepsilon$ und $v = \varepsilon$ gilt: $(u \circ v) = (u \circ \varepsilon) = u \in \Sigma^1$.
Für $u = \varepsilon$ und $v = \varepsilon$ gilt: $(u \circ v) = (\varepsilon \circ \varepsilon) = \varepsilon \in \Sigma^0$.

Dies führt zur Generalisierung der Konkatenation für Symbolfolgen aus Σ^*:

Definition 9 Seien $a, b \in \Sigma^*$ Symbolfolgen, $a_1 \in \Sigma, \dots, a_{|a|} \in \Sigma$ und $b_1 \in \Sigma, \dots, b_{|b|} \in \Sigma$ die in den Symbolfolgen enthaltenen Symbole. Dann gilt: $c = (a \circ b) = (a_1 \circ \dots \circ a_{|a|} \circ b_1 \circ \dots \circ b_{|b|}) \in \Sigma^*$ mit $|c| = |a| + |b|$.

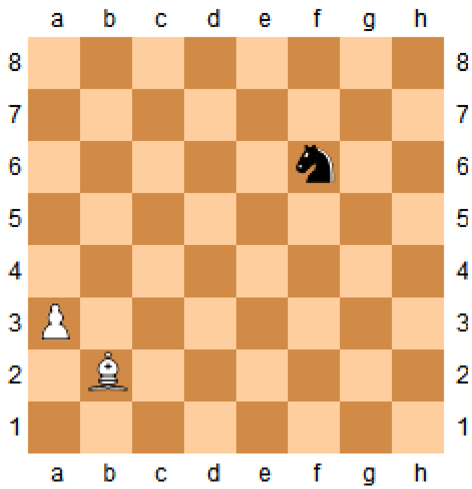


Abbildung 12 Teilstellung einer Schachstellung

Die in Abbildung 12 dargestellte Teilstellung könnte durch die Symbolfolge „Bwb2 Pwa3 Nbf6" beschrieben werden. Anzumerken ist, dass die Leerzeichen allein der Leserlichkeit dienen und nicht Bestandteil der Symbolfolge sind. Auch andere Reihenfolgen und die Vertauschung von Symbolen wären denkbar. So würde etwa die Folge „wKgbw8BR", die aus den Symbolen des Alphabets $\boldsymbol{\Sigma}$ gebildet wurde, zwar eine gültige Symbolfolge im Sinne von $\boldsymbol{\Sigma}^+$ sein, jedoch im Kontext des Modells von Schachfiguren bestimmter Farbe die auf definierten Feldern stehen, keine geordnete wiederverwendbare Information liefern.

Die nötige Ordnung – eine *Syntax* – liefert eine *formale Grammatik*. Die in ihr definierten Produktionsregeln bilden Symbolfolgen (*Wörter*), die nur innerhalb einer wohldefinierten Teilmenge der Kleenschen Hülle liegen. Diese Teilmenge $L \subseteq \Sigma^*$ wird als *formale Sprache* bezeichnet.

Definition 10 Sei L eine wohldefinierte Teilmenge aus Σ^*. Dann bezeichnet L eine *formale Sprache* über das Alphabet Σ. [10]

Definition 11 Eine *formale Grammatik* $G = (V, \Sigma, P, S)$ mit
- einer endlichen Menge aus Nichtterminalsymbolen V (das Vokabular),
- einer Teilmenge $\Sigma \subset V$ aus Terminalsymbolen (das Alphabet),
- einer endlichen Menge aus Produktionsregeln $P \subset (V^* \backslash \Sigma^*) \times V^*$ und
- dem Startsymbol $S \in V \backslash \Sigma$

beschreibt die formale Sprache $L(G)$. [11]

Im Fall des Beispielmodells könnte die folgenden Produktionsregeln eine Grammatik über die Sprache der gültigen Beschreibungen von Schachstellungen ausdrücken. Wir ignorieren hierbei die semantische Komponente (auf dem

gleichen Feld darf nur eine Figur stehen; es existiert nur ein König pro Farbe; usw.). Nichtterminalsymbole sind fettgedruckt.

Definition 12 Definition der Grammatik $\mathcal{G}$, durch die die Sprache $L(\mathcal{G})$ definiert ist.

$$
\begin{aligned}
&\boldsymbol{S} \rightarrow \boldsymbol{I}\,\boldsymbol{I} \mid \boldsymbol{I} \mid \varepsilon \\
&\boldsymbol{I} \rightarrow \boldsymbol{F}\,\boldsymbol{G} \\
&\boldsymbol{F} \rightarrow \boldsymbol{P}\,\boldsymbol{C} \\
&\boldsymbol{G} \rightarrow \boldsymbol{H}\,\boldsymbol{V} \\
&\boldsymbol{P} \rightarrow K \mid Q \mid B \mid N \mid R \mid P \\
&\boldsymbol{C} \rightarrow b \mid w \\
&\boldsymbol{H} \rightarrow a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \\
&\boldsymbol{V} \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8
\end{aligned}
$$

Kodierung von Symbolen

In digitalen Systemen werden Symbole auf unterster Ebene stets durch ein binäres Alphabet $\boldsymbol{B} = \{0,1\}$ repräsentiert. Für jedes Symbol des Modellalphabets $s \in \boldsymbol{\Sigma}$ existiert also ein *Code* $c(s) \in \boldsymbol{B}^+$.

Definition 13 Seien Σ das Alphabet des Modells und C das Codealphabet. Dann definiert die injektive Abbildung $c: \Sigma \rightarrow C^+$ die *Kodierung* von Symbolen des Modellalphabets auf das Codealphabet.

Die Kodierungen der Symbole müssen hierbei nicht zwangsläufig eine binäre Kodierung äquivalenter Länge (abhängig von der Anzahl der Symbole im Alphabet) besitzen, sondern können auch durch Codes unterschiedlicher Länge repräsentiert werden. Die einzige an die Kodierung vorausgesetzte Bedingung ist, dass keine Symbole und -folgen existieren, die durch den gleichen Binärcode kodiert werden. Dies ist genau dann sichergestellt, wenn alle Binärcodes paarweise *präfixfrei* sind. Um die Definition der Präfixfreiheit mathematisch zu beschreiben, benötigen wir zunächst eine Methode, um Präfixe einer vorgegebenen Länge von Symbolfolgen zu extrahieren.

Definition 14 Sei $prefix(c, a) : (\mathbb{N} \times \Sigma^*) \rightarrow \Sigma^*$ eine Funktion, die den größtmöglichen, maximal c langen Präfix der Symbolfolge a zurückgibt. Demnach ergibt sich:

$$prefix(c, a) = prefix\left(c, a_1, \ldots, a_{|a|}\right) = a_1 \circ \ldots \circ a_{\min(c,|a|)}$$

Nun ermöglicht sich schließlich die Definition der Präfixfreiheit:

Definition 15 Zwei Codes $c_1, c_2 \in C, c_1 \neq c_2$ sind präfixfrei, wenn sowohl $prefix(|c_1|,\ c_2) \neq c_1$ als auch $prefix(|c_2|,\ c_1) \neq c_2$ gilt.

In der Praxis bietet der Einsatz eines Entscheidungsbaums (siehe Abbildung 13) die Möglichkeit, präfixfreie Kodierungen herzustellen.

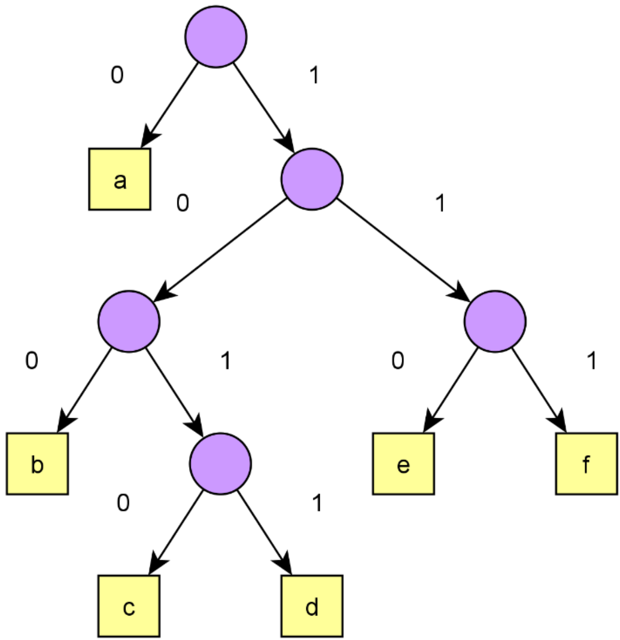


Abbildung 13 Entscheidungsbaum zur Entropiekopierung des Alphabets $\{a, b, c, d, e, f\}$

Minimale Kodierungen

1948 überträgt Shannon das aus der statistischen Mechanik bekannte H-Theorem von Boltzmann in die Informationstheorie [12] und definiert so eine Untergrenze an Bits, die nötig sind, um ein Symbol in Abhängigkeit seiner Auftrittswahrscheinlichkeit zu kodieren:

Definition 16 Sei $p(x)$ die Auftrittswahrscheinlichkeit eines Symbols $x \in \Sigma$ aus dem Alphabet Σ. Dann ist

$$I\big(p(x)\big) = -\log_a p(x)$$

der Informationsgehalt des Symbols. Hierbei beschreibt $a \in \mathbb{N}$ die Anzahl der Zustände des zu Grunde liegenden Informationssystems, in dem das Symbol kodiert ist.

Für binäre Codierungssysteme gilt $a = 2$. $I\big(p(x)\big)$ gibt hierbei direkt die ideelle untere Grenze an notwendigen Bits an, um das Symbol x zu kodieren, wobei in Kodierungssystemen mit endlicher Anzahl diskreter Zustände ($a \in \mathbb{N}$) die Untergrenze fast ausschließlich nur näherungsweise erreicht werden kann (es sei denn, die Auftrittswahrscheinlichkeiten für alle x folgen der Vorschrift $p(x) = \frac{1}{a^k}, k \in \mathbb{N}^0$).

Methoden zur *Entropiekodierung* (einer Familie von Algorithmen zur verlustfreien Komprimierung von Daten) nutzen den beschriebenen Ansatz, um die optimale Codelänge für jedes Symbol zu berechnen und reell anzunähern. Ansätze wie die Shannon-Fano-Kodierung oder Huffman-Kodierung generieren mit Hilfe dieses Theorems Kodierungen $c: \Sigma \to \boldsymbol{B}^+$ für gegebene Symbole und deren zugewiesene Auftrittswahrscheinlichkeiten. Symbole mit größerer Auftrittswahrscheinlichkeit erhalten unter den Voraussetzungen von Definition 16 eine kleinere Codelänge als Symbole mit geringerer Auftrittswahrscheinlichkeit. Auch der in dieser Arbeit beschriebene *Krimp*-Algorithmus basiert auf dem Ansatz der Entropiekodierung.

Minimum Description Length

1978 definierte Jorma Rissanen das Prinzip der *Minimum Description Length* (kurz: *MDL*).

> *„The fundamental idea behind the MDL Principle is that any regularity in a given set of data can be used to compress the data, i.e. to describe it using fewer symbols than needed to describe the data literally."* [13]

Demnach besagt das MDL-Prinzip, dass genau die Daten, die charakteristisch für einen gegebenen Datensatz sind – also signifikant häufiger vorkommende Daten – dazu genutzt werden können, um die Datenmenge zu komprimieren. Mit anderen Worten beschreiben diese somit die Grundgesamtheit der Datenmenge besser, als solche Daten, die eher seltener vorkommen.

2.2.2 Frequent Itemset Mining

Die Annahme, dass häufig auftretende Daten die relevanten Informationen einer Datenmenge enthalten, macht es notwendig Methoden zu entwickeln, um solche häufigen Daten aus einer Datenmenge zu extrahieren. Ein Ansatz ist das *Frequent Itemset Mining*.

Items & Itemsets

Items (zu Deutsch in etwa: *Entitäten*) stellen analog zu den Definitionen im Kapitel 2.2.1 die Informationseinheiten der gesamten Datenmenge dar. In unserem Beispielszenario (Schachfiguren einer bestimmten Farbe, die auf definierten Feldern stehen) sind Items genau die Wörter, die durch die Produktionsregel $\boldsymbol{I}$ der Grammatik $\boldsymbol{\mathcal{G}}$ (siehe Definition 12) gebildet werden können. Innerhalb der Datenmenge stellen Items die atomaren Symbole dar; sie bilden die elementaren Informationseinheiten.

Itemsets – Mengen von Items – beschreiben, welche Items innerhalb eines Datensatzes zusammen auftreten. Im vorangegangenen Beispiel besteht eine Schachstellung aus mehreren „Figuren-auf-Feldern“-Items, wobei die Schachstellung einen abgeschlossenen Datensatz – eine *Transaktion* – darstellt. Datenbanken fassen Transaktionen zusammen (Datenbanken von Schachstellungen).

Definition 17 Sei I die Menge aller möglichen Items. Ein Itemset s ist dann ein Element der Potenzmenge über alle möglichen Items: $s \in \mathcal{P}(I)$

Definition 18 Sei $t \in \mathcal{P}(I)$ eine Transaktion. Eine Datenbank db ist dann eine Ansammlung von Transaktionen. $t \in db$ notiert, falls die Transaktion t in der Datenbank db vorkommt.

Definition 19 Sei db eine Datenbank. Dann ist $freq_{db}(s) : \mathcal{P}(I) \rightarrow \mathbb{N}^0$ definiert als die Anzahl der Transaktionen in db, die das Itemset s enthalten.

Frequent Itemsets

Frequent Itemsets sind Teilmengen von Transaktionen, die mit einer vordefinierten Mindesthäufigkeit (dem *Minimum Support*) in der Datenbank auftreten. Der *Support* ist wie folgt definiert:

Definition 20 Sei $s \in \mathcal{P}(I)$ ein Itemset und db eine Datenbank. Dann ist der Support von s in db definiert als

$$supp_{db}(s) = \frac{freq_{db}(s)}{|db|}$$

mit $|db|$: Anzahl der Transaktionen in der Datenbank

Ein Algorithmus, der auch von *Krimp* verwendet wird, ist der APRIORI-Algorithmus. Der folgende Pseudocode beschreibt seine Funktionsweise:

```
APRIORI(db, minsup):
 # db     : Datenbank
 # minsup : Minimum-Support

 # alle Items, die über dem Minimumsupport liegen, einsammeln
 foreach s ∈ db,  |s| = 1:
       if supp_db(s) ≥ minsup:
             L_1 ← s
 # Iteration über alle nächstgrößeren Itemsets
 k ← 2
 while L_{k−1} ≠ ∅:
       # Alle Kandidaten der Größe k einsammeln
       C_k ← {c|c = a ∪ {b} ∧ a ∈ L_{k−1} ∧ b ∈ ⋃ L_{k−1} ∧ b ∉ a}
       foreach t ∈ db:
             # Alle Kandidaten der Größe k, die in dieser
             # Transaktion vorkommen, einsammeln
             C_t ← {c|c ∈ C_k ∧ c ⊆ t}
             foreach c ∈ C_t:
                   count[c] ← count[c] + 1
       # Alle Kandidaten über dem Minimum Support zu
       # den Frequent Itemsets hinzufügen
       L_k ← {c|c ∈ C_k ∧ count[c] ≥ minsup}
       # Mit den nächst größeren Itemsets fortfahren
       k ← k + 1
 return ⋃_k L_k
```

Listing 2 Pseudocode des APRIORI–Algorithmus [14]

Zur Bestimmung eines angemessenen Werts für den Minimum Support existiert keine global anwendbare Maßgabe. Ein Wert von 0 gibt alle möglichen Itemsets aus $(\mathcal{P}(I))$, ein Minimum Support von 1 liefert nur das Item oder Itemset, das in allen Transaktionen der Datenbank vorkommt (falls vorhanden).

2.3 Krimp

Das grundlegende Problem von Frequent Itemset Mining besteht darin, dass entweder nur solche Itemsets gefunden werden, die man schon kennt (Minimum Support zu hoch) oder die Zahl der gefundenen Itemsets explodiert (Minimum Support zu niedrig).

> *"The best set of frequent item sets is that set that compresses the database best." (Siebes, Leeuwen, 2006)* [15]

Krimp adressiert dieses Problem, indem der Algorithmus als ein dem Frequent Itemset Mining nachgelagerter Prozess die Datenbank mit Hilfe der resultierenden Frequent Itemsets komprimiert. Das Ergebnis des Algorithmus stellt eine *Codetabelle* dar, die die Frequent Itemsets und Items beinhaltet, die zur Kodierung verwendet wurden.

Krimp verwendet den in Abschnitt 2.2.2 („Frequent Itemsets") vorgestellten APRIORI-Algorithmus zum Mining der Frequent Itemsets.

2.3.1 Grundprinzip

Die oben genannten Annahmen stützen sich auf das *Minimum Description Length* – Prinzip (siehe Abschnitt 2.2.1, Minimum Description Length). In ihrer Publikation [15] beschreiben die Autoren heuristische Algorithmen, deren resultierende Kompressionsraten in Experimenten über diverse Modelldatenbanken [16] gemessen wurden.

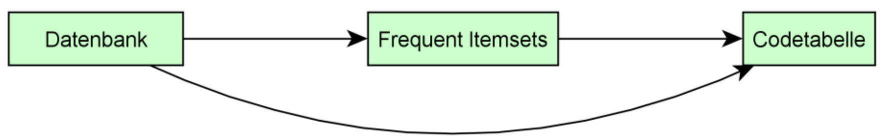


Abbildung 14 Transformationsschritte des Krimp-Algorithmus

2.3.2 Generierung der Codetabelle & Kompression der Datenbank

Hüllen, Kodierungsschemata

Analog zu den in Abschnitt 2.2.1 (Kodierung von Symbolen, Minimale Kodierungen) beschriebenen Definitionen und Methoden zur Kodierung von Modellalphabeten in Codealphabeten besteht die durch Krimp definierte Codetabelle aus Elementen des Codealphabets – einer Untermenge der aus der Datenbank extrahierten Frequent Itemsets.

Um die Kodierung einer Datenbank durch (Frequent) Itemsets und einzelnen Items zu erlauben, ist es zunächst notwendig zu definieren, wie die Transaktionen durch eine Codetabelle beschrieben werden können und welche Anforderungen an die Codetabelle gestellt werden.

Definition 21 Eine Menge von Itemsets C ist eine *Hülle* der Datenbank db, gdw. für jede Transaktion $t \in db$ eine Untermenge $C(t) \subseteq C$ (die Hülle der Transaktion) existiert, so dass gilt

1. $t = \bigcup_{c_i \in C(t)} c_i$
2. $\forall c_i, c_j \in C(t) : c_i \neq c_j \rightarrow c_i \cap c_j = \emptyset$
3. $\bigcup_{\forall t} C(t) = C.$

Wir sagen: $C(t)$ *umhüllt* t. (außer 3. Bedingung: [15])

Definition 22 Um eine Datenbank zu kodieren, benötigt man nun gemäß Definition 13 eine Abbildung von Transaktionen zu Elementen von C^+. Da die Reihenfolge der Abbildung im Fall von Mengen (Itemsets) keine Rolle spielt, reicht eine Abbildung nach $\mathcal{P}(C)$ aus:

Definition 23 Ein *Kodierungsschema* CS einer Datenbank db ist ein Paar (C, S), in dem C – die *Codemenge* – eine Hülle der Datenbank db und S eine Funktion $S : db \rightarrow \mathcal{P}(C)$ ist, so dass gilt: $S(t)$ ist die Hülle der Transaktion t. [15]

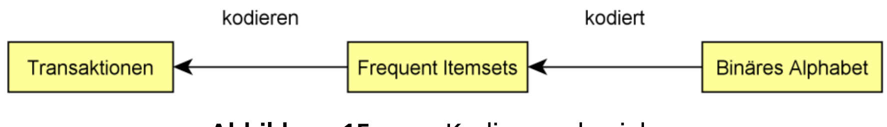


Abbildung 15 Kodierungsbeziehungen

Während die Transaktionen der Datenbank durch Item(set)s aus der Codemenge kodiert werden, muss jedem Element aus C gemäß Definition 13 ein (binärer) Code zugewiesen werden, der den Anforderungen aus Definition 15 genügt. Um zusätzlich zur Kodierung eine Komprimierung der Datenbank zu erreichen, müssen die häufiger zur Kodierung verwendeten Itemsets aus C entsprechend Abschnitt 2.2.1 („Minimale Kodierungen“) durch kürzere Codes repräsentiert werden. Die dafür benötigten Auftrittswahrscheinlichkeiten über C lassen sich mit Hilfe der relativen Häufigkeiten, wie oft ein entsprechendes Itemset zur Kodierung der Datenbank herangezogen wurde, approximieren:

Definition 24 Sei (C, S) das Kodierungsschema der Datenbank db und $cfreq(c) : C \rightarrow \mathbb{N}^0$ definiert als die Anzahl der Transaktionen aus db, die mit Hilfe des Itemsets c gemäß der Abbildung S kodiert werden. Dann ist

$$\forall c \in C : P(c) = \frac{cfreq(c)}{\sum_{d \in C} cfreq(d)}$$

die approximierte Verteilung über die Auftrittswahrscheinlichkeiten der zur Kodierung herangezogenen Itemsets – die *Kodierungsverteilung*. Für gegebenes c bezeichnen wir $P(c)$ als die *Kodierungswahrscheinlichkeit* von c. [15]

Nach Definition 16 lässt sich abschließend für die Codelänge eines Itemsets des Kodierungsschemas die untere Schranke $L(c) = -\log_2 p(c), c \in C$ berechnen, die es zu erreichen oder zumindest anzunähern gilt. $L(x)$ heißt die *Beschreibungslänge* von x und ist definiert als die Anzahl Bits, die man benötigt, um x zu beschreiben. Analog zu den Beschreibungslängen der Codemengenelemente lassen sich diese auch für Transaktionen definieren.

Definition 25 Sei $t \in db$ eine Transaktion der Datenbank db und CS ein Kodierungsschema von db. Dann gilt für die Beschreibungslänge von t:

$$L(t) = \sum_{c \in S(t)} L(c)$$

[15]

Die Beschreibungslänge einer Transaktion ist demnach die Summe der Beschreibungslängen der Elemente aus der Hülle der Transaktion.

Definition 26 Sei db eine Datenbank. Dann gilt für die Beschreibungslänge von db:

$$L(db) = \sum_{t \in db} L(t)$$

[15]

Vorgehensweise

Die *Codetabelle* beinhaltet alle Elemente der Codemenge und weist diesen ihre binären Codes zu. Da lediglich deren Länge zur Untersuchung der von Krimp erreichten Kompressionsraten notwendig ist, werden die Codes nicht explizit vom Algorithmus ausgegeben.

Die STANDARD–Funktion gibt die eingegebene Menge I von in der Datenbank auftretenden Items absteigend nach Supportwert zurück. Sie bilden die Grundlage der Codetabelle, welche anschließend durch die Hinzunahme der Frequent Itemsets ergänzt wird.

```
STANDARD(I, db):
  # I    : Menge von Items
  # db   : Datenbank
```

```
foreach item ∈ I:
  support_item ← supp_db(item)
return I absteigend sortiert nach support_item∈I
```

Listing 3 Pseudocode der STANDARD–Funktion für Items [15]

Die Sortierung der Frequent Itemsets und Items der Codetabelle erfolgt gemäß der *Standardordnung*, welche die durch die STANDARD-Funktion definierte Sortierung von Items auf Itemsets überträgt.

Definition 27 Sei db eine Datenbank von Transaktionen über Itemsets und I die Menge der in der Datenbank vorhandenen Items. Sei weiterhin $\mathcal{S} \in \mathcal{P}(I)$ eine geordnete Menge von Itemsets. $\mathcal{S}$ ist im Kontext von db genau dann in *Standardordnung*, wenn für beliebige $J_1, J_2 \in \mathcal{S}$ gilt:

1. $|J_1| \leq |J_2| \Leftrightarrow J_2 \preccurlyeq_{\mathcal{S}} J_1$
2. $|J_1| = |J_2| \wedge supp_{db}(J_1) \leq supp_{db}(J_2) \Leftrightarrow J_2 \preccurlyeq_{\mathcal{S}} J_1$

[15]

Man beachte die zur $\leq$-Relation entgegengesetzte Semantik der $\preccurlyeq_{\mathcal{S}}$-Relation: Ein Itemset ist gemäß der Standardordnung minimal, wenn es möglichst groß ist und einen möglichst großen Supportwert hat.

Möchte man nun eine Datenbank mit Hilfe der Codetabelle kodieren, wählt man für jede Transaktion den im Sinne der Standardordnung minimalsten Eintrag der Codetabelle aus, der die Transaktion umhüllt. Dieser Schritt wird solange wiederholt, bis die Hülle der Transaktion vollständig ist.

```
STANDARD(𝒥, db):
  # 𝒥   : Menge von Itemsets
  # db  : Datenbank

  foreach Itemset ∈ 𝒥:
        support_Itemset ← supp_db(Itemset)
        size_Itemset ← |Itemset|
  return 𝒥 absteigend sortiert nach size_Itemset∈𝒥 und support_Itemset∈𝒥
```

Listing 4 Erweiterung der STANDARD–Funktion für Itemsets

Während der Algorithmus die Codetabelle erstellt, werden Frequent Itemsets, die als Kandidaten in Frage kommen, nach der *Hüllenordnung* sortiert und der nach dieser Ordnung maximale Kandidat untersucht.

