PostScript
spielt

MASTER MIND

Das Spiel

- erfunden
  1970-71 vom Israeli
  M. Meirowitz
  (Post- und Telekom-
  munikationsbranche)
- im deutschspr. Raum
  früher: **SuperHirn**
- diverse Varianten

Die Regeln

- Spielziel: den vom Gegner vorgegebenen Code knacken
- Code: 4 Stifte in jeweils einer von 6 Farben und einer bestimmten Reihenfolge
- 10 Versuche

Die Regeln (2)

- Wie?
  Nach jedem Versuch
  gibt der Gegner Tips:
  - 1 schwarzen Stift
    pro Stift mit
    richtiger Farbe
    und richtiger Position
  - 1 weißen Stift
    pro Stift mit
    richtiger Farbe
    aber *falscher* Position

Einige Zahlen

- Lösung $\Leftrightarrow$ 4 schwarze Stifte
- $6^4 = 1296$ mögliche Codes
- 14 verschiedene gültige Bewertungen
- optimal:
  - $\approx 4{,}341$ Züge bei max. 5
  - $\approx 4{,}340$ Züge bei max. 6

- Verallgemeinerung: n Positionen, c Farben

Aufgabenstellung

- Programm soll vorgegebenen Code knacken
- optional: (komplexere) Varianten, d. h. $n \neq 4$ und / oder $c \neq 6$
- einfacher, konsistenter Algorithmus
  > optimale Anzahl an Zügen
- Wieso PostScript?
  - einfache graphische Repräsentation leicht möglich
  - high-level Sprache
  - wollten schon immer unsere Drucker besser verstehen :-)

Lösungsansätze

- Heuristik
  - neuen, wahrscheinlicheren, Zug auf Basis des vorangegangenen u. s. Bewertung generieren
    - welche Heuristiken?
    - findet u. U. keine Lösung bzw. braucht viele Züge
- Ausschlußverfahren
  - alle Codes streichen, die durch eine Bewertung ausgeschlossen werden
    - für alle Möglichkeiten muß gespeichert werden, ob sie noch im Rennen sind → hoher Speicherplatzbedarf, komplexe Algorithmen u. Datenstrukturen (Spielbaumbeschneidung, ...) bzw. Vorarbeit (Strategietabellen)

Lösungsansätze (2)

- zwei light-weight Strategien (vgl. Swaszek 1999):
  - jeweils einen *konsistenten* Zug *zufällig* auswählen
    - beinahe optimal, d. h. ≈ 4,638 Züge bei max. 10
  - die Stifte als Zahl mit Ziffern zwischen 0 und (c-1) interpretieren und ausgehend von einem Startwert bis zum nächsten konsistenten Zug „hochzählen“
    - Qualität hängt vom Startwert ab
    - bester Startwert für den allg. Fall nicht bekannt

Lösungsansätze (3)

- unser Ansatz:
  - ausgehend von einem zufälligen Startwert bis zum nächsten konsistenten Zug „runterzählen“
  - noch mögliche / ausgeschlossene Züge werden dabei *nicht* gespeichert
    - VORTEIL: geringer Platzbedarf
    - VORTEIL: einfache Implementierung
    - VORTEIL: bei allen Codetypen etwa gleich stark
    - NACHTEIL: jeden Zug werden *alle* Codes zwischen Startwert und erstem konsistenten Zug neu evaluiert

Lösungsansätze (4)

- Verbesserungen möglich:
  - 1. Zug momentan fix → Heuristik?
  - *ranges* inkonsistenter Züge zwischenspeichern
  - random restart
  - history nach schwarzen > weißen Stiften sortieren
  - ...
- „Wichtiger Code“
  - nächsten Zug generieren = runterzählen
  - Züge bewerten bzw.
  - Konsistenz von Zügen prüfen

Code

- „Graphik zählt nicht“ → zeigen wir nicht
- nächsten Zug generieren
  - an sich beliebige Fkt., die alle Möglichkeiten zyklisch durchläuft
  - [guess] = [c_1, c_2, ... c_n]

```
% generates the "next" guess from given init
% Stack: [init] => [neighbor]

/neighbor {
    0 1 n 1 sub { % for
        2 copy get
        dup 0 ne
            {
             1 sub
             2 index 4 1 roll
             put
             exit
            } {
             pop c 1 sub
             2 index 4 1 roll
             put
            } ifelse
    } for
} bind def
```

