Dynamic Programming

Reading: CLRS Chapter 15 & Section 25.2

CSE 2331 Algorithms
Steve Lai
Optimization Problems

- Problems that can be solved by dynamic programming are typically optimization problems.
- Optimization problems: Construct a set or a sequence of elements \( \{y_1, \ldots, y_k\} \) that satisfies a given constraint and optimizes a given objective function.
Problems and Subproblems

• Consider the sorting problem:
  Given a set of \( n \) elements, \( A = \{a_1, a_2, a_3, \ldots, a_n\} \), sort \( A \).

• Let \( P(i, j) \) denote the problem of sorting
  in \( A_{ij} = \{a_i, a_{i+1}, \ldots, a_j\} \), where \( 1 \leq i \leq j \leq n \).

• We have a class of similar problems, indexed by \( (i, j) \).

• The original problem is \( P(1,n) \).
Dynamic Programming: basic ideas (1)

- Problem: construct an optimal solution \((x_1, \ldots, x_k)\).
- There are several options for \(x_1\), say, \(op_1, op_2, \ldots, op_d\).
- Each option \(op_j\) leads to a subproblem \(P_j\): given \(x_1 = op_j\), find an optimal solution \((x_1 = op_j, x_2, \ldots, x_k)\).
- The best of these optimal solutions, i.e.,
  \[
  \text{Best}\left\{(x_1 = op_j, x_2, \ldots, x_k) : 1 \leq j \leq d\right\}
  \]
  is an optimal solution to the original problem.
Dynamic Programming: basic ideas (2)

- Apply the same reasoning to each subproblem, sub-subproblem, sub-sub-subproblem, and so on.
- Have a tree of the original problem (root) and subproblems.
- Dynamic programming works when these subproblems have many duplicates, are of the same type, and we can describe them using, typically, one or two parameters.
- The tree of problem/subproblems (which is of exponential size) now condenses to a smaller, polynomial-size graph.
- Now solve the subproblems from the "leaves".
Design a Dynamic Programming Algorithm

1. View the problem as constructing an opt. seq. \((x_1, \ldots, x_k)\).
2. There are several options for \(x_1\), say, \(op_1, op_2, \ldots, op_d\).
   Each option \(op_j\) leads to a subproblem.
3. Denote each problem/subproblem by a small number of parameters, the fewer the better.
4. Define the objective function to be optimized using these parameter(s).
5. Formulate a recurrence relation.
6. Determine the boundary condition and the goal.
7. Implement the algorithm.
Shortest Path

- Problem: Let $G = (V, E)$ be a directed acyclic graph (DAG). Let $G$ be represented by a matrix:

$$d(i, j) = \begin{cases} 
\text{length of edge } (i, j) & \text{if } (i, j) \in E \\
0 & \text{if } i = j \\
\infty & \text{otherwise}
\end{cases}$$

Find a shortest path from a given node $u$ to a given node $v$. 
Dynamic Programming Solution

1. View the problem as constructing an opt. seq. $\langle x_1, \ldots, x_k \rangle$. Here we want to find a sequence of nodes $\langle x_1, \ldots, x_k \rangle$ such that $(u, x_1, \ldots, x_k, v)$ is a shortest path from $u$ to $v$.

2. There are several options for $x_1$, say, $op_1, op_2, \ldots, op_d$. Each option $op_j$ leads to a subproblem.
   - Options for $x_1$ are the nodes $x$ which have an edge from $u$.
   - The subproblem corresponding to option $x$ is:
     Find a shortest path from $x$ to $v$. 
3. Denote each problem/subproblem by a small number of parameters, the fewer the better.

4. Define the objective function to be optimized using these parameter(s).
   - These two steps are usually done simultaneously.
   - Let \( f(x) \) denote the shortest distance from \( x \) to \( v \).

5. Formulate a recurrence relation.

\[
f(x) = \min \{d(x, y) + f(y) : (x, y) \in E\}, \text{ if } x \neq v \text{ and out-degree}(x) \neq 0.
\]
6. Determine the boundary condition.

\[ f(x) = \begin{cases} 
0 & \text{if } x = v \\
\infty & \text{if } x \neq v \text{ and } \text{out-degree}(x) = 0
\end{cases} \]

7. What's the goal?

- Our goal is to compute \( f(u) \).
- Once we know how to compute \( f(u) \), it will be easy to construct a shortest path from \( u \) to \( v \).
- I.e., we compute the shortest distance from \( u \) to \( v \), and then construct a path having that distance.

