Constraintprogrammierung

Eine Einführung

Niko Paltzer

Proseminar Programmiersysteme WS 03/04

Betreuer: Gert Smolka

2004/04/05 19:47:52

Inhaltsverzeichnis

1 Einführung **3**

2 Beispiele **4**
2.1 Send More Money 4
2.2 Safe 4
2.3 Map-Colouring 5
2.4 Supermarkt 5
2.5 Damen-Problem 6

3 Konkrete Anwendungen **7**

4 Begriffe und Definitionen **8**

5 Modell **9**
5.1 Variablenspeicher 9
5.2 Propagierer 9
5.3 Konfiguration 9

6 Lösungsstrategie **10**
6.1 Einengen der Bereiche 10
6.2 Distribution 10
6.2.1 Fallunterscheidung 11
6.3 Suche 12

7 Realisierung in Alice **13**
7.1 Skript 13
7.1.1 Variablen 13
7.1.2 Propagierer 13
7.1.3 Fallunterscheider 14
7.2 Suchmaschine 14

8 Beispiele in Alice **15**
8.1 Einfaches Beispiel 15
8.1.1 Variablen 15
8.1.2 Constraints 15
8.1.3 Fallunterscheider 15
8.1.4 Skript 15
8.2 n-Damen Problem 16
8.2.1 Variablen 16
8.2.2 Constraints 16
8.2.3 Fallunterscheider 16
8.2.4 neues Werkzeug 17
8.2.5 Skript 17

9 Literatur **18**

1 Einführung

Das Konzept von Constraints wurde erstmals 1963 in dem interaktiven Zeichen-Program *SKETCHPAD* von Ivan Sutherland benutzt. Seiten-, Längen- und Winkelverhältnisse wurden mittels Constraints beschrieben, um Zeichnungen korrekt und effizient skalieren zu können.

In den 70er Jahren wurden hauptsächlich im Bereich *Künstliche Intelligenz* mehrere Algorithmen für die Lösung eines *constraint satisfaction problem* (CSP) entwickelt. David Waltz erkannte bereits 1975, dass sich die Frage, ob eine zweidimensionale Zeichnung ein dreidimensionales Objekt darstellt, als CSP formulieren lässt.

Kurze Zeit später wurde mit der Entwicklung der ersten Programmiersprachen für Constraintprogrammierung begonnen. So wurde u.a. am ECRC in München um 1985 die praxis-orientierte Sprache CHIP realisiert, während man in Marseilles an Prolog II arbeitete.

In den 90ern schließlich begann man Forschungsergebnisse aus den Gebieten Künstliche Intelligenz, Computer Algebra und Mathemtische Logik zu kombinieren. Auch dabei entstanden neue Programmiersprachen und zugehörige Entwicklungsumgebungen wie z.B. der ILOG Solver und Oz/Mozart.

2 Beispiele

Es werden nachfolgend einige Beispiele aus [SS03] aufgeführt, die mit Hilfe von Constraints modelliert und mit den entsprechenden Werkzeugen gelöst werden können.

Zur Verdeutlichung wird stets eine Lösung angegeben.

2.1 Send More Money

Finden Sie eine eindeutige Ziffernbelegung für die Buchstaben

$$S \quad E \quad N \quad D \quad M \quad O \quad R \quad Y$$

so dass folgende Gleichung erfüllt ist:

$$S \quad E \quad N \quad D \quad + \quad M \quad O \quad R \quad E \quad = \quad M \quad O \quad N \quad E \quad Y$$

Führende Nullen sind dabei nicht erlaubt und zwei Buchstaben müssen unterschiedlichen Ziffern zugeordnet werden.

Lösung

$$S = 9 \quad E = 5 \quad N = 6 \quad D = 7 \quad M = 1 \quad O = 0 \quad R = 8 \quad Y = 2$$

$$9 \quad 5 \quad 6 \quad 7 \quad + \quad 1 \quad 0 \quad 8 \quad 5 \quad = \quad 1 \quad 0 \quad 6 \quad 5 \quad 2$$

Diese Lösung ist eindeutig, es gibt keine andere Ziffernbelegung, die die Bedingung erfüllt.