Definition 28 Sei db eine Datenbank von Transaktionen über Itemsets und I die Menge der in der Datenbank vorhandenen Items. Sei $\mathcal{C} \in \mathcal{P}(I)$ eine geordnete Menge von Itemsets. $\mathcal{C}$ ist im Kontext von db genau dann in *Hüllenordnung*, wenn für beliebige $J_1, J_2 \in \mathcal{C}$ gilt:

$$support_{db}(J_1) \leq support_{db}(J_2) \Leftrightarrow J_1 \preccurlyeq_{\mathcal{C}} J_2$$

Die Funktion COVER-ORDER gibt die eingegebene Menge $\mathcal{J}$ von in der Datenbank db enthaltenen Itemsets nach der durch die Hüllenordnung definierte Sortierung zurück.

```
COVER-ORDER(𝒥, db):
  # 𝒥   : Menge von Itemsets
  # db  : Datenbank

  foreach Itemset ∈ 𝒥:
```

```
        support_Itemset ← supp_db(Itemset)
return 𝓘 aufsteigend sortiert nach support_Itemset
```

Listing 5 Pseudocode der COVER-ORDER–Funktion

Die grundlegende Vorgehensweise zur Generierung der Codetabelle definiert sich nun wie folgt:

```
NAIVE-COMPRESSION(I, 𝓘, db):
 # I    : Menge der in der Datenbank vorkommenden Items
 # 𝓘    : Menge der Frequent Itemsets der Datenbank
 # db   : Datenbank

 CodeSet ← STANDARD(I, db)
 𝓘 ← 𝓘 \ I
 CanItems ← COVER-ORDER(𝓘, db):
 while CanItems ≠ ∅:
         candidate ← entferne maximales Element von CanItems
         CanCodeSet ← CodeSet ⊕ {candidate}
         if L_CanCodeSet(db) < L_CodeSet(db):
                 CodeSet ← CanCodeSet
 return CodeSet
```

Listing 6 Pseudocode des NAIVE-COMPRESSION–Algorithmus [15]

Der Infixoperator $\square \oplus \square$ gibt an, dass die Elemente der rechten Menge derart in die linke geordnete Menge eingefügt werden, so dass deren Ordnung erhalten bleibt.

Für jeden Kandidaten aus den Frequent Itemsets wird also geprüft, ob die Datenbank durch seine Hinzunahme zur Codemenge eine kürzere Beschreibungslänge erhält. Wenn dies der Fall ist, wird die neue Codemenge behalten, andernfalls verworfen.

Beispieliteration

Sei die Datenbank *sampledb* über die Items $\{I_1, I_2, I_3\}$ definiert durch folgende Transaktionen:

$T_1\colon \{I_1, I_2\}$

$T_2\colon \{I_1, I_3\}$

$T_3\colon \{I_1, I_2, I_3\}$

$T_4\colon \{I_1, I_2, I_3\}$

Hieraus ergeben sich die folgenden Supportwerte:

Itemset	Support (abs. / rel.)
$\{I_1\}$	4 / 100 %
$\{I_2\}$	3 / 75 %

$\{I_3\}$	3 / 75 %
$\{I_1, I_2\}$	3 / 75 %
$\{I_1, I_3\}$	3 / 75 %
$\{I_2, I_3\}$	2 / 50 %
$\{I_1, I_2, I_3\}$	2 / 50 %

Tabelle 3 Itemsets und Supportwerte der Datenbank *sampledb*

Gemäß dem NAIVE-COMPRESSION–Algorithmus ergeben sich nach der dritten Iteration die Codemenge $\{\{I_1\},\{I_2\},\{I_3\}\}$, die Hülle

$$S_{sampledb,i=3} = \{\mathrm{T}_1 \Rightarrow \{\{I_1\},\{I_2\}\}, \mathrm{T}_2 \Rightarrow \{\{I_1\},\{I_3\}\}, \mathrm{T}_3 \Rightarrow \{\{I_1\},\{I_2\},\{I_3\}\}, \mathrm{T}_4 \Rightarrow \{\{I_1\},\{I_2\},\{I_3\}\}\},$$

die Codelängen

$$L_{\{I_1\}} = \left\lceil -\log_2 \frac{4}{10} \right\rceil = 2, \qquad L_{\{I_2\}} = \left\lceil -\log_2 \frac{3}{10} \right\rceil = 2, \qquad L_{\{I_3\}} = \left\lceil -\log_2 \frac{3}{10} \right\rceil = 2$$

und die Beschreibungslänge der Datenbank

$$L_{sampledb,i=3} = 4 * L_{\{I_1\}} + 3 * L_{\{I_2\}} + 3 * L_{\{I_3\}} = 20$$

In der vierten Iteration erhält man durch Analyse des Kandidaten $\{I_1, I_2\}$ folgende Werte:

Test-Codemenge: $\{\{I_1, I_2\},\{I_1\},\{\mathrm{I}_2\},\{I_3\}\}$
Hülle: $S_{sampledb,i=4} = \{\mathrm{T}_1 \Rightarrow \{\{I_1, I_2\}\}, \mathrm{T}_2 \Rightarrow \{\{I_1\},\{I_3\}\}, \mathrm{T}_3 \Rightarrow \{\{I_1, I_2\},\{I_3\}\}, \mathrm{T}_4 \Rightarrow \{\{I_1, I_2\},\{I_3\}\}\}$
Codelängen: $L_{\{I_1,I_2\}} = \left\lceil -\log_2 \frac{3}{7} \right\rceil = 2,\ L_{\{I_1\}} = \left\lceil -\log_2 \frac{1}{7} \right\rceil = 3,\ L_{\{I_3\}} = \left\lceil -\log_2 \frac{3}{7} \right\rceil = 2$
Beschreibungslänge: $L_{sampledb,i=4} = 3 * L_{\{I_1,I_2\}} + 1 * L_{\{I_1\}} + 3 * L_{\{I_3\}} = 15$
Ergebnis: Codemenge wird übernommen

In der fünften Iteration erhält man durch Analyse des Kandidaten $\{I_1, I_3\}$ folgende Werte:

Codemenge: $\{\{I_1, I_3\},\{I_1, I_2\},\{I_1\},\{\mathrm{I}_2\},\{I_3\}\}$
Hülle: $S_{sampledb,i=5} = \{\mathrm{T}_1 \Rightarrow \{\{I_1, I_2\}\}, \mathrm{T}_2 \Rightarrow \{\{I_1, I_3\}\}, \mathrm{T}_3 \Rightarrow \{\{I_1, I_2\},\{I_3\}\}, \mathrm{T}_4 \Rightarrow \{\{I_1, I_2\},\{I_3\}\}\}$
Codelängen: $L_{\{I_1,I_2\}} = \left\lceil -\log_2 \frac{3}{6} \right\rceil = 1,\ L_{\{I_1,I_3\}} = \left\lceil -\log_2 \frac{1}{6} \right\rceil = 3,\ L_{\{I_3\}} = \left\lceil -\log_2 \frac{2}{6} \right\rceil = 2$
Beschreibungslänge: $L_{sampledb,i=5} = 3 * L_{\{I_1,I_2\}} + 1 * L_{\{I_1,I_3\}} + 2 * L_{\{I_3\}} = 10$
Ergebnis: Codemenge wird übernommen

In der sechsten Iteration erhält man durch Analyse des Kandidaten $\{I_2, I_3\}$ folgende Werte:

Codemenge: $\{\{I_2, I_3\},\{I_1, I_3\},\{I_1, I_2\},\{I_1\},\{\mathrm{I}_2\},\{I_3\}\}$
Hülle: $S_{sampledb,i=6} = S_{sampledb,i=5}$
Codelängen: *wie in Iteration 5*
Beschreibungslänge: $L_{sampledb,i=6} = L_{sampledb,i=5}$
Ergebnis: Codemenge wird nicht übernommen

In der siebten Iteration erhält man durch Analyse des Kandidaten $\{I_1, I_2, I_3\}$ folgende Werte:

Codemenge: $\{\{I_1, I_2, I_3\},\{I_1, I_3\},\{I_1, I_2\},\{I_1\},\{\mathrm{I}_2\},\{I_3\}\}$
Hülle: $S_{sampledb,i=7} = \{\mathrm{T}_1 \Rightarrow \{\{I_1, I_2\}\}, \mathrm{T}_2 \Rightarrow \{\{I_1, I_3\}\}, \mathrm{T}_3 \Rightarrow \{\{I_1, I_2, I_3\}\}, \mathrm{T}_4 \Rightarrow \{\{I_1, I_2, I_3\}\}\}$
Codelängen: $L_{\{I_1,I_2,I_3\}} = \left\lceil -\log_2 \frac{2}{4} \right\rceil = 1,\ L_{\{I_1,I_2\}} = \left\lceil -\log_2 \frac{1}{4} \right\rceil = 2,\ L_{\{I_1,I_3\}} = \left\lceil -\log_2 \frac{1}{4} \right\rceil = 2$
Beschreibungslänge: $L_{sampledb,i=7} = 2 * L_{\{I_1,I_2,I_3\}} + 1 * L_{\{I_1,I_2\}} + 1 * L_{\{I_1,I_3\}} = 6$
Ergebnis: Codemenge wird übernommen

Im Anschluss werden alle Itemsets der Codemenge, die nicht zur Kodierung der Datenbank verwendet werden, aus ihr entfernt. Die finale Codemenge lautet hier: $\{\{I_1, I_2, I_3\}, \{I_1, I_3\}, \{I_1, I_2\}\}$

Pruning

Da es unter Umständen vorkommen kann, dass die Entfernung eines Itemsets aus der Codemenge zur Verringerung der Beschreibungslänge führt, definieren die Autoren eine weitere Funktion, die solche Itemsets identifiziert und aus der Codemenge löscht:

```
PRUNE-ON-THE-FLY(CanCodeSet, CodeSet, db):
 # CanCodeSet        : Codemenge inklusive hinzugefügtem Kandidaten
 # CodeSet           : Codemenge vor der Hinzunahme des Kandidaten
 # db                : Datenbank

 PruneSet ← {J ∈ CodeSet | cfreq_CanCodeSet(J) < cfreq_CodeSet(J)}
 PruneSet ← STANDARD(PruneSet, db)
 while PruneSet ≠ ∅:
       candidate ← entferne letztes Element von PruneSet # Minimum-Support zuerst
       PosCodeSet ← PosCodeSet \ {candidate}
       if L_PosCodeSet(db) < L_CanCodeSet(db):
             CanCodeSet ← PosCodeSet

 return CanCodeSet
```

Listing 7 Pseudocode der PRUNE-ON-THE-FLY–Funktion [15]

Die PRUNE-ON-THE-FLY–Funktion prüft für jedes Itemset, das in der neuen Codemenge seltener zum Kodieren der Datenbank eingesetzt wird als in der vorhergehenden Codemenge, wie sich eine Entfernung des Itemsets auf die Beschreibungslänge der Datenbank auswirkt und entfernt es dann entsprechend.

Finaler Algorithmus

Schließlich bilden die Funktionen NAIVE-COMPRESSION und PRUNE-ON-THE-FLY gemeinsam den finalen COMPRESS-AND-PRUNE–Algorithmus, der in der Implementation von Krimp verwendet wird:

```
COMPRESS-AND-PRUNE(I, 𝓘, db):
 # I    : Menge der in der Datenbank vorkommenden Items
 # 𝓘    : Menge der Frequent Itemsets der Datenbank
 # db   : Datenbank

 CodeSet ← STANDARD(I, db)
 𝓘 ← 𝓘 \ I
 CanItems ← COVER-ORDER(𝓘, db)
 while CanItems ≠ ∅:
       candidate ← entferne maximales Element von CanItems
```

```
        CanCodeSet ← CodeSet ⊕ {candidate}
        if L_CanCodeSet(db) < L_CodeSet(db):
                CanCodeSet ← PRUNE-ON-THE-FLY(CanCodeSet, CodeSet, db)
                CodeSet ← CanCodeSet
return CodeSet
```

Listing 8 Pseudocode des COMPRESS-AND-PRUNE–Algorithmus [15]

2.3.3 Wahl des Minimum Supports

Die Autoren von Krimp schlagen zur Bestimmung eines geeigneten Minimum Supports folgendes Vorgehen vor:

1. Starte mit einem hohen Minimum Support.
2. Komprimiere die Datenbank und prüfe die Kompressionsrate.
3. Senke den Minimum Support und wiederhole Schritt 1 und 2, bis die Kompressionsrate nicht mehr signifikant steigt.

Klar ist, dass erst bei Betrachtung aller möglichen Itemsets die höchste Kompression der Datenbank erreicht werden kann. Da dies jedoch bereits bei einer überschaubaren Menge von verschiedenen Items nicht in angemessener Zeit durchführbar ist und der Verlauf der Kompressionsrate nicht zwangsläufig gleichmäßig steigend ist (siehe Abbildung 16), gibt diese Heuristik laut [15] zumindest eine Abschätzung dafür an, wann die Kompressionsrate für gegebenen Minimum Support konvergiert.

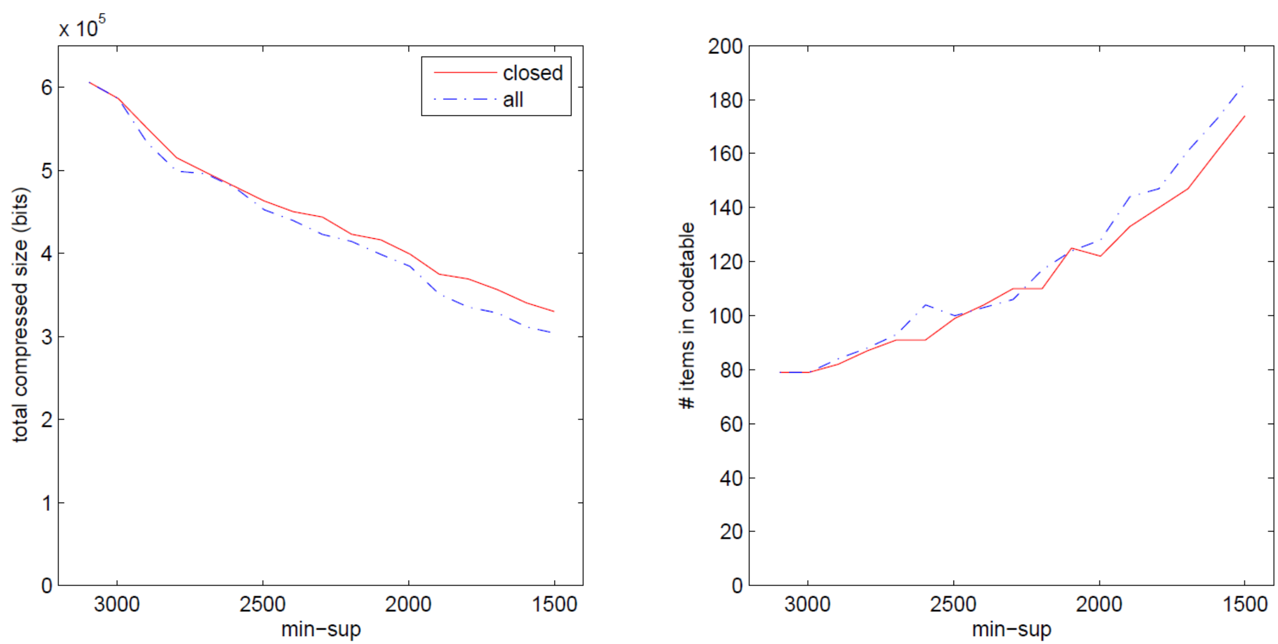


Abbildung 16 Vergleich der Kompressionsraten der UCI „chess“-Datenbank in Abhängigkeit vom Minimum Support [15]

2.4 Algorithmen & Methoden

Um die Schachpartien mit Hilfe von Krimp zu analysieren, bedarf es einiger Vorverarbeitungsschritte, die in diesem Abschnitt vorgestellt werden.

2.4.1 Konvertierung von Schachpartien zu –stellungen

Die im PGN-Format vorliegenden Schachpartien müssen zunächst in Listen von Schachstellungen überführt werden. Dazu wird jeder notierte Halbzug auf ein mit der Grundstellung initialisiertes Schachbrett angewendet und die so manipulierte Schachstellung in die resultierende Liste hinzugefügt. Die Funktion PARSESINGLEGAME übernimmt diese Aufgabe.

```
PARSESINGLEGAME(PGNGame):
  # PGNGame   : Schachzüge im PGN-Format

  chessBoards ← []
  chessBoards[1] ← Grundstellung
  i ← 1
  foreach move ∈ PGNGame:
        chessBoards[i + 1] ← (chessBoards[i] ↞ move) # Zug auf Stellung anwenden
        i ← i + 1
  return chessBoards
```

Listing 9 Pseudocode der PARSESINGLEGAME–Funktion

Der Infixoperator □ ↞ □ gibt hierbei an, dass der Halbzug auf der rechten Seite auf das Schachbrett angewendet wird, das sich auf der linken Seite des Operators befindet.

2.4.2 Extraktion von Abschnitten einer Partie

Um Segmente einer Schachpartie zu isolieren (Eröffnung, Mittelspiel, Endspiel) bedarf es einer Funktion, die aus einer Liste von Schachstellungen die gewünschten Intervalle extrahiert. Die Funktionen EXTRACTOPENING, EXTRACTMIDDLEGAME und EXTRACTENDGAME erfüllen diese Aufgaben.

```
EXTRACTOPENING(chessBoards, num):
  # chessBoards : Liste von Schachstellungen
  # num         : Anzahl der zu extrahierenden Stellungen

  openingBoards ← []
  i ← 1
  while i ≤ num and i ≤ |chessBoards|:
        openingBoards[i] ← chessBoards[i]
        i ← i + 1
  return openingBoards
```

Listing 10 Pseudocode der EXTRACTOPENING–Funktion

```
EXTRACTMIDDLEGAME(chessBoards, after, num):
```

```
  # chessBoards : Liste von Schachstellungen
  # after       : Anzahl der zu übergehenden Stellungen
  # num         : Anzahl der zu extrahierenden Stellungen

  middlegameBoards ← []
  i ← after + 1
  while i ≤ num + after and i ≤ |chessBoards| :
          middlegameBoards[i] ← chessBoards[i]
          i ← i + 1
  return middlegameBoards
```

Listing 11 Pseudocode der EXTRACTMIDDLEGAME–Funktion

```
EXTRACTENDGAME (chessBoards, num) :
  # chessBoards : Liste von Schachstellungen
  # num         : Anzahl der zu extrahierenden Stellungen

  endgameBoards ← []
  i ← |chessBoards| − num + 1
  while i ≤ |chessBoards| :
          endgameBoards[i] ← chessBoards[i]
          i ← i + 1
  return endgameBoards
```

Listing 12 Pseudocode der EXTRACTENDGAME–Funktion

2.4.3 Extraktion der Bauernstruktur

Für die Experimente 3.1 bis 3.3 benötigt man die charakteristische Bauernstruktur der Schachstellungen (siehe 2.1: Bauernstrukturen). Die Funktion EXTRACTPAWNSTRUCTURE extrahiert diese aus der übergebenen Schachstellung.

```
EXTRACTPAWNSTRUCTURE (chessBoard) :
  # chessBoard  : Schachstellung

  pawnStructure ← chessBoard
  foreach piece ∈ pawnStructure :
          if type(piece) ≠ pawn and type(piece) ≠ king :
                  pawnStructure ← pawnStructure \ {piece}
  return pawnStructure
```

Listing 13 Pseudocode der EXTRACTPAWNSTRUCTURE–Funktion

Die Funktion *type* gibt hierbei den Typ der Schachfigur zurück (*pawn*=Bauer, *king*=König, *queen*=Dame, *rook*=Turm, *knight*=Springer, *bishop*=Läufer).

2.4.4 Konvertierung von Schachstellungen zu Itemsets

Die Implementierung des Krimp–Algorithmus erwartet, dass die Items der Eingabedatenmenge als positive Ganzzahlen im Bereich von 0 bis 65535 kodiert werden. Im Falle der Datenbanken von Schachstellungen wird dies dadurch erreicht, dass die Eigenschaften der Stellungen durch Zahlen repräsentiert werden, welche durch eine lineare Abbildung zu der eindeutigen resultierenden Item-Id kombiniert werden.

Der jeder Eigenschaft zugeordnete Zahlenwert ist wie folgt definiert:

Figurenfarbe $colorOrdinal(piece)$

Farbe	Zugeordneter Wert
weiß	0
schwarz	1

Tabelle 4 Zuordnung von Figurenfarbe zu Zahlenwert

Figurentyp $typeOrdinal(piece)$

Typ	Zugeordneter Wert
König	0
Dame	1
Turm	2
Läufer	3
Springer	4
Bauer	5

Tabelle 5 Zuordnung von Figurentyp zu Zahlenwert

Horizontale Position (A-H) $hPosOrdinal(piece)$

Horizontale Position	Zugeordneter Wert
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7

Tabelle 6 Zuordnung von horizontaler Position zu Zahlenwert

Vertikale Position (1-8) $\boldsymbol{vPosOrdinal(piece)}$

Vertikale Position	Zugeordneter Wert
1	0
2	1
3	2
4	3
5	4
6	5
7	6
8	7

Tabelle 7 Zuordnung von vertikaler Position zu Zahlenwert

Linearisierung

Die resultierende Item-Id wird nach folgender Formel berechnet:

$$itemid(piece) = 10000 * colorOrdinal(piece) + 1000 * typeOrdinal(piece)$$
$$+10 * hPosOrdinal(piece) + vPosOrdinal(piece)$$

Die Linearisierung der einzelnen Eigenschaften durch Faktorisierung mit Zehnerpotenzen lassen die resultierenden Item-Ids lesbar bleiben.

2.4.5 Filterung nach Spielergebnis

Die Filterung nach Spielergebnis ist ein trivialer Prozess, bei dem die Partien der Datenbank in drei Teile aufgeteilt werden:

1. Partien, die weiß gewonnen hat
2. Partien, die schwarz gewonnen hat
3. Unentschiedene Partien

Vorverarbeitung der Schachstellungen

Um festzustellen, welchen Einfluss die Wahl der Daten auf das Resultat der Kompression durch den Krimp-Algorithmus hat (und nicht zuletzt auch aus Gründen der Performanz), wurden die zu komprimierenden Schachstellungen insgesamt nach drei verschiedenen Teilaspekten aus der Gesamtmenge der Stellungsdatenbank extrahiert.

Die erste Aufteilung erfolgte anhand der Spielphase, in der die Schachstellungen auftreten. Die Spielphasen wurden gemäß den in Tabelle 8 definierten Halbzugintervallen zugeordnet.

Spielphase	Halbzugintervall
Eröffnung	erste 20 Halbzüge
Endspiel	letzte 10 Halbzüge
Mittelspiel	restliche Halbzüge

Tabelle 8 Definition der Spielphasen über Intervalle von Halbzügen

Eine zweite Aufteilung erfolgte nach dem Endergebnis der Partie, in der die Stellungen auftreten (*weiß gewinnt*, *schwarz gewinnt*, *unentschieden*).

Als Letztes wurden Schachstellungen gemäß ihren spezifischen Bauernstrukturen aufgeteilt. Hierbei wurden zunächst die zehn häufigsten Bauernstrukturen aus den Mittelspielen aller Partiedatenbanken gemäß dem Algorithmus aus 2.5.6 extrahiert. In Abbildung 17 bis Abbildung 19 sind die identifizierten drei häufigsten Bauernstrukturen (kurz: BS) aufgelistet. Die angegebenen Häufigkeiten (kurz: N) beziehen sich auf das Auftreten der Bauernstrukturen innerhalb der Mittelspiele aller Schachpartien. Im anschließenden Schritt wurden alle Schachstellungen nach zutreffender Bauernstruktur gefiltert und die residuellen Stellungen (Stellungen nach Entfernung der Bauernstruktur) von Krimp komprimiert.

Anzumerken ist hierbei, dass eine Schachstellung nur dann als passend zu einer Bauernstruktur definiert wird, wenn die Schnittmenge der Bauern aus der Bauernstruktur und der Schachstellung leer ist, also keine weiteren Bauern in der Schachstellung enthalten sind, die nicht auch in der Bauernstruktur enthalten sind.

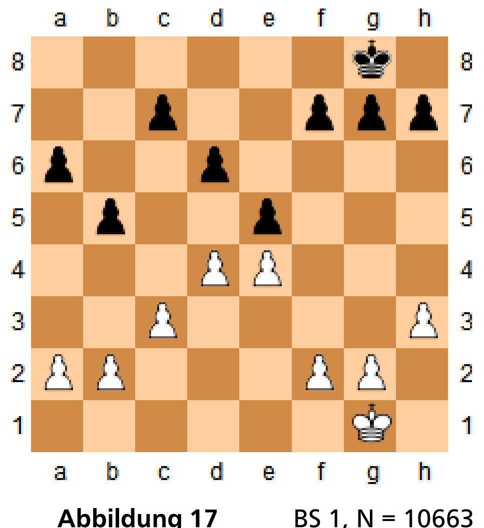


Abbildung 17 BS 1, N = 10663

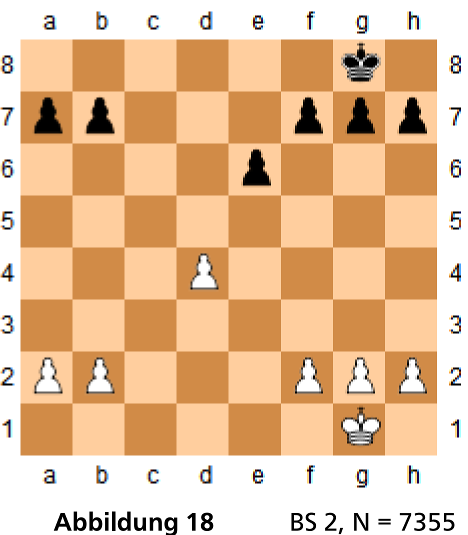


Abbildung 18 BS 2, N = 7355

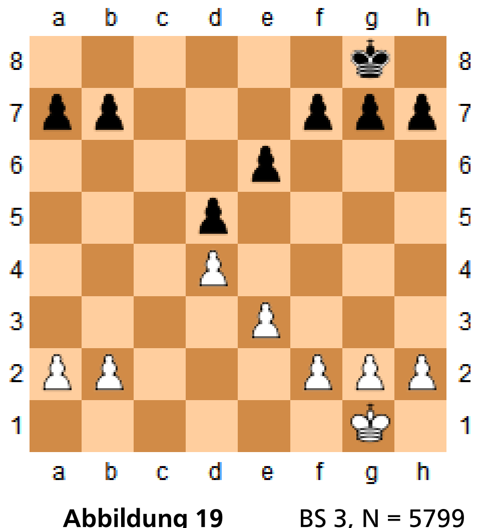


Abbildung 19 BS 3, N = 5799

Weitere Filtermaßnahmen

Aus Gründen der Performanz ergab sich, dass es hilfreich sein kann, Schachfiguren, die sich noch auf der Initialposition einer Schachpartie befinden, nicht in die gefilterten Daten zu übernehmen. Diese Filterung wird in Tabelle 9 durch den Bezeichner *nicht-initial* kenntlich gemacht.

