Exhaustion[1]

Inhaltsverzeichnis

1 Exhaustionsalgorithmen

1.1 Grundlagen

Es gibt eine große Klasse von Fragestellungen, die sich so charakterisieren lassen: Gegeben seien n Variablen x_0 bis x_{n-1}, die nur diskrete Werte aus einem gegebenen endlichen Bereich $\mathbb{D}$ annehmen können. Die Variablen bilden ein n-Tupel der Form $x = \langle x_0, \ldots, x_{n-1} \rangle$. Gesucht ist eine Belegung dieser Variablen mit Werten aus $\mathbb{D}$, sodass

- bestimmte Bedingungen für die Werte x_0 bis x_{n-1} erfüllt sind und
- eventuell eine gegebene Funktion $f(x)$ einen bestimmten Wert, meist ein Minimum oder ein Maximum, annimmt.

Eine konkrete Belegung von x, die alle angeführten Bedingungen erfüllt, nennen wir ein *Lösungstupel*. Drei „berühmte“ Beispiele für solche Problemstellungen sind die folgenden: Achtdamenproblem, Rucksackproblem und Rundreiseproblem. Sie werden im Weiteren vorgestellt.

[1] entnommen aus Vorlesungsunterlagen von Prof. Rechenberg, Universität Linz

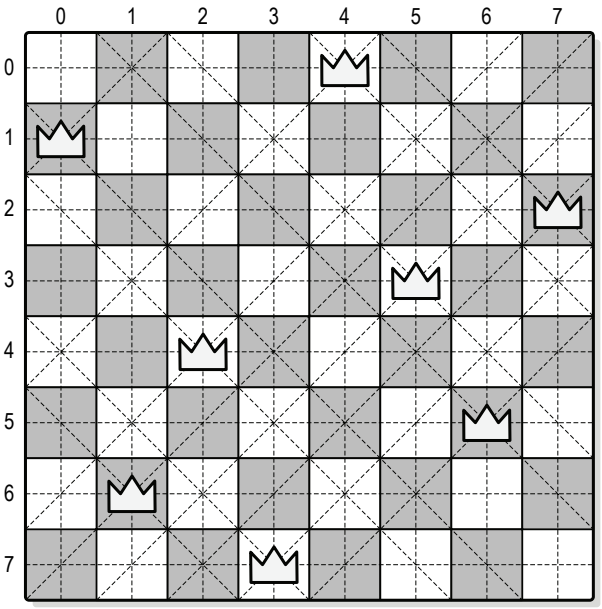

Abbildung 1: Eine der 92 Lösungen – wenn man alle symmetrischen Lösungen mitzählt – des Achtdamenproblems

1.1.1 Achtdamenproblem

Gesucht ist eine Anordnung von acht Damen auf einem Schachbrett, sodass keine Dame eine andere schlagen kann – siehe dazu Abbildung 1. In etwas formalerer Notation: Gesucht ist ein 8-Tupel $x = \langle x_0, \ldots, x_7 \rangle$ mit $x_i \in \mathbb{N}_8$ derart, dass die folgenden drei Bedingungen gelten. (Dabei bedeutet $x_r = c$, dass eine Dame auf einem Feld mit dem Zeilenindex r und dem Spaltenindex c steht.)

1. $\bigwedge_{i,j \in \mathbb{N}_8, i \neq j} x_i \neq x_j$ verschiedene Spalten,
2. $\bigwedge_{i,j \in \mathbb{N}_8, i \neq j} i - x_i \neq j - x_j$ verschiedene Hauptdiagonalen,
3. $\bigwedge_{i,j \in \mathbb{N}_8, i \neq j} i + x_i \neq j + x_j$ verschiedene Nebendiagonalen.

1.1.2 Rucksackproblem

Gegeben seien n Gegenstände mit verschiedenen Gewichten g_i und verschiedenen Werten w_i (mit $0 \leq i < n$). Aus den Gegenständen müssen welche ausgewählt und in einen Rucksack gepackt werden. Und zwar so, dass ein gegebenes maximales Gesamtgewicht $g_{\max}$ nicht überschritten wird und der Wert der im Rucksack vereinigten Gegenstände $w_{\max}$ möglichst groß wird. Wieder etwas formaler: Gesucht ist ein n-Tupel $x = \langle x_0, \ldots, x_{n-1} \rangle$ mit $x_i \in \{\textbf{false}, \textbf{true}\}$ derart, sodass gilt:

$$\sum_{i \in \mathbb{N}_n} x_i \cdot g_i \leq g_{\max} \quad \text{und} \quad \sum_{i \in \mathbb{N}_n} x_i \cdot w_i \text{ maximal.}$$

