Finden von häufigen Zugsequenzen in Schachdatenbanken

Finding Frequent Move Sequences in Chess Databases

Bachelor-Thesis von Karsten Will

April 2014

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
künstliche Intelligenz

Finden von häufigen Zugsequenzen in Schachdatenbanken
Finding Frequent Move Sequences in Chess Databases

Vorgelegte Bachelor-Thesis von Karsten Will

1. Gutachten:
2. Gutachten:

Tag der Einreichung:

Inhaltsverzeichnis

1 Einleitung **2**

2 Finden häufiger Sequenzen **3**
2.1 Einführung 3
2.2 MDL für Eventsequenzen 3

3 Finden häufiger Zugsequenzen **14**
3.1 Idee des Programms 14
3.2 Implementierung 15

4 Experimente **17**
4.1 Experiment 1: Damengambit 17
4.2 Experiment 2: Klassische Stonewall-Verteidigung 18
4.3 Experiment 3: Drachenvariante 20

5 Fazit **23**

6 Anhang **24**
6.1 Abbildungsverzeichnis 25
6.2 Tabellenverzeichnis 26
6.3 Literaturverzeichnis 26

1 Einleitung

Schach ist ein sehr komplexes Spiel: Alleine für den ersten Halbzug gibt es 8 verschiedene sinnvoll untersuchte Möglichkeiten. Die Literatur für Eröffnungen, Mittelspiele und Endspiele ist in allen denkbaren Sprachen praktisch unüberschaubar. Das alles nur für die Fragen „Welche Figur ziehe ich als nächstes?“ und „Wie kann ich am schnellsten gewinnen?“.

In dieser Arbeit soll für eine gegebene Stellung das Finden von vielversprechenden Fortsetzungen mithilfe von *Data Mining* untersucht werden. Dafür wird die Verwendung des *Minimum-Discription-Length-Prinzip* genauer betrachtet.
Für den Ansatz sollen aus einer Schachdatenbank alle gleichen Stellungen und deren Fortsetzungen gesucht und die entstandenen Sequenzen mithilfe eines *pattern-set-mining*-Ansatzes codiert werden. Dieser Ansatz gibt eine Codetabelle mit den am Häufigsten verwendeten Mustern zurück. Die Muster in dieser Tabelle werden zu Zugfolgen umgewandelt und deren Sinnhaftigkeit untersucht.
Im Laufe dieser Arbeit soll dieser Ansatz vorgestellt, implementiert und die Sinnhaftigkeit der Ergebnisse untersucht werden. Dafür wird im zweiten Kapitel eine Möglichkeit der Codierung von Sequenzdatenbanken vorgestellt. Im dritten Kapitel wird kurz erläutert, wie Zugsequenzen mit diesem Ansatz kombinierbar sind und wie die Implementierung realisiert wurde. Zum Abschluss wird die Sinnhaftigkeit dieser Möglichkeit genauer untersucht: Dazu sollen häufig auftretende Schachstellungen herangezogen und die vom Algorithmus vorgeschlagenen Fortführungen im Hinblick auf deren Güte untersucht werden.

2 Finden häufiger Sequenzen

2.1 Einführung

Angenommen man ist an den wichtigsten Mustern einer Datenbank interessiert, so würde man einen *frequent pattern miner* auf die Daten anwenden und alle Muster benutzen, die so und so oft vorkommen. Jedoch kann in diesem Fall das Problem auftreten, dass man große Mengen von redundanten Mustern erhält, was eine Analyse der erhaltenen Daten deutlich erschweren würde.
In [1] wird dafür ein anderer Ansatz vorgestellt: Anstelle einer individuellen Betrachtung von Mustern wird hier eine Menge von Mustern gesucht, die die vorgelegten Daten am besten beschreibt. Die wichtigsten Eigenschaften dieser Menge von Mustern sollten sein, dass diese klein ist, die Daten gut generalisiert und nicht redundant ist. Aus diesem Grund wird in diesem Ansatz das *Minimum-Discription-Length-Prinzip* (MDL) angewendet, welches die Menge von Mustern identifiziert, mit welcher die Daten am kürzesten codiert werden können.

Man kann dieses Prinzip wie folgt grob beschreiben: Angenommen es ist eine Menge $\mathscr{M}$ von möglichen Modellen gegeben. Als bestes Modell $M \in \mathscr{M}$ sei jenes bezeichnet, für welches $L(M) + L(D|M)$ minimal ist. Wobei hier durch $L(M)$ die Größe in Bits der Beschreibung von M und durch $L(D|M)$ die Größe der durch M codierten Daten in Bits bezeichnet sei. Um MDL zu verwenden, muss man definieren, was die Modelle $\mathscr{M}$ sind, wie ein $M \in \mathscr{M}$ die Daten beschreibt und wie diese in Bits codiert werden können.

In diesem Abschnitt wird im Wesentlichen der in [1] gezeigte Ansatz kurz zusammengefasst und die wichtigsten Resultate aufgezeigt. Die Ideen, Sätze und Codeteile werden in [1] eingeführt.

2.2 MDL für Eventsequenzen

2.2.1 Notation für Eventsequenzen

Für die Anwendung dieses Prinzips sind im vorhinein einige Notationen zu definieren.
Als Datentypen werden in diesem Fall Eventsequenzen verwendet. Eine Sequenzdatenbank D über einem Eventalphabet Ω besteht aus $|D|$ Sequenzen $S \in D$. Jedes $S \in D$ ist eine Sequenz von $|S|$ Events $e \in \Omega$, also $S \in \Omega^{|S|}$. Durch $S[i]$ sei das i-te Event in S bezeichnet und $S[i,j]$ bezeichne eine Teilsequenz von S, welche alle vom i-ten bis zum j-ten Event enthält $(S[i]...S[j])$. Weiter sei durch $||D||$ die Summe aller Längen der $S_i \in D$ bezeichnet, also $||D|| = \sum_{S_i \in D} |S_i|$.

Der *Support* eines Events e einer Sequenz S sei einfach die Anzahl des Auftretens von e in S, also $supp(e|S) = |\{i \in S | i = e\}|$. Der *Support* eines Events e in einer Datenbank D sei definiert als $supp(e|D) = \sum_{S \in D} supp(e|S)$.

Als Muster werden Serienepisoden verwendet, eine solche Serienepisode X sei einfach eine Sequenz von Events. Es wird gesagt, dass eine Sequenz S X enthält, falls es eine Teilsequenz von S gibt, die gleich X ist. Es ist hier zu beachten, dass Lücken zwischen den Events von X erlaubt werden.
Als *singleton Muster* sei ein einzelnes Event $e \in \Omega$ bezeichnet.

In dem vorgestellten Fall sollen als Modelle *Codetabellen* verwendet werden. Eine solche Tabelle ist eine Verweisdatenbank zwischen Mustern und dem Code. Eine solche Tabelle besteht aus vier Spalten. In der ersten Spalte werden die Muster abgespeichert und in der zweiten Spalte wird der Code zum Identifizieren der Muster gespeichert. Die beiden rechten Spalten sind musterabhängige Codes für die Identifizierung einer Lücke: In der dritten Spalte ist der Code, falls eine Lücke auftritt und in der vierten Spalte der Code, falls keine Lücke innerhalb eines Musters auftritt. Um sicher zu stellen, dass jede Sequenz über dem Alphabet Ω durch eine *Codetabelle* codiert werden kann, wird verlangt, dass alle *singleton Events* in dem Alphabet in der Codetabelle CT enthalten sind.

