Module Greedy Algorithms

Thanks to Anna Lubiw and other previous CS 341 instructors.

- Optimization Problems
- Greedy Algorithms
- Intro Example: Making Change
- Minimizing Completion Time
- Interval Scheduling
- Exchange Proof
- Fractional Knapsack

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).

Making Change

Problem

Making Change

Instance: *A set C of coin denominations for a coin system and a given amount M.*

Find: *The minimum number of coins of denominations from C that sum to M.*

For example: Make change for $3.47 using the Canadian coin system.

How did you make your choice for each coin?
Is your solution the minimal number of coins possible?
Does this work for all coin systems?

Greedy Algorithms

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*

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

Greedy Algorithms

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 $y \in \mathit{choice}(X)$ 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.

Greedy Algorithms

- 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!

Minimizing Completion Time

Problem

Minimizing Completion Time
Instance: *A set of jobs* $\{1, \ldots, n\}$ *with processing times* $t()$*; i.e. job* i *has processing time* $t(i)$*.*
Find: *An ordering of the jobs that minimizes the sum of completion times for all jobs* T*. Also, give* T*.*

For example: $n = 5$ with processing times $[2, 8, 1, 10, 5]$
If processed in order 1, 2, 3, 4, 5:

- $T = 2 + (8+2) + (1+8+2) + (10+1+8+2) + (5+10+1+8+2) = 70$

In order 3, 1, 2, 5, 4 processing times are $[1, 2, 8, 5, 10]$:

- $T = 1 + (2+1) + (8+2+1) + (5+8+2+1) + (10+5+8+2+1) = 57$

In order 3, 1, 5, 2, 4 processing times are $[1, 2, 5, 8, 10]$:

- $T = 1 + (2+1) + (5+2+1) + (8+5+2+1) + (10+8+5+2+1) = 54$

Greedy Algorithm

Algorithm: Order jobs in **non-decreasing** order of processing times.

Problem

To prove optimal

- *Let $L = [j_1, \ldots, j_n]$ be an ordering of the jobs that is not in non-decreasing order of processing times.*
- *Then there exists some i where $t(j_i) \geq t(j_{i+1})$.*
- *Show that we can find a better solution (or at least no worse) by inverting this pair.*
- *We can then continue to invert pairs until the order is in non-decreasing order of processing time, concluding our algorithm is optimal.*

To Prove Optimal

There exists some i where $t(j_i) \geq t(j_{i+1})$.

1. Note: the sum of time completions before job i and after job $i+1$ remain the same in both initial and modified orderings.
2. Let $T_{before} = \sum_{j=j_1 \ldots j_{i-1}} t(j)$
3. In L: $(T_{before} + t(j_i)) + (T_{before} + t(j_i) + t(j_{i+1}))$
   Inverted: $(T_{before} + t(j_{i+1})) + (T_{before} + t(j_{i+1}) + t(j_i))$
4. Change: $t(j_{i+1}) - t(j_i) \leq 0$ since $t(j_i) \geq t(j_{i+1})$
   These two jobs are now in non-decreasing order of processing time – One step closer to our solution.
   If $t(j_i) > t(j_{i+1})$ then inverted is a better solution; otherwise, it is no worse.

Recall from CS240

- Optimal static order for linked list implementation of dictionaries
- Same result (up to reverse), same proof

Interval Selection

Problem

Interval Scheduling *or* ***Activity Selection***

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

Find: *A subset $S \subseteq \mathcal{I}$ of pairwise disjoint intervals of maximum size (i.e., one that maximizes $|S|$).*

Possible Greedy Strategies for Interval Scheduling

1. Select the activity/interval that has the *earliest start time*; i.e. local evaluation criterion is s_i.
2. Select the activity that has the *shortest length*; i.e. the local evaluation criterion is $f_i - s_i$.
3. Select the activity with the *fewest conflicts* with other activities.
4. Select the activity with the *earliest finishing time*; i.e. the local evaluation criterion is f_i.