8. Implement the algorithm.
Computing $f(u)$ (version 1)

function shortest(x)

//computing $f(x)$//

global $d[1..n, 1..n]$

if $x = v$ then return (0)

elseif out-degree($x$) = 0 then return ($\infty$)

else return $\left( \min \{ d(x, y) + \text{shortest}(y) : (x, y) \in E \} \right)$

• Initial call: shortest($u$)

• Question: What's the worst-case running time?
Computing $f(u)$ (version 2)

function shortest(x)
    //computing $f(x)$//
    global $d[1..n, 1..n]$, $F[1..n]$, $Next[1..n]$
    if $F[x] = -1$ then
        if $x = \nu$ then $F[x] \leftarrow 0$
        elseif out-degree(x) = 0 then $F[x] \leftarrow \infty$
    else
        $F[x] \leftarrow \min \{d(x, y) + \text{shortest}(y) : (x, y) \in E\}$
        $Next[x] \leftarrow$ the node $y$ that yielded the min
    return($F[x]$)
Main Program

procedure shortest-path(u, v)
    // find a shortest path from u to v //
    global d[1..n, 1..n], F[1..n], Next[1..n]
    initialize Next[v] ← 0
    initialize F[1..n] ← −1
    SD ← shortest(u)  //shortest distance from u to v//
    if SD < ∞  then  //print the shortest path//
        k ← u
        while k ≠ 0 do {write(k);  k ← Next[k]}
Time Complexity

- Number of calls to shortest: $O(|E|)$
- How much time does shortest($x$) need for a particular $x$?
  - The first call: $O(1)$ + time to find $x$'s outgoing edges
  - Subsequent calls: $O(1)$ per call
- The over-all worst-case running time of the algorithm is
  - $O(|E|) \cdot O(1)$ + time to find all nodes' outgoing edges
  - If the graph is represent by an adjacency matrix: $O(|V|^2)$
  - If the graph is represent by adjacency lists: $O(|V| + |E|)$
Matrix-chain Multiplication

- Problem: Given $n$ matrices $M_1, M_2, \ldots, M_n$, where $M_i$ is of dimensions $d_{i-1} \times d_i$, we want to compute the product $M_1 \times M_2 \times \cdots \times M_n$ in a least expensive order, assuming that the cost for multiplying an $a \times b$ matrix by a $b \times c$ matrix is $abc$.

- Example: want to compute $A \times B \times C$, where $A$ is $10 \times 2$, $B$ is $2 \times 5$, $C$ is $5 \times 10$.
  - Cost of computing $(A \times B) \times C$ is $100 + 500 = 600$
  - Cost of computing $A \times (B \times C)$ is $200 + 100 = 300$
Dynamic Programming Solution

• We want to determine an optimal \((x_1, \ldots, x_{n-1})\), where
  \(x_1\) means which two matrices to multiply first,
  \(x_2\) means which two matrices to multiply next, and
  \(x_{n-1}\) means which two matrices to multiply lastly.

• Consider \(x_{n-1}\). (Why not \(x_1\)?)

• There are \(n-1\) choices for \(x_{n-1}\):
  \[
  (M_1 \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_n), \text{ where } 1 \leq k \leq n-1.
  \]

• A general problem/subproblem is to multiply \(M_i \times \cdots \times M_j\),
  which can be naturally denoted by \((i, j)\).
Dynamic Programming Solution

• Let $Cost(i, j)$ denote the minimum cost for computing $M_i \times \cdots \times M_j$.

• Recurrence relation:

$$Cost(i, j) = \min_{i \leq k < j} \left\{ Cost(i, k) + Cost(k + 1, j) + d_{i-1} d_k d_j \right\}.$$  

• Boundary condition: $Cost(i, i) = 0$ for $1 \leq i \leq n$.

