COMP3121: Algorithms & Programming Techniques
Summary notes – Week 5

Gerald Huang

Updated: August 13, 2020

Contents

1 Lecture 7A – Dynamic Programming

Date: August 13, 2020

1.1 Introduction to dynamic programming

- **Key point**: build an optimal solution to the problem from optimal solutions for (carefully chosen) smaller size subproblems.
- Subproblems are chosen in a way which allows *recursive* construction of optimal solutions.
- Efficiency comes from the fact that the subproblems are only solved once and the respective solution is stored in a table for later recalls.

1.2 Activity selection

(*Activity selection*)

- **Key points**:
  - You have a list of activities a_i for $1 \le i \le n$.
  - Each item has a starting time s_i and finishing time f_i.
  - No two activities can take place simultaneously.
- **Task**: Find a subset of compatible activities of *maximal total duration.*

- **Solution**:
  - Begin by sorting these activities based on their finishing times in non-decreasing order, so assume that $f_1 \le f_2 \le f_3 \le \cdots \le f_n$.
  - For every $i \le n$, solve the subproblems.
  - **Subproblem** $P(i)$: find a subsequence σ_i of the sequence of activities $S = \langle a_1, a_2, \dots, a_i \rangle$ such that
    1. σ_i consists of non-overlapping activities.
    2. σ_i ends with activity a_i (this is to simplify recursion).
    3. σ_i is of maximal total duration among all subsequences of S_i which satisfy 1 and 2.
  - **Pre-processing stage**: Let $T(i)$ be the total duration of the optimal solution $S(i)$ of the subproblem $P(i)$.
  - **Base case**: For $S(1)$, choose a_1 and thus, $T(1) = f_1 - s_1$.
  - **Recursion**: Assume that we have solved subproblems for all $j < i$ and stored them in a table. Let

$$T(i) = \max\{T(j) + f_i - s_i \,:\, j < i, \quad f_j < s_i\}.$$

- **Optimality**: *Similar argument to the greedy solution.*
  Let the optimal solution of subproblem $P(i)$ be the sequence $S = \langle a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}}, a_{k_m} \rangle$ where $k_m = i$. Claim that the truncated subsequence $S' = \langle a_{k_1}, a_{k_2}, \dots, a_{k_{m-1}} \rangle$ is an optimal solution to subproblem $P(k_{m-1})$ where $k_{m-1} < i$.

  If there were a sequence S^* of a larger total duration than the duration of sequence S' that also ends with activity $a_{k_{m-1}}$, we obtain a sequence $\hat{S}$ by **extending** the sequence S^* with activity a_{k_m} and obtain a solution for subproblem $P(i)$ with a longer total duration than the total duration of sequence S, contradicting the optimality of S.
- **Final solution**: Let $T_{\max} = \max\{T(i) \,:\, i \le n\}$.
- **Time complexity**: Having sorted the activities by their finishing times in time $\mathcal{O}(n \log n)$, we need to solve n subproblems $P(i)$ for solutions ending in a_i. For each such interval a_i, we have to find all preceding compatible intervals and their optimal solutions (via a table). Thus, the time complexity is $\mathcal{O}(n^2)$.

1.3 Longest increasing subsequence

(*Longest increasing subsequence*)

- **Key points**:
  - You are given a sequence of n real numbers $A[1, \dots, n]$.
- **Task**: Determine a subsequence (not necessarily contiguous) of maximum length in which the values in the subsequence are strictly increasing.

- **Solution**:
  - For each $i \leq n$, solve the following subproblems.
  - **Subproblem** $P(i)$: find a subsequence of the sequence $A[1, \dots, i]$ of maximum length in which the values are strictly increasing and which ends with $A[i]$.
  - **Recursion**: Assume we have solved all the subproblems for $j < i$. Look for all $A[m]$ such that $m < i$ and such that $A[m] < A[i]$. Among those, pick m which produced the longest increasing subsequence ending with $A[m]$ and extend it with $A[i]$ to obtain the longest increasing subsequence which ends with $A[i]$.
  - **Final solution**: From all such subsequences, pick the longest one.
  - **Time complexity**: $\mathcal{O}(n^2)$.

