Backtracking

Der Suchraum bei einem *NP*-vollständigen Problem, läßt sich oft so schreiben:

$$S_1 \times S_2 \times \cdots \times S_n$$

Jeder Punkt im Suchraum ist dann ein Vektor

$$(x_1, x_2, \ldots, x_n)$$

mit $x_i \in S_i$.

Die Grundidee ist es, nacheinander $x_1, x_2, \ldots, x_n$ zu wählen.

Bounding Functions

Sind bereits $x_1, \ldots, x_m$ gewählt, können wir manchmal bereits leicht entscheiden, daß es keine Lösung geben kann, die so beginnt.

Wir nehmen an, es gibt Prädikate $B_1, \ldots, B_n$, so daß wenn

$$B_k(x_1, x_2, \ldots, x_k)$$

falsch ist, dann gibt es keine Lösung $(y_1, \ldots, y_n)$ mit $y_i = x_i$ für $i = 1, \ldots, k$.

Der Algorithmus kann diese Prädikate verwenden, um den Suchraum einzuschränken.

Beispiel

Beim 8-Damenproblem können wir als Suchraum

$$\{1, \ldots, 8\}^8$$

wählen, denn in jeder Zeile steht eine Dame und sie muß in einer der 8 Spalten sein.

Wir setzen $B_k(x_1, \ldots, x_k) = \mathit{false}$, wenn sich bereits zwei der ersten k Damen in den ersten k Zeilen auf den Positionen $x_1, \ldots, x_k$ bedrohen.

Rekursiver Algorithmus

```
Backtrack(k) :
for each x_k ∈ S_k do
  if B_k(x_1, ..., x_k) then
    if (x_1, ..., x_k) ist eine Lösung
    then write (x_1, ..., x_k) fi
    if k < n then Backtrack(k + 1) fi
  fi
od
```

Die Variablen $x_1, \ldots, x_n$ sind global.

Der Algorithmus muß mit $Backtrack(1)$ aufgerufen werden.

Bounding Functions

Wie wählen wir die Prädikate B_k?

Ein Extremfall ist

$$B_k(x_1, \ldots, x_k) = \mathit{true}$$

für alle x_i. Hier wird der Suchbaum niemals abgeschnitten und der gesamte Suchraum durchsucht.

Der andere Extremfall ist

$$B_k(x_1, \ldots, x_k) = \begin{cases} \mathit{true} & \text{falls eine Lösung mit } x_1, \ldots, x_k \text{ beginnt,} \\ \mathit{false} & \text{sonst.} \end{cases}$$

Hier werden alle Teilbäume abgeschnitten, die keine Lösung enthalten.

Beide Extremfälle sind oft ungünstig. Entweder wird zuwenig abgeschnitten oder der Berechnungsaufwand für B_k wird zu groß.

Iterativer Algorithmus

Backtrack :
$k := 1$; $R_k := S_k$
while $k > 0$ **do**
  if $\{ x \in R_k \mid B_k(x_1, \ldots, x_{k-1}, x) \} \neq \emptyset$ **then**
    choose $x_k \in \{ x \in R_k \mid B_k(x_1, \ldots, x_{k-1}, x) \}$
    $R_k := R_k - \{x_k\}$;
    if $(x_1, \ldots, x_k)$ ist eine Lösung **then write** $x_1, \ldots, x_k$;
    $k := k + 1$; $R_k := S_k$
  else $k := k - 1$
od

Abschätzung der Suchbaumgröße

Die Größe des (abgeschnittenen) Suchbaums kann grob abgeschätzt werden, bevor der Algorithmus abläuft.

Falls sich der Suchbaum im Level i um m_i verzweigt, dann ist die Anzahl der Blätter im Suchbaum etwa

$$\prod_{i=1}^{n} m_i.$$

Die Verzweigungsgrade m_i lassen sich leicht für einen zufälligen Pfad im Suchbaum leicht bestimmen.

Experimente zeigen, daß die Schätzungen im allgemeinen ganz gut sind.

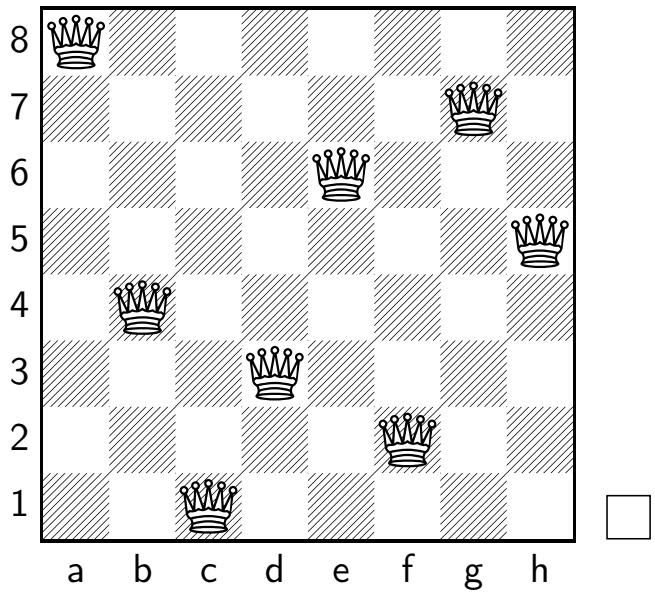
Es können mehrere Schätzungen kombiniert werden (aber nicht zu viele).

