Dynamic Programming

Stefan D. Bruda

CS 317, Fall 2024

Memoization and dynamic programming

- Recursive implementations can be expensive:

  algorithm RecFib(n):
    if $n \leq 1$ **then return** n
    else return RecFib($n-1$) + RecFib($n-2$)

  $O(2^n)$ time
  $O(1)$+recursion space

- Memoization: Remember intermediate results

  algorithm MemFib(n):
    if $n = 0 \vee n = 1$ **then return** 1
    else
      if F_n is undefined **then**
        $F_n \leftarrow$ MemFib($n-1$) + MemFib($n-2$)
      return F_n

  $O(n)$ time
  $O(n)$ (+recursion) space

- Dynamic programming: Remember intermediate results explicitly

  algorithm DynFib(n):
    $F_0 \leftarrow 0$; $F_1 \leftarrow 1$
    for $i = 1$ **to** n **do** $F_n \leftarrow F_{n-1} + F_{n-2}$
    return F_n

  $O(n)$ time
  $O(n)$ space

- Can also consider remembering intermediate results only as needed

  algorithm DynFib(n):
    $prev \leftarrow 0$; $curr \leftarrow 1$
    for $i = 1$ **to** n **do**
      $next \leftarrow prev + curr$
      $prev \leftarrow curr$
      $curr \leftarrow next$
    return $curr$

  $O(n)$ time
  $O(1)$ space

Dynamic programming

- Dynamic programming = recursion without repetition
  1. Formulate the problem recursively
     - Use a bottom-up approach (starting from the base cases)
  2. Build the dynamic programming solution
     1. Identify subproblems
     2. Choose memoization data structure
     3. Identify dependencies and so find evaluation order
- Often but not always applicable to optimization problems
  - But in this case only for problems that satisfy the principle of optimality: An optimal solution to the problem contains optimal solutions to subproblems

0/1 knapsack

- Given $w = \langle w_1, \ldots, w_n \rangle$ and $p = \langle p_1, \ldots, p_n \rangle$, find $x = \langle x_1, \ldots, x_n \rangle$, $x_i \in \{0, 1\}$ such that $\sum_{i=1}^{n} x_i p_i$ is maximized subject to $\sum_{i=1}^{n} x_i w_i \leq C$
- Bottom-up recursive solution ($O(2^n)$):

  algorithm RecKnapsack(i, C, n, p, w):   (handle the i-th object)
    if $i > n$ **then return** $(0, \langle\rangle)$
    else
      $(p_-, X_-) \leftarrow$ RecKnapsack($i+1, C, n, p, w$)   (do not pick item i)
      if $w_i \leq C$ **then**
        $(p_+, X_+) \leftarrow$ RecKnapsack($i+1, C-w_i, n, p, w$)   (pick item i)
      else
        $(p_+, X_+) \leftarrow (0, \langle\rangle)$ (we cannot pick item i so we set profit to minimum)
      return MaxFst($\{(p_-, \langle 0 \rangle + X_-), (p_+ + w_i, \langle 1 \rangle + X_+)\}$)

- Memoization structure must contain information related to the remaining items and the remaining capacity $\Rightarrow$ table of item $\times$ capacity
  - Increment of capacity smaller than the smallest w_i
- Each subproblem (entry in the table) depends on the “upper” and “upper-left” subproblems
- Table filled in top to bottom, left to right

0/1 KNAPSACK (CONT'D)

- Dynamic programming solution:

algorithm KNAPSACK(C, n, p, w):
  for $i = 1$ **to** n **do** $P_{i,0} \leftarrow 0$
  for $j = 1$ **to** C **do** $P_{0,j} \leftarrow 0$
  for $i = 1$ **to** n **do**
    for $j = 1$ **to** C **do**
      if $w_i > j$ **then** $P_{i,j} \leftarrow P_{i-1,j}$
      else $P_{i,j} \leftarrow \max\{P_{i-1,j}, p_i + P_{i-1,j-w_i}\}$

algorithm KNAPSACKTRACE:
  $j \leftarrow C$
  for $i = n$ **downto** 1 **do**
    if $P_{i,j} = P_{i-1,j}$ **then**
      $x_i \leftarrow 0$
    else
      $x_i \leftarrow 1$
      $j \leftarrow j - w_i$