1.4 Integer Knapsack Problem

(*Integer knapsack problem, duplicate items not allowed*)

- **Key points**:
  - You have n items (some of which can be identical).
  - Item I_i is of weight w_i and value v_i.
  - You also have a knapsack of capacity C.
- **Task**: Choose a combination of available items which all fit in the knapsack and whose value is as large as possible.

- **Solution**:
  - Begin by filling a table of size $n \times C$, row by row.
  - **Subproblem** $P(i, c)$: Choose from items $I_1, I_2, I_3, \dots, I_i$ a subset which fits in a knapsack of capacity c and is of the largest possible total value.
  - Fix now $i \leq n$ and $c \leq C$ and assume we have solved the subproblems for
    1. All $i < j$ and all knapsacks of capacities from 1 to C.
    2. For i, we have solved the problem for all capacities $d < c$.
  - **Recursion**: Look at optimal solutions $\text{opt}(i-1, c-w_i)$ and $\text{opt}(i-1, c)$.
  - If $\text{opt}(i-1, c-w_i) + v_i > \text{opt}(i-1, c)$, then $\text{opt}(i, c) = \text{opt}(i-1, c-w_i) + v_i$. Otherwise, $\text{opt}(i, c) = \text{opt}(i-1, c)$.
  - **Final solution**: Final solution will be given by $\text{opt}(n, C)$.

2 Lecture 7B – Dynamic Programming

Date: August 13, 2020

2.1 Matrix chain multiplication

(*Matrix chain multiplication*)

- **Key points**:
  - You are given a sequence of matrices $A_1 A_2 \dots A_n$.
- **Task**: Group them in such a way as to minimise the total number of multiplications needed to find the product matrix.

- **Note**: the total number of different distributions of brackets is equal to the number of binary trees with n leaves.
- The total number of different distributions of brackets satisfies the following recursion

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i).$$

  - We can group the matrices as $(A_1 A_2 A_3 \dots A_i)(A_{i+1} \dots A_n)$ for each $1 \leq i \leq n$. But this will run in exponential time!
- **Solution**:
  - **Subproblem** $P(i,j)$: Group matrices $A_i A_{i+1} \dots A_{j-1} A_j$ in such a way as to minimise the total number of multiplications needed to find the product matrix. Group such subproblems by the value of $j-i$ and perform a recursion on the value of $j-i$.
  - At each recursive step m, we solve all subproblems $P(i,j)$ for which $j-i=m$.
  - **Pre-processing stage**: Let $m(i,j)$ denote the minimal number of multiplications needed to compute the product $A_i A_{i+1} \dots A_{j-1} A_j$. Also, let the size of matrix A_i be $s_{i-1} \times s_i$.
  - **Recursion**: We examine all possible ways to place the (outermost) multiplication, splitting the product $(A_i \dots A_k)(A_{k+1} \dots A_j$.
    * Note that both $k-i < j-i$ and $j-(k+1) < j-i$. Thus we have the solutions of the subproblems $P(i,k)$ and $P(k+1,j)$ already computed and stored in slots $k-i$ and $j-(k+1)$, respectively, which precede slot $j-i$ we are presently filling.
  - **Recursion**: The recursion is

$$m(i,j) = \min\{\, m(i,k) + m(k+1,j) + s_{i-1}s_j s_k \quad : \quad i \leq k \leq j-1\},$$

    where $m(i,k)$ is the number of multiplications on the left matrix multiplication, $m(k+1,j)$ is the number of multiplications on the right matrix multiplication, and $s_{i-1}s_j s_k$ is the number of multiplications to multiply both left and right matrices.
- Recursion step is a brute force but the whole algorithm is not and there are only $\mathcal{O}(n^2)$ many subproblems.

2.2 Longest Common Subsequence

- **Key points**: You are given two sequences $S = \langle a_1, a_2, \dots, a_n \rangle$ and $S^* = \langle b_1, b_2, \dots, b_m \rangle$.
- **Task**: Find the longest common subsequence of S, S^*.