2.2 Safe

Knacken Sie den Safe! Der Code besteht aus einer Sequenz von 9 verschiedenen Ziffern $C_i \neq 0$ für die gilt:

$$\begin{gathered} C_4 - C_6 = C_7 \\ C_1 * C_2 * C_3 = C_8 + C_9 \\ C_2 + C_3 + C_6 < C_8 \\ C_9 < C_8 \\ C_1 \neq 1, \ldots, C_9 \neq 9 \end{gathered}$$

Lösung

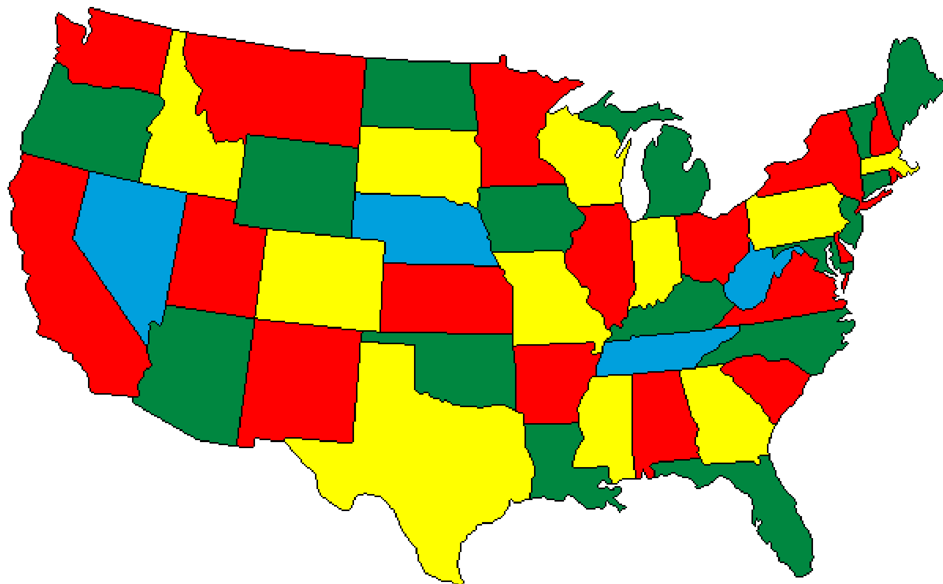
$C_1 = 4,\ C_2 = 3,\ C_3 = 1,\ C_4 = 8,\ C_5 = 9,\ C_6 = 2,\ C_7 = 6,\ C_8 = 7,\ C_9 = 5$

Auch hier gibt es keine alternative Lösung.

2.3 Map-Colouring

Färben Sie die Länder einer Landkarte so ein, dass zwei benachbarte Länder unterschiedliche Farben haben und benutzten Sie dabei möglichst wenige verschiedene Farben.

Lösung



Inzwischen ist bekannt, dass sich jede natürliche Karte mit maximal vier Farben in der geforderten Art kolorieren lässt.

Hier existieren für jede Karte üblicherweise mehrere Lösungen, die z.B. durch das Vertauschen von Farben zustande kommen.

2.4 Supermarkt

Sie haben im Supermarkt vier Artikel erstanden und dafür 7,11 € bezahlt. Erstaunt stellen Sie fest, dass das Produkt der Einzelpreise ebenfalls 7,11 € beträgt. Was kostet jeder der vier Artikel?

Lösung

3,16 € 1,20 € 1,25 € 1,50 €

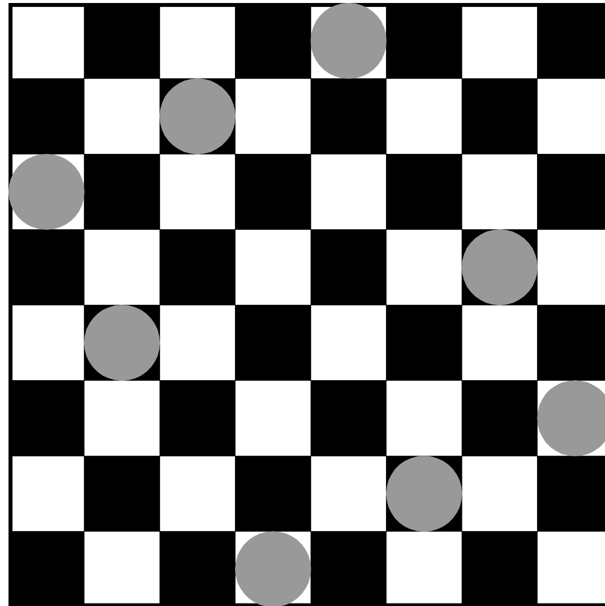
Die Korrektheit der Summe ist leicht zu sehen und bei der Berechnung des Produktes hilft teilweise Primfaktor-Zerlegung:

$$\frac{4 \cdot 79}{100} \cdot \frac{6}{5} \cdot \frac{5}{4} \cdot \frac{3}{2} = \frac{3 \cdot 79}{100} \cdot \frac{5}{5} \cdot \frac{4}{4} \cdot \frac{6}{2} = \frac{9 \cdot 79}{100} = \frac{711}{100}$$

2.5 Damen-Problem

Platzieren Sie 8 Damen auf einem Schachbrett, ohne dass sie sich gegenseitig bedrohen.

Lösung



Dies ist nur eine von 92 möglichen Lösungen. Die meisten davon sind zwar nur Spiegelungen an den denkbaren Achsen oder dem Mittelpunkt, aber es gibt auch mehrere grundverschiedene Möglichkeiten.

3 Konkrete Anwendungen

Abgesehen von Knobeleien und Rätseln findet Constraintprogrammierung eine ganze Reihe von Anwendungsmöglichkeiten in der Praxis.

Ablaufplanung Die Verteilung von Aufgaben oder Arbeitsgängen auf mehrere Maschinen, wobei die Arbeitsdauer und die korrekte Reihenfolge bei der Ausführung beachtet werden müssen. Natürlich wird hierbei eine beste (in diesem Fall natürlich schnellste) Lösung gesucht.

Turnierplanung Das Erstellen von Spielplänen und Ähnlichem gehört zum Alltag in fast jeder Sportart. Hierbei muss beachtet werden, dass in den Gruppen jede Mannschaft gegen jede andere Mannschaft gespielt hat, eine Mannschaft nicht gegen sich selbst spielt, ein Platz bei einem Spiel komplett belegt ist, u.s.w.

All diese Bedingungen lassen sich bequem in Constraints fassen.

Personalplanung Stunden-, Wach- und andere Pläne können ebenfalls erstellt werden, indem man das verfügbare Personal und die geforderten Anwesenheitszeiten durch Constraints ausdrückt.

Hier kann nach einer *besten* Lösung gesucht werden, wenn es gelingt, die Wünsche des Personals oder andere Prioritäten zu formalisieren.

Zuschnittprobleme Will man aus einem Material mehrere Teile heraus schneiden oder stanzen, so versucht man diese stets so anzuordnen, dass möglichst wenig Abfall entsteht, bzw. dass möglichst viele dieser Teile erzeugt werden. Auch hier kann mit Hilfe von Constraints ein Modell definiert werden und dann nach einer optimalen Lösung (maximale Anzahl von Teilen) gesucht werden.

Sprachverarbeitung Um natürliche Sprache erkennen und verarbeiten zu können sind komplexe Modelle notwendig. Eines dieser Modelle basiert darauf, Grammatiken als Teile von abstrakten Bäumen zu betrachten. Auf diesen Teilen werden Constraints definiert, die beschreiben, in welcher Art die Teile zusammen gesetzt werden dürfen um einen gültigen Satz zu bilden.

4 Begriffe und Definitionen

Nachfolgend werden kurze Erläuterungen zu Begriffen aufgeführt, die später häufig verwendet werden.

Variablen In dieser Einführung werden nur Variablen mit *finite domains*, also endlichen Definitionsbereichen behandelt. Um Verwirrungen zu vermeiden wollen wir festlegen, dass Variablen klein geschrieben werden und ihr Bereich genauso benannt ist wie die Variable, aber groß geschrieben wird (z.B. $x_i \in X_i$).

Außerdem beschränken wir uns auf Definitionsbereiche, die nur aus ganzen Zahlen bestehen. In [Apt03] findet man auch Beispiele mit Fließkommazahlen und abstrakten Datentypen.

Constraint Ein Constraint wird auf einer Menge von Variablen $\{x_1, \ldots, x_n\}$ definiert. Mathematisch betrachte ist ein Constraint eine Teilmenge von $X_1 \times \ldots \times X_n$. Ist $n = 1$ bzw. $n = 2$, so spricht man von unären bzw. binären Constraints.

