Chapter 4: Backtracking

Introduction

Backtracking is a more intelligent variation of the exhaustive-search technique. This approach makes it possible to solve some large instances of difficult combinatorial problems, though, in the worst case, we still face the same curse of exponential explosion encountered in exhaustive search.

The principal idea of backtracking is to construct solutions one component at a time and if no potential values of the remaining components can lead to a solution, the remaining components are not generated at all.

The state-space tree

Backtracking is based on the construction of a state-space tree whose nodes reflect specific choices made for a solution's components.

The root of this tree represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on. A node in a state-space tree is said to be *promising* if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called *nonpromising*. Leaves represent either nonpromising dead ends or complete solutions found by the algorithm.

Backtracking technique terminates a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants.

Backtracking algorithm – the first version

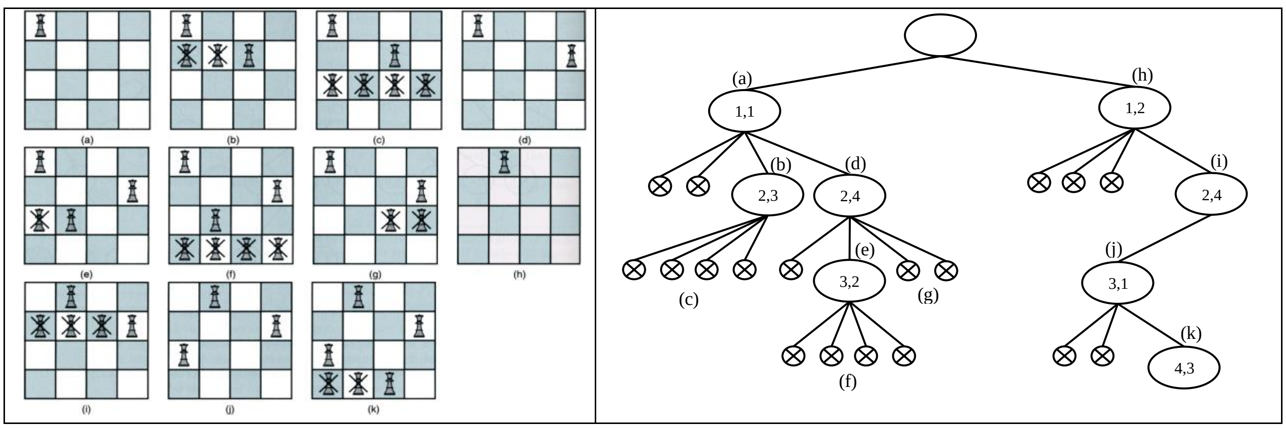
```
Backtracking(u) {
  if (promising(u))
    if (there is a solution at u)
      Output the solution;
    else
      for (each child v of u)
        Backtracking(v);
}
```

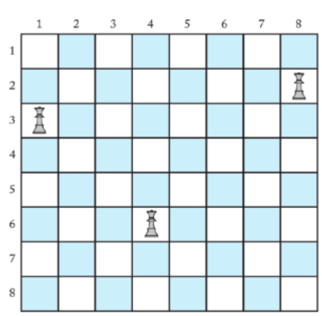
Backtracking algorithm – the second version

```
Backtracking(u) {
  for (each child v of u)
    if (promising(v))
      if (there is a solution at v)
        Output the solution;
      else
        Backtracking(v);
}
```

The n-Queens problem

The n-Queens is the problem of placing n chess queens on an $n \times n$ chessboard so that no two queens attack each other.





Algorithm

```
promising(i) {
  j = 1; flag = true;
  while (j < i && flag) {
    if (col[i] == col[j] || abs(col[i] - col[j]) == i - j)
      flag = false;
    j++;
  }
  return flag;
}

Version 1
n_Queens(i) {
  if (promising(i))
    if (i == n)
      print(col[1 .. n]);
    else
      for (j = 1; j ≤ n; j++) {
        col[i + 1] = j;
        n_Queens(i + 1);
      }
}
n_Queens(0);

Version 2
n_Queens(i) {
  for (j = 1; j ≤ n; j++) {
    col[i] = j;
    if (promising(i))
      if (i == n)
        print(col[1 .. n]);
      else
        n_Queens(i + 1);
  }
}
n_Queens(1);
```

The Knight's tour problem

A knight is placed on the first cell $\langle r_0, c_0 \rangle$ of an empty board of the size $n \times n$ and, moving according to the rules of chess, must visit each cell exactly once.

Note: Numbers in cells indicate move number of knight.

↙	4	←	3	↖	-2
5				2	-1
↓		♞		↑	0
6				1	1
↘	7		0	↗	2
-2	-1	0	1	2	

Algorithm

```
KnightTour(i, r, c) {
  for (k = 1; k ≤ 8; k++) {
    u = r + row[k];
    v = c + col[k];

    if ((1 ≤ u, v ≤ n) && (cb[u][v] == 0)) {
      cb[u][v] = i;

      if (i == n2)
        print(h);
      else
        KnightTour(i + 1, u, v);

      cb[u][v] = 0;
    }
  }
}

cb[r0][c0] = 1;
KnightTour(2, r0, c0);
```

Maze problem

A robot is asked to navigate a maze. It is placed at a certain position (the *starting* position) in the maze and is asked to try to reach another position (the *goal* position). Positions in the maze will either be open or blocked with an obstacle. Of course, the robot can only move to positions without obstacles and must stay within the maze.