Note: Choices above also assume that the selection chosen is also disjoint from all previously chosen activities.

Does one of these strategies yield a correct greedy algorithm?

Select Interval with Earliest Finish Time

1. Sort intervals $1..n$ by finish time and relabel so $f_1 \leq \ldots \leq f_n$
2. $S = \emptyset$
3. **for** $i \leftarrow 1$ **to** n **do**
4. **if** interval i is pairwise disjoint with all intervals in S **then**
5. $S \leftarrow S \cup \{i\}$

Analysis: $O(n \log n)$ to sort $+$ $O(n)$ loop $\Rightarrow O(n \log n)$

Correctness: 2 approaches

1. Greedy always stays ahead
2. "Exchange" proof

Proof of Correctness - Greedy always stays ahead

Lemma: The greedy algorithm (select earliest finish time) returns a maximum size set A of disjoint activities.

Proof: Let $A = \{a_1, \ldots, a_k\}$, sorted by finish time.

Compare A to an optimum solution $B = \{b_1, \ldots, b_\ell\}$, sorted by finish time. Thus, $\ell \geq k$ and we want to prove $\ell = k$.

Proof of Correctness - Greedy always stays ahead

Lemma: The greedy algorithm (select earliest finish time) returns a maximum size set A of disjoint activities.

Proof: Let $A = \{a_1, \ldots, a_k\}$, sorted by finish time.

Compare A to an optimum solution $B = \{b_1, \ldots, b_\ell\}$, sorted by finish time. Thus, $\ell \geq k$ and we want to prove $\ell = k$.

Idea: At every step i, we can do at least as well by choosing a_i.

Claim: $a_1, \ldots, a_i, b_{i+1}, \ldots, b_\ell$ is an optimal solution for all i.

Greedy always stays ahead - Induction!

Basis: $i = 1$

a_1 had the earliest finish time of all activities so $\mathit{finish}(a_1) \leq \mathit{finish}(b_1)$.
Thus, a_1 is disjoint from all b_i for $2 \leq i \leq \ell$.
Thus, we can replace b_1 with a_1.

Induction Step: Suppose $a_1, \ldots, a_{i-1}, b_i, \ldots, b_\ell$ is an optimal solution.

Greedy always stays ahead - Induction!

Basis: $i = 1$

a_1 had the earliest finish time of all activities so $\mathit{finish}(a_1) \leq \mathit{finish}(b_1)$.
Thus, a_1 is disjoint from all b_i for $2 \leq i \leq \ell$.
Thus, we can replace b_1 with a_1.

Induction Step: Suppose $a_1, \ldots, a_{i-1}, b_i, \ldots, b_\ell$ is an optimal solution.

b_i does not intersect a_{i-1} so, the greedy algorithm could have chosen b_i; however, it chose a_i instead, so $\mathit{finish}(a_i) \leq \mathit{finish}(b_i)$.

a_i is then also disjoint from from all b_k for $i + 1 \leq k \leq \ell$.
Thus, we can replace b_i with a_i.

Greedy always stays ahead - Induction!

Basis: $i = 1$
a_1 had the earliest finish time of all activities so $\mathit{finish}(a_1) \leq \mathit{finish}(b_1)$.
Thus, a_1 is disjoint from all b_i for $2 \leq i \leq \ell$.
Thus, we can replace b_1 with a_1.

Induction Step: Suppose $a_1, \ldots, a_{i-1}, b_i, \ldots, b_\ell$ is an optimal solution.

b_i does not intersect a_{i-1} so, the greedy algorithm could have chosen b_i; however, it chose a_i instead, so $\mathit{finish}(a_i) \leq \mathit{finish}(b_i)$.

a_i is then also disjoint from from all b_k for $i + 1 \leq k \leq \ell$.
Thus, we can replace b_i with a_i.

This proves the claim. To finish proving the lemma we argue that if $k < \ell$ then $a_1, \ldots, a_k, b_{k+1}, \ldots, b_\ell$ is an optimal solution. But then the greedy algorithm would have more choices after a_k, so k must equal ℓ.

