Approximation Algorithms

Introduction: Motivation for using approximation algorithms

"The greatest shortcoming of the human race is our inability to understand the exponential function"-Al Bartlett

One of the greatest challenges in all of computer science is to find a polynomial time algorithm for the NP-complete problem. For most practical inputs, algorithms that involve exponential growth *cannot* be solved in a reasonable time frame. Consider the famous traveling salesman problem. Using the Held-Karp algorithm, the TSP can be solved in $O(n^2 * 2^n)$ [8] Even if you are given the world's most powerful supercomputer, which can clock 10^{15} calculations a second, it would still take 10 million years to solve the problem:

$$\approx \frac{10^{15} * 10^7 \left(\frac{sec}{year}\right)}{2^{50}\ calculations} = \mathbf{10\ million\ years}$$

Given that there are more than three thousand known NP complete problems [7], it is very possible that at some point you will be faced with having to deal with finding a solution to this problem.

Rather than brush aside the problem and declare it to be impossible, instead you can reach into your bag of tools and find in there an **approximation algorithm**. These algorithms do not guarantee an exact solution, but for practical applications they may provide a solution that is "good enough", and they do so in polynomial time.

An **approximation algorithm** solves the *practical problem* of the exponential time it takes to reach a solution for an NP-Complete problem.

Type of Approximation

The word "approximation" can suggest several meanings, so let's first be clear about what is meant when we call an algorithm an approximation algorithm in this lecture:

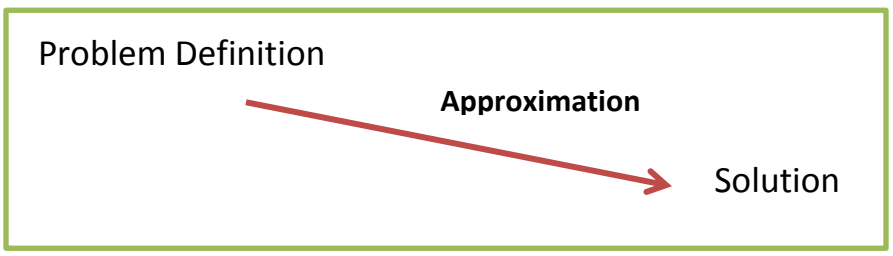
> **Definition:** Approximation algorithms approximate the set of solutions in the problem space.

To understand this point, consider a contrast to another situation where an approximation might occur. In a typical engineering or physics problem, the difficulty often lies in dealing with the uncertainty generated in mapping an overwhelmingly complex real world event onto a solution space. In order to handle the difficulty of this task, we approximate the real-world event itself. This produces uncertainty in the quantities of the actual model imported into the problem space. This is the popular, "assume a spherical cow" solution often used in physics, in which we take the complex shape of a cow and map it into a sphere to be handled more easily by the problem space.

Physics Problem:

You could imagine computer vision algorithms, for example, doing this type of approximation, and in a sense this would make it an approximation algorithm. But the approximation algorithms referred to here (which is how they are referred to generally) is not a tool to help in *this* type of approximation. Instead, an approximation algorithm begins from the problem space where the problem could be completely well-defined, but nonetheless uses an approximation to reach a solution.

Approximation Algorithm:



Remember here that the problem space is typically NP-complete, and the approximation steps taken to reach an approximate solution occur to reduce the time needed to reach that solution.

The Fundamental Characteristic of Approximation Algorithms

If we are going to approximate a solution to an algorithm, it seems natural to worry just how accurate the approximation will be. The question can be phrased in this manner:

"What's the worst result our approximation algorithm can return?"

This question brings up the fundamental – and perhaps most powerful - characteristic of approximation algorithms. *All* approximation algorithms answer this question by **guaranteeing** an upper bound on the worst result they can generate. In other words, when running an approximation algorithm, you *must* be able to say that your algorithm is guaranteed to return *at worst* a value that's within some exactly defined factor of the optimal solution.

More formally, let's say that a problem space has a range of possible solutions $s \epsilon S$ with an optimal solution C^* and the value returned by our algorithm$C$.