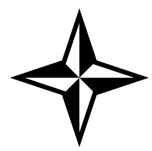
At any given moment, the robot can only move 1 step in one of 4 directions: North, East, South, and West.

The robot should search for a path from the starting position to the goal position (a solution path) until it finds one or until it exhausts all possibilities. In addition, it should mark the path it finds (if any) in the maze.

#	S	#	#	.	#
#	.	.	.	.	#
.	#	.	#	.	#
.	.	.	.	#	#
.	.	#	#	.	G
#	.	.	.	.	#
#	#	#	.	#	#

(a)

North

West  East

South

#	S	#	#	.	#
#	×	×	×	×	#
.	#	.	#	.	#
.	.	.	.	#	#
.	.	#	#	.	G
#	.	.	.	.	#
#	#	#	.	#	#

(b)

#	S	#	#	.	#
#	×	×	.	.	#
.	#	×	#	.	#
.	×	×	.	#	#
.	×	#	#	×	G
#	×	×	×	×	#
#	#	#	.	#	#

(c)

1	2	3	4	5	6
××#### #×#..# #×#..# #××#.# ###... G...##	××#### #×#..# #×#..# #××#.# ###... G...##	××#### #×#..# #×#..# #××#.# ###... G...##	××#### #×#..# #×#..# #××#.# ###... G...##	××#### #×#..# #×#..# #××#.# ###... G...##	××#### #×#..# #×#..# #×.#.# ###... G...##

Algorithm

```
bool Find_Path(r, c) {
  if ((r, c) ∉ Maze)
    return false;
  if (Maze[r][c] == 'G')
    return true;
  if (Maze[r][c] == 'x')
    return false;
  if (Maze[r][c] == '#')
    return false;

  Maze[r][c] = 'x';

  if (Find_Path(r - 1, c) == true)
    return true;
  if (Find_Path(r, c + 1) == true)
    return true;
  if (Find_Path(r + 1, c) == true)
    return true;
  if (Find_Path(r, c - 1) == true)
    return true;

  Maze[r][c] = '.';
  return false;
}
Find_Path(r0, c0);
```

Hamiltonian Circuit Problem

Algorithm

```
bool promising(int pos, int v) {
  if (pos == n && G[v][path[1]] == 0) // (3)
    return false;
  else
    if (G[path[pos - 1]][v] == 0)    // (2)
      return false;
    else
      for (int i = 1; i < pos; i++) // (4)
        if (path[i] == v)
          return false;
  return true;
}
Hamiltonian(bool G[1..n][1..n], int path[1..n], int pos) {
  if (pos == n + 1)
    print(path);
  else
    for (v = 1; v ≤ n; v++)
      if (promising(pos, v)) {
        path[pos] = v;
        Hamiltonian(G, path, pos + 1);
      }
}
path[1 .. n] = -1;
path[1] = 1;
Hamiltonian(G, path, 2);
```

Sum of Subsets Problem

Find a subset of a given set $\mathcal{W} = \{w_1, w_2, \dots, w_n\}$ of n positive integers whose sum is equal to a given positive integer t.

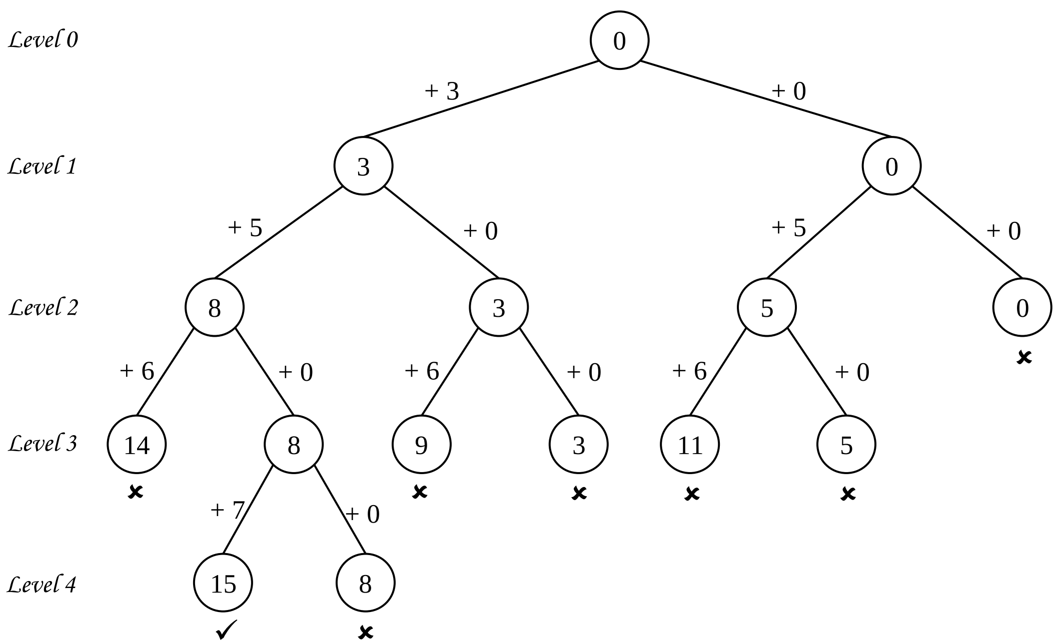
Note: It is convenient to sort the set's elements in increasing order. So, we will assume that

