Praktikum über effiziente Algorithmen

Abzählen von Sudoku-Gittern mit dem ,,dancing-links“-Algorithmus

FU-Berlin, Institut für Informatik

SS 06

Marco Kranz, Markus Luczak, Christopher Merkel

30. August 2006

Inhaltsverzeichnis

1 Einleitung

Ziel des Projektes war es, alle möglichen Sudoku-Gitter mit Hilfe des dancing-links Algorithmus'[1] (DLX) effizient abzuzählen.

Als erster Schritt erfolgte die Implementierung des DLX in Java mit der für Sudoku benötigten Codierung. Nachdem die Implementierung fehlerfrei arbeitete, ging es an die Optimierung. Trotz Optimierung war es unmöglich, ein 3x3-Feld in akzeptabler Zeit abzuzählen. Deshalb wurde durch Identifizierung von Äquivalenzklassen und entsprechender Ersparnisfaktoren die Eingabebelegung verkleinert.

Zum Abschluss wurde die Implementierung um Mehrprozessorfähigkeit erweitert.

2 Der dancing-links Algorithmus

Der dancing-links Algorithmus berechnet als nichtdeterministischer Tiefensuchalgorithmus alle Lösungen des Exact-Cover-Problems auf einer als zweifach-verkettete Liste codierten Binärmatrix.

Als exact cover bezeichnet man bei einer gegebenen Menge S von Mengen, die jeweils Elemente aus einem Universum U enthalten, eine Teilmenge S' mit der Eigenschaft, dass jedes Element aus U genau einmal in einer der Teilmengen auftaucht. Mit anderen Worten ist S' eine Untermenge von S, S eine Partition von U, die Mengen in S sind paarweise disjunkt und ihre Vereinigung ist U.

2.1 Datenstruktur

Die Datenstruktur der Eingabebelegung für den Algorithmus sieht so aus, dass eine zweifach-verkettete Knotenliste erstellt wird, wobei jedes vorhandene Listenobjekt eine Eins in der Matrix repräsentiert. Die Struktur der zweifach-verketteten Liste ermöglicht einfaches Löschen und Einfügen der Listenelemente an ihrer jeweiligen Position. Durch die Verwendung einer Kopfliste, auf deren Objekte alle Listenobjekte der Matrixspalten zeigen und die die jeweilige Anzahl von Einsen der Spalte speichern, kann auf das Anlegen von Objekten zur Repräsentation der Nullen verzichtet werden.

2.2 Funktionsweise

Der dancing-links Algorithmus beginnt seine Arbeit auf einer Eingabe der beschriebenen Datenstruktur durch Auswahl eines Kopflistenobjektes, also einer Spalte der Matrix, und rekursiver Löschung sämtlicher Knotenobjekte dieser Spalte aus der Belegung. Ferner werden alle Knotenobjekte gelöscht, die zu einem gelöschten Knotenobjekt der ausgewählten Spalte benachbart sind, also werden sogleich ganze Zeilen der Matrix gelöscht. Es wird darauf geachtet, dass bei jeder Löschung eines Objektes die Anzahl der vorhandenen Einsen im Kopflistenobjekt der zugehörigen Spalte aktualisiert wird.

Dieses sogenannte „covern“ wird solange durchgeführt, bis der Algorithmus erfolgreich terminiert oder in eine Sackgasse läuft. Erfolgreich terminieren bedeutet hier, dass der Algorithmus alle Spalten abarbeitet und erst am Ende die Anzahl der Einsen in allen Kopflistenelementen null ist. Speichert man sich den Auswahlpfad durch die Spalten, so erhält man eine Lösung für ein exact-cover auf der Eingabebelegung. Dementgegen steht eine Sackgasse als Identifikation einer falschen Auswahl für ein exact-cover, wenn während der „cover“-Rekursionen eine Spalte angesprochen wird, die bereits keine Einsen mehr gespeichert hat. In diesem Fall greift das Backtracking ebenfalls rekursiv und es wird der nächstmögliche Pfad getestet.

[1] `http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz`

Im Falle des zuletzt geschilderten Backtrackings greift zunächst das rekursive „uncovern“ der zuletzt „gecoverten“ Knotenobjekte, bevor mit der Auswahl einer anderen Spalte fortgefahren wird.

3 DLX-Matrix des Sudoku-Feldes

Um die Übertragung der DLX-Eingabematrix Sudokubelegungen zu verstehen betrachten wir zunächst eine einfache Eingabebelegung des DLX zur Ermittlung eines Exact-Covers.[2]

Sei $U = \{1, 2, 3, 4, 5, 6, 7\}$ ein Universum von sieben Elementen und sei $S = \{A, B, C, D, E, F\}$ eine Menge von sechs Mengen.