Optimizatino problems can come in two forms: either they are maximization problems or they are minimization problems.

The TSP is an example of a minimization problem, where:

$$0 \leq C^* \leq C$$

While a maximization problem has the property:

$$0 \leq C \leq C^*$$

Therefore, when defining an approximation algorithm, we first consider whether the problem is a maximization problem or a minimization problem. We now look for the ratio $\rho(n)$ that puts a boundary on the solutions our algorithm can return relative to the optimal solution.

Given that the problem is a *maximization* problem:

$$\rho(n) \leq \max\left(\frac{C^*}{C}\right)$$

Note that we take the maximum value of this ratio because we want to assume that the algorithm may return the worst (in this case, lowest) possible value of C.

If the problem is a minimization problem, then we of course must flip this ratio upside down:

$$\rho(n) \leq \max\left(\frac{C}{C^*}\right)$$

Take a moment to consider the importance of this ratio. We can run our algorithm with a well-known degree of assurance. If you are asked to implement an algorithm for some project, and you need reliable outputs, you will know just how reliable your approximation algorithm is.

This also more clearly defines what is meant when we speak of an approximation algorithm. In contrast to the simulated annealing algorithm we implemented in problem set 4, where we had virtually no clue what kind of solution our algorithm would output, true approximation algorithms are defined as those whose outputs are bounded by $\rho(n)$.

Fundamental Characteristic: The range of possible outputs for an approximation algorithm are bounded by a ratio $\rho(n)$, and we say that such an algorithm is a

$\rho(n)$-approximation-algorithm

Therefore, when we say we have a 5-approximation algorithm, it means that the worst possible value our algorithm can return is within a factor of 5 of the optimal solution. An even worse algorithm would be a 10-approximation algorithm, which returns at worst a solution within a factor of 10 of the optimal cost. The best an approximation algorithm could be is a 1-approximation algorithm, which would mean the algorithm would be guaranteed to return the optimal solution. Of course, you would never expect to see such an approximation algorithm that is guaranteed to return a 1-approximatino algorithm, and algorithms that are close to 1 such as a $(1 + \alpha)$ approximation where $alpha < 1$ are usually still NP-hard [9]

So, $\rho(n)$ provides a boundary on the worst possible value our approximation algorithm will return. Can we possibly get more information about the quality of this boundary? For some algorithms, the answer is yes. Some approximation algorithms can actually return better results the longer you let them run. Since this type of algorithm requires some kind of user defined scheme for how long he/she wants to run the algorithm, we call this type of algorithm an **approximation scheme** algorithm [2]. We will use the value ϵ $(where\ \epsilon > 0)$ as a parameter to help describe this approximation scheme.

Approximation Scheme: An approximation scheme algorithm is an approximation algorithm which returns improved results with time. We say that such an algorithm is a

$(1 + \epsilon)$-approximation algorithm

Since ϵ controls how long we run our approximation algorithm for, it makes sense to now include it in our run time analysis:

$$O(n) \rightarrow O(n, \epsilon)$$

Note that a *decrease* in ϵ makes for a better approximation as well as a longer runtime. Hence you would not expect to see $O(n * \epsilon)$, since this implies that a decrease in ϵ – which improves the approximation algorithm – also decreases the run time. Instead you might expect to see:

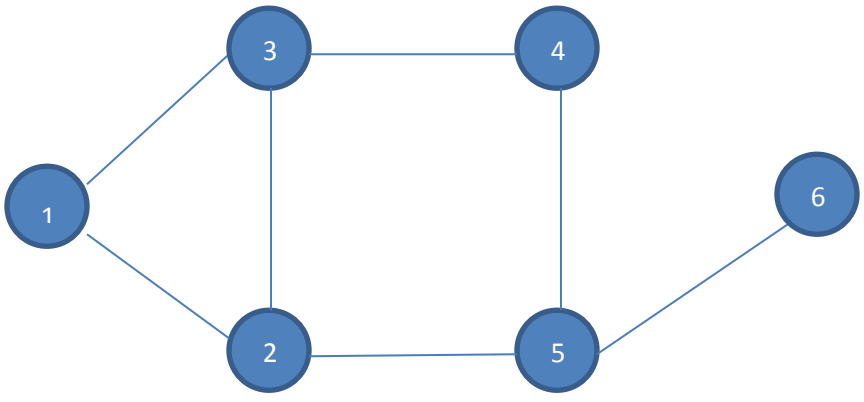
$$\underline{\text{Ex}}: \quad O\left(n^{\frac{2}{\epsilon}}\right) \quad \text{or} \quad O\left(\left(\frac{1}{\epsilon}\right)^2 * n\right)$$

For the run-time example on the left, note that the run-time increases exponentially as we decrease ϵ, so there is a rapidly increasing price to be paid in terms of time to yield better results (this is still a polynomial time algorithm though for a specific instance of ϵ).

Also note that because ρ is a function of n, sometimes the quality of the approximation algorithm is dependant on the number of inputs. This usually implies a decreased quality of approximation with more inputs, as in the set-cover problem discussed later in detail.

Vertex-Cover Algorithm

The first example of an approximation algorithm is the vertex-cover algorithm. Say we have the given graph $G(V, E)$

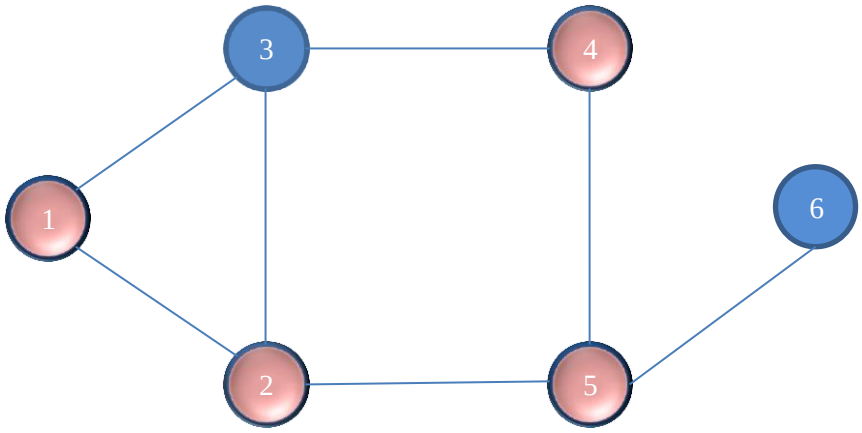


A vertex-cover is a subset of vertices in the graph such that if you were to pick any edge in the graph, at least one of the edges would be contained in that subset.

More formally:

- **Vertex-Cover:** Given a graph $G(V, E)$, a subset of the vertices $C \subseteq V$ is a vertex cover iff $\forall$ edge $(u, v) \epsilon\ E$ then at least $u \epsilon\ C\ or\ v \epsilon\ C$ (or both). The size of this vertex-cover is the number of elements in the set C

A possible vertex cover for our example graph would be:



Vertex-Cover

$C = \{1,2,4,5\}$

We can quickly verify that for every edge of the graph, at least one of the nodes that the edge touches is within our constructed subset C.

However, though this is subset is a vertex-cover, it is not what we would call a *minimum* vertex-cover of the graph. A minimum vertex-cover represents the smallest number of vertices needed to make a vertex-cover for a given graph. For this example, note that such a minimum vertex-cover might be $C = \{1,3,5\}$, which has a size of three.

The vertex-cover problem is to find a minimum vertex-cover. This problem is a well-known NP-complete problem. To tackle this, we will use the following approximation algorithm.

