Chapter 8

Dynamic Programming

Fibonacci sequence

- Fibonacci sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- $F_i = \begin{cases} \ i \ & \text{if } i \leq 1 \\ F_{i-1} + F_{i-2} & \text{if } i \geq 2 \end{cases}$
- Solved by a recursive program:

  - Much replicated computation is done.
  - It should be solved by a simple loop.

Dynamic Programming

- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.

The shortest path

- To find a shortest path in a multi-stage graph:
- Apply the greedy method:
  - the shortest path from $S$ to $T$: $1 + 2 + 5 = 8$
Shortest path in multistage graphs

- e.g.

Greedy method cannot lead to an optimal answer to this case: \((S, A, D, T)\) 1+4+18 = 23.

Dynamic Programming

- Dynamic programming approach (forward approach):

- \(d(B, T) = \min\{9+d(D, T), 5+d(E, T), 16+d(F, T)\}\)
  \(= \min\{9+18, 5+13, 16+2\} = 18.\)
- \(d(C, T) = \min\{2+d(F, T)\} = 2+2 = 4\)
- \(d(S, T) = \min\{1+d(A, T), 2+d(B, T), 5+d(C, T)\}\)
  \(= \min\{1+22, 2+18, 5+4\} = 9.\)
- The above way of reasoning is called backward reasoning.

Backward approach (forward reasoning)

- \(d(S, A) = 1\)
  \(d(S, B) = 2\)
  \(d(S, C) = 5\)
- \(d(S, D) = \min\{d(S, A)+d(A, D), d(S, B)+d(B, D)\}\)
  \(= \min\{1+4, 2+9\} = 5\)
- \(d(S, E) = \min\{d(S, A)+d(A, E), d(S, B)+d(B, E)\}\)
  \(= \min\{1+11, 2+5\} = 7\)
- \(d(S, F) = \min\{d(S, A)+d(A, F), d(S, B)+d(B, F)\}\)
  \(= \min\{2+16, 5+2\} = 7\)
\[ d(S,T) = \min \{ d(S,D) + d(D,T), d(S,E) + d(E,T), d(S,F) + d(F,T) \} \]
\[ = \min \{ 5+18, 7+13, 7+2 \} \]
\[ = 9 \]

**Principle of Optimality**

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions \( D_1, D_2, \ldots, D_n \). If this sequence is optimal, then the last \( k \) decisions, \( 1 < k < n \) must be optimal.
- E.g. the shortest path problem
  - If \( i, i_1, i_2, \ldots, j \) is a shortest path from \( i \) to \( j \), then \( i_1, i_2, \ldots, j \) must be a shortest path from \( i_1 \) to \( j \)
- In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

**Dynamic Programming**

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards, i.e. beginning with the last decision
  - On the other hand if the relations are formulated using the backward approach, they are solved forwards.
- To solve a problem by using dynamic programming:
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

**Resource Allocation Problem**

- There are \( m \) resources available to \( n \) projects
- Profit \( p(i,j) \) denotes the profits attained through allocating \( j \) resources to project \( i \).
- Goal: Find an allocation that maximizes the total profit.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Project</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Resource allocation problem can be described as a multistage graph. 

\((i, j) : i \text{ resources allocated to projects } 1, 2, \ldots, j\)

e.g. node \(H = (3, 2) : 3 \text{ resources allocated to projects 1, 2.}\)

Find the longest path from \(S\) to \(T:\)
\((S, C, H, L, T), \ 8 + 5 + 0 + 0 = 13\)
2 resources allocated to project 1.
1 resource allocated to project 2.
0 resource allocated to projects 3, 4.

A multistage graph can describe all possible tours of a directed graph.
Find the shortest path:
\((1, 4, 3, 2, 1) \quad 5 + 7 + 3 + 2 = 17\)

Find the shortest path:
\((S, C, H, L, T)\)
Representation of a node

- Suppose that we have 6 vertices in the graph.
- We can combine \{1, 2, 3, 4\} and \{1, 3, 2, 4\} into one node.

(3, 4, 5, 6) means that the last vertex visited is 3 and the remaining vertices to be visited are (4, 5, 6).

DP Approach

- Let \( g(i, S) \) be the length of a shortest path starting at vertex \( i \),
go through all vertices in \( S \) and terminating at vertex 1.
- The length of an optimal tour:
  \[ g(1, V - \{1\}) = \min_{i \in V} \{c_{in} + g(k, V - \{1, k\})\} \]
- The general form:
  \[ g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \]
- Time complexity:
  \[ n + \sum_{k=1}^{n} (n-1)(n-2)(n-k) = O(n^2 2^n) \]

Longest common subsequence (LCS) problem

- A string: \( A = b a c a d \)
- A subsequence of \( A \): deleting 0 or more symbols from \( A \) (not necessarily consecutive).
  e.g. \( ad, ac, bac, acad, bacad, bcd \).
- Common subsequences of \( A = b a c a d \) and
  \( B = a c b a d c b : ad, ac, bac, acad \)
- The longest common subsequence (LCS) of \( A \) and
  \( B : a c a d \).

The LCS algorithm

- Let \( A = a_1 a_2 \ldots a_m \) and \( B = b_1 b_2 \ldots b_n \)
- Let \( L_{i,j} \) denote the length of the longest common subsequence of \( a_1 a_2 \ldots a_i \) and \( b_1 b_2 \ldots b_j \).
- \[ L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } a_i = b_j \\ \max\{ L_{i-1,j}, L_{i,j-1}\} & \text{if } a_i \neq b_j \end{cases} \]
- \( L_{0,0} = L_{0,j} = L_{i,0} = 0 \) for \( 1 \leq i \leq m, 1 \leq j \leq n \).
Dynamic programming approach for solving the LCS problem:

Time complexity is $O(mn)$ because there are $O(mn)$ entries, each of which is determined in constant time.

