Technische Universität München

Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Research Group Prof. Grust

Proseminar: Selected Fun Problems of the ACM Programming Contest
Wintersemester 2006/2007

Thema: Mastermind (Central European Regionals 2000)
Betreuer: Prof. Dr. Torsten Grust, Jan Rittinger, Jens Teubner
Bearbeitung: Elmar Steinkasserer

Februar 2007

1 Einleitung

Die 1947 gegründete „Association for Computing Machinery“ (ACM) ist die älteste und größte Informatikervereinigung der Welt. Der Organisation gehören weltweit insgesamt ca. 80.000 Mitglieder an, die in über einhundert Ländern tätig sind. Der Hauptsitz der ACM befindet sich in New York.

Ziel der Organisation ist es die „Kunst“, Wissenschaft und Anwendung der Informationstechnologie zu fördern („advancing the arts, sciences, and applications of information technology“). Zu diesem Zweck werden verschiedene Veranstaltungen organisiert, sowie diverse Zeitschriften und Publikationen veröffentlicht.

Eine dieser Veranstaltungen ist der seit 1970 alljährlich stattfindende ACM International Collegiate Programming Contest, die „Weltmeisterschaft im Programmieren“. Der Wettbewerb wird in insgesamt drei Runden (Local Contests, Regional Contests, World Finals) ausgetragen. [WikiACM]

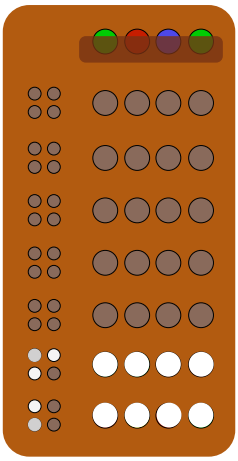
Im Zuge des Proseminars wird jeder Teilnehmer mit einer ausgewählten Problemstellung des ACM Programming Contest konfrontiert. Diese sowie die in einer gewählten Programmiersprache zu erarbeitende Lösung werden im Zuge des Seminarvortrags präsentiert. Abschließend wird eine Ausarbeitung der Lösung erstellt.

2 Das Spielprinzip von Mastermind

Bei Mastermind handelt es sich um ein Logikspiel mit zwei Spielern. Der erste Spieler bestimmt hierbei eine geheime vierstellige Farbkombination, wobei insgesamt sechs Farben zur Verfügung stehen, jedoch jede Farbe beliebig oft verwendet werden darf. Der zweite Spieler versucht mit der vorgegebenen Anzahl an Versuchen diesen Farbcode zu erraten. Nach jedem Rateversuch erhält dieser vom ersten Spieler Informationen darüber inwieweit sein Rateversuch der gesuchten Farbkombination entspricht. Dies geschieht in Form von schwarzen und weißen Punkten. Ein schwarzer Punkt bedeutet „Farbe richtig, Position richtig“ und ein weißer Punkt „Farbe richtig, Position falsch“. Ist die zu erratende Farbkombination beispielsweise „grün, rot, blau, grün“ und der Rateversuch „grün, grün, orange, rot“, so erhält der Spieler einen schwarzen (für grün an der richtigen Position) und zwei weiße Punkte (für grün an der zweiten und rot an der letzten Stelle; da „Farbe richtig, Position falsch“ gilt; siehe [Abbildung 1]). Bei vier schwarzen Punkten hat der Spieler demnach die richtige Farbkombination vollständig erraten. Errät der Spieler diese nicht mit der vorgegebenen Anzahl an Versuchen, so hat er das Spiel verloren.

Wie in [WikiMastermind] nachzulesen ist, entwickelte Donald E. Knuth 1976 einen Algorithmus, welcher eine beliebige Farbkombination der oben beschriebenen Form in maximal fünf Versuchen ermittelt.

Der Algorithmus benötigt im Durchschnitt $\frac{5626}{1296} = 4,341$ Versuche. Knuth führte auch den Beweis, dass jeder Algorithmus der mit maximal fünf Versuchen die korrekte Farbkombination bestimmt, im Durchschnitt mindestens 4,341 Versuche benötigt.