Vorbereitete Datenbanken

Die Schachstellungen aus den Eingangsdaten wurden mittels den vorgestellten Vorverarbeitungsschritten gefiltert. Eine Zusammenfassung der resultierenden Datenbanken liefert Tabelle 9. Für jede durchgeführte Kompression gibt die Spalte *Partien* an, wie viele individuelle Schachpartien gefiltert wurden. Die Spalte *Vorverarbeitung* gibt an, welche Vorverarbeitungsschritte durchgeführt wurden, *Stellungen* gibt die Anzahl der Schachstellungen nach der Filterung und *Min. Support* gibt den in Krimp konfigurierten Minimum Supportlevel sowohl relativ als auch absolut an.

Experiment	Partien	Vorverarbeitung	Stellungen	Min. Support (relativ / absolut)
1.1	100.000	weiß gewinnt	2.929.971	15,00 % / 439.495
1.2	100.000	schwarz gewinnt	2.386.675	15,00 % / 358.001
1.3	100.000	unentschieden	2.346.926	15,00 % / 352.038
2.1	10.000	Eröffnung	200.000	30,00 % / 60.000
2.2	10.000	Mittelspiel	88.677	0,10 % / 88
2.3	10.000	Endspiel	100.000	0,10 % / 100
2.4	100.000	Eröffnung, nicht-initial	1.900.000	0,10 % / 1.900
3.1	4.234.538	Bauernstruktur 1	77.400	0,001 % / 1
3.2	4.234.538	Bauernstruktur 2	102.718	0,001 % / 1
3.3	4.234.538	Bauernstruktur 3	20.561	0,005 % / 1

Tabelle 9 Vorverarbeitete Datenbanken & Kompressionskonfigurationen

Struktur der Aufbereitung

In den folgenden Abschnitten werden alle Ergebnisse gemäß diesem Schema präsentiert:

Zunächst beschreibt der Abschnitt *Konfiguration* sowohl die verwendete Eingabedatenmenge (gemäß Tabelle 9) und die durchgeführten Vorverarbeitungsschritte als auch den Minimum Support, der dem Krimp-Algorithmus als Parameter mitgegeben wurde. Der Minimum Support wurde dabei so gewählt, dass der Frequent Itemset Miner des Krimp-Algorithmus in angemessener Zeit (< 2 Stunden) terminiert.

Im Abschnitt *Quantitative Analyse* befindet sich unterhalb der Angabe, wie viele Frequent Itemsets gefunden wurden, eine Auflistung der gefundenen Teilstellungen (Codetabellenelemente) gruppiert nach der Anzahl der in der Teilstellung enthaltenen Figuren. Des Weiteren ist darunter die Reduktion der Frequent Itemsets angegeben, die durch die Anwendung von Krimp erreicht werden konnte.

Im Abschnitt *Ausgewählte Teilstellungen* werden exemplarisch die größten Teilstellungen mit den höchsten Supportwerten visualisiert.

Der Abschnitt *Analyse der Kompression* vergleicht die von Krimp erreichte Kompressionsrate mit den Raten der Kompressionsalgorithmen *Deflate* (ZIP) und RAR.

Der Abschnitt *Beobachtungen* benennt ausgewählte, im Rahmen des Experiments gemachte Erfahrungen und Eindrücke, die nicht durch die anderen Abschnitte erfasst werden.

3.1 Experiment 1.1

3.1.1 Konfiguration

Partien:	100.000
Vorverarbeitung:	weiß gewinnt
Extrahierte Stellungen:	2.929.971
Minimum Support (rel./abs.):	15,00 % / 439.495

3.1.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 20.181

Krimp Miner

Tabelle 10 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
10	6
9	47
8	197
7	919
6	1012
5	1392
4	1151
3	497
2	155
1	736

Tabelle 10 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 20.181 um 69,71 % auf 6.112 gesenkt.

3.1.3 Ausgewählte Teilstellungen

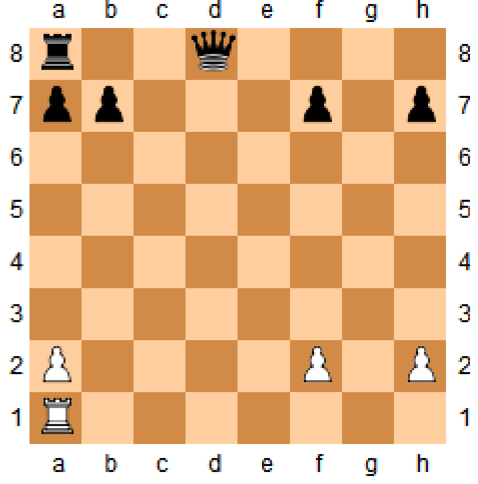


Abbildung 20 Support: 458.452 / 15,65 %

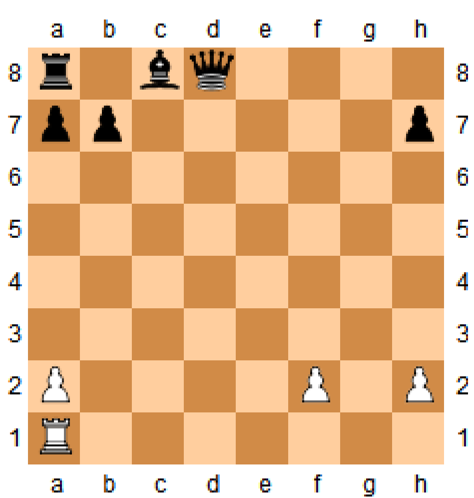


Abbildung 21 Support: 455.076 / 15,53 %

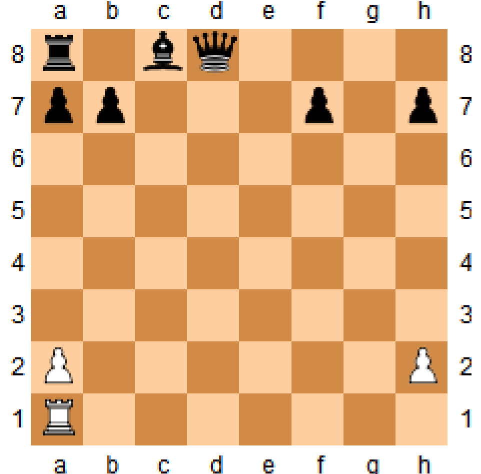


Abbildung 22 Support: 444.976 / 15,19 %

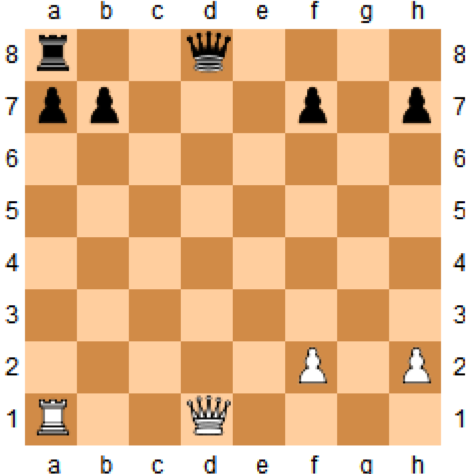


Abbildung 23 Support: 442.503 / 15,10 %

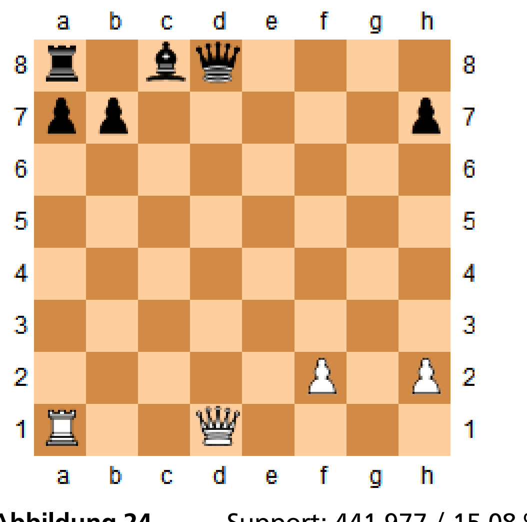


Abbildung 24 Support: 441.977 / 15,08 %

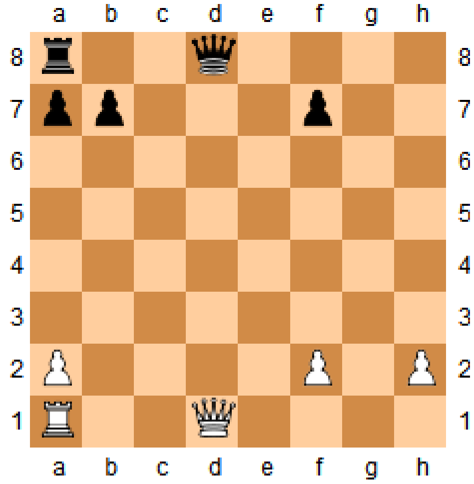


Abbildung 25 Support: 439.642 / 15,00 %

In Abbildung 20 bis Abbildung 25 sind alle Teilstellungen mit 10 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.1.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	223.132.007	100,00 %	± 0,00 %
ZIP	18.760.399	8,41 %	- 91,59 %
RAR	26.711.117	11,97 %	- 88,03 %
Krimp	50.345.069 DB: 50.310.643 CT: 34.426	22,56 %	- 77,44 %

Tabelle 11 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.1.5 Beobachtungen

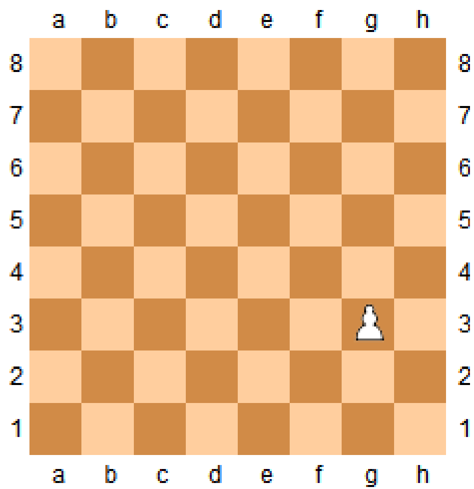


Abbildung 26 Teilstellung mit größtem Support, die nicht Teil der Initialstellung ist

Eine überwältigende Mehrheit der resultierenden Teilstellungen sind lediglich Ausschnitte aus der Grundstellung. Abbildung 26 illustriert die Teilstellung mit größtem Supportwert, die nicht Teil der Initialstellung ist. Diese Beobachtung wird sich in den nachfolgenden Experimenten, bei denen alleine nach Partieausgang gefiltert wurde, fortsetzen.

Eine mögliche Erklärung dafür ist einerseits die Limitierung der Ausgangsdaten auf 100.000 Spiele: Durch die Dünnbesetztheit des Schachfelds und die Menge an unterschiedlichen Figuren treten gleiche Teilstellungen nur mit einer geringen Wahrscheinlichkeit auf. Das sich daraus natürlich ergebende Resultat ist, dass die Initialstellung alle anderen Stellungen verdrängt.

3.2 Experiment 1.2

3.2.1 Konfiguration

Partien:	100.000
Vorverarbeitung:	schwarz gewinnt
Extrahierte Stellungen:	2.386.675
Minimum Support (rel./abs.):	15,00 % / 358.001

3.2.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 16.574

Krimp Miner

Tabelle 12 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
10	8
9	98
8	251
7	632
6	1056
5	1113
4	1066
3	395
2	160
1	736

Tabelle 12 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 16.574 um 33,28 % auf 5.515 gesenkt.

3.2.3 Ausgewählte Teilstellungen

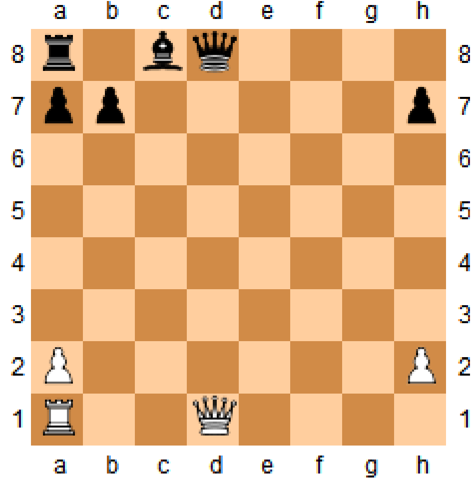


Abbildung 27 Support: 380.366 / 15,94 %

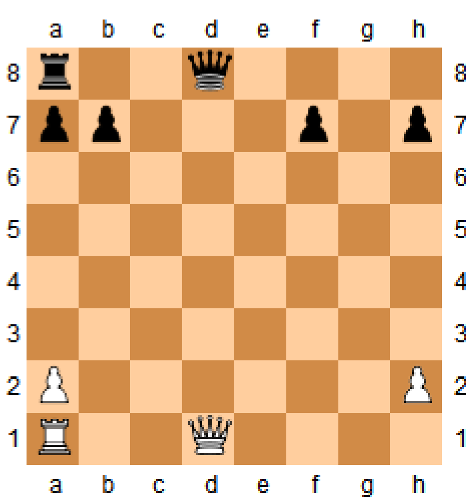


Abbildung 28 Support: 380.148 / 15,93 %

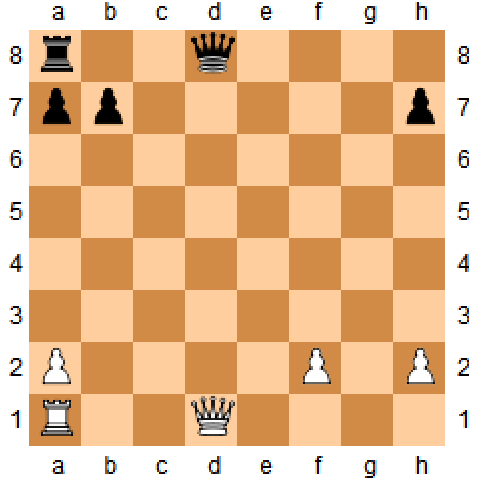


Abbildung 29 Support: 376.930 / 15,79 %

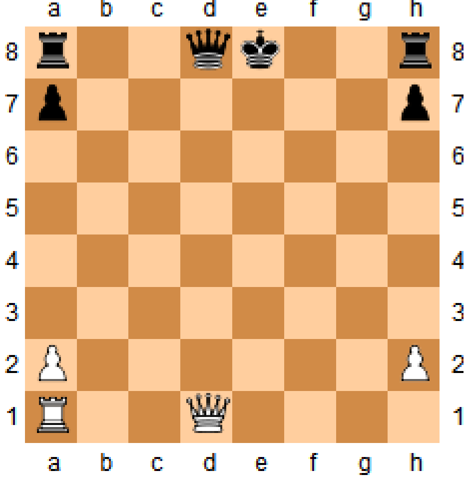


Abbildung 30 Support: 366.663 / 15,36 %

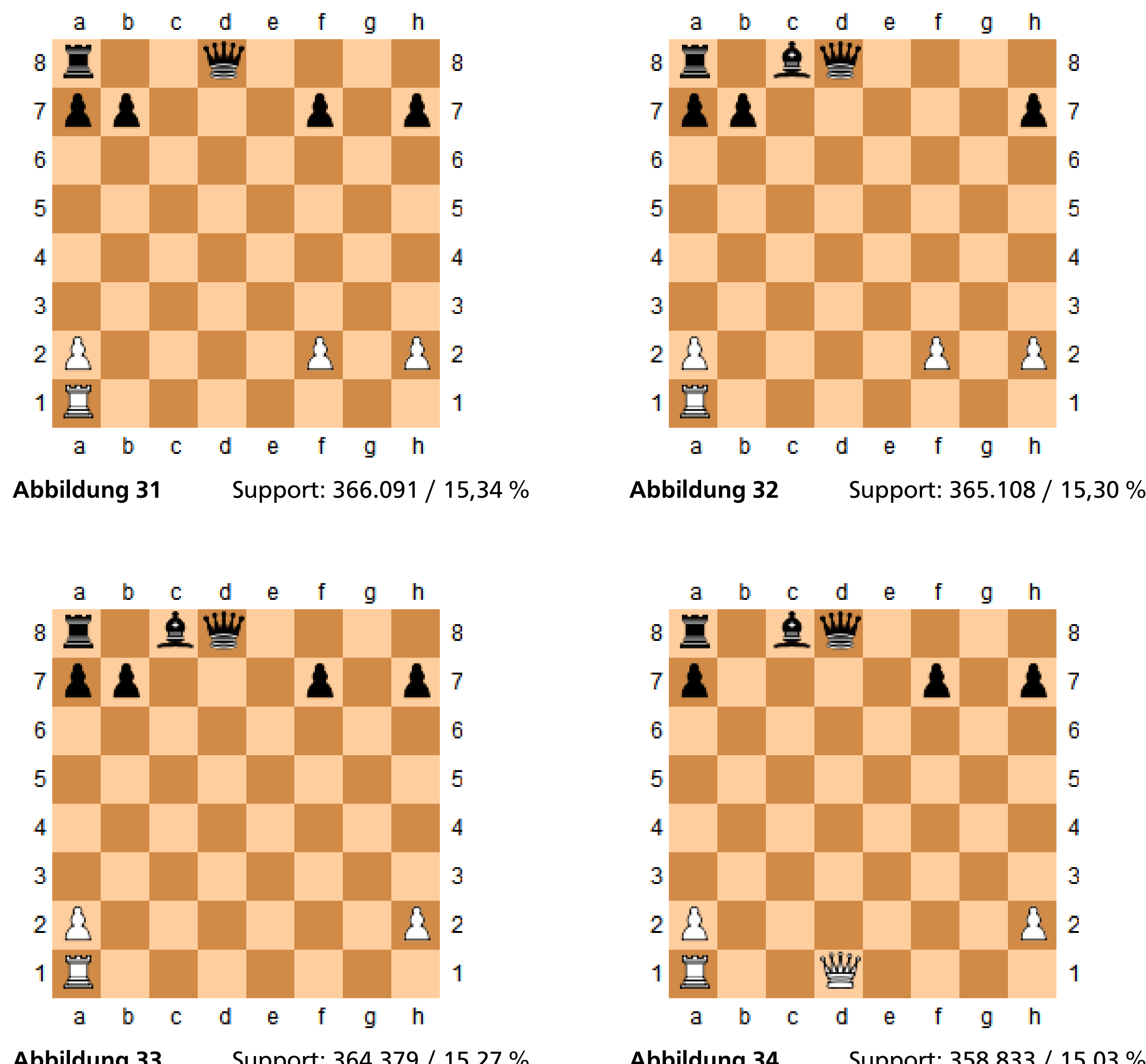


Abbildung 31 Support: 366.091 / 15,34 %

Abbildung 32 Support: 365.108 / 15,30 %

Abbildung 33 Support: 364.379 / 15,27 %

Abbildung 34 Support: 358.833 / 15,03 %

In Abbildung 27 bis Abbildung 34 sind alle Teilstellungen mit 10 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.2.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	182.332.583	100,00 %	± 0,00 %
ZIP	14.575.973	7,99 %	- 92,01 %
RAR	21.869.991	11,99 %	- 88,01 %
Krimp	41.367.461 DB: 41.336.586 CT: 30.875	22,69 %	- 77,31 %

Tabelle 13 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.2.5 Beobachtungen

Wie auch schon in Experiment 1.1 besteht ein großer Teil der gefundenen Teilstellungen lediglich aus Teilen der Initialstellung.

3.3 Experiment 1.3

3.3.1 Konfiguration

Partien:	100.000
Vorverarbeitung:	unentschieden
Extrahierte Stellungen:	2.346.926
Minimum Support (rel./abs.):	15,00 % / 352.038

3.3.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 20.181

Krimp Miner

Tabelle 14 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
10	6
9	47
8	197
7	919
6	1012
5	1392
4	1151
3	497
2	155
1	736

Tabelle 14 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 20.181 um 69,71 % auf 6.112 gesenkt.

3.3.3 Ausgewählte Teilstellungen

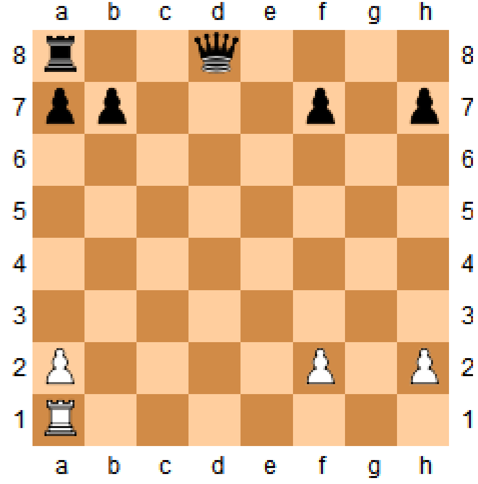


Abbildung 35 Support: 458.452 / 15,65 %

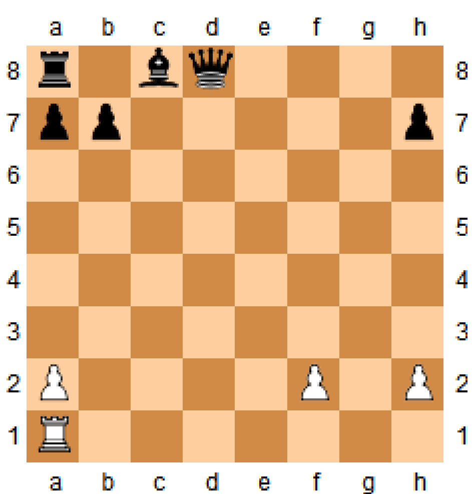


Abbildung 36 Support: 455.076 / 15,53 %

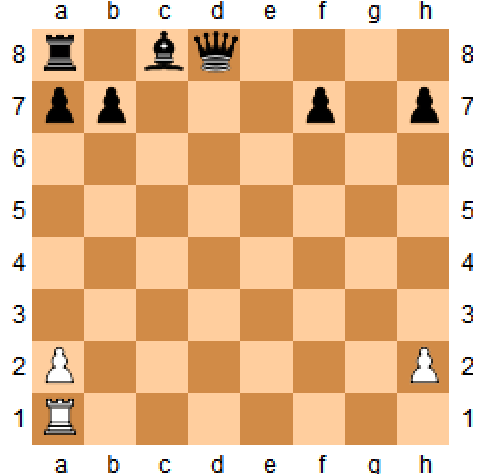


Abbildung 37 Support: 444.976 / 15,19 %

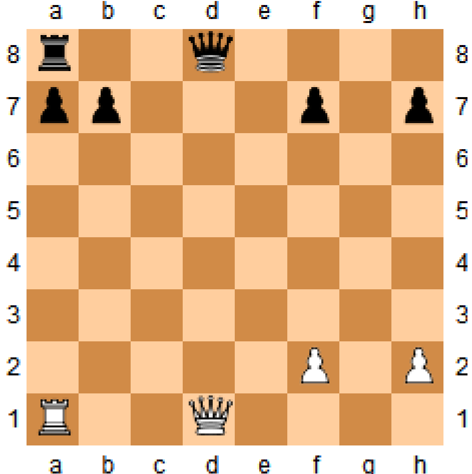


Abbildung 38 Support: 442.503 / 15,10 %

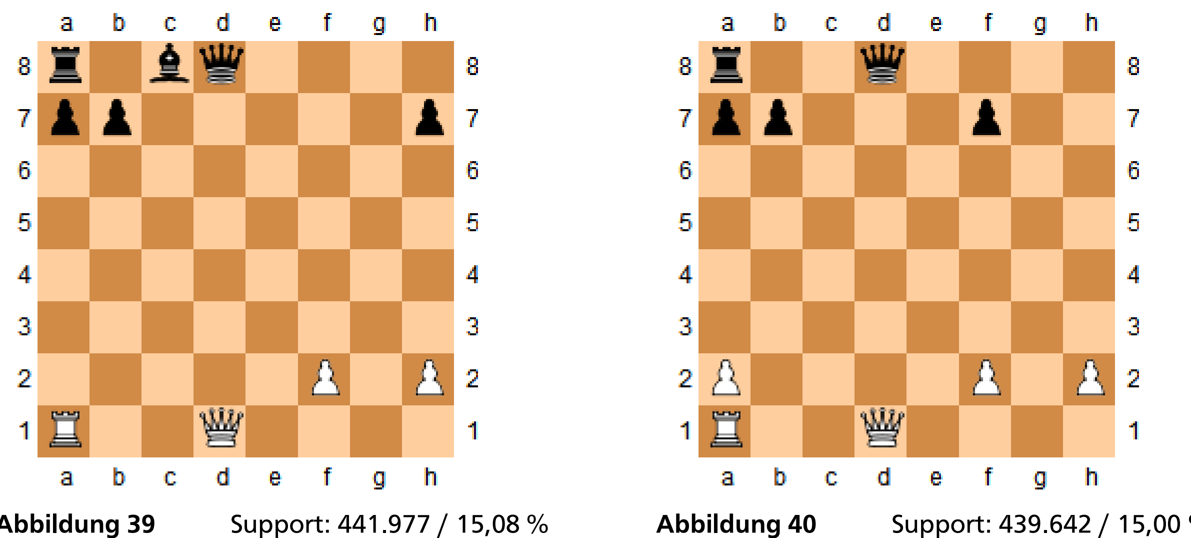


Abbildung 39 Support: 441.977 / 15,08 %

Abbildung 40 Support: 439.642 / 15,00 %

In Abbildung 35 bis Abbildung 40 sind alle Teilstellungen mit 10 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.3.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	176.848.640	100,00 %	± 0,00 %
ZIP	13.915.569	7,87 %	- 92,13 %
RAR	19.952.774	11,28 %	- 88,72 %
Krimp	37.661.167 DB: 37.615.193 CT: 45.974	21,30 %	- 78,70 %

Tabelle 15 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.3.5 Beobachtungen

Wie auch schon in den Experimenten 1.1 und 1.2 besteht ein großer Teil der gefundenen Teilstellungen lediglich aus Teilen der Initialstellung.

3.4 Experiment 2.1

3.4.1 Konfiguration

Partien:	10.000
Vorverarbeitung:	Eröffnung
Extrahierte Stellungen:	200.000
Minimum Support (rel./abs.):	30,00 % / 60.000

3.4.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 6.194.620

Krimp Miner

Tabelle 16 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
15	4
14	31
13	161
12	292
11	299
10	230
9	169
8	144
7	103
6	93
5	88
4	71
3	57
2	61
1	517

Tabelle 16 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 6.194.620 um 99,96 % auf 2.320 gesenkt.