- $A = \{1, 4, 7\}$
- $B = \{1, 4\}$
- $C = \{4, 5, 7\}$
- $D = \{3, 5, 6\}$
- $E = \{2, 3, 6, 7\}$
- $F = \{2, 7\}$

Dann ist eine Untermenge $S' = \{B, D, F\}$ ein Exact-Cover, wenn jedes Element aus U genau einmal in einer der Mengen $B = \{1, 4\}$, $D = \{3, 5, 6\}$ oder $F = \{2, 7\}$ vorkommt.

Sehen wir uns nun die $6 * 7$ Matrix an, die die sechs Mengen aus S und die sieben Elemente aus U repräsentiert.

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

Die Auswahl S' stellt sich als Exact-Cover dann wie folgt dar:

	1	2	3	4	5	6	7
B	1	0	0	1	0	0	0
D	0	0	1	0	1	1	0
F	0	1	0	0	0	0	1

Dieses Beispiel und die Erläuterungen zur Funktionsweise des Algorithmus sowie der Datenstruktur der Eingabebelegung führen uns zur Kodierung einer Sudokufeldbelegung als Matrix.

Wenn auch dieses Problem durch eine Matrix von Nullen und Einsen repräsentiert werden soll, muss man sich veranschaulichen, welche Parameter ein Sudokufeld mitbringt. Zunächst wären da die Anzahl der Zeilen m und Spalten n je Block des Sudokufeldes.

[2] `http://en.wikipedia.org/wiki/Exact_cover`

Diese definieren implizit den Zahlenraum $Z = \{1, 2, ..., m * n\}$ in dem sich das Problem des Ausfüllens des Feldes bewegt. Beim Lösen eines Sudokus gilt es jedoch außer der Einzigartigkeit jeder Zahl pro Zeile und Spalte zusätzlich die Einzigartigkeit jeder Zahl pro $m*n$-Box, von denen es genau $b = m*n$ Stück gibt, zu beachten. Für unsere Repräsentation haben wir außerdem noch den Parameter der $f = (m * n)^2$ Kästchen festgehalten.

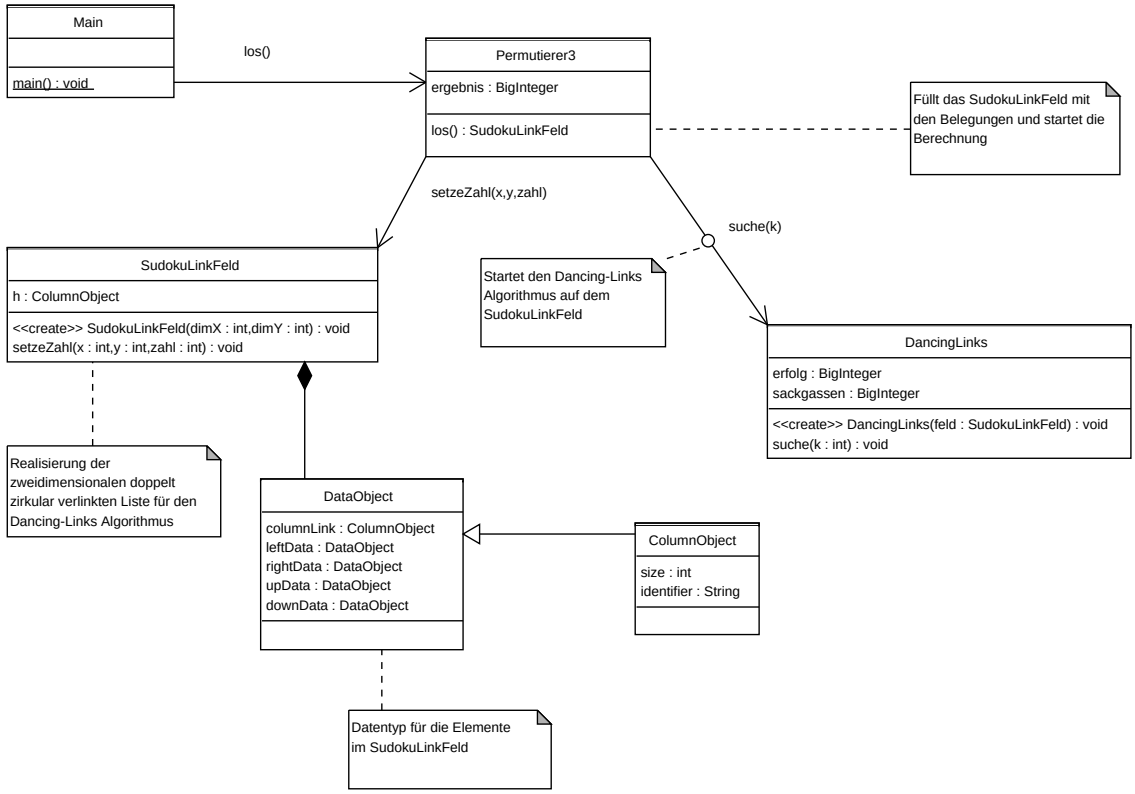
Der erste Block in der Matrix weist einer Zahl im Sodokufeld eine Position nach Zeile und Spalte zu. Im Folgenden Block wird festgehalten, welche Zahl in welcher Zeile steht. Der dritte Block legt selbiges für die Spalte fest. Im vierten und letzten Block wird festgehalten, in welcher Box sich die Zahl befindet.

