St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 1: Introduction
Fall 2021

1 What Are Approximation Algorithms?

This course is all about the topic of *approximation algorithms*. But before we get into what an "approximation algorithm" is, let's think back to our undergrad classes on algorithms and complexity theory.

You might recall that there exist certain problems that are very difficult for computers to solve in a reasonable amount of time. This is particularly the case for *optimization problems*, or problems where we want to find the best solution out of all possible solutions. For example, if you wanted a computer to find the shortest path between all of the capital cities of the world[1], you would be waiting quite a long time to get the answer. If you were to write the simplest-possible program, which just computes and compares all of the different paths through a set of n cities and returns the shortest path, your program would take $O(n!)$ time to complete its task. Since there are 193 countries (according to the United Nations), we would have 193 capital cities to visit, and our algorithm would need to perform on the order of $193! = 6.8514 \times 10^{358}$ checks before giving you the answer. To put this absolutely gargantuan number in perspective, it is estimated that there are between 10^{78} and 10^{82} atoms in the observable universe.

Obviously, then, the naïve approach won't work here. What if we tried a slightly more sophisticated approach? You might be familiar with dynamic programming, and we can put that technique to use on our present problem. First, we make an observation:

> Every subpath of a minimum-length path is itself of minimum length.

With this in mind, we can formulate our more sophisticated approach. We choose a start/end city, c_1, and for all subsets of cities S that don't include c_1, we compute the minimum distance of the path starting at c_1, visiting all cities in S, and ending at c_1. This approach[2] is clearly "better", in the sense that we're computing smaller subpaths of some cities instead of one big path containing all cities, but it's still not fast enough: our sophisticated algorithm will still take $O(2^n n^2)$ time, and exponentials like 2^n in our runtime analysis are generally no good.

It seems we're at a loss then. If we want an efficient algorithm that gives us an exact answer, then it seems we'd first need to make some incredible groundbreaking research discovery like $\mathsf{P} = \mathsf{NP}$. So, what can we do besides that?

Let's focus on one specific line from the last paragraph: *"an efficient algorithm that gives us an exact answer"*. Since we can't have the best of both worlds, we need to lose one of these attributes to gain the other. Our two attempts to find an algorithm for our earlier problem sacrificed *efficiency* to gain *exactness*. The other direction—gaining *efficiency* by sacrificing *exactness*—is at the core of approximation algorithms.

2 The Basics

In order to find an answer to a problem in a reasonable amount of time, an approximation algorithm makes certain concessions with respect to the precision of the answer it finds. If the optimal answer to the problem is denoted by OPT, then an approximation algorithm might give us an answer that is at most twice as bad

[1] This problem comes up quite frequently for traveling salespeople. If only they had a background in computer science instead of sales and marketing...

[2] Those of you familiar with this approach will recognize it as the Held-Karp algorithm.

as OPT (in the case where we want a minimal solution), or at most one-third as good as OPT (in the case where we want a maximal solution), or anything in between depending on how we construct the algorithm. We can formalize this notion in the following way.

Definition 1 (α-approximation algorithm)**.** An α-approximation algorithm for an optimization problem with optimal solution OPT is a polynomial-time algorithm that, for all instances of the problem, returns a solution with a value that is within a factor of α of the value of OPT.

The factor α is sometimes referred to as the *performance guarantee* of the algorithm. By convention, $\alpha > 1$ if our optimization problem involves minimization (finding the smallest solution), and $\alpha < 1$ if our optimization problem involves maximization (finding the largest solution).

Note that Definition 1 doesn't imply that there exists only one approximation algorithm for any given problem, nor does it imply that α is the only performance guarantee we can obtain for that problem. Indeed, for certain problems, we can find a whole family of approximation algorithms, and we can refine exactly how close-to-optimal we want our solution to be.

Definition 2 (Polynomial-time approximation scheme)**.** A polynomial-time approximation scheme, or PTAS, for an optimization problem is a family of approximation algorithms $\{A_\epsilon\}$ where, for each $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (for minimization problems) or a $(1 - \epsilon)$-approximation algorithm (for maximization problems) in the family.

At first glance, this seems like a great discovery! If we have a PTAS for an optimization problem, then we can choose any value $\epsilon > 0$ we want and we're guaranteed to have an approximation algorithm that gives us a solution within ϵ of the optimal solution. Even better, a PTAS exists for many common optimization problems, as we will see throughout this course.