3.4.3 Ausgewählte Teilstellungen

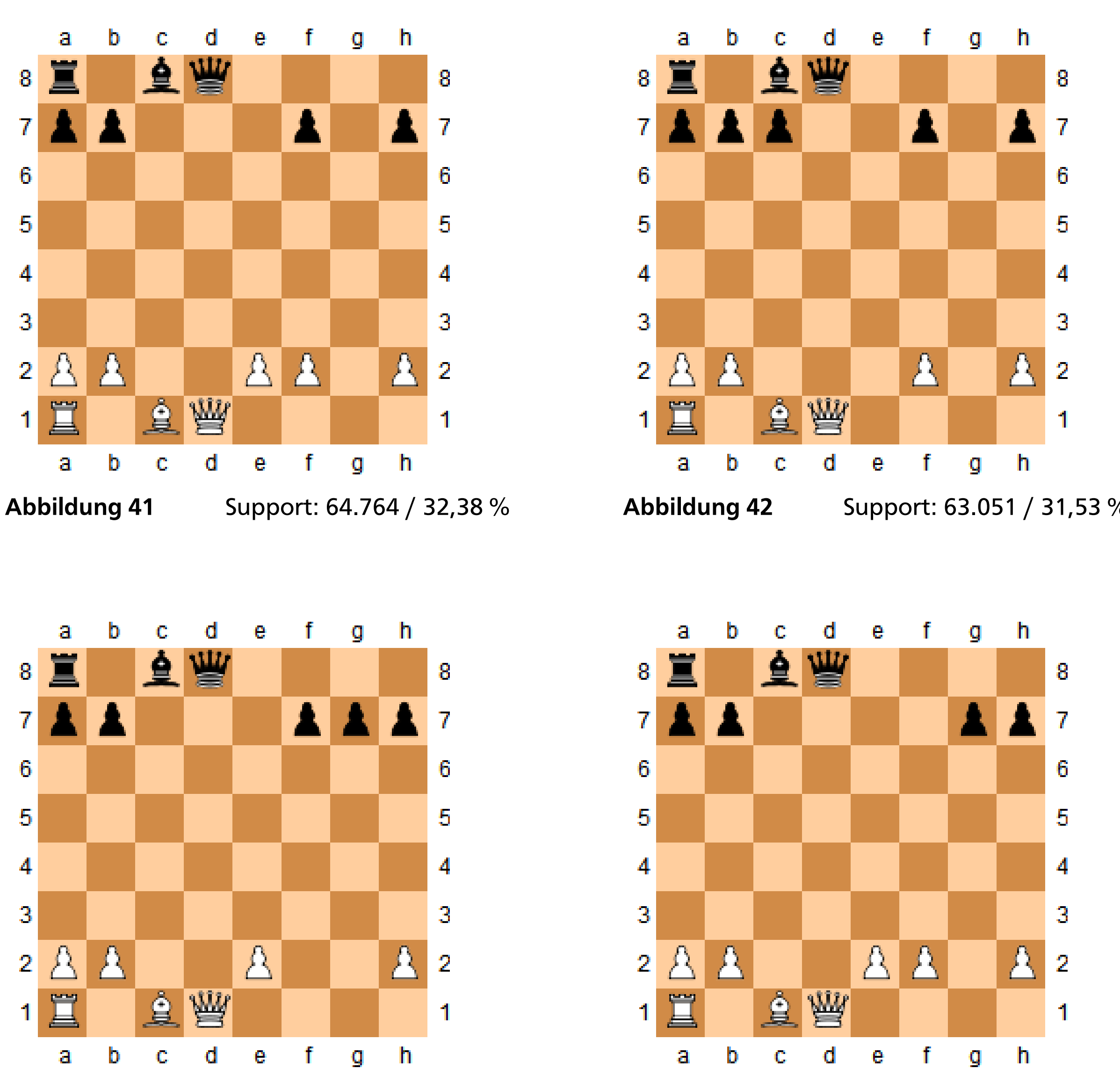


Abbildung 41 Support: 64.764 / 32,38 %

Abbildung 42 Support: 63.051 / 31,53 %

Abbildung 43 Support: 60.491 / 30,25 %

Abbildung 44 Support: 60.189 / 30,09 %

In Abbildung 41 bis Abbildung 44 sind alle Teilstellungen mit 15 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.4.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	18.007.368	100,00 %	± 0,00 %
ZIP	1.129.176	6,27 %	- 93,73 %
RAR	984.019	5,46 %	- 94,54 %
Krimp	2.083.169 DB: 2.066.766 CT: 16.403	11,57 %	- 88,43 %

Tabelle 17 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.4.5 Beobachtungen

Betrachtet man nur die ersten 20 Halbzüge, ist zunächst ein noch höherer Minimumsupport nötig als in den Experimenten 1.1 bis 1.3. Erst ein Wert von 30,00 % erlaubt eine (zeitnahe) Terminierung des Algorithmus. Die meisten Ergebnisse stellen – wie zu erwarten war – fast ausschließlich Teile der Initialstellung dar.

Die Vorbedingung, nur die Eröffnungen der Partien zu komprimieren, verdichtet jedoch die Ergebnisse insoweit, dass mehr Teilstellungen gefunden werden, auf denen sich eine größere Anzahl an Figuren befindet.

Das Experiment 2.4 begegnet diesem Problem, indem dort alle Figuren, die sich auf ihren Initialstellungen befinden, aus der Datenmenge entfernt wurden. Trotzdem sind Figuren der Grundstellung nicht unwichtig: Wenn die sie verdeckenden Bauern weiterziehen oder geschlagen werden, erhöhen sich auch ihre Aktionsradien.

3.5 Experiment 2.2

3.5.1 Konfiguration

Partien:	10.000
Vorverarbeitung:	Mittelspiel
Extrahierte Stellungen:	88.677
Minimum Support (rel./abs.):	0,1 % / 88

3.5.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 6.572.980

Krimp Miner

Tabelle 18 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
12	6
11	11
10	46
9	136
8	563
7	802
6	1010
5	1198
4	1636
3	2110
2	2870
1	736

Tabelle 18 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 6.572.980 um 99,84 % auf 10.388 gesenkt.

3.5.3 Ausgewählte Teilstellungen

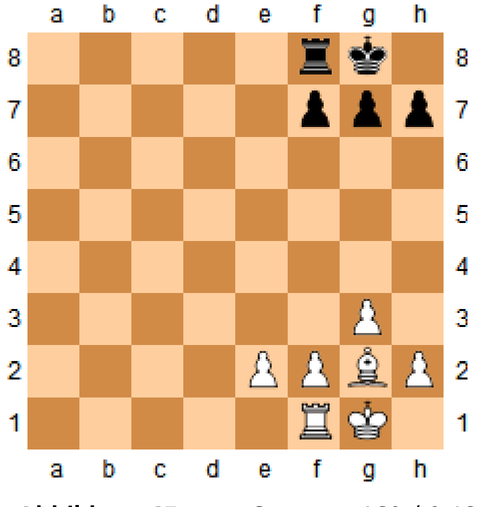


Abbildung 45 Support: 160 / 0,18 %

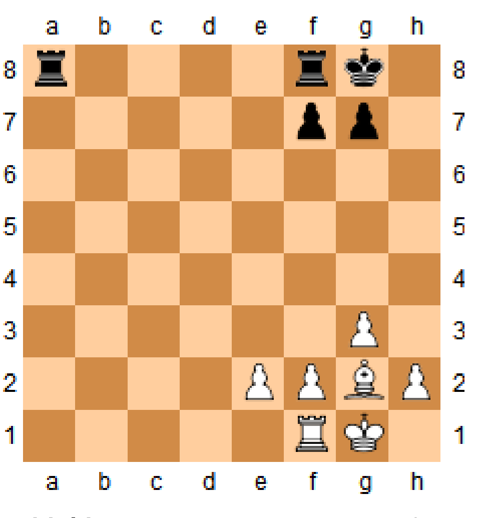


Abbildung 46 Support: 154 / 0,17 %

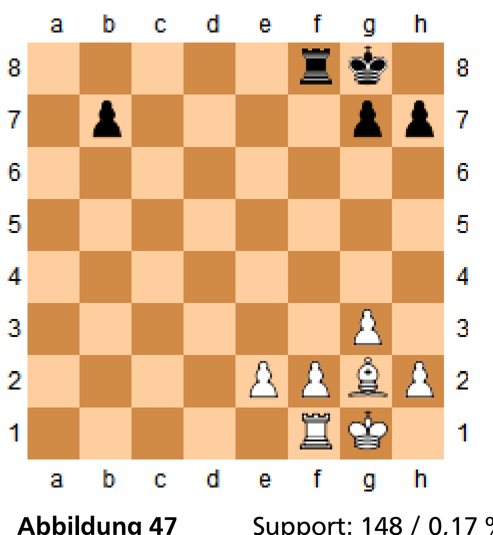


Abbildung 47 Support: 148 / 0,17 %

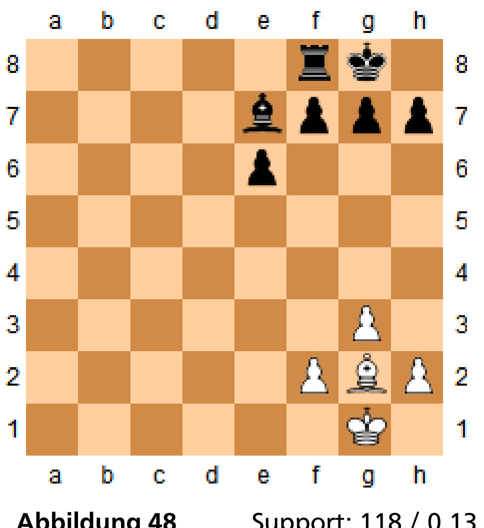


Abbildung 48 Support: 118 / 0,13 %

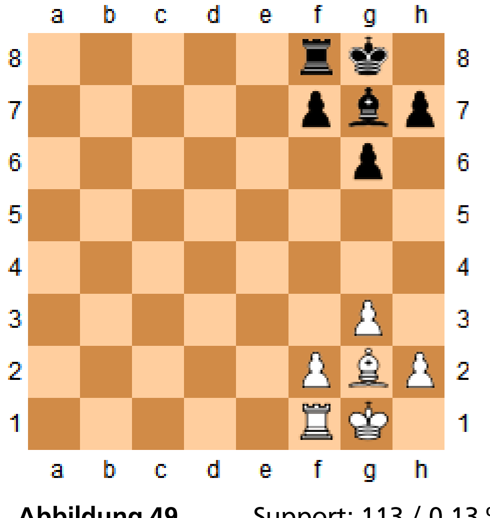


Abbildung 49 Support: 113 / 0,13 %

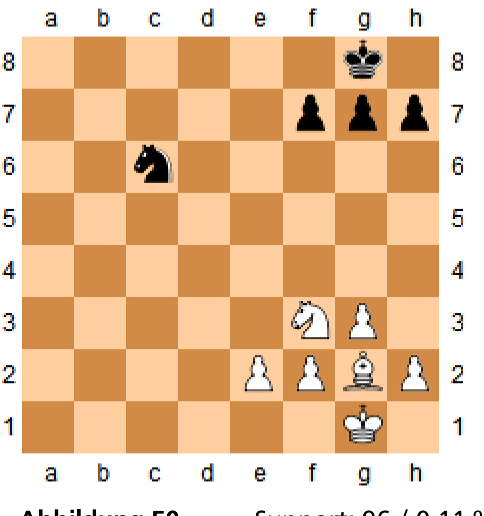


Abbildung 50 Support: 96 / 0,11 %

In Abbildung 45 bis Abbildung 50 sind alle Teilstellungen mit 15 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.5.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	5.067.633	100,00 %	± 0,00 %
ZIP	773.648	15,27 %	- 84,73 %
RAR	855.805	16,89 %	- 83,11 %
Krimp	1.116.053 DB: 1.057.798 CT: 58.255	22,02 %	- 77,98 %

Tabelle 19 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.5.5 Beobachtungen

In diesem Experiment sind die in den Abbildungen dargestellten, größten Teilstellungen zum ersten Mal nicht nur Teile der Initialstellungen. Sie zeigen den Drang der Schachspieler, im Mittelspiel eine königsseitige Rochade anzustreben und ihren König mit bis zu drei Bauern und bis zu zwei höherwertigen Figuren zu decken (Läufer, Turm, seltener: Springer).

Andere, hier nicht aufgeführte Teilstellungen, beinhalten zum überwiegenden Teil einzelne Figuren – meist Bauern – die sich auf verstreuten Schachfeldern befinden. Größere Chunks werden erst bei sehr niedrigen Supportwerten gefunden, was die Abbildungen der größten ausgewählten Teilstellungen exemplarisch belegen.

3.6 Experiment 2.3

3.6.1 Konfiguration

Partien:	10.000
Vorverarbeitung:	Endspiel
Extrahierte Stellungen:	100.000
Minimum Support (rel./abs.):	0,1 % / 100

3.6.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 22.351.814

Krimp Miner

Tabelle 20 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
14	1
13	10
12	15
11	46
10	187
9	492
8	715
7	904
6	1074
5	1276
4	1756
3	2213
2	2995
1	736

Tabelle 20 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 22.351.814 um 99,94 % auf 12.420 gesenkt.

3.6.3 Ausgewählte Teilstellungen

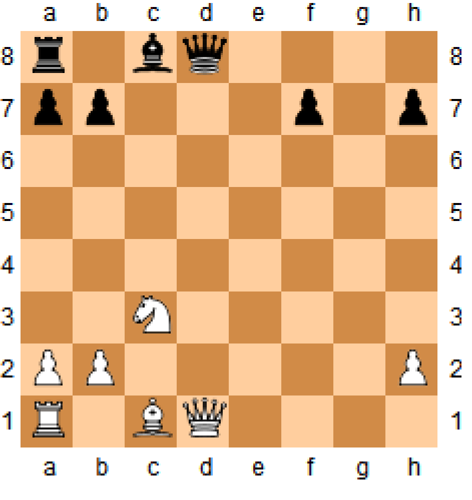


Abbildung 51 Support: 177 / 0,18 %

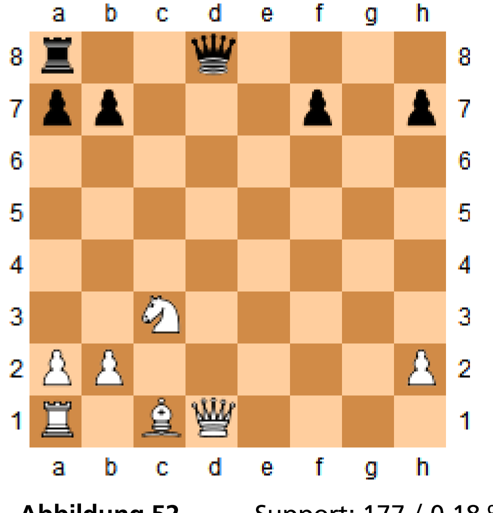


Abbildung 52 Support: 177 / 0,18 %

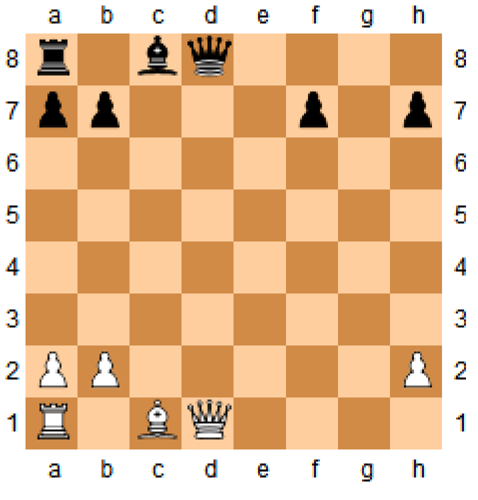


Abbildung 53 Support: 173 / 0,17 %

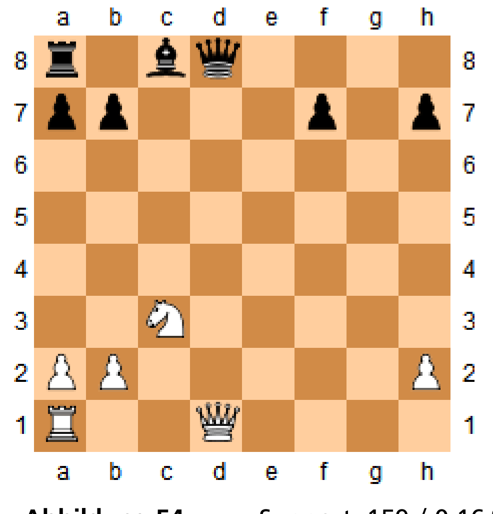


Abbildung 54 Support: 159 / 0,16 %

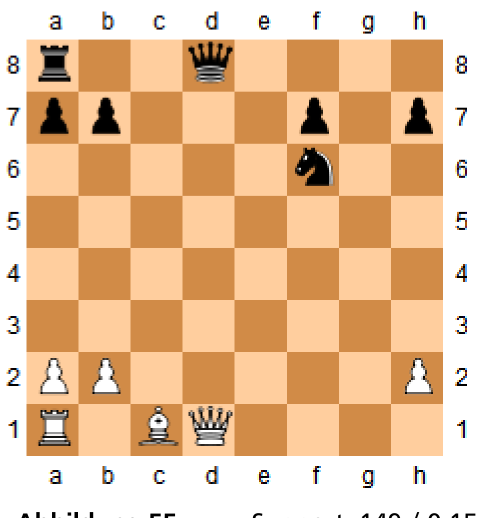


Abbildung 55 Support: 149 / 0,15 %

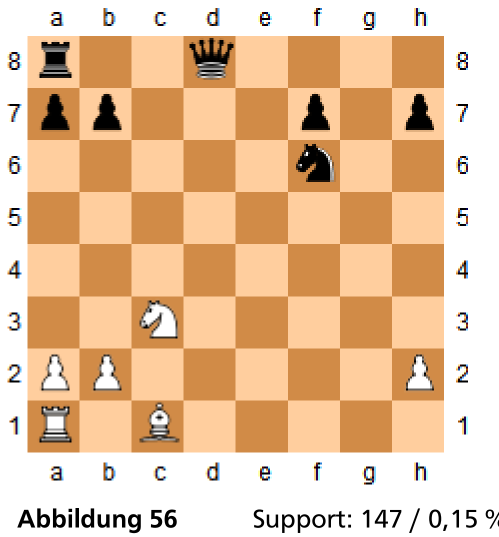


Abbildung 56 Support: 147 / 0,15 %

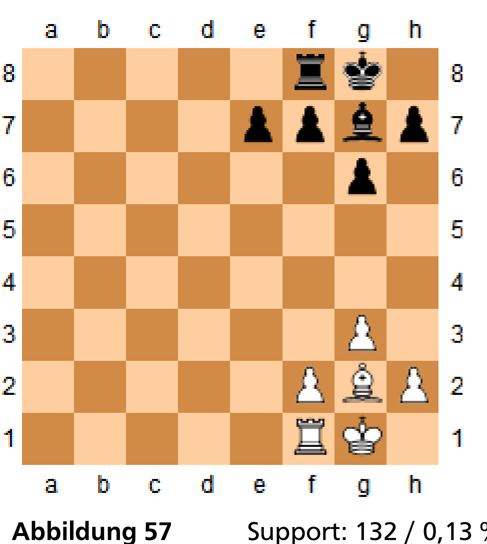


Abbildung 57 Support: 132 / 0,13 %

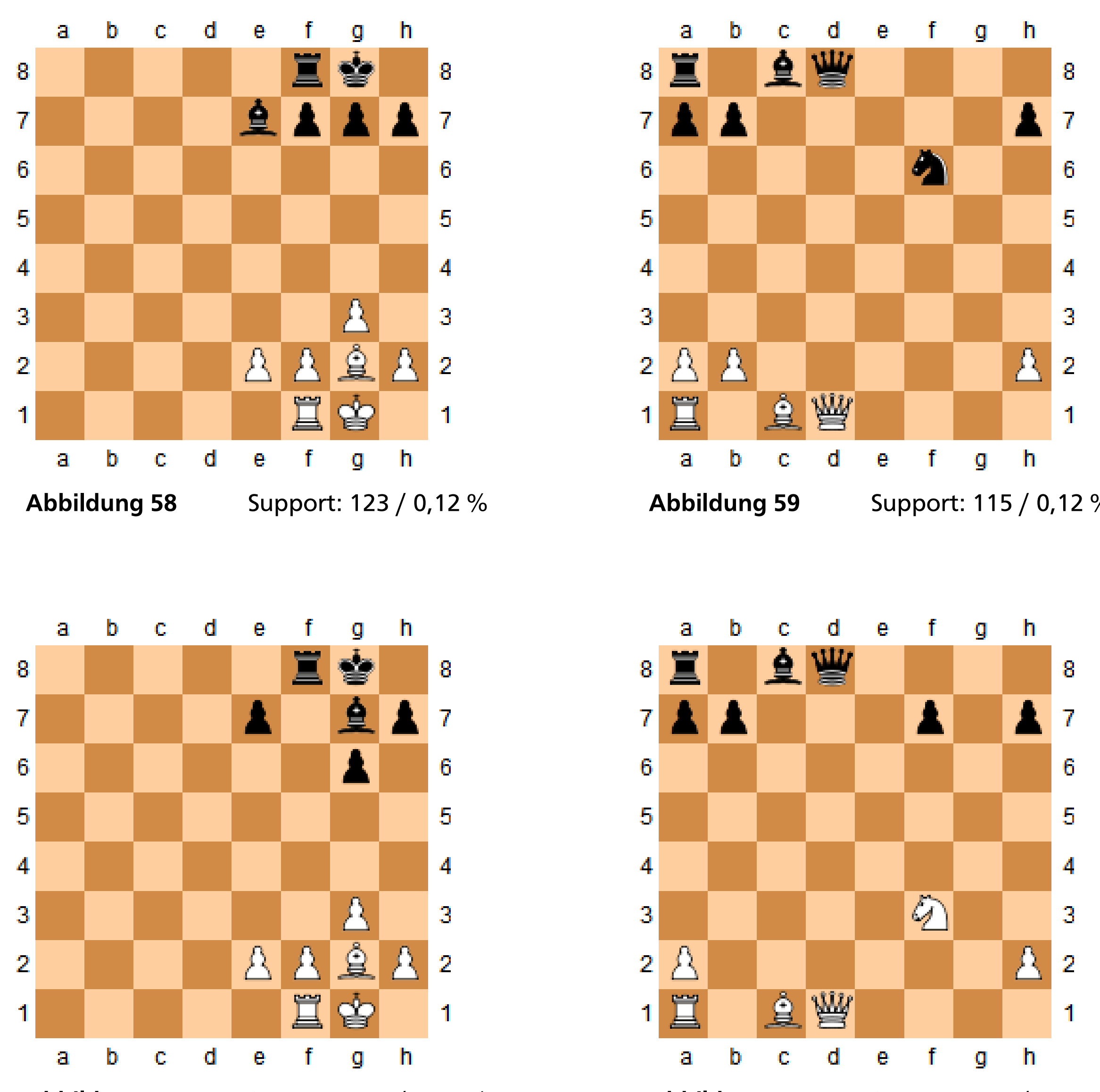


Abbildung 58 Support: 123 / 0,12 %

Abbildung 59 Support: 115 / 0,12 %

Abbildung 60 Support: 110 / 0,11 %

Abbildung 61 Support: 102 / 0,10 %

Abbildung 51 zeigt die gefundene Teilstellung mit 14 enthaltenen Figuren. In Abbildung 52 bis Abbildung 61 sind alle Teilstellungen mit 13 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.6.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	5.799.036	100,00 %	± 0,00 %
ZIP	854.652	14,74 %	- 85,26 %
RAR	967.031	16,68 %	- 83,32 %
Krimp	1.257.697 DB: 1.189.198 CT: 68.499	21,69 %	- 78,31 %

Tabelle 21 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.6.5 Beobachtungen

Parallel zu den bereits bei der Analyse der Mittelspiele gefundenen Teilstellungen, in denen eine Königsrochade durchgeführt wurde, finden sich neben den Figuren der Grundstellung insbesondere die – ausgehend von der Initialstellung – in einem Zug nach C6, F3, B3 und F6 bewegten Springer.

3.7 Experiment 2.4

3.7.1 Konfiguration

Partien:	100.000
Vorverarbeitung:	Eröffnung, nicht-initial
Extrahierte Stellungen:	1.900.000
Minimum Support (rel./abs.):	0,1 % / 1.900

3.7.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 2.768.116

Krimp Miner

Tabelle 22 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
18	3
17	25
16	71
15	153
14	289
13	502
12	658
11	796
10	1010

Figuren	Gefundene Teilstellungen
9	1117
8	1225
7	1296
6	1288
5	980
4	951
3	851
2	706
1	575

Tabelle 22 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 2.768.116 um 99,94 % auf 12.496 gesenkt.

3.7.3 Ausgewählte Teilstellungen

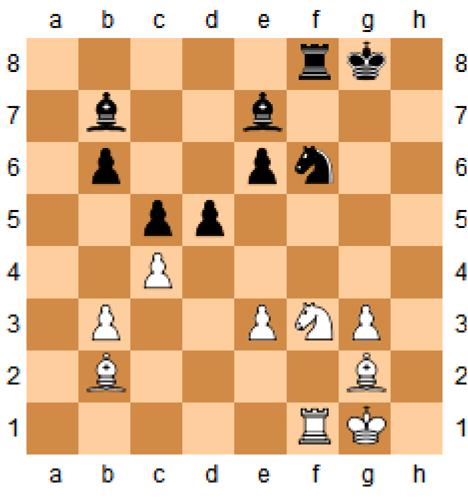


Abbildung 62 Support: 2.326 / 0,12 %

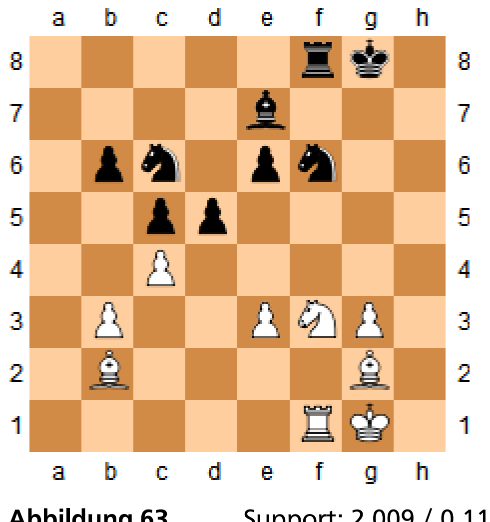


Abbildung 63 Support: 2.009 / 0,11 %

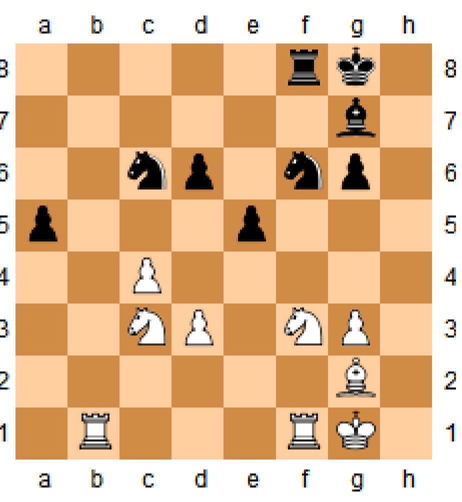


Abbildung 64 Support: 1.993 / 0,10 %

In Abbildung 62 bis Abbildung 64 sind alle Teilstellungen mit 18 enthaltenen Figuren geordnet nach Auftreten in den Ausgangsdaten vor der Kompression visualisiert.

