CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Winter, 2015

Table of Contents

Optimization Problems

Problem: Given a problem instance, find a feasible solution that maximizes (or minimizes) a certain objective function.

Problem Instance: **Input** for the specified problem.

Problem Constraints: **Requirements** that must be satisfied by any feasible solution.

Feasible Solution: For any problem instance I, *feasible*(I) is the set of all outputs (i.e., solutions) for the instance I that satisfy the given constraints.

Objective Function: A function f : *feasible*$(I) \rightarrow \mathbb{R}^+ \cup \{0\}$. We often think of f as being a **profit** or a **cost** function.

Optimal Solution: A feasible solution $X \in$ *feasible*(I) such that the profit $f(X)$ is maximized (or the cost $f(X)$ is minimized).

The Greedy Method

partial solutions

Given a problem instance I, it should be possible to write a feasible solution X as a tuple $[x_1, x_2, \ldots, x_n]$ for some integer n, where $x_i \in \mathcal{X}$ for all i. A tuple $[x_1, \ldots, x_i]$ where $i < n$ is a **partial solution** if no constraints are violated. Note: it may be the case that a partial solution cannot be extended to a feasible solution.

choice set

For a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, we define the **choice set**

$$\textit{choice}(X) = \{y \in \mathcal{X} : [x_1, \ldots, x_i, y] \text{ is a partial solution}\}.$$

The Greedy Method (cont.)

local evaluation criterion

For any $y \in \mathcal{X}$, $g(y)$ is a **local evaluation criterion** that measures the cost or profit of including y in a (partial) solution.

extension

Given a partial solution $X = [x_1, \ldots, x_i]$ where $i < n$, choose $y \in$ *choice*(X) so that $g(y)$ is as small (or large) as possible. Update X to be the $(i+1)$-tuple $[x_1, \ldots, x_i, y]$.

greedy algorithm

Starting with the "empty" partial solution, repeatedly extend it until a feasible solution X is constructed. This feasible solution may or may not be optimal.

Features of the Greedy Method

Greedy algorithms do no **looking ahead** and no **backtracking**.

Greedy algorithms can usually be implemented efficiently. Often they consist of a **preprocessing step** based on the function g, followed by a **single pass** through the data.

In a greedy algorithm, only **one feasible solution** is constructed.

The execution of a greedy algorithm is based on **local criteria** (i.e., the values of the function g).

Correctness: For certain greedy algorithms, it is possible to prove that they always yield optimal solutions. However, these proofs can be tricky and complicated!

Interval Selection

Problem

Interval Selection

Instance: *A set* $\mathcal{A} = \{A_1, \ldots, A_n\}$ *of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, *where* s_i *is the* **start time** *of interval* A_i *and* f_i *is the* **finish time** *of* A_i.

Feasible solution: *A subset* $\mathcal{B} \subseteq \mathcal{A}$ *of* **pairwise disjoint intervals**.

Find: *A feasible solution of maximum size (i.e., one that maximizes* $|\mathcal{B}|$*).*

Possible Greedy Strategies for Interval Selection

1. Choose the **earliest starting** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is s_i).
2. Choose the interval of **minimum duration** that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is $f_i - s_i$).
3. Choose the **earliest finishing** interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is f_i).

Does one of these strategies yield a correct greedy algorithm?

A Greedy Algorithm for Interval Selection

Algorithm: *GreedyIntervalSelection*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $f_1 \leq \cdots \leq f_n$

$\mathcal{B} \leftarrow \{A_1\}$

$prev \leftarrow 1$

comment: $prev$ is the index of the last selected interval

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

$$\textbf{do} \begin{cases} \textbf{if } s_i \geq f_{prev} \\ \quad \textbf{then } \begin{cases} \mathcal{B} \leftarrow \mathcal{B} \cup \{A_i\} \\ prev \leftarrow i \end{cases} \end{cases}$$

return $(\mathcal{B})$

Interval Colouring

Problem

Interval Colouring

Instance: *A set $\mathcal{A} = \{A_1, \dots, A_n\}$ of* **intervals**.
For $1 \leq i \leq n$, $A_i = [s_i, f_i)$, where s_i is the **start time** *of interval A_i and f_i is the* **finish time** *of A_i.*

Feasible solution: *A* **c-colouring** *is a mapping $col : \mathcal{A} \rightarrow \{1, \dots, c\}$ that assigns each interval a* **colour** *such that two intervals receiving the same colour are always disjoint.*

Find: *A c-colouring of $\mathcal{A}$ with the minimum number of colours.*

Greedy Strategies for Interval Colouring

As usual, we consider the intervals one at a time.