Fasst man diese durch Bitvektoren dargestellten Teilaussagen zusammen, so kann man durch einen einzigen Bitvektor der Länge $4 * f$ mit insgesamt vier Einsen und $f - 4$ Nullen eine eindeutige Aussage für die Positionierung einer bestimmten Zahl in einem bestimmten Kästchen eines Sudokufeldes treffen. Eine komplette Eingabebelegung für ein Sudokufeld würde nun genau $m * n$ Bitvektoren der o.g. Struktur umfassen, da davon ausgegangen werden darf, dass für eine gültige Belegung jedes Kästchen ausgefüllt sein muss.

Unsere Problemstellung befasste sich mit der Zählung aller möglichen validen Belegungen für Sudokufelder beliebiger Größe $m * n$. Daher ist hierfür noch zu beachten, dass sich die Eingabematrix bezüglich der Anzahl der Zeilen um den Faktor $m * n$ vergrößert, da wir sämtliche Permutationen der Gestalt betrachten, dass jede Zahl letztendlich einmal in jedem Kästchen gestanden haben muss.

4 Implementierung

Für die Implementierung haben wir Java benutzt. Die Struktur der Klassen ist im folgenden Diagramm dargestellt, wobei in den Klassen nur die wichtigsten Funktionen und Variablen dargestellt sind.



Nach dem Start durch die Main-Klasse erstellt die Klasse *Permutierer* zuerst ein *SudokuLinkFeld* und füllt es mit den Zahlen im Sudokugitter. Die *setzeZahl*-Funktion hat uns bei der Implementierung die meisten Schwierigkeiten bereitet, da sie die eingefügten Zahlen für verschiedendimensionale Felder an die richtigen Positionen setzen musste. Wegen der Größe der erstellten Matrix mussten wir uns noch eine eigene Ausgabefunktion erstellen, die die Tabelle in Swing dargestellt hat. Für die gängigen Tabellenkalkulationen enthielt die Matrix zu viele Spalten. Diese Ausgabefunktion haben wir dann für die Fehlersuche verwendet. Vom Permutierer hatten wir mehrere Varianten erstellt, in denen wir jeweils die weiter unten beschriebenen Optimierungen vorgenommen haben.

Das *SudokuLinkFeld* ist die Repräsentation der vom Dancing-Links Algorithmus benötigten Datenstruktur aus einer zweidimensionalen zirkular doppelt verketteten Liste. Über eine eigene Headerliste kann man in der Datenstruktur leicht auf die einzelnen Spalten zugreifen. Für die Repräsentation der Einsen in der Datenstruktur haben wir einen eigenen Datentyp namens *DataObject* verwendet, in dem wir die Links auf die benachbarten Einsen links, rechts, oben und unten speichern. Die Headerliste besteht aus einem von DataObject abgeleiteten Datentyp namens *ColumnObject*, in dem zusätzlich noch die Anzahl der Einsen in einer Spalte und der Name der Spalte gespeichert sind.

Wenn im Feld alle benötogten Zahlen gesetzt wurden, wird der Dancing-Links Algorithmus mit der *suche*-Funktion aus der Klasse *DancingLinks* aufgerufen. Die vom Algorithmus gefundenen Anzahlen an Lösungen und Sackgassen werden vom Permutierer ausgelesen und ausgegeben. In der naiven Implementierung wird der Algorithmus einfach auf ein mit allen möglichen Belegungen gefülltes Feld angesetzt, wobei nur ein Endergebnis anfällt. In den optimierten Versionen wird der Algorithmus auf Teilmengen der möglichen

Belegungen angesetzt und der Permutierer muss sich die Zwischenergebnisse merken und das Gesamtergebnis berechnen.

