Chapter 5 Backtracking

- The Backtracking Technique
- The $n$-Queens Problem
- The Sum-of-Subsets Problem
  - Graph Coloring
- The 0-1 Knapsack Problem

Backtracking

- maze puzzle
- following every path in maze until a dead end is reached.
- go back to a fork and pursue another path
- $2^n$ cases (exponential-time in the worst case)
- if we can find some signs while generating subsets, we can avoid unnecessary labor
5.1 The Backtracking Technique

- A sequence of objects is chosen from a specified set.
- The sequence satisfies some criterion.
- n-Queens Problem.
- n Queens place in n×n chessboard s.t.
- no two Queens are in the same column, row, or diag.
  - sequence (n positions)
  - set (n×n positions)
  - criterion (no two queens threaten each other)
- Sequence generated by depth-first search.
  - Visiting root, left, right.
  - See Fig. 5.1 pp. 199.
5.1 The backtracking Technique

N-Queens Problem with n=4
4 queens on a 4×4 chessboard, no two queens threaten each other (same row, column, diag)
- assigning each queen a different row
- checking which column combinations yield solutions
- there are 4×4×4×4=256(44) candidate solutions

Fig. 5.2, state space tree
- a path from root to a leaf forms a candidate solution
- <i, j> node denotes to place i queen in row i column j
- depth first search to generate paths
5.1 The backtracking Technique

Figure 5.2 A portion of the state space tree for the instance of the n-Queens problem in which n = 4. The ordered pair $<i,j>$, at each node, means that the queen in the $i$th row is placed in the $j$th column. Each path from the root to a leaf is a candidate solution.

5.1 The backtracking Technique

Figure 5.3 Diagram showing that if the first queen is placed in column 1, the second queen cannot be placed in column 1 (a) or column 2 (b).
5.1 The backtracking Technique

- a general algorithm for backtracking

```c
void checknode (node v) ← root
{
    if (promising (v)) → possible lead to a solution
        if (there is a solution at v)
            write the solution;
        else → not form a solution yet
            for (each child u of v)
                checknode(v);
}
```

- See Example 5.1
- See pp. 204 last paragraph
5.1 The backtracking Technique

5.2 The n-Queens Problem
5.2 The n-Queens Problem

- promising(v): whether two queens are in the same column or diagonal
- col(i): the column where queen i in row i is located
- two queens i, k (note queens i, k are located in row i, k)
- in the same column $\Rightarrow$ $col(i) = col(k)$
- in the same diagonal
- see Fig. 5.6 pp. 206
  - $col(i) - col(k) = i - k$ or $k - i$
- See Algorithm 5.1
- See Table 5.1 for analysis, pp.209
Algorithm 5.1

The Backtracking Algorithm for the n-Queens Problem

Problem: Place n queens on an n \times n chessboard so that no two are in the same row, column, or diagonal.

Inputs: positive integer n.

Outputs: all possible ways n queens can be placed on an n \times n chessboard so that no two queens threaten each other. Each output consists of an array of integers \( \text{col} \) indexed from 1 to n, where \( \text{col}[i] \) is the column where the queen in the \( i \)-th row is placed.

void queens (index i)
{
    index j;
    if (promising(j))
        \( \text{col}[i] \) through \( \text{col}[n] \):
    else
        for (j = 1; j <= n; j++) {
            \( \text{col}[i] = j; \)
            if (can be positioned in each of the \( j \)-th column,
                \( \text{queens}(i+1); \)
        }
    bool promising (index i)
    {
        index k;
        bool switch;
        k = 1;
        switch = true;
        // Check if any queen threatens
        while (k < i & & switch) {
            if (\( \text{col}[i] = \text{col}[k] \) \| \( \text{abs} \text{col}[i] - \text{col}[k] = i - k \))
                switch = false;
        k += 1;
    }
    return switch;
}

5.2 The n-Queens Problem

Table 5.1

<table>
<thead>
<tr>
<th>n</th>
<th>Number of Nodes Checked by Algorithm 1</th>
<th>Number of Candidate Solutions Checked by Algorithm 2</th>
<th>Number of Nodes Checked by Backtracking</th>
<th>Number of Nodes Found Promising by Backtracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>341</td>
<td>24</td>
<td>61</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>19,173,961</td>
<td>40,320</td>
<td>15,721</td>
<td>2057</td>
</tr>
<tr>
<td>12</td>
<td>9.73 \times 10^{12}</td>
<td>4.79 \times 10^8</td>
<td>1.01 \times 10^8</td>
<td>8.56 \times 10^5</td>
</tr>
<tr>
<td>14</td>
<td>1.20 \times 10^{16}</td>
<td>8.72 \times 10^{10}</td>
<td>3.78 \times 10^5</td>
<td>2.74 \times 10^7</td>
</tr>
</tbody>
</table>

*Entries indicate numbers of checks required to find all solutions.

\( ^1 \) Algorithm 1 does a depth-first search of the state space tree without backtracking.