However, if a PTAS existed for *every* optimization problem, then our course would essentially be over at this point. Unfortunately, there are certain problems for which approximation is quite difficult. This class of problems, known as MaxSNP, won't be defined formally here. However, it is known that any problem in MaxSNP has a fixed-ratio approximation algorithm. ("Fixed" here means that the performance guarantee is constant, as opposed to a PTAS where we can take an arbitrary performance guarantee.) The following result is also known:

Theorem 3. *There does not exist a PTAS for any* MaxSNP*-hard problem unless* P = NP*.*

Thus, unless we get that incredible groundbreaking research discovery, the hardest of the hard problems in this class remain out of reach by any PTAS.

3 An Approximate Tour

In this course, we will be taking a look at a variety of basic techniques for designing approximation algorithms for different problems. For starters, let's consider one specific problem and compare how we can apply different techniques to it.

The problem we will be considering here is the (weighted) *set cover* problem.

Weighted-Set-Cover
Given: a ground set of elements $E = \{e_1, \ldots, e_n\}$, subsets $S_1, \ldots, S_m \subseteq E$, and a nonnegative weight $w_j \geq 0$ for each subset S_j
Determine: the set $I \subseteq \{1, \ldots, m\}$ that minimizes $\sum_{i \in I} w_i$ such that $\bigcup_{i \in I} S_i = E$

In other words, the weighted set cover problem asks us to find the minimum-weight collection of subsets that "covers" every element of E. Note that if $w_j = 1$ for all j, then we get the *unweighted set cover* problem.

Example 4. Let $E = \{1, 2, 3, 4, 5\}$ and consider subsets

$$S_1 = \{1, 3, 4\}; S_2 = \{1, 2\}; S_3 = \{4, 5\}; S_4 = \{2, 4, 5\}; \text{ and } S_5 = \{2\}.$$

Also let the corresponding subset weights be $w_1 = 8; w_2 = 3; w_3 = 9; w_4 = 11; \text{ and } w_5 = 2$.

A weighted subset cover for this instance of the problem is the set $I = \{1, 3, 5\}$, since $S_1 \cup S_3 \cup S_5 = E$. Furthermore, this cover has total weight $w_1 + w_3 + w_5 = 19$, and it's straightforward to see how this is indeed the minimal weight for a set cover.

3.1 Prelude: Unweighted Greediness

Before we consider algorithms to find a solution to the weighted set cover problem, let's try a naïve technique—the *greedy approach*—on the unweighted set cover problem. Here, we will simply take an element from our ground set E each round and add the corresponding subset index to our set I.

Algorithm 1: Unweighted set cover—greedy

$I \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
  Choose $e_i \in E$
  $I \leftarrow I \cup \{j \mid e_i \in S_j\}$
  $E \leftarrow E \setminus \bigcup_{j \in I} S_j$

Let $f = \max_i |\{j \mid e_i \in S_j\}|$; that is, f denotes the maximum frequency of an element e_i, or the largest number of sets that contain an element e_i.

We must first prove the correctness of our algorithm.

Lemma 5. *Algorithm 1 returns an unweighted set cover.*

Proof. During each iteration of the while loop, we choose one element from the ground set E and cover it. We then remove that element from the ground set. Since an element is only removed after it is covered, and the algorithm terminates when $E = \emptyset$, we must end up with an unweighted set cover. □

We now prove the performance guarantee of our algorithm.

Theorem 6. *Algorithm 1 is an f-approximation algorithm for the unweighted set cover problem.*

Proof. Clearly, the algorithm runs in polynomial time, since it iterates through the while loop at most n times; that is, at most once for each element of the ground set E.

Suppose the algorithm iterates through the while loop X times. Each element e_i chosen in the while loop must be covered by a distinct set in the optimal solution, OPT. For, if not, then there would exist an element e_a chosen in one iteration and an element e_b chosen in another iteration that both belong to the same subset S_j in OPT. However, by our construction, when we choose the element e_a, we also choose all sets that contain e_a, including S_j, and we remove all elements from E that belong to these sets. Thus, e_b would be removed from E and would not be chosen in a later iteration, leading to a contradiction. This implies that $X \leq \text{OPT}$.

