Dynamic Programming and Greedy Search

Leticia Peres & Héctor Palacios

Agenda

- Context: Description of the problem to be solved
- Dynamic Programming (DP)
  - Elements of DP
  - Example: Matrix-chain multiplication
  - Recursive vs. Iterative algorithms
- Greedy Algorithms (GA)
  - General Idea
  - Example: Activity-selection problem
  - Summarizing: Elements to develop a GA
  - Other examples: knapsack problem, minimum spanning tree.

Context of use

- Optimization problems, *i.e.*...
  - State naming a configuration: $s \in S$
  - Cost asociated to states: $c(s) : S \to \mathbb{R}$
- Some of them are **solutions**: $Sols \subset S$
- Objetive: find an **optimal** solution: $S' \in Sols$
- Optimal means **minimum cost**
- May be that there are **many** optimal solutions, **one is enough**

First aproximation: General Idea

- Divide-and-Conquer
  - Decompose the problem into **subproblems**
  - Merge the solutions of the subproblems into the solution of the big problem
- If there are subproblems that **overlap** between them:

  Dynamic Programming

Elements of DP: Optimal Substructure

- Optimal solution to a problem **contains** optimal solutions to subproblems

- Solve first subproblem and use it to **construct** the optimal for the problem
- But you have to check **all the possible ways** of obtain a decomposition into subproblems

- It leads to a **recursive** algoritm

Elements of DP: Overlapping Subproblem

- **Naive** recursive solution seems to be exponential
- It is possible solve each subproblem just **one** time

- Two solutions:
  - **Memoization**: Each time you need to solve a problem, look if it was already solved.
  - **Iterative** version that proceeds bottom-up

Example: Matrix-chain multiplication

- Sequence of n matrices to be **multiplied**:

$$A_1 \times A_2 \times \ldots \times A_n$$

- We can parenthesize in many ways, obtaning very differents number of operations:
  - Ej: $A \times B \times C$ of dimensions $10 \times 100, 100 \times 5, 5 \times 50$
  - Using naive functions for multiplying pairs of matrices:
    - $((A \times B) \times C)$ takes $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7,500$
    - $(A \times (B \times C))$ takes $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75,000$
- The **order** of multiplication does matters
- Our **problem**: obtain an optimal **order**
  (check all possible parenthesizations! $\longrightarrow O(2^n)$)

Matrix-chain Mult: Optimal substructure

- Let $A_{i..j}$, $i \leq j$, the result of $A_i A_{i+1} \ldots A_j$
- Any parenthesization must split between A_k and A_{k+1}, $i \leq k < j$.
  i.e. $A_{i..j}$ is calculated multiplying $A_{i..k}$ and $A_{k+1..j}$
- The cost of this **way** to calculate, is the cost of **both** parts.
- So, if we have an optimal order $O_{i..j}$ for calculate $A_{i..j}$, it split the product at some k
- Then, using that (sub)order for $A_{i..k}$ and $A_{k+1..j}$ must be optimal too
- In other case, it will exists a **better** order than $O_{i..j}$
  (combining the new optimal suborders to construct a new globally optimal order)

Matrix-chain Mult: Recursive solution

- Let $m[i,j]$ the minimum number of multiplications needed to compute $A_{i..j}$
- The cost of the global problem ($A_{1..n}$) is $m[1,n]$: we want to minimize it (optimization problem)
- Cases on i,j
  - $i=j$, no cost
  - $i<j$, assuming a split point k, the cost is

$$m[i,k] \quad + \quad m[k+1,j] \quad + \quad p_{i-1}\, p_k\, p_j$$

  where matrix A_i have dimensions $p_{i-1} \times p_i$

Matrix-chain Mult: Recursive solution

- But we have to choose a k that **minimize** the cost
- It leads to

$$m[i,k] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} m[i,k] \quad + \quad m[k+1] \quad + \quad p_{i-1}\, p_k\, p_j & \text{if } i < j \end{cases}$$

- Keep records of choices to recover the solution:

  $s[i,j] = k$ choosed to minimize in previous equation

- Easy to implement as a **naive recursive** algorithm, but $O(2^n)$

Matrix-chain Mult: Overlapping structure

- The **naive recursive** algorithm is $O(2^n)$
- Number of subproblems: choice i and j s.t. $1 \leq i \leq j \leq n$ which is $\approxeq \binom{n}{2} \approxeq \Theta(n^2)$
- Executing the naive recursive algorithm, it will find each subproblem many times
- Idea: Remember the solution to each new subproblem we find (**Memoization**)