Um auf die verschiedenen Codes in der Codetabelle CT zu verweisen, wird durch $code_p(X|CT)$ auf den zum Muster X gehörenden in der zweiten Spalte von CT abgelegten Code verwiesen. Genauso wird mittels $code_g(X|CT)$ und $code_n(X|CT)$ auf die Codes in der dritten und vierten Spalte zugegriffen, welche zeigen, ob oder ob nicht das nächste Symbol Teil einer Lücke bei der Nutzung des Musters X ist. Für die Lesbarkeit wird häufig auf die explizite Nennung von CT verzichtet, sofern es aus dem Kontext hervorgeht.

2.2.2 Decodieren einer Datenbank

In diesem Abschnitt soll kurz das Decodieren einer Datenbank erläutert werden. Eine codierte Datenbank besteht immer aus zwei Code-Streams: C_p und C_g. Diese folgen immer aus dem Cover, durch welches die Datenbank codiert wurde.

Zum einen ist der *Muster-Stream* C_p eine Liste von $|C_p|$ Codes ($code_p(.)$) für Muster $X \in CT$, die zu den von dem Cover-Algorithmus ausgewählten Mustern gehöhren. Zum Beispiel wird die Sequenz *abc* durch $code_p(a)code_p(b)code_p(c)$ codiert.

Solche Serienepisoden sind jedoch keine einfachen Teilsequenzen, da in diesem Ansatz ja Lücken erlaubt sind: Ein Muster *de* sagt lediglich, dass nach einem Event d irgendwann ein Event e folgt. Dabei ist es durchaus möglich, dass zwischen diesen Events andere liegen. Dieses Muster tritt dann sowohl in der Sequenz *de*, als auch in der Sequenz *dfe* auf. Dabei ist in der ersten Sequenz keine Lücke zwischen den beiden Events und in der zweiten Sequenz ist eine Lücke der Größe eins, in welcher das Event f auftritt.
Angenommen man speichert die decodieren Events in einer Sequenz S_k, dann bedeutet dies, dass nur, wenn man den Code für ein *singleton Muster* X liest, man X eindeutig an die Sequenz S_k anhängen kann, da auf jeden Fall keine Lücke existiert. Ist X hingegen kein *singleton Muster*, so kann man nur das erste Event $x_1 \in X$ an S_k anhängen. Bevor man nun mit $x_2 \in X$ weiter verfährt, muss man erst überprüfen, ob bei dieser Anwendung des Musters X zwischen den beiden Events eine Lücke vorhanden ist oder nicht, und falls dem so ist, welche Events in die

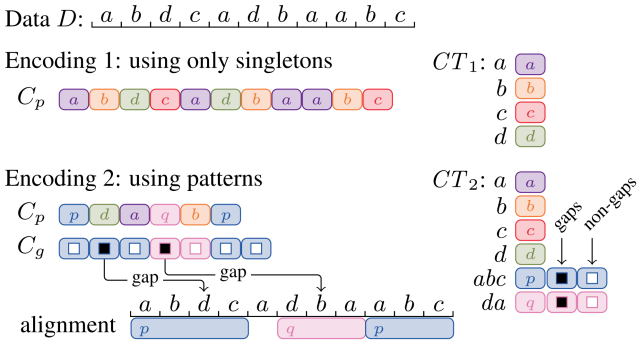


Abbildung 2.1: Beispiel für zwei mögliche Codierungen [1]

entstandene Lücke gehören.

Für diese Frage ist nun der zweite Code-Stream, der *Lücken-Stream* C_g, zuständig. Dieser ist eine Liste von Codes aus der dritten oder vierten Spalte von *CT*, die anzeigen, ob Lücken vorliegen oder nicht. Ist dieser Stream gegeben, kann man entscheiden, ob das nächste Event von S_k aus dem aktuellen Muster X stammt oder ob man das nächste Muster Y auslesen muss, um die Lücke zu füllen. Dies wird weiter so verfahren, es wird also in C_g nachgesehen, ob eine weitere Lücke vorliegt die gefüllt werden muss oder nicht, solange bis der Code für keine Lücke ($code_n(X)$) in C_p erscheint, welcher anzeigt, dass das nächste Event aus X an S_k angehängt werden kann.

Als Beispiel sei Abbildung 2.1 gegeben: Es ist möglich, nur durch *singleton Muster* zu codieren, was bedeuten würde, dass der *Lücken-Stream* leer wäre. Eine andere Möglichkeit wäre Muster zuzulassen. Das würde bedeuten, dass man für die Codierung von *abdc* zuerst den Code für *abc* in den *Muster-Stream* legt und ein „keine Lücke Symbol“ (weiß) gefolgt von einem „Lückensymbol“ (schwarz) in den *Lücken-Stream*. Darauf müsste dann der Code für *d* in den *Muster-Stream* und ein „keine Lücke Symbol“ (weiß) in C_g.

2.2.3 Codieren einer Datenbank

Durch das oben genannte Schema ist nun klar, welcher Code wann und wie gelesen werden muss.
Zuerst muss formal die Definition der Länge dieser Codes sowie die Codierungslänge der Codetabelle und der Datenbank aufgestellt werden.

Darauf soll in dieser Arbeit jedoch verzichtet werden. Eine formale Definition ist in [1] zu finden. Hier soll nur das *Minimale Codetabellen Problem* formal eingeführt werden:

Definition 2.2.1 (Minimales Codetabellen Problem)**:**
Sei Ω eine Menge von Events und sei D eine gegebene Sequenzdatenbank über Ω. Nun ist eine minimale Menge von Serienepisoden $\mathscr{P}$ zu finden, sodass für ein optimales Cover C von D – unter der Nutzung von $\mathscr{P}$ und Ω – die totalen Codierungskosten $L(CT, D)$ minimal sind, wobei CT die codeoptimale Codetabelle für C sei.

Es ist klar zu erkennen, dass der Suchraum für dieses Problem beliebig groß ist. Natürlich ist das Codieren oder Covern einer Sequenz wesentlich komplizierter als dessen Decodierung. Der Grund ist einfach: Hat man bei der Decodierung genau eine Möglichkeit zu verfahren, hat man bei der Codierung viele Möglichkeiten, eine Sequenz zu covern.

Abdecken eines Strings

Sei nun angenommen, dass die Sequenz $S_k \in D$ decodiert werden soll. Weiter soll die Decodierung eines Musters X bei $S_k[i]$ beginnen und bei $S_k[j]$ enden. Dann beschreibt $S_k[i,j]$ ein *aktives Fenster* für X.

Sei nun $\mathscr{P}$ die Menge der *nichtsingleton* Muster, die für die Codierung verwendet wurden. Dann sei das *Alignment* A als Menge aller *aktiven Fenster* der nichtsingleton Muster $X \in \mathscr{P}$ definiert:

$$A = \{(i,j,X,k) | S_k[i,j] \text{ ist ein aktives Fenster für } X, S_k \in D\}. \tag{2.1}$$

Geht man erneut auf das Beispiel in Abbildung 2.1 zurück, so wäre das *Alignment* für die zweite Codierung gegeben durch $\{(1,4,abc,1),(6,8,da,1),(9,11,abc,1)\}$.

