

TEIL 3

Optimierung

Kurzzinhalt

20. Ressourcenumlagerung	44
<i>Die Suche nach Optimierungsmöglichkeiten</i>	
21. Die Vielseitigkeit der 64 Bits	45
<i>Der Nutzen und die Funktionsweise von Bitboards</i>	
22. Zuggeneration Teil 1	48
<i>Das Erzeugen von Springer, Bauern- und Königszügen</i>	
23. Zuggeneration Teil 2	49
<i>Das Erzeugen von Turm-, Läufer- und Damenzügen</i>	
24. Sinnlos doppelt gemoppelt	53
<i>Grundlagen des Hashings</i>	
25. Der Hash-Schlüsselbund	55
<i>Die drei Anwendungen des Hashings</i>	
26. Ein letzter grosser Schliff	57
<i>Finale Optimierung durch Assembler-Code</i>	

20. Ressourcenumlagerung

Die Suche nach Optimierungsmöglichkeiten

Ziel In den ersten beiden Teilen haben wir uns die grundsätzlichen Überlegungen erarbeitet, mit welchen wir einem Computer mit Hilfe von Bewertungsregeln und Suchalgorithmen zu relativ intelligentem Schachspiel bewegen können. Nun gilt es, diese Verhaltensweisen in einer Programmiersprache möglichst effizient umzusetzen. Ziel ist es, den Suchbaum in möglichst hoher Geschwindigkeit zu durchsuchen, um in der gegebenen Zeit möglichst viele Stellungen bewerten zu können. Dabei soll sowohl die Bewertung selbst, als auch das Generieren neuer Züge möglichst zeitsparend durchgeführt werden.

pattern matching Betrachten wir kurz die Funktionsweise dieser beiden Funktionen, sehen wir, dass hier hauptsächlich bestimmte Muster und Strukturen auf dem Brett gesucht und interpretiert werden: In der Informatik nennt man dies *pattern matching*. Möchten wir beispielsweise als menschlicher Spieler überprüfen, ob sich auf der e-Linie eigene Bauern befinden, nehmen wir uns die 8 Felder dieser Linie heraus und untersuchen sie einzeln. Wir legen also gewissermassen eine virtuelle Schablone der e-Linie über das Schachbrett, welche nur die für uns interessanten Felder aufdeckt. Würden wir nun eine zweite Schablone darüberlegen, welche nur alle eigenen Bauern aufdeckt, wären als Resultat nur noch diejenigen Felder sichtbar, die auf der e-Linie sind und einen eigenen Bauern enthalten. Dazu müssten wir zwar stets eine ziemlich grosse Menge Schablonen zur Verfügung haben, von welchen sich einige im Laufe des Spiels verändern (z.b. diejenige, welche alle eigenen Bauern aufdeckt), dafür könnten wir uns das Untersuchen jedes einzelnen Feldes sparen. Dies bringt uns einerseits zum Prinzip der Bitboards (Kapitel 2), führt aber generell zu einer wichtigen Entscheidung:

Soll ein Schachprogramm lieber für möglichst geringen Speicherverbrauch oder für möglichst wenige Rechenoperationen optimiert sein?

Führen wir heute zehnjährige oder ältere Schachprogramme auf zeitgemässen Computern aus, bemerken wir, dass diese zwar nicht allzu schlecht spielen, gegen zeitgemässe Rechner jedoch nicht die Spur einer Chance haben. Achten wir beim Vergleich eines antiquierten Programms mit einem modernen Programm auf den Speicherverbrauch bzw. die Prozessorauslastung, stellen wir einen wichtigen Unterschied fest. Der Prozessor ist zwar bei beiden Engines voll ausgelastet, die Suche nach dem besten verfügbaren Zug ist also voll im Gange. Hingegen verwendet das moderne Programm gerne 200-300 MB Arbeitsspeicher ohne mit der Wimper zu zucken, während sich das ältere Programm immer bescheiden im Kilobyte-Bereich aufhält. Das modernere Programm nützt die Systemressourcen also wesentlich besser aus.

Dies zeigt, dass wir als Programmierer heutzutage mit dem Speicherplatz grosszügig umgehen dürfen, wann immer sich dadurch die Zahl der Rechenoperationen verringert.

21. Die Vielseitigkeit der 64 Bits

Der Nutzen und die Funktionsweise von Bitboards

Dieses und das folgende Kapitel wird sich mit dem Erkennen von Strukturen auf dem Brett mit Hilfe der im ersten Kapitel beschriebenen Schablonen-Methode beschäftigen. Eine sehr gute technische Implementierung dazu bieten sogenannte Bitboards. Dies sind einfache 64-bit-Variablen, welche in PHILEMON zu zehntausenden im Speicher bestehen. In ANSI C werden sie durch die Typenbezeichnung `long long` definiert, der Compilerhersteller Borland brachte die optimierte Variante `__int64` an den Tag. Im Gegensatz zum herkömmlichen Datentyp `long`, welcher 32 Bits umfasst, können wir in solchen Variablen jedem Feld des Schachbretts ein Bit (also einen Zustand 1 oder 0) zuordnen. Am einfachsten lässt sich der Verwendungszweck dieser Bitboards anhand eines Beispiels erklären: Wir möchten herausfinden, ob in der Stellung in Abbildung 9 irgend eine schwarze Figur durch den weissen Spieler angegriffen wird und ungedeckt ist.



Abb. 9:
Paul Morphy - Dominguez

Betrachten also zunächst eines der Bitboards, welches zum Zeitpunkt dieser Spielstellung im Speicherbereich PHILEMONS sein wird: das Bitboard aller schwarzen Figuren auf dem Brett (Abbildung 10). Eine Liste aller Bitboards der aktuellen Spielsituation, die ständig aktualisiert werden, ist in der Datei `<types.h>` in der SITUATION-Struktur zu finden.

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
1	1	0	1	1	0	0	1
1	0	1	1	1	1	0	0

Abb. 10:
Bitboard aller schwarzen Figuren auf dem Brett

Achtung: die Schachfelder haben in den Bitboards eine andere Anordnung als auf dem Brett. Auf dem Schachbrett aus der Sicht von Weiss finden wir beispielsweise das Feld a1 links unten, aus der Sicht von Schwarz aber rechts oben. Intern ist die Abbildung logischer, wenn die Felder in der Reihenfolge a1, a2, ... h7, h8 verteilt werden. Somit befindet sich das Feld a1 in diesen tabellarischen Darstellungen der Bitboards links oben, h8 rechts unten.

Feldanordnung

Schreiben wir das Bitboard aus Abbildung 10 als Binärwert in einer Zeile auf, lautet es folgendermassen:

00111101 10011011 01000100 00000000 00001000 00000000 00000000 00000000

Binärwert

Hier ist die Reihenfolge der Bits erneut umgekehrt, da in der Binärdarstellung die Bits in absteigender Wertfolge stehen: An der ersten Stelle steht das Bit für 263 (für uns das Feld h8), an der letzten Stelle dasjenige für 20 (für uns das Feld a1). Diese Art der binären Notation nennt man *Big Endian*.