Scheduling to Minimize Lateness

Suppose you are given a number of tasks to complete:

Job	Time Required	Deadline
CS341	4 hours	in 9 hours
Stat231	2 hours	in 6 hours
Psych	4 hours	in 14 hours
CS350	10 hours	in 25 hours

Can you do everything by its deadline?
Greedy Strategy?
Can we generalize this problem?

Scheduling to Minimize Lateness

Problem

Scheduling to Minimize Lateness
Instance: *A set of jobs* $\{1, \ldots, n\}$ *where job* i *requires time* t_i *to complete and has a deadline of* d_i.
Find: *A schedule, allowing some jobs to be late but minimizing the maximum lateness.*

Note: this is different from minimizing the sum of lateness or minimizing average lateness.

A schedule computes all jobs on time $\iff$ its maximum lateness is 0.

Exchange Proofs

General Idea: Show how we can covert an optimal solution into the greedy solution.

- Let G be the solution produced by the greedy algorithm.
  Let O be an optimal solution.
- If G is the same as O then greedy is also optimal.
  If $G \neq O$ then find a pair of items that are out of order in O when compared with G.
- Show that by *exchanging* the order of these two items, we create a new solution that is better (or at least no worse); i.e. the resulting solution remains optimal.
  Note: the reasoning is typically based on how the greedy algorithm makes its choice.
- By making a number of exchanges we will obtain the greedy solution (similar to bubblesort) and since each exchange makes the solution no worse, the greedy algorithm is also optimal.

Knapsack Problems

Problem

Knapsack
Instance: *A set if items* $1, \ldots, n$ *with values* $v_1, \ldots, v_n$*, weights* $w_1, \ldots, w_n$ *and a capacity,* W*. These are all positive integers.*
Feasible solution: *An n-tuple* $X = [x_1, \ldots, x_n]$ *where* $\sum_{i=1}^{n} w_i x_i \leq W$*.*
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*** *or* ***Fractional 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* $\sum_{i=1}^{n} v_i x_i$*.*

Note: $\mathbb{Q}$ is the set of rational numbers.

Possible Greedy Strategies for Knapsack Problems

1. Consider the items in decreasing order of value (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 value divided by weight (i.e., the local evaluation criterion is v_i/w_i).

Does one of these strategies yield a correct greedy algorithm for the **Fractional Knapsack** problem?

Knapsack Problems

Consider the following example where capacity $W = 6$.

Item i	Value v_i	Weight w_i	v_i/w_i
1	12	4	3
2	7.5	3	2.5
3	6	3	2

Does ordering by value per weight help?

Knapsack Problems

Consider the following example where capacity $W = 6$.

Item i	Value v_i	Weight w_i	v_i/w_i
1	12	4	3
2	7.5	3	2.5
3	6	3	2

Does ordering by value per weight help?
Fractional Knapsack: choosing highest value per weight is optimal.

Note: none of the greedy choices seem to be optimal for the 0-1 Knapsack problem.

Greedy Algorithm for Fractional Knapsack

Greedy Algorithm: Choose item with highest value per weight and choose as much of it as possible.

x_i is the weight of item i taken

1. Sort items $1..n$ by value per weight and relabel so $(v_1/w_1) \geq \ldots \geq (v_n/w_n)$
2. $freeW \leftarrow W$
3. **for** $i \leftarrow 1$ **to** n **do**
4. $\quad x_i \leftarrow min\{w_i, freeW\}$
5. $\quad freeW \leftarrow freeW - x_i$

A solution then looks like

Item:	1	2	...	j	$j+1$	...	n
Weight Taken:	x_1	x_2	...	x_j	0	...	0

Final weight is $\sum x_i = W$ (if $\sum w_i \geq W$)
Final value: $\sum \frac{v_i}{w_i} x_i$
Running time: $O(n \log n)$ to sort, $O(n)$ to choose weights for each item.