Input: A graph $G(V, E)$ given as an adjacency list

Output: A subset of vertices $C \subseteq V$ that is a vertex-cover for the graph

---

Vertex-Cover-Approx (G(V,E))

1. $C\{\} = \emptyset$	//$O(1)$
2. $E'\{\} = G.E$	//$O(E)$
3. while $E'.size\ != 0$	
4. $(u, v) = E'.random()$	//$O(1)$
5. $C.add(u)\ ;\ C.add(v)$	//$O(1)$
7. $E'.remove(u.edges\_covered)$	//$O(V + E)$
8. $E'.remove(v.covered\_edges)$	//$O(V + E)$
8. **Return** C	//$O(1)$

The idea behind this algorithm is simple. We begin with an empty set C which will eventually contain our vertex cover. We then make another set E' which contains all the edges of the graph that have not yet been covered. Since at the beginning of the algorithm no edges have been covered, E' must be initialized to contain all edges of the graph. We then pick a random edge from this set. We add both of the vertices connected to this edge to our vertex cover set C[1] , and delete all edges covered by these vertices from our set E'. Since the input graph is in adjacency list form, removing all covered edges simply requires going to that vertex and traversing the list, each time deleting that edge from the list E'. We then repeat this process for all edges of the graph

The run time for this algorithm can easily be verified as polynomial. The run times to remove the covered edges have a run time of $O(V + E)$, and for each edge that is removed there is one less execution for the while loop. The run time of the entire algorithm is therefore $O(V + E)$.

1

*Note: The first time you look at this algorithm, it may not be entirely obvious why we add *both* edges to our cover list C each time we examine an uncovered edge. Indeed, other algorithms exist that taken different approaches. However, these algorithms all have bigger $\rho(n)$ then the algorithm described here, and so are not discussed in detail. Just as an idea of a different approach, though, one idea is to pick a vertex of maximum degree, remove all edges that it covers, and then continue to loop through this process until all edges with degree >0 have been removed.

The proof that this algorithm works is trivial. By the definition of the vertex cover, every edge within our graph must contain at least one of the vertices that edge connects, or in other words, our vertex cover list must contain vertices which cover all edges on the graph. The list E' represents all edges that have not been covered, and this holds as a loop invariant because each time we cycle through the loop and pick one of the edges from E', we delete all edges covered by the two vertices. The loop is run until E' is empty, thus it runs until every edge is covered.

Proving the algorithm works is the first step. But since this is an approximation algorithm, we must now be able to place a boundary on the worst possible result our value can return. In other words, we now need to find $\rho(n)$.

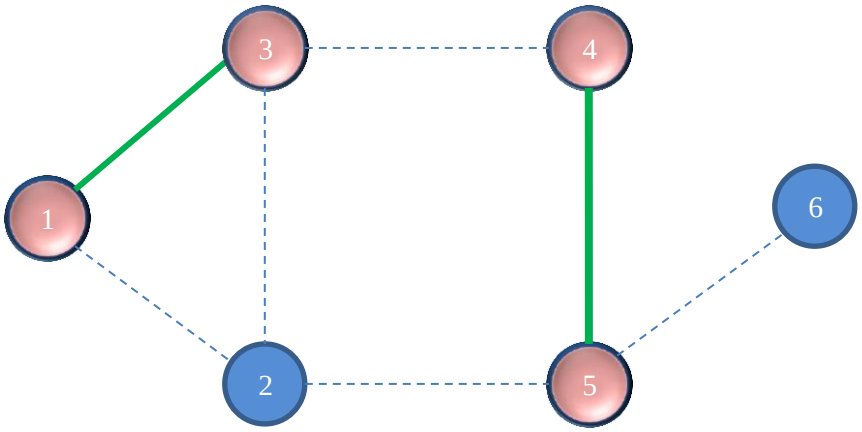
Recall that since we are trying to find the minimum vertex-cover, this is a minimization problem and therefore:

$$\rho(n) \leq \max\left(\frac{C}{C^*}\right)$$

On the surface this seems to be an impossible task. If we don't even know what the optimal solution is, how can we figure out $\rho(n)$?

Remarkably, we do not need to know the optimal solution to figure this out, and the example of the vertex-cover problem was chosen exactly because it provides a clear and simple proof that elucidates this concept.