Klar muss sein, dass so ein *Alignment* kein eindeutiges Cover einer Sequenz definiert, da in keinster Weise berücksichtigt wird, wie die möglicherweise innerhalb eines *aktiven Fensters* liegenden anderen Symbole codiert werden können. Ist aber für eine Datenbank D ein *Alignment* A gegeben, so kann nach [1] die Anzahl der für ein Codierungsschema benötigten Bits eines solchen Covers bestimmt werden.

Um dies zu sehen, sei X ein Muster und weiter $W = \{(i,j,X,k) \in A\}$. Dann ist klar, dass

$$usage(X) := |\{Y \in C_p | Y = code_p(X)\}| = |W|, \tag{2.2}$$

und wenn durch $gaps(X)$ die Anzahl der Lücken während der Nutzung des Musters X im Cover D bezeichnet ist, gilt

$$gaps(X) = gaps(W) := \sum_{(i,j,X,k) \in W} j - i - |X| - 1. \tag{2.3}$$

Die restlichen Symbole werden durch singleton Muster codiert, die *usage* eines solchen Musters $s \in \Omega$ ist also gleich

$$usage(s) = supp(s|D) - \sum_{s \in X} usage(X). \tag{2.4}$$

Hat man ein *Alignment* A für eine Datenbank D gegeben, dann kann man einfach ein Cover C konstruieren: Man arbeite A durch und überdecke S_k, sofern möglich durch Muster und durch Singletons anderenfalls. Dann kann man entweder direkt von C oder von A die verbundene codeoptimale Codetabelle CT herleiten.

Für die Notation sei eine solche – von A hergeleitete Tabelle – durch $CT(A)$ beschrieben, und für $L(D|CT(A)), L(D,CT(A))$ seien $L(D|A), L(D,A)$ eine Kurzschreibweise.
Die Frage, die sich noch stellt, ist, welche Art von Fenstern in einem optimalen *Alignment* auftreten können. Ein Fenster $W = S[i,j]$ wird als minimal für ein Muster X bezeichnet, falls W X enthält, jedoch ist X in keinem anderen echten Teilfenster von W enthalten. Wieder im Beispiel in Abbildung 2.1 ist für *da* $S[6,8]$ ein minimales Fenster, $S[6,9]$ jedoch nicht.
Nach [1] sind für ein *Alignment* A, welches eine optimale Codierungslänge produziert, alle aktiven Fenster minimal. Klar ist, dass daraus folgt, dass nur minimale Fenster zu untersuchen sind. Sei $\mathscr{F}$ eine Menge von Episoden und sei $X \in \mathscr{F}$. Da nun ein Event $S_k[i]$ nur der Startpunkt eines minimalen Fensters von X sein kann, sind höchstens $||D||$ minimale Fenster von X in D. Also ist die Anzahl der zu untersuchenden Fenster durch $||D|||\mathscr{F}|$ beschränkt. Nach [1] kann dann der FINDWINDOWS-Algorithmus verwendet werden, um alle minimalen Fenster eines Musters X in $\mathcal{O}(|X|||D||)$ zu finden.

Aufgrund der komplexen Beziehung zwischen der Codelänge und des *Alignment* ist das Finden des optimalen *Alignment* aber nicht trivial. Wenn man jedoch das *Alignment* festlegt, so liefern die Gleichungen (2.2), (2.3) und (2.4) die Codes, um $L(D|A)$ zu optimieren.
Nun soll aber wie in [1] das Gegenteil betrachtet werden: Angenommen man fixiert die Codes, so kann das $L(D|A)$-optimierende *Alignment* einfach gefunden werden.
Angenommen $w = (i,j,X,k)$ sei ein minimales Fenster für ein Muster X. Der *Gain* (etwa Nutzen) wird dann definiert als

$$gain(w) = -L(code_p(X)) - (j-i-|X|)L(code_p(X)) - (|X|-1)L(code_n(X)) + \sum_{x \in X} L(code_p(x)). \tag{2.5}$$

Aus [1] ist bekannt, dass für eine Datenbank D und ein *Alignment* A die Länge der Verschlüsselung von D gleich

$$L(D|A) = c - \sum_{w \in A} gain(w) \tag{2.6}$$

ist, wobei c eine von A unabhängige Konstante sei.

Daraus ist zu schließen, dass man nach der Fixierung der Codelängen nur den totalen *Gain* maximieren muss. Also muss zu einer gegebenen Menge W, die aus allen minimalen Fenstern gegebener Muster besteht, nur eine Teilmenge $O \subset W$ mit disjunkten Fenstern gefunden werden, die den *Gain* maximiert.

O.B.d.A. sei angenommen, dass W nach dem ersten Index jeden Fensters sortiert sei. Für ein Fenster w sei nun durch $next(w)$ das nächste disjunkte Fenster in W beschrieben. Sei weiter $o(w)$ der optimale totale *Gain* von w und dessen darauffolgenden Fenstern. Sei v das nächste Fenster von w, dann gilt $o(w) = \min(o(v), gain(w) + o(next(w))$. Dieser Zusammenhang liefert das folgende einfache dynamische Programm ALIGN:

ALIGN(W)

```
1 Eingabe: minimale Fenster W, sortiert nach dem ersten Event
```

```
2  Ausgabe: disjunkte Teilmenge von W mit optimalem Gain
3  o(N+1) = 0; opt(N+1) = none;
4  for i=N,...,1 do
5   c = 0;
6   if(next(i)) then c = o(next(i));
7   if(gain(w_i ) + c > o(i+1) then
8    o(i) = gain(w_i  ) + c;opt(i) = i;
9   else
10   o(i) = o(i+1); opt(i) + opt(i+1);
11 O = optimale Alignment
```

Dieser Algorithmus kann dann iterativ verwendet werden: Kennt man die Codes, so kann man die optimalen *Alignments* bestimmen und daraus die optimalen Codes herleiten. Das kann wiederholt werden, bis eine Konvergenz eintritt, welche durch eine heuristische Approximation des optimalen *Alignment* A^* von D unter der Nutzung der Muster $\mathscr{P}$ gegeben ist.
Die initialen Werte sind durch die Anzahl der minimalen Fenster als *usage* gegeben und die Größe der Lücken-Codes wird auf ein Bit gesetzt. Der Pseudocode ist durch *SQS* (Summarising event seQuenceS) gegeben.

SQS(D, $\mathscr{P}$)

```
1  Eingabe: Sequenzdatenbank D , Menge von Mustern P
2  Ausgabe: Alignment A
3  for s∈Ω do usage(s) = supp(s|D)
4  for X∈P,|X|>1 do
5    W_X  = FINDWINDOWS(X,D);
6   usage(X) = |W_X| ;
7   gaps(X) = |X| − 1;
8  W = MERGESORT( {W_X}_{X∈P} , nach dem ersten Element);
9  for changes do
10   Berechne den gain für alle w∈W
11   A = ALIGN(W);
12   berechne usage und gaps durch A;
13 return A;
```

Die Komplexität für einzelne Iterationen ist nach [1] die Komplexität von ALIGN(W), welche $\mathscr{O}(|W|) \subset \mathscr{O}(|\mathscr{P}|, ||D||)$ ist. Zu beachten ist auch, dass *next* berechnet ist, bevor ALIGN aufgerufen wird und dass das auch durch eine einfache Abfrage getan werden kann, welche $\mathscr{O}(|W|)$ Schritte benötigt. Auch zu beachten ist, dass die Verschlüsselungslänge in jeder Iteration verbessert wird, und da es nur eine endliche Anzahl an *Alignments* gibt, wird SQS gegen ein lokales Optimum in einer endlichen Zeit konvergieren.
In den Experimenten in [1] sind es in der Regel weniger als 10 Schritte.

