Chapter 5

Backtracking
The idea

- A sequence of objects is chosen from a specified set so that the sequence satisfies some criterion
- Example: $n$-Queens problem
  - Sequence: $n$ positions on the chessboard
  - Set: $n^2$ possible positions
  - Criterion: no two queens can threaten each other
- Depth-first search of a tree (preorder tree traversal)
Depth first search

Figure 5.1  A tree with nodes numbered according to a depth-first search.
The algorithm

```c
void depth_first_tree_search (node v)
{
    node u;
    visit v;
    for (each child u of v)
        depth_first_tree_search(u);
}
```
4-Queens problem

- State space tree
If checking each candidate solution …

\[
\begin{align*}
[&< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 1 >] \\
[&< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 2 >] \\
[&< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 3 >] \\
[&< 1, 1 >, < 2, 1 >, < 3, 1 >, < 4, 4 >] \\
[&< 1, 1 >, < 2, 1 >, < 3, 2 >, < 4, 1 >]
\end{align*}
\]
Looking for signs for dead ends

(a)  

(b)
Backtracking

- Backtracking is the procedure whereby, after determining that a node can lead to dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child.

- We call a node nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it promising.
Backtracking

- Backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent.
- This is called pruning the state space tree.
The generic algorithm

```c
void checknode (node v)
{
    node u;

    if (promising(v))
        if (there is a solution at v)
            write the solution;

    else
        for (each child u of v)
            checknode(u);
}
```
4-Queens problem (1)
4-Queens problem (2)
4-Queens problem (3)
4-Queens problem (4)
4-Queens problem (5)
Pruned state space tree
Avoid creating nonpromising nodes

```c
void expand(node v)
{
    node u;

    for (each child u of v)
        if (promising(u))
            if (there is a solution at u)
                write the solution;
        else
            expand(u);
}
```
The \( n \)-Queens Problem

- Check whether two queens threaten each other:
  - \( Col(i) \) is the column where the queen in the \( i \)th row is located,

- Check diagonal
  - \( col(i) - col(k) = i - k \)
  - \( col(i) - col(k) = k - i \)
The algorithm

```cpp
void queens (index i)
{
    index j;
    if (promising (i))
    {
        if (i == n)
            cout << col [1] through col [n];
        else
            for (j = 1; j <= n; j++) // See if queen in
                col [i + 1] = j; // (i + 1) st row can be
                queens (i + 1); // positioned in each of
                                // the n columns.
    }
}
```
The algorithm (2)

```cpp
bool promising (index i) {
    index k;
    bool switch;
    k = 1;
    switch = true;  // Check if any queen threatens
    while (k < i && switch) {
        // queen in the ith row.
        if (col [i] == col [k] || abs (col [i] - col [k]) == i - k)
            switch = false;
        k++;
    }
    return switch;
}
```
Efficiency

- Checking the entire state space tree (number of nodes checked)

\[ 1 + n + n^2 + n^3 + \cdots + n^n = \frac{n^{n+1} - 1}{n - 1}. \]

- Taking the advantage that no two queens can be placed in the same row or in the same column

\[ 1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! \]

promising nodes
Comparison

Table 5.1 An illustration of how much checking is saved by backtracking in the $n$-Queens problem*

<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>$1.01 \times 10^7$</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>$3.78 \times 10^8$</td>
<td>$2.74 \times 10^7$</td>
</tr>
</tbody>
</table>

*Entries indicate numbers of checks required to find all solutions.
†Algorithm 1 does a depth-first search of the state space tree without backtracking.
‡Algorithm 2 generates the $n!$ candidate solutions that place each queen in a different row and column.
The Sum-of-Subsets Problem

Suppose that \( n = 5 \), \( W = 21 \), and

\[
\begin{align*}
    w_1 &= 5 \\
    w_2 &= 6 \\
    w_3 &= 10 \\
    w_4 &= 11 \\
    w_5 &= 16.
\end{align*}
\]

Because

\[
\begin{align*}
    w_1 + w_2 + w_3 &= 5 + 6 + 10 = 21, \\
    w_1 + w_5 &= 5 + 16 = 21, \text{ and} \\
    w_3 + w_4 &= 10 + 11 = 21,
\end{align*}
\]

the solutions are \( \{w_1, w_2, w_3\} \), \( \{w_1, w_5\} \), and \( \{w_3, w_4\} \).
State Space Tree

- $w_1 = 2$, $w_2 = 4$, $w_3 = 5$

Figure 5.7 • A state space tree for instances of the Sum-of-Subsets problem in which $n = 3$. 
When $W = 6$ and $w_1 = 2$, $w_2 = 4$, $w_3 = 5$
To check whether a node is promising

- Sort the weights in nondecreasing order
- To check the node at level $i$
  - $weight + w_{i+1} > W$
  - $weight + total < W$
When $W = 13$ and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$
The algorithm 5.4

```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";
            sum_of_subsets (i + 1, weight + w[i + 1], total - w[i + 1]);
            include [i + 1] = "no";
            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);
}
```
Time complexity

- The first call to the function `sum_of_subsets(0, 0, total)` where
  \[ total = \sum_{j=1}^{n} w[j] \]

- The number of nodes in the state space tree are
  \[ 1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} \]
Graph coloring

- The $m$-Coloring problem
  - Finding all ways to color an undirected graph using at most $m$ different colors, so that no two adjacent vertices are the same color.
Example

- 2-coloring problem
  - No solution!

- 3-coloring problem

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>color1</td>
</tr>
<tr>
<td>$v_2$</td>
<td>color2</td>
</tr>
<tr>
<td>$v_3$</td>
<td>color3</td>
</tr>
<tr>
<td>$v_4$</td>
<td>color2</td>
</tr>
</tbody>
</table>
Application: Coloring of maps

- **Planar** graph
  - It can be drawn in a plane in such a way that no two edges cross each other.