To see how this works, let's run through the algorithm once step by step on the example graph:



The algorithm picked out the green edges. Let's put these edges into an array P

$$P = \{(1,3), (4,5)\}$$

When the algorithm picked each of these edges, it added both vertices for each edge to the vertex cover-list C and deleted all edges those vertices covered.

$$C = \{1,3,4,5\}$$

By the definition of a vertex cover, every edge in the graph needs to have at least one of its vertices in the vertex-cover-set. This means that for each edge in P, specifically, the optimal set needs to have at least one vertex from each of P's edges. Notice that, because we delete all edges covered by the vertices for each green edge chosen, no edges in P ever have the same endpoint, and therefore none of these edges are ever covered by the same vertex from C^*. Therefore, the cardinality of C^* is always at least as big as P.

$$|C^*| \geq |P|$$

It is easy to relate C to P. For every edge that is added to P, we add that edges two vertices to C. Thus:

$$|C| = 2|A|$$

Combining these two equations:

$$|C| \leq 2|C^*|$$

$$\frac{|C|}{|C^*|} \leq 2$$

Since $\rho(n) \leq \max\left(\frac{|C|}{|C^*|}\right)$ we can say that $\rho(n)$ is at most 2, and therefore this algorithm is a

2-approximation algorithm. A good question to ask then would be if there exists an algorithm with an even tighter bounding ratio. Proving that they're might exist one is an ongoing debate in computer science [6];

Set Cover Problem

The set cover problem falls under the category of NP-hard problems. It was shown to be NP-complete by Richard Karp in 1972 when he released a paper describing 21 NP-complete problems. Just what is the set cover problem? Let's say we are given the following family F of subsets S and a finite number of elements X. Each subset in S contains one or more elements from X and every element in X belongs to at least one subset S in F. As depicted in Figure 1 below, subsets may overlap and elements from X may belong to multiple subsets. Our goal is to find the minimum number of subsets covering all elements in X. We say an element is covered when it is contained by a subset. Figure 1 below depicts an example of the set-cover problem. S_1 and S_3 make a set cover C that entirely covers the elements in X. Subset S_2 is not needed in the set cover.

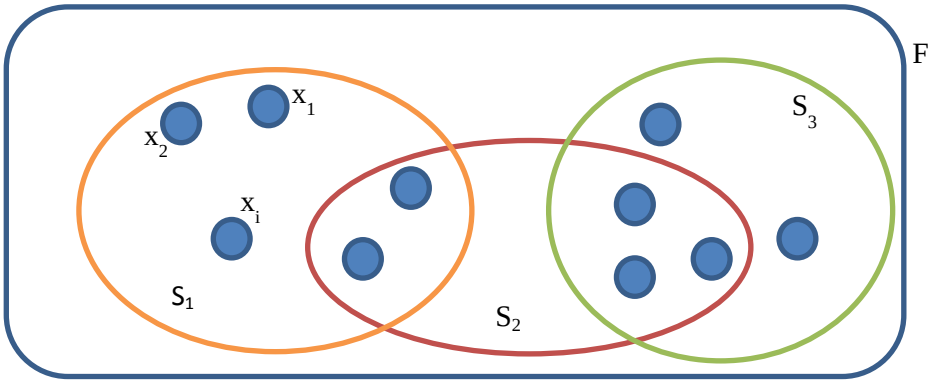


Figure 1: Example Family of Subsets Containing Elements

Here is the set cover problem stated more formally. Given:

$$X = \bigcup_{S \in F} S$$

Find a minimum size subset cover $C \subseteq F$ whose members cover all of X:

$$X = \bigcup_{S \in C} S$$

Practical Application

Why should we concern ourselves with the set cover problem? There are multiple real-world applications for set cover. First, IBM used set cover for virus detection. They had 5000 known viruses with 9000 substrings of 20 or more bytes not found in "good" code. A set cover of size about 180 was found that covered all 9000 strings. When scanning for viruses, it was sufficient to compare code to 180 substrings instead of all 9000. Scanning for 180 substrings is faster than scanning for 9000 substrings, and 9000 substrings may "over-fit" the data and produce more false positives. Set cover can also be used to help determine facility location. Given a set of potential sites and points to be serviced by the facility, an optimal location can be determined. Third, set cover can be used to minimize cost from suppliers. For example, supplier A may provide 1 pallet of lumber and 20 boxes of nails for $X, supplier B may provide 2 pallets of lumber and 4 boxes of nails for $Y, etc.

