Dynamic Programming

Kleinberg and Tardos, Chapter 6

Dynamic Programming Applications

Areas:
- Bioinformatics.
- Control theory.
- Operations research.
- Computer science: graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms:
- Unix diff for comparing two files.
- Viterbi alg. for hidden Markov models.
- Smith-Waterman alg. for sequence alignment.
- Bellman-Ford for shortest path routing in networks.

Fibonacci numbers

\[ F(1) = F(2) = 1 \]
\[ F(n) = F(n-1) + F(n-2) \quad n > 2 \]

Simple recursive solution:

```python
def fib(n):
    if n <= 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

What is the size of the call tree?

Efficient computation using a memo table

```python
def fib(n, table):
    # precondition: n\(\geq 0\), table\([i]\) either 0 or contains fib\((i)\)
    if n <= 2:
        return 1
    if table[n] > 0:
        return table[n]
    result = fib(n-1, table) + fib(n-2, table)
    table[n] = result
    return result
```

Fibonacci numbers

\[ F(1) = F(2) = 1 \]
\[ F(n) = F(n-1) + F(n-2) \quad n \geq 2 \]
Efficient computation using a memo table

```
def fib(n, table):
    # precondition: n>0, table[i] either 0 or contains fib(i)
    if n <= 2:
        return 1
    if table[n] > 0:
        return table[n]
    result = fib(n-1, table) + fib(n-2, table)
    table[n] = result
    return result
```

We used a memo table, never computing the same value twice.

Look ma, no table