Abschätzung der Suchbaumgröße

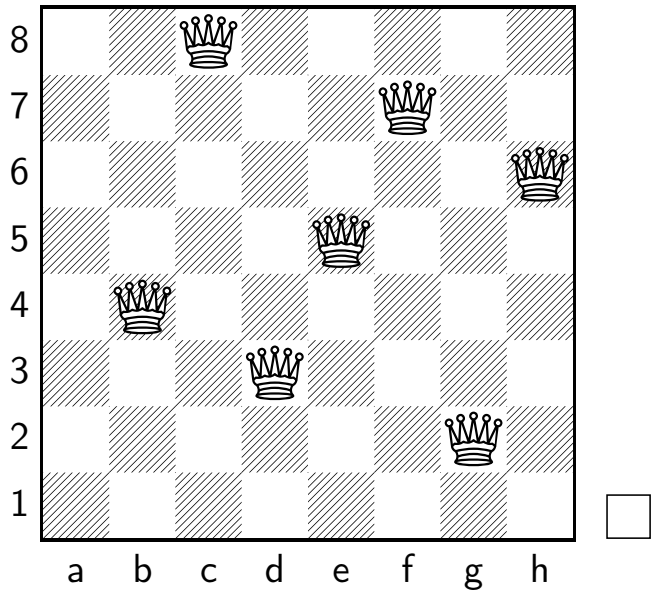
Estimate:
$k := 1$; $m := 1$; $r := 1$;
repeat forever do
  $T_k := \{\, x_k \in S_k \mid B_k(x_1, \ldots, x_k) \,\}$
  if $|T_k| = 0$ **then return** m;
  $r := r \cdot |T_k|$;
  $m := m + r$;
  $k := k + 1$;
  $x_k :=$ zufälliges Element $\in T_k$
od

Dieser Algorithmus schätzt die Suchbaumgröße der rekursiven oder iterativen Backtrackalgorithmen.

Beispiel



Das 8-Damen-Problem



Wir verwenden den Abschätzungsalgorithmus. Es gibt jeweils 8, 5, 3, 3, 2, 1, 1 Möglichkeiten, die nächste Dame zu setzen. Das ergibt geschätze 2689 Positionen im Suchbaum.

Ein Programm für das 8-Damen-Problem mit Backtracking:

```
#include⟨stdio.h⟩
#define N 14
int x[N], c[N], d1[2 * N − 1], d2[2 * N − 1];

int main(void)
{
  int k = 0, i;
  while(k ≥ 0) {
    i = x[k];
    if(i == N) k−−, i = x[k]++, c[i] = d1[k − i + N − 1] = d2[i + k] = 0;
    else if(!c[i] && !d1[k − i + N − 1] && !d2[i + k])
      if(k == N/2 − 1) {
        for(i = 0; i ≤ k; i++) printf("%d ", x[i]);
        printf("\n");
        x[k]++;
      }
      else c[i] = d1[k − i + N − 1] = d2[i + k] = 1, k++, x[k] = 0;
    else x[k]++;
  }
  return 0;
}
```

Ist das Programm korrekt?

Abschätzung der Suchbaumgröße

Der Schätzwert für die Suchbaumgröße war 2689.

Lassen wir das Programm laufen, ergibt sich, daß 2056 gültige Positionen getestet werden.

Insgesamt gibt es 92 Lösungen.

Folgendes Programm schätzt die Suchbaumgröße automatisch:

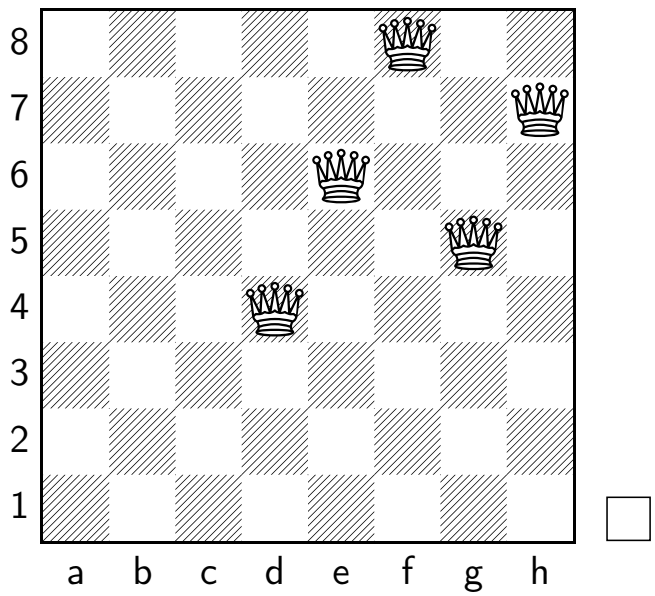
```
#include⟨stdio.h⟩
#include⟨stdlib.h⟩
#define N 28
int c[N], d1[2 * N − 1], d2[2 * N − 1], t[N];

int main(void)
{
  long k = 0, i, z; double r = 1, m = 1;
  while(1) {
    z = 0;
    for(i = 0; i < N; i++)
      if(!c[i] && !d1[k − i + N − 1] && !d2[i + k]) t[z] = i, z++;
    if(z == 0) break;
    m += r *= z;
    i = t[random()%z];
    printf("%2ld %2ld %g\n", i, z, m);
    c[i] = d1[k − i + N − 1] = d2[i + k] = 1;
    k++;
  }
  return 0;
}
```

Lassen wir dieses Programm laufen, ergibt sich folgende Ausgabe:

```
5   8  9
7   5  49
4   4  209
6   3  689
3   2  1649
```

Die mit Hand geschätzte Zahl war 2689 und die tatsächliche Suchbaumgröße 2056.

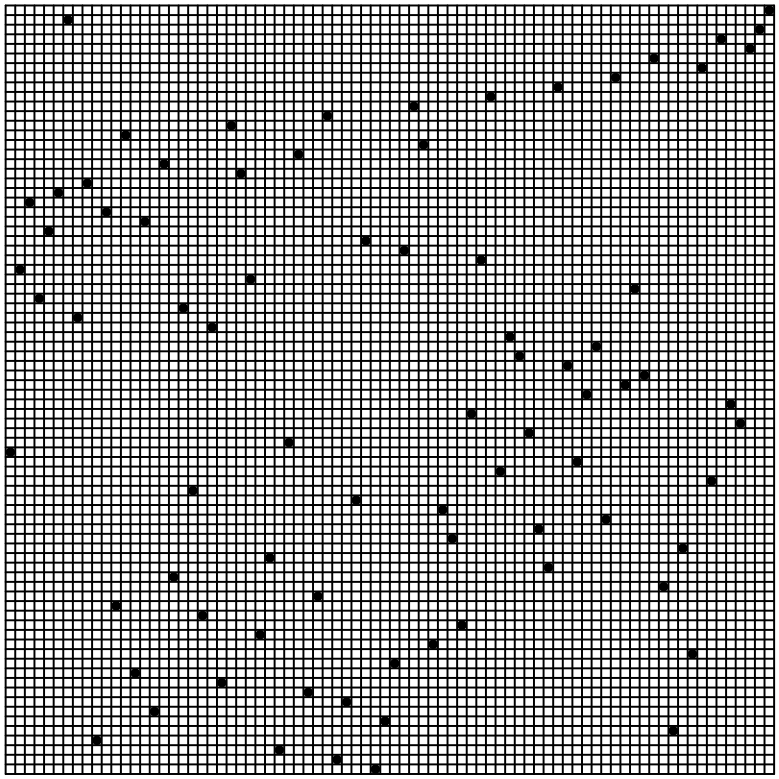


Dies ist die geratene Belegung des Programms.