[Abbildung 1]:
Spielsituation nach 2 Versuchen

Kenji Koyama und Tony W. Lai fanden 1994 einen Algorithmus, welcher für das Problem im Durchschnitt nur $\frac{5625}{1296} = 4,340$, im schlechtesten Fall jedoch sechs Versuche benötigt. Auch sie führten den entsprechenden Beweis, dass jeder Algorithmus, der mit maximal sechs Versuchen die korrekte Farbkombination bestimmt, im Durchschnitt mindestens 4,340 Versuche benötigt. Wichtiger ist jedoch der zusätzliche Beweis, dass für die Lösung des Problems im Durchschnitt mindestens 4,340 Versuche benötigt werden.

Koyama und Lai fanden den Algorithmus im Zuge einer Bruteforce-Berechnung zur Lösung des Problems. Nur durch einen genialen Trick, auf den in den zwei Jahrzehnten davor niemand gekommen war, ist es ihnen gelungen den Suchraum so entscheidend einzuschränken, dass eine vollständige Bruteforce-Suche mit der 1994 (und heute) zur Verfügung stehenden Hardware überhaupt erst möglich war.

Mastermind wurde 1973 von dem in Paris lebenden Israeli Marco Meirovitz erfunden. In der ursprünglichen Version als Brettspiel wurden die Farbkombinationen, sowie die schwarzen und weißen Punkte in Form von Farbstiften in ein spezielles Spielbrett gesteckt (siehe [Abbildung 2]). Mittlerweile existieren noch weitere Varianten des Spiels.

[Abbildung 2]:
Mastermind als Brettspiel

3 Aufgabenstellung des ACM Programming Contest

Die Aufgabenstellung betrachtet t verschiedene Mastermind-Spiele, im Folgenden als Testfälle bezeichnet. Es werden $p\,(1 \leqslant p \leqslant 10)$ als die Anzahl der Stellen der Farbkombination sowie $c\,(1 \leqslant c \leqslant 100)$ als die zu Verfügung stehenden Farben eines Testfalls definiert. Die Farben werden durch Zahlen $(1, 2, ..., 100)$ repräsentiert. Ein Testfall besteht nun aus $m\,(1 \leqslant m \leqslant 100)$ Versuchen mit den dazugehörigen Hinweisen (schwarze bzw. weiße Punkte).

Spezifikation der Eingabe:	**Beispiel 1:**
t	3
$p_1\ c_1\ m_1$	4 3 2
$w_{1\ 1}$	1 2 3 2
$h_{1\ 1}$	1 1
$w_{1\ 2}$	2 1 3 2
$h_{1\ 2}$	1 1
...	4 6 2
$w_{1\ m_1}$	3 3 3 3
$h_{1\ m_1}$	3 0
...	4 4 4 4
$p_t\ c_t\ m_t$	2 0
$w_{t\ 1}$	8 9 3
$h_{t\ 1}$	1 2 3 4 5 6 7 8
$w_{t\ 2}$	0 0
$h_{t\ 2}$	2 3 4 5 6 7 8 9
...	1 0
$w_{t\ m_t}$	3 4 5 6 7 8 9 9
$h_{t\ m_t}$	2 0

$w_{i\ j}$ =j-ter Versuch des i-ten Testfalles bestehend aus p_i Stellen
$h_{i\ j}$ =j-ter Hinweis des i-ten Testfalles
Hinweis = $b\ w$
b = Anzahl schwarze Punkte
w = Anzahl weiße Punkte

Hinweis:

Um die Eingabedatei übersichtlicher zu gestalten, wurde die Eingabespezifikation um einen Kommentar zu Beginn eines jeden Testfalles erweitert. Das Programm zur Generierung von Testfällen (GuessMaker.java) fügt im Kommentar zusätzlich die dem Testfall zugrunde liegende Farbkombination ein. Es handelt sich hierbei also um die gesuchte Farbkombination des Mastermind-Spiels und ist nicht mit der im Folgenden definierten Lösung v der Aufgabenstellung zu verwechseln.