5 Optimierungen

Nachdem der Algorithmus problemlos funktionierte, war die Geschwindigkeit eher ernüchternd. Leider schaffte das Programm nur etwa 20-40 Ergebnisse pro Sekunde zu berechnen. Deshalb lag unsere oberste Priorität darin, die Leistungsprobleme in den Griff zu bekommen.

5.1 Codeoptimierungen

Als erster Fehler zeigte sich, dass der verwendete Vektor, in dem die Elemente während der Rekursion gespeichert wurden, nicht wie erwartet geleert worden ist. Außerdem änderten wir den Datentyp auf ein Feld (Array). Allein diese Änderungen brachte eine Verbesserung von Ergebnissen pro Sekunde auf Ergebnisse pro Millisekunde.

Über Nacht ließen wir das Programm laufen und stellten am nächsten Tag fest, dass unser Zahlenbereich übergelaufen ist. Der Datentyp wurde auf BigInteger geändert, brachte aber Einbußen von bis zu 30% Leistung mit sich. Nachdem die Anzahl der neu erzeugten BigInteger verbessert, wir in den den Schleifen normale Integer verwendeten, diese am Ende des Zahlenbereichs leerten und in das BigInteger schrieben, stellte BigInteger keine Verschlechterung der Laufzeit dar.

5.2 Eingabeoptimierung

Für den ersten Block gibt es 9! mögliche Belegungen. Es gibt damit nur noch 10 erlaubte Belegungen für die letzten sechs Stellen der ersten Zeile. Das gleiche gilt für die letzten sechs Stellen der ersten Spalte. Da man sowohl in der ersten Zeile als auch in der ersten Spalte normaler Weise 6! mögliche Belegungen hätte, erreicht man so einen Ersparnisfaktor von jeweils 72. Durch die Vorgaben bei den Belegungen lassen sich die Möglichkeiten, die ersten drei Reihen zu füllen, nun in 71 Äquivalenzklassen einteilen. Jede dieser Klassen enthält eine unterschiedliche Anzahl möglicher Belegungen, die dann in Form von Faktoren in das jeweilige Zwischenergebnis einfließen. Den unbelegten Rest des Sudoku-Gitters haben wir mit allen möglichen Kombinationen belegt und haben dann die 71 Belegungen der ersten drei Reihen mal den 10 Belegungen der ersten Spalte (710 Möglichkeiten) mit dem DLX berechnet. Die Faktoren für die 71 Äquivalenzklassen waren bereits bekannt.

Durch die Vorgaben erreicht man insgesamt einen Ersparnisfaktor von $9! \cdot 72^2$. Dank dessen konnten wir ein 3x3-Feld in akzeptabler Zeit (7,6 Stunden) abzählen.

5.3 Multiprozessorfähigkeit

Durch die Eingabeoptimierung wurde das Problem in mehrere Teilprobleme zerteilt. Diese Teilprobleme ließen sich unabhängig voneinander berechnen und lösen. So lag als weitere Optimierung die Mehrprozessorfähigkeit auf der Hand, die durch die Unäbhängigkeit und Parallelisierung relativ einfach zu implementieren wäre.

Bedingt durch die Eingabebelegung haben wir 710 mögliche Durchläufe. Theoretisch hätten wir so bis zu 710 Threads starten können. Am Fachbereich standen uns jedoch Rechner mit maximal 4 Prozessoren zur Verfügung, weshalb wir uns entschieden nur 10 Threads zu starten. Beim 3x3 Feld werden die oberen 3 Boxen fest belegt und jeder einzelne Thread berechnet dann dazu mit der ersten festgelegten Spalte das Ergebnis. Dann wird gewartet bis alle zehn Threads fertig sind und am Ende das jeweilige Ergebnis mit dem Faktor multipliziert und auf das Gesamtergebnis addiert. Da die einzelnen Threads von

der Größenordnung der Ergebnisse sich kaum unterscheiden, stellt das Warten auf die restlichen Threads nur eine sehr geringe Leistungseinbuße dar.

Auf unserem 64-Bit Dual Opteron System konnte eine 1,46-fache Beschleunigung mittels Mehrprozessorfähigkeit erreicht werden.