$$w_1 < w_2 < \cdots < w_n$$

The first approach

The solution S is a vector of the size n: $\{s_1, s_2, \dots, s_n\}$ where $s_i \in \{0,1\}$. For each $i \in \{1,2, \dots, n\}$, the value of s_i indicates whether w_i is in the subset or not.

Example: $\mathcal{W} = \{3,5,6,7\}$ và $t = 15$.



Algorithm

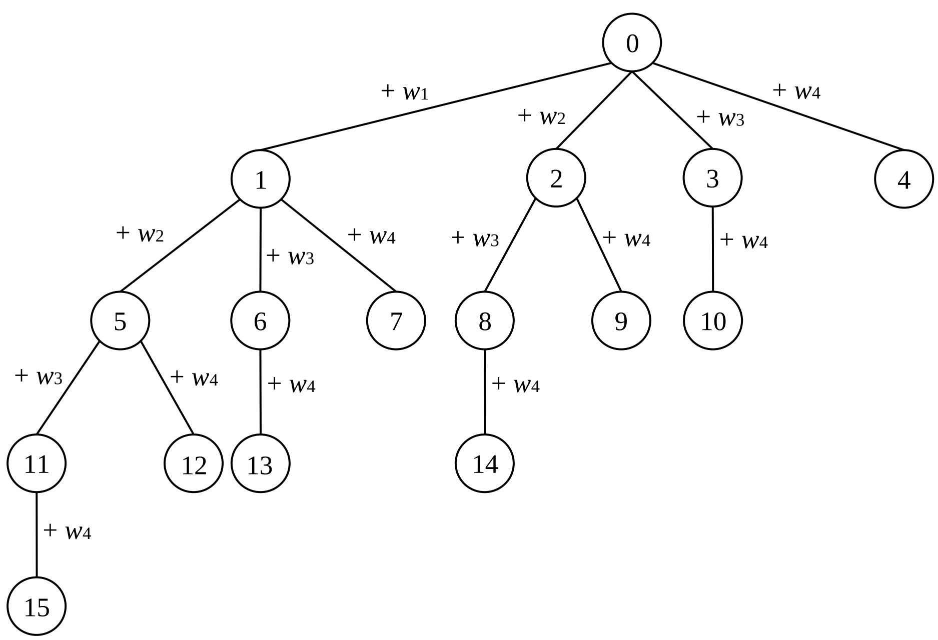
```
bool s[1 .. n] = {false};
total = ∑_{i=1}^{n} w[i];
sort(w);
if (w[1] ≤ t ≤ total)
   SoS(1, 0, total, w, s);
...
```

```
SoS(k, sum, total, w[1 .. n], s[1 .. n]) {
  if (sum == t)
    print(s);
  else
    if ((sum + total ≥ t) && (sum + w[k] ≤ t)) {
      s[k] = true;
      SoS(k + 1, sum + w[k], total - w[k], w, s);
      s[k] = false;
      SoS(k + 1, sum, total - w[k], w, s);
    }
}
```

The second approach

The solution S is the set of selected items.

Assume that initially the given set has 4 items $\mathcal{W} = \{w_1, w_2, w_3, w_4\}$. The state-space tree will be constructed as follows:



Algorithm

```
SoS(s[1 .. n], size, sum, start) {
   if (sum == t)
      print(s, size);
   else
      for (i = start; i ≤ n; i++) {
         s[size] = w[i];
         SoS(s, size + 1, sum + w[i], i + 1);
      }
}
s[1 .. n] = {0};
total = ∑_{i=1}^{n} w[i];
if (min(w) ≤ t && t ≤ total)
   SoS(s, 1, 0, 1);
```

Algorithm (upgraded version)

```
SoS(s[1 .. n], size, sum, start, total) {
   if (sum == t)
      print(s, size);
   else {
      lost = 0;
      for (i = start; i ≤ n; i++) {
         if ((sum + total - lost ≥ t) && (sum + w[i] ≤ t)) {
            s[size] = w[i];
            SoS(s, size+1, sum+w[i], i+1, total-lost - w[i]);
         }
         lost += w[i];
      }
   }
}
s[1 .. n] = {0};
total = ∑_{i=1}^{n} w[i];
sort(w);
if (w[1] ≤ t ≤ total)
   SoS(s, 1, 0, 1, total);
```