Für $N = 16$ ergibt sich diese Ausgabe:

```
13 16 17
 1 13 225
10 10 2305
 4 10 23105
15  7 168705
 0  5 896705
12  7 5992705
 9  3 21280705
 6  3 67144705
 8  3 204736705
14  1 342328705
```

Lösung für $N = 80$:



Erfüllbarkeitsproblem

Die Lösung für $N = 80$ kann nicht mehr durch den normalen Backtracking-Algorithmus gefunden werden.

Sie wurde tatsächlich durch Lösung eines Erfüllbarkeitsproblem gefunden.

Die zugehörige Formel hat für jedes Feld eine Variable.

Es gibt Klauseln die sicherstellen, daß jede Zeile mindestens eine Dame enthält.

Es gibt Klauseln die verhindern, daß eine Spalte, Zeile oder Diagonale mehr als eine Dame enthält.

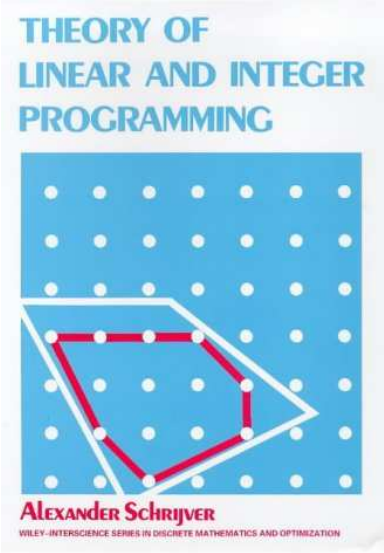
Erfüllbarkeitsproblem

Diese Formel sieht so aus:

$$\bigwedge_{i=1}^{n}\bigvee_{j=1}^{n} x_{ij} \wedge \bigwedge_{i,j=1}^{n}\Bigg(\bigwedge_{k=i+1}^{n}(\neg x_{ij} \vee \neg x_{kj})\wedge \bigwedge_{k=j+1}^{n}(\neg x_{ij} \vee \neg x_{ik})\wedge \bigwedge_{k=1}^{\min\{n-i,n-j\}}(\neg x_{ij} \vee \neg x_{i+k,j+k})\wedge \bigwedge_{k=1}^{\min\{n-i,j-1\}}(\neg x_{ij} \vee \neg x_{i+k,j-k})\Bigg)$$

Diese Formel ist in konjunktiver Normalform.

Literatur

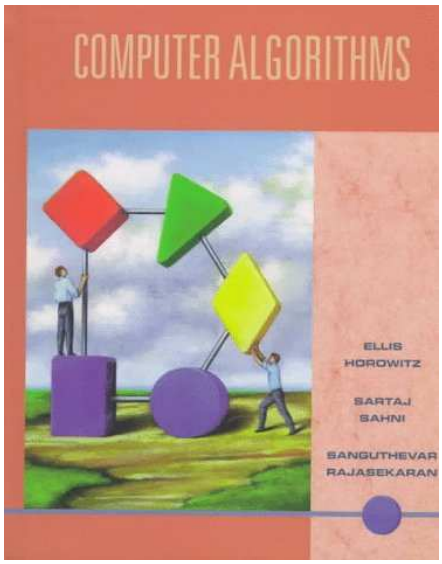


Dieses Buch behandelt eher die mathematische Theorie der Linearen und Ganzzahligen Programmierung und ist geeignet, sich in dieses Gebiet genauer einzuarbeiten.

Alexander Schrijver: *Theory of Linear and Integer Programming.* John Wiley & Sons.

Preis: US $60.90, Paperback.

Literatur



Dieses Buch ist anders aufgebaut als die meisten Algorithmenbücher. Es ist nach Lösungstechniken gegliedert. Insbesondere enthält es viel Information über das Lösen von *NP*-vollständigen Problemen.

Horowitz, Sahni, Rajasekaran: *Computer Algorithms.* Computer Science Press.

Preis: US $79.95, Hardcover (oder Rs 279 als Indian Edition).

Branch-and-Bound

Wir betrachten allgemein Probleme, deren Suchraum durch Bäume dargestellt werden kann.

Innerhalb des Suchraums suchen wir

1. nach einer Lösung oder
2. nach einer Lösung mit minimalen Kosten.

Beispiel: Beim N-Damen-Problem wird nach einer Lösung gesucht.

Beispiel: Beim Problem des *Handlungsreisenden* wird nach der kürzesten Rundreise durch gegebene Orte gesucht.

Branch-and-Bound

Die Klasse der Branch-and-Bound-Algorithmen, die wir jetzt betrachten, haben folgendes gemeinsam:

- Es gibt stets eine Menge von *lebendigen Knoten* des Suchraums, unter deren Unterbäumen noch nach Lösungen gesucht wird.
- Zu Beginn ist genau die Wurzel lebendig.
- Es wird stets ein lebendiger Knoten, der *E-Knoten* ausgesucht, und durch all seine Kinder zu den lebendigen Knoten hinzugefügt.
- Lebendige Knoten können durch *bounding functions* entfernt werden.
- Die lebendigen Knoten werden in einer geeigneten Datenstruktur gehalten.
- Zur Auswahl des E-Knotens gibt es verschiedene Strategien.

Branch-and-Bound

Werden die lebendigen Knoten mit einer Warteschlange (FIFO-queue) implementiert, dann entspricht dies einer Breitensuche.

Verwenden wir für die lebendigen Knoten dagegen einen Keller (LIFO-queue, stack), dann erhalten wir im wesentlichen eine Tiefensuche.