- Running time: $\Theta(n \times C) \rightarrow$ no better than $\Theta(2^n)$!
- Many problems are very similar to 0/1 Knapsack
  - Example (subset sum): Given an array $A_{1\ldots n}$ of positive integers and an integer T, does any subarray of A sums up to T
    - Subproblems: $SS(i,t) =$ TRUE iff some subset of A sums to t
    - Recursive solution:

$$SS(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0 \\ \text{FALSE} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < A_i \\ SS(i+1,t) \vee SS(i+1, t - A_i) & \text{otherwise} \end{cases}$$

    - Memoization structure: table $S_{1\ldots n, 0\ldots T}$
    - Evaluation order: rows bottom to top, arbitrary order in a row

MATRIX CHAIN MULTIPLICATION

- Given $M = M_1 \times M_2 \times \ldots \times M_n$ with the dimensions of the matrices stored in $r_{0\ldots n}$, such that each M_i has r_{i-1} rows and r_i columns, find how to bracket the matrix multiplications to minimize the total number of multiplications
  - Example: $r = \langle 2, 10, 1, 3\rangle$ that, is $A(2 \times 10) \times B(10 \times 1) \times C(1 \times 3)$
    - $A \times (B \times C)$ needs 90 integer multiplications
    - $(A \times B) \times C$ needs 26 integer multiplications (faster)
  - Subproblems: m_{ij} is the cost of computing $M_i \times \ldots \times M_j$
  - Recursive solution:

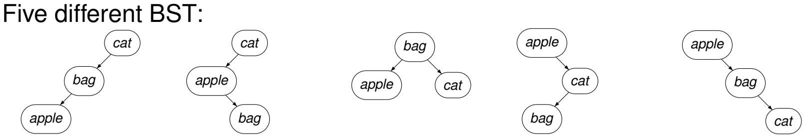
$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j}(m_{i,k} + m_{k+1,j} + r_{i-1} \times r_k \times r_j) & \text{if } i < j \end{cases}$$

  - Memoization structure: table $m_{1\ldots n-1, 1\ldots n}$ to store the result of subproblems
  - Evaluation order: by diagonal top to bottom with arbitrary order within a diagonal

algorithm MATRIXCHAINMULT: $O(n^3)$
  for $i = 1$ **to** n **do** $m_{ii} \leftarrow 0$
  for $r = 1$ **to** $n - 1$ **do**
    for $i = 1$ **to** $n - r$ **do**
      $j \leftarrow i + r$
      $m_{i,j} \leftarrow \min_{i \le k < j}(m_{i,k} + m_{k+1,j} + r_{i-1} \times r_k \times r_j)$

OPTIMAL BST

- Given n keywords along with their probabilities $p_1, p_2, \ldots, p_n$, store them in a binary search tree such that the average search time is minimized
  - Example: *cat* (0.1), *bag* (0.2), *apple* (0.7)
  - Sorted: *apple* (0.7), *bag* (0.2), *cat* (0.1)
  - Five different BST:



Average search time:

$0.1 + 2 \times 0.2 + 3 \times 0.7 = 2.6$	$0.1 + 2 \times 0.7 + 3 \times 0.2 = 2.1$	$0.2 + 2 \times 0.7 + 3 \times 0.1 = 1.8$	$0.7 + 2 \times 0.1 + 3 \times 0.2 = 1.5$	$0.7 + 2 \times 0.2 + 3 \times 0.1 = 1.4$

- Subproblems: $A_{i,j}$ is the average search time for a BST with keywords from i to j
- Recursive solution ($O(n^3)$ with memoization):

$$A_{i,j} = \begin{cases} p_i \text{ (root } i) & \text{if } i = j \\ 0 \text{ (null)} & \text{if } i > j \\ \min_{i \le k \le j}(A_{i,k-1} + A_{k+1,j} + \sum_{m=i}^{j} p_m) \text{ (root } k) & \text{if } i < j \end{cases}$$