Greedy Approximation

Since we do not have a polynomial time algorithm to solve set cover exactly, we use an approximation algorithm to achieve a reasonable result. The simple set cover approximation algorithm uses a greedy approach.

Greedy-Set-Cover(X, F)

1. $C = \emptyset$
2. $Y = X$
3. while $Y \neq \emptyset$
4. Select $S \in F$ that maximizes $|S \cap Y|$ (or minimizes α)
5. $C = C \cup \{S\}$
6. Y = Y – S
7. return C

X represents the elements
Y represents the uncovered elements
F represents the family of subsets
C is the set cover
S is a subset in F

Example 1 – Greedy Set Cover

The following group of sets shown in Figure 2 is a typical example where the greedy algorithm achieves an approximation ratio of $\frac{\log_2 n}{2}$. The greedy algorithm always selects the subset with

the most uncovered elements. In this example, S_3 contains eight uncovered elements, while S_4 and S_5 each contain 7 uncovered elements. After S_3 is selected, S_2 now has four uncovered elements, while S_4 and S_5 each have only three uncovered elements. In the last step, S_1 has two uncovered elements while S_4 and S_5 each have just one uncovered element. So our greedy set covering algorithm selects S_3, then S_2, then S_1 and $C = S_1 \cup S_2 \cup S_3$ and has a size of three. The optimal set cover is $C = S_4 \cup S_5$, which has a size of two.

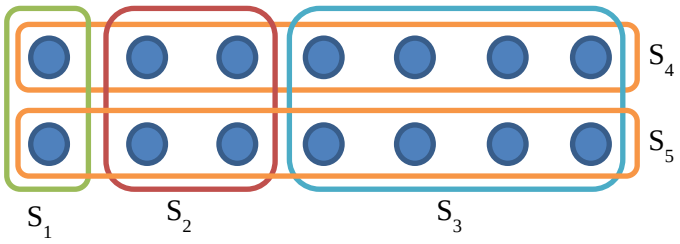


Figure 2: Group of Subsets without Costs

Subsets with Costs

Subsets may also be given a cost, denoted as $cost(S_i)$. Costs are arbitrarily assigned to subsets in the examples in these notes; however, in practical application, costs may be dollars, distance, time, etc. When subsets have a cost, our goal is to find the minimum cost for covering all elements in X. Stated more formally, find subset cover $C \subseteq F$ that minimizes:

$$\sum_{x \in X} \alpha_x$$

α_x denotes the cost of an element the first time it is covered by a subset. We divide the cost of a subset by the number of uncovered elements in the subset. We exclude elements already covered by another subset. If the subsets do not have a cost, then the cost of each subset is set to one so we can still compute a per element cost. The calculation for α_x is written as the following equation:

$$\alpha_x = \frac{cost(S_i)}{\#\ Uncovered\ Elements\ in\ S_i}$$

The per element cost can also be written as the following equation, where S_i is the set currently being added to our cover C:

$$\alpha_x = \frac{cost(S_i)}{|S_i - C|}$$

When we break apart the subset cover C into its defining subsets, α_x is calculated as follows, where S_i is the set currently being added to our cover C:

$$\alpha_x = \frac{cost(S_i)}{|S_i - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$$

Example 2 – Subsets with Costs

The next group of sets in Figure 3 shows subsets association with a cost. The given costs are per set, not per element in the set. Costs per element are calculated as part of the greedy algorithm by the following equation, which was shown in a previous section above.

$$\alpha_x = \frac{cost(S_i)}{|S_i - C|}$$

The set cover starts as the empty set ($C = \emptyset$), so the initial per element cost is the cost of the set divided by the number of elements in the set. As subsets are added to C, the number of uncovered elements in a subset may change, which changes the value of α_x.

