CS 5321: Advanced Algorithms – Greedy Algorithms

Ali Ebnenasir
Department of Computer Science
Michigan Technological University

Acknowledgement

- Eric Torng
- Moon Jung Chung
- Charles Ofria

Greedy Algorithms

- Basic idea
- Theoretical foundation (Matroids)
- Connection to dynamic programming
- Proof Techniques

Basic Idea

- Always make a choice that looks best at the moment
- Hopefully …

Locally optimal choices will lead to a globally optimal solution

- A top-down strategy
- Not always leads to finding the optimal solution

Activity Selection Problem

Activity Selection Problem

- *n* activities need exclusive use of a shared resource; e.g., shared use of a classroom
- Activities a_1, a_2, …, a_n
- Each activity a_i occurs in a specific ***half-open*** time interval $[s_i , f_i)$, where
  - s_i is the start time of a_i
  - f_i is the finish time of a_i
- Two intervals a_i and a_j are ***compatible*** (non-overlapping) if
  - $s_i >= f_j$, or
  - $s_j >= f_i$

Goal: Find the largest possible set of compatible activities

Possible Greedy Strategies

- Choose the shortest interval and recurse
- Choose the interval that starts earliest and recurse
- Choose the interval that ends earliest and recurse
- Choose the interval that starts latest and recurse
- Choose the interval that ends latest and recurse
- Do any of these strategies work?

Earliest End Time Algorithm

- Sort intervals by end time
  - **A**: Choose the interval with the earliest end time breaking ties arbitrarily
  - Prune intervals that overlap with this interval and goto **A**
- Running time?

Example: Activity Selection Problem

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

- What are the solutions?
- Is the solution unique?

Finding an Optimal Substructure

- Let S_{ij} be the subset of compatible activities between a_i and a_j
  - I.e., set of activities that start after a_i finishes and finish before a_j starts
  - $S_{ij} = \{a_k: f_i =< s_k < f_k =< s_j\}$
- Wlog, consider fictitious activities a_0 and a_{n+1} such that $f_0 = 0$ and $s_{n+1} = \infty$
- Prune the subproblem space by excluding the instances S_{ij} of the problem where i >= j
  - i.e., if i >= j then $S_{ij} = \emptyset$

Finding an Optimal Substructure

- Thus, we focus on instances where we have
  $$f_0 =< f_1 =< f_2 \ \ldots \ =< f_n < f_{n+1}$$
- Consider an optimal subproblem S_{ij} with a solution A that contains a_k
- Consider two subproblems S_{ik} and S_{kj}
- A_{ik} and A_{kj} should be optimal subsolutions. Why?
- Therefore, we have
  $$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$$

A Recursive Solution

- Let c[i, j] be the maximum number of compatible activities in S_{ij}
- Then we have

1. $c[i, j] = 0$ if $S_{ij} = \emptyset$

2. $c[i, j] = \max_{i<k<j}\{c[i, k] + c[k, j] + 1\}$ if $S_{ij} \neq \emptyset$

Simplifying the Solution

- *Theorem*
  - Consider any non-empty subproblem S_{ij}, and
  - Let $f_m = \min\{f_k : a_k \in S_{ij}\}$
  - Then
    1. Activity a_m is used in some maxium-size subset of compatible activities of S_{ij}
    2. Subproblem S_{im} is empty, thus choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty

What is the significance of this theorem?

A Recursive Greedy Routine

REC-ACTIVITY-SELECTOR(s, f, i, n)
$m \leftarrow i + 1$
while $m \leq n$ and $s_m < f_i$ ▷ Find first activity in $S_{i,n+1}$.
  do $m \leftarrow m + 1$
if $m \leq n$
  then return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR(s, f, m, n)
  else return $\emptyset$

Initial call: REC-ACTIVITY-SELECTOR$(s, f, 0, n)$.

What is the worst-case complexity?
Can we transform it to an iterative algorithm?

Proof of Optimality

- For any instance of the problem, there exists an optimal solution that includes an interval with earliest end time
  - Let S be an optimal solution
  - Suppose S does not include an interval with the earliest end time
  - We can swap the interval with earliest end time in S with an interval with earliest overall end time to obtain feasible solution S'
  - S' has at least as many intervals as S and the result follows
- We recursively apply this observation to the subproblem induced by selecting an interval with earliest end time to see that greedy produces an optimal schedule

Shortest Interval Algorithm

- Sort intervals by interval length
  - **A**: Choose the interval with shortest length breaking ties arbitrarily
  - Prune intervals that overlap with this interval and goto **A**
- Running time?