At a given point in time, suppose we have coloured the first $i < n$ intervals using d colours.

We will colour the $(i+1)$st interval with the **any permissible colour**. If it cannot be coloured using any of the existing d colours, then we introduce a **new colour** and d is increased by 1.

Question: In **what order** should we consider the intervals?

A Greedy Algorithm for Interval Colouring

Algorithm: *GreedyIntervalColouring*$(\mathcal{A})$

rename the intervals, by sorting if necessary, so that $s_1 \leq \cdots \leq s_n$

$d \leftarrow 1$

$colour[1] \leftarrow 1$

$finish[1] \leftarrow f_1$

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

do $\begin{cases} flag \leftarrow \textbf{ false} \\ c \leftarrow 1 \\ \textbf{while } c \leq d \textbf{ and } (\textbf{ not } flag) \\ \quad \textbf{do } \begin{cases} \textbf{if } finish[c] \leq s_i \textbf{ then } \begin{cases} colour[i] \leftarrow c \\ finish[c] \leftarrow f_i \\ flag \leftarrow \textbf{ true} \end{cases} \\ \textbf{else } c \leftarrow c + 1 \end{cases} \\ \textbf{if not } flag \textbf{ then } \begin{cases} d \leftarrow d + 1 \\ colour[i] \leftarrow d \\ finish[d] \leftarrow f_i \end{cases} \end{cases}$

return $(d, colour)$

Comments and Questions

In the algorithm on the previous slide, at any point in time, $\mathit{finish}[c]$ denotes the finishing time of the **last interval** that has received colour c. Therefore, a new interval A_i can be assigned colour c if $s_i \geq \mathit{finish}[c]$.

The complexity of the algorithm is $O(n \times D)$, where D is the value of d returned by the algorithm.

If it turns out that $D \in \Omega(n)$, then the best we can say is that the complexity is $O(n^2)$.

What **inefficiencies** exist in this algorithm?

What **data structure** would allow a more efficient algorithm to be designed?

What would be the complexity of an algorithm making use of an appropriate data structure?

The Stable Marriage Problem

Problem

Stable Marriage

Instance: *A set of n* **men***, say $M = [m_1, \ldots, m_n]$, and a set of n* **women***, $W = [w_1, \ldots, w_n]$.*

Each man m_i has a **preference ranking** *of the n women, and each woman w_i has a preference ranking of the n men: $pref(m_i, j) = w_k$ if w_k is the j-th favourite woman of man m_i; and $pref(w_i, j) = m_k$ if m_k is the j-th favourite man of woman w_i.*

Find: *A* **matching** *of the n men with the n women such that there does not exist a couple (m_i, w_j) who are* **not** *engaged to each other, but prefer each other to their existing matches. A matching with this this property is called a* **stable matching***.*

Overview of the Gale-Shapley Algorithm

Men propose to women.

If a woman accepts a proposal, then the couple is **engaged**.

An unmatched woman **must accept** a proposal.

If an engaged woman receives a proposal from a man whom she prefers to her current match, the she **cancels** her existing engagement and she becomes engaged to the new proposer; her previous match is no longer engaged.

If an engaged woman receives a proposal from a man, but she prefers her current match, then the proposal is **rejected**.

Engaged women never become unengaged.

A man might make a number of proposals (up to n); the order of the proposals is determined by the man's preference list.