Each time we choose an element e_i, we add $|\{j \mid e_i \in S_j\}| \leq f$ subset indices to I, which means that $|I| \leq f \cdot X \leq f \cdot \text{OPT}$. Therefore, the algorithm is an f-approximation algorithm. □

3.2 Interlude: Integer and Linear Programming

Many of the techniques we will employ in this course make use of *integer programs* and *linear programs*, and the examples in this section are no different. Thus, before we continue, let's review what each of these notions mean and formulate our weighted set cover problem in these terms.

Both integer programs (IP) and linear programs (LP) are formulations of an optimization problem in terms of *decision variables* that represent the decisions being made to solve the problem. Decision variables are *constrained*, generally by linear inequalities. An assignment of values to each decision variable that satisfies all of the constraints is called a *feasible solution*.

As an example, let's formulate the weighted set cover problem as an integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \qquad \text{for all } i \in \{1, \ldots, n\} \\ & x_j \in \{0,1\}, \quad \text{for all } j \in \{1, \ldots, m\} \end{aligned} \tag{WSC-IP}$$

The first line is our *objective function*: this often represents the value that we're trying to minimize or maximize using our algorithm. In the case of the integer program WSC-IP, given decision variables x_j and weights w_j, we're trying to minimize the weight of the decision variables that we select. This corresponds to minimizing the weight of the subsets we choose for our set cover. Note that the feasible solution that minimizes or maximizes the objective function is our *optimal solution* for the problem.

The second and third lines specify the constraints on our objective function. The second line specifies that, for each element, we must choose at least one subset that covers the element. The third line further constrains us to choose each subset either zero or one times. Thus, we can only add a given subset to our set cover at most once.

Since the integer program WSC-IP exactly models the weighted set cover problem, the optimum feasible solution to WSC-IP, denoted z^*_{IP}, will also be the optimal solution for the problem. Thus, $z^*_{\text{IP}} = \text{OPT}$. However, obtaining the optimal solution isn't as straightforward as you might think. Unfortunately, integer programs generally can't be solved in polynomial time. Indeed, integer programming is NP-hard (in fact, NP-complete).

So, what can we do to get around this? The main issue that makes integer programming NP-hard is that taking integer values for each decision variable allows us to encode instances of the Boolean satisfiability problem using integer programs. Instead, we can choose to *relax* the integer-valued condition to allow decision variables to take on real values, and in doing so we convert our integer program to a linear program:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j:e_i \in S_j} x_j \geq 1, \quad \text{for all } i \in \{1, \ldots, n\} \\ & x_j \geq 0, \quad \text{for all } j \in \{1, \ldots, m\} \end{aligned} \tag{WSC-LP}$$

Note that the only change we made to our program is in the third line, where we now have $x_j \geq 0$ instead of $x_j \in \{0,1\}$. Such a small change has a profound impact: we can now solve this linear program in polynomial time! Moreover, every feasible solution for the original integer program is also a feasible solution for the corresponding linear program. Indeed, if z^*_{LP} is the optimal solution for the linear program WSC-LP, then by the previous observation, we have that $z^*_{\text{LP}} \leq z^*_{\text{IP}} = \text{OPT}$.

Now that we have all of the necessary machinery, if we can find an integral feasible solution within a certain factor α of z^*_{LP}, then we can conclude that the solution will be no more than $\alpha \cdot \text{OPT}$ and we get an α-approximation algorithm.

To find such a solution, we just follow the steps we outlined earlier:

1. Formulate the problem as an integer program.
2. Relax the integer program to a linear program.
3. Use a solution to the linear program to obtain, in polynomial time, a solution to the integer program that is close to optimal.

The third step is where our different techniques get to shine.

3.3 Approach 1: Rounding

If we have a feasible solution to a linear program, one way we can convert it to an integral feasible solution is by *rounding*. Simply put, if z^*_{LP} is the optimal solution for the linear program, then we round each decision variable up to 1 if it surpasses some threshold, or otherwise we round it down to 0.

Let's recall our value $f = \max_i |\{j \mid e_i \in S_j\}|$ from earlier. We will use this value f in the following algorithm to determine the threshold for rounding a fractional solution x^*, which is the optimal linear program solution, to an integral solution $\hat{x}$.

