Dynamic Programming

Reading: CLRS Chapter 15 & Section 25.2

CSE 6331: 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.

- The closest pair problem is an optimization problem.

- The convex hull problem is an optimization problem.
Problems and Subproblems

- Consider the closest pair problem:
  Given a set of $n$ points, $A = \{p_1, p_2, p_3, \ldots, p_n\}$, find a closest pair in $A$.

- Let $P(i, j)$ denote the problem of finding a closest pair in $A_{ij} = \{p_i, p_{i+1}, \ldots, p_j\}$, 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

- 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 \((x_1 = op_j, x_{2j}, \ldots, x_{kj})\).
- The best of these optimal solutions, i.e.,

\[
\text{Best of } \left\{ \left( x_1 = op_j, x_{2j}, \ldots, x_{kj} \right) : 1 \leq j \leq d \right\}
\]

is an optimal solution to the original problem.
- DP works only if the \(P_j\) is a problem similar to the original problem.
Dynamic Programming: basic ideas

- 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 condensed 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. E.g., \(P(i, j), 1 \leq i \leq j \leq n\).
4. Define the objective function to be optimized using these parameter(s). E.g., \(f(i, j) = \) the optimal value of \(P(i, j)\).
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. \((x_1,\ldots,x_k)\).

   Here we want to find a sequence of nodes \((x_1,\ldots,x_k)\) 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.

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

7. What's the goal (objective)?

- 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 \( \nu \).
- I.e., we compute the shortest distance from \( u \) to \( \nu \), 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 $x = \nu$ 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 $x = v$ then $F[x] ← 0$
    
    elseif out-degree($x$) = 0 then $F[x] ← \infty$
  
  else
    
    $F[x] ← \min\{d(x, y) + \text{shortest}(y) : (x, y) \in E\}$

    $Next[x] ←$ 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: $O(|E|)$
- Is it $\Omega(|E|)$ or $\Theta(|E|)$?

- How much time is spent on shortest($x$) for any $x$?
  - 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: $O(|V| + |E|)$
Forward vs Backward approach
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 \(x_{n-1}\):
  \[
  \left( M_1 \times \cdots \times M_k \right) \times \left( M_{k+1} \times \cdots \times M_n \right), \text{ where } 1 \leq k \leq n-1.
  \]

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

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

- Recurrence relation:
  \[
  Cost(i, j) = \min_{i \leq k < j} \left\{ Cost(i, k) + Cost(k + 1, j) + d_{i-1} d_k d_j \right\}
  \]
  for $1 \leq i < j \leq n$.

- 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, $Cost[i, j] \leftarrow 0$ if $i = j$, and $Cost[i, j] \leftarrow -1$ if $i \neq j$

  if $Cost[i, j] < 0$
    then

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

      $Cut[i, j] \leftarrow$ the index $k$ that gave the minimum in the last statement

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

procedure MinCost

global \(d[0..n], \text{Cost}[1..n, 1..n], \text{Cut}[1..n, 1..n]\)

initialize \(\text{Cost}[i, i] \leftarrow 0\) for \(1 \leq i \leq n\)

for \(i \leftarrow n - 1\) to 1 do

for \(j \leftarrow i + 1\) to \(n\) do

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

\(\text{Cut}[i, j] \leftarrow \) 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($\text{MatrixProduct}(i, k) \times \text{MatrixProduct}(k + 1, j)$)

Time complexity: $\Theta(n^3)$
Paragraphing

- Problem: Typeset a sequence of words \( w_1, w_2, \ldots, w_n \) into a paragraph with minimum cost (penalty).
  
  Words: \( w_1, w_2, \ldots, w_n \).

  \( |w_i| \): length of \( w_i \).

  \( L \): length of each line.

  \( b \): ideal width of space between two words.

  \( \varepsilon \): minimum required space between words.

  \( b' \): actual width of space between words if the line is right justified.

- Assume that \( |w_i| + \varepsilon + |w_{i+1}| \leq L \) for all \( i \).
• If words $w_i, w_{i+1}, \ldots, w_j$ are typeset as a line, where $j \neq n$, the value of $b'$ for that line is

$$b' = \frac{L - \sum_{k=i}^{j} |w_k|}{(j - i)}$$

and the penalty is defined as:

$$Cost(i, j) = \begin{cases} |b' - b| \cdot (j - i) & \text{if } b' \geq \varepsilon \\ \infty & \text{if } b' < \varepsilon \end{cases}$$