Allgemeiner können wir jedem Knoten x einen Wert $\hat{c}(x)$ zuordnen und als E-Knoten das lebendige x mit kleinstem $\hat{c}(x)$ wählen. Diese Variante wird *least-cost-search* (LC-search) genannt.

Terminierung

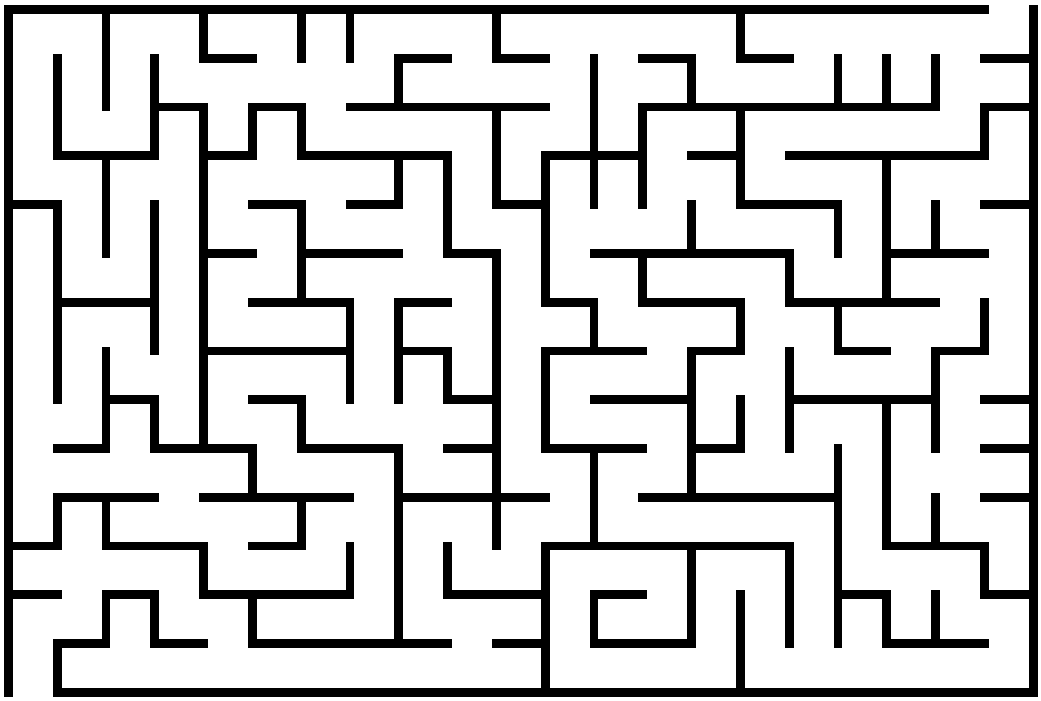
Falls der Suchbaum endlich ist, dann terminieren alle Branch-and-Bound-Verfahren.

Falls der Suchbaum unendlich ist, aber eine Lösung enthält, dann terminiert die Breitensuche.

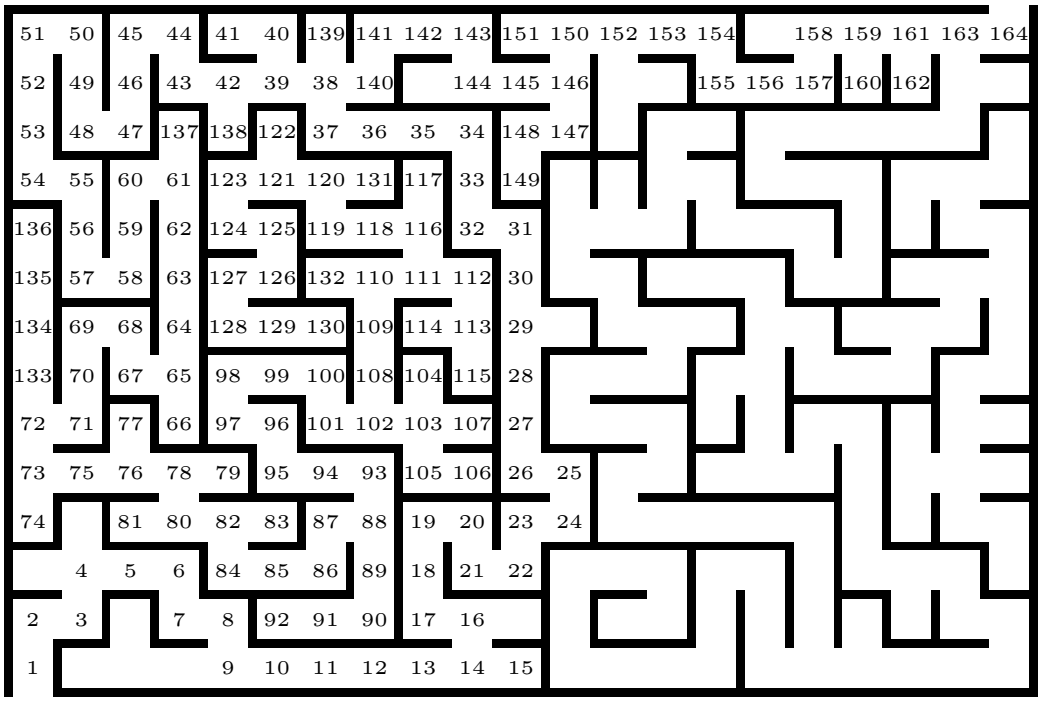
Die Tiefensuche muß nicht terminieren!

Bei LC-Suche kommt es auf die Funktion $\hat{c}(x)$ an. Wenn man den Aufwand x zu erreichen mit einbezieht, kann man Terminierung garantieren.