• Goal: $Cost(1, n)$
Algorithm (recursive version)

function MinCost(i, j)

global d[0..n], Cost[1..n, 1..n], Cut[1..n, 1..n]

// initially, Cost[i, j] ← 0 if i = j, and Cost[i, j] ← −1 if i ≠ j

if Cost[i, j] < 0 then
    Cost[i, j] ← \text{min}_{i \leq k < j} \{ \text{MinCost}(i, k) + \text{MinCost}(k + 1, j) + d[i - 1] \cdot d[k] \cdot d[j] \}

Cut[i, j] ← the index k that gave the minimum in the last statement

return (Cost[i, j])
Algorithm (non-recursive version)

procedure MinCost

global $d[0..n]$, $Cost[1..n, 1..n]$, $Cut[1..n, 1..n]$

initialize $Cost[i, i] \leftarrow 0$ for $1 \leq i \leq n$

for $i \leftarrow n - 1$ to 1 do

for $j \leftarrow i + 1$ to $n$ do

\[
Cost[i, j] \leftarrow \min_{i \leq k < j} \left\{ Cost(i, k) + Cost(k + 1, j) + d[i - 1] \cdot d[k] \cdot d[j] \right\}
\]

$Cut[i, j] \leftarrow$ the index $k$ that gave the minimum in the last statement
Computing $M_i \times \cdots \times M_j$

function MatrixProduct($i, j$)

// Return the product $M_i \times \cdots \times M_j$ //

global $Cut[1..n, 1..n]$, $M_1, \ldots, M_n$

if $i = j$ then return($M_i$)

else

$k \leftarrow Cut[i, j]$

return$(\text{MatrixProduct}(i, k) \times \text{MatrixProduct}(k + 1, j))$
Main Program

global $d[0..n]$, $Cost[1..n, 1..n]$, $Cut[1..n, 1..n]$

global $M_1, \ldots, M_n$

Call MinCost (or MinCost(1, $n$), the recursive version)
Call MatrixProduct(1, $n$)

Time complexity: $\Theta(n^3)$
Longest Common Subsequence

- **Problem:** Given two sequences

\[ A = (a_1, a_2, \ldots, a_n) \]
\[ B = (b_1, b_2, \ldots, b_n) \]

find a longest common subsequence of \( A \) and \( B \).

- To solve it by dynamic programming, we view the problem as finding an optimal sequence \((x_1, x_2, \ldots, x_k)\) and ask: what choices are there for \( x_1 \)? (Or what choices are there for \( x_k \)?)
Approach 1  (not efficient)

- View \((x_1, x_2, \ldots)\) as a subsequence of \(A\).
- So, the choices for \(x_1\) are \(a_1, a_2, \ldots, a_n\).
- Let \(L(i, j)\) denote the length of a longest common subseq of \(A_i = (a_i, a_{i+1}, \ldots, a_n)\) and \(B_j = (b_j, b_{j+1}, \ldots, b_n)\).
- Recurrence: \(L(i, j) = \max_{i \leq k \leq n} \{ L(k + 1, \varphi(k, j) + 1) \} + 1\).
- \(\varphi(k, j)\) is the index of the first character in \(B_j\) equal to \(a_k\), or \(n + 1\) if no such character.
- Boundary condition: \(L(i, j) = 0\), if \(i = n + 1\) or \(j = n + 1\)
  \(L(i, n + 2) = -\infty\), \(1 \leq i \leq n + 1\)
- Running time: \(\Theta(n^3)\)
Approach 2  (not efficient)

- View \((x_1, x_2, \ldots)\) as a sequence of 0/1, where \(x_i\) indicates whether or not to include \(a_i\).
- The choices for each \(x_i\) are 0 and 1.
- Let \(L(i, j)\) denote the length of a longest common subseq of \(A_i = (a_i, a_{i+1}, \ldots, a_n)\) and \(B_j = (b_j, b_{j+1}, \ldots, b_n)\).
- Recurrence:  
  \[
  L(i, j) = \max \begin{cases} 
  1 + L(i + 1, \varphi(i, j) + 1) \\
  L(i + 1, j)
  \end{cases}
  \]
- \(\varphi(k, j)\) is as defined in approach 1.
- Boundary condition: same as in approach 1.
- Running time:  \(\Theta(n^2)\) + time for computing \(\varphi(1..n,1..n)\).
Approach 3

