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

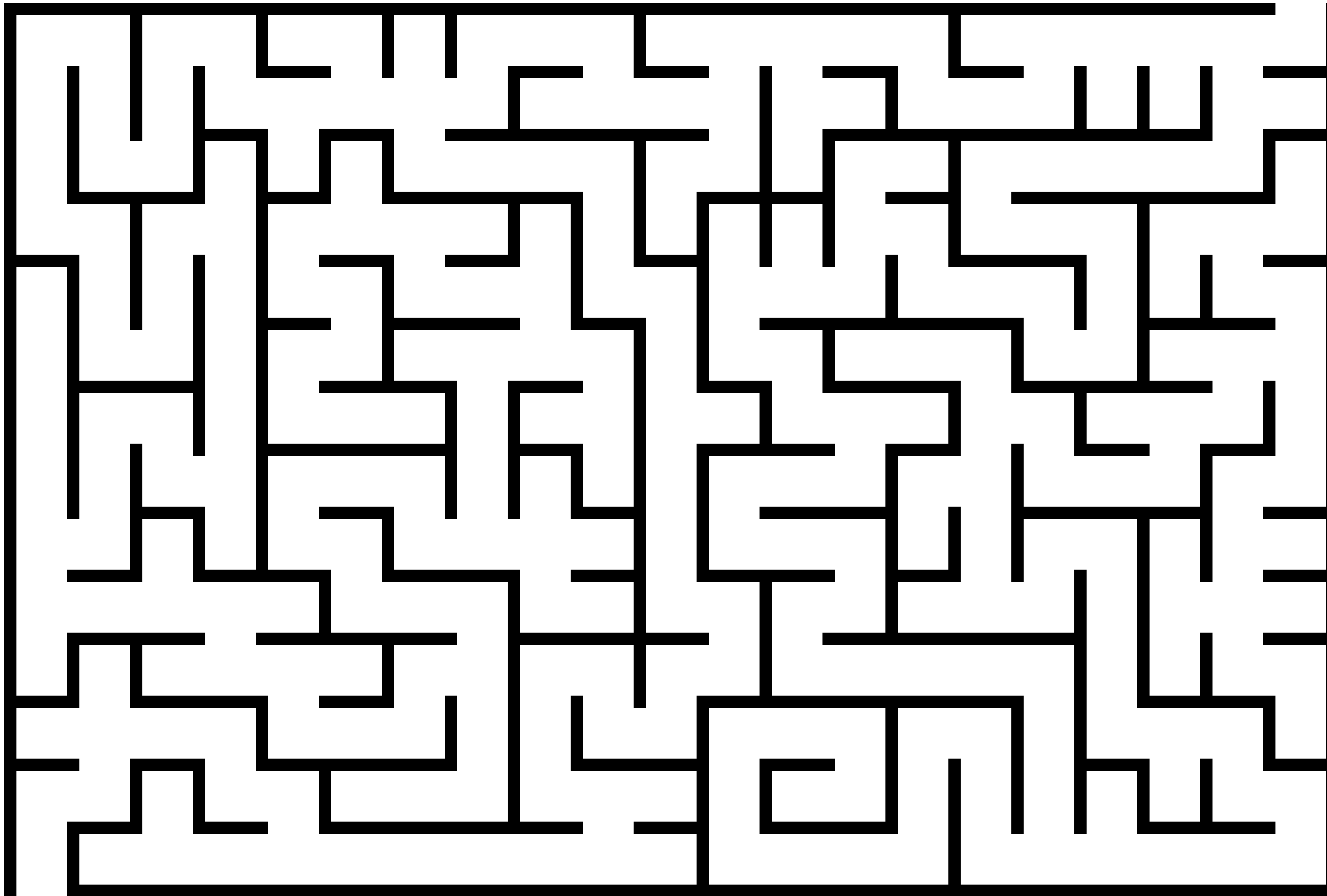
Falls der Suchbaum endlich ist, dann terminieren alle Branch-and-Bound-Verfahren.