3.7.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	52.323.293	100,00 %	± 0,00 %
ZIP	8.304.878	15,87 %	- 84,13 %
RAR	6.067.004	11,60 %	- 88,40 %
Krimp	5.938.521 DB: 5.852.989 CT: 85.532	11,35 %	- 88,65 %

Tabelle 23 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.7.5 Beobachtungen

Die Ergebnisse dieses Experiments lassen erstmals einige (geometrische) Muster erkennen, die in den Teilstellungen auftauchen.

Bemerkenswert ist das unter anderem in Abbildung 64 auftauchende „Springerquadrat“ (weiße Springer auf C3, F3, schwarze Springer auf C6, F6). Zusammen mit den Bauern auf C4, D3, D6, E5, G3, G6, den Läufern auf G2, G8, den rochierten Königen auf G1, G8 und den Türmen auf F1, F8 ergibt sich ein symmetrisches Muster. Teile davon tauchen in einer Vielzahl von anderen Teilstellungen auf, die hier nicht explizit dargestellt sind.

3.8 Experiment 3.1

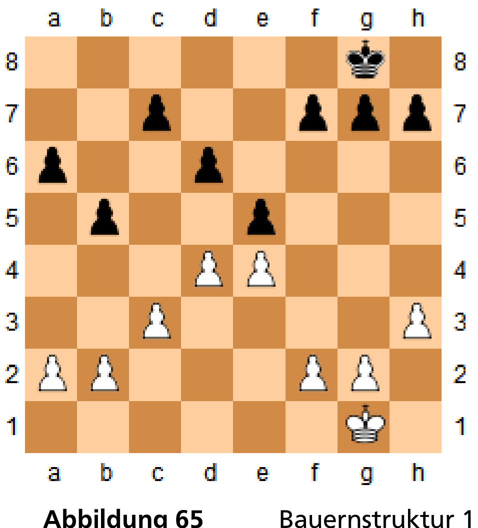


Abbildung 65 Bauernstruktur 1

3.8.1 Konfiguration

Partien:	4.234.538
Vorverarbeitung:	Bauernstruktur 1
Extrahierte Stellungen:	77.400
Minimum Support (rel./abs.):	0,001 % / 1

3.8.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 292.820

Krimp Miner

Tabelle 24 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
14	227
13	103
12	147
11	123
10	94
9	63
8	31
7	16
6	11
5	2
4	2
3	17
2	45
1	154

Tabelle 24 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 292.820 um 99,65 % auf 1.035 gesenkt.

3.8.3 Ausgewählte Teilstellungen & Beobachtungen

Größte Teilstellungen

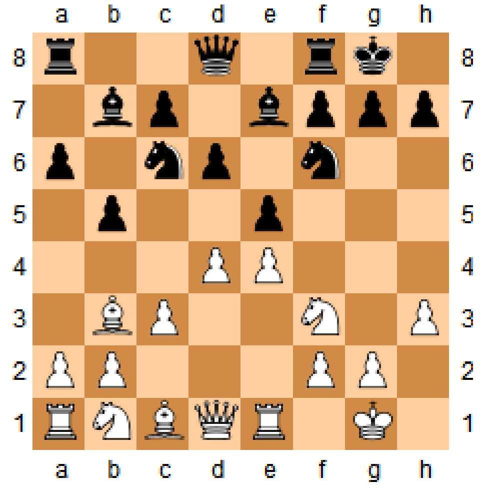


Abbildung 66 Support: 7.870 / 10,17 %

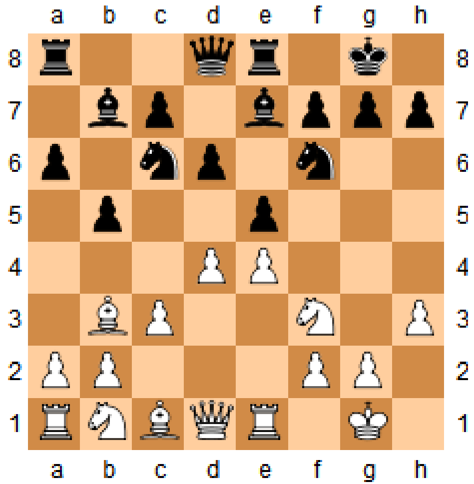


Abbildung 67 Support: 7.004 / 9,05 %

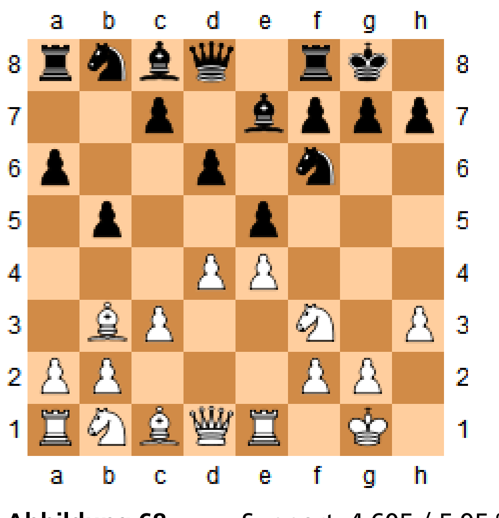


Abbildung 68 Support: 4.605 / 5,95 %

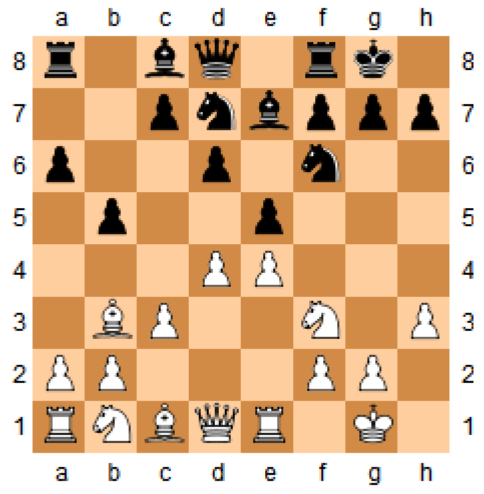


Abbildung 69 Support: 4.577 / 5,91 %

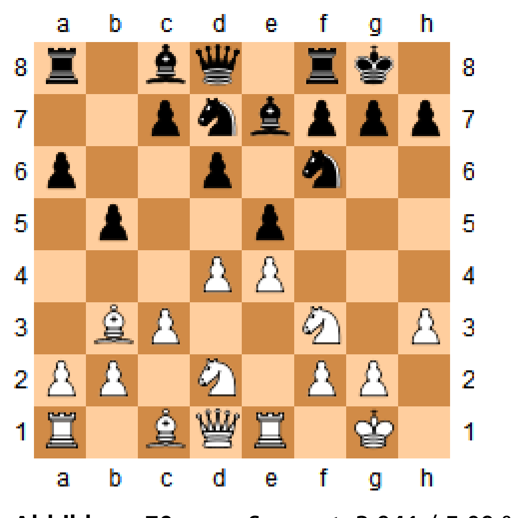


Abbildung 70 Support: 3.941 / 5,09 %

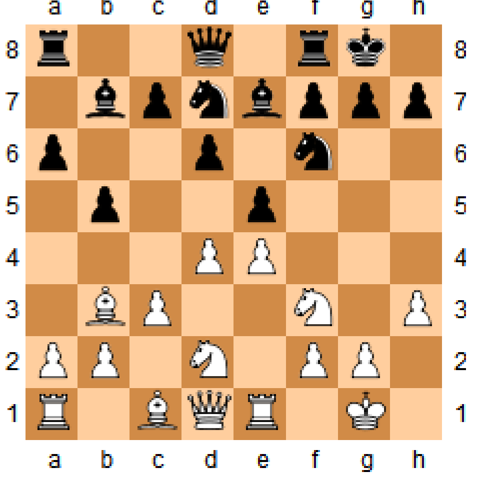


Abbildung 71 Support: 3.848 / 4,97 %

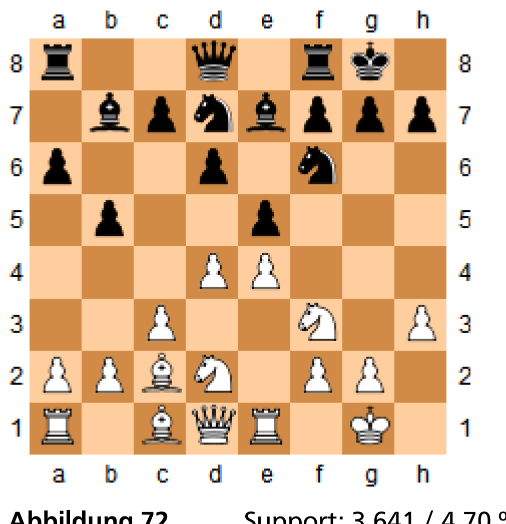


Abbildung 72 Support: 3.641 / 4,70 %

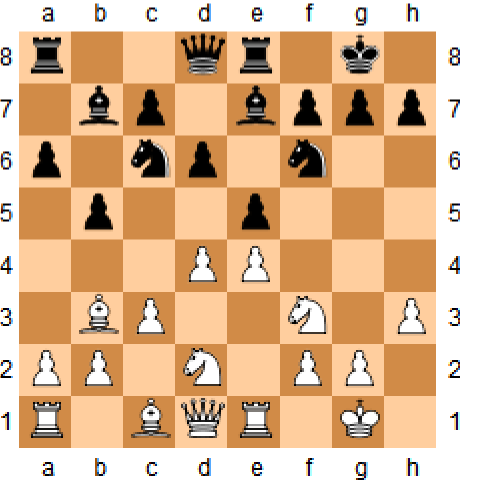


Abbildung 73 Support: 3.535 / 4,57 %

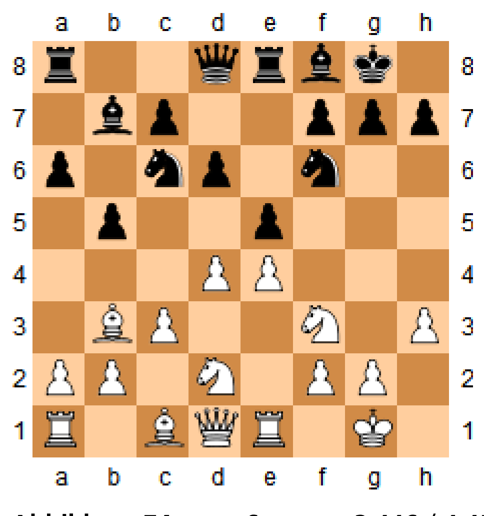


Abbildung 74 Support: 3.446 / 4,45 %

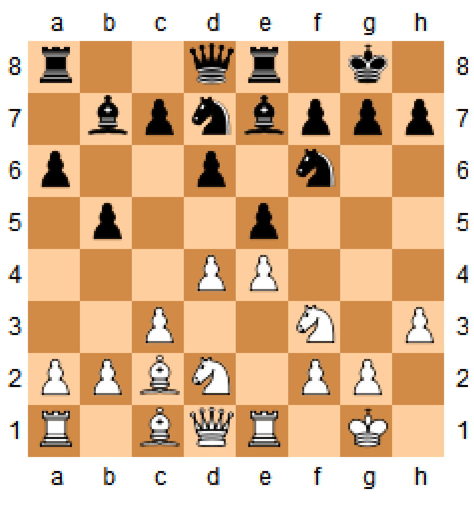


Abbildung 75 Support: 3.249 / 4,20 %

In Abbildung 66 bis Abbildung 75 sind alle größten Teilstellungen mit 32 enthaltenen Figuren (18 Figuren aus der Bauernstruktur, 14 Figuren aus den Ausgangsdaten) absteigend nach Support visualisiert.

Häufigste Teilstellungen

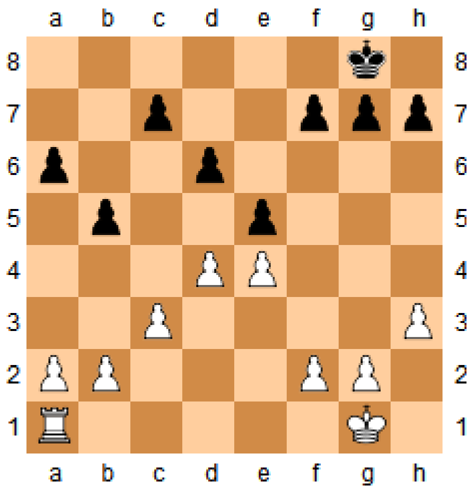


Abbildung 76 N: 19, Support: 77.312 / 26,40 %

Abbildung 76 illustriert die häufigste Stellung. Neben der Bauernstruktur ist nur der weiße Turm auf A1 hinzugekommen. Die entsprechend symmetrische Stellung mit dem schwarzen Turm ist die zweithäufigste Stellung und hat einen Support von 76.591 (26,16 %).

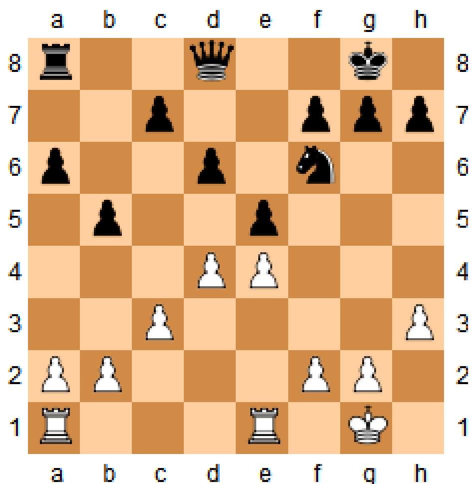


Abbildung 77 N: 23, Support: 65.898 / 85,14 %

Die erste größere Teilstellung mit 5 weiteren Figuren ist in Abbildung 77 dargestellt: Der schwarze Springer auf F6 bedroht den weißen Bauer auf E4, der vom weißen Turm auf E1 geschützt wird. Der weiße Turm auf A1 deckt ebendiesen. Der Springer auf F6 wird von der schwarzen Dame auf D8 geschützt, der Turm auf A8 deckt diese.

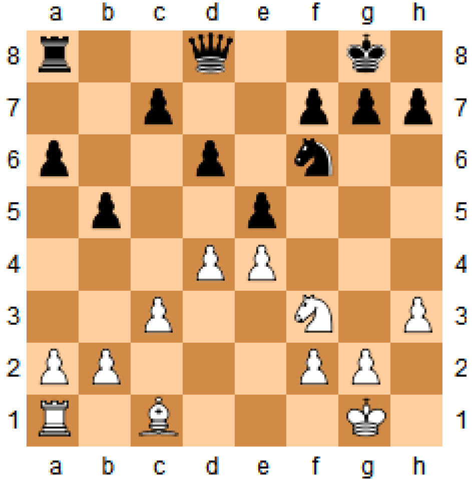


Abbildung 78 N: 24, Support: 60.209 / 77,79 %

In der gefundenen Teilstellung, die in Abbildung 78 dargestellt ist, kommen der weiße Läufer auf C1 und der weiße Springer auf F3 hinzu. Mit dem Zug C1-G5 könnte der schwarze Springer auf F6 bewegungsunfähig gemacht werden (die schwarze Dame wäre direkt bedroht).

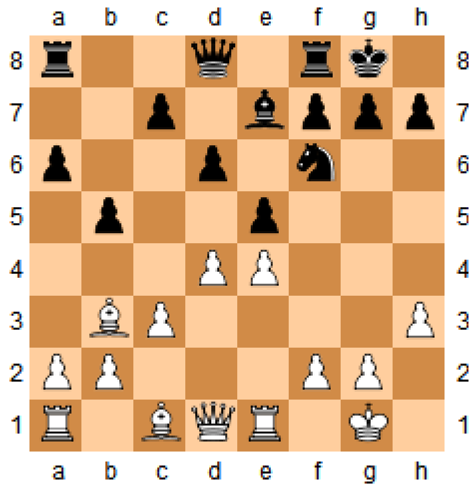


Abbildung 79 N: 28, Support: 28.744 / 37,14 %

In der Teilstellung auf Abbildung 79 steht auf E7 zusätzlich ein immobiler schwarzer Läufer, der bei der zuvor geschilderten Situation (C1-G5, F6 bewegt sich) den einseitigen Damenverlust in einen beidseitigen Läuferverlust umwandelt.

3.8.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	2.917.154	100,00 %	± 0,00 %
ZIP	82.007	2,81 %	- 97,19 %
RAR	69.642	2,39 %	- 91,61 %
Krimp	76.479 DB: 68.297 CT: 8.182	2,62 %	- 97,38 %

Tabelle 25 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.9 Experiment 3.2

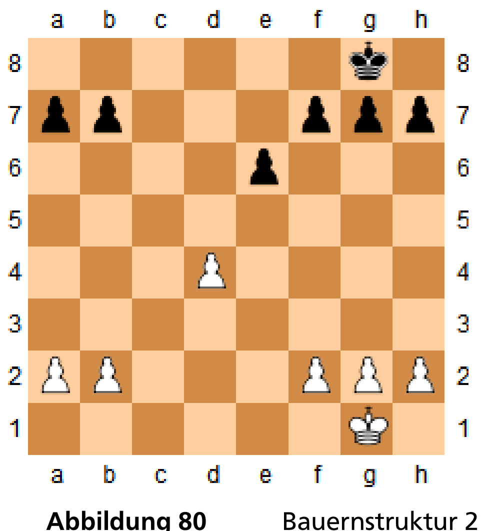


Abbildung 80 Bauernstruktur 2

3.9.1 Konfiguration

Partien:	4.234.538
Vorverarbeitung:	Bauernstruktur 2
Extrahierte Stellungen:	102.718
Minimum Support (rel./abs.):	0,001 % / 1

3.9.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 4.069.951

Krimp Miner

Tabelle 26 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
14	652
13	325
12	480
11	495
10	534
9	503
8	466
7	359
6	246
5	140
4	194
3	295
2	414
1	321

Tabelle 26 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 4.069.951 um 99,87 % auf 5.424 gesenkt.

3.9.3 Ausgewählte Teilstellungen & Beobachtungen

Größte Teilstellungen

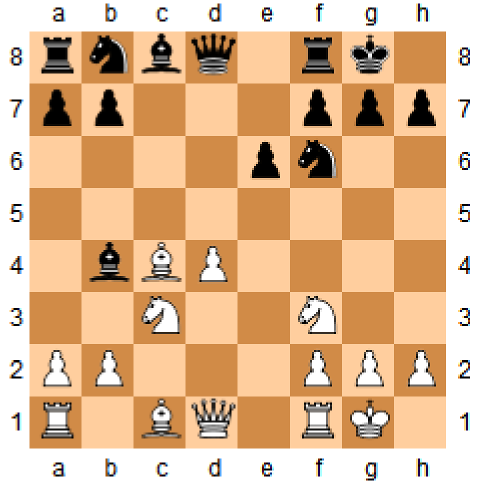


Abbildung 81 Support: 2751 / 2,68 %

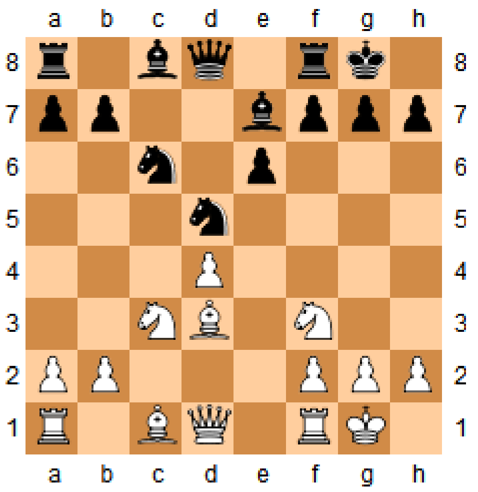


Abbildung 82 Support: 2597 / 2,53 %

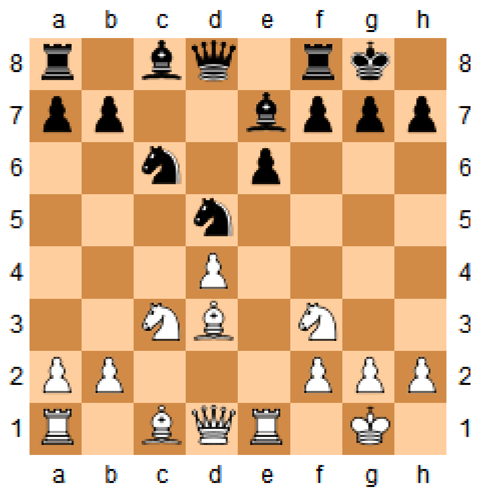


Abbildung 83 Support: 2077 / 2,02 %

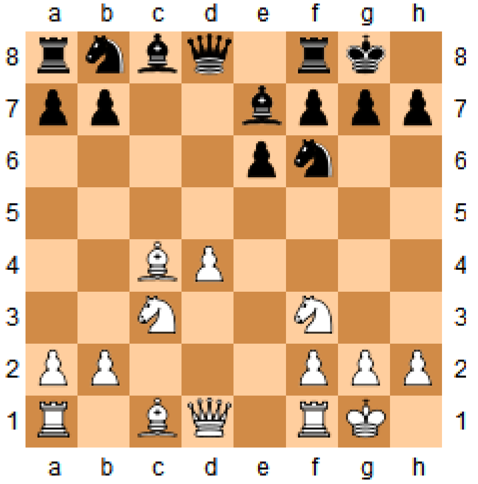


Abbildung 84 Support: 1161 / 1,13 %

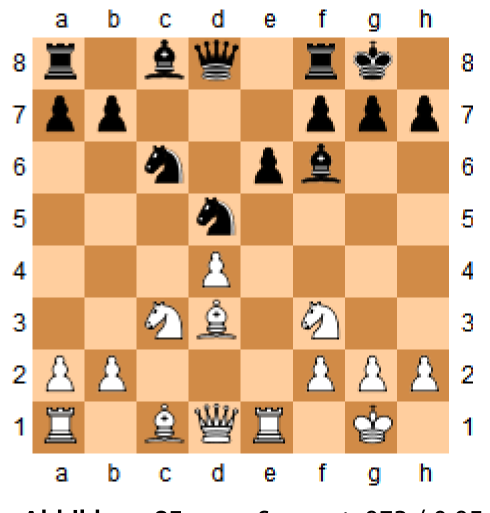
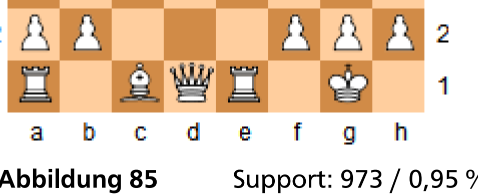



Abbildung 85 Support: 973 / 0,95 %

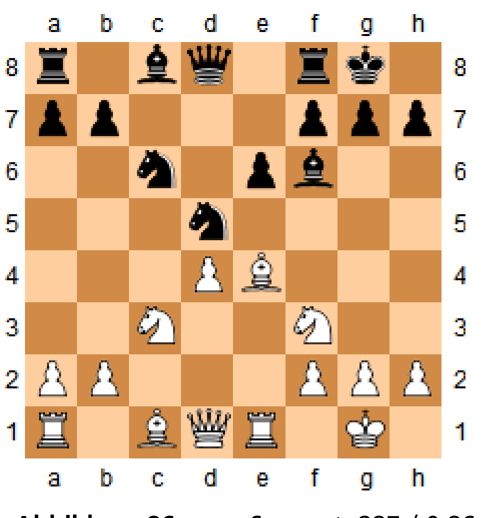


Abbildung 86 Support: 887 / 0,86 %

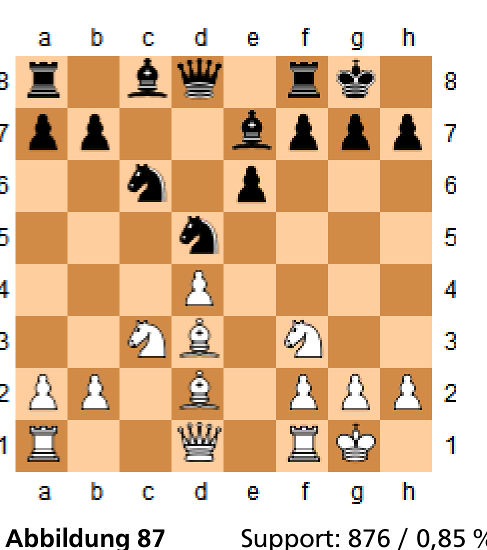


Abbildung 87 Support: 876 / 0,85 %

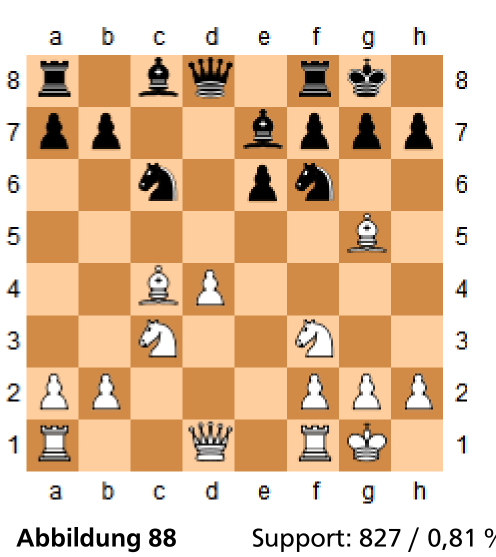


Abbildung 88 Support: 827 / 0,81 %

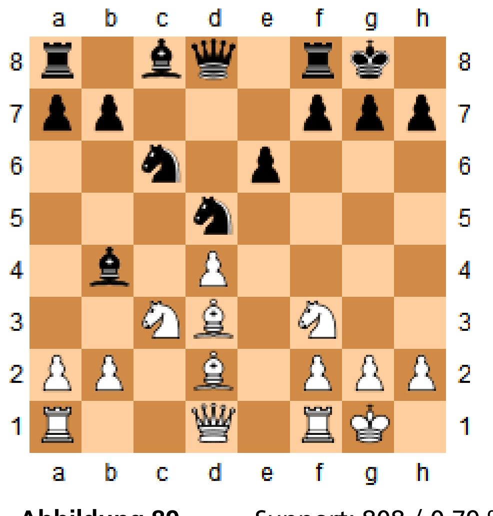


Abbildung 89 Support: 808 / 0,79 %

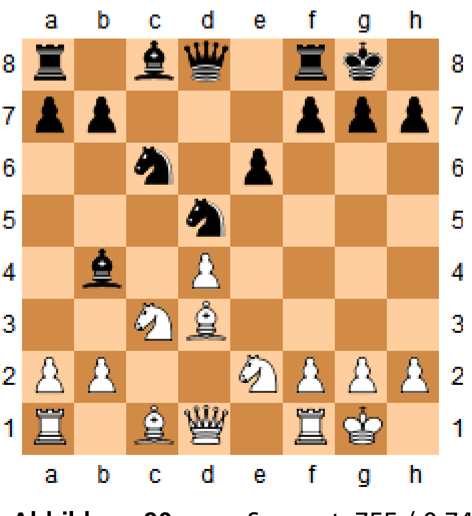


Abbildung 90 Support: 755 / 0,74 %

In Abbildung 81 bis Abbildung 90 sind die zehn größten Teilstellungen mit 28 enthaltenen Figuren (14 Figuren aus der Bauernstruktur, 14 Figuren aus den Ausgangsdaten) absteigend nach Support visualisiert.