Proof of Optimality?

- For any instance of the problem, there exists an optimal solution that includes a shortest interval
  - Let S be an optimal solution
  - Suppose S does not include a shortest interval
  - Can we produce an equivalent solution S' from S that includes a shortest interval?
  - If not, how does this lead to a counterexample?

Knapsack Problem

Knapsack

- Given n items with
  - weights $w_1, \ldots, w_n$
  - values $v_1, \ldots, v_n$, and
  - a knapsack with maximum capacity W
  - *Which items would you put in the knapsack to maximize the value?*
- Two versions:
  - *0-1 (all or nothing)*: either take an item or leave it
  - *fractional*: portions of an item can be taken

Formulating Knapsack

- 0-1 knapsack:

  Identify $x_1, \ldots, x_n$ (where $x_i = 0$ or 1) such that

  maximize $\Sigma_{i=1}^{n} v_i x_i$ while satisfying

  $\Sigma_{i=1}^{n} x_i w_i <= W$

- Fractional knapsack:

  Identify $x_1, \ldots, x_n$ $(0 =< x_i =< 1)$ such that

  maximize $\Sigma_{i=1}^{n} v_i x_i$ while satisfying

  $\Sigma_{i=1}^{n} x_i w_i <= W$

Optimal Substructure

- What is an optimal substructure for knapsack?
- 0-1 knapsack:
  - given an optimal load, remove an item j
  - What is the weight of the remaining subsolution?
    - $W - w_j$ should be an optimal load for the subsolution
- Fractional knapsack:
  - given an optimal load, remove a portion x_j of item j
  - What is the weight of the remaining subsolution?
    - $W - (x_j . w_j)$ should be an optimal load for the subsolution
- What are the differences?

A Greedy Solution

- What would be a strategy for fractional knapsack?
  - Calculate the $\alpha_j = v_j/w_j$ ratio
  - Sort items based on α_j
  - Pick the item with maximum α_j
- Example: W = 50

	1	2	3
v_j	60	100	120
w_j	10	20	30
α_j	6	5	4

Optimal value = 60 + 100 + 80 = 240

Does this greedy strategy work for the 0-1 knapsack? Why?

Matroids: A Theoretical Foundation for Greedy Algorithms

Definitions

- Matroid:
  an ordered pair M = (S, L), where
  - S is a finite set of elements, and
  - L is a non-empty set of subsets of S, and
  - the following properties hold:
    - **Independence** property, and
    - **Exchange** property.

Independence and Exchange Properties

- Independence:

$$(Y \in L) \wedge (X \subseteq Y) \Rightarrow (X \in L)$$

  - We say L is *hereditary* if the independence property holds

- Exchange: M satisfies the exchange property if

$$((Y \in L) \wedge (X \in L) \wedge (|X|<|Y|)) \Rightarrow$$
$$(\exists\, a: a \in (Y - X):\ (X \cup \{a\} \in L))$$

Example: Graphic Matroids

- $M_G = (S_G, L_G)$ for an undirected graph $G = (V, E)$
  - S_G is E
  - L_G is the set of subsets of edges of E that are acyclic
  - Independence: a set X of edges is independent iff (if and only if) the subgraph of G induced by X is a forest
- Why is $M_G = (S_G, L_G)$ a Matroid?
- We have to show that independence & exchange hold

Example: Graphic Matroids - Continued

- Independence is due to the acyclic nature of subsets
  - Removing an edge from a set does not create cycles
- Exchange property
  - Consider two forests X and Y of G such that $|Y| > |X|$
  - Show that
    - there is a tree in Y that has vertices in two trees in X
    - there is an edge e that connects two different trees in Y
    - E cannot create a cycle. Why?

Extensions and Maximal Independent Subsets

- In a Matroid M = (S, L), an element $e \notin X$ is an ***extension*** of $X \in L$ if $X \cup \{e\} \in L$ (i.e., $X \cup \{e\}$ is independent)
- In the context of graphic Matroids?
- If X is an independent subset in M, then X is ***maximal*** if it has no extensions.
- All maximal independent subsets in a Matroid have the same size. Why?
- What is a maximal independent subset in a graphic Matroid?

Weighted Matroids

- A weight function assigns a strictly positive weight to each element of S in a Matroid M = (S, L)
- The weight of a subset X in L is

$$w(X) = \sum_{a \in X} w(a)$$

- Think of this in the context of graphic Matroids!
  - How can this notion be used?