Tracing back in the LCS algorithm

- e.g. $A = b a c a d, B = a c c b a d c b$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>c</th>
<th>b</th>
<th>a</th>
<th>d</th>
<th>c</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

- After all $L_{i,j}$’s have been found, we can trace back to find the longest common subsequence of $A$ and $B$.

0/1 knapsack problem

- $n$ objects, weight $W_1, W_2, \ldots, W_n$
- profit $P_1, P_2, \ldots, P_n$
- capacity $M$

maximize $\sum P_i x_i$
subject to $\sum W_i x_i \leq M$
$x_i = 0$ or $1$, $1 \leq i \leq n$

- e.g.

<table>
<thead>
<tr>
<th>i</th>
<th>W_i</th>
<th>P_i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>30</td>
</tr>
</tbody>
</table>

$M = 10$

Multistage Graph

- 0/1 knapsack problem can be described by a multistage graph.
Partition Problem

- Given integers $x_1, x_2, \ldots, x_n$, and integer bound $B$, is there a subset of integers whose sum is exactly $B$?
- DP formulation: Define $f(i, K) = \text{"Y"}$ if there is a subset of integers $\{x_1, x_2, \ldots, x_i\}$ whose sum is exactly $K$; “N”, otherwise.
  
  $f(0, 0) = \text{"Y"}$
  $f(i, K) = \text{"Y"}$ if $f(i-1, K-x_i)$ or $f(i-1, K)$ is “Y”; “N”, otherwise.
  
  Find $f(n, B)$?

Multistage Graph

- 0/1 knapsack problem can be described by a multistage graph.

DP Approach

- The longest path represents the optimal solution:
  
  $x_1=0, x_2=1, x_3=1$
  
  $\sum P_i x_i = 20 + 30 = 50$
  
  - Let $f_i(Q)$ be the value of an optimal solution to objects $1, 2, 3, \ldots, i$ with capacity $Q$.
  - $f_i(Q) = \max \{ f_{i-1}(Q), f_{i-1}(Q-W_i)+P_i \}$
  - The optimal solution is $f_n(M)$. 
Optimal binary search trees

- e.g. binary search trees for 3, 7, 9, 12;

3
7
9
12

3
7
9
12

3
7
9
12

3
7
9
12

(a) (b) (c) (d)

Optimal binary search trees

- n identifiers: \(a_1 < a_2 < a_3 < \ldots < a_n\)
- \(P_i, 1 \leq i \leq n\): the probability that \(a_i\) is searched.
- \(Q_i, 0 \leq i \leq n\): the probability that \(x\) is searched

where \(a_i < x < a_{i+1}\) (\(a_0 = -\infty, a_{n+1} = \infty\)).

\[
\sum_{i=1}^{n} P_i + \sum_{i=1}^{n} Q_i = 1
\]

Dynamic Programming Approach

- Let \(C(i, j)\) denote the cost of an optimal binary search tree containing \(a_i, \ldots, a_j\).
- The cost of the optimal binary search tree with \(a_k\) as its root:

\[
C(1, n) = \min_{i=1}^{n} \left\{ P_i + \left[ \sum_{j=i}^{k-1} (P_j + Q_j) + C(1, k-1) \right] + \left[ Q_k + \sum_{j=k+1}^{n} (P_j + Q_j) + C(k+1, n) \right] \right\}
\]
General formula

\[ C(i, j) = \min_{k \in (i, j)} \left\{ P_k + \left[ Q_i + \sum_{m=1}^{k-1} (P_m + Q_m) + C(i, k-1) \right] \right\} + \left[ Q_j + \sum_{m=k+1}^{j} (P_m + Q_m) + C(k+1, j) \right] \]

\[ = \min_{k \in (i, j)} \left\{ C(i, k-1) + C(k+1, j) + Q_i + \sum_{m=1}^{k-1} (P_m + Q_m) \right\} \]

Matrix-chain multiplication

- \( n \) matrices \( A_1, A_2, \ldots, A_n \) with size
  \( p_0 \times p_1, p_1 \times p_2, p_2 \times p_3, \ldots, p_{n-1} \times p_n \)
  To determine the multiplication order such that number of scalar multiplications is minimized.
- To compute \( A_i \times A_{i+1} \), we need \( p_i \times p_{i+1} \) scalar multiplications.

E.g., \( n=4, A_1: 3 \times 5, A_2: 5 \times 4, A_3: 4 \times 2, A_4: 2 \times 5 \)
- \((A_1 \times A_2) \times A_3 \times A_4\) # of scalar multiplications:
  \(3 \times 5 \times 4 + 3 \times 4 \times 2 + 3 \times 2 \times 5 = 114\)
- \((A_1 \times A_2) \times (A_3 \times A_4)\) # of scalar multiplications:
  \(3 \times 5 \times 2 + 5 \times 4 \times 2 + 3 \times 2 \times 5 = 100\)
- \((A_1 \times A_2) \times (A_3 \times A_4)\) # of scalar multiplications:
  \(3 \times 5 \times 4 + 3 \times 4 \times 5 + 4 \times 2 \times 5 = 160\)

Computation relationships of subtrees

- E.g. \( n=4 \)

Time complexity: \( O(n^3) \)

When \( j-i=m \), there are \( (n-m) \) \( C(i, j) \)’s to compute.
Each \( C(i, j) \) with \( j-i=m \) can be computed in \( O(m) \) time.

\( O(\sum_{m=1}^{n-1} m(n-m)) = O(n^3) \)