- Obvious memoization
- Evaluation order: down by diagonal, arbitrary order within diagonal

ALL-PAIRS SHORTEST PATH

- Given a weighted (directed or undirected) graph $G = (V, E)$ wits $|V| = n$ and $|E| = m$, find the shortest path from each vertex to all other vertices
- Floyd's algorithm: Find shortest paths of rank k for increasing k
  - Uses the adjacency matrix $G_{1\ldots n, 1\ldots n}$ of G
  - Path of rank k: path that only traverses vertices 1 to k (not counting the source and the destination)
  - Subproblems: $P_k = (A^k_{i,j}, \pi^k_{i,j})_{1 \le i \le n, 1 \le j \le n}$
    - $A^k_{i,j}$ is the cost of the minimum path of rank k from i to j
    - $\pi^k_{i,j}$ is the predecessor of j in the minimum cost path of rank k from i to j
    - Recursive solution:

$$A^k_{i,j} = \begin{cases} G_{i,j} & \text{if } k = 0 \\ \min\{A^{k-1}_{i,j}, A^{k-1}_{i,k} + A^{k-1}_{k,j}\} & \text{otherwise} \end{cases}$$

$$\pi^k_{i,j} = \begin{cases} i & \text{if } k = 0 \\ \pi^{k-1}_{i,j} & \text{if } A^{k-1}_{i,j} \le A^{k-1}_{i,k} + A^{k-1}_{k,j} \\ k & \text{if } A^{k-1}_{i,j} > A^{k-1}_{i,k} + A^{k-1}_{k,j} \end{cases}$$

Floyd's algorithm (cont'd)

- Memoization: arrays A^k and π^k for cost and predecessor
- Evaluation order: increasing k, arbitrary for i and j

```
for i = 1 to n do
    for j = 1 to n do
        A^0_{i,j} ← G_ij
        π_{i,j} ← i

for k = 1 to n do        O(n^3)
    for i = 1 to n do
        for j = 1 to n do
            if A^{k-1}_{i,j} ≤ A^{k-1}_{i,k} + A^{k-1}_{k,j} then
                A^k_{i,j} ← A^{k-1}_{i,j}
            else
                A^k_{i,j} ← A^{k-1}_{i,k} + A^{k-1}_{k,j}
                π_{i,j} ← k
```

  - Optimization: A single predecessor array π
    - When computing $\pi^k_{i,j}$ we only need $\pi^{k-1}_{i,j}$ and then we can overwrite it
  - Further optimization: At any step we only need A^{k-1} and A^k, so we only need two matrices for the cost (current and previous)

The travelling salesman problem

- Given a weighted directed graph $G = (\{1, 2, \ldots, n\}, E)$ find the Hamiltonian Cycle of minimum cost
  - Naïve solution: try all the permutations, retain the one with minimal cost ($O(n2^n)$ time)
- Crux:
  - Start the cycle at vertex 1
  - Let the next vertex be k
  - The path from k to 1 must be an optimal (minimum cost) Hamiltonian path for the graph induced by $V \setminus \{1\}$
- Recursive solution:
  - Let $g(i, S)$ be the length of the shortest path starting at i and going through all the vertices in S back to 1

$$g(i, S) = \begin{cases} \min_{(i,j)\in E}(w(i,j)) & \text{if } S = \emptyset \\ \min_{j \in S}(w((i,j)) + g(j, S \setminus \{j\})) & \text{otherwise} \end{cases}$$

  - Memoization: Table $(g_{i,j})_{i\in\{1,\ldots,n\}, j \in 2^{\{1,\ldots,n\}}}$
  - Order of evaluation: increasing second dimension, do not care for the first
  - Running time: $O(n2^n)$
- Unknown (million dollar question, literally) whether we can do better than the naïve solution for the travelling salesman and the 0/1 knapsack (and many more problems)