5.4 Weitere Tests

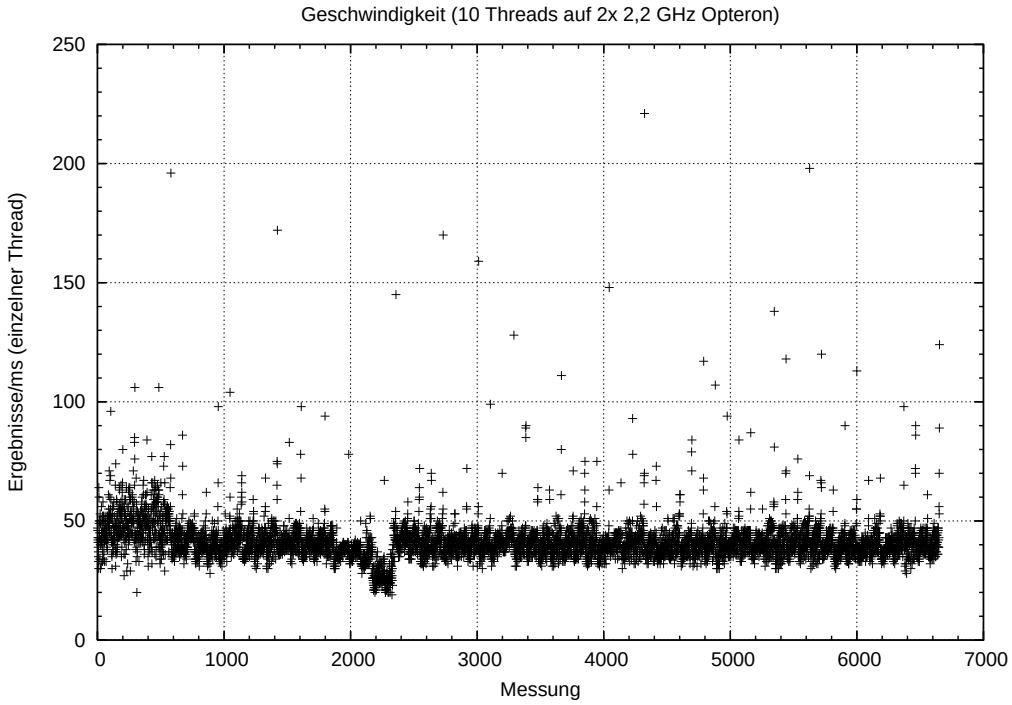
Neben dem Opteron System testeten wir unser Programm auch auf diversen anderen Rechnern. Bemerkenswert fanden wir die Leistung des Systems auch ohne Mehrprozessorfähigkeit, weshalb wir uns auf die Suche nach der Ursache machten. Ein vergleichbares Dual Xeon System brachte nicht annähernd die Anzahl an Ergebnissen in der Millisekunde. Auf dem Opteron System hatten wir nicht die Rechte ein anderes Betriebssystem zu installieren, um Vergleiche ziehen zu können.

Glücklicherweise stand uns kurzzeitig ein Core 2 Duo 2,16 GHz zur Verfügung auf dem wir Knoppix in der 32-Bit Version und Kanotix, ein Knoppixableger, in der 64-Bit Version starteten. Beide Betriebssysteme nutzen die Version 5.0 der Java Virtual Machine (JVM).