Code (2)

- Bewertung:
  Anzahl der exakten Übereinstimmungen
  - entspricht Anzahl an schwarzen Stiften
  - wirft --mark-- ([ ) und eine 1 pro match auf den Stack und ...
- Summe
  - summiert alles bis zur letzten --mark-- auf

```
% exact matches
% Stack: [code] [code] => exact_matches
/matchingpositions {
  [ 3 1 roll
    0 1 n 1 sub { % for each position
        3 copy exch 1 index
        get
        3 1 roll
        get
        eq {1 4 1 roll} if
        pop
    } for
    pop pop sum
} bind def

% sums the stack back to mark
% Stack: [ i1 i2 ... in => sum_i
/sum {
    counttomark 0 exch {add} repeat
    exch pop
} bind def
```

Code (3-1)

- Bewertung:
  Anzahl der übereinstimmenden Farben
  - entspricht Gesamtanzahl der Stifte
  - Variante 1:
    Häufigkeit einer Farbe in beiden Codes bestimmen, Minimum bilden ... und über alle Farben summieren

```
% counts occurences of a single color in a code
% Stack [code] color => n

/countcolor {
    [ 3 1 roll exch
    {
     1 index 3 1 roll
     eq {1 exch} if
    } forall
    pop sum
} bind def
```

Code (3-2)

- Bewertung:
  Anzahl der übereinstimmenden Farben
  - entspricht Gesamtanzahl der Stifte
  - Variante 1:
    Häufigkeit einer Farbe in beiden Codes bestimmen, Minimum bilden ... und über alle Farben summieren

```
% color matches
% Stack: [code] [code] => color_matches

/matchingcolors {
    [ 3 1 roll
    0 1 c 1 sub { % for each color
        3 copy exch 1 index
        countcolor
        3 1 roll
        countcolor
        min
        4 1 roll pop
    } for
    pop pop sum
} bind def
```

Code (3-3)

- Bewertung:
  Anzahl der übereinstimmenden Farben
  - entspricht Gesamtanzahl der Stifte
  - Variante 2
    (bis 1,5 x so schnell):
    Häufigkeiten aller Farben in einem Code auf einmal ...

```
% generates a "histogram" ary for a given code
% Stack: code => [h_c0 h_c1 ... h_c(c-1)]

/countcolors {
    [c {0} repeat] exch
    {    % forall positions
        2 copy {1 add} aryreplace
        pop
    } forall
} bind def

% apply a proc to an ary element
% DESTRUCTIVE like the other ary ops
% Stack: ary i {proc} => _

/aryreplace {
    3 1 roll
    2 copy get
    4 -1 roll
    exec put
} bind def
```

Code (3-4)

- Bewertung:
  Anzahl der übereinstimmenden Farben
  - entspricht Gesamtanzahl der Stifte
  - Variante 2

    (bis 1,5 x so schnell):
    Häufigkeiten aller Farben in einem Code auf einmal ...

```
% Sum ( min(hist1[i], hist2[i]) )

% Stack: hist1 hist2 => color_matches


/colminsum {

    [ 3 1 roll

        aload pop c 1 add -1 roll aload pop

        c { % repeat

            c index min

            c 2 mul 1 roll

        } repeat

        c {pop} repeat sum

} bind def


% color matches

% Stack: [code] [code] => color_matches


/matchingcolors2 {

    countcolors exch

    countcolors

    colminsum

} bind def
```

Code (4)

- Bewertung:
  liegen zwei Codes in derselben Klasse?
  - wenn eval(*g*, *code*) die Bewertung (*b*, *w*) ergeben hat, liegen alle Versuche *X* mit eval(*g*, *X*) in derselben Klasse ...
  - negatives Ergebnis sollte schnell berechnet werden können

```
% checks if two codes belong to the same class

% Stack: eval [code] [code] => bool


/eligible-single {

    2 copy

    6 -1 roll 3 1 roll

    matchingpositions eq {

        matchingcolors2 eq

    } {

        pop pop pop false

    } ifelse

} bind def
```

Code (5)