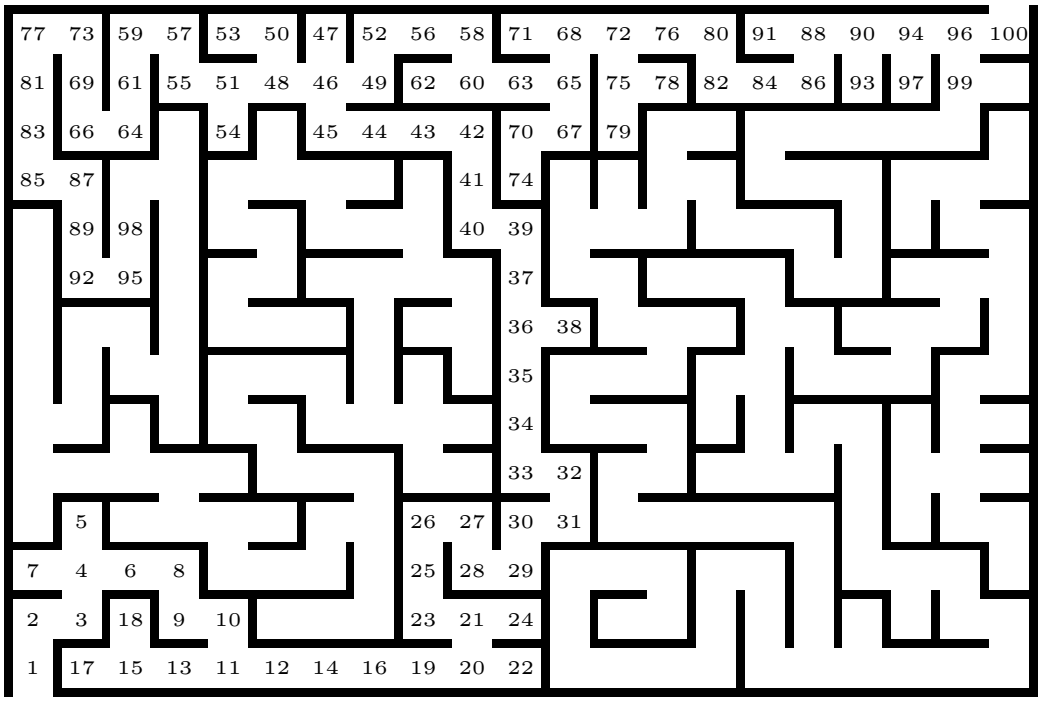
Beispiel



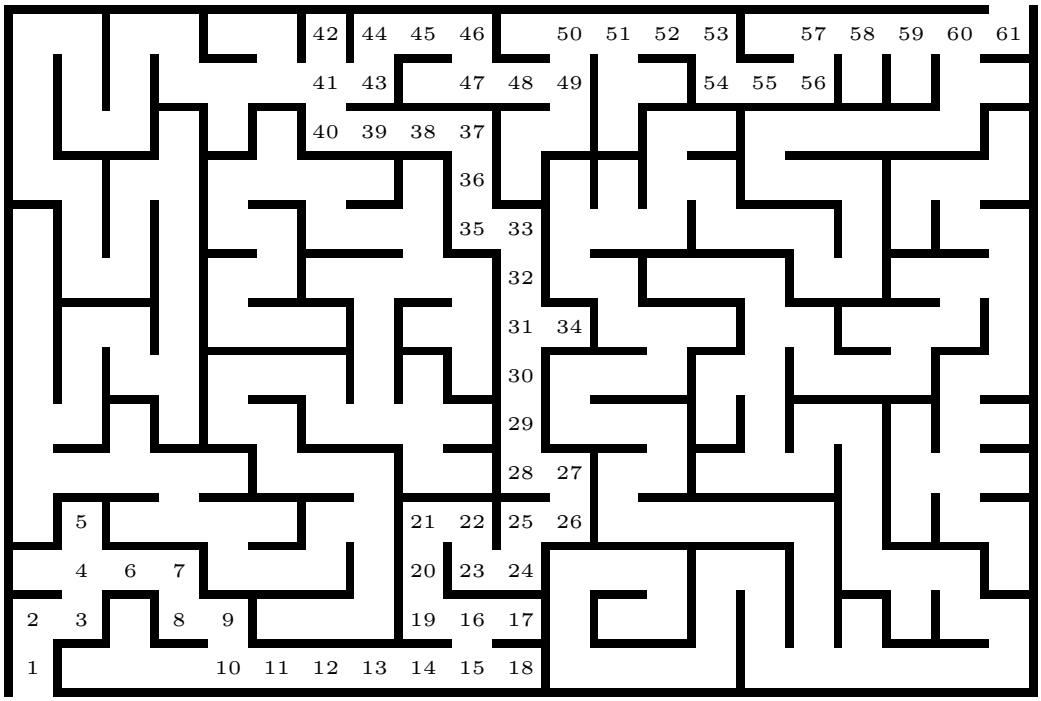
Beispiel: DFS



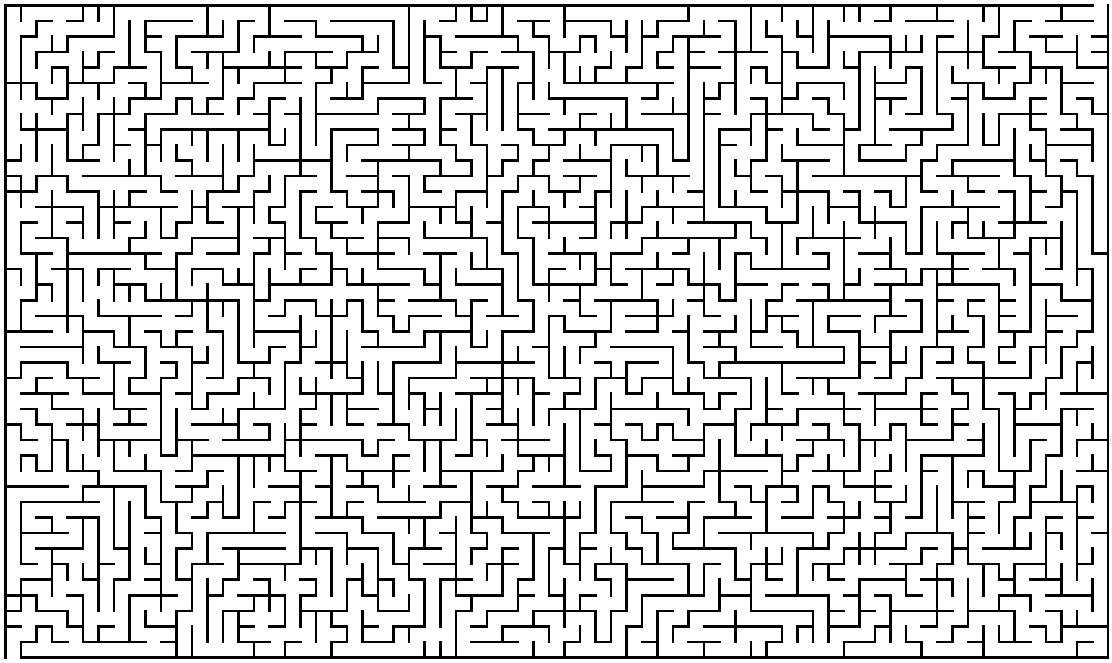
Beispiel: BFS



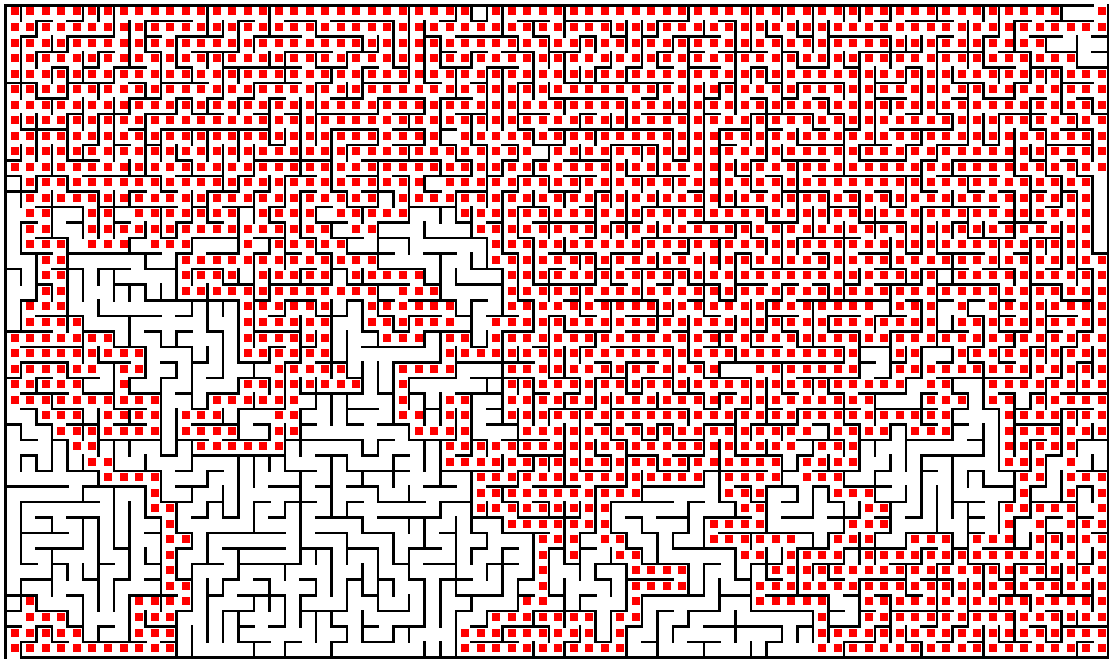
Beispiel: LC


42 44 45 46 50 51 52 53 57 58 59 60 61
41 43 47 48 49 54 55 56
40 39 38 37
36
35 33
32
31 34
30
29
28 27
5 21 22 25 26
4 6 7 20 23 24
2 3 8 9 19 16 17
1 10 11 12 13 14 15 18