Man unterscheidet zwei Formen von Constraints:

- **basic constraints**
  Constraints der Form $x_1 \in \{4, \ldots, 10\}$ oder $x_2 = x_6$.
- **nonbasic constraints**
  Constraints der Form $x_3 = x_4 + 3 \cdot x_5$ oder *distinct* $(x_1, \ldots, x_5)$

distinct $(x_1, \ldots, x_n)$ ist eine abkürzende Schreibweise für eine Menge von Constraints: $\{x_i \neq x_j | 1 \leq i < j \leq n\}$

constraint satisfaction problem (CSP) Ein CSP besteht aus einer Menge von Variablen $V = \{x_1, \ldots, x_n\}$ und einer Menge von Constraints $\{C_1, \ldots, C_n\}$, die jeweils auf einer Teilmenge von V definiert sind.

Ein CSP hat eine Lösung, wenn es eine Variablenbelegung $x_1, \ldots, x_n$ gibt, so dass alle Constraints $C_1, \ldots, C_n$ erfüllt sind.

Für ein CSP kann es mehrere, aber nur endlich viele Lösungen geben, da wir uns auf Constraints über Variablen mit endlichen Bereichen beschränken.

5 Modell

Die Grundstruktur und die Vorgehensweise zur Lösung von Contstraint-Problemen wird in diesem Abschnitt näher erläutert.

5.1 Variablenspeicher

Der Variablenspeicher enthält eine Menge von Variablen über denen Constraints definiert werden sollen. Zusätzlich speichert er zu jeder Variablen die zugehörige *finite domain*, die durch die Problem-Modellierung festgelegt wird.

Üblicherweise bildet eine Belegung der Variablen aus dem Variablenspeicher (oder zumindest eine Teilmenge von ihnen) später eine Lösung des CSP.

5.2 Propagierer

Constraints werden von Propagierern realisiert. Diese können mit dem Variablenspeicher kommunizieren und die Bereiche der darin enthaltenen Variablen einengen.

Ist ein Propagierer für alle möglichen Variablenbelegungen erfüllt, so wird er überflüssig (*entailed*) und verschwindet.

Beispiel: Der Propagierer für $x_1 < x_2$ auf $X_1 = \{2, 3, 4\}$ und $X_2 = \{5, 8\}$ ist überflüssig, was bei $X_1 = \{2, 3, 4\}$ und $X_2 = \{3, 8\}$ nicht der Fall ist.

Gibt es für einen Propagierer keine mögliche Variablenbelegung, so ist er fehlgeschlagen (*failed*). Ein CSP mit einem fehlgeschlagenen Propagierer hat keine Lösung.

5.3 Konfiguration

Variablenspeicher und Propagierer zusammen ergeben eine Konfiguration (*space*) eines CSP.

Eine Konfiguration ist **instabil**, wenn es in ihr einen Propagierer gibt, der mindestens den Bereich einer Variablen aus dem Variablenspeicher einengen kann.

Eine Konfiguration ist **stabil**, wenn es in ihr keinen Propagierer gibt, der einen Bereich einer Variablen aus dem Variablenspeicher einengen kann.

Eine Konfiguration ist **fehlgeschlagen**, wenn sie einen fehlgeschlagenen Propagierer enthält.

Eine Konfiguration ist **gelöst** (*solved*), wenn die Bereiche aller Variablen aus dem Variablenspeicher nur ein Element enthalten und es keinen fehlgeschlagenen Propagierer gibt.

6 Lösungsstrategie

Um eine Lösung eines CSP zu finden kombiniert man drei Verfahren: Einengen der Bereiche, Distribution und Suche

6.1 Einengen der Bereiche

Solange eine Konfiguration instabil ist, gibt es einen Propagierer, der den Bereich einer Variablen einengen kann.

In einer beliebigen (aber zu Beginn festgelegten) Reihenfolge engen die Propagierer also die Bereiche der Variablen ein, bis die Konfiguration stabil oder ein Propagierer fehlgeschlagen ist.

Ist die Konfiguration gelöst, so hat man eine Lösung für das CSP gefunden.

Ist die Konfiguration fehlgeschlagen (es gibt also noch mindestens eine Variable mit mehrelementigem Bereich), so fährt man mit Distribution fort.