- Bewertung:
  Wann ist ein Zug konsistent?
  - wenn er jeweils in einer Klasse mit allen bisherigen Versuch-Bewertungs-Paaren liegt.

```
% guess history

% entries: [ b w [guess] ]

/history tries 1 add array def


% checks if a guess is consistent with the
    guess history

% Stack: [guess] => bool


/eligible-all {

    history

    { % forall

        dup null eq { pop pop true exit } if

        aload pop 3 index

        eligible-single not

          { pop false exit } if

    } forall

} bind def
```

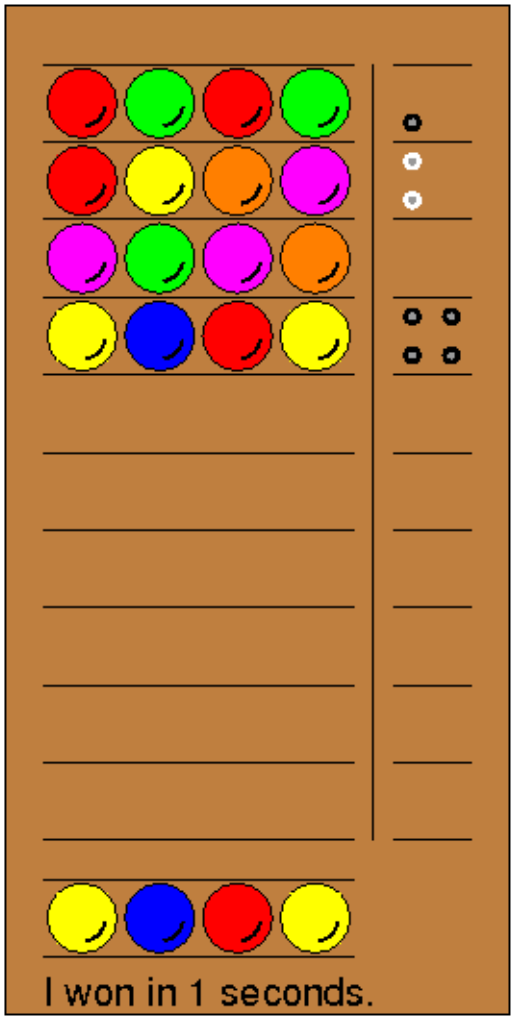
Code (6)

- farbige Reste:

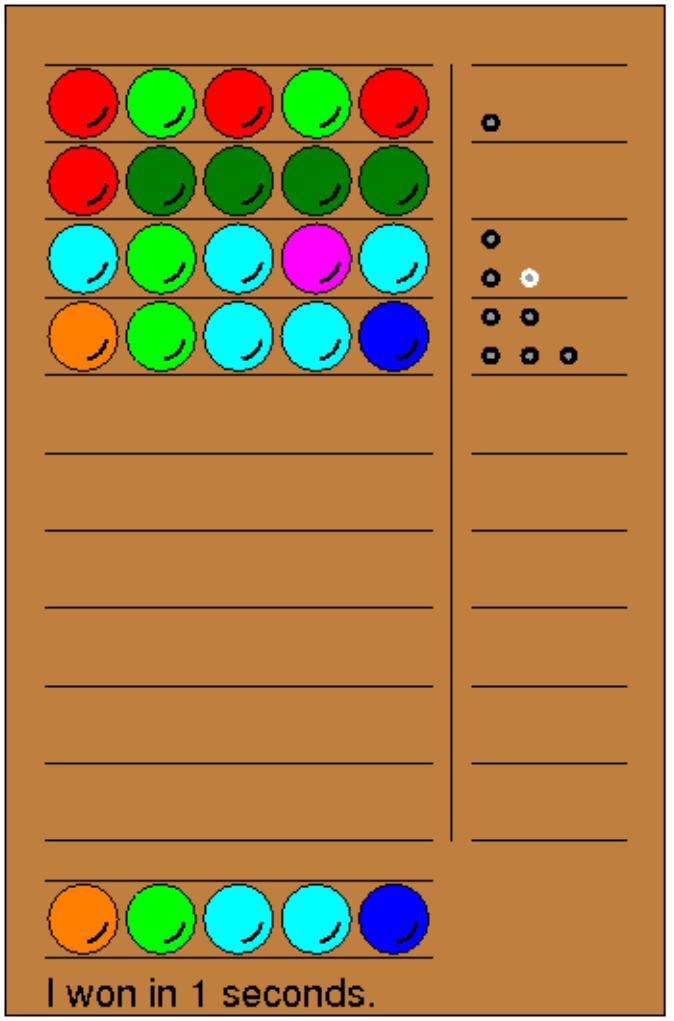
```
% generates a random consistent guess for
    the n-th move

% Stack: n => [guess]


/guess {

    0 gt {

        randomguess

        {

            dup

            eligible-all {exit} if

            neighbor

        } loop

    } { firstguess } ifelse

} bind def
```

```
% makes the n-th move

% Stack: n => _


/move {

    history exch

    [

        1 index guess

        dup code eval

        3 -1 roll

    ] put

} bind def


% color setters - CHANGE GSTATE

/red    {1 0 0 setrgbcolor} bind def

...


% colors (as defined above) to use for
    the pegs and board

/peg-colors [ /red /green /blue /yellow /orange
    /pink /cyan /dark-green /violet /white ]
    def

/board-color /brown load def
```