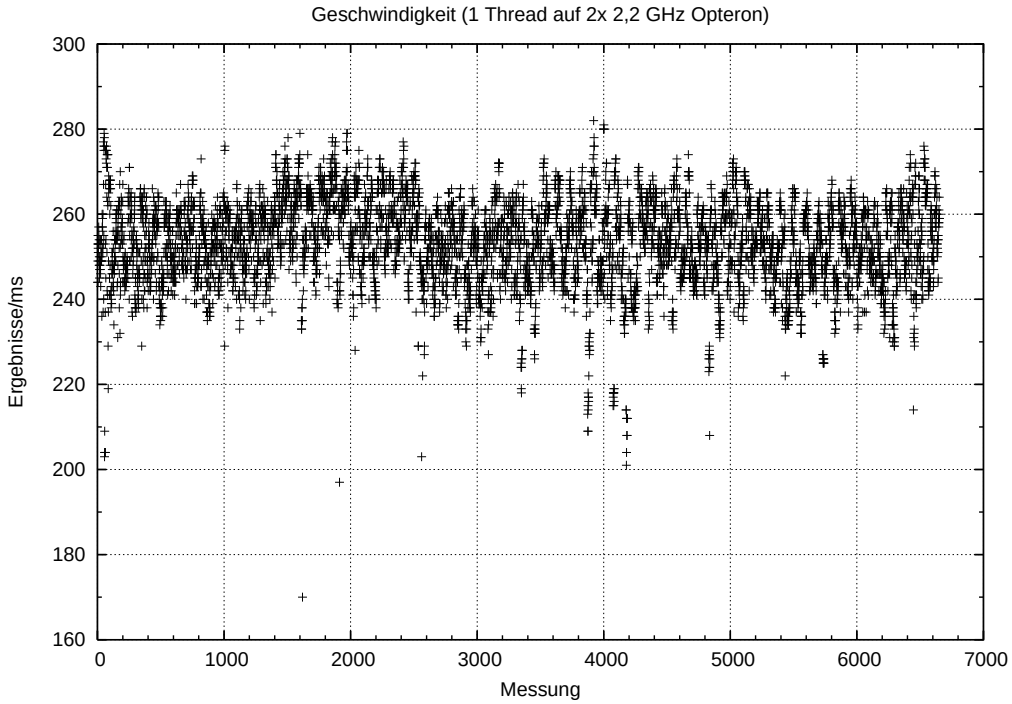
Der direkte Vergleich belegte die Vermutung, dass unser Programm wesentlich schneller mit der 64-Bit Version arbeitete. Ein kurzer Test zeigte beinahe die doppelte Leistung. Die meisten Operationen des Programms betreffen das Umbiegen der verketteten Listen. Möglicherweise kann dort die AMD64 Architektur mit ihren acht zusätzlichen Registern die Leistung steigern.

6 Statistiken für die Berechnung eines 3x3-Feldes

Rechner	2x 2,2 GHz Opteron
Ersparnisfaktor	$9! \cdot 72^2$
Gefundene Lösungen	3.546.146.300.288
Gesamtergebnis	6.670.903.752.021.072.936.960
Sackgassen	2.391.550
Dauer mit 10 Threads	18843 s $\approx$ 5,2 h
Geschwindigkeit (10 Threads)	41,39 Lösungen/ms (pro Thread)
Dauer mit 1 Thread	27454 s $\approx$ 7,6 h
Geschwindigkeit (1 Thread)	257,18 Lösungen/ms