Als gewöhnliche `long long` bzw. `__int64` Variable würden die 64 Bits dann als die Zahl 34'364'234'557 interpretiert, was für uns aber nicht weiter interessant ist. Schliesslich benötigen wir aus dem Bitboard nur die einzelnen Bits, nicht deren symbolischen Wert, interpretiert als Acht-Byte-Zahl.

Danach werden wir uns ein zweites Bitboard zu Gemüte führen: Dasjenige aller Felder, die von Weiss zum jetzigen Zeitpunkt angegriffen werden (Abbildung 11).

➡ Abb. 11:
Bitboard aller von
Weiss angegriffe-
nen Felder

1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	0	1	1	1	1	1
0	0	1	1	1	1	0	0
0	0	1	0	1	1	1	0
0	1	0	0	0	0	1	1

Dieses Bitboard existiert in der fertigen PHILEMON-Version zwar nicht mehr, war aber für lange Zeit Teil der Berechnung angegriffener Figuren. Es wurde bei jeder Bewegung einer weissen Figur neu erstellt. Notieren wir dessen Binärwert, sieht es so aus:

```
11000010 01110100 00111100 11111000
00111100 11111111 11110001 11111111
```

bitweises AND

Könnten wir nun diejenigen Felder bestimmen, welche in beiden Bitboards an ihrer Position ein gesetztes Bit (in der Binärdarstellung eine 1) besitzen, erhielten wir alle durch Weiss angegriffenen schwarzen Figuren. Diese Möglichkeit bietet uns die *bitweise AND-Operation*: Sie vergleicht hier zwei Variablen a und b auf Bit-Ebene miteinander und liefert einen Wert r zurück, in dem nur diejenigen Bits auf 1 gesetzt sind, welche auch in den beiden Variablen gesetzt sind:

```
typedef BITBOARD long long;
BITBOARD r = a&b;                                     // bitweises AND

a: 00111101 10011011 01000100 00000000 00001000 00000000 00000000 00000000
b: 11000010 01110100 00111100 11111000 00111100 11111111 11110001 11111111
-----
r: 00000000 00010000 00000100 00000000 00001000 00000000 00000000 00000000
```

➡ Abb. 12:
Bitboard aller von
Weiss angegrif-
fenen schwarzen
Figuren

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0

Aufgabenstellung herausfinden, welche dieser Figuren tatsächlich auch ungedeckt sind. Dazu bilden wir wieder die Schnittmenge der gesetzten Bits aus dem erhaltenen Bitboard sowie aus dem Bitboard aller Felder, die von Schwarz angegriffen werden. Nennen wir die Variable aller von Schwarz angegriffenen Felder s, führen wir also eine bitweise AND-Operation zwischen r und s durch. Danach erhalten wir folgendes Bitboard:

```
00000000 00000000 00000000 00000000 00001000 00000000 00000000 00000000
```

das Problem der
Auswertung

Wir haben unser gewünschtes Bitboard nun also erhalten, können es bisher aber erst als Integer-Wert anzeigen. Da uns dieser Wert (134'217'728) nicht viel hilft, müssen wir die gesetzten Bits innerhalb des Variable auswerten, um deren Position zu erhalten. In der Datei <bitops.h> finden wir dazu drei Funktionen:

Bitmasken

- MSB(b) - das *most significant bit*. Diese Funktion liefert uns dasjenige Bit des Bitboards b zurück, welches den geringsten symbolischen Wert besitzt. Dazu besitzt das Programm eine sogenannte Bitmaske mask64[] für jedes der 64 Bits. Diese Bitmasken sind Bitboards, in welchen jeweils nur ein bestimmtes Bit gesetzt ist. Bildet man per AND-Operation die Schnittmenge eines Bitboards und einer bestimmten Bitmaske, erhält man den Wert 0, falls das betreffende Bit im Bitboard nicht gesetzt ist. Für jedes Resultat grösser als 0 ist klar, dass das Bit gesetzt ist.

Die MSB-Funktion "schneidet" also das Bitboard `b` mit jeder der 64 Bitmasken in aufsteigender Reihenfolge, bis sie als Resultat einen Wert grösser als 0 erhält:

```
int MSB (BITBOARD b) {
    signed int sq;
    for (sq=0; sq<64 && !(b&mask64[sq]); sq++)
        ;
    return sq;
}
```

- `LSB(b)` - die *least significant bit*. Diese Funktion funktioniert gleich wie die MSB-Funktion, durchläuft die 64 Bitmasken allerdings in absteigender Reihenfolge:

```
int LSB (BITBOARD b) {
    signed int sq;
    for (sq=63; sq>=0 && !(b&mask64[sq]); sq--)
        ;
    return sq;
}
```

- `CntBits(b)` - die Anzahl der Bits in einem Bitboard. Bis kurz vor der Fertigstellung des Programms und dem Einbau der Assembler-Optimierung verwendete ich einen relativ simplen Algorithmus, um diese Bits auszuzählen: Ich verwendete die *shift-right-Operation (SHR)* des Prozessors, welche alle Bits innerhalb einer Variable um eine Stelle nach rechts schiebt. Das Bit an der nullten Stelle wird also "herausgeschoben" und verworfen, während an der hintersten Stelle ein neues 0-Bit eingesetzt wird. Die `CntBits()`-Funktion wendet diese Operation iterativ an und schneidet bei jedem Mal das verschobene Bitboard mit dem Binärwert der Zahl 1, um zu überprüfen, ob das Bit an der vordersten Stelle gesetzt ist. Falls dies der Fall ist, wird der Zähler `cnt` um eins erhöht. Sobald das Bitboard nach einer SHR-Operation den symbolischen Wert 0 besitzt, kann die Funktion abgebrochen werden, da keine Bits mehr zu finden sind:

SHR-Operation

```
int CntBits (BITBOARD b) {
    int cnt;
    for (cnt=0; b; b>>=1)
        if(b&1) cnt++;
    return cnt;
}
```

Möchten wir mehr als nur ein Bit untersuchen, übernehmen wir den Code der MSB/LSB-Funktionen und bauen ihn für den jeweiligen Gebrauch aus: Beispielsweise könnten wir nach jedem Wert grösser als 0 einen positionsabhängigen Punkteabzug für die jeweilige Seite verteilen und anschliessend das Bitboard nach weiteren Bits durchsuchen.

Untersuchung
mehrerer Bits

Intern werden 64-Bit-Variablen auf gängigen (32-Bit) Prozessoren in zwei `long`-Variablen umgewandelt, wodurch für die AND-Operation zwei Prozessorzyklen verwendet werden. Selbst wenn diese Bit-Operationen mehrmals nacheinander ausgeführt und die resultierenden Bitboards noch ausgewertet werden, kommen wir auf ein paar Dutzend CPU-Befehle. Für dasselbe Resultat hätten wir mit konventionellen Methoden (Arrays, welche in Schleifen durchsucht werden) hunderte oder sogar tausende von Befehlen verschwendet.