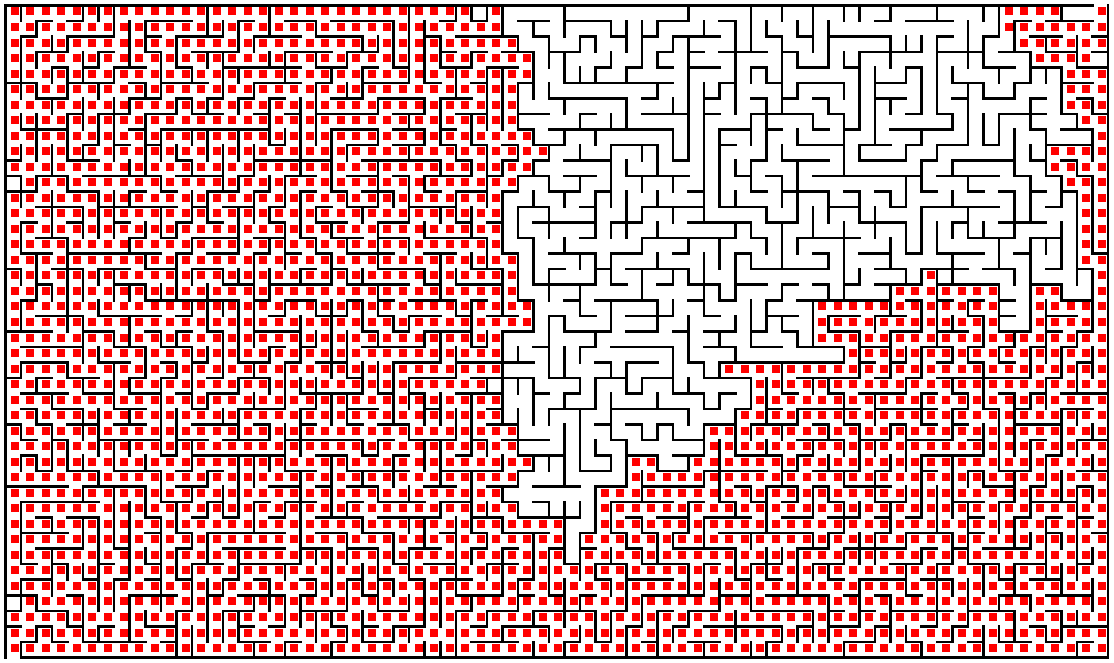
Beispiel



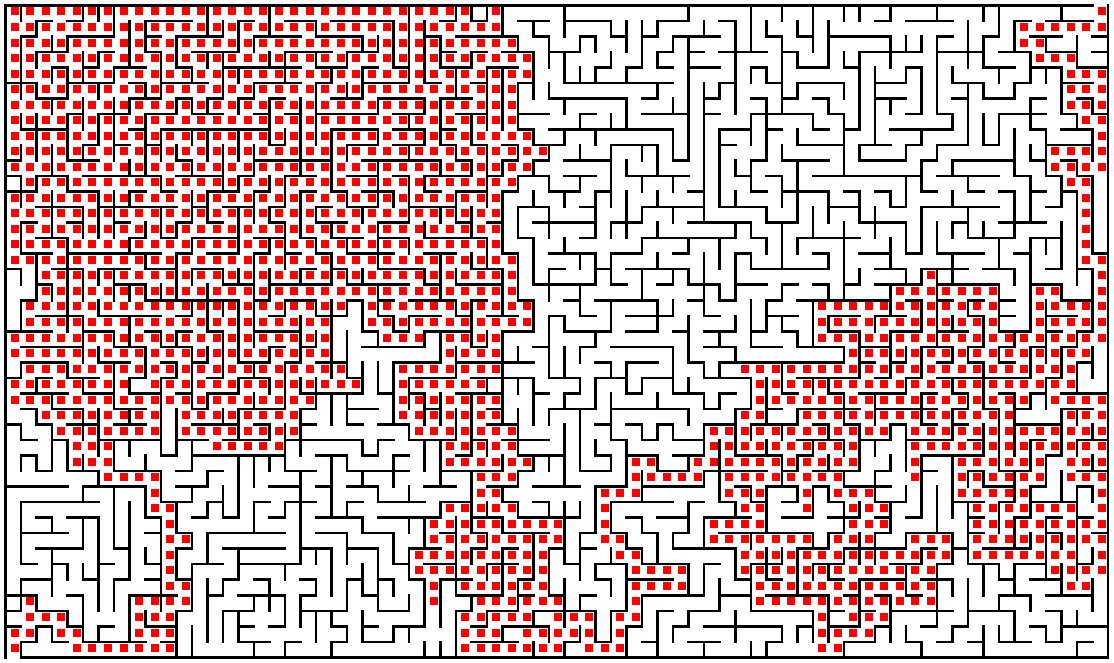
Beispiel: DFS



Beispiel: BFS



Beispiel: LC



LC-Suche

LC-Search(t):
if t is a solution **then return** t;
$L := \{t\}$; (Lebendige Knoten)
forever do
  $E := \min L$; (bezüglich $\hat{c}(x)$)
  $L := L - \{E\}$;
  for each child x of E **do**
    if x is a solution **then return** x;
    $L := L \cup \{x\}$
  od;
  if $L = \emptyset$ **then return** "no solution"
od

LC-Suche

Wie wählen wir $\hat{c}(x)$?

- Falls wir $\hat{c}(x) =$ Tiefe von x im Suchbaum wählen, erhalten wir wieder eine Breitensuche.
- Seien $c(x)$ die Entfernung zu einem Lösungsknoten im Unterbaum von x. Wählen wir $\hat{c}(x) = c(x)$, dann wird stets ein kürzester Pfad zu einer Lösung gefunden. Problem: $c(x)$ ist nicht bekannt.
- Sei $\hat{g}(x)$ eine Schätzung des Aufwands, eine Lösung im Unterbaum von x zu finden. Wir können $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ wählen, wobei f eine monoton steigende Funktion und $h(x)$ der bisherige Aufwand war, x von der Wurzel zu erreichen.
- $h(x) = 0$ führt zu bisweilen DFS-ähnlichem Verhalten.

Branch-and-Bound für Optimierungsprobleme

Eine bounding function diente dazu, Zweige des Suchbaums abzuschneiden, in welchen garantiert keine Lösung zu finden ist.

Wenn wir *Optimierungsprobleme* betrachten, dann kann diese Technik weiter verwendet werden und wir können sie sogar verfeinern: Ist eine *Schranke* L bekannt, dann können wir sogar Zweige abschneiden, in welchen nur Lösungen $> L$ vorkommen können (bei einem Minimierungsproblem).

Branch-and-Bound für Optimierungsprobleme

Woher erhalten wir eine geeignete Schranke?

- durch eine Heuristik
- durch Approximationsalgorithmen
- triviale Schranken (z.B. $-\infty$ oder ∞)
- durch Lösungen, die durch den Algorithmus selbst gefunden werden
- durch Kombinationen obiger Möglichkeiten


PRAXIS-WORKSHOP
Einsatz mathematischer Optimierung zur Planung logistischer Netzwerke
Donnerstag,
17.01.2019, 9:00 – 16:00 Uhr
RWTH – Informatik
Zentrum, Gebäude 3
Anmeldung unter:
www.inform-software.de/praxis-workshop
INFORM

Beispiel: Scheduling

Wir betrachten folgendes Problem:

Es müssen n Aufgaben bewältigt werden. Für jede dieser Aufgaben gibt es eine *deadline* d_i, eine *Strafe* p_i und eine *Bearbeitungsdauer* t_i.

Es dauert t_i Minuten, Aufgabe i zu lösen. Sie muß spätestens nach d_i Minuten gelöst sein, sonst wird eine Strafe von $\$p_i$ fällig.

Gesucht ist eine Reihenfolge, in der die Aufgaben bearbeitet werden sollen, um eine minimale Strafe zu bezahlen.

Beispiel

Es sind vier Aufgaben gegeben:

i	p_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

Der Lösungsraum besteht hier aus allen Plänen.

Eine optimale Reihenfolge ist leicht zu finden. Es genügt also, die Teilmengen von $\{1, 2, 3, 4\}$ zu betrachten, die wir ohne Strafe planen können.

Beispiel: Scheduling

Wir können mit einem leeren Plan beginnen, der natürlich eine legale (aber suboptimale) Lösung ist.

Wir verwenden die boolschen Variablen $x_1, \ldots, x_n$, um einen Plan darzustellen.

Wir können den Baum abschneiden, wenn der aktuelle Teilplan bereits teurer ist, als ein bekannter Plan.

Wir können auch eine obere Schranke aus einem Teilplan gewinnen: Wir addieren zu seiner Strafe noch die Strafen aller Aufgaben hinzu, die noch betrachtet werden müssen.

