CS 341: Algorithms
Module 5: Greedy Algorithms

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

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 $f : feasible(I) \rightarrow \mathbb{R}^+ \bigcup 0$. 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 solution

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. A tuple $[x_1, x_2, \ldots x_i]$ 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 $X = [x_1, x_2, \ldots x_i]$ where $i < n$, we define the choice set

$$choice(X) = \{y \in \mathcal{X} : [x_1, x_2, \ldots x_i, y] \text{ is a partial solution}\}.$$

The Greedy Method (cont.)

local evaluation criterion

A local evaluation criterion is a function g such that, for any partial solution $X = [x_1, ..., x_i]$ and any $y \in choice(X)$, $g(x_1, ..., x_i, y)$ measures the cost or profit of extending the partial solution X to include y.

extension

Given a partial solution $X = [x_1, x_2, \ldots x_i]$ 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 $[x_1, x_2, \ldots x_i, y]$.

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.

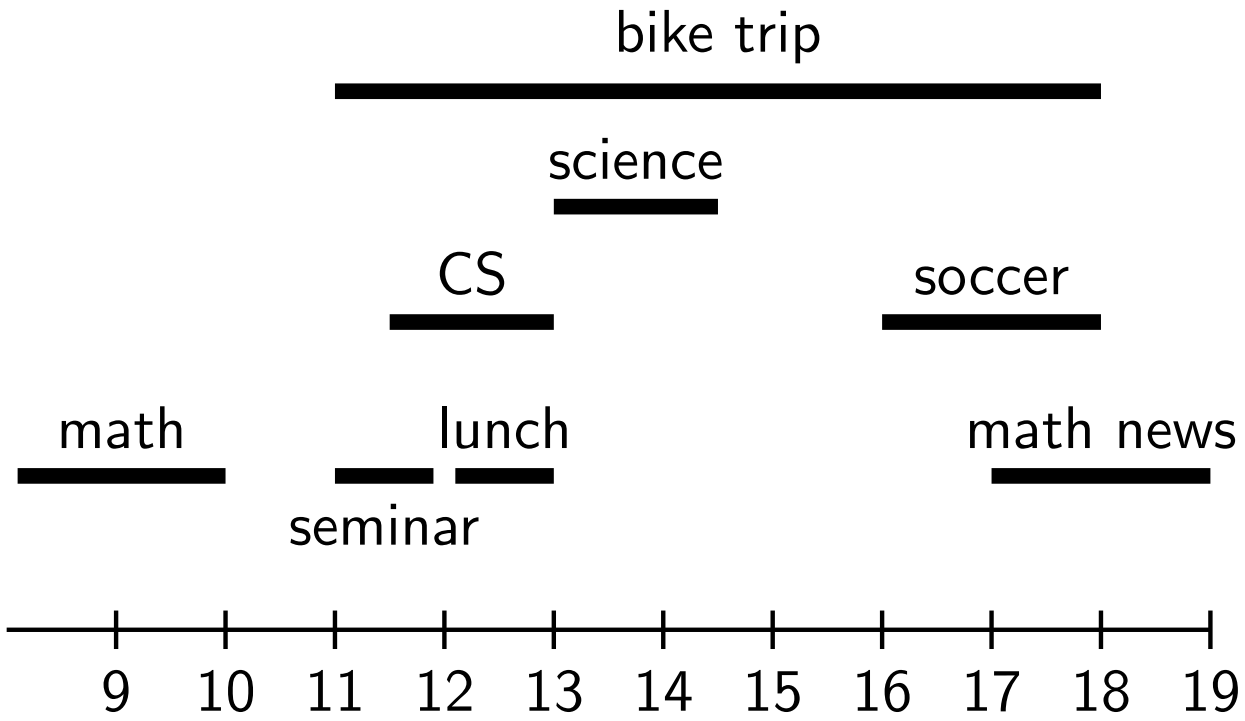
For a greedy algorithm to be efficient, we need a fast way to find the best extension.
Note that for a greedy algorithm to be "correct", it has to find the optimal solution for every problem instance.