1.1.3 Rundreiseproblem

Gegeben seien n Städte 0 bis $n-1$ und die Entfernungen zwischen ihnen. Der Ausdruck $d(i,j)$ bezeichnet die Entfernung zwischen den beiden Städten i und j. Gesucht ist ein kürzester Weg, der alle Städte ohne Schleifen kreisförmig miteinander verbindet, also in Stadt 0 anfängt, alle anderen Städte genau einmal berührt und wieder in Stadt 0 endet. Formal formuliert: Gesucht ist ein n-Tupel $x = \langle x_0, \ldots, x_{n-1} \rangle$ mit $x_i \in \mathbb{N}_n$ derart, sodass gilt:

$$x_i \neq x_j \text{ für } i,j \in \mathbb{N}_n \quad \text{und} \quad \sum_{i \in \mathbb{N}_n} d(x_i, x_{(i+1) \bmod n}) \text{ minimal.}$$

1.1.4 Allgemeiner Algorithmus

Es handelt sich bei allen drei Aufgaben um kombinatorische Suchprobleme, bei denen man aus allen möglichen Kombinationen der Eingabewerte diejenigen suchen muss, die die gestellten Bedingungen erfüllen. Man kann für solche Probleme immer eine Lösung finden, auch wenn man keinen speziellen Algorithmus dafür kennt, indem man systematisch alle möglichen Wertetupel erzeugt – es sind endlich viele, da jede der n Lösungsvariablen nur endlich viele Werte annehmen kann – und prüft, ob das Wertetupel eine gültige Lösung darstellt.

```
1  function EXHAUSTION (↓n, ↓condition, ↑x)
2      x ← erste mögliche Wertebelegung
3
4      loop
5          if x erfüllt condition then
6              return true
7
8          else if x ist letzte mögliche Wertebelegung then
9              return false
10
11         else
12             x ← nächste mögliche Wertebelegung
13         end if
14     end loop
15 end function
```

Da die Anzahl der Kombinationen von n Elementen, bei denen jedes b Werte annehmen kann, b^n beträgt, ist die Anzahl der Schleifendurchläufe und damit die Zeitkomplexität dieses Algorithmus $O(b^n)$. Etwas „besser“ ist es, das Lösungstupel x elementweise aufzubauen. Man beginnt mit dem ersten möglichen Wert der ersten Komponente des n-Tupels x (hier für $n = 4$):

```
1  x ← ⟨x0, −, −, −⟩
```

Dann prüft man, ob diese Teillösung zum Ziel führen kann. Wenn ja, dann fügt man den ersten möglichen Wert für die zweite Komponente ein:

```
1  x ← ⟨x0, x1, −, −⟩
```

Wenn nein, so gibt man x_0 den nächsten möglichen Wert. Wenn x_0 und x_1 zu einer Lösung führen können, so fügt man einen passenden Wert für x_2 an. Wenn nicht, verändert man x_1 solange, bis x_0 und x_1 zu einer Lösung führen können. Wenn man keinen solchen Wert für x_1 findet, heißt das, dass der Wert von x_0 nicht zu einer Lösung führen kann. Dann muss man zurück und x_0 ändern. Dieses Verfahren setzt man fort, bis man entweder ein vollständiges

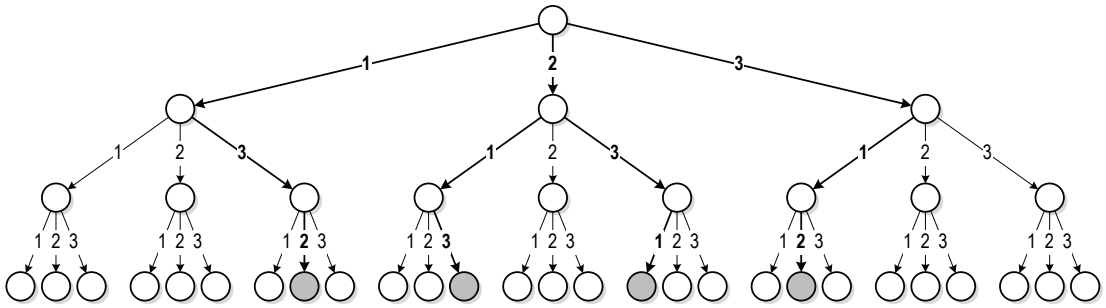

Abbildung 2: Diejenigen Wege, deren Blätter markiert sind, sind Lösungstupel. Sie lauten $\langle 1,3,2\rangle$, $\langle 2,1,3\rangle$, $\langle 2,3,1\rangle$ und $\langle 3,1,2\rangle$.

Lösungstupel gefunden oder alle Möglichkeiten durchprobiert hat. Wegen der Möglichkeit, dass eine Teillösung wieder zurückgenommen werden muss, weil sie nicht zum Ziel führt, heißen Algorithmen, die so arbeiten, im Englischen *Backtracking-Algorithmen*.