```
def fib(n):
    if n<=2:
        return 1
    a, b = 1
    c = 0
    for i in range(3, n+1):
        c = a + b
        a = b
        b = c
    return c
```

Avoid the table, only store the previous two.

How many calls now?

Pitfalls in recursive algorithms

A recursive algorithm can be inefficient if it solves the same sub problem many times.

Dynamic programming avoids this repetition by solving the problem bottom up and storing solutions to sub-problems.

---

6.1 Weighted Interval Scheduling

**Weighted Interval Scheduling**

- Weighted interval scheduling problem.
- Job j starts at s_j finishes at f_j, and has value v_j.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum value subset of compatible jobs.

**Unweighted Interval Scheduling**

**Greedy** algorithm works if all values are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary values are allowed.
Weighted Interval Scheduling

Assume jobs sorted by finish time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

\( p(j) \) = largest index \( i \) that is less than \( j \) such that job \( i \) is compatible with \( j \). Think of \( p(j) \) as the latest possible predecessor of \( j \).

Example: \( p(8) = 5, p(7) = 3, p(2) = 0. \)

Dynamic Programming: Recursive Solution

Notation:

\[ \text{OPT}(j) \text{: optimal value to the problem consisting of job requests } 1, 2, \ldots, j. \]

- Case 1: \( \text{OPT}(j) \) includes job \( j \).
  - add \( v_j \) to total value
  - can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \)
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \)

- Case 2: \( \text{OPT}(j) \) does not include job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j-1 \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise}
\end{cases}
\]

Analysis of the recursive solution

Structure of the recursion tree for family of "layered" instances:

What can we say about the running time of the recursive version of the alg?

Growing exponentially like the Fibonacci series

Lots of redundant work in recomputing solutions
Weighted Interval Scheduling: Memoization

**Memoization** Store results of each sub-problem in a cache; lookup as needed.

- **Input:** n, s₁, s₂, ..., sₙ, v₁, ..., vₙ
- Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.
- Compute p(1), p(2), ..., p(n)
- For j = 1 to n
  - M[j] = empty
  - M[0] = 0
  - M-Compute-Opt(j) {
    - If M[j] is empty
      - M[j] = max(vⱼ + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    - Return M[j]
  }

**Running Time?**

**Claim.** Memoized version of M-Compute-Opt(n) takes O(n) time.

- M-Compute-Opt(n) fills in all entries of M ONCE
- Each time a recursive call is made it fills an entry of M.
- Since M has n+1 entries, there can be only O(n) calls to M-Compute-Opt(n) and each invocation takes O(1) time.
- Overall running time of M-Compute-Opt(n) is O(n).

Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

- **Input:** n, s₁, s₂, ..., sₙ, v₁, ..., vₙ
- Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ
- Compute p(1), p(2), ..., p(n)
- **Iterative-Compute-Opt**
  - M[0] = 0
  - For j = 1 to n
    - M[j] = max(vⱼ + M[p(j)], M[j-1])

By going in bottom up order, M[p(j)] and M[j-1] are available when M[j] is computed.

Weighted Interval Scheduling: Finding a Solution

**Q.** Dynamic programming algorithm computes optimal value. What if we want the solution itself?

**A.** Do some post-processing.

- Run M-Compute-Opt(n)
- Run Find-Solution(n)
- **Find-Solution(j)**
  - If (j = 0) output nothing
  - Else if (vⱼ + M[p(j)] > M[j-1])
    - Print j
    - Find-Solution(p(j))
  - Else
    - Find-Solution(j-1)
- # of recursive calls is n = O(n).

**Example**

Let’s fill the matrix M for the following example:

<table>
<thead>
<tr>
<th>Time</th>
<th>v₁ = 2</th>
<th>v₂ = 4</th>
<th>v₃ = 4</th>
<th>v₄ = 7</th>
<th>v₅ = 2</th>
<th>v₆ = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>p₁</td>
<td>v₁</td>
<td>v₂</td>
<td>v₄</td>
<td>v₃</td>
<td>v₁</td>
<td>v₆</td>
</tr>
<tr>
<td>p₂</td>
<td>v₂</td>
<td>v₃</td>
<td>v₅</td>
<td>v₄</td>
<td>v₂</td>
<td>v₇</td>
</tr>
<tr>
<td>p₃</td>
<td>v₃</td>
<td>v₅</td>
<td>v₆</td>
<td>v₇</td>
<td>v₃</td>
<td>v₈</td>
</tr>
<tr>
<td>p₄</td>
<td>v₄</td>
<td>v₆</td>
<td>v₈</td>
<td>v₇</td>
<td>v₄</td>
<td>v₉</td>
</tr>
<tr>
<td>p₅</td>
<td>v₅</td>
<td>v₇</td>
<td>v₉</td>
<td>v₈</td>
<td>v₅</td>
<td>v₁₀</td>
</tr>
<tr>
<td>p₆</td>
<td>v₆</td>
<td>v₈</td>
<td>v₁₀</td>
<td>v₉</td>
<td>v₆</td>
<td>v₁₁</td>
</tr>
<tr>
<td>p₇</td>
<td>v₇</td>
<td>v₉</td>
<td>v₁₁</td>
<td>v₁₀</td>
<td>v₇</td>
<td>v₁₂</td>
</tr>
<tr>
<td>p₈</td>
<td>v₈</td>
<td>v₁₀</td>
<td>v₁₂</td>
<td>v₁₁</td>
<td>v₈</td>
<td>v₁₃</td>
</tr>
<tr>
<td>p₉</td>
<td>v₉</td>
<td>v₁₁</td>
<td>v₁₃</td>
<td>v₁₂</td>
<td>v₉</td>
<td>v₁₄</td>
</tr>
<tr>
<td>p₁₀</td>
<td>v₁₀</td>
<td>v₁₂</td>
<td>v₁₄</td>
<td>v₁₃</td>
<td>v₁₀</td>
<td>v₁₅</td>
</tr>
<tr>
<td>p₁₁</td>
<td>v₁₁</td>
<td>v₁₃</td>