Algorithm 2: Weighted set cover—rounding

```
Solve the LP to get optimal solution x*
I ← ∅
for each subset S_j do
    if x*_j ≥ 1/f then
        I ← I ∪ {j}
```

As before, we must first prove the correctness of the algorithm.

Lemma 7. *Algorithm 2 produces a weighted set cover.*

Proof. Suppose by way of contradiction that there exists some element e_i such that $e_i \notin \bigcup_{j \in I} S_j$; that is, e_i is not covered. Then, for each subset S_j to which e_i belongs, we must have that $x^*_j < 1/f$ in the solution. As a result,

$$\begin{aligned}\sum_{j : e_i \in S_j} x^*_j &< \frac{1}{f} \cdot |\{j \mid e_i \in S_j\}| \\ &\leq 1,\end{aligned}$$

since $|\{j \mid e_i \in S_j\}| \leq f$. However, this violates the first constraint of WSC-LP. Therefore, every element e_i must be covered. □

Next, we establish that this rounding technique indeed produces an approximation algorithm.

Theorem 8. *Algorithm 2 is an f-approximation algorithm for the weighted set cover problem.*

Proof. Clearly, the algorithm runs in polynomial time.

Furthermore, we have that

$$\begin{aligned}\sum_{j \in I} w_j &\leq \sum_{j=1}^{m} w_j x_j^* f \quad \text{(since } x_j^* f \geq 1 \text{ for each } j \in I\text{)} \\ &= f \sum_{j=1}^{m} w_j x_j^* \\ &= f \cdot z_{\text{LP}}^* \\ &\leq f \cdot \text{OPT} \quad \text{(since } z_{\text{LP}}^* \leq \text{OPT}\text{).}\end{aligned}$$

□

3.4 Approach 2: Dual Rounding

For our next approach, we will again make use of rounding, but this time around with a slight twist. Before we continue, we will need to define one notion.

If we have a linear program L, we can construct another linear program derived from L by making three changes. This new linear program, which is closely related to L, is called the *dual* of L.

Definition 9 (Dual of linear program)**.** Given a linear program L, its dual is a linear program where:

- Every decision variable in L becomes a constraint of the dual;
- Every constraint of L becomes a decision variable in the dual; and
- The objective function of L is reversed in the dual (i.e., maximization becomes minimization and vice versa).

Since we gave a fancy name to the dual, we will refer to the original linear program as the *primal*.

Recalling our linear program for weighted set cover,

$$\begin{aligned}\text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & \sum_{j : e_i \in S_j} x_j \geq 1, \quad \text{for all } i \in \{1, \ldots, n\} \\ & x_j \geq 0, \quad \text{for all } j \in \{1, \ldots, m\}\end{aligned} \tag{WSC-LP}$$

we can get the dual of WSC-LP by making the aforementioned three changes (and changing the names of variables to y instead of x). This gives us the following linear program:

$$\begin{aligned}\text{maximize} \quad & \sum_{i=1}^{n} y_i \\ \text{subject to} \quad & \sum_{i : e_i \in S_j} y_i \leq w_j, \quad \text{for all } j \in \{1, \ldots, m\} \\ & y_i \geq 0, \quad \text{for all } i \in \{1, \ldots, n\}\end{aligned} \tag{WSC-DualLP}$$

The dual linear program has a decision variable y_i for each constraint $\sum_{j : e_i \in S_j} x_j \geq 1$ of the primal linear program. Likewise, the dual linear program has a constraint $\sum_{i : e_i \in S_j} y_i \leq w_j$ for each decision variable x_j of the primal linear program.

If the weighted set cover problem asks us to find a set of subsets of minimum weight that covers each element of a given set, then how do we reason about this seemingly abstract dual problem? Consider each element e_i in the ground set E, and suppose that e_i is associated with some cost $y_i \geq 0$ that we must pay to cover that element. Every subset then has a cost, which is the sum of costs of each element in that subset. If an element can be covered by a low-weight subset in the original problem, then we will associate a low cost to

that element; likewise, elements covered by high-weight subsets will receive high costs. However, we must ensure that the sum of costs of elements in a subset does not exceed that subset's weight. The dual linear program WSC-DualLP therefore maximizes the total price that can be charged for all of the elements.