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

51 50 45 44 41 40 139 141 142 143 151 150 152 153 154 158 159 161 163 164
52 49 46 43 42 39 38 140 144 145 146 155 156 157 160 162
53 48 47 137 138 122 37 36 35 34 148 147
54 55 60 61 123 121 120 131 117 33 149
136 56 59 62 124 125 119 118 116 32 31
135 57 58 63 127 126 132 110 111 112 30
134 69 68 64 128 129 130 109 114 113 29
133 70 67 65 98 99 100 108 104 115 28
72 71 77 66 97 96 101 102 103 107 27
73 75 76 78 79 95 94 93 105 106 26 25
74 81 80 82 83 87 88 19 20 23 24
4 5 6 84 85 86 89 18 21 22
2 3 7 8 92 91 90 17 16
1 9 10 11 12 13 14 15

Beispiel: BFS

77 73 59 57 53 50 47 52 56 58 71 68 72 76 80 91 88 90 94 96 100

81 69 61 55 51 48 46 49 62 60 63 65 75 78 82 84 86 93 97 99

83 66 64 54 45 44 43 42 70 67 79

85 87 41 74

89 98 40 39

92 95 37

36 38

35

34

33 32

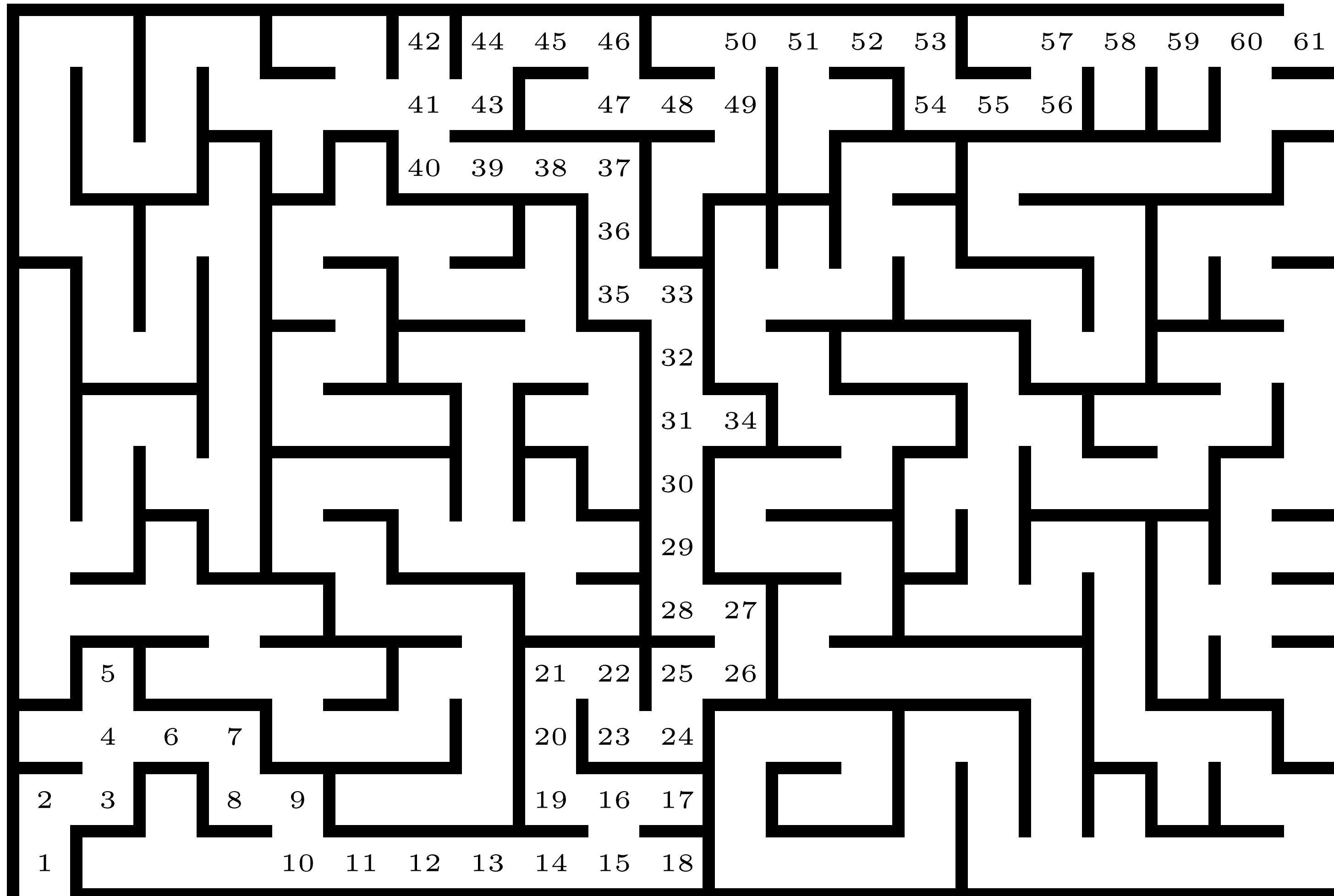
5 26 27 30 31

7 4 6 8 25 28 29