• Right justification is not needed for the last line. So the width of space for setting $w_i, w_{i+1}, \ldots, w_j$ when $j = n$ is

$$\min(b, b')$$, and the penalty is

$$Cost(i, j) = \begin{cases} |b' - b| \cdot (j - i) & \text{if } \varepsilon \leq b' < b \\ 0 & \text{if } b \leq b' \\ \infty & \text{if } b' < \varepsilon \end{cases}$$
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 \(B_j = (b_j, b_{j+1}, \ldots, b_n)\).
- Let \(\varphi(k, j)\) be the index of the first character in \(B_j\) that
  is equal to \(a_k\), or \(n+1\) if no such character.
- Recurrence: \(L(i, j) = \begin{cases} 1 + \max_{\varphi(k, j) \leq n} \{L(k+1, \varphi(k, j)+1)\} \\ 0 \text{ if the set for the max is empty} \end{cases}\)
- Boundary condition: \(L(n+1, j) = L(i, n+1) = 0, \ 1 \leq i, j \leq n+1\).
- Running time: \(\Theta(n^3) + O(n^3) = \Theta(n^3)\)
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:

\[
L(i, j) = \begin{cases} 
\max & \left\{ 1 + L(i+1, \varphi(i, j)+1) \right\} & \text{if } \varphi(i, j) \leq n \\
L(i+1, j) & \text{otherwise}
\end{cases}
\]

• Running time: \(\Theta(n^2) + O(n^3)\)
Algorithm (non-recursive)

procedure Compute-Array-L

global $L[1..n+1, 1..n+1]$, $\varphi[1..n, 1..n]$

initialize $L[i, n+1] \leftarrow 0$, $L[n+1, j] \leftarrow 0$ for $1 \leq i, j \leq n+1$

compute $\varphi[1..n, 1..n]$

for $i \leftarrow n$ to 1 do

for $j \leftarrow n$ to 1 do

    if $\varphi(i, j) \leq n$ then

        $L[i, j] \leftarrow \max\{1 + L[i+1, \varphi(i, j) + 1], L[i+1, j]\}$

    else

        $L[i, j] \leftarrow L[i+1, j]$
Algorithm (recursive)

procedure Longest(i, j)
    //print the longest common subsequence//
    //assume $L[1..n+1, 1..n+1]$ has been computed//
    global $L[1..n+1, 1..n+1]$
    if $L[i, j] = L[i+1, j]$ then
        Longest $(i+1, j)$
    else
        Print $(a_i)$
        Longest $(i+1, \varphi(i, j)+1)$
    Initial call: Longest(1,1)
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 subsequence 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: 
  \(L(i, j) = 0\), if \(i = n + 1\) 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 $(x_1, x_2, ..., x_k)$.
  • 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?

• What about $L(i, j) = \min \{d(i, z) + L(z, j) : (i, z) \in E\}$?

• Let $L^k(i, j)$ denote the length of a shortest path from $i$ to $j$ with at most $k$ intermediate nodes.

• $L^k(i, j) = \min \{d(i, z) + L^{k-1}(z, j) : (i, z) \in E\}$. 
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)
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 $D^k[i, j] \leftarrow D^{k-1}[i, j]$
      $P^k[i, j] \leftarrow 0$

Straightforward implementation
Print paths

Procedure $\text{Path}(k, i, j)$

//shortest path from $i$ to $j$ w/o going thru $k+1, \ldots, n$ //

global $D^k[1..n, 1..n], P^k[1..n, 1..n], 0 \leq k \leq n$.

if $k = 0$ then
    if $i = j$ then print $i$
    elseif $D^0(i, j) < \infty$ then print $i, j$
    else print "no path"
elseif $P^k[i, j] = 1$ then
    $\text{Path}(k - 1, i, k), \text{Path}(k - 1, k, j)$
else
    $\text{Path}(k - 1, i, j)$
Print paths

Procedure ShortestPath(i, j)

//shortest path from i to j //

global $D^k[1..n, 1..n], P^k[1..n, 1..n]$, $0 \leq k \leq n$.

let $k' \leftarrow \begin{cases} 
\text{the largest } k \text{ such that } P^k[i, j] = 1 \\
0 \text{ if no such } k
\end{cases}$

if $k' = 0$ then

if $i = j$ then print $i$

elseif $D^0(i, j) < \infty$ then print $i, j$

else print "no path"

else

$\text{ShortestPath}(k' - 1, i, k'), \text{ ShortestPath}(k' - 1, k', j)$
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 $P[1..n, 1..n] \leftarrow 0$
for $k \leftarrow 1$ to $n$ do
    for $i \leftarrow 1$ to $n$ do
        for $j \leftarrow 1$ to $n$ do
            if $D[i, k] + D[k, j] < D[i, j]$ then
                $D[i, j] \leftarrow D[i, k] + D[k, j]$
                $P[i, j] \leftarrow k$
Sum of Subset

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

• This problem is NP-hard.
Job Scheduling on Two Machines

There are $n$ jobs to be processed, and two machines $A$ and $B$ are available. If job $i$ is processed on machine $A$ then $a_i$ units of time are needed. If it is processed on machine $B$ then $b_i$ units of processing time are needed. Because of the peculiarities of the jobs and the machines, it is possible that $a_i > b_i$ for some $i$ while $a_j < b_j$ for some other $j$. Schedule the jobs to minimize the completion time. (If jobs in $J$ are processed by machine $A$ and the rest by machine $B$, the completion time is defined to be $\max\left\{\sum_{i \in J} a_i, \sum_{i \notin J} b_i\right\}$.) Assume $1 \leq a_i, b_i \leq 3$ for all $i$. 