Häufigste Teilstellungen

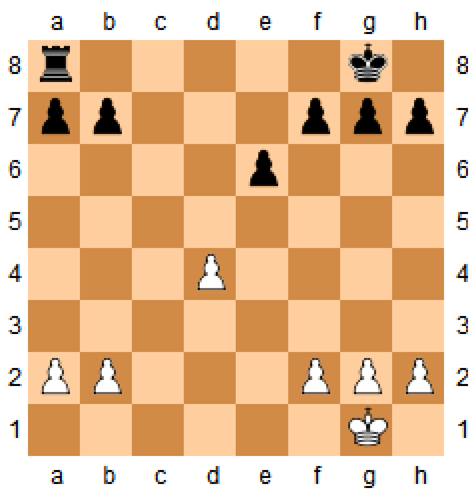


Abbildung 91 N: 15, Support: 95129 / 92,61 %

Abbildung 91 illustriert die häufigste Stellung. Außer der Bauernstruktur ist lediglich der schwarze Turm auf A8 hinzugekommen. Die entsprechend symmetrische Stellung mit dem weißen Turm ist die dritthäufigste Stellung und hat einen Support von 85.668 (83,40 %). Davor – mit einem Supportwert von 91.308 (88,89 %) – steht der schwarze Turm auf F8.

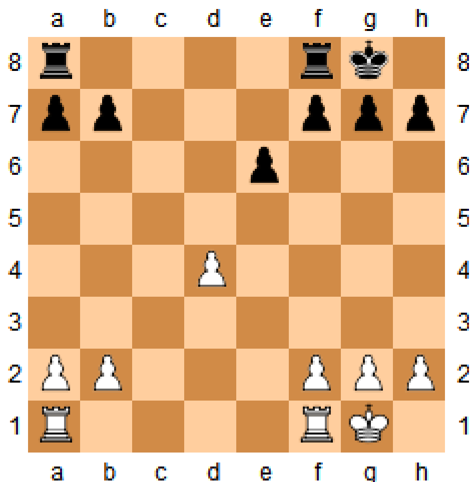


Abbildung 92 N: 18, Support: 53184 / 51,78 %, „Turmquadrat“

Das „Turmquadrat“ (schwarze Türme auf A8, F8; weiße Türme auf A1, F1) besitzt einen Support von 53.184 (51,78 %). Die Damen auf Initialposition stoßen bei einem Support von 33.853 (32,96 %) dazu, gefolgt von den Läufern bei 24.879 (24,22 %). Weitere gefundene Teilstellungen enthalten zum Großteil einzelne Figuren auf Nicht-Initialpositionen. Größere Chunks werden bei höheren Supportwerten nicht gefunden.

3.9.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	3.891.694	100,00 %	± 0,00 %
ZIP	356.678	9,17 %	- 90,83 %
RAR	320.200	8,23 %	- 91,77 %
Krimp	331.078 DB: 290.007 CT: 41.071	8,51 %	- 91,49 %

Tabelle 27 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

3.10 Experiment 3.3

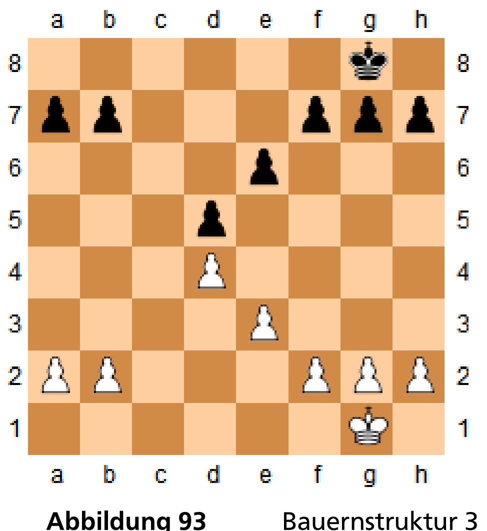


Abbildung 93 Bauernstruktur 3

3.10.1 Konfiguration

Partien:	4.234.538
Vorverarbeitung:	Bauernstruktur 3
Extrahierte Stellungen:	20.561
Minimum Support (rel./abs.):	0,005 % / 1

3.10.2 Quantitative Analyse

Frequent Itemset Miner

Gefundene Frequent Itemsets: 810.917

Krimp Miner

Tabelle 28 listet die Anzahl der gefundenen Teilstellungen auf, die die finale Codetabelle bilden. Die Ergebnisse sind nach der Anzahl der Figuren gruppiert, die sich in den Teilstellungen befinden.

Figuren	Gefundene Teilstellungen
14	67
13	36
12	108
11	103
10	120
9	122
8	148
7	140
6	122
5	68
4	58
3	94
2	157
1	254

Tabelle 28 Gefundene Stellungen über Anzahl der Figuren in der Stellung

Die Gesamtanzahl der Itemsets wurde durch die Komprimierung von 810.917 um 99,80 % auf 1.588 gesenkt.

3.10.3 Ausgewählte Teilstellungen & Beobachtungen

Größte Teilstellungen

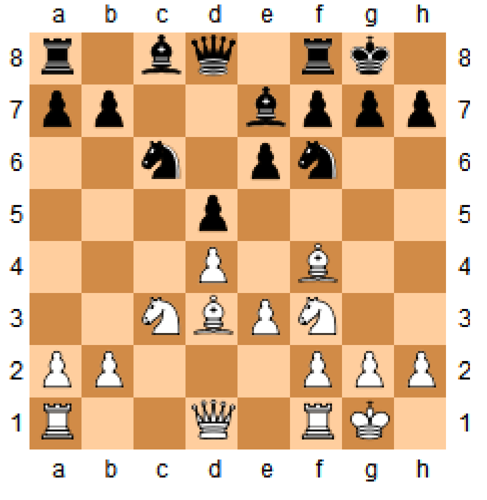


Abbildung 94 Support: 185 / 0,90 %

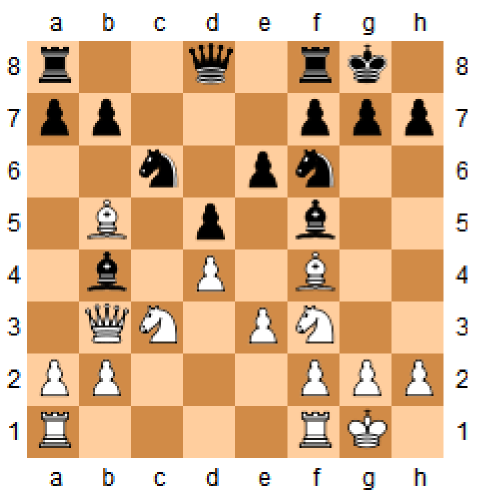


Abbildung 95 Support: 184 / 0,89 %

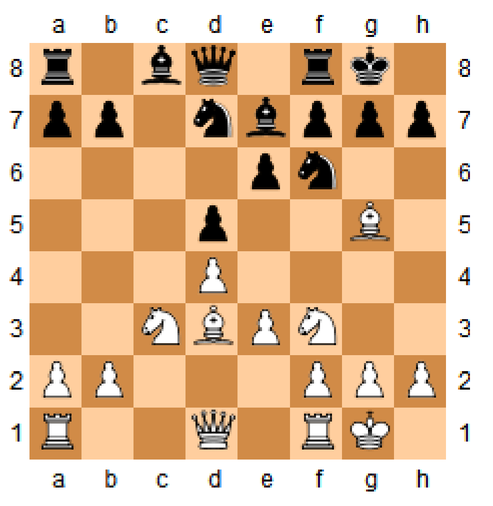


Abbildung 96 Support: 150 / 0,73 %

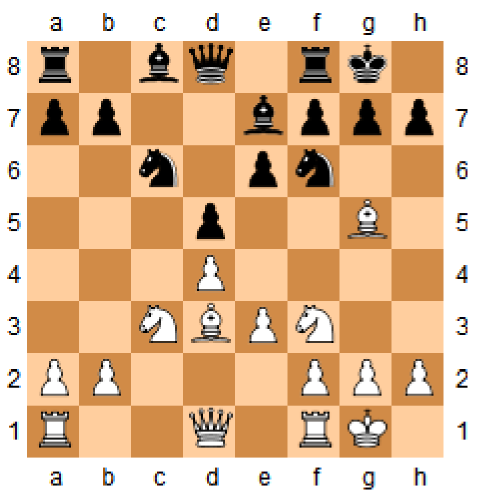


Abbildung 97 Support: 123 / 0,60 %

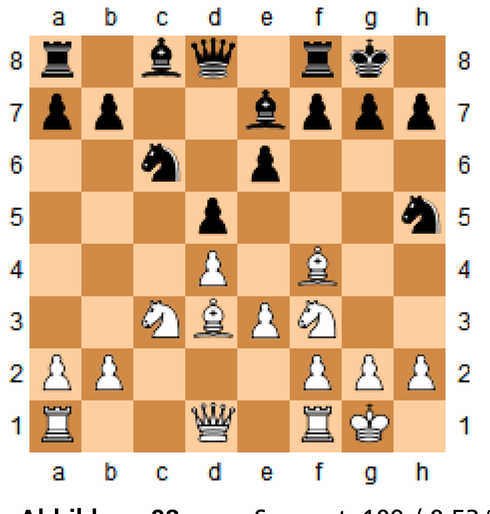


Abbildung 98 Support: 109 / 0,53 %

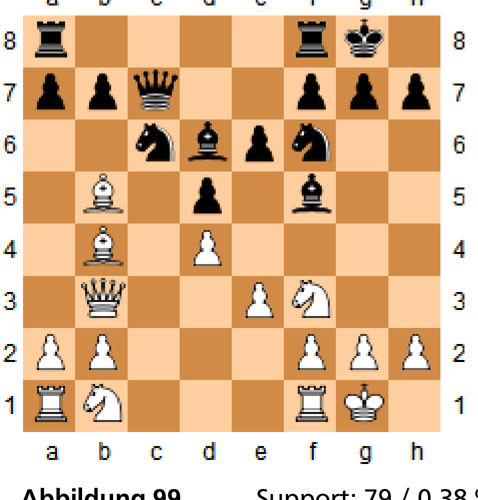


Abbildung 99 Support: 79 / 0,38 %

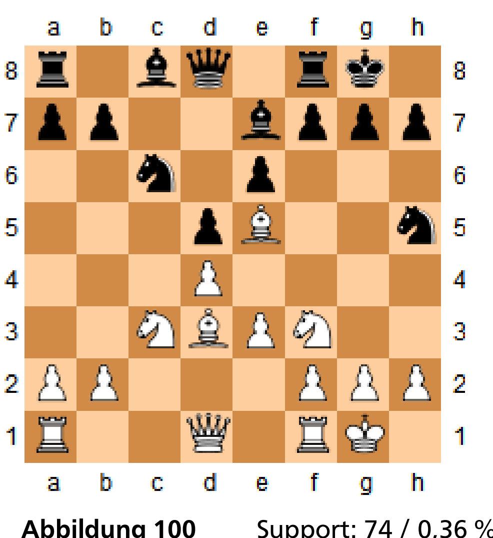


Abbildung 100 Support: 74 / 0,36 %

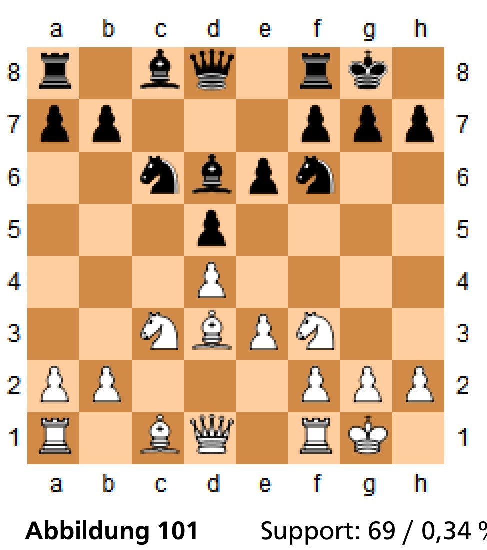


Abbildung 101 Support: 69 / 0,34 %

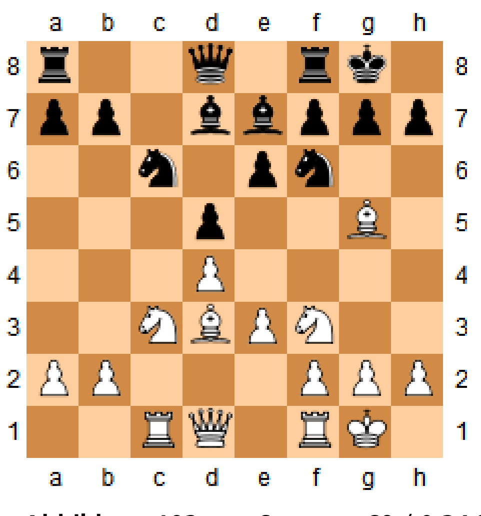


Abbildung 102 Support: 69 / 0,34 %

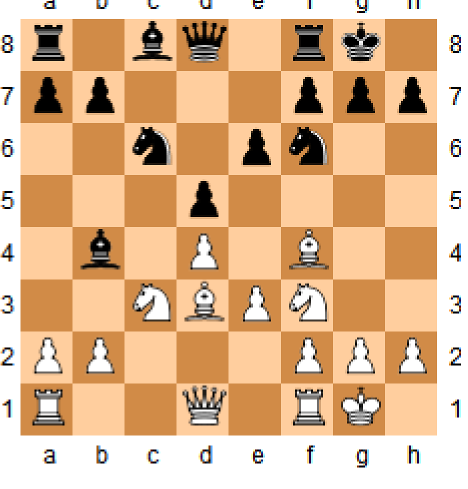


Abbildung 103 Support: 60 / 0,29 %

In Abbildung 94 bis Abbildung 103 sind die zehn größten Teilstellungen mit 30 enthaltenen Figuren (14 Figuren aus der Bauernstruktur, 16 Figuren aus den Ausgangsdaten) absteigend nach Support visualisiert.

Häufigste Teilstellungen

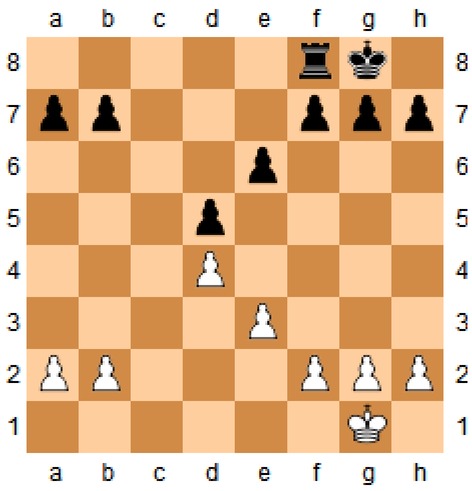


Abbildung 104 N: 17, Support: 16574 / 80,61 %

Abbildung 104 illustriert die häufigste Stellung. Außer der Bauernstruktur ist lediglich der schwarze Turm auf F8 hinzugekommen. Die entsprechend symmetrische Stellung mit dem weißen Turm ist die zweithäufigste Stellung und hat einen Support von 16.091 (78,26 %).

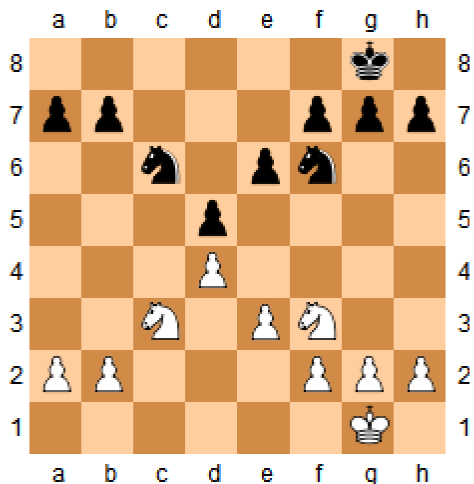


Abbildung 105 N: 20, Support: 8681 / 42,22 %, „Springerquadrat"

Abbildung 105 zeigt das „Springerquadrat" (schwarze Springer auf C6 und F6, weiße Springer auf C3 und F3), welches in 42,22 % aller Schachstellungen der dritten untersuchten Bauernstruktur vorkommt.

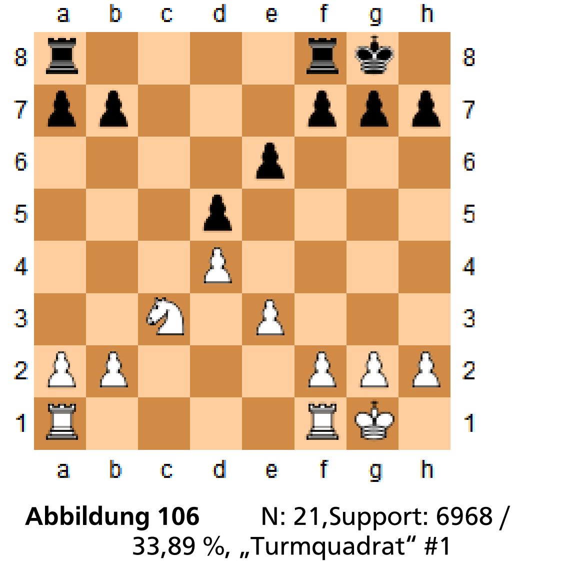


Abbildung 106 N: 21,Support: 6968 / 33,89 %, „Turmquadrat" #1

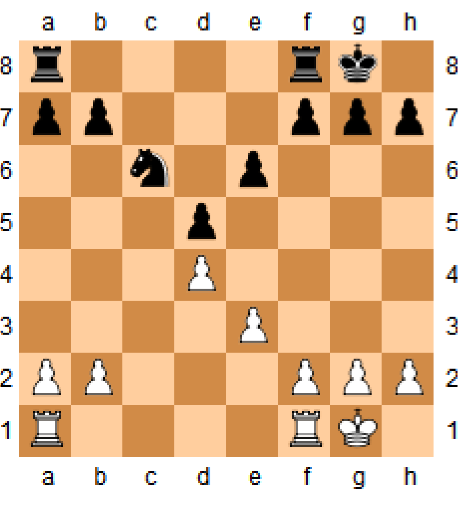


Abbildung 107 N: 21, Support: 6299 / 30,64 %, „Turmquadrat" #2

In Abbildung 106 und Abbildung 107 sind die ersten vollständigen Vorkommnisse des Turmquadrats illustriert. Bei dieser Bauernstruktur ist jeweils der schwarze Springer auf C6 bzw. weiße Springer auf C3 Teil dieser Formation. Bei einem Supportlevel von 4416 (21,48 %) erscheinen das Springer- und Turmquadrat schließlich vereint (siehe Abbildung 108). Mit fallenden Supportwerten kommen des Weiteren noch die Damen auf D6 (schwarz) und D3 (weiß) sowie die selbigen auf ihren Initialpositionen hinzu.

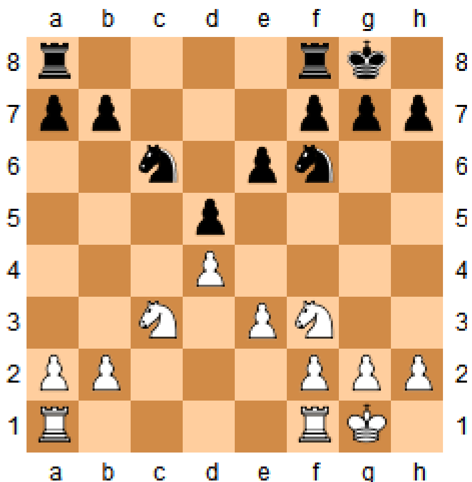


Abbildung 108 N: 24, Support: 4416 / 21,48 %, „Springerquadrat“ und „Turmquadrat“

3.10.4 Analyse der Kompression

Kompression	Größe in Bytes	Relative Größe	Größenveränderung
keine	700.121	100,00 %	± 0,00 %
ZIP	74.139	10,59 %	- 89,41 %
RAR	63.734	9,10 %	- 90,90 %
Krimp	64.713 DB: 55.156 CT: 9.557	9,24 %	- 90,76 %

Tabelle 29 Vergleich von Krimp mit gängigen Kompressionsalgorithmen

4 Realisierung

Die zur Vor- und Nachbereitung der Daten benötigten Programme wurden in Java implementiert. Abbildung 109 illustriert die Paketstruktur, Tabelle 30 enthält einen Überblick über die in den Paketen definierten Funktionalitäten.

Abschnitt 4.1 beschreibt die Verantwortlichkeiten der Klassen, Abschnitt 4.2 erläutert die definierten Dateiformate. Abschnitt 4.3 gibt einen Überblick über verwendete Fremdsoftware und Libraries, Abschnitt 4.4 liefert Hinweise zur Einrichtung und Verwendung der Toolchain.

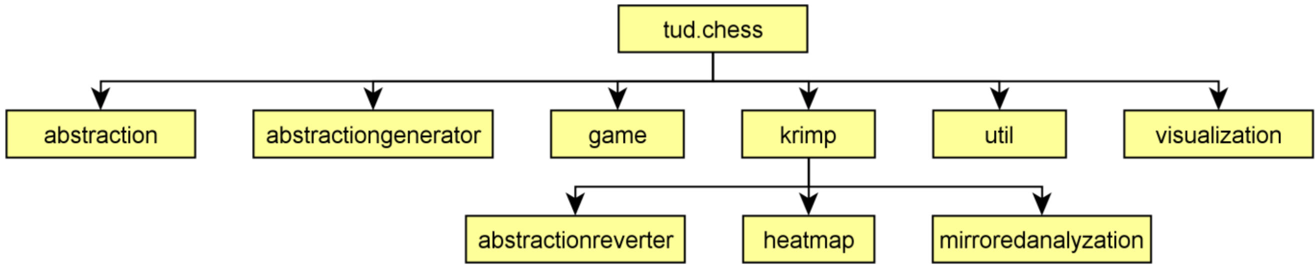


Abbildung 109 Paketstruktur der Java-Implementation

Package	Beschreibung
tud.chess	Wurzelpackage der Implementation.
abstraction	Enthält die Itemklassen, die durch Item-Ids repräsentiert werden.
abstractiongenerator	Enthält Klassen, die Itemklassen aus dem Modell erstellen können.
game	Enthält Modellklassen (Schachbrett, -figur, -farbe, etc.).
krimp	Enthält Hilfsklassen zur Analyse der von Krimp generierten Daten.
abstractionreverter	Enthält Klassen, die die Krimp-Codetabelle in das Objektmodell zurück überführen und die Schachbretter rendern.
heatmap	Enthält Klassen, die Heatmaps aus den Krimp-Daten generieren können.
mirroredanalyzation	Enthält Klassen, die vergleichen können, wie oft eine Teilstellung eines Spielers gespiegelt bei dem anderen Spieler auftritt.
util	Enthält Hilfsklassen zur Bestimmung von Helligkeitswerten aus Farbwerten und zur Verwaltung von Dateien.
visualization	Enthält Klassen zur Visualisierung von Modellklassen.

Tabelle 30 Beschreibung der Paketstruktur

4.1 Responsibilities

Im Folgenden finden sich die Verantwortlichkeiten (Responsibilities) der implementierten Klassen. Ausführbare Klassen sind unterstrichen.

4.1.1 Paket *tud.chess*

Klasse	Responsibility
ItemsetFrequencyAnalyzer	Zählt identische Itemsets der übergebenen Itemset-Datenbank.
ItemsetFrequencyRenderer	Rendert Itemsets aus Frequenzzählungen als PGN-Dateien.
ItemFrequencySorter	Sortiert Frequenzzählungen identischer Itemsets absteigend.
PgnParserWorkerForChessPiecePosition	Wandelt die Zuginformationen aus einem im PGN-Format abgespeicherten Spiel in Schachbrettmodelle um.
PGNtoGameStateParser	Koordiniert die Umwandlung von PGN-Schachspielen in Itemsetdaten.
ThreadedExecutor	Bietet Thread-basierte Parallelisierung.

Tabelle 31 Beschreibung der im Paket *tud.chess* enthaltenen Klassen

4.1.2 Paket *tud.chess.abstraction*

Klasse	Responsibility
Abstraction	Repräsentiert Abstraktionen, die durch eine eindeutige Id dargestellt werden können.
ChessPiecePosition	Verbindet die Informationen über Schachfigur, -farbe und Position der Figur, besitzt eindeutige Id.