Fazit

Eine andere Erklärung der Bitboards befindet sich auf dem Internet auf [ChessPrg].

22. Zuggeneration Teil 1

Das Erzeugen von Springer, Bauern- und Königszügen

Die Erkenntnisse aus dem vorherigen Kapitel lassen sich nun auf die Funktion anwenden, welche am häufigsten ausgeführt wird: Die Erzeugung von möglichen Zügen. Hier unterscheiden wir zwischen den Figuren, welche sich unabhängig von der momentanen Brettsituation bewegen und denjenigen, welche sich nur auf freien Linien und Diagonalen bewegen können. Zur ersten Sorte gehören der Springer, der König und der Bauer.

Während in älteren Schachprogrammen die Generierung von Springerzügen teilweise ziemlich aufwändig umgesetzt ist, erstellen wir beim Start des Programms

einfach 64 Bitboards für jede mögliche Position des Springers. Begegnen wir im Spiel nun einem Springer, holen wir uns das passende Bitboard aus dem Speicher und erhalten sofort all seine Zugmöglichkeiten: Im gespeicherten Bitboards sind ausschliesslich die Bits aller Felder, welche der Springer angreift, auf 1 gesetzt. In PHILEMON wird in <init.c> ein Vektor von 64 Bitboards knight_attacks[] passend gefüllt und später bei der Generierung von Zügen verwendet. Ein Beispiel

für die Angriffsmöglichkeiten eines Springers auf dem Feld f4 bietet das Bitboard knight_attacks[29] in Abbildung 13 (Die Elemente dieser Vektoren sind stets in der Reihenfolge a1,a2,...,h7,h8 abgelegt).

Dieselbe Methode wenden wir auch für den König an: seine Angriffsfelder sind stets alle, die ihn direkt umgeben. Dabei spielt es hier noch keine Rolle, ob diese

Felder vom Gegner angegriffen werden, da wir nur pseudolegale Züge generieren: Sollte der König nach Durchführung einer seiner Möglichkeiten ins Schach geraten, wird die Suchfunktion in der ersten Suche-bene entdecken, dass der König als wertvollste Figur geschlagen wird und damit den Zug widerlegen. Um dies zu erreichen, erteilen wir dem König einfach den höchstmöglichen Materialwert, nämlich den Wert 32'767 (mit höheren Werten rechnet PHILEMON nicht).

Ein Beispielbitboard knight_attacks[62] für den König auf g8 sehen wir in Abbildung 14.

Bei den Bauern ist die Angelegenheit ein wenig schwieriger. Hier benötigen wir zwei verschiedene Bitboards pro Bauernposition: eines für potentielle schlagende Züge, eines für "ruhige" Züge. Das Bitboard für die schlagenden Züge besitzt seine gesetzten Bits in den beiden Feldern, die diagonal vor dem Bauern liegen. Dabei ist

➡ Abb. 13:
Angriffsbitboard
des Springers
auf f4

0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	0	0	1	0	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

➡ Abb. 14:
Angriffsbitboard
des Königs auf g8

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1
0	0	0	0	0	1	0	1

die Zuggeneration
der Bauern

der Begriff "vor dem Bauern" von der aktuellen Farbe abhängig, wir benötigen also erneut doppelt so viele Bitboards. Überlagern wir dieses Bitboard mit demjenigen aller gegnerischen Figuren auf dem Brett, erhalten wir ein neues Bitboard, welches alle Möglichkeiten für schlagende Züge enthält.

Um nun die gesamten Zugmöglichkeiten für einen Bauern zu erhalten, bilden wir die Vereinigungsmenge beider Bitboards: die *bitweise OR-Operation*. Diese untersucht zwei Variablen und liefert als Resultat ein Bitboard, in dem nur diejenigen Bits auf 1 gesetzt sind, welche auch in mindestens einer der beiden Variablen gesetzt sind. Als Beispiel sehen wir uns für a die schlagenden Züge, für b die ruhigen Züge eines weissen Bauern auf e4 an, der zwei schwarze Springer auf d5 und f5 angreift und auf das Feld e5 vorrücken könnte:

bitweises OR

```
typedef BITBOARD long long;
BITBOARD r = a|b;

a 00000000 00000000 00000000 00000000 00001000 00000000 00000000 00000000
b 00000000 00000000 00000000 00000000 00010100 00000000 00000000 00000000
-----
r 00000000 00000000 00000000 00000000 00011100 00000000 00000000 00000000
```

Spezielle Bauernzüge wie *en passant* und der Doppelschritt von der zweiten Linie aus werden separat behandelt, allerdings würde es nun den Rahmen dieser Dokumentation sprengen, dies ebenfalls zu beschreiben. Das Makro, welches im Spiel schlussendlich das Bitboard aller möglichen Züge des Bauern generiert, ist als TPawn(sq) in der Datei <macros.h> zu finden. Angriffszüge werden mit APawn(sq), ruhende Züge (inklusive Doppelschritt) mit NPawn(sq) generiert. Am selben Ort im Code sind auch sämtliche andere Zuggenerations-Routinen definiert.

Spezialzüge der Bauern

23. Zuggeneration Teil 2

Das Erzeugen von Turm-, Läufer- und Damenzügen

Bei der zweiten Figuresorte funktioniert diese Technik nicht mehr ganz so simpel: So hängen die Zugmöglichkeiten eines Turms vom momentanen Status der horizontalen Reihe sowie der vertikalen Linie ab. Ist das Feld f3 beispielsweise durch einen Bauern besetzt, kann ein Turm auf f5 das Feld f2 nicht erreichen. Befindet sich derselbe Bauer jedoch auf e3, ist der Weg auf f2 unter Umständen frei. Die Bewegungsfreiheit des Turms hängt also immer vom momentanen Besetzungszustand der aktuellen Linie bzw. Reihe ab. Es gibt also genau so viele unterschiedliche Szenarien für einen Turm, sich über eine Linie zu bewegen, wie es Möglichkeiten gibt, die Linie zu besetzen. Jedes Feld auf der Linie kann entweder besetzt oder nicht besetzt sein, es ergeben sich also $2^8=256$ verschiedene mögliche Zustände einer vertikalen Linie. Dasselbe gilt für eine Reihe (horizontal): Wir können eine solche ebenfalls auf 256 verschiedene Arten mit Figuren bestücken.

Abb. 15:
Sakharov - Rovner



Total existieren für den Turm also zweimal 256, also 512 verschiedene Szenarien, die bei der Generierung seiner Züge berücksichtigt werden müssen. Möchten wir jedes dieser Szenarien zu Beginn des Spiels vorberechnen und als Bitboards im Speicher ablegen, klingt dies zunächst völlig ineffizient. Rechnen wir aber mit 8 Bytes (64 Bits) pro Bitboard, ergeben sich gerade 256 KB, welche durch die Zuggenerierung der Türme besetzt würden. Im Vergleich zu den 200-300 MB Speicher, welche ein zeitgemässes Schachprogramm konsumiert, ist diese Menge jedoch verschwindend gering. Aus diesem Grunde werden wir dem in Kapitel 20 besprochenen Prinzip folgen und unseren Speicherplatz ausnutzen, da wir so eine ziemliche Menge an Prozessorleistung sparen.