- **Solution**:
  - We first find the length of the longest common subsequence of S, S^*.
  - For all $1 \le i \le n$ and all $1 \le j \le m$, let $c(i,j)$ be the length of the longest subsequence of the truncated sequences
  $$S_i = \langle a_1, a_2, \dots, a_i \rangle, \quad S_j^* = \langle b_1, b_2, \dots, b_j \rangle.$$
  - **Recursion**: Fill the table row by row, so the ordering of subproblems is the lexicographical ordering (alphabetical ordering):
  $$c(i,j) = \begin{cases} 0 & i = 0, j = 0 \\ c(i-1, j-1) + 1 & i,j > 0, a_i = b_j \\ \max\{c(i-1,j), c(i,j-1)\} & i,j > 0, a_i \neq b_j \end{cases}.$$

2.3 Edit Distance

- **Key points**:
  - You are given two text strings A of length n and B of length m.
  - You want to transform A into B.
  - You are allowed to insert a character, delete a character and to replace a character with another one.
  - An insertion costs c_I, a deletion costs c_D and a replacement costs c_R.
- **Task**: Find the lowest total cost transformation of A into B.

- **Solution**:
  - **Subproblem** $P(i,j)$: Find the minimum cost $C(i,j)$ of transforming the sequence $A[1, \dots, n]$ into the sequence $B[1, \dots, j]$ for all $i \le n$ and all $j \le m$.
  - **Recursion**: Fill the table of solutions $C(i,j)$ for subproblems $P(i,j)$ row by row (transformation can proceed from left to right, we only operate on the ends of the string):
  $$C(i,j) = \min \begin{cases} c_D + C(i-1,j) \\ C(i,j-) + c_I \\ \begin{cases} C(i-1,j-1) & A[i] = B[j] \\ C(i-1,j-1) + c_R & A[i] \neq B[j] \end{cases} \end{cases}.$$
  - **Final solution**: Final solution is simply $\min\{C(i,j)\}$ for all $1 \le i \le n$ and $1 \le j \le m$.

2.4 Bellman Ford algorithm

- **Key points**: You have a directed weighted graph $G = (V, E)$ with weights which can be negative, but without cycles of negative total weight and a vertex $s \in V$.
- **Task**: Find the shortest path from vertex s to every other vertex t.

- **Solution**:
  - Since there are no negative weight cycles, the shortest path cannot contain cycles.
  - **Subproblem**: For every $v \in V$ and every i, let $\text{opt}(i, v)$ be the length of a shortest path from s to v which contains at most i edges.
  - Goal: find, for every vertex $t \in G$ the value of $\text{opt}(n-1, t)$ and the path which achieves such a length.
  - Denote the length of the shortest path from s to v among all paths which contain at most i edges by $\text{opt}(i, v)$ and let $\text{pred}(i, v)$ be the *immediate* predecessor of vertex v on such shortest path.
  - **Recursion**:

$$\text{opt}(i, v) = \min\{\text{opt}(i-1, v), \min_{p \in V}\{\text{opt}(i-1, p) + w(e(p, v))\}\}$$

$$\text{pred}(i, v) = \begin{cases} \text{pred}(i-1, v) & \min_{p \in V}\{\text{opt}(i-1, p) + w(e(p, v))\} \geq \text{pred}(i-1, v) \\ \arg\min_{p \in V}\{\text{opt}(i-1, p) + w(e(p, v))\} & \text{otherwise} \end{cases}$$

  - **Time complexity**: Computation of $\text{opt}(i, v)$ runs in time $\mathcal{O}(|V| \times |E|)$.

2.5 Floyd-Warshall algorithm

- Let $G = (V, E)$ be a directed weighted graph where $V = \{v_1, v_2, \dots, v_n\}$ and where weights $w(e(v_p, v_q))$ of edges $e(v_p, v_q)$ can be negative, but there are no negative weight cycles.
- Let $\text{opt}(k, v_p, v_q)$ be the length of the shortest path from a vertex v_p to a vertex v_q such that all intermediate vertices are among vertices $\{v_1, v_2, \dots, v_k\}$ for $1 \leq k \leq n$.
- **Recursion**: The recursion is

$$\text{opt}(k, v_p, v_q) = \min\{\text{opt}(k-1, v_p, v_q), \text{opt}(k-1, v_p, v_k) + \text{opt}(k-1, v_k, v_q)\}.$$

- We gradually *relax* the constraint that the intermediary vertices have to belong to $\{v_1, v_2, \dots, v_k\}$.
- **Time complexity**: Algorithm runs in time $|V|^3$.