\( ^2 \) Algorithm 2 generalizes the n candidate solutions that place each queen in a different row and column.
5.4 The Sum-of-Subsets Problem

- given \( n \) positive integers (weights) \( w_1, w_2, ..., w_n \)
- given a positive integer \( W \)
- finding all subsets of \( n \) integers that sum to \( W \)
  - e.g., \( w_i + w_j + ... + w_k = W \)
  - See Example 5.2 pp. 214
5.4 The Sum-of-Subsets Problem

- create a state space tree
  - See Fig. 5.7 pp. 215
- each **left** edge denotes we **include** $w_i$ (weight $w_i$)
- each **right** edge denotes we **exclude** $w_i$ (weight 0)
- any path from root to a leaf forms a subset
- See Fig. 5.8 pp. 216
5.4 The Sum-of-Subsets Problem

- significant signs (backtracking)
  - sorting the weights in nondecreasing order
  - weight be the subtotal from root to node i at level I
  - weight + $w_{i+1} > W$
    - any descendant of node i will be nonpromising (because is $w_{i+1}$ the lightest weight remaining)
  - weight + all remaining items < W
    - any descendant of node i will be nonpromising

Example 5.4 and Fig. 5.9 pp. 217

See Algorithm 5.4
5.4 The Sum-of-Subsets Problem

![Diagram showing a tree structure for the sum-of-subsets problem]

Figure 5.4: The printed state space tree produced using backtracking in Example 5.4. Shaded at each node is the total weight included up to that node. The only solution is found at the shaded node. Each nonpromising node is marked with a cross.

Algorithm 5.4 The Backtracking Algorithm for the Sum-of-Subsets Problem

**Problem:** Given a positive integers (weights) and a positive integer W, determine all combinations of the integers that sum to W.

**Inputs:** positive integer \( n \), sorted (non-decreasing order) array of positive integers \( w \) indexed from 1 to \( n \), and a positive integer \( W \).

**Output:** all combinations of the integers that sum to \( W \).

```cpp
void sum_of_subsets(index i,
    int weight, int total)
{
    if (promising(i))
        if (weight == W)
            cout << "include[1] through include[\( i \)];";
        else
            include[i + 1] = "yes"; // Include \( w[i + 1] \);
            sum_of_subsets(i + 1, weight + \( w[i + 1] \), total - \( w[i + 1] \));
    include[i + 1] = "no"; // Do not include \( w[i + 1] \);
        sum_of_subsets(i + 1, weight, total - \( w[i + 1] \));
}

bool promising(index i);
{
    return (weight + total >= W) && (weight == W || weight + \( w[i + 1] \) <= W);
}
```
5.5 Graph Coloring

- m-coloring problem
  - finding all ways to color vertices using at most m colors s.t. no two adjacent vertices are the same color
  - Example 5.5 pp. 220
- state space tree
  - Fig. 5.12 pp. 222
  - each possible color is tried for vertex vi at level i s.t. no two adjacent vertices are the same color

→ sign (backtracking)

- See Algorithm 5.5
5.5 Graph Coloring

Figure 5.10 Graph for which there is no solution to the 2-Coloring Problem. A solution to the 3-Coloring Problem for this graph is shown in Example 5.5.
5.5 Graph Coloring

Algorithm 5.5

The backtracking algorithm for the $m$-coloring problem:

**Problem:** Determine all ways in which the vertices of an undirected graph can be colored using only $m$ colors, so that adjacent vertices are not the same color.

**Input:** positive integers $n$ and $m$, and an undirected graph containing $n$ vertices.

**Output:** all possible colorings of the graph, using at most $m$ colors, so that no two adjacent vertices are the same color. The output is an array `vcolor` indexed from 1 to $n$, where `vcolor[v]` is the color (an integer between 1 and $m$) assigned to the $v$th vertex.

```c
void m_coloring(Index i)
{
    color :=
    if (promising())
        if (i == n)
            color := color[i] through color[n];
        else
            for (color[i] = 1; color[i] <= m; color[i]++)
            { // Try every color for vcolor[i] = i
                m_coloring(i + 1);
                m_uncolor(i);
            }
    else
        for (color[i] = 1; color[i] <= m; color[i]++)
        { // Try every color for vcolor[i] = i
            m_coloring(i + 1);
            m_uncolor(i);
        }
}

bool promising(Index i)
{
    Index j;
    boolean switch;
    vcolor := new;
    j = 0;
    while (j < n & & switch)
        { // Check if an adjacent
            if (W[j][i] & & vcolor[j] == vcolor[i])
                if (vcolor is already this color)
                    switch := false;
            j := j + 1;
        }
    return switch;
}
```
5.7 The 0-1 Knapsack Problem

- using a state space tree like the Sum-of-Subset Problem
- each level is used to decide whether to include an item i or not
  - (left edge: include it and right edge: exclude it)
- each path from root to a leaf is a candidate solution
  - 0-1 Knapsack Problem is an optimization problem;
- we can't know the optimal solution until the search is over
5.7 0-1 Knapsack Problem

void checknode (node v) ← root
{
    if (value(v) is better than best)
        best=value(v) ← total profit up to v
    if (promising (v)) ←stealing more items
        for (each child u of v)
            checknode(v);
}
★ best: the value of best solution found so far

5.7 0-1 Knapsack Problem
- promising (v): whether we can steal more items into knapsack
- 1. weight >= W → nonpromising
- 2. greedy consideration
  - sort all items according to \( \frac{p_i}{w_i} \) in nondecreasing order
  - decide a node at level \( i \) be promising (expanding)
    - maxprofit : the best profit found so far
    - profit : the sum of profits up to the node
    - weight: the sum of weights up to the node
5.7 0-1 Knapsack Problem

- greedily grab item_{i+1}, item_{i+2}, ..., item_k (sorted)
  - s.t. total weight of item_1, ..., item_k above W
  - \( \text{totweight} = \text{weight} + \sum_{j=i+1}^{k} w_j \)

\[
\text{bound} = (\text{profit}^+ \sum_{j=i+1}^{k-1} p_j) + \left( \frac{W - \text{totweight}}{w_k} \right) \times p_k
\]

- bound \(\leq\) maxprofit \(\Rightarrow\) node i is nonpromising

- See Example 5.6 pp. 229
- See Algorithm 5.7
The End