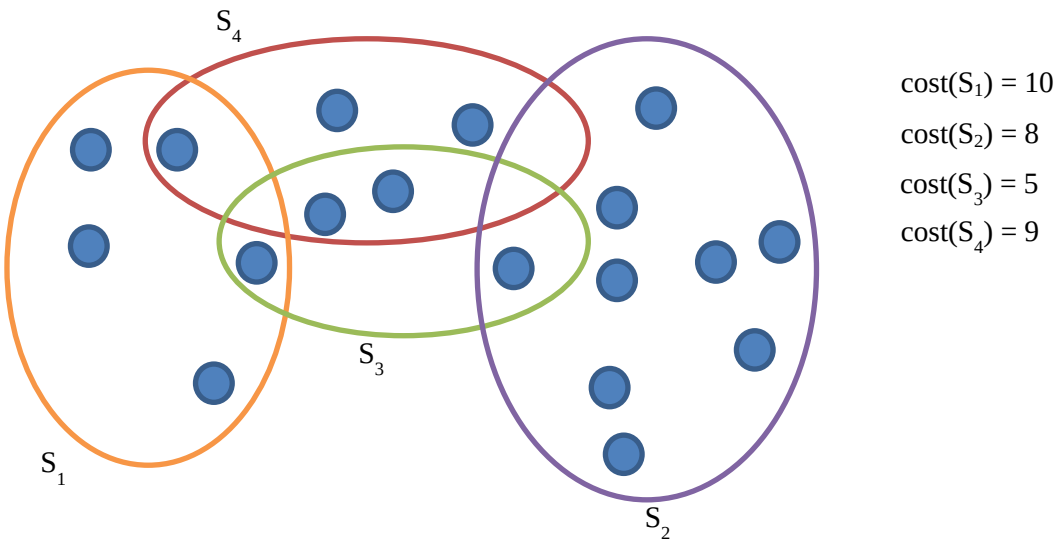


Figure 3: Group of Subsets with Costs

During each iteration of the greedy algorithm, we calculate α_x for the uncovered elements in each subset S that is not part of the cover C. The following table shows the per element cost at each iteration of the algorithm. During Iteration 1, S_2 has the lowest per element cost α_x, so we add S_2 to the set cover, which is more formally written by $C = C \cup S_2$. During Iteration 2, the

cost of elements in S_3 increases because three elements are now uncovered while four were uncovered during Iteration 1. However, S_3 still contains the lowest cost per element, so S_3 is added to the set cover. In Iteration 3 we select S_1 and Iteration 4 we select S_4. Now our set cover contains all elements in X and has a value of 32. Note the greedy algorithm did not find the optimal solution of 27.

	Iteration 1 α_x	**Iteration 2** α_x	**Iteration 3** α_x	**Iteration 4** α_x
S_1	$\frac{10}{5} = 2.00$	$\frac{10}{5} = 2.00$	$\frac{10}{4} = 2.50$	
S_2	$\frac{8}{9} = 0.89$			
S_3	$\frac{5}{4} = 1.25$	$\frac{5}{3} = 1.67$		
S_4	$\frac{9}{5} = 1.80$	$\frac{9}{5} = 1.80$	$\frac{9}{3} = 3.00$	$\frac{9}{2} = 4.50$

Table 1: Costs per Element at Each Iteration of the Greedy Algorithm

Greedy Set Cover Theorem

Greedy-Set-Cover is a polynomial time $\rho(n)$-approximation algorithm where:

$$\rho(n) = H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \log_2 n$$

$H_n$is the harmonic series where n is the n^{th} harmonic number in the series.
This means $\rho(n) \leq \max\left(\frac{C}{C^*}\right) \approx \log_2 n$, where $C^*$is the optimum result. The optimum result will be called OPT through the proof. In this case, n is the number of elements in X, so $n = |X|$.