Die eigentliche Aufgabe besteht nun darin einen Algorithmus zu entwickeln und zu implementieren der möglichst effizient zu jedem Testfall t_i den lexikographisch kleinstmöglichen Versuch v bestimmt der für t_i gültig ist. Ein Versuch v ist bezüglich eines Testfalles t_i gültig, sofern er nicht durch die m Versuche bzw. Hinweise von t_i ausgeschlossen wird. Da v in der Regel nicht eindeutig bestimmt ist, wird aus der Menge aller gültigen Versuche der lexikographisch kleinste ausgewählt.

Versuch $v = v_1, v_2, ..., v_p$ ist lexikographisch kleiner als Versuch $w = w_1, w_2, ..., w_p$ $\iff \exists k \in \mathbb{N}$, sodass $v_i = w_i$ $(i = 1, ..., k)$ und $v_{k+1} < w_{k+1}$

Ein Testfall t_i ist ungültig, wenn sich die m Versuche bzw. deren Hinweise von t_i widersprechen. In diesem Fall kann v nicht bestimmt werden und es wird stattdessen „You are cheating!“ ausgegeben.

Spezifikation der Ausgabe:	**Ausgabe zu Beispiel 1:**
v_1	1 1 1 3
v_2	You are cheating!
...	9 9 9 9 9 9 9
v_t	

4 Algorithmus, Implementierung

Die Grundidee des Algorithmus besteht in einer Bruteforce-Berechnung aller möglichen Lösungen eines Testfalles ausgehend vom kleinstmöglichen Versuch $v = \underbrace{1, ..., 1}_{p}$.

Das Hauptproblem besteht im möglicherweise sehr großen Suchraum, da im schlimmsten Fall c^p mögliche Lösungen auf Gültigkeit bezüglich der m Versuche eines Testfalles überprüft werden müssen. Mit den konkreten Werten der Aufgabenstellung müssten also pro Testfall im schlimmsten Fall 100^{10} Lösungen auf Gültigkeit bezüglich 100 Versuchen überprüft werden.

Folglich muss der Suchraum möglichst stark eingeschränkt werden. Dies geschieht mit der getrennten Analyse eines jeden Versuchs und dem zugehörigen Hinweis (Kapitel 4.1), sowie in der anschließenden Auswertung der dabei insgesamt für alle Versuche gewonnen Ergebnisse (Kapitel 4.2). Das Hauptziel ist hierbei festzustellen, ob bzw. wie oft welche Farbe in v auftritt. Im Folgenden wird, soweit sinnvoll, gleichzeitig auf die konkrete Implementierung in Java eingegangen. Zur Implementierung des Algorithmus wurde Java als Programmiersprache gewählt, da der Autor diese bis dato sehr häufig verwendet hat und infolgedessen gute Kenntnisse im Umgang mit dieser besitzt. Für die Implementierung werden jedoch grundsätzlich keine besonderen sprachspezifischen Konzepte benötigt.

4.1 Getrennte Analyse jedes Versuches

(Implementiert in der Methode analyzeGuess())

Hierbei werden ein Versuch g, sowie der zugehörige Hinweis (Anzahl der schwarzen und weißen Punkte) wie folgt ausgewertet:

w =Anzahl weiße Punkte
b =Anzahl schwarze Punkte
p =Anzahl der Stellen der Farbkombination des Testfalles
c_g =Anzahl der Farben in g

1. Fall: $w + b = 0 \Rightarrow$ sämtliche Farben von g treten in v *nicht* auf
2. Fall: $w + b = p \Rightarrow$ sämtliche Farben von g treten in v auf
3. Fall: $b = p \Rightarrow g$ ist mögliche Lösung des Testfalls
4. Fall: $c_g = 1 \wedge w + b > 0 \Rightarrow$ die gefundene Farbe tritt $(w + b)$-mal in v auf

Bereits hier kann es vorkommen, dass Widersprüche zwischen den Versuchen bestehen und der betrachtete Testfall somit ungültig ist. Dies ist genau dann der Fall, wenn bei der Analyse eines Versuches festgestellt wird, dass Farbe i in v vorkommt, diese jedoch bei einem vorhergehenden Aufruf bereits ausgeschlossen wurde bzw. wenn der entgegen gesetzte Fall eintritt (Farbe i wird ausgeschlossen, es wurde jedoch zuvor festgestellt, dass diese in v vorkommt).