One of the reasons why we care about dual linear programs in the context of approximation algorithms is because of the following fact. If we have a feasible solution x to the linear program WSC-LP, and we have a feasible solution y to the dual linear program WSC-DualLP, then the value of the dual solution y is

$$\sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} y_i \sum_{j:e_i \in S_j} x_j,$$

since $\sum_{j:e_i \in S_j} x_j \geq 1$ for all e_i. Using the dual, we can then rewrite the right-hand side of the inequality to get

$$\sum_{i=1}^{n} y_i \leq \sum_{j=1}^{m} x_j \sum_{i:e_i \in S_j} y_i.$$

Finally, since y is a feasible solution to the dual linear program, we can establish an upper bound on each y_i on the right-hand side (since $\sum_{i:e_i \in S_j} y_i \leq w_j$):

$$\sum_{i=1}^{n} y_i \leq \sum_{j=1}^{m} x_j w_j.$$

Now, we know what the right-hand side is: it's the feasible solution to the primal linear program! Therefore, we have established that any feasible solution to the dual cannot have a value greater than any feasible solution to the primal. In other terms,

$$\sum_{i=1}^{n} y_i \leq z^*_{\text{LP}} \leq \text{OPT}.$$

This is called the *weak duality property* of linear programs.

Let us now get back to the topic of approximation algorithms, and actually present an algorithm that uses the dual linear program. Here, our algorithm will solve the dual linear program, and then choose all subsets for which the price of all elements in the subset is equal to the weight of the subset.

Algorithm 3: Weighted set cover—dual rounding

```
Solve the dual LP to get optimal solution y*
I ← ∅
for each subset S_j do
    if Σ_{i:e_i∈S_j} y*_i = w_j then
        I ← I ∪ {j}
```

Once again, we prove the correctness of this algorithm.

Lemma 10. *Algorithm 3 produces a weighted set cover.*

Proof. Suppose by way of contradiction that there exists some element e_i such that $e_i \notin \bigcup_{j \in I} S_j$; that is, e_i is not covered. Then, for each subset S_j to which e_i belongs, we have

$$\sum_{i:e_i \in S_j} y^*_i < w_j.$$

As a result, we can increase y^* by some positive amount and still produce a feasible solution. However, this contradicts the optimality of y^*. Therefore, every element e_i must be covered. □

We can now consider the approximation aspect of the algorithm. For this, we once again need to recall our value f. The idea behind this proof is that, if subset S_j is chosen for the set cover, then we pay y_i^* for each of the subset's elements e_i. Since we pay for each element at most once for each subset that contains it, and since any element e_i can be in at most f subsets, our total cost will be at most f times the dual's optimal solution.

Theorem 11. *Algorithm 3 is an f-approximation algorithm for the weighted set cover problem.*

Proof. Since a subset S_j is only chosen if $\sum_{i:e_i \in S_j} y_i^* = w_j$, we have that

$$\begin{aligned}
\sum_{j \in I} w_j &= \sum_{j \in I} \sum_{i:e_i \in S_j} y_i^* \\
&= \sum_{i=1}^{n} y_i^* \cdot |\{j \in I \mid e_i \in S_j\}| \\
&\leq f \sum_{i=1}^{n} y_i^* \\
&\leq f \cdot \text{OPT}.
\end{aligned}$$

□

3.5 Approach 3: Primal-Dual

You might have noticed that, in both of our previous approaches, the very first step in each of our algorithms had us solving a linear program. While it is true that we can perform this step in polynomial time, it is very much a bottleneck in our algorithms. We might like to have a faster algorithm that bypasses this bottleneck, and we can do so while still using the notions of primal and dual linear programs.

The *primal-dual method* originates from the field of combinatorial optimization. In the context of approximation algorithms, the primal-dual method begins with a dual feasible solution, and then uses that to obtain a primal solution that may or may not be feasible. In the case where the primal solution is not feasible, the dual solution is modified.

Although Algorithm 3 also used a feasible solution to the dual linear program, it didn't rely on that solution being optimal. Since we don't care about optimality, we didn't really need to solve the dual linear program; we just needed *some* feasible solution.