Abb. 16:
Bitboard aller
Figuren auf dem
Brett

Sehen wir uns dies in Abbildung 15 anhand eines Beispiels an: der weisse Turm auf d6 möchte sich entlang der 6. Reihe bewegen. Dazu benötigen wir zunächst das Bitboard aller auf dem Feld vorhandenen Figuren (Abbildung 16; a1 befindet sich wiederum links oben, h8 rechts unten). Für uns ist nur die 6. Reihe interessant, wir müssen die Bits 01010010 der sechsten Reihe also in eine 8-Bit-Variable extrahieren. Dazu werden wir einfach so viele SHR-Operationen (bitweise Rechtsverschiebung, siehe Kapitel 21) wie nötig durchführen. In Falle des Bitboards aus Abbildung 16 müssen wir fünf Reihen, also 40 Felder, aus dem Brett schieben, um die Reihe 6 an die vorderste Position zu bringen. Generell schieben wir das Bitboard also immer $r * 8 - 1$ Bits nach rechts, wenn r für die besagte Reihe steht.

0	1	1	0	0	0	1	0
0	0	0	0	0	1	1	0
0	0	0	0	1	0	1	1
1	0	1	0	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	0	0	1	0
0	0	0	0	0	1	1	1
0	1	0	1	1	0	1	0

Binär ausgeschrieben sieht das resultierende Bitboard also so aus:

verschobene Bits
aller Figuren

00000000 00000000 00000000 00000000 00000000 01011010 00000111 01001010

Uns interessieren jedoch nur die vordersten 8 Bits, also benötigen wir die Zahl 255 (in dieser Zahl $2^8 - 1$ sind nur die acht vordersten Bits gesetzt), um eine Schnittmenge mit unserem Bitboard zu bilden:

Bitmaske 255

00000000 00000000 00000000 00000000 00000000 00000000 00000000 11111111

Binär ausgeschrieben sieht das Resultat danach so aus:

fertiger
Zustandscode

00000000 00000000 00000000 00000000 00000000 00000000 00000000 01001010

Nun können wir die restlichen 56 Bits verwerfen, indem wir das Bitboard in eine 8-Bit char Variable konvertieren. Diese können wir nun verwenden, um das an das richtige Angriffsbitboard zu gelangen, da dieses ja vom Zustand der aktuellen Reihe abhängt. Diese Angriffsbitboards über eine horizontale Reihe sind in PHILEMON im Array rank_attacks[][] gespeichert, wobei der erste Index für die Nummer des Feldes der betreffenden Figur steht, der zweite Index ist ein 8-Bit-Wert, der den «Zustandscode» der jeweiligen Reihe anzeigt. Da der symbolische Wert für die Binärfolge 01001010 die Zahl 74 ist und der Turm auf dem Feld Nr. 43 (d6) steht, ist das Bitboard aller für den Turm zugängigen Felder als rank_attacks[43][74] gespeichert (siehe Abbildung 17).

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	1	0	1	1	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Dieses Bitboard wird wie alle anderen vorberechneten Bitboards in der Datei `<init.c>` initialisiert. Diese Initialisierung möchte ich hier nicht weiter ausführen.

Möchten wir nun die für unseren Turm d6 zugänglichen Felder auf der d-Linie berechnen, benötigen wir diesmal den Zustandscode dieser Linie. Diesmal funktioniert der obige Trick nicht mehr, die passende Reihe einfach an den Beginn des Bitboards zu schieben: Wir müssen uns den 8-Bit-Schlüssel der aktuellen Linie mit Hilfe von Bitmasken (siehe Kapitel 21) selbst konstruieren. In `<macros.h>` finden wir ein Makro, welches dies erledigt:

```
#define LineState(line) ((unsigned char) ( \
    (mask64[ line] & AllFigures ? mask8[0] : 0) \
    | (mask64[ 8+line] & AllFigures ? mask8[1] : 0) \
    | (mask64[16+line] & AllFigures ? mask8[2] : 0) \
    | (mask64[24+line] & AllFigures ? mask8[3] : 0) \
    | (mask64[32+line] & AllFigures ? mask8[4] : 0) \
    | (mask64[40+line] & AllFigures ? mask8[5] : 0) \
    | (mask64[48+line] & AllFigures ? mask8[6] : 0) \
    | (mask64[56+line] & AllFigures ? mask8[7] : 0)) )
```

☞ Abb. 17:
Alle Felder der
6. Reihe, die hier
vom d6-Turm
angegriffen sind

C-Makro für die
Berechnung des
Zustandscodes

AllFigures ist hierbei das Bitboard aller Figuren, line ist die Nummer der vertikalen Linie der aktuellen Figur, mask8[] und mask64[] sind 8-Bit- bzw. 64-Bit-Masken. Die 64-Bit-Masken benötigen wir, um das Bitboard zu interpretieren, mit den 8-Bit-Masken erzeugen wir den Zustandscode. Demzufolge würde in der obigen Stellung der Makroaufruf LineState(3) den Wert 00000101 ergeben (die d-Linie hat die Nummer 3, da die Zählung der Linien bei 0 beginnt). Dieser Weg scheint zwar einfacher zu sein als derjenige, den wir bei den horizontalen Reihen verwendet haben, benötigt aber ungleich mehr Prozessorleistung. Nun erhalten wir mit Hilfe dieses Zustandscodes durch das Bitboard-Array line_attacks[][] das richtige Bitboard, welches für den d6-Turm die Zugmöglichkeiten entlang der d-Linie enthält. Der Binärcode 00000101 als Dezimalzahl hat den Wert 5, also ist das Bitboard unter line_attacks[43][5] zu finden. Bilden wir nun mittels bitweisem OR die Vereinigungsmenge von diesem Bitboard mit demjenigen, welches die horizontalen Zugmöglichkeiten enthält, erhalten wir das fertige Turm-Bitboard (Abbildung 18). Dieses wird im Spiel mit dem Makro TBook(sq) in der Datei `<macros.h>` auf die hier beschriebene Weise generiert.

0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0
0	1	1	0	1	1	1	0
0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0

☞ Abb. 18:
OR-Vereinigung
der Bitboards
rank_attacks[43][74]
und
line_attacks[43][5]

Die Generierung der Läuferzüge funktioniert auf dieselbe Weise, wie wir die Turmzüge entlang einer vertikalen Linie generieren: Wir überprüfen jedes einzelne Feld der Diagonalen, müssen dabei zusätzlich aber noch ihre Länge kennen. Dies war bei den Turmzügen nicht nötig, da jede Linie eine Länge von genau acht Feldern hat. Für Details möge man sich die Makros TBishop(sq), DiagLRState(sq) und DiagRLState(sq) in `<macros.h>` sowie alle damit in Verbindung stehenden Matrizen in `<matrices.h>` anschauen.