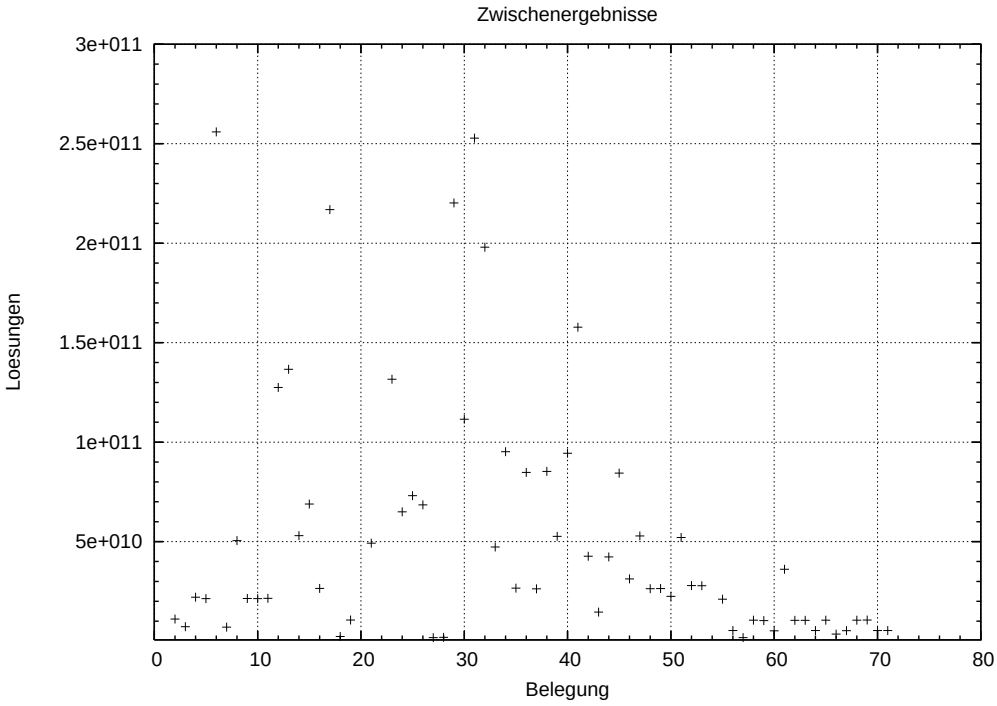


Die obige Grafik zeigt, wieviele Ergebnisse pro Millisekunde von den einzelnen Threads erreicht wurden. Die Geschwindigkeit wurde gemessen, indem die Zeit für das Finden von einer Million Lösungen für jeden einzelnen Thread ermittelt wurde. Man sieht, dass die meisten Threads eine ähnliche Geschwindigkeit erreichten. Die Ausreißer lassen sich dadurch erklären, dass wir immer auf die Fertigstellung aller Threads gewartet haben, bis neue Threads gestartet wurden. Einige Threads wurden deswegen alleine und damit schneller abgearbeitet als andere.