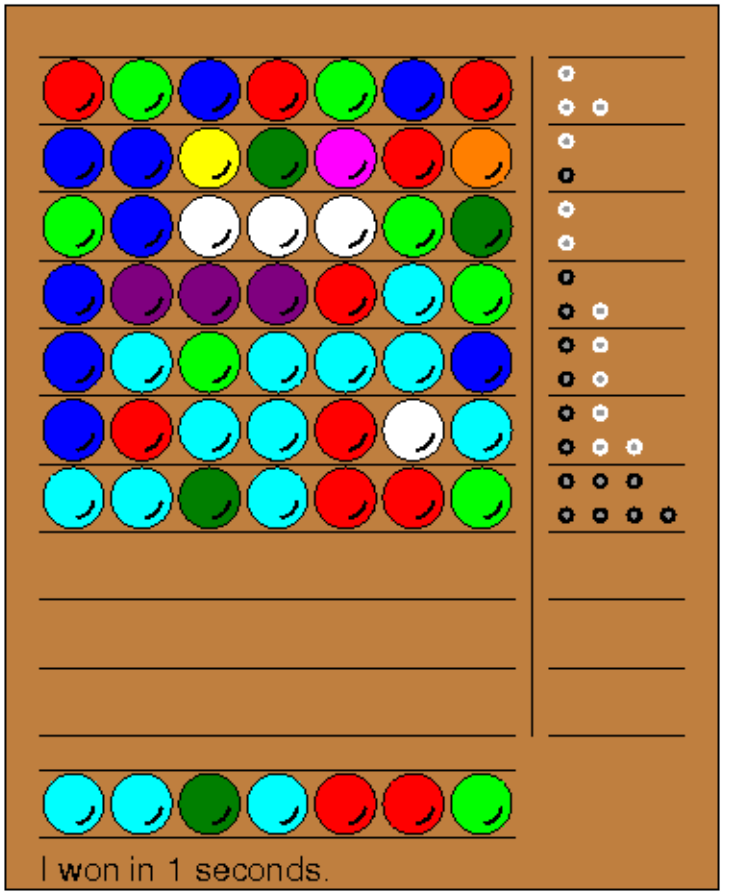
Beispiele



4/6 MM (0:01 min)



5/8 MM (0:01 min)



7/10 MM (5:48 min)

Literatur

Knuth , Donald E.

1976 „The computer as Master Mind“, *Journal of Recreational Mathematics* 9 (1976-77), 1-6.

Koyama Kenji u. Tony W. Lai

1993 „An optimal Mastermind strategy“, *Journal of Recreational Mathematics* 25 (1993), 251-256.

Nelson, Toby

1999 Investigations into the Master Mind board game. Break the hidden code. http://www.tnelson.demon.co.uk/mastermind/.

Swaszek, Peter F.

1999 „The Mastermind novice“, *Journal of Recreational Mathematics* 30 (1999-2000), 193-198.

Sonstige Quellennachweise

- Bild 1 (MasterMind Schachtel): Images of Master Mind, http://www.tnelson.demon.co.uk/mastermind/images1.html (images/mastermind7.jpg)
- Bild 2 (SuperHirn Brett): Doppelplusspiel, http://home.pages.at/ottodix/ (superhirn02.JPG)