2.2.4 Erstellen der Codetabellen

Im ersten Teil dieses Abschnitts soll nun skizziert werden, wie aus einer großen Menge $\mathscr{F}$ von Mustern die besten Muster gefiltert werden können, um das optimale *Alignment A* zu erhalten,

sodass die damit verbundene Codetabelle CT $L(D, CT)$ minimiert.

Als Notation sei $L(D, \mathscr{P})$ als Kurzschreibweise für die gesamte Codierungslänge $L(D, CT)$, welche aus der Codetabelle CT gewonnen wird, die eine Menge von Mustern $\mathscr{P}$ und Singletons Ω enthält und codeoptimal für das *Alignment A* ist.

Am Anfang werden die Kandidaten $X \in \mathscr{F}$ nach $L(D, \{X\})$ von klein nach groß sortiert. Danach wird nach dem Greedy-Prinzip jedes Muster $X \in \mathscr{F}$ getestet: Wenn nach dem Hinzufügen von X zu $\mathscr{P}$ der Wert verbessert wird – also weniger Bits benötigt werden – so wird X in $\mathscr{P}$ behalten, sonst wird es dauerhaft entfernt. Der Pseudocode für dieses Verfahren ist in SQS-CANDIDATES zu finden:

SQS-CANDIDATES($\mathscr{F}, D$)

```
1  Eingabe: Kandidatenmenge F, Datenbank D,
2  Ausgabe: Menge von nicht singleton Mustern P, welche heuristisch das MCTP
       lösen;
3  sortiere die Muster X ∈ F nach L(D,{X});
4  P = ∅;
5  for X ∈ F do
6    if L(D, P ∪ X) < L(D, P) then
7      mathcalP = PRUNE(P ∪ X, D; false);
8  P = PRUNE(P, D, true);
9  sortiere die Muster X ∈ P nach L(D, P) − L(D, P \ X);
10 return P;
```

PRUNE($\mathscr{P}, D, full$)

```
1  Eingabe: Menge von Mustern P, Datenbank D, bool−Variable full,
2  Ausgabe: geprunte Menge von Mustern P;
3  for X ∈ P do
4   CT = Codetabelle zu SQS(D, F);
5   CT' = Codetabelle zu CT ohne X;
6   g = Σ_{w∈A} gain(w);
7   if full or g < L(CT) − L(CT') then
8     if L(D, P \ X) < L(D, P) then
9       P = P \ X;
10 return P;
```

Während der Suche wird die Codetabelle iterativ aktualisiert. Es kann daher passieren, dass im Laufe der Zeit hinzugefügte Muster anfangen, der Kompression zu schaden, und dass deren Rolle von anderen, spezifischeren Mustern übernommen wird. Daher werden sie möglicherweise redundant und aus $\mathscr{P}$ entfernt.
Aus diesem Grund werden redundante Muster nach jedem erfolgreichen Schritt gepruned. Während dem Pruning wird jedes Muster $Y \in \mathscr{P}$ - sortiert nach dem Hinzufügen - betrachtet. Angenommen $\mathscr{P} \setminus Y$ verbessert die Codierungslänge, so wird Y aus $\mathscr{P}$ entfernt.
Dieser Test kann in jedem erfolgreichen Schritt sehr zeitintensiv werden, daher wird in diesem Fall eine einfache Heuristik verwendet: Ist der totale Gewinn der Fenster von X höher, als die

Kosten für X in der Codetabelle, so wird X nicht getestet.

Ist der Algorithmus SQS-CANDIATES beendet – wurden also alle Muster aus $\mathscr{F}$ untersucht – wird ein letzter Pruning-Schritt ohne Heuristik angewendet. Am Ende werden die Muster in $\mathscr{P}$ nach der Auswirkung sortiert, die das Entfernen von X aus $\mathscr{P}$ hätte, also nach der Differenz $L(D,\mathscr{P}) - L(D,\mathscr{P} \setminus X)$. Diese Sortierung gibt an, welche Muster in $\mathscr{P}$ als am wichtigsten zu werten sind.

Die Ausführung von SQS-CANDIDATES ist nach [1] in $\mathscr{O}(|\mathscr{F}^3|||D||)$ möglich, in der Realität meist jedoch viel schneller.

Nun soll zum Abschluss noch aus [1] skizziert werden, wie gute Codetabellen direkt von den Daten gewonnen werden können.

Um die generelle Idee zu erläutern, sei $\mathscr{P}$ eine Menge von Mustern. Nun sollen iterativ Muster der Form XY gefunden werden, die $L(D,\mathscr{P} \cup \{XY\})$ minimieren, wobei $X,Y \in \mathscr{P} \cup \Omega$ gelte. XY wird zu $\mathscr{P}$ hinzugefügt, bis keine Verbesserung mehr möglich ist. Der Nachteil dieses direkten Verfahrens ist, dass das Testen jeder möglichen Kombination nach [1] in $\mathscr{O}((|\mathscr{P}| + |\Omega|)^2(|\mathscr{P}| + 1)||D||)$ zu lösen ist.

Aus diesem Grund müssen zum Lösen Heuristiken verwendet werden. Um das Finden von guten Kandidaten zu garantieren, wurde in [1] eine Heuristik entwickelt, die für ein gegebenes Muster P ein neues Muster PQ findet, mit einem hohen Gewinn, in nur $\mathscr{O}(||P| + |\Omega| + ||D||)$.
Für den ersten Schritt sei folgendes Lemma gegeben:

Lemma 2.2.2:
Sei eine Datenbank D und ein Alignment *A gegeben. Seien P und Q zwei Muster. Weiter seien $V = \{v_1,...,v_N\}$ und $W = \{w_1,...,w_N\}$ zwei Megen von möglichen Fenstern für P bzw. Q. Nun sei angenommen, dass P bzw. Q singletons sind oder jedes $v_i \in V$ bzw. $w_i \in W$ in A auftritt. Weiter sei angenommen, dass v_i und w_i in derselben Sequenz k_i auftreten und man schreibe $v_i = (a_i, b_i, P, k_i)$ und $w_i = (c_i, d_i, Q, k_i)$. Weiter gelte $b_i < c_i$. Sei nun $R := PQ$ und $U := \{(a_1, d_1, R, k_1),...,(a_N, d_N, R, k_N)\}$.*
Nun gelte, dass U keine überlappenden Fenster und keine überlappenden Fenster mit $A \setminus (V \cup W)$ beinhaltet.
Dann hängt die Differenz

$$L(D, A \cup U \setminus (V \cup W)) - L(D,A) \tag{2.7}$$

nur von N, $gaps(V)$, $gaps(W)$ und $gaps(U)$ ab und kann in konstanter Zeit berechnet werden.

Beweis. Siehe Anhang [1]. □

Dieses Lemma sagt nun aus, dass, wenn man N aktive Fenster von P sowie N aktive Fenster von Q betrachtet und diese in N aktive Fenster von PQ überführt, dann die Differenz der kompletten Codierungslänge in konstanter Zeit berechnet werden kann. Die Voraussetzungen zu diesem Lemma werden benötigt, damit $A \setminus (V \cup U)$ ein korrektes *Alignment* ist.
Sei die oben in Lemma 2.2.2 genannte Differenz im Folgenden durch $diff(V, W, U; A, D)$ bezeichnet. Klar ist, dass dieser Wert auch von A und D abhängt, diese aber in diesem Fall konstant

sind.
Da nun deutlich geworden ist, wie der Gewinn bei der Nutzung von Fenstern für PQ berechnet werden kann, stellt sich nun die Frage, welche Fenster in einem *Alignment* genutzt werden sollten.