Anwendung
auf den Läufer

Anwendung
auf die Dame

Hat man die Angriffsfunktionen dieser beiden Figuren, erhält man das Zugmöglichkeiten-Bitboard der Dame quasi umsonst: Das Makro TQueen(sq) generiert nämlich schlicht per bitweisem OR die Schnittmenge der Turm- und Läufer-Angriffsfunktionen.

24. Sinnlos doppelt gemoppelt

Grundlagen des Hashings

Beispiel anhand
Endspiel und
Eröffnung

Stellen wir uns eine typische Turmendspiel-Situation vor, in der beide Spieler über Türme mit viel Freiheit verfügen und alles daran setzen, ihre Bauern zur gegnerischen Grundreihe zu führen. Da relativ wenige Figuren auf dem Brett vorhanden sind, werden wir mit der Suchfunktion relativ schnell in tiefere Ebenen des Suchbaums vordringen und alle Varianten untersuchen. Nun befinden sich unter den resultierenden Stellungen sehr oft Duplikate von Stellungen, die wir bereits untersucht haben. So entsteht in der Grundposition beispielsweise nach den Zügen 1. e4 d6 2. d4 dieselbe Stellung wie nach den Zügen 1. d4 d6 2. e4. Ideal wäre es, wenn das Programm diese Duplikate erkennen würde und die bereits erhaltenen Untersuchungsergebnisse für eine solche Stellung in die Suche einbeziehen würde. Den Ansatz dafür liefern sogenannte *transposition tables*, die wie die in der Informatik bekannten Hashtabellen funktionieren.

Definition • **Hashtabellen** (nach [Algorithm99], Kapitel 8)

Hashtabellen unterstützen eine der effizientesten Suchformen: das Hashing. Grundsätzlich besteht eine Hashtabelle aus einem Array, bei dem der Datenzugriff über einen speziellen Index, den sogenannten Schlüssel (Key), erfolgt. Die Hauptidee einer Hashtabelle besteht nun darin, über eine sogenannte Hashfunktion alle möglichen Schlüssel auf die entsprechenden Positionen im Array abzubilden. Eine Hashfunktion erwartet einen Schlüssel und gibt den Hashwert zurück.

Idee eines
gigantischen Arrays

Gelänge es uns also, einen Weg zu finden, jeder Spielstellung einen eindeutigen Schlüssel zuzuweisen, könnten wir unseren gesamten verfügbaren Speicher in ein gigantisches Array investieren. In diesem Array wären zu jeder indizierten Stellung gewisse Informationen wie beispielsweise der zuletzt erzielte Stellungswert gespeichert (siehe Kapitel 25). Die Lösung zur Berechnung eines eindeutigen Schlüssels für die momentane Stellung bilden sogenannte Zobrist-Schlüssel.

Zobrist-Schlüssel

Die Idee ist, jeder Stellung einen 64-Bit-Schlüssel zuzuweisen, wobei wir davon ausgehen, dass 64 Bits für eine eindeutige Bestimmung genügen. Dazu erzeugen wir beim Start des Programms zufällige 64-Bit-Werte und speichern diese in einem Array `zobrist_figure[][][]` (siehe dazu die Funktionen `Random64()` in `<tool.c>` und `InitZobristKeys()` in `<init.c>`). Dieses Array ist dreidimensional und verfügt über je eine Dimension für Farbe, Figur und Feld einer auf bestimmte Weise

positionierten Figur. So erhält ein weisser Springer auf b2 z.B. einen eigenen 64-Bit-Schlüssel. Danach erstellen wir je einen Schlüssel für jedes mögliche en-passant-Feld, je einen für jede mögliche Kombination von Rochaderechten und zuletzt einen, den wir *WhiteToMove* nennen. Können wir nun die für unsere Stellung zutreffenden Schlüssel auf geschickte Art und Weise miteinander kombinieren, hätten wir einen Schlüssel für die aktuelle Stellung. Dazu verwenden wir die bitweise XOR-Operation. Diese generiert aus zwei Variablen einen neuen Wert, bei dem nur jene Bits gesetzt sind, die entweder in der einen oder in der anderen Variable gesetzt sind, jedoch nicht bei beiden. Wenden wir diese Operation auf zwei zufällig gewählte Schlüssel an, sieht dies etwa so aus:

```
typedef BITBOARD long long;           // selber Datentyp wie in Kapitel 21
BITBOARD r = a^b;                    die XOR-Operation

a 11010101 10110001 10000101 11010101 11101001 10110101 00101101 11011110
b 10001111 10001010 10100111 10111100 10011001 11011001 11010100 01001001
-----
r 01011010 00111011 00100010 01101001 01110000 01101100 11111001 10010111
```

Dabei könnte a zum Beispiel der Schlüssel eines weissen Turms auf f7, b der Schlüssel einer schwarzen Dame auf a4 sein. Beginnen wir mit dem Wert 0 und lassen per bitweisem XOR einen relevanten 64-Bit-Schlüssel nach dem andern in unseren Stellungsschlüssel einfließen, erhalten wir zum Schluss einen Key, der das Bild der Stellung eindeutig repräsentiert. Nehmen wir noch den Schlüssel für das aktuelle en-passant-Feld, für beide Farben die Rochadeberechtigungs-Schlüssel und - falls Weiss am Zug ist - den *WhiteToMove*-Schlüssel hinzu, umfasst der Key alle nötigen Informationen der aktuellen Spielsituation.

Vervollständigung

Haben hier also zwei Spielsituationen zwei identische Keys, sind von hier aus die Zugmöglichkeiten exakt dieselben, die Bewertung der beiden Spielsituationen müsste also identisch sein. Da diese 64-Bit-Schlüssel auch nur dem Zufall unterliegen, besteht tatsächlich ein minimales Risiko, dass zwei Stellungen auf den selben Key abgebildet werden. Auf die Frage, wie man mit dieser Tatsache umgehen soll, antwortete Robert Hyatt, der Programmierer der weltbesten Open-Source-Engine Crafty im Usenet: *«We deal with it by ignoring it. If you use an array of 64x12 random numbers (first value=square #, second value=piece type) and all of these numbers are 64bit random values, when you exclusive or the set of values that corresponds to the current board position, the result is random enough and unique enough that you can <take a chance> that collisions won't occur. They will, of course, but so infrequently that they don't distort the search in any way. I trusted this at a million nodes per second on a Cray C90, Deep Blue trusts it at 300 million nodes per second, and we all trust it at 30-100K nodes per second on PC-class machines.»*

Hashkollisionen

Diese sogenannten Kollisionen treten also mit einer derart hohen Unwahrscheinlichkeit auf, dass sie die Suche nicht merklich stören. Das Risiko von Kollisionen steht in keinem Verhältnis zum Profit, den wir durch die Hashtabellen erhalten.