Greedy Algorithm for Fractional Knapsack is correct

Claim: The greedy algorithm gives the optimal solution to the fractional knapsack problem.

Proof: Assume items are ordered by $\frac{v_i}{w_i}$.
Let the greedy solution be $x_1, x_2, \ldots, x_{k-1}, x_k, \ldots, x_\ell, \ldots, x_n$.
Let an optimal solution be $y_1, y_2, \ldots, y_{k-1}, y_k, \ldots, y_\ell, \ldots, y_n$.

Suppose y is an optimal solution that
matches x on a maximum number of indices, say M indices.

If $M = n$ then we are done, so assume $M < n$; i.e. this implies the greedy solution is not optimal

Greedy Algorithm for Fractional Knapsack is correct

Claim: The greedy algorithm gives the optimal solution to the fractional knapsack problem.

Proof: Assume items are ordered by $\frac{v_i}{w_i}$.
Let the greedy solution be $x_1, x_2, \ldots, x_{k-1}, x_k, \ldots, x_\ell, \ldots, x_n$.
Let an optimal solution be $y_1, y_2, \ldots, y_{k-1}, y_k, \ldots, y_\ell, \ldots, y_n$.

Suppose y is an optimal solution that
matches x on a maximum number of indices, say M indices.

If $M = n$ then we are done, so assume $M < n$; i.e. this implies the greedy solution is not optimal (so we should then be able to find a contradiction).

Contradiction: show that there exists an optimal solution that matches x on at least $M + 1$ indicies.

The Stable Marriage Problem

Note: rephrased using co-op students and employers offering jobs.

Problem

Stable Marriage

Instance: *A set of n co-op students $S = [s_1, \ldots, s_n]$, and a set of n employers offering jobs, $E = [e_1, \ldots, e_n]$.*

Each employer e_i has a preference ranking of the n students, and each student s_i has a preference ranking of the n employers:

***pref**$(e_i, j) = s_k$ if s_k is the j-th preference of employer e_i and*

***pref**$(s_i, j) = e_k$ if e_k is the j-th favourite employer of student s_i.*

Find: *A matching of the n students with the n employers such that there does not exist a pair (s_i, e_j) who are not matched 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

- Employers offer jobs to students.
- If a student accepts a job offer, then the pair are **matched**; the student is employed.
- An unemployed student **must accept** a job if they are offered one.

Overview of the Gale-Shapley Algorithm

- Employers offer jobs to students.
- If a student accepts a job offer, then the pair are **matched**; the student is employed.
- An unemployed student **must accept** a job if they are offered one.
- However, if an employed student receives an offer from an employer whom they prefer to their current match, then they **cancel** their existing match and the student becomes employed by (matched with) their new employer; the previous employer no longer has a match.

Overview of the Gale-Shapley Algorithm

- Employers offer jobs to students.
- If a student accepts a job offer, then the pair are **matched**; the student is employed.
- An unemployed student **must accept** a job if they are offered one.
- However, if an employed student receives an offer from an employer whom they prefer to their current match, then they **cancel** their existing match and the student becomes employed by (matched with) their new employer; the previous employer no longer has a match.
- If an employed student receives an offer from an employer, but they prefer the job they already have, the offer is **rejected**.
- Matched/Employed students never become unmatched/unemployed.
- An employer might make a number of offers (up to n); the order of the offers is determined by the employer's preference list.

Gale-Shapley Algorithm

Gale-Shapley$(S, E, pref)$

$Match \leftarrow \emptyset$
while there exists an employer e_i still looking to hire **do**
    Let s_j be the next student in e_i's preference list
    if s_j is unemployed **then**
        $Match \leftarrow Match \cup \{(e_i, s_j)\}$
    else
        if s_j prefers e_i (over their current match e_k) **then**
            $Match \leftarrow Match\{(e_k, s_j)\} \cup \{(e_i, s_j)\}$
        Note: employer e_k is now looking to hire again
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 employer still looking to hire 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?