<td>v₁₅</td>
<td>v₁₄</td>
<td>v₁₁</td>
<td>v₁₆</td>
</tr>
<tr>
<td>p₁₂</td>
<td>v₁₂</td>
<td>v₁₄</td>
<td>v₁₆</td>
<td>v₁₅</td>
<td>v₁₂</td>
<td>v₁₇</td>
</tr>
<tr>
<td>p₁₃</td>
<td>v₁₃</td>
<td>v₁₅</td>
<td>v₁₇</td>
<td>v₁₆</td>
<td>v₁₃</td>
<td>v₁₈</td>
</tr>
<tr>
<td>p₁₄</td>
<td>v₁₄</td>
<td>v₁₆</td>
<td>v₁₈</td>
<td>v₁₇</td>
<td>v₁₄</td>
<td>v₁₉</td>
</tr>
<tr>
<td>p₁₅</td>
<td>v₁₅</td>
<td>v₁₇</td>
<td>v₁₉</td>
<td>v₁₈</td>
<td>v₁₅</td>
<td>v₂₀</td>
</tr>
</tbody>
</table>

Example

Let’s figure out which jobs belong to the optimal solution.
Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming

- Characterize the structure of the problem: show how a problem can be solved using solutions to sub-problems
- Recursively define the optimum
- Compute the optimum bottom up, storing solutions of sub-problems
- Construct the optimum from the stored data

6.4 The Knapsack Problem


Subset Sums

Given a set of n objects with weights $w_i$ and a capacity $W$, find a subset $S$ with the largest sum of weights such that total weight is less than $W$

Does greedy work?

Largest first: No (3, 2, 2) $W=4$
Smallest first: No (1, 2, 2) $W=4$

Discrete Optimization Problems

**Discrete Optimization Problem** $(S, f)$

- $S$: space of feasible solutions (satisfying some constraints)
- $f: S \rightarrow \mathbb{R}$: cost function associated with feasible solutions
- Objective: find an optimal solution $x_{opt}$ such that $f(x_{opt}) \leq f(x)$ for all $x$ in $S$ (minimization)
  or $f(x_{opt}) \geq f(x)$ for all $x$ in $S$ (maximization)

- Ubiquitous in many application domains
  - planning and scheduling
  - VLSI layout
  - pattern recognition

Subset Sums

Given a set of n objects with weights $w_i$ and a capacity $W$, find a subset $S$ with the largest sum of weights such that total weight is less than $W$
Recursive Approach

- Either take object \( i \) or don’t.

Assume the current available capacity is \( w \).
- If we take object \( i \), leftover capacity is \( w - w_i \).
- If we don’t, leftover capacity is \( w \).

\[ \text{OPT}(i, w) = \text{weight of max weight subset that uses items } 1, \ldots, i \text{ with weight limit } w. \]

Recursion for OPT?

Recursion for Subset-Sum Problem

Notation: \( \text{OPT}(i, w) = \text{weight of max weight subset that uses items } 1, \ldots, i \text{ with weight limit } w. \)

Case 1: item \( i \) is not included:
- \( \text{OPT} \) includes best of \( \{1, 2, \ldots, i-1\} \) using weight limit \( w \).

Case 2: item \( i \) is included:
- new weight limit = \( w - w_i \).
- \( \text{OPT} \) includes best of \( \{1, 2, \ldots, i-1\} \) using remaining capacity \( w - w_i \).

\[ \text{OPT}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{OPT}(i-1, w) & \text{if } w_i > w \\ \max \{ \text{OPT}(i-1, w), w_i + \text{OPT}(i-1, w - w_i) \} & \text{otherwise} \end{cases} \]

Input: \( n, W, w_1, \ldots, w_n \)

for \( w = 0 \) to \( W \)

\[ M[0, w] = 0 \]

for \( i = 1 \) to \( n \)

for \( w = 0 \) to \( W \)

if \( w_i > w \)

\[ M[i, w] = M[i-1, w] \]

else

\[ M[i, w] = \max(M[i-1, w], w_i + M[i-1, w - w_i]) \]

return \( M[n, W] \)

Subset Sum Problem: Bottom-Up Dynamic Programming

Approach: Fill an \( n \)-by-\( W \) array.

Knapsack Problem

- Given \( n \) objects and a “knapsack” of capacity \( W \).
- Item \( i \) has a weight \( w_i > 0 \) and value \( v_i > 0 \).
- Goal: fill knapsack so as to maximize total value.

What would be a Greedy solution?
- repeatedly add item with maximum \( v_i / w_i \) ratio …

Does Greedy work?
- Capacity \( M = 7 \), Number of objects \( n = 3 \)
- \( w = [5, 4, 3] \) ordered by \( v_i / w_i \) ratio

What is the relation between Subset-Sum and Knapsack?
Knapsack Problem: Bottom-Up Dynamic Programming

Knapsack: Fill an n-by-W array.

**Input:** n, W, w₁,...,wₙ, v₁,...,vₙ

for w = 0 to W
  M[0, w] = 0
for i = 1 to n
  for w = 0 to W
    if (wᵢ > w)
      M[i, w] = M[i-1, w]
    else
      M[i, w] = max {M[i-1, w], vᵢ + M[i-1, w-wᵢ]}
return M[n, W]

Knapsack Algorithm

**Knapsack Problem: Running Time**

- **Running time:** Θ(nW).
  - Not polynomial in input size (W = 2ⁿ⁹nin).
  - "Pseudo-polynomial".
  - Decision version of Knapsack is NP-complete. [Chapter 8]

**Knapsack approximation algorithm.** There exists a polynomial-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Chapter 11]

Dynamic Programming

Dynamic Programming has the following steps
- Characterize the structure of the problem, i.e. show how a larger problem can be solved using solutions to sub-problems
- Recursively define the optimum
- Compute the optimum bottom up, storing values of sub-solutions
- Construct the optimum from the stored data

Algorithmic Paradigms

- **Greedy.** Build up a solution incrementally, optimizing some local criterion.
- **Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- **Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.