Der Vorteil dieser Hashkeys besteht darin, dass wir die oben beschriebene Prozedur nur bei der Initialstellung durchführen müssen. Da die XOR-Operation durch eine zweite Ausführung umkehrbar ist, können wir bei einer Änderung der Spielsituation

Vorteil der umkehrbaren Operation

den Schlüssel der bewegten Figur auf ihrem Ursprungsort per XOR wieder aus dem Stellungsschlüssel entfernen. Anschliessend fügen wir einen anderen Schlüssel per XOR ein: denjenigen der bewegten Figur an ihrem Zielort.

25. Der Hash-Schlüsselbund

Die drei Anwendungen des Hashings

Haben wir nun einen solchen Hashkey der aktuellen Stellung erzeugt, wenden wir diesen an drei verschiedenen Orten im Spiel an:

Die Hashtabelle

1. Zunächst speichern wir einige relevanten Daten zu dieser Stellung in unseren grossen Vektor `transposition_table`, dessen Grösse wir zu Beginn des Spiels per `malloc()` festlegen. Standardmässig ist diese Grösse in der Variable `hash_megabytes` in `<globals.h>` auf 32 MB festgelegt. Zu diesen relevanten Dingen zählen etwa die folgenden:

- der beste Zug in dieser Stellung
- die Art von Knoten im Suchbaum wir es zu tun haben: *fail-low*, *fail-high* oder *exakt* (siehe Kapitel 6)
- den Stellungswert, den wir bei der letzten Untersuchung erhalten haben
- die Tiefe der letzten Untersuchung; wurde der Stellungswert nämlich mit einer geringeren Tiefe erzielt, als wir nun untersuchen möchten, ist er für uns wertlos.

Abgespeicherte
Werte

Für all diese Informationen reichen insgesamt 8 Bytes, also 64 Bits: 30 für den besten Zug, 2 für die Knotenart, 16 für den Stellungswert und 16 für die Tiefe. Zusätzlich zu diesen 8 Bytes speichern wir noch den kompletten Stellungsschlüssel an der jeweiligen Position ab, um überprüfen zu können, ob es sich wirklich um die richtige Stellung handelt. Total benötigt ein Hash-Eintrag also 16 Bytes. In unserer Hashtabelle finden somit 32 MB geteilt durch 16 Bytes, also rund 2 Millionen Einträge Platz. Um nun den korrekten Platz des Hash-Eintrags innerhalb des Arrays zu finden, rechnen wir den 64 Bit grossen Stellungsschlüssel *modulo* die Anzahl der Einträge (standardmässig $2^{097'152}$). Auf diese Weise können wir an dieser Position sowohl einen neuen Hash-Eintrag erstellen als auch einen Hash-Eintrag auslesen. Schreiben wir einen neuen Hash-Eintrag an einen Ort der Tabelle, an dem sich bereits ein Eintrag befindet, wird dieser überschrieben. Je grösser die Hashtabelle ist, desto geringer ist dazu das Risiko.

Lesen wir einen Hash-Eintrag aus, vergleichen wir zunächst den aktuellen Stellungsschlüssel mit demjenigen, den wir sicherheitshalber in der Tabelle abgespeichert haben. Sind die beiden Schlüssel unterschiedlich, werden durch die modulo-Operation zufällig zwei Schlüssel an den selben Ort in der Tabelle abgebildet.

Die beiden Funktionen hierzu sind nach den *transposition tables* benannt, heissen `TTStoreEntry()` und `TTFetchEntry()` und befinden sich in der Datei `<hash.c>`.

2. Die nächste Anwendung ist das Remis durch dreifache Stellungswiederholung. Das Programm muss hier zunächst die Menge aller Stellungen seit dem letzten Zug, der nicht rückgängig gemacht werden kann (also ein Bauernzug oder ein beliebiger schlagender Zug), kennen. Danach muss überprüft werden, ob die aktuelle Stellung schon zweimal in besagter Menge vorkommt. Falls ja, erhält die Position sofort den Wert +0.00 für ein Remis und wird an diesem Knoten abgebrochen, ansonsten geht die Berechnung normal weiter. Die technische Umsetzung einer solchen Menge erfolgt über einen sogenannten Stack (siehe [Algorithm99] für Details). Für uns ist ein solcher Stack ein einfaches Array, in welches wir nach jedem Zug den Zobrist-Schlüssel der aktuellen Stellung ablegen. Befindet sich nun in einer Spielsituation eine gewisse Anzahl von Elementen auf diesem Stack, überprüft die DetectDraw()-Funktion in <tool.c>, ob der Zobrist-Schlüssel der aktuellen Stellung bereits zweimal auf dem Stack vorhanden ist. Dabei sucht die Funktion vom zuletzt gespielten Zug rückwärts und bricht ab, sobald der letzte irreversible Zug erreicht worden ist. Vor einem solchen Zug ist eine identische Stellung nicht mehr möglich. Die DoMove()-Funktion in <domove.c> muss also bei jedem Zug, der nicht rückgängig gemacht werden kann, den Zähler g->last_irreversible_move auf die momentane Zugnummer setzen. Remis durch
Stellungsrepetition
3. Die dritte Anwendung betrifft die Rückverfolgung der Hauptvariante, falls diese im Interface ausgegeben werden soll. Erinnern wir uns an die Suchfunktionen aus Teil I, entdecken wir, dass die Explore()-Funktion nur den Bestwert der Stellung zurückgibt. Sehen wir uns aber die Ausgabe von PHILEMON an, erhalten wir stets einen durchgerechneten Pfad der besten Variante. Intern sucht sich das Programm vor der Ausgabe der Hauptvariante aus der Hashtabelle die Stellung heraus und entdeckt darin den besten Zug. Auf dem virtuellen Schachbrett führt es diesen Zug nochmals kurz aus und sucht die resultierende Stellung abermals im Hashspeicher. Dieser Vorgang wird solange wiederholt, bis eine Stellung nicht mehr in der Hashtabelle gefunden wird, danach werden sämtliche Züge wieder rückgängig gemacht. In dieser Zeit hat sich das Programm in einem kleinen Array die Hauptvariante notiert und kann diese in der Folge ausgeben. Die Hauptvariante