6.2 Distribution

Die Distribution basiert auf der mathematischen Aussagenlogik:

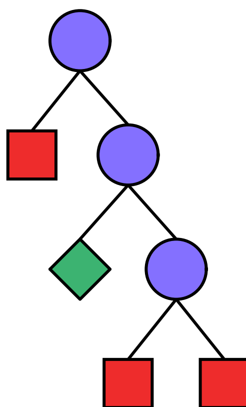
$$A \Leftrightarrow (A \wedge C) \vee (A \wedge \neg C)$$

Wir *erfinden* einen neuen Constraint C und machen eine Fallunterscheidung nach $(A \wedge C)$ und $(A \wedge \neg C)$.

Das geschieht, indem eine Konfiguration zweimal kopiert und jeweils ein neuer Propagierer für C bzw. $\neg C$ hinzugefügt wird.

Dann wird wieder das Einengen der Bereiche gestartet.

Durch diese Methode wird ein binärer Baum aus Konfigurationen aufgebaut. Die inneren Knoten dieses Baumes bestehen aus stabilen, ungelösten Konfigurationen. Die Blätter sind entweder gelöste oder fehlgeschlagene Konfigurationen.



Grüne Karos: gelöste Konfiguration
Rote Quadrate: fehlgeschlagene Konfiguration

6.2.1 Fallunterscheidung

Wie *erfindet* man Constraints um eine Fallunterscheidung zu machen?

Dafür gibt es verschiedene Strategien. Man legt zuerst eine beliebige Reihenfolge für die Variablen fest um eine Interpretation für **erste** Variable zu erhalten.

Zuerst wählt man sich eine Variable aus dem Variablenspeicher nach einem der folgenden Kriterien aus:

- **naive**
  nimm die erste Variable, deren Bereich mehr als ein Element enthält
- **firstfail**
  nimm die erste Variable, deren Bereich (mit) die wenigsten aber mehr als ein Element enthält

Hat man so eine Variable x ausgewählt, dann hat man folgende Möglichkeiten für die Fallunterscheidung:

- $x = l$, wobei l der kleinste mögliche Wert für x ist
- $x = u$, wobei l der größte mögliche Wert für x ist
- $x = m$, wobei m ungefähr der mittlere Wert für x ist
- $x \leq m$, wobei m ebenfalls ungefähr in der Mitte der *domain* von x liegt (der mögliche Bereich für x wird geteilt)

Anmerkung: Natürlich kann man sich beliebige andere Fallunterscheidungen konstruieren, hier sind nur die Standart-Möglichkeiten beschrieben.

Auch die nicht-deterministische Generierung von Fallunterscheidungen ist denkbar, hat aber den Nachteil, dass der Distributionsbaum bei jedem Aufbau anders aussieht.

6.3 Suche

Um in dem durch die Distributionsstrategie implizit festgelegten binären Baum eine Lösung zu finden verwendet man eine Suchmaschine.

Diese ist meist so konstruiert, dass sie nur den Teil des Baumes wirklich aufbaut, in dem sie auch sucht. Dieses *on demand*-Verhalten spart Speicherplatz und Rechenzeit.

Im Zusammenhang mit CSPs sind mehrere Fragestellungen möglich:

- Gibt es (mindestens) eine Lösung?
- Wie viele Lösungen gibt es?
- Wie sehen die Lösungen aus?
- Was ist die optimale Lösung?

Um Antworten auf diese Fragen zu bekommen, verwendet man unterschiedliche Suchmaschinen.

Bei der Suche nach einer Lösung z.B. ist eine Tiefensuche am effizientesten.

Sollen alle Lösungen gefunden werden, so könnte man über den Einsatz von Breitensuche nachdenken. Diese wird in der Praxis aufgrund des hohen Speicherplatzverbrauchs jedoch selten eingesetzt.

Um nach einer optimalen Lösung suchen zu können, muss man zuerst spezifizieren, was man unter *optimal* versteht.

Dies geschieht mit Hilfe einer Kostenfunktion, die der Suchmaschine übergeben wird. Diese Kostenfunktion ist in der Lage, eine schon gefundene Lösung mit dem Variablenspeicher einer stabilen Konfiguration zu vergleichen.

Fällt der Vergleich positiv aus, so könnte die Konfiguration eine besser Lösung als die bereits gefundene enthalten. Dann wird der entsprechende Teilbaum aufgebaut.