Das Ergebnis der Analyse der m Versuche eines Testfalles sind somit Informationen über das Auftreten der Farben in v bzw. bei Widersprüchen die Feststellung, dass der Testfall ungültig ist.

Im Programm werden die gewonnen Informationen im Feld *int[] coloursA* wie folgt abgespeichert:

n= *coloursA*$[i]$ $(i = 0, ..., c-1)$:

1. Fall: $n < 0$ $\Rightarrow$ Farbe $i+1$ tritt in v *nicht* auf
2. Fall: $n > 0$ $\Rightarrow$ Farbe $i+1$ tritt in v n-mal auf
3. Fall: $n = 0$ $\Rightarrow$ keine Aussage möglich

Bemerkung:

Es sei noch erwähnt, dass der Autor trotz massiven Kopfzerbrechens keine weiteren als die oben angeführten (und relativ einfachen) Kriterien gefunden hat, die allgemeingütig zur Bestimmung der Farben von v hätten verwendet werden können. Es wurde jedoch festgestellt, dass bei einer globalen Sicht bzw. Analyse sämtlicher Versuche der Suchraum unter Umständen noch weiter eingeschränkt werden kann.

4.2 Bestimmung der zulässigen Farben für v

(Implementiert in der Methode getValidColours())

Mit Hilfe der in (Kapitel 4.1) gewonnen Informationen wird nun versucht alle zulässigen Farben von v zu bestimmen. Sei p' als die Summe über die Anzahl der bereits bestimmten Farben definiert. Im Programm entspricht dies der Summe aller Werte aus *coloursA[i]* $(i = 0, ..., c-1)$ die größer Null sind. Die Variable p' ist somit nichts anderes als die Anzahl der Stellen von v deren Farbe bekannt ist.

Es können folgende Fälle auftreten:

1. Fall: $0 \leq p' < p \Rightarrow$ zulässige Farben teilweise bestimmt, bei der Bestimmung von v werden somit nur ausgeschlossene Farben *nicht* berücksichtigt
2. Fall: $p' = p \Rightarrow$ sämtliche zulässigen Farben von v bestimmt
3. Fall: $p' > p \Rightarrow$ Testfall ungültig, da die Anzahl der bestimmten Stellen jene von v übersteigt

4.3 Bestimmung von v

(Implementiert in der Methode getSmallestGuess())

Wurde bereits in (Kapitel 4.1) eine mögliche Lösung gefunden, so wird überprüft ob diese für den Testfall gültig ist (wenn der Testfall mehrere mögliche Lösungen enthält, wird die zuletzt gefundene verwendet). Andernfalls werden aus den in (Kapitel 4.2) bestimmten zulässigen Farben alle möglichen Kombinationen gebildet und auf Gültigkeit bezüglich des Testfalles überprüft. Im Programm wird hierbei v auf Gültigkeit bezüglich der m Versuche des Testfalles überprüft (Implementiert in der Methode isValid()). Versuch v ist genau dann gültig bezüglich eines Versuchs g_i $(i = 1, ..., m)$, wenn sich die Hinweise von g_i nicht mit v widersprechen. Konnte v nicht bestimmt werden, so ist der Testfall ungültig.

Im schlimmsten Fall benötigt der Algorithmus somit exponentielle Laufzeit $(O(c^p))$, da sämtliche c^p Möglichkeiten überprüft werden müssen. Ist der Testfall gültig so benötigt der Algorithmus im besten Fall lineare Laufzeit.

5 Quellenverzeichnis

[WikiACM] http://de.wikipedia.org/wiki/Association_for_Computing_Machinery, (abgerufen am 14.10.2006)

[WikiMastermind] http://de.wikipedia.org/wiki/Mastermind, (abgerufen am 14.10.2006)

[Abbildung 1] Bild von Thomas Steiner (Lizenz: GNU FDL)

[Abbildung 2] Bild von ZeroOne (Lizenz: CC BY SA)

6 Anlagen

- Vortragspräsentation
- Quelldateien:
  - Mastermind.java
  - GuessMaker.java (Programm zur Generierung von Testfällen)