26. Ein letzter grosser Schliff

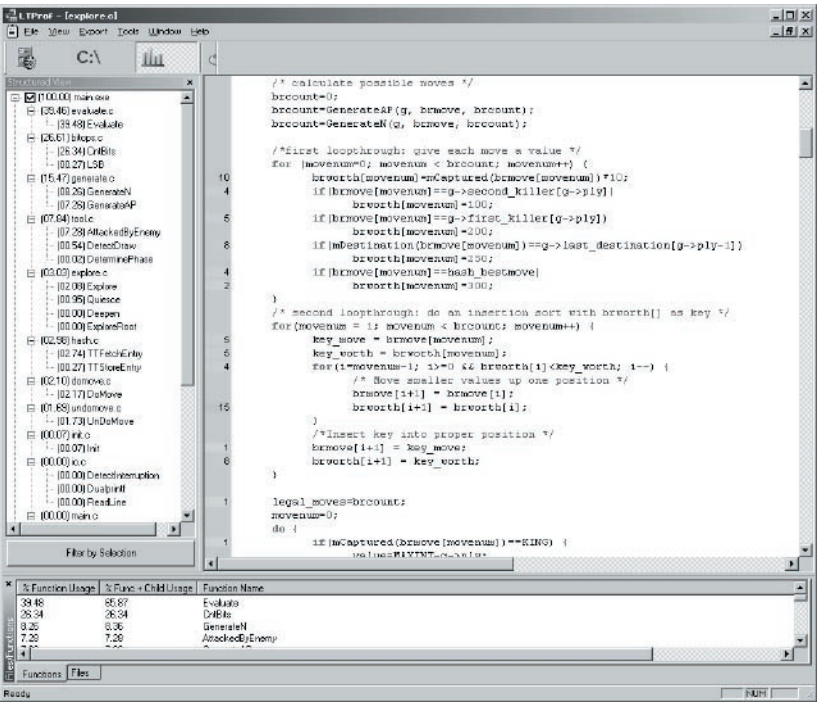
Finale Optimierung durch Assembler-Code

Kurz vor der Deadline der Wette gegen Lukas Messmer kam ich durch Usenet-Browsing auf die Idee, mittels eines sogenannten Profilers ausfindig zu machen, wieviel Zeit das Programm in den einzelnen Funktionen verbringt. Auf diese Weise würden sich unter Umständen Prozeduren zeigen, deren Geschwindigkeit noch optimierbar wäre. Dazu verwendete ich Borlands Linker ILINK32 mit der Option /v, um alle Debug-Informationen in die EXE-Datei zu linken und installierte anschliessend den Profiler LTPProf von <http://www.lw-tech.com/>. Dieses Tool startet eine beliebige ausführbare Datei mit Debug-Informationen, lässt sie eine bestimmte Zeit laufen und erstellt anschliessend eine Statistik, welche Zeile im Quellcode während der Aus-

Der Profiler

führung wieviel Zeit benötigt. Als ich PHILEMON mit Hilfe des PHRI-Kommandos .h («hint», siehe Anhang B) in den Analysemodus schaltete und den Profiler etwa eine Minute lang zuschauen liess, erhielt ich ein interessantes Resultat (siehe Abbildung 19, nächste Seite).

Abb. 19:
Analyse des
Profilers LTProf von
www.lw-tech.com



Die Statistik der wichtigsten Funktionen habe ich hier zusammengefasst.

zeitintensivste
Funktionen

Zeitanteil	Funktion
39.46%	Evaluate()
26.34%	CntBits()
8.26%	GenerateN()
7.28%	AttackedByEnemy()
7.26%	GenerateAP()
2.74%	TTFetchEntry()
2.17%	DoMove()
2.08%	Explore()
1.73%	UnDoMove()
0.95%	Quiesce()
0.54%	DetectDraw()
0.27%	TTStoreEntry()
0.27%	LSB()

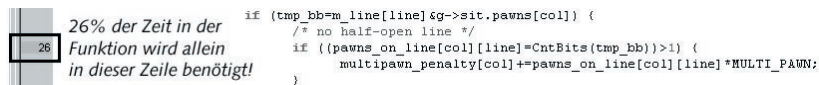
Die Prozentzahlen geben die Zeitanteile der gesamten Laufzeit des Programms an, die das Programm in der entsprechenden Funktion aufwendet.

Die Funktionen `Quiesce()` und `Explore()` rufen sich rekursiv auf, es gelten bei allen Prozentwerten daher nur jene Programmzeilen, in denen keine andere Funktion des Programms gestartet wird - letztlich geschehen ja beinahe alle Funktionen innerhalb der `Explore()`-Funktion. Offensichtlich benötigt die Stellungsbewertungsfunktion am meisten Zeit, da wir hier die komplexesten Aktionen auf dem Brett vollziehen. Dass sowohl `GenerateN()` als auch `GenerateAP()` einige Zeit benötigen, liegt ebenfalls auf der Hand. Jedoch konnten wir bei all diesen Funktionen durch den ausgiebigen Gebrauch von Bitboards die Prozentwerte im Verhältnis tief halten.

das Problem
der Rekursion

Wirklich bedenklich ist, dass das Programm über einen Viertel der gesamten Zeit in der Funktion `CntBits()` verbraucht, obwohl diese ja nur an wenigen Stellen verwendet wird. Am auffälligsten ist dies bei der Überprüfung von Doppelbauern in der `Evaluate()`-Funktion zu sehen: In dieser rund 350zeiligen Funktion (die in früheren Versionen mangels Optimierungen etwa doppelt so lang war) wird 26% der gesamten Zeit in einer einzigen Zeile verwendet (Abbildung 20).

ineffiziente
Auszählung



```
26% der Zeit in der Funktion wird allein in dieser Zeile benötigt!  
if (tmp_bb==m_line[line]&g->sit.pawns[col]) {  
    /* no half-open line */  
    if ((pawns_on_line[col][line]==CntBits(tmp_bb))>1) {  
        multipawn_penalty[col]+=pawns_on_line[col][line]*MULTI_PAWN;  
    }
```

Abb. 20:
Analyse der
ineffizienten Doppel-
bauern-Überprüfung

Die Ineffizienz meiner Auszählmethode zeigt sich leider allzu deutlich, somit machte ich mir Gedanken über eine neue, schnellere Funktion. Da diese recht einfach aufgebaut sein muss, wagte ich hier den Schritt, diese in der Maschinensprache Assembler zu codieren. Anschliessend sollte sie in das C-Projekt eingebunden werden, nachdem ich meinen Algorithmus revidiert hatte.

Der neue Algorithmus basiert auf der folgenden Idee: Subtrahiert man auf Bit-Ebene von einem beliebigen Wert die Zahl eins, wandelt man intern so lange alle 0-Bits zu 1-Bits um, bis man ein 1-Bit erreicht. Dazu beginnt man beim *least significant bit* (LSB), in unserer binären Schreibweise also demjenigen Bit, das ganz rechts steht. Danach arbeitet man sich nach links vorwärts. Sobald man auf ein 1-Bit stösst, wird dieses in ein 0-Bit umgewandelt und die Aktion wird abgebrochen.

Funktionsweise der
Subtraktion

Genau dieses Verhalten der Subtraktion können wir uns zu Nutze machen: Vergleichen wir auf Bit-Ebene nämlich den ursprünglichen Wert mit dem dekrementierten Wert, werden sich bis zur besagten Stelle immer nur im dekrementierten Wert gesetzte Bits befinden. Dort, wo wir schliesslich ein 1-Bit in ein 0-Bit umgewandelt haben, wird sich hingegen nur im ursprünglichen Wert ein gesetztes Bit befinden. Erstellen wir also mittels *bitweisem AND* die Schnittmenge der beiden Werte, erhalten wir bis und mit dieser Stelle alles 0-Bits. Bis auf dieses eine Bit, das wir somit eliminiert haben, hat sich im bearbeiteten Bereich nichts verändert. Demzufolge haben wir mit diesen beiden Operationen (dekrementieren und anschliessendes Bilden der Schnittmenge) einen Weg gefunden, jeweils das LSB zu löschen. Wiederholen wir dies nun solange, bis die Schnittmenge leer ist und inkrementieren bei jedem Durchlauf dazu noch eine Zählervariable, liefert uns dieser Zähler schlussendlich genau die Anzahl der gesetzten Bits im ursprünglichen Wert.