Nach [1] gilt folgendes: Ist eine Datenbank D und ein *Alignment* A gegeben und sei $v = (i, j, X, k) \in A$. Gibt es dann ein Fenster $S_t[a, b]$, welches X enthält, sodass $w = (a, b, X, l)$ mit keinem Fenster in A überlappt und $b - a < j - i$ gelte, dann ist A kein optimales *Alignment*.

Daraus wird in [1] eine Idee für eine Heuristik entwickelt: Man fängt an, die minimalen Fenster von PQ von klein nach groß zu sortieren. In jedem Schritt wird nun der Score mittels Lemma 2.2.2 berechnet und unter diesen der optimale ausgewählt.
Betrachtet man nun jedes Q unabhängig, so kann eine Abarbeitung nicht unbedingt in linearer Zeit garantiert werden. Aus diesem Grund wird nach allen Kandidaten gleichzeitig gesucht. Um Linearität zu garantieren, werden nur aktive Fenster von P und von Q betrachtet und keine *Singletons*, die in den Lücken auftreten. Der Scan startet mit dem Finden aller aktiven Fenster in P und es wird fortgefahren durch das Scannen aller darauffolgenden Muster. Das Vorgehen wird so geordnet, dass die neuen, minimalen Fenster der Größe nach geordnet sind, von klein nach groß. Der Scan wird gestoppt entweder wenn die Sequenz zu Ende ist oder P das nächste Mal auftritt.
Jedoch gibt es zwei weitere Einschränkungen, die in diesem Fall beachtet werden sollten: Wenn minimale Fenster von PQ gefunden werden sollen, muss sichergestellt werden, dass diese auch zum *Alignment* hinzugefügt werden können. Also dürfen sich zwei neue minimale Fenster nicht schneiden. Weiter dürfen die einzigen beiden Fenster, mit denen sich ein neues minimales Fenster in dem *Alignment* schneiden darf, die beiden sein, aus denen es konstruiert wurde.

ESTIMATE(P,A,D)

```
1  Eingabe: Datenbank D, aktuelle \emph{Alignment} A, Muster P∈CT,
2  Ausgabe: Muster PX mit X∈CT und geringem L(D,A∪PX);
3  for X∈CT do
4   V_x = ∅;
5   W_x = ∅;
6   U_x = ∅;
7   d_x = ∅;
8  T = ∅;
9  for Vorkommen v von P in der Codierung (ignorieren von Lücken) do
10  (a,b,P,k) = v;
11  d = der letzte Index des aktiven Fensters nach v;
12  t = (v,d,0);
13  l(t) = d − a;
14 while T nicht leer do
15  t = argmin_{u∈T} l(u);
16  (v,d,s) = t;
17  a = erster Index von v;
18  w = (c,d,X,k) = aktives Fenster eines Musters, das bei d endet;
19   if X = P and Event bei a oder d ist markiert then
```

```
20    lösche t aus T;
21    continue;
22   if S_k[a,d] ist ein minimales Fenster von PX then
23    füge v zu V_X;
24    füge w zu W_X;
25    füge (a,d,PX,k) zu U_X;
26    d_X = min(diff(V,W,U;A)+s,d_X);
27    if |X| > 1 then
28     s = s + gain(w);
29    if X = P then
30     markiere die Events bei a und d;
31     lösche t von T;
32     continue;
33   if w ist das letzte Fenster der Sequenz then
34    lösche t von T;
35   else
36    d = der letzte Index des aktiven Fensters nach w;
37    update t zu (v,d,s);
38    update l(t) zu d − a;
39  return PXP mit dem niedrigsten Wert d_X;
```

Nach [1] liefert ESTIMATE$(P,\emptyset,D)$ ein Muster mit optimalem Wert und kann in $\mathcal{O}(|\Omega|+|\mathscr{P}|+||D||)$ ausgeführt werden.

Die tatsächliche Suche wird vom Algorithmus SQS-SEARCH ausgeführt. Dieser ruft ESTIMATE für jedes Muster P auf. Weiter sortiert dieser Algorithmus die erhaltenen Mustern nach dem berechneten Score und versucht, diese zur Codierung hinzuzufügen, ähnlich wie SQS-CANDIDATES.

SQS-SEARCH(D)

```
1   Eingabe: Datenbank D,
2   Ausgabe: wichtige Muster 𝒫;
3   𝒫 = ∅;
4   A = SQS(D,∅);
5   while changes do
6    ℱ = ∅;
7    for P ∈ CT do
8     füge ESTIMATE(P,A,D) zu ℱ;
9    for X ∈ ℱ ordered by ESTIMATE do
10    if L(D,𝒫 ∪ X) < L(D,𝒫) then
11     mathcalP = PRUNE(𝒫 ∪ X,D;false);
12    if X wurde hinzugefügt then
13     Teste rekursiv X erweitert um Events, die in den Lücken auftreten;
14  𝒫 = PRUNE(𝒫,D,true);
15  sortiere die Muster X ∈ 𝒫 nach L(D,𝒫) − L(D,𝒫 \ X);
16  return 𝒫;
```

War der Schritt erfolgreich - ist also ein Muster X hinzugefügt worden, so wird nach den Lückenevents gesucht, die in den aktiven Fenstern von X auftreten. Im Anschluss werden Muster, die aus der Zusammensetzung von X mit Lückenevents gewonnen wurden, getestet.

3 Finden häufiger Zugsequenzen

In diesem Abschnitt soll vorgestellt werden, wie der oben beschriebene Ansatz auf das Finden häufiger Zugsequenzen übertragen wurde und wie die Implementierung realisiert wurde.

3.1 Idee des Programms

Für jede Ausführung muss eine zu untersuchende Stellung und eine Schachdatenbank gegeben sein. Die Vergleichsstellung muss dabei in der normalen Schachnotation gegeben sein – es wird dann die Endstellung zur weiteren Arbeit verwendet.
Nachdem die Daten eingelesen wurden, wird für die Vergleichsstellung ein Hashcode berechnet. Dieser dient zum einfacheren Vergleich von zwei Stellungen. Klar muss sein, dass dieser Hashcode zwar kein hinreichendes Kriterium für das Vorliegen von gleichen Stellungen darstellt, jedoch ein notwendiges.
Der Hashcode einer Stellung ist in diesem Fall die Summe der Hashcodes der darauf existierenden Figuren. Der Hashcode einer Figur ist wiederum gleich (10 * Farbe + Typ) * Linie * Reihe. Dabei ist Typ der Wert des Figurentyps (siehe dazu Tabelle 3.1), Farbe die Farbe der Figur (Weiß (1), Schwarz (2)), der Linie die aktuelle Position in y-Richtung (0,...,7) und Reihe die Position in x-Richtung (a (0), ..., h (7)).

Nun werden nacheinander alle in der Schachdatenbank gegebenen Partien abgearbeitet. Dazu werden alle in einer Partie vorkommenden Stellungen mit der Vergleichsstellung verglichen: Zuerst wird überprüft, ob der Hashcode gleich ist. Ist dies der Fall, werden alle Schachquadrate der beiden Stellungen nacheinander verglichen, ob die exakt gleiche Stellung vorliegt. Nach jedem nicht erfolgreichen Vergleich wird der nächste Zug der Partie auf die Stellung angewendet, der Hashcode aktualisiert und der Vergleich von vorne begonnen.