Tabelle 32 Beschreibung der im Paket *tud.chess.abstraction* enthaltenen Klassen

4.1.3 Paket *tud.chess.abstractiongenerator*

Klasse	Responsibility
BoardAbstractionGenerator	Generische Klasse, die Abstraktionen aus einem Schachbrett extrahieren kann.
ChessPiecePositionGenerator	Erstellt ChessPiecePosition-Instanzen aus einem übergebenen Schachbrett

Tabelle 33 Beschreibung der im Paket *tud.chess.abstractiongenerator* enthaltenen Klassen

4.1.4 Paket *tud.chess.game*

Klasse	Responsibility
ChessPiece	Repräsentiert eine Schachfigur.
ChessPieceColor	Repräsentiert die Farbe einer Schachfigur (weiß, schwarz).
ChessPieceType	Repräsentiert den Typ einer Schachfigur (König, Dame, etc.).
Dimension	Repräsentiert eine Dimension (X, Y).
GameResult	Repräsentiert das Ergebnis einer Schachpartie (weiß gewinnt, schwarz gewinnt, unentschieden).
GameState	Repräsentiert ein Schachbrett mit aufgestellten Figuren.
Point2D	Repräsentiert einen zweidimensionalen Punkt.

Tabelle 34 Beschreibung der im Paket *tud.chess.game* enthaltenen Klassen

4.1.5 Paket *tud.chess.krimp*

Klasse	Responsibility
ChessPiecePositionDatabaseReader	Liest eine Itemset-Datenbank ein und wandelt die Item-Ids in Instanzen von ChessPiecePosition um.
ChessPiecePositionFrequencies	Repräsentiert einen Häufigkeitsvergleich von Figuren auf Schachfeldern für beide Spieler.
CodeTableItemFrequencies	Repräsentiert die Frequenzinformationen der Krimp-Codetabelle.
DatabaseReader	Bietet Funktionalität, um durch eine Itemset-Datenbank zu iterieren.
ItemIdTranslator	Erlaubt die Rückübersetzung von Krimp-Item-Ids in die Item-Ids der Ausgangsdatenbank

Tabelle 35 Beschreibung der im Paket *tud.chess.krimp* enthaltenen Klassen

4.1.6 Paket *tud.chess.krimp.abstractionreverter*

Klasse	Responsibility
AbstractionReverter	Wandelt Item-Ids in Abstraktionen um.
CodeTableRenderer	Rendert die Schachstellungen der Krimp-Codetabelle als PGN-Dateien
CodeTableStatistics	Erstellt Statistiken über Häufigkeitsverteilungen von Schachstellungen innerhalb der Codetabelle

Tabelle 36 Beschreibung der im Paket *tud.chess.krimp.abstractionreverter* enthaltenen Klassen

4.1.7 Paket *tud.chess.krimp.heatmap*

Klasse	Responsibility
ChessPiecePostitionFrequencyHeuristic	Repräsentiert eine abstrakte Zählheuristik.
HeatmapGenerator	Erzeugt Heatmaps aus gespiegelten Itemset-Analysen.

Tabelle 37 Beschreibung der im Paket *tud.chess.krimp.heatmap* enthaltenen Klassen

4.1.8 Paket *tud.chess.krimp.mirroredanalyzation*

Klasse	Responsibility
MirroredChessPieceAnalyzationReader	Bietet ein Interface zum Lesen von gespiegelten Itemset-Analysen von Schachstellungen.
MirroredChessPiecePositionAnalyzer	Erstellt gespiegelte Itemset-Analysen von Schachstellungen.
MirroredChesspiecePositionAnalyzerWorker	Worker-Thread für gespiegelte Itemset-Analysen von Schachstellungen.

Tabelle 38 Beschreibung der im Paket *tud.chess.krimp.mirroredanalyzation* enthaltenen Klassen

4.1.9 Paket *tud.chess.util*

Klasse	Responsibility
ColorUtils	Bietet Funktionalität, um Helligkeitswerte aus Farbwerten zu berechnen.
FileUtils	Bietet Funktionalität, um Dateisystemoperationen auszuführen

Tabelle 39 Beschreibung der im Paket *tud.chess.util* enthaltenen Klassen

4.1.10 Paket *tud.chess.visualization*

Klasse	Responsibility
BoardRenderer	Rendert übergebene Kollektionen von Schachfiguren auf – feldern als Schachbretter in PGN-Dateien

Tabelle 40 Beschreibung der im Paket *tud.chess.util* enthaltenen Klassen

4.2 Datenformate & -pfade

Neben den von Krimp erwarteten und ausgegebenen Dateiformaten wurden bei der Implementierung der Toolchain eigene Dateiformate entworfen, die zur Darstellung der aufbereiteten Daten notwendig sind. Die nachfolgenden Abschnitte definieren die syntaktischen und semantischen Eigenschaften der verwendeten Dateiformate. Des Weiteren wird angegeben, welche Tools und Klassen Dateien des entsprechenden Formats ausgeben und welche dieses als Eingabe akzeptieren.

4.2.1 Dateiformat „Itemset-Datenbank“

Syntax & Semantik

Eine Itemset-Datenbank enthält positive, durch Leerzeichen getrennte Ganzzahlen, die die Item-Ids verkörpern. Jede Transaktion der Datenbank ist in einer separaten Zeile gespeichert. Die Item-Ids einer Transaktion sind monoton steigend.

```
130 220 250 332 342 422 452 10137 10207 10257 10327 10355 10425 10436
```

Listing 14 Eine Transaktion aus einer Itemset-Datenbank

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
PGNtoGameStateParser	generierend	Wandelt Schachpartien im PGN-Format in Itemset-Datenbanken um.
MirroredChessPiecePositionAnalyzer	annehmend	Spiegelt die Schachstellungen der übergebenen Codetabelle und zählt, wie oft die gespiegelte Stellung in der übergebenen Datenbank vorkommt
ItemsetFrequencyAnalyzer	annehmend	Zählt die gleichen Transaktionen einer Itemset-Datenbank
Krimp: convertdb	annehmend	Wandelt Itemset-Datenbanken in Krimp-Datenbanken um.

Tabelle 41 Tools und Klassen, die Itemset-Datenbanken generieren oder als Eingabe annehmen

4.2.2 Dateiformat „Krimp-Itemset-Datenbank“

Syntax & Semantik

Eine Krimp-Itemset-Datenbank wird von Krimp aus einer Itemset-Datenbank erstellt und enthält positive, durch Leerzeichen getrennte Ganzzahlen, die die Item-Ids verkörpern. Jede Transaktion der Datenbank ist in einer separaten Zeile gespeichert, wobei vor jedem Itemset einer Transaktion die Anzahl der in der Transaktion enthaltenen Items vermerkt ist. Die Item-Ids sind sowohl je Transaktion monoton steigend als auch in Bezug auf die originale Itemset-Datenbank so sortiert, dass das Item mit der kleinsten Id das häufigste Item der Originaldatenbank darstellt.

```
9: 0 2 3 5 6 16 31 41 128
```

Listing 15 Eine Transaktion aus einer Krimp-Itemset-Datenbank

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
Krimp: convertdb	generierend	Wandelt Itemset-Datenbanken in Krimp-Datenbanken um.
Krimp: compress	annehmend	Führt die Kompression auf der Datenbank aus und gibt die Codetabelle zurück

Tabelle 42 Tools und Klassen, die Krimp-Itemset-Datenbanken generieren oder als Eingabe annehmen

4.2.3 Dateiformat „Krimp Codetabelle"

Syntax & Semantik

Die Codetabelle, die Krimp während der Kompression der Eingabedatenbank erstellt, enthält alle Frequent Itemsets der Krimp-Itemset-Datenbank, die zur Kodierung der komprimierten Datenbank verwendet werden. Neben den zeilenweise, durch Leerzeichen getrennten Item-Ids der Frequent Itemsets sind in Klammern noch zwei Maßzahlen notiert: Die erste Zahl gibt an, wie oft das Itemset in der Hülle vorkommt, die zweite gibt die Anzahl der Transaktionen, die dieses Itemset beinhalten, an.

```
0 1 2 3 4 5 6 9 10 11 15 22 23 (950,1027)
```

Listing 16 Eine Zeile der Codetabelle

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
Krimp: compress	generierend	Führt die Kompression auf der Datenbank aus und gibt die Codetabelle zurück.
CodeTableRenderer	annehmend	Rendert die in der Codetabelle enthaltenen Frequent Itemsets als Schachbretter im PGN-Format. Ausgabeort: *boards/* – Verzeichnis
CodeTableStatistics	annehmend	Gruppiert die in der Codetabelle enthaltenen Frequent Itemsets nach Anzahl der enthaltenen Items und gibt eine Statistik der Frequenzen aus.
MirroredChessPiecePositionAnalyzer	annehmend	Spiegelt die Schachstellungen der übergebenen Codetabelle und zählt, wie oft die gespiegelte Stellung in der übergebenen Datenbank vorkommt

Tabelle 43 Tools und Klassen, die Krimp-Itemset-Datenbanken generieren oder als Eingabe annehmen

4.2.4 Dateiformat „Krimp Datenbankanalyse"

Syntax & Semantik

Bei der Umwandlung von Itemset-Datenbanken nach Krimp-Itemset-Datenbanken findet eine Permutation der Item-Ids statt. Dabei erhält das Item, das den größten Supportwert besitzt, die Id „0" – die restlichen Items werden nach dem gleichen Prinzip vergeben. Um die Item-Ids der im Komprimierungsschritt resultierenden Codetabelle wieder rückübersetzen zu können, erstellt Krimp eine Datenbankanalyse-Datei, die die Übersetzung der Item-Ids beinhaltet. In jeder Zeile enthält diese neben der Übersetzung noch weitere statistische Informationen über das Item.

```
0=>10207      65812 (97.9%)      3.737
```

Listing 17 Eine Zeile der Datenbankanalyse

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
Krimp: analysedb	generierend	Generiert die Datenbankanalyse für gegebene Itemset-Datenbank und Krimp-Itemset-Datenbank.
CodeTableRenderer	annehmend	Rendert die in der Codetabelle enthaltenen Frequent Itemsets als Schachbretter im PGN-Format. Benötigt dafür die Übersetzungen der Item-Ids. Ausgabeort: *boards/* – Verzeichnis
CodeTableStatistics	annehmend	Gruppiert die in der Codetabelle enthaltenen Frequent Itemsets nach Anzahl der enthaltenen Items und gibt eine Statistik der Frequenzen aus. Benötigt dafür die Übersetzungen der Item-Ids.
MirroredChessPiecePositionAnalyzer	annehmend	Spiegelt die Schachstellungen der übergebenen Codetabelle und zählt, wie oft die gespiegelte Stellung in der übergebenen Datenbank vorkommt. Benötigt dafür die Übersetzungen der Item-Ids.

Tabelle 44 Tools und Klassen, die Krimp-Datenbankanalysen generieren oder als Eingabe annehmen

4.2.5 Dateiformat „Itemset-Frequenzanalyse“

Syntax & Semantik

Um beispielsweise die häufigsten Bauernstrukturen aus einer (Itemset-)Datenbank zu extrahieren, müssen diese zunächst gezählt werden. Das Dateiformat „Itemset-Frequenzanalyse“ beinhaltet hierfür pro Zeile eine Transaktion einer Itemset-Datenbank. Vor jeder Transaktion ist vermerkt, wie häufig das Itemset der Transaktion in der Datenbank vorkommt.

```
3:60 501 523 533 552 562 571 10067 10526 10535 10556 10566 10576
```

Listing 18 Eine Transaktion aus einer Itemset-Frequenzanalyse

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
ItemsetFrequencyAnalyzer	generierend	Generiert die Frequenzanalyse für die übergebene Itemset-Datenbank.
ItemsetFrequencySorter	annehmend	Sortiert die übergebene Itemset-Frequenzanalyse

Tabelle 45 Tools und Klassen, die Itemset-Frequenzanalysen generieren oder als Eingabe annehmen

4.2.6 Dateiformat „Sortierte Itemset-Frequenzanalyse“

Syntax & Semantik

Die sortierte Itemset-Frequenzanalyse entspricht der Itemset-Frequenzanalyse mit dem Unterschied, dass die Itemsets innerhalb der Datei absteigend nach Frequenz sortiert sind.

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
ItemsetFrequencySorter	generierend	Sortiert die übergebene Itemset-Frequenzanalyse
ItemsetFrequencyRenderer	annehmend	Rendert die Schachbretter der (sortierten) Itemset-Frequenzanalyse als PNG-Dateien

Tabelle 46 Tools und Klassen, die sortierte Itemset-Frequenzanalysen generieren oder als Eingabe annehmen

4.2.7 Dateiformat „Gespiegelte Itemset-Analyse"

Syntax & Semantik

Das Dateiformat speichert, wie häufig eine Schachstellung mit weißen Figuren gespiegelt beim schwarzen Spieler oder umgekehrt vorkommt. Es ist kompatibel zum CSV-Format, die Spalten sind wie folgt definiert:

1. Anzahl der Items (Figuren) im Itemset (in der Stellung)
2. Items des Itemsets
3. Auftrittshäufigkeit des Itemsets für weiß
4. Auftrittshäufigkeit des Itemsets für schwarz

```
3;7571 20 6561;8881;5935
```

Listing 19 Eine Transaktion aus einer gespiegelten Itemset-Analyse

Tools & Klassen

Tool / Klasse	Funktion	Beschreibung
MirroredChessPiecePositionAnalyzer	generierend	Vergleicht das Auftreten von Teilstellungen bei schwarzen und weißen Spielern, erstellt die Analyse.
HeatmapGenerator	annehmend	Erzeugt Heatmaps aus gespiegelten Itemset-Analysen.

Tabelle 47 Tools und Klassen, die gespiegelte Itemset-Analysen generieren oder als Eingabe annehmen

4.2.8 Illustration des Datenpfads

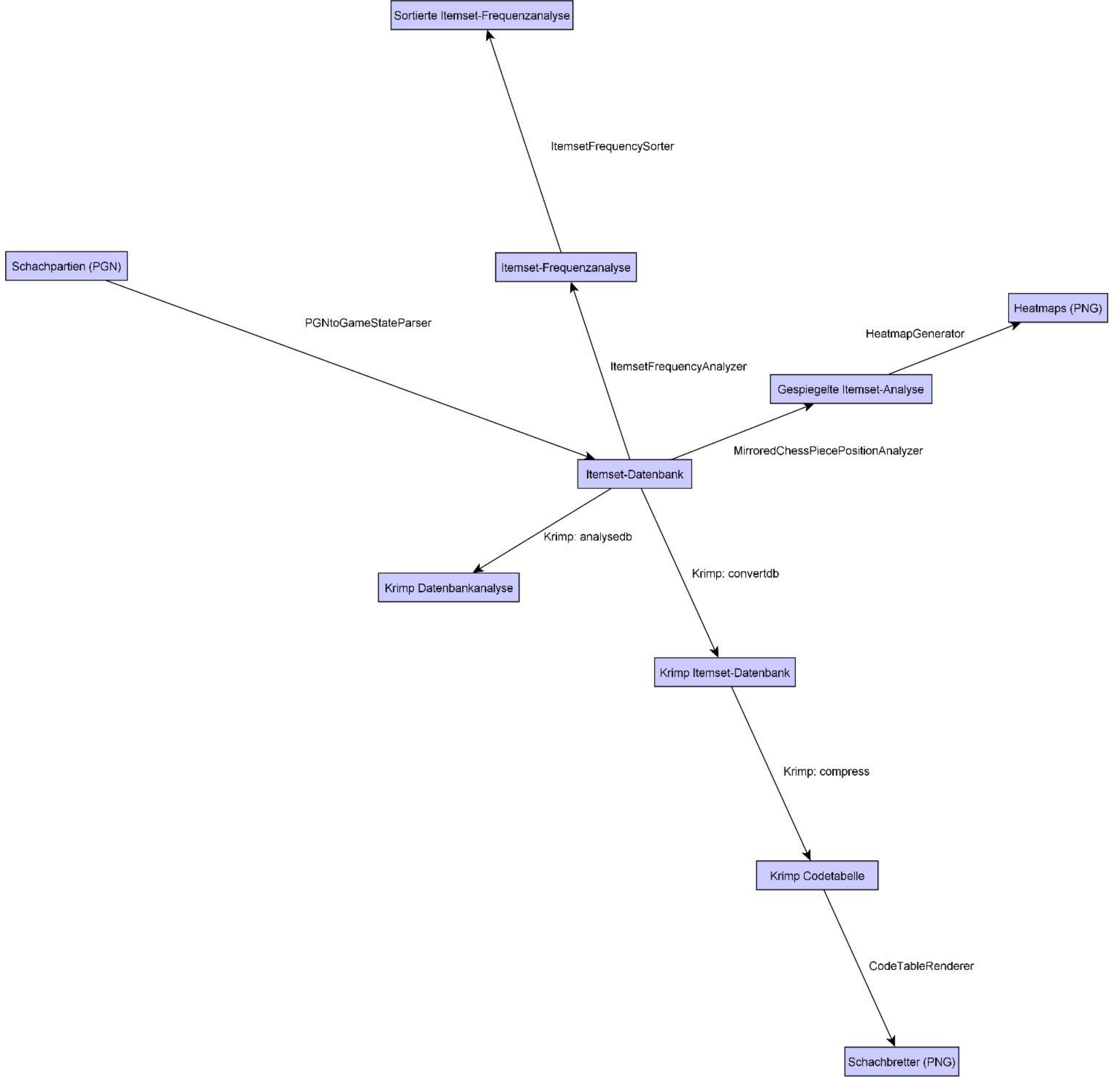


Abbildung 110 Datenpfad der Toolchain

4.3 Externe Programme & Bibliotheken

Neben der implementierten Toolchain wurde auf externe Programme und Bibliotheken zurückgegriffen, die in Tabelle 48 aufgelistet sind.

Programm / Bibliothek	Quelle
Krimp	http://www.cs.uu.nl/groups/ADA/krimp/
Apache Commons CLI	http://commons.apache.org/proper/commons-cli/
Apache Commons Lang	http://commons.apache.org/proper/commons-lang/
JHeatChart	http://www.tc33.org/projects/jheatchart/

Tabelle 48 Verwendete externe Programme und Bibliotheken

4.4 Einrichtung & Verwendung

Die JAR-Archive der Toolchain befinden sich im Unterverzeichnis *toolchain/* des ZIP-Archivs, das unter der URL http://www.viathinksoft.de/downloads/tud_bachelorthesis_toolchain_vn.zip heruntergeladen werden kann. Die Dateinamen entsprechen den in Tabelle 31 bis Tabelle 40 durch Unterstreichung markierten Hauptklassen.

Zur Ausführung der JARs benötigt man eine aktuelle Version des *Java Runtime Environment* (JRE) oder des *Java Development Kit* (JDK). Beide werden in der von Oracle® entwickelten Version unter der Adresse http://www.oracle.com/technetwork/java/javase/downloads/index.html zum Download zur Verfügung gestellt.

Die benötigten Aufrufparameter der entwickelten Werkzeuge werden in den nachfolgenden Abschnitten erläutert.

4.4.1 Tool *PGNtoGameStateParser*

Syntax

```
PGNtoGameStateParser [options] OUTFILE PGNFILE_0 [PGNFILE_1] …
```

Listing 20 Aufrufsyntax des Tools *PGNtoGameStateParser*

Benötigte Parameter

Parameter	Beschreibung
OUTFILE	Ausgabedateiname (ohne „.dat“-Endung)
PGNFILE_#	Dateiname der Schachpartiedateien im PGN-Format

Tabelle 49 Benötigte Parameter von *PGNtoGameStateParser*

Optionale Parameter

Parameter	Beschreibung
-b, --black	Nur schwarze Figuren werden extrahiert.
-e, --end <num>	Nur die <num> letzten Schachstellungen werden extrahiert.
-i, --index	Die Schachfiguren werden durchnummeriert, so dass sie voneinander unterschieden werden können.
-l, --limit <num>	Maximal <num> Schachpartien werden verarbeitet.
-m, --middle <num0 num1>	Die ersten <num0> Schachstellungen werden verworfen und nur die nächsten <num1> Stellungen extrahiert.
-n, --normalize	Nur Partien, die nicht unentschieden ausgegangen sind, werden extrahiert. Alle Partien, in denen schwarz gewinnt, werden gespiegelt und die Farben getauscht, so dass weiß immer gewinnt.
-o, --opening <num>	Nur die <num> ersten Schachstellungen werden extrahiert.
-p, --pawnstructure	Nur die charakteristischen Bauernstrukturen werden extrahiert.
-s, --split	Die Partien werden nach Spielergebnis getrennt. Die resultierenden Dateinamen der Itemset-Datenbanken werden durch die Suffixe *_whiteWins*, *_blackWins* und *_draw* ergänzt.
-t, --psmatch <psfile limit>	Die Partien werden nach enthaltener Bauernstruktur aus der Datei <psfile> gruppiert. <limit> gibt die Anzahl der in <psfile> enthaltenen Bauernstrukturen an, gegen die verglichen werden soll.
-w, --white	Nur weiße Figuren werden extrahiert.
-x, --noinitial	Figuren, die sich auf ihren Initialpositionen befinden, werden ignoriert.

Tabelle 50 Optionale Parameter von *PGNtoGameStateParser*

4.4.2 Tool *CodeTableRenderer*

Syntax

```
CodeTableRenderer CODETABLE DBANALYSIS
```

Listing 21 Aufrufsyntax des Tools *CodeTableRenderer*

Benötigte Parameter

Parameter	Beschreibung
CODETABLE	Dateiname der Krimp-Codetabelle
DBANALYSIS	Dateiname der Krimp Datenbankanalyse

Tabelle 51 Benötigte Parameter von *CodeTableRenderer*

4.4.3 Tool *ItemsetFrequencyAnalyzer*

Syntax

```
ItemsetFrequencyAnalyzer [options] FREQFILE DATABASE
```

Listing 22 Aufrufsyntax des Tools *ItemsetFrequencyAnalyzer*

Benötigte Parameter

Parameter	Beschreibung
FREQFILE	Dateiname der zu erstellenden Itemset-Frequenzanalyse
DATABASE	Dateiname der Itemset-Datenbank

Tabelle 52 Benötigte Parameter von *ItemsetFrequencyAnalyzer*

Optionale Parameter

Parameter	Beschreibung
-b, --blocksize	Anzahl der Transaktionen, die gleichzeitig verarbeitet werden

Tabelle 53 Optionale Parameter von *ItemsetFrequencyAnalyzer*

4.4.4 Tool *ItemsetFrequencyRenderer*

Syntax

```
ItemsetFrequencyRenderer [options] FREQFILE OUTDIR
```

Listing 23 Aufrufsyntax des Tools *ItemsetFrequencyRenderer*

Benötigte Parameter

Parameter	Beschreibung
FREQFILE	Dateiname der Itemset-Frequenzanalyse
OUTDIR	Verzeichnis, in das die Schachbretter der Itemset-Frequenzanalyse gerendert werden sollen

Tabelle 54 Benötigte Parameter von *ItemsetFrequencyRenderer*

Optionale Parameter

Parameter	Beschreibung
-m, --max	Maximale Anzahl von zu renderenden Schachbrettern

Tabelle 55 Optionale Parameter von *ItemsetFrequencyRenderer*

4.4.5 Tool *ItemsetFrequencySorter*

Syntax

```
ItemsetFrequencySorter FREQFILE
```

Listing 24 Aufrufsyntax des Tools *ItemsetFrequencySorter*

Benötigte Parameter

Parameter	Beschreibung
FREQFILE	Dateiname der zu sortierenden Itemset-Frequenzanalyse

Tabelle 56 Benötigte Parameter von *ItemsetFrequencySorter*

4.4.6 Tool *MirroredChessPiecePositionAnalyzer*

Syntax

```
MirroredChessPiecePositionAnalyzer CODETABLE DBANALYSIS DATABASE
```

Listing 25 Aufrufsyntax des Tools *MirroredChessPiecePositionAnalyzer*

Benötigte Parameter

Parameter	Beschreibung
CODETABLE	Dateiname der Krimp-Codetabelle
DBANALYSIS	Dateiname der Krimp Datenbankanalyse
DATABASE	Dateiname der Itemset-Datenbank

Tabelle 57 Benötigte Parameter von *MirroredChessPiecePositionAnalyzer*

4.4.7 Tool *HeatmapGenerator*

Syntax

```
HeatmapGenerator MIRROREDANALYSIS
```

Listing 26 Aufrufsyntax des Tools *HeatmapGenerator*

Benötigte Parameter

Parameter	Beschreibung
MIRROREDANALYSIS	Dateiname der gespiegelten Itemset-Analyse

Tabelle 58 Benötigte Parameter von *HeatmapGenerator*

5 Resümee

5.1 Frequent Itemset Mining

Die durch Krimp erreichte Reduktion der Frequent Itemsets lag bei den durchgeführten Experimenten im Durchschnitt bei 90,48 % – ein beachtlicher Wert. Allerdings hängt er sehr stark von der Vorauswahl der Daten ab, was in Abbildung 111 bis Abbildung 113 deutlich wird: Während in den Experimenten 1.1 bis 1.3 vollständige Schachpartien an Krimp übergeben wurden (hier konnte die Anzahl der Frequent Itemsets nur um durchschnittlich 68,72 % gesenkt werden), verbesserte sich die Reduktion signifikant, als nur einzelne Spielphasen betrachtet (durchschnittliche Reduktion um 99,82 %), oder die Schachstellungen nach enthaltener Bauernstruktur gefiltert wurden (99,77 %).