Greedy Set Cover Proof

The following summation equals the cost of all the sets $S_1 \cup S_2 \cup \ldots \cup S_i$ which are part of the cover C.

$$\sum_{x \in X} \alpha_x = cost(S_1) + cost(S_2) + \cdots + cost(S_i)$$

We will show $\alpha_k \leq \frac{OPT}{n-k+1}$, where α_k is the cost of covering the k^{th}element.

The cost of covering all the elements is then:

$$\sum_{k \in X} \alpha_k \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1}$$

Let's call the sets which construct the optimal solution $O_1, O_2, \ldots, O_p$. Then,

$$OPT = cost(O_1) + cost(O_2) + \cdots + cost(O_p) = \sum_i cost(O_i)$$

Now assume the greedy algorithm has not yet covered all the elements in X. It has just covered the elements in our unfinished cover C. Then we know the uncovered elements $|X - C|$ are at most the intersection of all of the optimal sets intersected with the uncovered elements:

$$|X - C| \leq |O_1 \cap (X - C)| + |O_2 \cap (X - C)| + \cdots + |O_p \cap (X - C)|$$

In the greedy algorithm, we select a set with cost α, where:

$$\alpha \leq \frac{cost(O_i)}{|O_i \cap (X - C)|}, where\ i = 1 \ldots p$$

The greedy algorithm always chooses the subset with the smallest cost α, which will be less than or equal to the set chosen by the optimal algorithm. Now we take the equation above and multiply both sides by the denominator to get:

$$cost(O_i) \geq \alpha * |O_i \cap (X - C)|$$

Using substitution with our equation for OPT, we get:

$$OPT = \sum_i cost(O_i) \geq \alpha * \sum_i |O_i \cap (X - C)| \geq \alpha * |X - C|$$

Rearranging the equation we can see that:

$$\alpha \leq \frac{OPT}{|X - C|}$$

Therefore, the cost of the kth element is:

$$\alpha \leq \frac{OPT}{n - (k-1)} = \frac{OPT}{n-k+1}$$

Now we need to sum the cost of each element:

$$\sum_{x \in X} \alpha_x \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1}$$

Expanding the summation, we see the harmonic series:

$$OPT * \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = OPT * H_n$$

Therefore, the Greedy-Set-Cover algorithm has an approximation ratio $\rho(n) \approx \log_2 n$.

References

[1] Chawla, Shuchi. (30 Jan 2006). *Set Cover*. CS880: Approximation Algorithms. Retrieved from http://pages.cs.wisc.edu/~shuchi/courses/880-S07/scribe-notes/lecture03.pdf

[2] Cormen, Thomas H, Leiserson, Charles E, Rivest, Ronald L, and Stein, Clifford. (2009). *Introduction to Algorithms*, Third Edition, MIT Press.

[3] Set Cover Problem. (2011). In *Wikipedia*. Retrieved April 18, 2011, from http://en.wikipedia.org/wiki/Set_cover_problem

[4] Stern, Tamara. (2000). *What is the Set Cover Problem?*. Theoretical Computer Science, 1-5. Retrieved from http://math.mit.edu/~goemans/18434S06/setcover-tamara.pdf

[5] Williamson, David P. (1998). *Lecture Notes on Approximation Algorithms*. IBM Research Report. Retrieved from http://www.orie.cornell.edu/~dpw/cornell.ps

[6] Vazirani, Vijay V. *Approximation Algorithms*. Berlin: Springer, 2001. Print

[7] "Problem with a Capital P." *Homepage*. 27 Feb. 2011. Web. 19 Apr. 2011. http://www.mathscareers.org.uk/viewItem.cfm?cit_id=383148

[8] Bellman, R. (1960), "Combinatorial Processes and Dynamic Programming", in Bellman, R., Hall, M., Jr. (eds.), *Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics 10,*, American Mathematical Society, pp. 217–249.

[9] Robert D. Carr, Santosh Vempala. On the Held-Karp relaxation for the asymmetric and symmetric traveling salesman problems. Math. Program., 2004: 569~587