Ist eine gleiche Stellung gefunden, wird die Fortsetzung – also die als Nächstes auftretenden Züge der Partie – in eine Liste von Zugsequenzen gespeichert und die weitere Abarbeitung der Datenbank vorgenommen.

Sind alle Partien untersucht, wird die gespeicherte Liste von bekannten Fortsetzungen für die Anwendung auf den *SQS-Algorithmus* vorbereitet. Da es ein Unterschied in der theoretischen

Figur	HashCode
König	1
Dame	2
Läufer	3
Springer	4
Turm	5
Bauer	6

Tabelle 3.1: Hashcode Tabelle

Idee ist, ob ein Zug vom weißen oder vom schwarzen Spieler getätigt wird, werden allen Zügen voranstehend der Buchstabe *w* für einen Zug vom weißen Spieler und der Buchstabe *s* für einen Zug vom schwarzen Spieler zugeordnet.

Dann wird die Liste von Zugsequenzen in eine Liste von Integersequenzen umgewandelt. Dies geschieht kurzerhand, indem dem ersten auftretenden Zug die Eins, dem zweiten Zug die Zwei und so weiter zugeordnet wird. Tritt derselbe Zug häufiger auf, bekommt er immer dieselbe bereits vorher gewählte Zahl zugeordnet. Zum schnellen Finden werden *HashMaps* Zug-Zahl und Zahl-Zug verwendet. Die zweite Map ist für eine möglichst schnelle Decodierung nach dem Anwenden des Algorithmus gedacht.
Nun wird der *SQS-Algorithmus* mit drei verschiedenen Listen von Sequenzen gestartet: mit den Zügen beider Spieler, nur mit den Zügen des weißen Spielers und nur mit den Zügen des schwarzen Spielers. Die entstandenen Codetabellen werden durch die *HashMap* in Zugsequenzen codiert und dem Betrachter ausgegeben.

3.2 Implementierung

Die zur Verarbeitung der Daten genutzten Programme wurden in Java implementiert. Es wurde mit *tud.compress.main* nur ein neues Package implementiert. Die anderen Funktionalitäten ergaben sich aus der Nutzung und Weiterentwicklung von *Open Source Software*.
Im nächsten Abschnitt sind die Aufgaben der Klassen des Packages *tud.compress.main* beschrieben. Am Ende des Kapitels wird die externe Software erläutert.

3.2.1 Responsibilities

In diesem Abschnitt sollen die Verantwortlichkeiten der in dem Package *tud.compress.main* implementierten Klassen erläutert werden.

tud.compress.main

- *ContinuationFinder:* Liest die Vergleichsstellung ein und koordiniert das Programm
- *MainConverter:* Ruft die Liste der gleichen Stellungen ab, startet den externen Algorithmus und gibt das Resultat aus
- *StringToIntConverter:* Ist für die Konvertierung der Zugfolgen in Integer-Reihen und umgekehrt zuständig
- *PgnIoReader:* Wandelt die Resultate des externen Programms in Zugfolgen um
- *FindEqualityStates:* Liefert der Vergleichsstellung identische Stellungen und deren Fortsetzungen

3.2.2 Externe Software

Es wurden mehrere externe Programme und Bibliotheken verwendet.

SQS

SQS ist ein in C++ geschriebenes Programm zum Finden von signifikanten Mustern. Es liefert die in Abschnitt 2 beschriebenen Möglichkeiten: Zu einer Anzahl von Integersequenzen werden die – im oben beschriebenen Sinne – besten Sequenzen ausgegeben. Der Quellcode und eine Dokumentation ist unter `http://people.mmci.uni-saarland.de/~jilles/prj/sqs/` zu finden. Der Code wurde zu einer ausführbaren Datei umgewandelt und im Javacode eingebunden.

Java PGN Parser

Der *Java PGN Parser* ist ein Open Source Projekt, mit welchem *PGN-Dateien* in Zugfolgen umgewandelt werden können. Der Code ist unter `http://sourceforge.net/projects/pgnparse/` zu finden. Da in der aktuellen Version alle Partien einer Datei zuerst eingelesen und dann verarbeitet werden, mussten die Abläufe leicht angepasst werden, um Speicherplatz zu sparen.

Toolchain VN

Toolchain VN ist eine in [2] beschriebene Software. Diese bietet die Möglichkeit Zugfolgen in Stellungen zu konvertieren. Dieses Programm wurde für diese Verwendung noch um die Möglichkeit des Stellungsvergleichs erweitert.

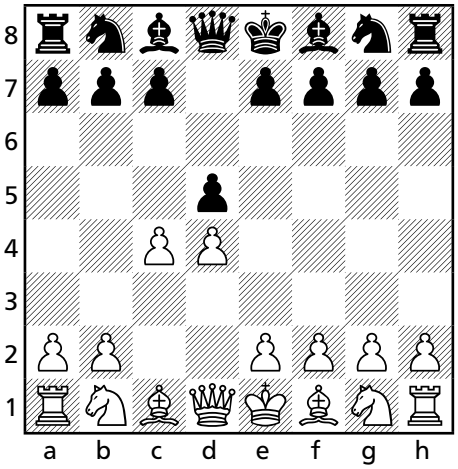
4 Experimente

In den Experimenten sollen nun verschiedene Stellungen auf mögliche Zugstrategien untersucht werden. Dazu wird dem *SQS*-Algorithmus sowohl die aneinanderhängenden Halbzüge beider Spieler übergeben, als auch nur die Züge vom weißen Spieler und nur die schwarzen Halbzüge in zwei weiteren Rechenschritten. Die vorgestellten Buchstaben *w,s* markieren – wie im letzten Kapitel beschrieben – von welcher Spielfarbe der Zug ausgeführt wurde.

Die Berechnungen wurden auf dem Rechnercluster der Forschungsgruppe *künstliche Intelligenz* der TU-Darmstadt ausgeführt.

4.1 Experiment 1: Damengambit

Im ersten Experiment soll eine Stellung aus der Eröffnung untersucht werden, das Damengambit. Dazu wird folgende Stellung zur Untersuchung gegeben:
1 d4 d5 2 c4



Es wurde aufgrund des häufigen Auftretens dieser Stellung die recht kleine Schachdatenbank *ICOfY Base*[1] verwendet. Es wurden im Test 667 übereinstimmende Stellungen ermittelt, die alle verwendet wurden. Die Suchtiefe betrug höchstens 20 Halbzüge.
Die ausgewerteten Ergebnisse sind – sortiert nach der Häufigkeit des Vorkommens – in Tabelle 4.1 und sortiert nach der Größe der Sequenz in Tabelle 4.2 zu sehen. Die Berechnung benötigte 37 Sekunden. Es wurden für beide Zugfarben 78 Zugmuster gefunden, für die schwarzen Züge 34 und für die weißen Züge 24.
In der längsten Zugfolge für beide Farben sind alle Züge nacheinander ausführbar. Die Anwendung liefert dann die folgende Stellung:

[1] http://sourceforge.net/projects/icofybase

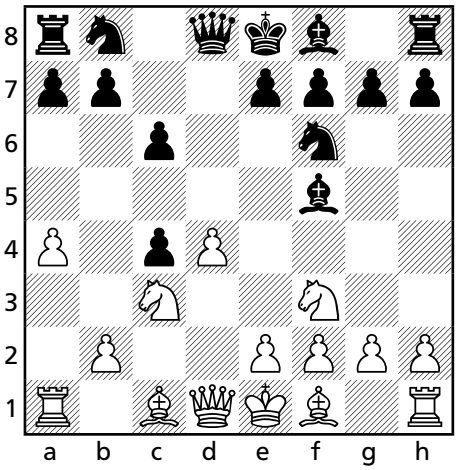
Züge beider Spielfarben		Züge von Weiß		Züge von Schwarz	
#	Zugfolge	#	Zugfolge	#	Zugfolge
102	bc6 wNf3 bNf6	103	we3 wBd3 wO-O	88	bBe7 bO-O
89	wcxd5 bexd5	70	wNf3 wg3 wBg2 wO-O	81	be6 bNf6
70	we3 wBd3	64	we3 wBe2 wO-O	77	be6 bNf6 bBe7 bO-O
65	bdxc4 wBxc4	59	we3 wBd3	61	bc6 bNf6
62	be6 wNc3	55	we3 wBxc4 wO-O	60	bb6 bBb7

Tabelle 4.1: Zugfolgen des ersten Experiments sortiert nach dem Auftreten

Züge beider Spielfarben		
#	Länge	Zugfolge
25	7	bc6 wNf3 bNf6 wNc3 bdxc4 wa4 bBf5
Züge von Weiß		
#	Länge	Zugfolge
35	5	wNc3 wcxd5 wBg5 we3 wBd3
Züge von Schwarz		
#	Länge	Zugfolge
17	6	bc6 bNf6 be6 bNbd7 bBd6 bO-O

Tabelle 4.2: Längste Zugfolgen des ersten Experiments

1 d4 d5 2 c4 c6 3 ♘f3 ♘f6 4 ♘c3 d×c4 5 a4 ♗f5



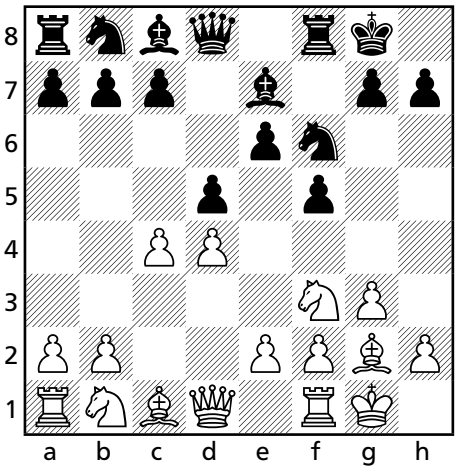
4.2 Experiment 2: Klassische Stonewall-Verteidigung

In diesem Schritt soll die *klassische Stonewall-Verteidigung* untersucht werden. Dies ist eine Stellung in der *Holländischen Verteidigung*. Die zu untersuchende Stellung ist:

Züge beider Spielfarben		Züge von Weiß		Züge von Schwarz	
#	Zugfolge	#	Zugfolge	#	Zugfolge
128	wcxd5 bexd5	132	wb3 wBb2	108	bb6 bBb7
79	bb6 bBb7	67	wb3 wBa3 wBxe7	68	bc6 bNe4 bNd7
74	bQe8 bQh5	60	wNd3 wNfe5	61	bc6 bBd7 bBe8
71	bBd7 bBe8	45	wNbd2 wb3 wBb2	59	bc6 bQe8 bNbd7
59	wBxe7 bQxe7	44	wNbd2 wNe5 wNdf3	58	bc6 bNbd7

Tabelle 4.3: Zugfolgen des zweiten Experiments sortiert nach dem Auftreten

1 d4 f5 2 g3 ♘f6 3 ♗g2 e6 4 ♘f3 ♗e7 5 O-O O-O 6 c4 d5



Es wurde die Schachdatenbank *Millionbase 2.2*[2] mit ca. 2,2 Millionen Partien verwendet. Durch das Programm wurden 812 gleiche Stellungen gefunden, die Suchtiefe betrug 20 Halbzüge.
Die Ergebnisse sind in den Tabellen 4.3 und 4.4 zu sehen. Die Berechnung benötigte 357 Sekunden. Es wurden für beide Zugfarben 124 Zugmuster gefunden, für die schwarzen Züge 47 und für die weißen Züge 51.

Die am häufigst gespielte Fortsetzung ist mit 7. cxd5 exd5 das Öffnen des Zentrums. Auch wird oft - mit verschiedenen Zügen von Weiß - durch 7. ... b6 8. ... Bb7 der schwarze Läufer fianchettiert. Weiter tritt der schwarze Damenausfall - durch 7. ... Qe8 8. ... Qh5 - zum Angriff oft auf.

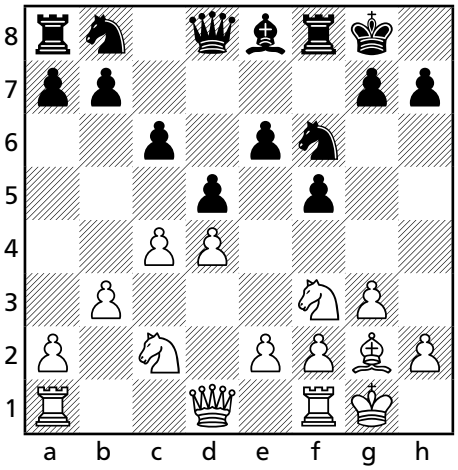
In der längsten Zugfolge für beide Farben sind alle Züge nacheinander ausführbar. Die Anwendung liefert dann die folgende Stellung:

[2] http://www.top-5000.nl/pgn.htm

Züge beider Spielfarben		
#	Länge	Zugfolge
10	8	wb3 bc6 wBa3 bBxa3 wNxa3 bBd7 wNc2 bBe8
Züge von Weiß		
#	Länge	Zugfolge
40	5	wb3 wBa3 wNxa3 wNc2 wNce1
Züge von Schwarz		
#	Länge	Zugfolge
17	5	bc6 bBxa3 bBd7 bBe8 bNbd7

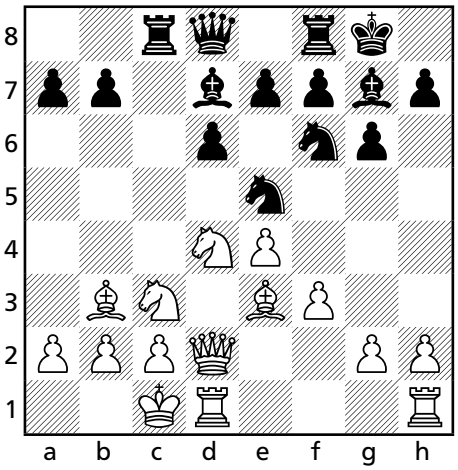
Tabelle 4.4: Längste Zugfolgen des zweiten Experiments

1 d4 f5 2 g3 ♘f6 3 ♗g2 e6 4 ♘f3 ♗e7 5 O-O O-O 6 c4 d5 7 b3 c6 8 ♗a3 ♗×a3 9 ♘×a3 ♗d7 10 ♘c2 ♗e8



4.3 Experiment 3: Drachenvariante

In diesem Schritt soll die *Drachenvariante* untersucht werden. Dies ist eine Stellung in der *Sizilianischen Verteidigung*. Die zu untersuchende Stellung ist:
1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 g6 6 ♗e3 ♗g7 7 f3 O-O 8 ♗c4 ♘c6 9 ♕d2 ♗d7 10 O-O-O ♖c8 11 ♗b3 ♘e5