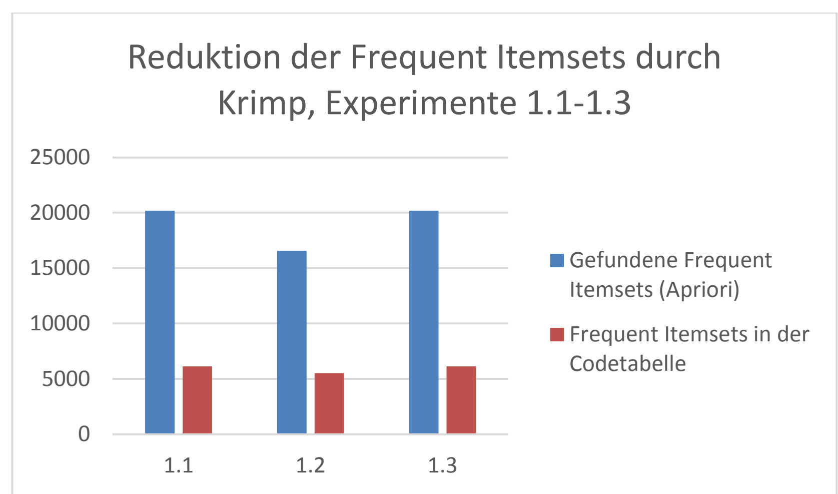


Abbildung 111 Reduktion der Frequent Itemsets durch Krimp, Experimente 1.1 – 1.3

Dies lässt sich dadurch erklären, dass mittels einer guten Vorfilterung die Anzahl der unterschiedlichen Stellungen deutlich reduziert werden kann. In der Folge konnte der Frequent Itemset Miner mit einem deutlich reduzierten Minimum Support arbeiten (0,10 % in den Experimenten 2.2 bis 2.4 anstatt 15,00 % in den Experimenten 1.1 bis 1.3) und dadurch die Kompression verbessert werden (wie bereits in Abschnitt 2.3.3 erwähnt erlaubt erst die Betrachtung aller Itemsets – also ein Minimum Support von 0 % – eine optimale Kompression).

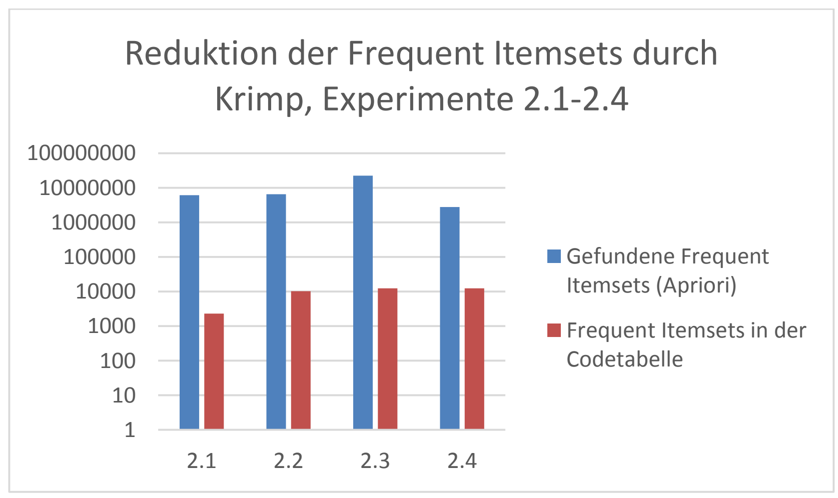


Abbildung 112 Reduktion der Frequent Itemsets durch Krimp, Experimente 2.1 – 2.4

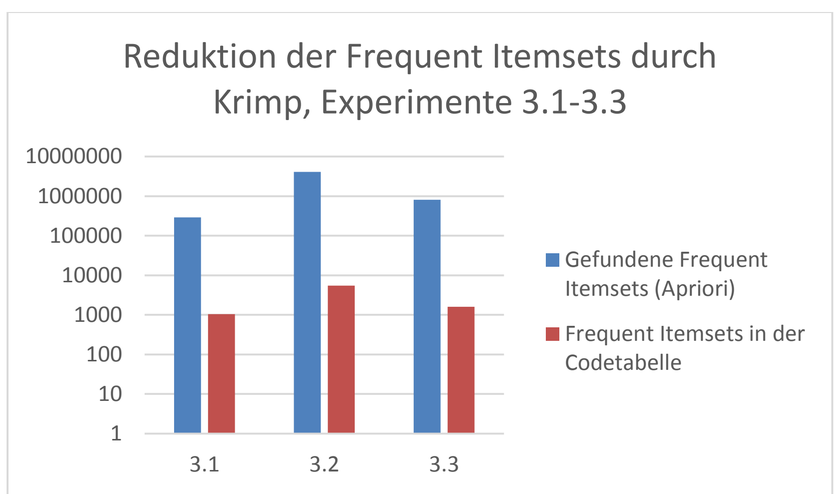


Abbildung 113 Reduktion der Frequent Itemsets durch Krimp, Experimente 3.1 – 3.3

Besonders in Experiment 2.1 wurde deutlich, wie sehr die Performanz von Krimp von der des Frequent Itemset Miners abhängt. Während der Eröffnung einer Schachpartie enthalten alle Stellungen Teile der Grundstellung, die je nach gewählter Eröffnung anders ausfallen. Der von Krimp verwendete Algorithmus zur Extraktion der Frequent Itemsets (APRIORI, siehe Abschnitt 2.2.2) muss die Datenbank in jeder Iteration einmal durchlaufen, was bei großen Datenbanken sehr zeit- und speicherintensiv sein kann. Zwar existieren bereits zahlreiche verbesserte Algorithmen wie der TD-FP-GROWTH–Algorithmus [17], doch wurden außer dem APRIORI–Algorithmus bisher keine weiteren Frequent Itemset Miner in Krimp implementiert.

5.2 Kompression

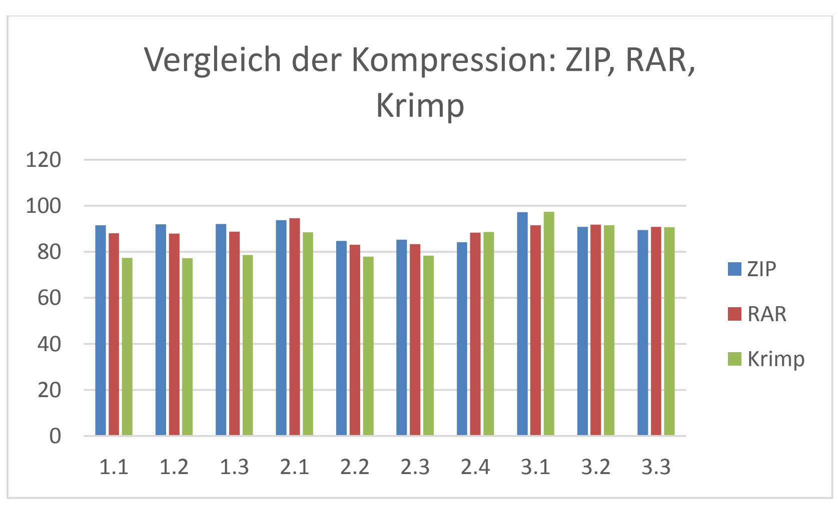


Abbildung 114 Vergleich der Kompression: ZIP, RAR, Krimp

Vergleicht man die Kompression von Krimp mit üblichen Kompressionsverfahren wie ZIP und RAR, fällt auf, dass Krimp meistens etwas schlechter abschneidet. Erst bei minimalem Supportwert und starker Vorfilterung der Daten (Experimente 2.4 und 3.1 bis 3.3) erreicht oder übertrifft Krimp die anderen Verfahren. Dies deckt sich mit den Beobachtungen des vorherigen Abschnitts: Die Performanz des Frequent Itemset Miners bestimmt maßgeblich das Resultat der Kompression.

5.3 Ausblick

Es wurde deutlich, dass Krimp das Potential besitzt, die Datenmenge bei idealen Bedingungen deutlich zu komprimieren. Trotzdem ist es erst nach ausreichender Vorverarbeitung möglich, größere Datenmengen zu verarbeiten und somit die Informationen aus der Datenbank zu verdichten. Eine mögliche Verbesserung des Algorithmus liegt klar in der Implementation eines leistungsfähigeren Frequent Itemset Miner: Der aktuell implementierte APRIORI–Miner beschränkt derzeit die Performanz des Verfahrens erheblich.

Hinsichtlich der Vorverarbeitung könnte möglicherweise eine gleichzeitige Aufteilung nach Spielergebnis und Spielphase die Qualität der resultierenden Teilstellungen erhöhen. Um die daraus gewonnenen Daten für ein Schachprogramm nutzbar zu machen, könnte es im Anschluss daran sinnvoll sein zu vergleichen, welche Teilstellungen häufiger bei gewonnenen als bei verlorenen Partien auftreten. Auf diese Weise könnten Knoten im Spielbaum identifiziert werden, die bei der Zugauswahl eine Präferenz gegenüber anderen Knoten erhalten (siehe 2.1.1).

Weiterhin wäre eine Übertragung der Methodik auf andere weniger komplexe Spiele möglich, um die Heuristiken, die aus den resultierenden Daten generiert werden könnten, zunächst an Modellproblemen zu testen (beispielsweise an Spielen, für die bereits eine vollständige Lösung existiert) und anschließend wieder schrittweise auf komplexere Spiele zu übertragen.

Abbildungsverzeichnis

Abbildung 1 Initialpositon eines Schachspiels [1] 3

Abbildung 2 Der König darf stets einen Schritt in jede Richtung gehen, sofern dieses Feld nicht von einer gegnerischen Figur direkt besetzbar ist, der König also im Schach stehen würde. Steht er im Schach, gilt es ihn aus dem Schach zu bewegen. Ist dies unmöglich, bezeichnet man dies als Schachmatt – der Gegner hat gewonnen. [1] 3

Abbildung 3 Die Dame darf sowohl in horizontaler, vertikaler als auch in beide diagonale Richtungen beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf einer der Linien befindet. [1] 4

Abbildung 4 Der Läufer darf in beide diagonalen Richtungen beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf einer der Linien befindet. [1] 4

Abbildung 5 Der Springer springt pro Zug entweder erst ein Feld in vertikaler Richtung und anschließend zwei Felder in horizontaler Richtung oder erst ein Feld in horizontaler Richtung und dann zwei Felder in vertikaler Richtung. Dabei ist es ihm als einzige Figur erlaubt, andere Schachfiguren zu überspringen. [1] ..4

Abbildung 6 Der Turm darf sowohl horizontal als auch vertikal beliebig weit vorrücken oder eine gegnerische Figur schlagen, die sich auf eine der Linien befindet. [1] 5

Abbildung 7 Der Bauer darf ein Feld vorrücken falls dies leer ist. Ausgehend von seiner Initialposition darf er zwei Felder vorrücken, sofern beide Felder unbesetzt sind. [1] 5

Abbildung 8 Der weiße Bauer darf den schwarzen Turm oder Springer schlagen. [1] 6

Abbildung 9 Der weiße Bauer schlägt den schwarzen Bauern „en passant“, in dem er auf das gekennzeichnete Feld vorrückt. [1] 6

Abbildung 10 Charakteristische Bauernstruktur einer Schachstellung 7

Abbildung 11 Auszug aus einem Suchbaum von Schachstellungen 8

Abbildung 12 Teilstellung einer Schachstellung 12

Abbildung 13 Entscheidungsbaum zur Entropiekopierung des Alphabets $\{a, b, c, d, e, f\}$ 14

Abbildung 14 Transformationsschritte des Krimp-Algorithmus 17

Abbildung 15 Kodierungsbeziehungen 18

Abbildung 16 Vergleich der Kompressionsraten der UCI „chess“-Datenbank in Abhängigkeit vom Minimum Support [15] 23

Abbildung 17 BS 1, N = 10663 29

Abbildung 18 BS 2, N = 7355 29

Abbildung 19 BS 3, N = 5799 29

Abbildung 20 Support: 458.452 / 15,65 % 32

Abbildung 21 Support: 455.076 / 15,53 % 32

Abbildung 22 Support: 444.976 / 15,19 % 32

Abbildung 23 Support: 442.503 / 15,10 % 32

Abbildung 24 Support: 441.977 / 15,08 % 33

Abbildung 25 Support: 439.642 / 15,00 % 33
Abbildung 26 Teilstellung mit größtem Support, die nicht Teil der Initialstellung ist 34
Abbildung 27 Support: 380.366 / 15,94 % 36
Abbildung 28 Support: 380.148 / 15,93 % 36
Abbildung 29 Support: 376.930 / 15,79 % 36
Abbildung 30 Support: 366.663 / 15,36 % 36
Abbildung 31 Support: 366.091 / 15,34 % 37
Abbildung 32 Support: 365.108 / 15,30 % 37
Abbildung 33 Support: 364.379 / 15,27 % 37
Abbildung 34 Support: 358.833 / 15,03 % 37
Abbildung 35 Support: 458.452 / 15,65 % 40
Abbildung 36 Support: 455.076 / 15,53 % 40
Abbildung 37 Support: 444.976 / 15,19 % 40
Abbildung 38 Support: 442.503 / 15,10 % 40
Abbildung 39 Support: 441.977 / 15,08 % 41
Abbildung 40 Support: 439.642 / 15,00 % 41
Abbildung 41 Support: 64.764 / 32,38 % 43
Abbildung 42 Support: 63.051 / 31,53 % 43
Abbildung 43 Support: 60.491 / 30,25 % 43
Abbildung 44 Support: 60.189 / 30,09 % 43
Abbildung 45 Support: 160 / 0,18 % 46
Abbildung 46 Support: 154 / 0,17 % 46
Abbildung 47 Support: 148 / 0,17 % 46
Abbildung 48 Support: 118 / 0,13 % 46
Abbildung 49 Support: 113 / 0,13 % 47
Abbildung 50 Support: 96 / 0,11 % 47
Abbildung 51 Support: 177 / 0,18 % 49
Abbildung 52 Support: 177 / 0,18 % 49
Abbildung 53 Support: 173 / 0,17 % 49
Abbildung 54 Support: 159 / 0,16 % 50
Abbildung 55 Support: 149 / 0,15 % 50
Abbildung 56 Support: 147 / 0,15 % 50
Abbildung 57 Support: 132 / 0,13 % 50
Abbildung 58 Support: 123 / 0,12 % 51
Abbildung 59 Support: 115 / 0,12 % 51

Abbildung 60 Support: 110 / 0,11 % 51
Abbildung 61 Support: 102 / 0,10 % 51
Abbildung 62 Support: 2.326 / 0,12 % 54
Abbildung 63 Support: 2.009 / 0,11 % 54
Abbildung 64 Support: 1.993 / 0,10 % 54
Abbildung 65 Bauernstruktur 1 56
Abbildung 66 Support: 7.870 / 10,17 % 58
Abbildung 67 Support: 7.004 / 9,05 % 58
Abbildung 68 Support: 4.605 / 5,95 % 58
Abbildung 69 Support: 4.577 / 5,91 % 58
Abbildung 70 Support: 3.941 / 5,09 % 59
Abbildung 71 Support: 3.848 / 4,97 % 59
Abbildung 72 Support: 3.641 / 4,70 % 59
Abbildung 73 Support: 3.535 / 4,57 % 59
Abbildung 74 Support: 3.446 / 4,45 % 60
Abbildung 75 Support: 3.249 / 4,20 % 60
Abbildung 76 N: 19, Support: 77.312 / 26,40 % 60
Abbildung 77 N: 23, Support: 65.898 / 85,14 % 61
Abbildung 78 N: 24, Support: 60.209 / 77,79 % 61
Abbildung 79 N: 28, Support: 28.744 / 37,14 % 62
Abbildung 80 Bauernstruktur 2 63
Abbildung 81 Support: 2751 / 2,68 % 65
Abbildung 82 Support: 2597 / 2,53 % 65
Abbildung 83 Support: 2077 / 2,02 % 65
Abbildung 84 Support: 1161 / 1,13 % 65
Abbildung 85 Support: 973 / 0,95 % 66
Abbildung 86 Support: 887 / 0,86 % 66
Abbildung 87 Support: 876 / 0,85 % 66
Abbildung 88 Support: 827 / 0,81 % 66
Abbildung 89 Support: 808 / 0,79 % 67
Abbildung 90 Support: 755 / 0,74 % 67
Abbildung 91 N: 15, Support: 95129 / 92,61 % 67
Abbildung 92 N: 18, Support: 53184 / 51,78 %, „Turmquadrat“ 68
Abbildung 93 Bauernstruktur 3 69
Abbildung 94 Support: 185 / 0,90 % 71

Abbildung 95 Support: 184 / 0,89 % ... 71
Abbildung 96 Support: 150 / 0,73 % ... 71
Abbildung 97 Support: 123 / 0,60 % ... 71
Abbildung 98 Support: 109 / 0,53 % ... 72
Abbildung 99 Support: 79 / 0,38 % ... 72
Abbildung 100 Support: 74 / 0,36 % ... 72
Abbildung 101 Support: 69 / 0,34 % ... 72
Abbildung 102 Support: 69 / 0,34 % ... 73
Abbildung 103 Support: 60 / 0,29 % ... 73
Abbildung 104 N: 17, Support: 16574 / 80,61 % ... 73
Abbildung 105 N: 20, Support: 8681 / 42,22 %, „Springerquadrat“ ... 74
Abbildung 106 N: 21,Support: 6968 / 33,89 %, „Turmquadrat“ #1 ... 74
Abbildung 107 N: 21, Support: 6299 / 30,64 %, „Turmquadrat“ #2 ... 74
Abbildung 108 N: 24, Support: 4416 / 21,48 %, „Springerquadrat“ und „Turmquadrat“ ... 75
Abbildung 109 Paketstruktur der Java-Implementation ... 76
Abbildung 110 Datenpfad der Toolchain ... 84
Abbildung 111 Reduktion der Frequent Itemsets durch Krimp, Experimente 1.1 – 1.3 ... 90
Abbildung 112 Reduktion der Frequent Itemsets durch Krimp, Experimente 2.1 – 2.4 ... 90
Abbildung 113 Reduktion der Frequent Itemsets durch Krimp, Experimente 3.1 – 3.3 ... 91
Abbildung 114 Vergleich der Kompression: ZIP, RAR, Krimp ... 91

Tabellenverzeichnis

Tabelle 1 Auflistung aller Schachfiguren und möglicher Züge 5
Tabelle 2 Enthaltene Dateien in der ICOfY Base 2011.1 – Datenbank 10
Tabelle 3 Itemsets und Supportwerte der Datenbank *sampledb* 21
Tabelle 4 Zuordnung von Figurenfarbe zu Zahlenwert 26
Tabelle 5 Zuordnung von Figurentyp zu Zahlenwert 26
Tabelle 6 Zuordnung von horizontaler Position zu Zahlenwert 27
Tabelle 7 Zuordnung von vertikaler Position zu Zahlenwert 27
Tabelle 8 Definition der Spielphasen über Intervalle von Halbzügen 28
Tabelle 9 Vorverarbeitete Datenbanken & Kompressionskonfigurationen 30
Tabelle 10 Gefundene Stellungen über Anzahl der Figuren in der Stellung 31
Tabelle 11 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 33
Tabelle 12 Gefundene Stellungen über Anzahl der Figuren in der Stellung 35
Tabelle 13 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 38
Tabelle 14 Gefundene Stellungen über Anzahl der Figuren in der Stellung 39
Tabelle 15 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 41
Tabelle 16 Gefundene Stellungen über Anzahl der Figuren in der Stellung 42
Tabelle 17 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 44
Tabelle 18 Gefundene Stellungen über Anzahl der Figuren in der Stellung 45
Tabelle 19 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 47
Tabelle 20 Gefundene Stellungen über Anzahl der Figuren in der Stellung 48
Tabelle 21 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 52
Tabelle 22 Gefundene Stellungen über Anzahl der Figuren in der Stellung 53
Tabelle 23 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 55
Tabelle 24 Gefundene Stellungen über Anzahl der Figuren in der Stellung 57
Tabelle 25 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 62
Tabelle 26 Gefundene Stellungen über Anzahl der Figuren in der Stellung 64
Tabelle 27 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 68
Tabelle 28 Gefundene Stellungen über Anzahl der Figuren in der Stellung 70
Tabelle 29 Vergleich von Krimp mit gängigen Kompressionsalgorithmen 75
Tabelle 30 Beschreibung der Paketstruktur 76
Tabelle 31 Beschreibung der im Paket *tud.chess* enthaltenen Klassen 77

Tabelle 32 Beschreibung der im Paket *tud.chess.abstraction* enthaltenen Klassen 77

Tabelle 33 Beschreibung der im Paket *tud.chess.abstractiongenerator* enthaltenen Klassen........ 77

Tabelle 34 Beschreibung der im Paket *tud.chess.game* enthaltenen Klassen 78

Tabelle 35 Beschreibung der im Paket *tud.chess.krimp* enthaltenen Klassen 78

Tabelle 36 Beschreibung der im Paket *tud.chess.krimp.abstractionreverter* enthaltenen Klassen 78

Tabelle 37 Beschreibung der im Paket *tud.chess.krimp.heatmap* enthaltenen Klassen 79

Tabelle 38 Beschreibung der im Paket *tud.chess.krimp.mirroredanalyzation* enthaltenen Klassen........ 79

Tabelle 39 Beschreibung der im Paket *tud.chess.util* enthaltenen Klassen 79

Tabelle 40 Beschreibung der im Paket *tud.chess.util* enthaltenen Klassen 79

Tabelle 41 Tools und Klassen, die Itemset-Datenbanken generieren oder als Eingabe annehmen 80

Tabelle 42 Tools und Klassen, die Krimp-Itemset-Datenbanken generieren oder als Eingabe annehmen 81

Tabelle 43 Tools und Klassen, die Krimp-Itemset-Datenbanken generieren oder als Eingabe annehmen 81

Tabelle 44 Tools und Klassen, die Krimp-Datenbankanalysen generieren oder als Eingabe annehmen 82

Tabelle 45 Tools und Klassen, die Itemset-Frequenzanalysen generieren oder als Eingabe annehmen........ 82

Tabelle 46 Tools und Klassen, die sortierte Itemset-Frequenzanalysen generieren oder als Eingabe annehmen 83

Tabelle 47 Tools und Klassen, die gespiegelte Itemset-Analysen generieren oder als Eingabe annehmen 83

Tabelle 48 Verwendete externe Programme und Bibliotheken 85

Tabelle 49 Benötigte Parameter von *PGNtoGameStateParser* 86

Tabelle 50 Optionale Parameter von *PGNtoGameStateParser* 87

Tabelle 51 Benötigte Parameter von *CodeTableRenderer* 87

Tabelle 52 Benötigte Parameter von *ItemsetFrequencyAnalyzer* 88

Tabelle 53 Optionale Parameter von *ItemsetFrequencyAnalyzer* 88

Tabelle 54 Benötigte Parameter von *ItemsetFrequencySorter* 89

Tabelle 55 Benötigte Parameter von *MirroredChessPiecePositionAnalyzer* 89

Tabelle 56 Benötigte Parameter von *HeatmapGenerator* 89

Listingverzeichnis

Listing 1 Beispiel einer Schachpartie im PGN-Format [6] ... 9
Listing 2 Pseudocode des APRIORI–Algorithmus [14] ... 16
Listing 3 Pseudocode der STANDARD–Funktion für Items [15] ... 19
Listing 4 Erweiterung der STANDARD–Funktion für Itemsets ... 19
Listing 5 Pseudocode der COVER-ORDER–Funktion ... 20
Listing 6 Pseudocode des NAIVE-COMPRESSION–Algorithmus [15] ... 20
Listing 7 Pseudocode der PRUNE-ON-THE-FLY–Funktion [15] ... 22
Listing 8 Pseudocode des COMPRESS-AND-PRUNE–Algorithmus [15] ... 23
Listing 9 Pseudocode der PARSESINGLEGAME–Funktion ... 24
Listing 10 Pseudocode der EXTRACTOPENING–Funktion ... 24
Listing 11 Pseudocode der EXTRACTMIDDLEGAME–Funktion ... 25
Listing 12 Pseudocode der EXTRACTENDGAME–Funktion ... 25
Listing 13 Pseudocode der EXTRACTPAWNSTRUCTURE–Funktion ... 25
Listing 14 Eine Transaktion aus einer Itemset-Datenbank ... 80
Listing 15 Eine Transaktion aus einer Krimp-Itemset-Datenbank ... 80
Listing 16 Eine Zeile der Codetabelle ... 81
Listing 17 Eine Zeile der Datenbankanalyse ... 81
Listing 18 Eine Transaktion aus einer Itemset-Frequenzanalyse ... 82
Listing 19 Eine Transaktion aus einer gespiegelten Itemset-Analyse ... 83
Listing 20 Aufrufsyntax des Tools *PGNtoGameStateParser* ... 86
Listing 21 Aufrufsyntax des Tools *CodeTableRenderer* ... 87
Listing 22 Aufrufsyntax des Tools *ItemsetFrequencyAnalyzer* ... 88
Listing 23 Aufrufsyntax des Tools *ItemsetFrequencyRenderer* ... 88
Listing 24 Aufrufsyntax des Tools *ItemsetFrequencySorter* ... 89
Listing 25 Aufrufsyntax des Tools *MirroredChessPiecePositionAnalyzer* ... 89
Listing 26 Aufrufsyntax des Tools *HeatmapGenerator* ... 89

Literatur- & Quellenverzeichnis

[1] http://de.wikipedia.org/wiki/Schach

[2] DE GROOT, A. D. *Thought and choice in chess*. The Hague: Mouton, 1965.

[3] http://de.wikipedia.org/wiki/ECO-Code

[4] http://de.wikipedia.org/wiki/Schachprogramm

[5] METROPOLIS, NICHOLAS; Ulam, S. *The Monte Carlo Method*. In *Journal of the American Statistical Association, Vol. 44, No. 247*, 1949

[6] http://de.wikipedia.org/wiki/Portable_Game_Notation

[7] http://ingo-schwarz.de/schach/icofy-base-schach/

[8] NAISBITT, JOHN. Megatrends, 1982.

[9] http://de.wikipedia.org/wiki/Kleenesche_und_positive_H%C3%BClle

[10] http://de.wikipedia.org/wiki/Formale_Sprache

[11] http://de.wikipedia.org/wiki/Formale_Grammatik

[12] SHANNON, C. E. *A Mathematical Theory of Communication*. In *The Bell System Technical Journal*, 1948

[13] GRÜNWALD, PETER. *A Tutorial Introduction to the Minimum Description Length Principle*, 1998.

[14] http://en.wikipedia.org/wiki/Apriori_algorithm

[15] SIEBES, ARNO; VREEKEN, JILLES; VAN LEEUWEN, MATTHIJS. *Item Sets That Compress*, 2006.

[16] http://archive.ics.uci.edu/ml/index.html

[17] WANG, K.; TANG L.; HAN J. *Top Down FP-Growth for Association Rule Mining*. In *Advances in Knowledge Discovery and Data Mining*, Vol. 2336, 2002