Optimal Subset

- An optimal subset of a Matroid M = (S, L) is a independent subset $X \in L$ whose weight is maximum
  - This is because the weight of each element is positive
- Example: graphic Matroid for a graph G = (V, E)
  - What is an optimal subset whose weight is minimum?
  - How do we compute it?
  - Define an inverted weight function
  - $w'(X) = (|V|-1)\, w_m - w(X)$, where w_m is greater than the maximum possible weight of all elements

Matroid-based Greedy Algorithm

- GREEDY(M, w) // M = (S, L) is a matroid
  // w is its weight function

1. $X \leftarrow \varnothing$
2. Sort all element of M in a monotonically decreasing order by weight w
3. For each element $a \in X$ selected in order
   - If $X \cup \{a\} \in L$ then $X \leftarrow X \cup \{a\}$
4. Return X

Does GREEDY return an optimal subset? (i.e., an independent subset with maximum weight)

Theorems

- Let x be the first element picked by GREEDY for a weighted Matroid M = (S, L). Finding an optimal subset containing x amounts to finding an optimal subset in the weighted Matroid M' = (S', L'), where
  - $S' = \{y \in S: \{x, y\} \in L\}$,
  - $L' = \{A \subseteq (S - \{x\}): A \cup \{x\} \in L\}$, and
  - M' has the same weight function.
- GREEDY returns and optimal subset.

Task Scheduling with Deadlines

Task Scheduling Problem

- Given is a set S of n tasks $\{a_1, \ldots, a_n\}$ and a single processor
- Each task takes a unit time to execute
- Tasks have deadlines $d_1, \ldots, d_n$
- Tasks have non-negative penalties $w_1, \ldots, w_n$ if miss the deadline

- Goal: find a schedule S where the total penalty of the tasks that miss their deadline is minimized
  - I.e., the total penalty of the tasks that meet their deadline is maximized

Task Scheduling – Basic Concepts

- *Late* task: a task that finishes after its deadline
- *Early* task: a task that is not late
- *Early-first form*:
  - a schedule in which the early tasks precede the late ones
- *Canonical form*:
  1. a schedule that has an early-first form, and
  2. the early tasks are scheduled in order of monotonically increasing deadlines
- *Independence*:
  - a set A of tasks is independent if there is a schedule for tasks in A such that no tasks are late

Task Scheduling – Basic Concepts

- A set *A* is independent iff $N_t(A) <= t$
  - where $N_t(A)$ is the number of tasks whose deadline is t or earlier

- A set *A* is independent iff
  - if the tasks in *A* are scheduled in order of monotonically increasing deadlines
  - then no task is late

- *Observation*: if a set is independent then all its subsets are independent as well

- Given a set *A*, what is the worst-case complexity of verifying the independence property of *A*?

Task Scheduling – Basic Concepts

- A task a^* is an *extension* of an independent set of A if $A \cup \{a^*\}$ is also independent
- An independent set A is *maximal* if it has no extensions
- *Theorem*:
  - All maximal independent subsets have the same size

A Greedy Strategy

1. Find a maximal set A of independent tasks that are early in the optimal schedule
2. List the elements of A in order of monotonically increasing deadline
3. Append the late tasks in an arbitrary order to create a canonical order of the optimal schedule

Greedy Strategy – Step 1

1. Find a maximal set A of independent tasks that are early in the optimal schedule

```
Greedy(S)
{
    1) A := ∅
    2) Sort tasks in order of monotonically decreasing penalty
    3) For each task a_i taken in order
        3-1) If A ∪ {a_i} is independent then A := A ∪ {a_i};
    4) return A
}
```

Example: Task Scheduling

1. $\{a_1\}$
2. $\{a_1, a_2\}$
3. $\{a_1, a_2, a_3\}$
4. $\{a_1, a_2, a_3, a_4\}$
5. $\{a_1, a_2, a_3, a_4\} \cup \{a_5\}$ not an independent set!
6. $\{a_1, a_2, a_3, a_4\} \cup \{a_6\}$ not an independent set!
7. $\{a_1, a_2, a_3, a_4, a_7\}$

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Example: Task Scheduling – cont'd

- Greedy Strategy - Step 2
  - List the elements of A in order of monotonically increasing deadline
- $A = \{a_1, a_2, a_3, a_4, a_7\}$
- Ordered set $\{a_2, a_4, a_1, a_3, a_7\}$
- Optimal schedule $\{a_2, a_4, a_1, a_3, a_7, a_5, a_6\}$