1.2 Interpretation als Baumdurchsuchen

Die b^n Kombinationen des Tupels $\langle x_0,\ldots,x_{n-1}\rangle$ lassen sich als Baum darstellen, in dem jeder Weg von der Wurzel zu einem Blatt eine Kombination der Lösungsvariablen bildet Das Backtrackingverfahren lässt sich als *Tiefensuchen* in diesem Baum interpretieren. Es durchläuft den Baum von der Wurzel bis zu den Blättern und benutzt dazu nur solche Wege, deren Knoten Bestandteile von Teillösungen sind. Ein Beispiel: Wenn wir annehmen, dass ein Lösungstupel der Länge 3 bestehend aus den Zahlen 1, 2 und 3 gesucht wird, sodass jede Zahl einmal vorkommt und die Differenzen $|x_1 - x_0|$ und $|x_2 - x_1|$ verschieden sind, so sieht der Lösungsbaum wie in Abbildung 2 aus.

1.3 Komplexität

Wie bereits erwähnt besitzt der Baum bei n Lösungsvariablen mit je b Werten b^n Blätter, die zum Auffinden einer Lösung möglicherweise alle besucht werden müssen. Die Laufzeit kann zwar durch geschickte Auswahl von Algorithmen für ein bestimmtes Exhaustionsproblem verkleinert werden, die exponentielle asymptotische Laufzeitkomplexität bleibt jedoch immer erhalten. Exhaustionsalgorithmen sind daher nur für kleine Probleme einsetzbar. Es lassen sich drei Aufgabentypen unterscheiden:

- Gesucht ist irgendein Lösungsvektor: *einfacher Backtrackingalgorithmus*.
- Gesucht sind alle Lösungsvektoren: *Exhaustionsvariante*.
- Gesucht ist ein Lösungsvektor, der eine bestimmte Optimalitätsbedingung erfüllt: *Optimierungsvariante*.

1.4 Algorithmen

1.4.1 Einfacher rekursiver Backtrackingalgorithmus

Der einfache Backtrackingalgorithmus lässt sich am klarsten rekursiv formulieren. Dazu stellen wir das Lösungstupel x als Vektor $x_{0,n-1}$ dar und füllen ihn, bei x_0 beginnend, Element für Element auf. Wenn $x_{0,i-1}$ gefüllt ist und eine

mögliche Teillösung darstellt, und wir finden ein x_i, sodass $x_{0,i}$ eine mögliche Teillösung ist, dann sagen wir dafür abgekürzt „x_i passt (vorerst)“. Diesem Algorithmus entspricht die rekursive Funktion LOESUNG $(\downarrow i)$:

```
1  function LOESUNG (↓i)        — x_{0,i−1} bildet eine Teillösung
2      for alle möglichen Werte von x_i do
3          if x_i passt zur bisherigen Teillösung then
4              erzeuge mit x_i eine neue Teillösung
5
6              if i = n − 1 then
7                  WRITE (↓x)        — Lösung gefunden
8                  halt
9              end if
10
11             LOESUNG (↓(i+1))
12
13             nimm Teillösung mit x_i zurück        — Teillösung mit x_{0,i} nicht möglich
14         end if
15     end for        — Teillösung mit x_{0,i−1} nicht möglich
16 end function
```

Die Funktion LOESUNG $(\downarrow i)$ besitzt die folgende Spezifikation: Bei Eintritt in die Funktion ist $x_{0,i-1}$ eine Teillösung des gegebenen Problems. Der Aufruf von LOESUNG $(\downarrow i)$ hat die Aufgabe, die restlichen Lösungsvariablen x_i bis x_{n-1} so zu bestimmen, dass sich eine vollständige Lösung ergibt. Wenn das gelingt, so druckt die Funktion LOESUNG den Vektor $x_{0,n-1}$ als Ergebnis aus und hält an. Wenn es nicht gelingt, so kehrt LOESUNG zum Aufrufer zurück. Bei der Rückkehr von LOESUNG $(\downarrow i)$ gilt deshalb die Zusicherung (auf Englisch: *assertion*): „Die Teillösung mit den augenblicklichen Werten von $x_{0,i-1}$ führt nicht zum Ziel.“ Das Lösungstupel $x_{0,n-1}$ ist in LOESUNG global bekannt. Der Aufruf von LOESUNG lautet:

```
1 LOESUNG (↓0)
2 WRITE (↓"Keine Lösung gefunden.")
```

Man beachte, dass dieser Algorithmus zwei Enden hat: Das **halt** mit der Semantik *Erfolg* und das reguläre Ende mit der Semantik *Misserfolg*. Das Ergebnis *Erfolg* oder *Misserfolg* ist für Backtrackingalgorithmen charakteristisch, es wird dem Leser aber nicht schwer fallen, den Algorithmus so abzuändern, dass er immer regulär endet und Erfolg oder Misserfolg durch einen booleschen Ausgangsparameter *success* signalisiert.