- To every map there corresponds a planar graph
Example (1)

- Map
Example (2)

- corresponded planar graph
The pruned state space tree
Algorithm 5.5 (1)

```c
void m_coloring (index i) {
    int color;
    if (promising (i))
        if (i == n)
            cout << vcolor [1] through vcolor [n];
        else
            for (color = 1; color <= m; color++){
                vcolor [i + 1] = color;
                m_coloring (i + 1);
            }
    }
```
Algorithm 5.5 (2)

```cpp
bool promising (index i) {
    index j;
    bool switch;
    switch = true;
    j = 1;
    while (j<i && switch){
        if (W[i][j] && vcolor[i] == vcolor[j])
            switch = false;
        j++;
    }
    return switch;
}
```
Algorithm 5.5 (3)

- The top level call to $m_{\text{coloring}}$
  - $m_{\text{coloring}}(0)$
- The number of nodes in the state space tree for this algorithm

\[ 1 + m + m^2 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1} \]
The Hamiltonian Circuits Problem

- The traveling sales person problem
  - Chapter 3: Dynamic programming
  - \( T(n) = (n-1)(n-2)2^{n-3} \)

- **Hamiltonian Circuit** (also called a tour)
  - Given a connected, undirected graph
  - A path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex
Example (1)

- Hamiltonian Circuit
  - $[v1, v2, v8, v7, v6, v5, v4, v3, v1]$
Example (2)

- No Hamiltonian Circuit!
Algorithm 5.6 (1)

```cpp
void hamiltonian (index i) {
    index j;
    if (promising (i))
        if (i == n-1)
            cout << vindex [0] through vindex [n - 1];
        else
            for (j = 2; j <=n; j++){
                vindex [i + 1] = j;
                hamiltonian (i + 1);
            }
    }
```
bool promising (index i) {
    index j;
    bool switch;
    if (i == n-1 && !W[vindex[n - 1]] [vindex [0]])
        switch = false;
    else if (i > 0 && !W[vindex[i - 1]] [vindex [i]])
        switch = false;
    else{
        switch = true;
        j = 1;
        while (j < i && switch){
            if (vindex[i] == vindex [j])
                switch = false;
            j++;
        }
    }
    return switch;
}
Algorithm 5.6 (3)

- The top level call to hamiltonian:
  - vindex [0] = 1; //Make v_1 the starting vertex.
  - hamiltonian (0);

- The number of nodes in the state space tree is

\[
1 + (n - 1) + (n - 1)^2 + ... + (n - 1)^{n-1} = \frac{(n-1)^n - 1}{n - 2}
\]
The 0-1 Knapsack Problem

- A state space tree exactly like the one in the Sum-of-Subsets problem
- This problem is different from the others discussed in this chapter in that it is an optimization problem.
- We do not know if a node contains a Solution until the search is over.
A general algorithm for backtracking in the case of optimization problems.

```c
void checknode (node v) {
    node u;
    if (value(v) is better than best)
        best = value(v);
    if (promising(v))
        for (each child u of v)
            checknode(u);
}
```

In the case of optimization problems, "promising" means that we should expand to the children.
Promising check

\[ \text{totweight} = \text{weight} + \sum_{j=i+1}^{k-1} w_j \]

The node at level \( k \) is the one that would bring the sum of the weights above \( W \)

\[ \text{bound} = \underbrace{\left( \text{profit} + \sum_{j=i+1}^{k-1} p_j \right)}_{\text{Profit from first } k-1 \text{ items taken}} + \left( W - \text{totweight} \right) \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}} \]

If \( \text{maxprofit} \) is the value of the profit in the best solution found so far, then a node at level \( i \) is nonpromising if

\[ \text{bound} \leq \text{max profit} \]
Example

- $n=4$
- $W = 16$
- The items is ordered according to $p_i/w_i$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$p_i$</th>
<th>$w_i$</th>
<th>$\frac{p_i}{w_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$40$</td>
<td>2</td>
<td>$20$</td>
</tr>
<tr>
<td>2</td>
<td>$30$</td>
<td>5</td>
<td>$6$</td>
</tr>
<tr>
<td>3</td>
<td>$50$</td>
<td>10</td>
<td>$5$</td>
</tr>
<tr>
<td>4</td>
<td>$10$</td>
<td>5</td>
<td>$2$</td>
</tr>
</tbody>
</table>
The pruned state space tree produced using backtracking
Algorithm 5.7: The Backtracking Algorithm for the 0–1 Knapsack (1)

```c
void knapsack (index i, int profit, int weight) {
    if (weight <= W && profit > maxprofit) {
        maxprofit = profit;
        numbest = i;
        bestset = include;
    }
    if (promising(i)) {
        include [i + 1] = "yes"; // Include w[i + 1].
        knapsack (i + 1, profit + p[i + 1], weight + w[i + 1]);
        include [i + 1] = "no"; // Do not include w[i+1]
        knapsack (i + 1, profit, weight);
    }
}
```
Algorithm 5.7: The Backtracking Algorithm for the 0–1 Knapsack (2)

```plaintext
bool promising (index i) {
    index j, k; int totweight; float bound;
    if (weight >= W)
        return false;
    else {
        j = i + 1;
        bound = profit;
        totweight = weight;
        while (j <= n && totweight + w[j] <= W){
            totweight = totweight + w[j];
            bound = bound + p[j]; j++;
        }
        k = j;
        if (k <= n)
            bound = bound + (W - totweight) * p[k]/w[k];
        return bound > maxprofit;
    }
}
```
Exercises

- 16
- 22
- 23
- 26
- 30