For our present approach, we will use an idea in the proof of Lemma 10 to write an algorithm that constructs its own dual solution without the need to solve a linear program. Our resulting algorithm will work as follows:

- Given a feasible dual solution $\overline{y}$, let $T = \{j \mid \sum_{i:e_i \in S_j} \overline{y}_i = w_j\}$. In other words, T is the set of subset indices where the price of all elements in the subset is equal to the weight of the subset.
- If T is a set cover, then the algorithm returns T.
- If T is not a set cover, then there exists some element e_i that is not covered, and we can use the proof of Lemma 10 to improve the dual solution while still ensuring the solution is feasible. Specifically, we increase the cost $\overline{y}_i$ until we reach the weight of the next minimal subset S_j containing e_i.

Since, in the case where T is not a set cover, we cover exactly one new element, this algorithm will loop at most n times. Let's formalize this algorithm now.

Algorithm 4: Weighted set cover—primal-dual

$I \leftarrow \emptyset$
$\overline{y} \leftarrow 0$
while there exists $e_i \notin \bigcup_{j \in I} S_j$ **do**
  increase $\overline{y}_i$ until there exists ℓ such that $e_i \in S_\ell$ with $\sum_{j:e_j \in S_\ell} \overline{y}_j = w_\ell$
  $I \leftarrow I \cup \{\ell\}$

Before we continue, let's verify that our algorithm indeed constructs what we want for the dual linear program.

Lemma 12. *Algorithm 4 constructs a dual feasible solution.*

Proof. We prove this statement by induction.

For the base case, we initially have that $\overline{y} = 0$. As a result,

$$\sum_{i:e_i \in S_j} \overline{y}_i = 0 \leq w_j,$$

which is obviously true for all j. Thus, the base case holds.

For the inductive case, assume that, after entering some iteration of the while loop, we have

$$\sum_{i:e_i \in S_j} \overline{y}_i \leq w_j$$

for all subsets S_j. Since we only change the value of the dual decision variable $\overline{y}_i$ in the while loop, the inequality remains unchanged for all subsets S_j where $e_i \notin S_j$. On the other hand, if $e_i \in S_j$, then we add some value $\epsilon_\ell = \left(w_\ell - \sum_{i:e_i \in S_\ell} \overline{y}_i\right)$ to $\overline{y}_i$, obtaining

$$\begin{aligned}
\sum_{i:e_i \in S_j} \overline{y}_i + \epsilon_\ell &= \sum_{i:e_i \in S_j} \overline{y}_i + \left(w_\ell - \sum_{i:e_i \in S_\ell} \overline{y}_i\right) \\
&\leq \sum_{i:e_i \in S_j} \overline{y}_i + \left(w_j - \sum_{i:e_i \in S_j} \overline{y}_i\right) \\
&\leq w_j.
\end{aligned}$$

Thus, the inductive case holds. □

As always, we prove two other things related to our algorithm. Our first step, proving the correctness of the algorithm, is quite easy.

Lemma 13. *Algorithm 4 produces a weighted set cover.*

Proof. Clearly, the algorithm produces a weighted set cover, since that is the termination condition. □

We next consider the performance guarantee.

Theorem 14. *Algorithm 4 is an f-approximation algorithm for the weighted set cover problem.*

Proof. From the specification of the algorithm, we know that if $j \in I$, then $\sum_{i:e_i \in S_j} \overline{y}_i = w_j$, since in the iteration of the while loop where we add the subset index j to I, we simultaneously increase $\overline{y}_i$ by exactly enough to make the constraint tight.

As a result, we have that

$$\begin{aligned}\sum_{j\in I} w_j &= \sum_{j\in I}\sum_{i:e_i\in S_j} \overline{y}_i \\ &\leq \sum_{i=1}^{n} \overline{y}_i \cdot |\{j \in I \mid e_i \in S_j\}| \\ &\leq f \cdot \sum_{i=1}^{n} \overline{y}_i \\ &\leq f \cdot \mathrm{OPT}.\end{aligned}$$

□

3.6 Approach 4: Weighted Greediness

Now, after all of the approaches we've seen so far, you might have noticed that we keep getting the same performance guarantee: an f-approximation algorithm. In the very beginning, though, we used an approach on the unweighted set cover problem, but not the weighted version. Let's come full circle now and bring that approach back to use on the weighted set cover problem.