Züge beider Spielfarben		Züge von Weiß		Züge von Schwarz	
#	Zugfolge	#	Zugfolge	#	Zugfolge
185	wh4 bh5	143	wNd5 wexd5	379	bNc4 bRxc4
124	bNc4 wBxc4 bRxc4	126	wBh6 wBxg7	144	ba5 ba4
121	wBxg7 bKxg7	112	wh4 wBg5	140	bh5 bNc4 bRxc4
121	wh4 bh5 wBg5 bRc5	96	wh4 wBxc4 wh5 wg4	94	bNc4 bRxc4 bNxh5 bNf6
104	wg5 bNh5	83	wh5 whxg6	81	bQc7 bRc8

Tabelle 4.5: Zugfolgen des dritten Experiments sortiert nach dem Auftreten

Züge beider Spielfarben		
#	Länge	Zugfolge
33	15	wh4 bh5 wBh6 bNc4 wBxc4 bRxc4 wBxg7 bKxg7 wg4 bhxg4 wh5 bRh8 whxg6 bfxg6 wf4
Züge von Weiß		
#	Länge	Zugfolge
14	10	wKb1 wBxc4 wg4 wb3 wNdxb5 wa4 wNd5 wNxe7 wRxd2 wNxd6
Züge von Schwarz		
#	Länge	Zugfolge
15	9	bNc4 bRxc4 bb5 bRc8 bQa5 ba6 bQxd2 bKh8 bRce8

Tabelle 4.6: Längste Zugfolgen des dritten Experiments

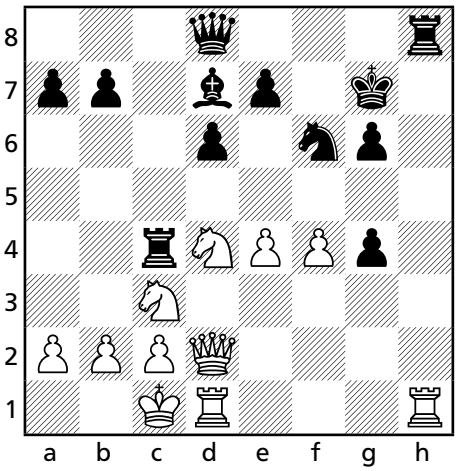
Es wurde erneut die Schachdatenbank *Millionbase 2.2*[3] mit ca. 2,2 Millionen Partien verwendet. Durch das Programm wurden 1897 gleiche Stellungen gefunden, die Suchtiefe betrug 20 Halbzüge.
Die Ergebnisse sind in den Tabellen 4.5 und 4.6 zu sehen. Die Berechnung benötigte 478 Sekunden. Es wurden für beide Zugfarben 336 Zugmuster gefunden, für die schwarzen Züge 132 und für die weißen Züge 160.

Die am häufigsten auftretende Zugfolge für beide Farben ist der Vorstoß auf dem Königsflügel 12. h4 h5. Die am zweit häufigsten auftretende Idee ist, nach verschiedener Fortsetzung des weißen Spielers, der Figurentausch auf c4: 12. ... Nc4 13. Bxc4 Rxc4.

Die längste Zugfolge ist am Stück spielbar und liefert folgende Stellung:

[3] http://www.top-5000.nl/pgn.htm

1 e4 c5 2 ♘f3 d6 3 d4 c×d4 4 ♘×d4 ♘f6 5 ♘c3 g6 6 ♗e3 ♗g7 7 f3 O-O 8 ♗c4 ♘c6 9 ♕d2 ♗d7 10 O-O-O ♖c8 11 ♗b3 ♘e5 12 h4 h5 13 ♗h6 ♘c4 14 ♗×c4 ♖×c4 15 ♗×g7 ♔×g7 16 g4 h×g4 17 h5 ♖h8 18 h×g6 f×g6 19 f4



5 Fazit

In dieser Arbeit wurde die Möglichkeit des Findens von Zugfortsetzungen im Schach mittels *Data Mining* und dem *Minimum-Discription-Length-Prinzip* untersucht.

Dazu wurde zuerst das *MDL*-Prinzip allgemein beschrieben und eine Anwendung auf eine Datenbank von Eventsequenzen vorgestellt.

Im nächsten Kapitel wurde beschrieben, wie dieses Prinzip auf eine Datenbank von Zugsequenzen übertragbar ist: Es werden aus einer Schachdatenbank alle gleichen Stellungen extrahiert und die so erhaltene Liste von Zugsequenzen wird in Integersequenzen umgewandelt. Diese Liste wird dann in eine Implementierung des oben vorgestellten Prinzips übergeben.

Im experimentellen Teil wurden drei Stellungen aus dem Eröffnungsbereich untersucht. Dabei war auffällig, dass die gefundenen Fortsetzungen die Ideen von diesen Stellungen korrekt widerspiegeln. Es war jedoch nicht möglich längere Strategien zu finden: Die meisten Sequenzen waren eher kurz (zwei bis drei Halbzüge) und zusammenhängend.

Zukünftig wäre eine weitere Betrachtung dieser Herangehensweise interessant, da die theoretische Anwendung zwar erfolgreich war, nur die Resultate nicht die gewünschten Ergebnisse liefern konnten. Weiter wäre es interessant, ob eine Betrachtung ausschließlich der Bauernstruktur bessere Resultate liefert: Sind die Strukturen gleich, werden alle folgenden Bauernzüge untersucht.

Eine Übertragung auf das Mittelspiel oder sogar Endspiel wäre auch interessant. Jedoch würden dann wohl nicht mehr viele komplett identische Stellungen gefunden werden können. Es ist dann zu überlegen, ob es sinnvoll wäre, nicht eine komplette Übereinstimmung der Stellungen zu fordern, sondern zum Beispiel nur die interessantesten Brettabschnitte zu vergleichen. Eine weitere Möglichkeit wäre es, eine bestimmte Anzahl von unterschiedlichen Figuren bei den Stellungen zu tolerieren.

6 Anhang

6.1 Abbildungsverzeichnis

2.1 Beispiel für zwei mögliche Codierungen [1] 5

6.2 Tabellenverzeichnis

3.1 Hashcode Tabelle 14

4.1 Zugfolgen des ersten Experiments sortiert nach dem Auftreten 18
4.2 Längste Zugfolgen des ersten Experiments 18
4.3 Zugfolgen des zweiten Experiments sortiert nach dem Auftreten 19
4.4 Längste Zugfolgen des zweiten Experiments 20
4.5 Zugfolgen des dritten Experiments sortiert nach dem Auftreten 21
4.6 Längste Zugfolgen des dritten Experiments 21

6.3 Literaturverzeichnis

[1] Nikolaj Tatti and Jilles Vreeken, *The Long and the Short of It: Summarising Event Sequences with Serial Episodes*. Department of Mathematics and Computer Science, Universiteit Antwerpen.

[2] Victor-Philipp Negoescu, *Wissensgewinn aus Spieldatenbanken*. Fachbereich Informatik, Fachgebiet Knowledge Engineering, Technische Universität Darmstadt 2013.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und alle benutzten Quellen einschließlich der Quellen aus dem Internet und alle sonstigen Hilfsmittel angegeben habe.

Bischofsheim, 30. April 2014

Karsten Will