Fällt der Vergleich negativ aus, so überspringt man die Konfiguration und geht zur nächsten im Baum noch nicht ausgewerteten Konfiguration über (*backtracking*).

Dieses Verfahren nennt man *branch and bound*-Suche. Es ist effizient, wenn früh eine gute Lösung gefunden wird, denn dann werden viele Teilbäume nicht berechnet.

7 Realisierung in Alice

In der Programmiersprache Alice ([Tea03]) werden für CSPs sogenannte Skripte erstellt, die einer Suchmaschine übergeben werden. Alice stellt eine Bibliothek `Search` mit Suchmaschinen zur Verfügung.

Außerdem besteht die Möglichkeit, sich mit Hilfe des Alice-Explorers den Suchbaum anzeigen zu lassen und interaktiv die Suche zu steuern. Die entsprechenden Prozeduren sind:

```
Explorer.exploreOne : (unit -> 'a) -> unit
Explorer.exploreAll : (unit -> 'a) -> unit
```

7.1 Skript

Ein Skript hat den Typ `unit -> 'a`. Dabei ist `'a` der Ergebnistyp der Lösung.

In einem Skript werden lokal Variablen deklariert. Diese Variablen haben den Typ `FD.fd` (fd steht für *finite domain*).

7.1.1 Variablen

Initialisierung von Variablen z.B. mit:

```
FD.range : int * int -> FD.fd
FD.rangeVec : int * (int * int) -> FD.fd vector

val x = FD.range (1, 6)
val v = FD.rangeVec (4, (3, 7))
```

Erzeugt werden eine Variable $x \in \{1, \ldots, 6\}$ und ein Vector v mit 4 Variablen aus $\{3, \ldots, 7\}$.

7.1.2 Propagierer

Propagierer für Constraints werden ebenfalls mit Hilfe des Moduls `FD` erzeugt, z.B:

```
FD.equal : FD.fd * FD.fd -> unit
FD.plus : FD.fd * FD.fd * FD.fd -> unit
FD.distinct : FD.fd vector -> unit

FD.equal (x, y)
FD.plus (x, y, z)
FD.distinct (v)
```

Erzeugt werden Propagierer für $x = y$, $x + y = z$ und die paarweise Ungleichheit aller Variablen in v.

7.1.3 Fallunterscheider

Fallunterscheider können angelegt werden mit:

```
FD.distribute : FD.dist_mode * FD.fd vector -> unit
```

Dabei wird der Fallunterscheider für einen Vektor von Variablen angelegt, damit die bereits erwähnte Sortierung und die Bedeutung von *erste* Variable klar ist.

Zu `FD.dist_mode` gehören z.B.

- `FD.NAIVE`
  Für die erste, nicht determinisierte Variable x mit einem Bereich $\{l, \ldots, u\}$ werden die Fälle $x = l$ und $x \neq l$ unterschieden.
- `FD.FIRSTFAIL`
  Wie `FD.NAIVE`, nur wird die erste Variable mit dem *kleinsten* Bereich ausgewählt.

7.2 Suchmaschine

Es werden verschiedene Suchmaschinen zur Verfügung gestellt, z.B.

```
Search.searchOne : (unit -> 'a) -> 'a option
Search.searchAll : (unit -> 'a) -> 'a list
```

Sie finden für ein Skript die Lösung bzw. alle Lösungen, falls diese existieren.

8 Beispiele in Alice

In diesem Abschnitt möcht ich zwei Beispiel-Skripte vorstellen, die in Alice implementiert sind.

8.1 Einfaches Beispiel

Ein möglichst einfaches Beispiel modellieren wir wie folgt:

8.1.1 Variablen

$$x \in \{1, 2, 3\} \quad y \in \{2, 3\} \quad z \in \{1, 2, 3, 4\}$$

8.1.2 Constraints

$$x < y \qquad x^2 = z$$

8.1.3 Fallunterscheider

First-Fail-Distribution nach dem kleinsten Wert

8.1.4 Skript

Zunächst werden Variablen x, y und z angelegt und in einen Vektor gepackt. Dann werden die Propagierer und der Fallunterscheider erzeugt. Zurück gegeben wird der Vektor v.

```
fun example () =
   let
      val x = FD.range (1, 3)
      val y = FD.range (2, 3)
      val z = FD.range (1, 4)
      val v = #[x, y, z]
   in
      FD.less (x, y);
      FD.times (x, x, z);
      FD.distribute (FD.FIRSTFAIL, v);
      v
   end
```