Anwendung als
Auszählmöglichkeit

Auf der nächsten Seite sehen wir hierzu ein Beispiel anhand der 16-Bit-Zahl 8738:

(a = Zählervariable, c = Ursprünglicher Wert, d = dekrementierter Wert, danach wird in c die Schnittmenge und zugleich der neue «ursprüngliche» Wert gespeichert)

```
Auszählung mit      a 0
Hilfe der neuen      c 00100010 00100010           // ursprünglicher Wert: 8738
Methode              d 00100010 00100001           // dekrementierter Wert: 8737

                    a 1
                    c 00100010 00100000           // Schnittmenge und neuer «ursprünglicher» Wert: 8736
                    d 00100010 00011111           // dekrementierter Wert: 8735

                    a 2
                    c 00100010 00000000           // Schnittmenge und neuer «ursprünglicher» Wert: 8704
                    d 00100001 11111111           // dekrementierter Wert: 8703

                    a 3
                    c 00100000 00000000           // Schnittmenge und neuer «ursprünglicher» Wert: 8192
                    d 00011111 11111111           // dekrementierter Wert: 8191

                    a 4
                    c 00000000 00000000           // Schnittmenge: 0 -> Abbruch
```

In diesem Beispiel sehen wir erstmals beim Wert 8736, dass durch die Dekrementierung alle Bits rechts des LSBs von 0-Bits in 1-Bits umgewandelt werden, während das LSB selbst auf 0 gesetzt wird.

zwei Arbeitsschritte

Möchten wir allerdings mit acht Byte grossen Bitboards hantieren, müssen wir in zwei Arbeitsschritten vorgehen: da wir (leider noch) mit 32-Bit-Prozessoren arbeiten, haben die internen Register jeweils nur eine Länge von 4 Bytes. Wir müssen also zunächst die ersten 4 Bytes in ein Register einlesen und überprüfen ob darin gesetzte Bits vorhanden sind. Fällt diese Überprüfung negativ aus, können wir gleich die zweiten 4 Bytes aus dem Speicher holen, andernfalls müssen wir das oben beschriebene Auszählverfahren anwenden. Die fertige Funktion sieht anschliessend folgendermassen aus:

```
fertige Assembler-   mov     ecx, dword ptr b           // erstes Einlesen
Routine              xor     eax, eax           // eax-Register löschen
                    test    ecx, ecx           // gesetzte Bits vorhanden?
                    jz      read_2           // falls ja, Teil 1 überspringen
                    count_1:
                    lea     edx, [ecx-1]      // von ecx 1 subtrahieren, Resultat in edx speichern
                    inc     eax               // Zähler eax um eins erhöhen
                    and     ecx, edx          // Schnittmenge von ecx und edx in ecx speichern
                    jnz     count_1          // wiederholen, bis die Schnittmenge leer ist
                    read_2:
                    mov     ecx, dword ptr b+4 // zweites Einlesen, 4 Bytes dahinter
                    test    ecx, ecx           // gesetzte Bits vorhanden?
                    jz      finish           // falls ja, Teil 2 überspringen
                    count_2:
```



```

lea    edx, [ecx-1]    // von ecx 1 subtrahieren, Resultat in edx speichern
inc    eax              // Zähler eax um eins erhöhen
and    ecx, edx        // Schnittmenge von ecx und edx in ecx speichern
jnz    count_2
finish:

```

Dieses Codefragment wird mit `__asm` in die Funktion `CntBits()` der Datei `<bitops.c>` eingebunden. Als Rückgabewert der Funktion wird automatisch der Wert in `eax` verwendet.

Einbindung in C

Als ich nun zum ersten Mal mit LTProf das Resultat dieser Optimierung untersuchte, war ich absolut sprachlos: Die Funktion `CntBits()` benötigte noch sage und schreibe 0.66% der ganzen Programmzeit, während sie vorher über 26% verbrauchte. Nachdem ich einige Stellungen im Analysemodus berechnen liess, erhielt ich die Bestätigung: die *NPS-Rate* (*nodes per second*) stieg um mehr als einen Viertel. Dies war weit über meinem persönlichen Erwartungswert, insbesondere nachdem ich Dutzende von Stunden damit verbrachte, durch kleine Verbesserungen wie den Einbau eines zweiten aspiration windows die Effizienz immer wieder um 1-2% zu steigern.

Fazit

Leider bin ich nur mit dem x86-Assembler einigermaßen vertraut und kenne die Befehlssätze der Macintosh-Prozessoren (PowerPC) überhaupt nicht. Aus diesem Grunde ist diese Assembler-Optimierung nur in der Windows-Version vorhanden, für alle andern Plattformen wird die alte, in Kapitel 21 beschriebene Methode verwendet.

Unterschiede
der Prozessoren

```

1  * COUNT BITS: This function is used to determine the number of '1' bits in a
2  * bitboard. Update 2004-01-06: improved with some x86 assembler code
3  *
4  *****/
5  int CntBits (BITBOARD b) {
6      #ifdef BCC32
7          __asm {
8              /* copy the first 4 bytes of the bitboard into ecx for further work */
9              mov     ecx, dword ptr b
10             /* empty eax */
11             xor     eax, eax
12             /* if ecx==0, the first 4 bytes seem to be already clear -> continue at read_2nd */
13             test    ecx, ecx
14             jz      read_2
15
16             /* subtract 1 from ecx (the second four bytes) and fold the result into edx */
17             count_1: lea     edx, [ecx-1]
18                     /* increase eax by 1 */
19                     inc     eax
20                     /* bitwise "and" with ecx and ecx-1 (edx) to remove the MSB */
21                     and     ecx, edx
22                     /* loop until ecx becomes zero */
23                     jnz     count_1
24
25             /* copy the second 4 bytes of the bitboard into ecx for further work */
26             read_2: mov     ecx, dword ptr b+4
27                     /* if ecx==0, the second 4 bytes seem to be already clear -> finish at 13 */
28                     test    ecx, ecx
29                     jz      finish
30
31             /* subtract 1 from ecx (the first four bytes) and fold the result into edx */
32             count_2: lea     edx, [ecx-1]
33                     /* increase eax by 1 */
34                     inc     eax
35                     /* bitwise "and" with ecx and ecx-1 (edx) to remove the MSB */
36                     and     ecx, edx
37                     /* loop until ecx becomes zero */
38                     jnz     count_2
39
40             finish:
41             }
42         #else
43             int cnt;
44             for (cnt=0; b>>=1; b>>1)
45                 if (b&1) cnt++;
46             return cnt;
47         #endif
48     }
49 }

```

➔ Abb. 21:
Neue `CntBits()`-
Funktion, analysiert
in LTProf