1.4.2 Einfacher iterativer Backtrackingalgorithmus

Auch nichtrekursive Backtrackingalgorithmen sind möglich – entweder durch Entrekursivierung des rekursiven Algorithmus oder durch die direkte Entwicklung eines nichtrekursiven Algorithmus. Zum Studium der dabei auftretenden Schwierigkeiten sei die Entwicklung eines nichtrekursiven Backtrackingalgorithmus dem Leser ausdrücklich empfohlen. Einen besonders einfachen und leicht zu durchschauenden nichtrekursiven Backtrackingalgorithmus zeigt der nächste Algorithmus. Es ist ein zustandsgesteuerter Algorithmus mit den drei Zuständen *check*, *failure* und *success*:

```
1  function LOESUNG ()
2      ⟨i, state, x_i⟩ ← ⟨0, check, erster möglicher Wert⟩
3
4      loop
5          if state = check then
6              STATECHECK (↓i, ↕state)
7
8          else if state = failure then
9              STATEFAILURE (↕i, ↕state)
10
11         else if state = success then
12             STATESUCCESS (↕i, ↕state)
13         end if
14     end loop
15 end function
```

```
1  function STATECHECK (↓i, ↕state)
2      if x_i passt then
3          state ← success
4      else
5          state ← failure
6      end if
7  end function
```

```
1  function STATEFAILURE (↓i, ↕state)
2      if x_i = letzter möglicher Wert then
3          i ← i − 1
4
5          if i < 0 then
6              WRITE (↓"Keine Lösung gefunden.")
7              halt
8          end if
9      else
10         ⟨state, x_i⟩ ← ⟨check, nächster möglicher Wert⟩
11     end if
12 end function
```

```
1  function STATESUCCESS (↓i, ↕state)
2      if i = n − 1 then
3          WRITE (↓x)
4          halt
5      end if
6
7      ⟨i, state, x_i⟩ ← ⟨i + 1, check, erster möglicher Wert⟩
8  end function
```

Die Semantik der drei Zustände dieses Algorithmus zeigt die folgende Tabelle:

Zustand	Semantik	
check	$x_{0,i-1}$	ist eine Teillösung
failure	$x_{0,i}$	ist keine Teillösung
success	$x_{0,i}$	ist eine Teillösung

1.4.3 Exhaustionsvariante

Der einfache Backtrackingalgorithmus findet nur eine, nämlich die erste Lösung. Mit der Exhaustionsvariante will man alle Lösungen finden. Es ist interessant, dass man den einfachen Backtrackingalgorithmus nur um eine Kleinigkeit ändern muss, um den Exhaustionsalgorithmus zu bekommen. Wir geben ohne weitere Erläuterungen den rekursiven Algorithmus dafür an.

```
1  function LOESUNG (↓i)
2      for alle möglichen Werte von x_i do
3          if x_i passt zur bisherigen Teillösung then
4              erzeuge mit x_i eine neue Teillösung
5
6              if i = n − 1 then
7                  WRITE (↓x)
8              else
9                  LOESUNG (↓(i+1))
10             end if
11
12             nimm Teillösung mit x_i zurück
13         end if
14     end for
15 end function
```

1.4.4 Optimierungsvariante

Oft ist nicht irgendeine Lösung bzw. es sind nicht alle Lösungen gesucht, sondern eine hinsichtlich bestimmter Eigenschaften optimale Lösung. Hier müssen ebenfalls im ungünstigsten Fall alle Lösungen gesucht und zusätzlich muss die bisher beste Lösung extra gespeichert werden. Man kann aber die betrachteten Teillösungen oft dadurch einschränken, dass man nur solche verfolgt, die zu Lösungen führen können und die die bisher optimale Lösung übertreffen. Der entsprechende Algorithmus lautet:

```
1  function LOESUNG (↓i)
2      for alle möglichen Werte von x_i do
3          if x_i passt zur bisherigen Teillösung und kann zu einem neuen Optimum führen then
4              erzeuge mit x_i eine neue Teillösung
5
6              if i = n − 1 then
7                  if neue Lösung ist besser als das bisherige Optimum then
8                      ⟨x_opt, optimum⟩ ← ⟨x, f(x)⟩
9                  end if
10             else
11                 LOESUNG (↓(i+1))
12             end if
13
14             nimm Teillösung mit x_i zurück
15         end if
16     end for
17 end function
```

Der Aufruf von LOESUNG lautet:

```
1  optimum ← schlechtester möglicher Wert
2  LOESUNG (↓0)
3
4  if Optimum ist besser als der schlechtest mögliche Wert then
5      WRITE (↓x_opt, ↓optimum)
6  else
7      WRITE (↓"Keine Lösung gefunden.")
8  end if
```