The greedy approach generally makes decisions in the following way: at any given step of the computation, a greedy algorithm makes the decision that is best *at that moment*. In other words, the decision the algorithm makes is *locally optimal*. Greedy algorithms are very easy to implement for this reason; we simply need to take the biggest, or the greatest, or the best value at any given step. A greedy algorithm, however, is incapable of seeing whether the series of locally optimal decisions it's making is leading to a *globally optimal* solution, That's the main drawback of the greedy approach: sometimes, greed is not good.

When we used the greedy approach on the unweighted set cover problem, we simply took *some* element at a given step and added it to our set cover. Since all elements had equal weight, we didn't care which element we took or when, so long as we eventually obtained a set cover. For the weighted version of the problem, though, our greedy algorithm must take into account the weights of each subset. Since we want to minimize the total weight of the subsets we take, we should strive to find a balance or tradeoff between a subset's weight and the number of elements that subset contributes to our set cover. Thus, our algorithm will choose the subset that minimizes the ratio of its weight to the number of uncovered elements it contains. (In the event of a tie between subsets, we just pick one of the minimal-ratio subsets arbitrarily.)

Algorithm 5: Weighted set cover—greedy

$I \leftarrow \emptyset$
$\overline{S}_j \leftarrow S_j$ for all j
while $\bigcup_{j\in I} S_j \neq E$ **do**
  $\ell \leftarrow \arg\min_{j:\overline{S}_j\neq\emptyset} w_j/|\overline{S}_j|$ ▷ arg min chooses the minimal-ratio subset
  $I \leftarrow I \cup \{\ell\}$
  $\overline{S}_j \leftarrow \overline{S}_j \setminus S_\ell$ for all j

Before we continue, let's introduce one important sequence that we will need. The nth *harmonic number*, denoted H_n, is the number produced by summing the reciprocals of the natural numbers starting from 1 up to n; that is,

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}.$$

It is known that $H_n \approx \ln(n)$, but we won't get into explaining why that is the case here[3]. We will simply take it as a fact for now.

[3]If you must know, we can approximate the sum by the integral $\int_1^n \frac{1}{x}\,\mathrm{d}x$.

Getting back to the algorithm, let's prove our two statements.

Lemma 15. *Algorithm 5 produces a weighted set cover.*

Proof. Clearly, the algorithm produces a weighted set cover, since that is the termination condition. □

The performance guarantee is where we require our notion of a harmonic number.

Theorem 16. *Algorithm 5 is an H_n-approximation algorithm for the weighted set cover problem.*

Proof. Let n_k denote the number of uncovered elements at the start of the kth iteration of the algorithm. If the algorithm iterates ℓ times, then $n_1 = n$ and $n_{\ell+1} = 0$.

Suppose the algorithm is on its kth iteration. We claim that, for the subset S_j chosen in this iteration,

$$w_j \leq \frac{n_k - n_{k+1}}{n_k} \cdot \text{OPT}.$$

(We will not prove this claim here.)

We can use our claim to prove the desired statement:

$$\begin{aligned}
\sum_{j \in I} w_j &\leq \sum_{k=1}^{\ell} \frac{n_k - n_{k+1}}{n_k} \cdot \text{OPT} \\
&\leq \sum_{k=1}^{\ell} \left(\frac{1}{n_k} + \frac{1}{n_k - 1} + \cdots + \frac{1}{n_{k+1} + 1} \right) \cdot \text{OPT} \quad (\text{since } \tfrac{1}{n_k} \leq \tfrac{1}{n_k - i} \text{ for all } 0 \leq i < n_k) \\
&= \sum_{i=1}^{n} \frac{1}{i} \cdot \text{OPT} \\
&= H_n \cdot \text{OPT}.
\end{aligned}$$

□

Finally, after trying so many different approaches, we get a performance guarantee for the weighted set cover problem that isn't f! But now, this raises the question of whether we can do better than H_n with our performance guarantee. Unfortunately, the current state-of-the-art says that no better approximation algorithm exists—unless, of course, we get that incredible groundbreaking research discovery.

Theorem 17. *There exists no $c \ln(n)$-approximation algorithm for the weighted set cover problem with $c < 1$, unless $\mathsf{P} = \mathsf{NP}$.*

Nevertheless, we will not let this fact deter us as we embark on our study of approximation algorithms. There are many more problems out there, and we now have the toolkit we need to tackle each of these problems.