CS261: A Second Course in Algorithms Lecture #15: Introduction to Approximation Algorithms[*]

Tim Roughgarden[†]

February 23, 2016

1 Coping with NP-Completeness

All of CS161 and the first half of CS261 focus on problems that can be solved in polynomial time. A sad fact is that many practically important and frequently occurring problems do not seem to be polynomial-time solvable, that is, are NP-hard.[1]

As an algorithm designer, what does it mean if a problem is NP-hard? After all, a real-world problem doesn't just go away after you realize that it's NP-hard. The good news is that NP-hardness is not a death sentence — it doesn't mean that you can't do anything practically useful. But NP-hardness does throw the gauntlet to the algorithm designer, and suggests that compromises may be necessary. Generally, more effort (computational and human) will lead to better solutions to NP-hard problems. The right effort vs. solution quality trade-off depends on the context, as well as the relevant problem size. We'll discuss algorithmic techniques across the spectrum — from low-effort decent-quality approaches to high-effort high-quality approaches.

So what are some possible compromises? First, you can restrict attention to a relevant special case of an NP-hard problem. In some cases, the special case will be polynomial-time solvable. (Example: the Vertex Cover problem is NP-hard in general graphs, but on Problem Set #2 you proved that, in bipartite graphs, the problem reduces to max flow/min cut.) In other cases, the special case remains NP-hard but is still easier than the general case. (Example: the Traveling Salesman Problem in Lecture #16.) Note that this approach requires non-trivial human effort — implementing it requires understanding and articulating

[†] Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1] I will assume that you're familiar with the basics of NP-completeness from your other courses, like CS154. If you want a refresher, see the videos on the Course site.

whatever special structure your particular application has, and then figuring out how to exploit it algorithmically.

A second compromise is to spend more than a polynomial amount of time solving the problem, presumably using tons of hardware and/or restricting to relatively modest problem sizes. Hopefully, it is still possible to achieve a running time that is faster than naive brute-force search. While NP-completeness is sometimes interpreted as "there's probably nothing better than brute-force search," the real story is more nuanced. Many NP-complete problems can be solved with algorithms that, while running in exponential time, are significantly faster than brute-force search. Examples that we'll discuss later include 3SAT (with a running time of $(4/3)^n$ rather than 2^n) and the Traveling Salesman Problem (with a running time of 2^n instead of $n!$). Even for NP-hard problems where we don't know any algorithms that provably beat brute-force search in the worst case, there are almost always speed-up tricks that help a lot in practice. These tricks tend to be highly dependent on the particular application, so we won't really talk about any in CS261 (where the focus is on general techniques).

A third compromise, and the one that will occupy most of the rest of the course, is to relax correctness. For an optimization problem, this means settling for a feasible solution that is only approximately optimal. Of course one would like the approximation to be as good as possible. Algorithms that are guaranteed to run in polynomial time and also be near-optimal are called *approximation algorithms*, and they are the subject of this and the next several lectures.

2 Approximation Algorithms

In approximation algorithm design, the hard constraint is that the designed algorithm should run in polynomial time on every input. For an NP-hard problem, assuming $P \neq NP$, this necessarily implies that the algorithm will compute a suboptimal solution in some cases. The obvious goal is then to get as close to an optimal solution as possible (ideally, on every input).

There is a massive literature on approximation algorithms — a good chunk of the algorithms research community has been obsessed with them for the past 25+ years. As a result, many interesting design techniques have been developed. We'll only scratch the surface in our lectures, and will focus on the most broadly useful ideas and problems.

One take-away from our study of approximation algorithms is that the entire algorithmic toolbox that you've developed during CS161 and CS261 remains useful for the design and analysis of approximation algorithms. For example, greedy algorithms, divide and conquer, dynamic programming, and linear programming all have multiple killer applications in approximation algorithms (we'll see a few). And there are other techniques, like local search, which usually don't yield exact algorithms (even for polynomial-time solvable problems) but seem particularly well suited for designing good heuristics.

The rest of this lecture sets the stage with four relatively simple approximation algorithms for fundamental NP-hard optimization problems.

2.1 Example: Minimum-Makespan Scheduling

We've already seen a couple of examples of approximation algorithms in CS261. For example, recall the problem of minimum-makespan scheduling, which we studied in Lecture #13. There are m identical machines, and n jobs with processing times $p_1, \ldots, p_n$. The goal is to schedule all of the jobs to minimize the makespan (the maximum load, where the load of a machine is the sum of the processing times of the jobs assigned to it) — to balance the loads of the machines as evenly as possible.