- View \((x_1, x_2, \ldots)\) as a sequence of decisions, where
  - \(x_1\) indicates whether to
    - include \(a_1 = b_1\) (if \(a_1 = b_1\))
    - exclude \(a_1\) or exclude \(b_1\) (if \(a_1 \neq b_1\))
- Let \(L(i, j)\) denote the length of a longest common subseq
of \(A_i = (a_i, a_{i+1}, \ldots, a_n)\) and \(B_j = (b_j, b_{j+1}, \ldots, b_n)\).
- Recurrence: \(L(i, j) = \begin{cases} 
1 + L(i + 1, j + 1) & \text{if } a_i = b_j \\
\max\{L(i + 1, j), L(i, j + 1)\} & \text{if } a_i \neq b_j 
\end{cases}\)
- Boundary: \(L(i, j) = 0, \text{ if } i = n + 1 \text{ or } j = n + 1\)
- Running time: \(\Theta(n^2)\)
All-Pair Shortest Paths

• Problem: Let $G(V, E)$ be a weighted directed graph. For every pair of nodes $u, v$, find a shortest path from $u$ to $v$.

• DP approach:
  • $\forall u, v \in V$, we are looking for an optimal sequence $(x_1, x_2, \ldots, x_k)$.
  • What choices are there for $x_1$?
  • To answer this, we need to know the meaning of $x_1$. 

Approach 1

- $x_1$: the next node.
- What choices are there for $x_1$?
- How to describe a subproblem?
Approach 2

- $x_1$: going through node 1 or not?
- What choices are there for $x_1$?
- Taking the backward approach, we ask whether to go through node $n$ or not.
- Let $D^k(i, j)$ be the length of a shortest path from $i$ to $j$ with intermediate nodes in $\{1, 2, \ldots, k\}$.
- Then, $D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\}$.

\[ D^0(i, j) = \begin{cases} 
\text{weight of edge (i, j)} & \text{if } (i, j) \in E \\
0 & \text{if } i = j \\
\infty & \text{otherwise}
\end{cases} \] (1)
Straightforward implementation

initialize $D^0[1..n, 1..n]$ by Eq. (1)

for $k \leftarrow 1$ to $n$ do

    for $i \leftarrow 1$ to $n$ do

        for $j \leftarrow 1$ to $n$ do

            if $D^{k-1}[i, k] + D^{k-1}[k, j] < D^{k-1}[i, j]$ then

                $D^k[i, j] \leftarrow D^{k-1}[i, k] + D^{k-1}[k, j]$

                $P^k[i, j] \leftarrow 1$

            else $D^k[i, j] \leftarrow D^{k-1}[i, j]$

            $P^k[i, j] \leftarrow 0$
Eliminate the $k$ in $D^k[1..n, 1..n], P^k[1..n, 1..n]$

- If $i \neq k$ and $j \neq k$:
  
  We need $D^{k-1}[i, j]$ only for computing $D^k[i, j]$.
  
  Once $D^k[i, j]$ is computed, we don't need to keep $D^{k-1}[i, j]$.

- If $i = k$ or $j = k$:  
  $D^k[i, j] = D^{k-1}[i, j]$.

- What does $P^k[i, j]$ indicate?

- Only need to know the largest $k$ such that $P^k[i, j] = 1$. 
Floyd's Algorithm

initialize $D[1..n, 1..n]$ by Eq. (1)
initialize $P[1..n, 1..n] ← 0$
for $k ← 1$ to $n$ do
  for $i ← 1$ to $n$ do
    for $j ← 1$ to $n$ do
      if $D[i, k] + D[k, j] < D[i, j]$ then
        $D[i, j] ← D[i, k] + D[k, j]$
        $P[i, j] ← k$
Sum of Subset

• Given a multiset of positive integers $A = \{a_1, a_2, \ldots, a_n\}$ and another positive integer $M$, determine whether there is a subset $B \subseteq A$ such that $\text{Sum}(B) = M$, where $\text{Sum}(B)$ means the sum of integers in $B$.

• This problem is NP-hard.