Ergebnisse

Wir betrachten 20 Aufgaben mit $d_i = 100i$. Die Strafen p_i werden zufällig zwischen 0 und 999 gewählt. Ebenso wählen wir t_i zufällig zwischen 0 und 299.

Soviele Knoten wurden im Experiment betrachtet:

- FIFO: 459009 Knoten
- LIFO: 459009 Knoten
- FIFO Branch-and-Bound: 9726 Knoten
- LIFO Branch-and-Bound: 348 Knoten
- LC Branch-and-Bound: 303 Knoten

Als $\hat{c}(x)$ wurde die Strafe des Knotens x verwendet.

```
#include⟨stdio.h⟩
#include"schedule.h"

int main(void) {
  int i, z = 0; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2; int k = E.k;
    z++;
    if(k == N) {
      for(i = 0; i < N; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p);
      continue;
    }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    insert(E1);
    if(E2.t ≤ jobs[k].d) insert(E2);
  }
  printf("Nodes: %d\n", z);
  return 0;
}
```

Implementierung für LIFO-Variante:

```
#include"schedule.h"

int l = 0;
struct node life[200000];

void insert(struct node n)
{
  if(l == 200000) exit(100);
  life[l++] = n;
}

struct node extract(void)
{
  return life[--l];
}

int isempty(void)
{
  return l == 0;
}
```

Ausgabe für LIFO:

```
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 -> 3717
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 -> 3728
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 -> 3670
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 0 -> 3681
.
.
.
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 -> 8966
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 -> 9348
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 -> 9359
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 -> 9024
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 -> 9035
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 -> 9417
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -> 9428
Nodes: 459009
```

```
#include⟨stdio.h⟩
#include"schedule.h"
int main(void) {
  int i, z = 0, upper = 100000; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2;
    int k = E.k, u = E.p; z++;
    for(i = k; i < N; i++) u += jobs[i].p;
    if(u < upper) upper = u;
    if(k == N) {
      for(i = 0; i < k; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p);
      continue;
    }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    if(E1.p ≤ upper) insert(E1);
    if(E2.p ≤ upper && E2.t ≤ jobs[k].d) insert(E2);
  }
  printf("Nodes: %d\n", z); return 0;
}
```

Ausgabe für FIFO Branch-and-Bound:

```
0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2914
0 1 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2903
0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 0 1 1 0 -> 2886
0 1 0 0 1 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 -> 2877
0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 0 1 0 -> 2783
0 1 0 0 1 0 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 -> 2772
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
Nodes: 9726
```

Ausgabe für LIFO Branch-and-Bound:

```
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 1 -> 3717
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 -> 3728
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 -> 3670
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 0 -> 3681
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 1 -> 3346
0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 0 1 0 -> 3357
0 1 0 1 1 1 0 1 0 1 0 0 1 1 0 1 0 0 1 1 -> 3342
0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 1 -> 3303
0 1 0 1 1 1 0 1 0 0 1 0 1 1 0 1 0 0 1 0 -> 3314
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 1 -> 3271
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 1 0 0 -> 3282
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 1 -> 2947
0 1 0 1 1 1 0 1 0 0 0 1 1 1 0 1 0 0 1 0 -> 2958
0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 1 -> 2842
0 1 0 1 1 0 1 0 1 0 0 1 1 1 0 1 0 0 1 0 -> 2853
0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 -> 2769
0 1 0 1 1 0 0 1 1 0 0 1 1 1 0 1 0 0 1 0 -> 2780
0 1 0 0 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1 1 -> 2655
0 1 0 0 1 1 1 0 1 0 1 1 1 1 0 1 0 0 1 0 -> 2666
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
Nodes: 348
```

Ausgabe für LC Branch-and-Bound:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 1 -> 2582
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 -> 2593
Nodes: 303
```

Hier wurden bereits im Inneren des Suchbaums so gute Schranken berechnet, daß im Endeffekt nur zwei Blätter besucht werden mußten.

Selbst für $N = 40$ bleibt der Suchbaum angenehm klein:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 -> 3583
Nodes: 19969
```

Eine bessere Schranke

Der LC-Suchalgorithmus berechnete eine Schranke für die zu zahlende Strafe, indem er die Strafen der noch nicht betrachteten Aufgaben zur bisher zu zahlenden Strafe addierte.

Dies ist eine sehr konservative Abschätzung.

Wir können diese Abschätzung leicht Verbessern:

Seien $x_1, \ldots, x_{k-1}$ schon festgelegt. Wir gehen $i = k, \ldots, n$ durch und dabei setzen $x_i = 1$, falls dadurch die Deadline für die Aufgabe i nicht verletzt wird.

Dies führt zu einer Menge von Aufgabe, die ohne Strafe durchgeführt werden können.

```
#include⟨stdio.h⟩
#include"schedule.h"
int main(void) {
  int i, z = 0, upper = 100000; struct node root = { 0, 0, 0} ;
  create_jobs(); insert(root);
  while(!isempty()) {
    struct node E = extract(), E1, E2;
    int k = E.k, u = E.p, ut = E.t; z++;
    for(i = k; i < N; i++)
      if(ut + jobs[i].t ≤ jobs[i].d) ut += jobs[i].t;
      else u += jobs[i].p;
    if(u < upper) upper = u;
    if(k == N) {
      for(i = 0; i < k; i++) printf("%d ", E.x[i]);
      printf("-> %d\n", E.p); continue; }
    E1 = E; E1.k = k + 1; E1.x[k] = 0; E1.p += jobs[k].p;
    E2 = E; E2.k = k + 1; E2.t += jobs[k].t; E2.x[k] = 1;
    if(E1.p ≤ upper) insert(E1);
    if(E2.p ≤ upper && E2.t ≤ jobs[k].d) insert(E2);
  }
  printf("Nodes: %d\n", z); return 0;
}
```

$N = 40$, LC-Suche ohne Heuristik:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 -> 3583
Nodes: 19969
```

$N = 40$, LC-Suche mit Heuristik:

```
0 1 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 1 0 0 1 1 0 \
1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 -> 3540
Nodes: 12867
```

Mit Heuristik: 5 mal weniger Speicherverbrauch

(Bei LIFO allerdings noch viel, viel, viel weniger.)