Gale-Shapley Algorithm

Algorithm: *Gale-Shapley*$(M, W, pref)$

$Match \leftarrow \emptyset$

while there exists an unengaged man m_i

do
- let w_j be the next woman in m_i's preference list
- **if** w_j is not engaged
  - **then** $Match \leftarrow Match \cup \{m_i, w_j\}$
  - **else**
    - suppose $\{m_k, w_j\} \in Match$
    - **if** w_j prefers m_i to m_k
      - **then**
        - $Match \leftarrow Match \backslash \{m_k, w_j\} \cup \{m_i, w_j\}$
        - comment: m_k is now unengaged

return $(Match)$

Questions

How do we prove that the *Gale-Shapley* algorithm always **terminates**?

How many **iterations** does this algorithm require in the worst case?

How do we prove that this algorithm is **correct**, i.e., that it finds a stable matching?

Is there an efficient way to **identify** an unengaged man at any point in the algorithm? What **data structure** would be helpful in doing this?

What can we say about the **complexity** of the algorithm?

Knapsack Problems

Problem

Knapsack

Instance: **Profits** $P = [p_1, \ldots, p_n]$; **weights** $W = [w_1, \ldots, w_n]$; *and a* **capacity**, M. *These are all positive integers.*

Feasible solution: *An n-tuple $X = [x_1, \ldots, x_n]$ where $\sum_{i=1}^{n} w_i x_i \leq M$.*

In the **0-1 Knapsack** *problem (often denoted just as* **Knapsack***), we require that $x_i \in \{0, 1\}$, $1 \leq i \leq n$.*

In the **Rational Knapsack** *problem, we require that $x_i \in \mathbb{Q}$ and $0 \leq x_i \leq 1$, $1 \leq i \leq n$.*

Find: *A feasible solution X that maximizes $\sum_{i=1}^{n} p_i x_i$.*

Possible Greedy Strategies for Knapsack Problems

1. Consider the items in decreasing order of **profit** (i.e., the local evaluation criterion is p_i).
2. Consider the items in increasing order of **weight** (i.e., the local evaluation criterion is w_i).
3. Consider the items in decreasing order of **profit divided by weight** (i.e., the local evaluation criterion is p_i/w_i).

Does one of these strategies yield a correct greedy algorithm for the **0-1 Knapsack** or **Rational Knapsack** problem?

A Greedy Algorithm for Rational Knapsack

Algorithm: *GreedyRationalKnapsack*$(P, W : array; M : integer)$

rename the items, sorting if necessary, so that $p_1/w_1 \geq \cdots \geq p_n/w_n$

$X \leftarrow [0, \ldots, 0]$

$i \leftarrow 1$

$CurW \leftarrow 0$

while $(CurW < M)$ **and** $(i \leq n)$

$$\textbf{do} \begin{cases} \textbf{if } CurW + w_i \leq M \\ \quad \textbf{then } \begin{cases} x_i \leftarrow 1 \\ CurW \leftarrow CurW + w_i \end{cases} \\ \quad \textbf{else } \begin{cases} x_i \leftarrow (M - CurW)/w_i \\ CurW := M \end{cases} \end{cases}$$

return (X)

Coin Changing

Problem

Coin Changing

Instance: *A list of* **coin denominations***,* $d_1, d_2, \ldots, d_n$*, and a positive integer* T*, which is called the* **target sum***.*

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$*, such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

In the **Coin Changing** problem, a_i denotes the number of coins of denomination d_i that are used, for $i = 1, \ldots, n$.

The total value of all the chosen coins must be exactly equal to T. We want to **minimize** the number of coins used, which is denoted by N.

A Greedy Algorithm for Coin Changing

Algorithm: *GreedyCoinChanging*$(D : array; T : integer)$

comment: $D = [d_1, \ldots, d_n]$

rename the coins, by sorting if necessary, so that $d_1 > \cdots > d_n$

$N \leftarrow 0$

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

do $\begin{cases} a_i \leftarrow \left\lfloor \frac{T}{d_i} \right\rfloor \\ T \leftarrow T - a_i d_i \\ N \leftarrow N + a_i \end{cases}$

if $T > 0$

then return $(fail)$

else return $([a_1, \ldots, a_n], N)$