8.2 n-Damen Problem

Als Beispiel für ein parametrisiertes Problem betrachten wir eine abgewandelte Form des Damen-Problems aus Abschnitt Beispiele, das *n-Damen-Problem*. Dieses modellieren wir wie folgt:

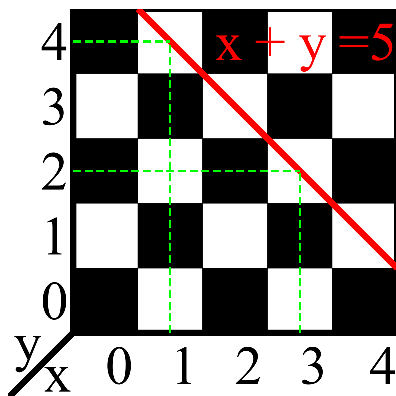
8.2.1 Variablen

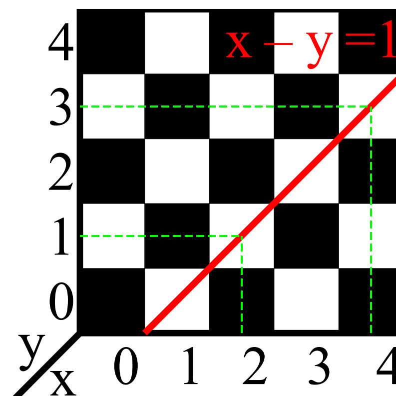
Eine Variable für jede Reihe des Spielbretts: $R_0, \ldots, R_{n-1} \in \{0, \ldots, n-1\}$

8.2.2 Constraints

Nur eine Dame pro Spalte: $\forall i, j, i < j : R_i \neq R_j$
Nur eine Dame pro Diagonale:

$\forall i, j, i < j : R_i + i \neq R_j + j$ $\forall i, j, i < j : R_i - i \neq R_j - j$





Die Zeichnungen verdeutlichen den Sinn der letzten beiden Constraints. Auf einer Diagonalen sind Summe bzw. Differenz von Spalte und Zeile gleich. Da auf jeder Diagonalen nur eine Dame stehen darf, müssen die Ergebnisse also paarweise verschieden sein.

8.2.3 Fallunterscheider

First-Fail-Distribution nach dem kleinsten Wert

8.2.4 neues Werkzeug

Um dieses Modell umsetzen zu können benötigen wir eine neue Art von Propagierer:

```
FD.distinctOffset : (FD.fd * int) vector -> unit
```

Diese Funktion ist eine Erweiterung von `FD.distinct` mit folgenden Eigenschaften:

$v =$

fd_0	fd_1	fd_2	...	fd_{n-1}
i_0	i_1	i_2	...	i_{n-1}

$$\begin{aligned}&\mathit{distinctOffset}\ \ v = \\ &\quad \mathit{distinct}\ (fd_0 + i_0,\ fd_1 + i_1,\ fd_2 + i_2,\ \ldots,\ fd_{n-1} + i_{n-1})\end{aligned}$$

8.2.5 Skript

Bei diesem Beispiel wird ein Vektor v mit n Variablen angelegt und zwei weitere Vektoren, die für die Verwendung mit `FD.distinctOffset` ausgelegt sind.

```
fun nQueens n () =
   let
      val v  = FD.rangeVec (n, (0, n-1))
      val v1 = Vector.mapi (fn (i,x)=>(x, i)) v
      val v2 = Vector.mapi (fn (i,x)=>(x,~i)) v
   in
      FD.distinct v;
      FD.distinctOffset v1;
      FD.distinctOffset v2;
      FD.distribute(FD.FIRSTFAIL, v);
      v
   end
```

9 Literatur

[Apt03] Krzysztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[Sch00] Christian Schulte. *Programming Constraint Services.* Dissertation, 2000.

[SS03] Christian Schulte and Gert Smolka. *Finite Domain Constraint Programming in Oz. A Tutorial.* www.mozart-oz.org/documentation/fdt/, 2003.

[Tea03] The Alice Team. *The Alice System, Online Manual.* www.ps.uni-sb.de/alice/, 2003.