Matrix-chain Mult: Memoized version

```
MATRIX-CHAIN-ORDER(p)
    n ← length[p] − 1
    for i ← 1 to n
        do for j ← i to n
            do m[i, j] ← ∞ (initialize to "undefined" table entries)
    return LOOKUP-CHAIN(p, 1, n)
LOOKUP-CHAIN(p, i, j)
    if m[i, j] < ∞ (see if we know or not)
        then return m[i, j]
    if i = j
        then m[i, j] = 0
        else for k ← i to j − 1
                do q ← LOOKUP-CHAIN(p, i, k) +
                       LOOKUP-CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
                   if q < m[i, j]
                      then m[i, j] ← q
    return m[i, j]
```

Matrix-chain Mult: Memoized version

- Each subproblem is computed just one time: $O(n^2)$
- Each time a subproblem is calculated, it requires $O(n)$ calls
- So, it is $O(n^3)$

- We can also give a **iterative** version, without memoization
- It is useful when you have to calculate **all** the subproblems
- Or make optimizations on time or space
- (Literature uses to present DP going directly to iterative or **tabular** version)

Matrix-chain Mult: Iterative version

```
MATRIX-CHAIN-ORDER(p)
  n ← length[p]
  for i ← 1 to n
      do m[i, i] ← 0
  for l ← 2 to n
      do for i ← 1 to n − l + 1
             do j ← i + l − 1
                m[i, j] ← ∞
                for k ← i to j − 1
                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                   if q < m[i, j]
                      then m[i, j] ← q
                           s[i, j] ← k
   return m and s
```

- Note the filling of $s[i, j]$

Context of use GA

- *Optimization problems, i.e....*
  - *State naming a configuration:* $s \in S$
  - *Cost asociated to states:* $c(s) : S \rightarrow \mathbb{R}$
- *Some of them are* **solutions***:* $Sols \subset S$
- *Objetive: find an* **optimal** *solution:* $S' \in Sols$
- *Optimal means* **minimum cost**
- *May be that there are* **many** *optimal solutions,* **one is enough**

First aproximation: General Idea

- *If there are subproblems that* **overlap** *between them:* **maybe** there is a Greedy Algorithm (GA)
- GA decides for the solution that seems better at the moment (local decision)
- GA will work when this decision reaches to the globally optimal solution

Using the initial idea of GA: An Example

- Problem: **scheduling** of several **competing activities** that require exclusive use of a common resource (ex scheduling many activities that can be conduced in a lecture hall, but only one at a time)
- Goal: to select a **maximum-size set** of mutually compatible activities

An activity-selection problem (ASP)

- Suppose we have a set
  $S = \{a_1, a_2, ..., a_n\}$ of activities that wish use a resource.
- Each activity a_i has
  - a start time s_i
  - a finish time f_i

  where $0 \leq s_i < f_i < \infty$
- Activities a_i and a_j are **compatible** if $s_i \geq f_j$ or $s_j \geq f_i$

Starting with a dynamic-programming solution

- We start developing a dynamic-programming solution to the ASP
  1. to find the **optimal substructure**
  2. to use it to construct an **optimal solution to the problem** from **optimal solutions to subproblems**

To find the optimal substructure

- Define an appropriate **space of subproblems**

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

S_{ij} is the subset of all compatible activities in S that can start after activity a_i finishes and finish before activity a_j starts

- In order to represent the entire problem, we adopt:

a_0 and a_{n+1}, $f_0 = 0$ and $s_{n+1} = \infty$

- Then

$S = S_{0,n+1}$ and $0 \leq i, j \leq n+1$

Space of subproblems

- Let us assume that we have sorted the activities in monotonically increasing order of finish time:

$$f_0 \leq f_1 \leq f_2 \leq ... \leq f_n \leq f_{n+1}$$

($S_{ij} = \{\ \}$ whenever $i \geq j$)

- We can conclude that our space of subproblems is to **select a maximum-size subset** of mutually compatible activities **from** S_{ij}, for $0 \leq i < j \leq n+1$ knowing that all other S_{ij} are empty.