In Lecture #13, we studied the online version of this problem, with jobs arriving one-by-one. But it's easy to imagine applications where you get to schedule a batch of jobs all at once. This is the offline version of the problem, with all n jobs known up front. This problem is NP-hard.[2]

Recall Graham's algorithm, which processes the jobs in the given (arbitrary) order, always scheduling the next job on the machine that currently has the lightest load. This algorithm can certainly be implemented in polynomial time, so we can reuse it as a legitimate approximation algorithm for the offline problem. (Now the fact that it processes the jobs online is just a bonus.) Because it always produces a schedule with makespan at most twice the minimum possible (as we proved in Lecture #13), it is a *2-approximation algorithm.* The factor "2" here is called the *approximation ratio* of the algorithm, and it plays the same role as the competitive ratio in online algorithms.

Can we do better? We can, by exploiting the fact that an (offline) algorithm knows all of the jobs up front. A simple thing that an offline algorithm can do that an online algorithm cannot is sort the jobs in a favorable order. Just running Graham's algorithm on the jobs in order from largest to smallest already improves the approximation ratio to $\frac{4}{3}$ (a good homework problem).

2.2 Example: Knapsack

Another example that you might have seen in CS161 (depending on who you took it from) is the Knapsack problem. We'll just give an executive summary; if you haven't seen this material before, refer to the videos posted on the course site.

An instance of the Knapsack problem is n items, each with a value and a weight. Also given is a capacity W. The goal is to identify the subset of items with the maximum total value, subject to having total weight at most W. The problem gets its name from a silly story of a burglar trying to fill up a sack with the most valuable items. But the problem comes up all the time, either directly or as a subroutine in a more complicated problem — whenever you have a shared resource with a hard capacity, you have a knapsack problem.

Students usually first encounter the Knapsack problem as a killer application of dynamic programming. For example, one such algorithm, which works as long as all item weights

[2] For the most part, we won't bother to prove any NP-hardness results in CS261. The NP-hardness proofs are all of the exact form that you studied in a course like CS154 — one just exhibits a polynomial-time reduction from a known NP-hard problem to the current problem. Many of the problems that we study were among the first batch of NP-complete problems identified by Karp in 1972.

are integers, runs in time $O(nW)$. Note that this is not a polynomial-time algorithm, since the input size (the number of keystrokes needed to type in the input) is only $O(n \log W)$. (Writing down the number W only takes $\log W$ digits.) And in fact, the knapsack problem is NP-hard, so we don't expect there to be a polynomial-time algorithm. Thus the $O(nW)$ dynamic programming solution is an example of an algorithm for an NP-hard problem that beats brute-force search (unless W is exponential in n), while still running in time exponential in the input size.

What if we want a truly polynomial-time algorithm? NP-hardness says that we'll have to settle for an approximation. A natural greedy algorithm, which processes the items in order of value divided by size ("bang-per-buck") achieves a $\frac{1}{2}$-approximation, that is, is guaranteed to output a feasible solution with total value at least 50% times the maximum possible.[3] If you're willing to work harder, then by rounding the data (basically throwing out the lower-order bits) and then using dynamic programming (on an instance with relatively small numbers), one obtains a $(1 - \epsilon)$-approximation, for a user-specified parameter $\epsilon > 0$, in time polynomial in n and $\frac{1}{\epsilon}$. (By NP-hardness, we expect the running time to blow up as ϵ gets close to 0.) This is pretty much the best-case scenario for an NP-hard problem — arbitrarily close approximation in polynomial time.

2.3 Example: Steiner Tree

Next we revisit the other problem that we studied in Lecture #13, the Steiner tree problem. Recall that the input is an undirected graph $G = (V, E)$ with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$. Recall also that there is no loss of generality in assuming that G is the complete graph and that the edge costs satisfy the triangle inequality (i.e., $c_{uw} \leq c_{uv} + c_{vw}$ for all $u, v, w \in V$); see Exercise Set #7. Finally, there is a set $R = \{t_1, \ldots, t_k\}$ of vertices called "terminals." The goal is to compute the minimum-cost subgraph that spans all of the terminals. We previously studied this problem with the terminals arriving online, but the offline version of the problem, with all terminals known up front, also makes perfect sense.

In Lecture #13 we studied the natural greedy algorithm for the online Steiner tree problem, where the next terminal is connected via a direct edge to a previously arriving terminal in the cheapest-possible way. We proved that the algorithm always computes a Steiner tree with cost at most $2 \ln k$ times the best-possible solution in hindsight. Since the algorithm is easy to implement in polynomial time, we can equally well regard it as a $2 \ln k$-approximation algorithm (with the fact that it processes terminals online just a bonus). Can we do something smarter if we know all the terminals up front?

As with job scheduling, better bounds are possible in the offline model because of the ability to sort the terminals in a favorable order. Probably the most natural order in which to process the terminals is to always process next the terminal that is the cheapest to connect to a previous terminal. If you think about it a minute, you realize that this is equivalent to running Prim's MST algorithm on the subgraph induced by the terminals. This motivates:

[3] Technically, to achieve this for every input, the algorithm takes the better of this greedy solution and the maximum-value item.