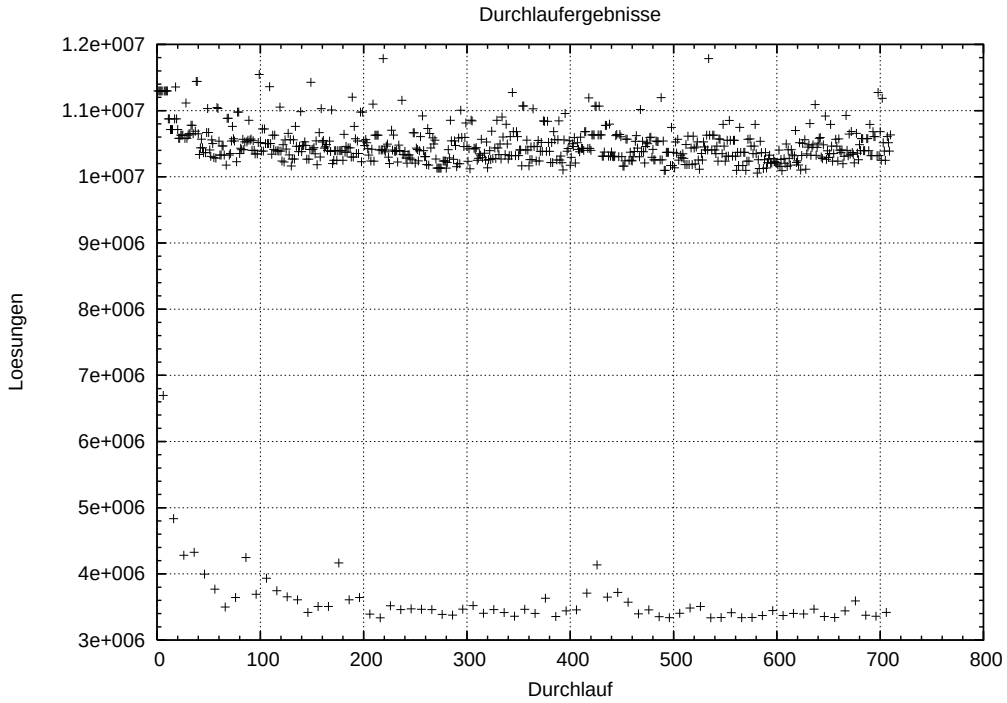


Hier sieht man die Geschwindigkeit des Programms, wenn nur ein einzelner Thread

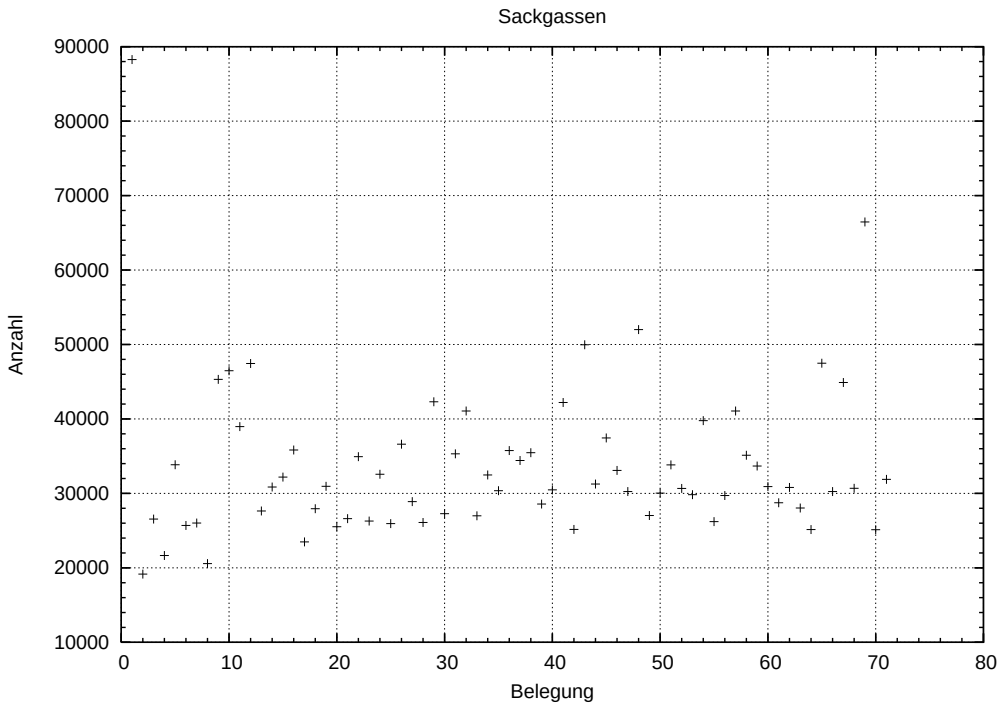
benutzt wurde. Logischer Weise erreicht ein einzelner Thread mehr Ergebnisse pro Millisekunde als einer von 10 Threads. Die Geschwindigkeiten der einzelnen Messpunkte sind wieder recht konstant. Die Abweichungen lassen sich mit wechselnden Belastungen des Systems erklären, dass parallel noch von anderen Benutzern verwendet wurde. Die Ermittlung der Geschwindigkeiten erfolgte auf die gleiche Weise wie bei der Grafik für 10 Threads.



Diese Grafik zeigt die Verteilung der für die einzelnen Belegungen für die ersten drei Reihen gefundenen Lösungen. Es sind zwar Unterschiede zu erkennen, die Mengen bewegen sich aber alle in der selben Größenordnung. Es ist allerdings deutlich zu sehen, dass einige Belegungen günstiger zu sein scheinen als andere. Die Faktoren für die Äquivalenzklassen sind hier bereits berücksichtigt.



Hier wird das gleiche nochmal für jeden einzelnen Durchlauf, also für jede Belegung der ersten drei Reihen und für jede Belegung der ersten Spalte gezeigt. Auffällig sind die 71 Ausreißer nach unten. Es scheint also eine Belegung für die letzten sechs Stellen der ersten Spalte zu geben, die weniger günstig ist als die anderen. Da jede dieser Belegungen für 71 Belegungen der ersten drei Reihen verwendet wird, entstehen 71 Ausreißer.



Anhand dieser Grafik kann man sehen, dass die Anzahl der Sackgassen für die einzelnen Belegungen der ersten drei Reihen deutlich schwankt. Wie bereits aus der Grafik zu den Zwischenergebnissen zu sehen war, kann man auch hier darauf schließen, dass es günstige und ungünstige Arten der Belegung für die ersten drei Reihen gibt.

7 Fazit

Bei der Implementierung gestaltete sich die Umsetzung der abstrakten Datenstruktur am schwierigsten. Nachdem wir das bewältigt hatten, hatten wir zu Beginn des Praktikums Zweifel, ob der Algorithmus jemals schnell genug arbeiten würde, um größere Felder als 2x2 berechnen zu können. Erst durch die Optimierungen der Programmierung konnten meherere tausend Ergebnisse pro Sekunde berechnet werden. Die Berechnung von 3x3 Feldern wurde allerdings erst möglich, nachdem auch noch die Eingabe optimiert wurde. Danach konnten wir ein solches Feld in etwa 8 Stunden berechnen. An Berechnungsservern standen uns durchweg Maschinen mit mehr als einem Prozessor zur Verfügung. So lag es auf der Hand, diese auch auszunutzen und unseren Algorithmus mehrprozessorfähig zu machen. Im Endeffekt berechnen wir ein 3x3 Feld nun in etwa 5 Stunden auf einem Dual-Opteron mit jeweils 2,2 GHz. Überraschend war auch, dass unser Programm stark von 64-Bit-Prozessoren profitieren konnte.

8 Literatur

1. Donald E. Knuth: Dancing Links
   `http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz`
2. Wikipedia: Knuth's Algorithm X
   `http://en.wikipedia.org/wiki/Algorithm_X`
3. Wikipedia: Dancing Links
   `http://en.wikipedia.org/wiki/Dancing_links`
4. Wikipedia: Exact cover
   `http://en.wikipedia.org/wiki/Exact_cover`