... to see the substructure of the ASP

- Subset S_{ij} can be seen like a **subproblem**
- Consider now some non-empty S_{ij} a
- Suppose that a solution to S_{ij} includes some activity a_k so that $f_i \leq s_k < f_k \leq s_j$
- Activity a_k generates two subproblems: S_{ik} and S_{kj}, s.t. S_{ik}, $S_{kj} \in S_{ij}$
- Our solution to S_{ij} is the **union** of the solutions to S_{ik} and S_{kj}, along with the activity a_k
- The size of solution $S_{ij} =$
  size of solution to S_{ik} + size of solution to S_{kj} + one (a_k)

At least, the optimal substructure!

- Suppose,

  A_{ij} is an optimal solution to S_{ij}

  A_{ij} includes activity a_k

- Then the solutions A_{ik} to S_{ik} and A_{kj} to S_{kj} used within this optimal solution to S_{ij} **must be optimal as well**.

- In other case, we will have a solution better than A_{ik}, named A'_{ik}. We can construct A'_{ij} combining A'_{ik} with A_{kj} that will be a better solution than A_{ij} **which is a contradiction**.

- Similarly with A_{kj} and A'_{kj} to S_{kj} and A_{ij}

Constructing an optimal solution

- We see that
  - Any solution to a nonempty subproblem S_{ij} includes some activity a_k,
  - Any optimal solution **contains optimal solutions to subproblem** instances S_{ik} and S_{kj}.
- It is possible to build a maximum-size subset of mutually compatible activities in S_{ij} by:

  1. splitting the problem into **two subproblems** S_{ik} and S_{kj}

  2. finding **maximum-size subsets** A_{ik} and A_{kj} of mutually compatible activities for these problems, and

  3. **forming** our maximum-size subset A_{ij} of mutually compatible activities as

$$A_{ij} = A_{ik} \cup a_k \cup A_{kj}$$

  4. An optimal solution to the entire problem is a solution to $S_0, n+1$

Now, a recursive solution...

The second step in developing a dynamic-programming solution is to recursively define the value of an optimal solution,

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \{\,\} \\ \max\limits_{i<k<j, a_k \in S_{ij}} c[i,k] \;+\; c[k,j] \;+\; 1 & \text{if } S_{ij} \neq \{\,\} \end{cases}$$

Finally a greedy solution!

- Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time:
  $f_m = min\{f_k : a_k \in S_{ij}\}$
- Then

  1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}

  2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Proof

- (2) Suppose that S_{im} is nonempty, so that there is some activity a_k such that $f_i \leq s_k < f_k \leq s_m < f_m$.
- Then a_k is also S_{ij} and it has an earlier finish time than a_m, which contradicts our choice of a_m. We conclude that S_{im} is empty.

Proof

- (1) Suppose that A_{ij} is a maximum-size subset of mutually compatible activities of S_{ij}, and the activities in A_{ij} are **monotonically increasing ordered by finish time**.
- Let a_k be the first activity in A_{ij}.
- If $a_k = a_m$, a_m is used in A_{ij}.
- If $a_k \neq a_m$, we construct the subset $A'_i j = A_{ij} - a_k \cup a_m$.

1. The activities in A'_{ij} are disjoint, since the activities in A_{ij} are.
a_k is the first activity in A_{ij} to finish.
$f_m \leq f_k$.

2. Noting that A'_{ij} has the same number of activities as A_{ij}, we see that A'_{ij} is a maximum-size subset of mutually compatible activities of S_{ij} that includes a_m.

Importance of this theorem

- Reduces to only one subproblem to be used in an optimal solution of the example
- To solve the subproblem S_{ij}, we need consider only one choice: the one with the **earliest** finish time in S_{ij}.
- Yet, we can solve each subproblem in a top-down fashion:
  - To solve S_{ij}

    1. **Choose** a_m in S_{ij} with the earliest finish time

    2. Let S_{mj} the set of activities used in an optimal solution to the subproblem A_{mj}

    3. **Add** A_{mj} to S_{ij}
  - Having chosen a_m, we will use a solution to S_{mj} in our optimal solution to S_{ij}
  - So, we **do not** need to solve S_{mj} before solving S_{ij}
  - To solve S_{ij}, we can **first** choose a_m as the activity in S_{ij} with the earliest finish time and **then** solve S_{mj}.