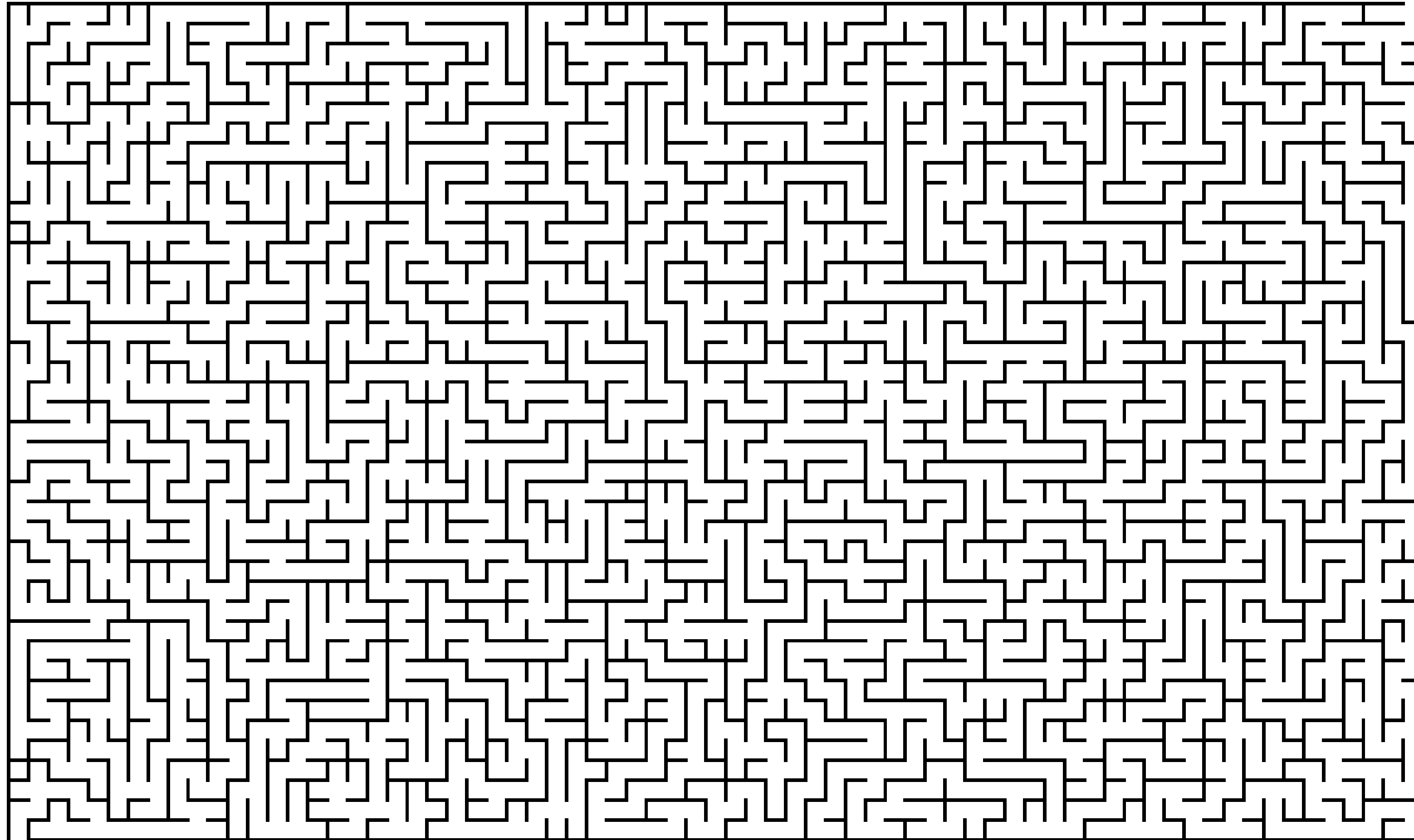
2 3 18 9 10 23 21 24

1 17 15 13 11 12 14 16 19 20 22

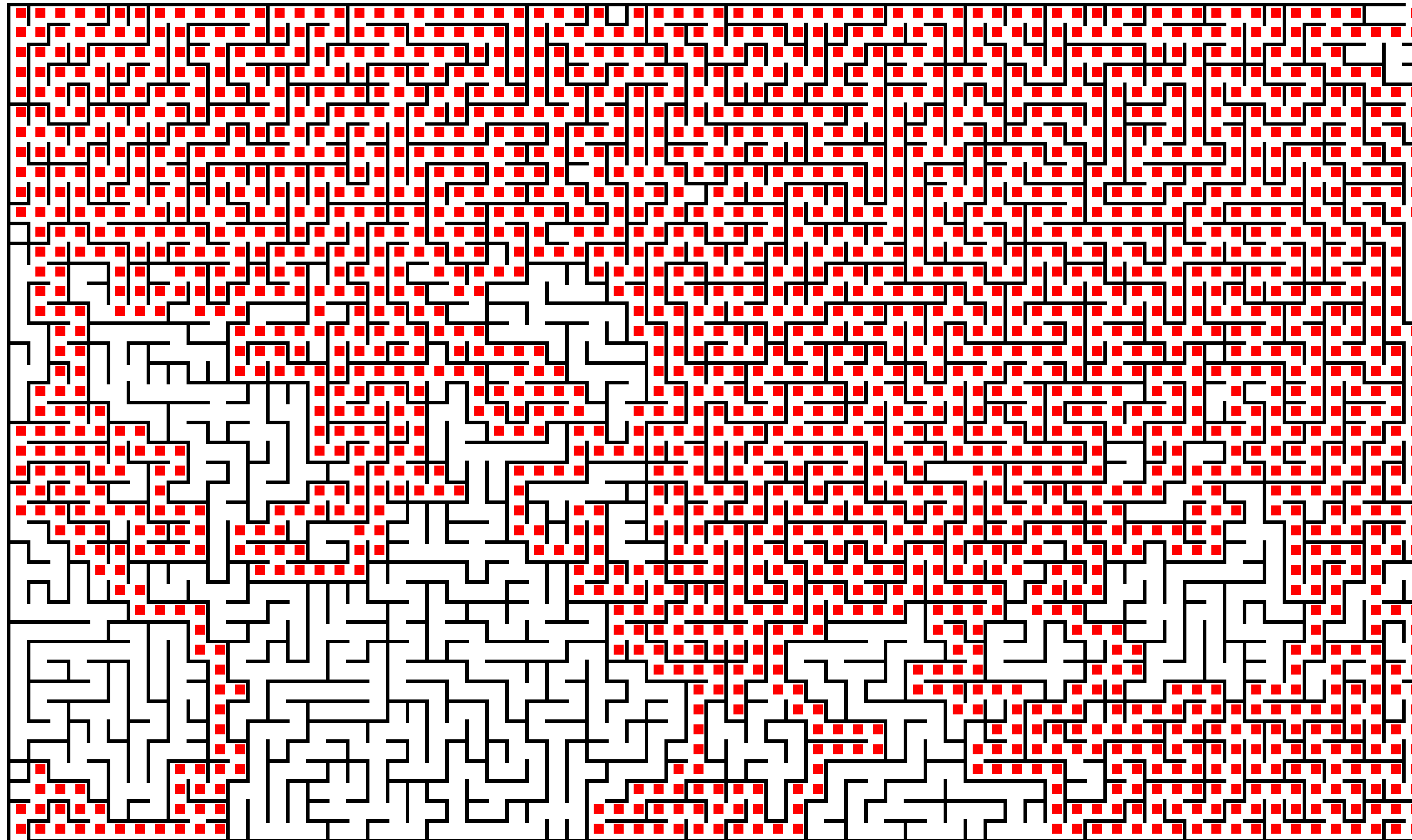
Beispiel: LC



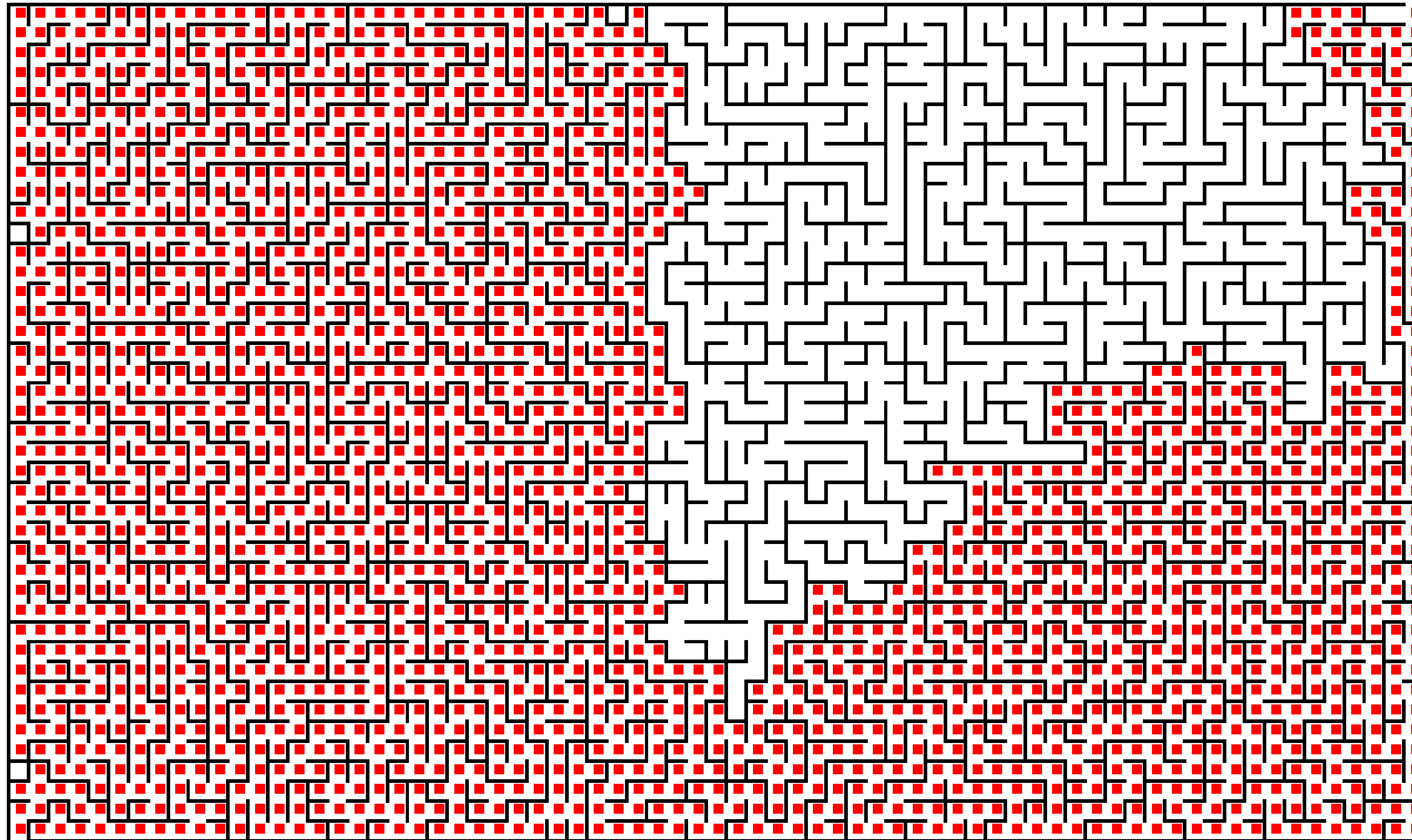
Beispiel



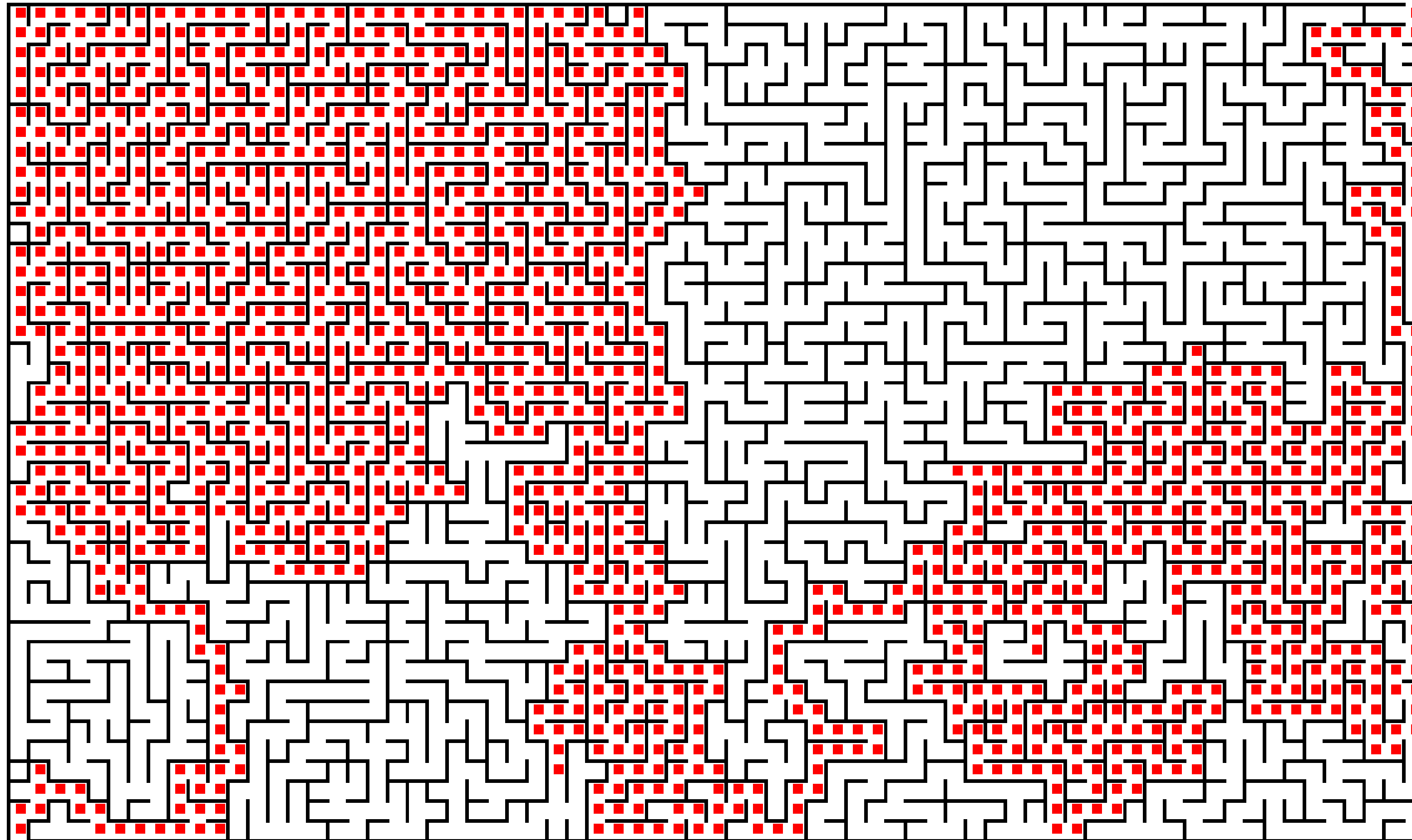
Beispiel: DFS



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

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 1 1 0 -> 8966
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 1 0 0 -> 9359
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 1 0 -> 9035
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 -> 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;
}
```