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	70	60	50	40	30	20	10

Making Change

Example: Making Change

- Input
  - Positive integer n
- Task
  - Compute a minimal multiset of coins from C = {d_1, d_2, d_3, …, d_k} such that the sum of all coins chosen equals n
- Example
  - n = 73, C = {1, 3, 6, 12, 24}
  - Solution: 3 coins of size 24, 1 coin of size 1

Dynamic Programming Solution 1

- Subsolutions: T(k) for $0 \leq k \leq n$
- Recurrence relation
  - $T(n) = \min_i (T(i) + T(n-i))$
  - $T(d_i) = 1$
  - Linear array of values to compute
    - Time to compute each entry?

Dynamic Programming Solution 2

- Subsolutions: T(k) for $0 \leq k \leq n$
- Recurrence relation
  - $T(n) = \min_i (T(n-d_i) + 1)$
    - There has to be a “first/last” coin
  - $T(d_i) = 1$
  - Linear array of values to compute
    - Time to compute each entry?

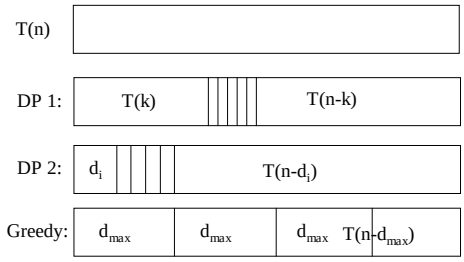
Greedy Solution

- From dynamic programming 2:

$$T(n) = \min_i (T(n-d_i) + 1)$$

- Key observation
  - For many (but not all) sets of coins, the optimal choice for the first/last coin d_i will always be the maximum possible d_i
  - That is, $T(n) = T(n-d_{max}) + 1$ where d_{max} is the largest $d_i \leq n$
- Algorithm
  - Choose largest d_i smaller than n and recurse

Comparison



Example 1: Making Change Proof 1

- Greedy is optimal for coin set C = {1, 3, 9, 27, 81}
- Structural property of any optimal solution:
  - In any optimal solution, the number of coins of denomination 1, 3, 9, and 27 must be at most 2.
  - Why?
- This structural property immediately leads to the fact that the greedy solution must be optimal
  - Why?

Example 1: Making Change Proof 2

- Greedy is optimal for coin set C = {1, 3, 9, 27, 81}
- Let S be an optimal solution and G be the greedy solution
- Let A_k denote the number of coins of size k in solution A
- Let *kdiff* be the largest value of k s.t. $G_k \neq S_k$
- Claim 1: $G_{kdiff} \geq S_{kdiff}$. Why?
- Claim 2: For some $d_i < d_{kdiff}$, we should have $S_i \geq 3$. Why?
- Claim 3: We can create a better solution than S by performing a "swap". What swap?
- These three claims imply *kdiff* does not exist and G_k is optimal.

Proof that Greedy is NOT optimal

- Consider the following coin set
  - C = {1, 3, 6, 12, 24, 30}
- Prove that greedy will not produce an optimal solution for all n
- What about the following coin set?
  - C = {1, 5, 10, 25, 50}

Greedy Technique

- When trying to solve a problem, make a local greedy choice that optimizes progress towards global solution and recurse
- Implementation/running time analysis is typically straightforward
  - Often implementation involves use of a sorting algorithm or a data structure to facilitate identification of next greedy choice
- Proof of optimality is typically the hard part

Proofs of Optimality

- We will often prove some structural properties about an optimal solution
  - Example: Every optimal solution to the activity selection problem has a task with earliest end time
- We will often prove that **an** optimal solution is the one generated by the greedy algorithm
  - If we have an optimal solution that does not obey the greedy constraint, we can "swap" some elements to make it obey the greedy constraint
- Always consider the possibility that greedy is not optimal and consider counter-examples

Example: Minimizing Sum of Completion Times

- Input
  - Set of n jobs with lengths x_i
- Task
  - Schedule these jobs on a single processor so that the sum of all job completion times are minimized
- Example
  - {2, 1, 3}
  - Solution:
    Completion times: 3, 1, 6 for a sum of 10
- Develop a greedy strategy and prove it is optimal

Questions

- What is the running time of your algorithm?
- Does it ever make sense to preempt a job? That is, start a job, interrupt it to run a second job (and possibly others), and then finally finish the first job?
- Can you develop a swapping proof of optimality for your algorithm?