A recursive greedy algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, i, n)
  $m \leftarrow i + 1$
  while $m \leq n$ **and** $s_m \leq f_i$
    do $m \leftarrow m + 1$
  if $m \leq n$
    then return
      $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
    else return $\{\}$

A iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

    $n \leftarrow \text{length}[S]$

    $A \leftarrow \{a_1\}$

    $i \leftarrow 1$

    for $m \leftarrow 2$ **to** n

        do if $s_m \geq f_i$

            then $A \leftarrow A \cup \{a_m\}$

                    $i \leftarrow m$

    return A

Patterns in subproblems and activities

- Note that there is a pattern to the subproblems that we solve.
  - Our original problem is $S = S_{0,n+1}$. Suppose that we choose a_{m_1} as the activity in $S_{0,n+1}$ with the earliest finish time. Our next subproblem is $S_{m_1,n+1}$.
  - Now suppose that we choose a_{m_2} as as the activity in $S_{m_1,n+1}$ with the earliest finish time. Our next subproblem is in $S_{m_2,n+1}$. and continuing... Each subproblem will be of the form $S_{m_i,n+1}$, for some activity number m_i.

Patterns in subproblems and activities

- There is also a pattern to the activities that we choose:
  - Because we always choose the activity with the **earliest** finish time in $S_{m_i,n+1}$, the finish times of the activities chosen over all subproblems will be **strictly increasing** over time.
  - The activity a_m that we choose when solving a subproblem is always the one with the earliest finish time that can be legally scheduled.
  - The activity picked is thus a **"greedy" choice**: one that maximizes the amount of unscheduled time remaining.

Elements of Greedy Strategy

- What have we done on activity-selection example?
  - Determined the optimal substructure
  - Developed a recursive version
  - Shown that **always one** of the optimal choices is the greedy choice
  - Shown that just left **one** non-empty subproblem
  - Given the recursive and iterative algorithm (it doesn't matter so much in greedy search)
- It is necessary to follow all these steps?
- If we are focused on a greedy solution
  - See the optimization problem as **make a greedy choice** and **solve the subproblem**
  - Show that **there is always** an optimal solution that makes the greedy choice
  - Prove that **combining** the solution to the subproblem with the greedy choice, we obtain an **globally** optimal solution

Elements of GS: Greedy-choice property

- **Greedy choice property**: An optimal solution can be obtained by making locally optimal choice
- Note that in DP we make also a choice, but it **depends on the solutions** to subproblems
- We **must prove** that a greedy choice at each step yields a globally optimal solution
  - It usually involve examine the solution to a problem
  - Then show that it can be modified to use the greedy choice
  - It will left **just one** and simpler subproblem
- It is common to make some **preprocessing** on data:

  order data, etc.

Elements of GS: Optimal substructure

- Instead of think the optimal substructure in general (DP)

  It is enough show that an optimal solution to the subproblem, combined with the greedy choice, yields a globally optimal solution

GA vs DP: two variations of Knapsack

- A thief has a knapsack that holds at most W pounds
- **0 - 1 knapsack problem**:
  - Each item must be taken or left (0 - 1)
  - thief must choose items to maximize the value stolen and still fit into the knapsack
  - For each item i: (v_i, w_i) (v = value, w = weight)
- **Fractional knapsack problem**:

  takes parts, as well as wholes

GA vs DP: two variations of Knapsack

- Both the 0 - 1 and fractional problems have the optimal substructure property
- **0 - 1 knapsack problem**:
  - Consider the **most valuable load** that weights at most W pounds
  - Removing item j from the load, the **remaining load** must be the most valuable load weighing at most $W - w_j$ that the thief can take (excluding item j)
- **Fractional knapsack problem**:
  - Removing a weight w of one item j, the remaining load must be the most valuable load weigthing at most $W - w$ that the thief can take (including $w_j - w$ for item j)

GA vs DP: two variations of Knapsack

- But Fractional can be solved by a GA, and 0-1 can't
- **Fractional knapsack problem**:
  - Greedy choice: Take as much as possible of the item with the greatest value per pound

GA vs DP: two variations of Knapsack

- **0 - 1 knapsack problem**:

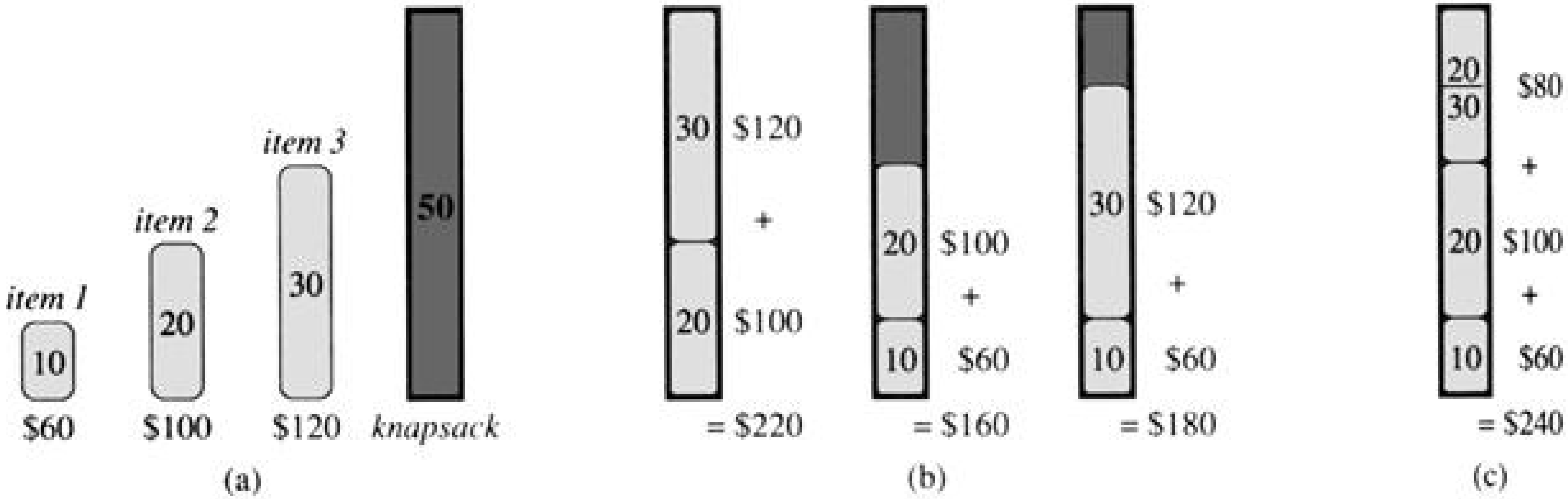

Figure 17.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Minimun Spanning Tree (MST)

- Having a graph G with a weight $w(e)$ associated to each edge e
- Problem: obtain a tree T covering G with minimun cost

$$\sum_{e \in Edges(T)} w(e)$$

- Start from a vertex (a simple partial MST)
- Greedy Choice: grown from the current spanning tree by **adding the nearest vertex**. (**Prim's** algorithm)
- The nearest vertex will be going througt the edge with **minimum** cost

Finally

- When solving optimization problems it is useful study the **structure** of the space
- Becasu some structures suggest some kinds of algorithms

- Be sure that the problem have **optimal substructure**
- Sometimes it will have a **greedy choice**

- Greedy search is also useful to obtain **sub-optimal** solutions
- It is related with other **local search** algorithms:
  simulated annealing, tabu search, genetic algorithms, etc

Finished

- Thank you!

- Questions? Comments?

Credits

- Introduction to Algorithms (2nd Edition) by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein.
- Many sources on Internet
- Andreas Kaltenbrunnner and Dani Martí for provide us the Latex format and even posterior support

Prim Algorithm

```
Prim(G) Select an arbitrary vertex to start
While (there are vertices dont cover by the tree)
      select minimum-weight edge between tree and fringe
      add the selected edge and vertex to the tree
```

Why is Prim's algorithm correct?

Theorem: Let G be a connected, weighted graph and let be a subset of the edges in a MST .

Let V' be the vertices incident with edges in E'. If (x,y) is an edge of minimum weight such that and y is not in V', then is a subset of a minimum spanning tree.

Proof: If the edge is in T, this is trivial.

Suppose (x,y) is not in T Then there must be a path in T from x to y since T is connected.

If (v,w) is the first edge on this path with one edge in V',

if we delete it and replace it with (x, y) we get a spanning tree.

This tree must have smaller weight than T, since W(v,w)¿W(x,y).

Thus T could not have been the MST.