Interval Scheduling or "Activity Selection"

Problem

Given a set of activities, each with a specified time interval, select a maximum set of disjoint (non-intersecting) intervals.

Example:



There are several possible greedy approaches.

- select activity that starts earliest (find counterexample).
- select the shortest interval (find counterexample).
- select the interval with fewest conflict (find counterexample).
- select the interval that ends earliest.

In our example we get math, seminar, lunch, science, soccer.

Algorithm 1: ActivitySelection

```
1 Sort activities 1..n by end time
2 A ← ∅
3 for i = 1..n
4     if {activity i does not overlap with any previous
5        (check the last one) activity in A } then
6         A ← A ∪ {i}
7 return(A)
```

Correctness

Lemma

ActivitySelection algorithm returns a max size set A of disjoint intervals.

Proof

Let $A = \{a_1, \ldots, a_k\}$ sorted by end time.
compare to an optimum solution $B = \{b_1, \ldots, b_\ell\}$ ordered by end time. Thus $\ell \geq k$ and we want to prove $\ell = k$.
Idea: at every step we can do better with a's.
Claim: $\forall i$, $a_1, \ldots, a_i, b_{i+1}, \ldots, b_\ell$ is an opt. solution.

Correctness (cont.)

Proof (cont.)

Proof by induction:
basis: $i = 1$. a_1 had earliest end time of all intervals so,

$$end(a_1) \leq end(b_1).$$

Hence replacing b_1 by a_1 gives disjoint intervals.
Induction step: suppose $a_1, \ldots, a_{i-1}, b_i, \ldots, b_\ell$ is an opt. solution. b_i does not intersect a_{i-1} so the greedy algorithm could have chosen it. Instead, it chose a_i so

$$end(a_i) \leq end(b_i)$$

and replacing b_i by a_i leaves disjoint intervals.
To finish proving the lemma note that if $k < \ell$ then $a_1, \ldots, a_k, b_{k+1}, \ldots, b_\ell$ is an opt. solution. But then the greedy algorithm had more choices after a_k.

Coin Changing

Problem

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

Divide and Conquer? How?

Algorithm 2: GreedyCoins1 (What's wrong?)

```
1  rename the coins, by sorting if necessary, so that d_1 > ... > d_n and set
     all a_i to 0;
2  i := 1; N := 0;
3  while T > 0 & i ≤ n do
4      if T > d_i then
5          T := T − d_i; a_i++; N++;
6      else
7          i++;
8  if T > 0 then
9      return (FAIL)
10 else
11     return ([a_1, ..., a_n], N)
```

GreedyCoins2

Algorithm 3: GreedyCoins2

1 rename the coins, by sorting if necessary, so that $d_1 > \ldots > d_n$;
2 $N := 0$;
3 **for** $i = 1$ **to** n **do**
4 $\quad a_i := \left\lfloor \frac{T}{d_i} \right\rfloor$;
5 $\quad T := T - a_i d_i$;
6 $\quad N := N + a_i$;
7 **if** $T > 0$ **then**
8 $\quad$ **return** (FAIL)
9 **else**
10 $\quad$ **return** $([a_1, \ldots, a_n], N)$

We will generally assume that there is a coin with value 1, so we can make change for any T. For some sets of coins, the greedy algorithm always gives the correct solution. Some examples:

1. If $d_j | d_{j-1}$ for all $j, 2 \leq j \leq n$, then greedy = optimal.
2. If $D = [200, 100, 25, 10, 5, 1]$ (Canadian coin system, including pennies) then greedy = optimal.
3. If $D = [30, 24, 12, 6, 3, 1]$ (old English system), then (sometimes) greedy $\neq$ optimal. For example, suppose $T = 48$. The greedy algorithm yields $30, 12, 6$ whereas the optimal solution is $24, 24$.

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!