The MST heuristic for metric Steiner tree: output the minimum spanning tree of the subgraph induced by the terminals.

Since the Steiner tree problem is NP-hard and the MST can be computed in polynomial time, we expect this heuristic to produce a suboptimal solution in some cases. A concrete example is shown in Figure 1, where the MST of $\{t_1, t_2, t_3\}$ costs 4 while the optimal Steiner tree has cost 3. (Thus the cost can be decreased by spanning additional vertices; this is what makes the Steiner tree problem hard.) Using larger "wheel" graphs of the same type, it can be shown that the MST heuristic can be off by a factor arbitrarily close to 2 (Exercise Set #8). It turns out that there are no worse examples.

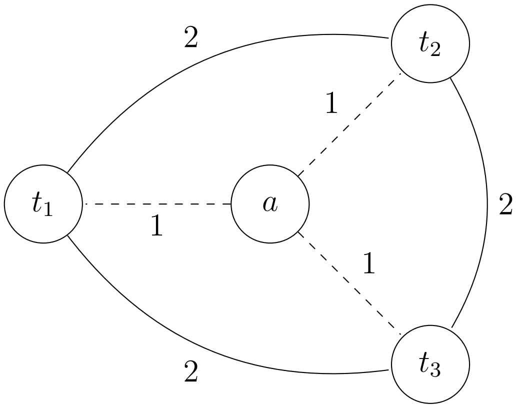

Figure 1: MST heuristic will pick $\{t_1, t_2\}, \{t_2, t_3\}$ but best Steiner tree (dashed edges) is $\{a, t_1\}, \{a, t_2\}, \{a, t_3\}$.

Theorem 2.1 *In the metric Steiner tree problem, the cost of the minimum spanning tree of the terminals is always at most twice the cost of an optimal solution.*

Proof: The proof is similar to our analysis of the online Steiner tree problem (Lecture #13), only easier. It's easier to relate the cost of the MST heuristic to that of an optimal solution than for the online greedy algorithm — the comparison can be done in one shot, rather then on an edge-by-edge basis.

For the analysis, let T^* denote a minimum-cost Steiner tree. Obtain H from T^* by adding a second copy of every edge (Figure 2(a)). Obviously, H is Eulerian (every vertex degree got doubled) and $\sum_{e \in H} c_e = 2OPT$. Let C denote an Euler tour of H — a (non-simple) closed walk using every edge of H exactly once. We again have $\sum_{e \in C} c_e = 2OPT$.

The tour C visits each of $t_1, \ldots, t_k$ at least once. "Shortcut" it to obtain a simple cycle $\widehat{C}$ on the vertex set $\{t_1, \ldots, t_k\}$ (Figure 2(b)); since the edge costs satisfy the triangle inequality, this only decreases the cost. $\widehat{C}$ minus an edge is a spanning tree of the subgraph induced by R that has cost at most $2OPT$; the MST can only be better. ■

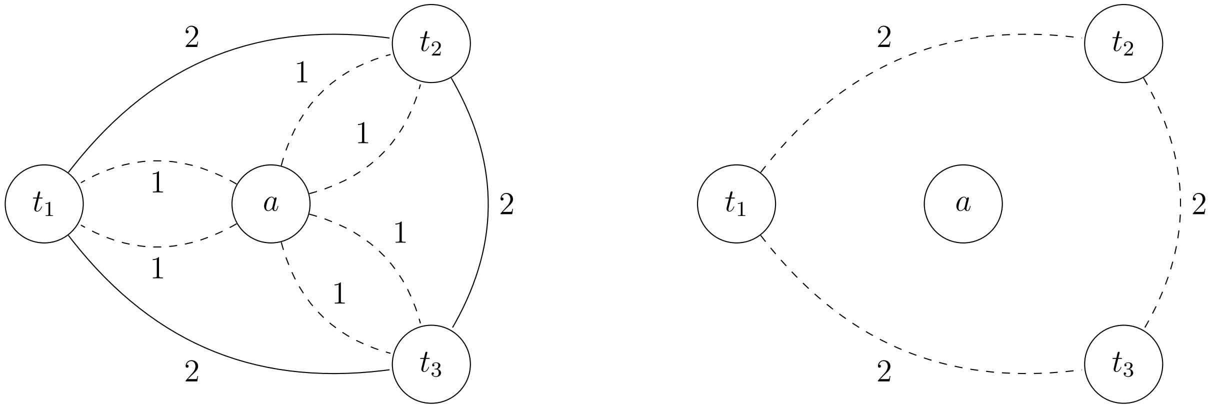

Figure 2: (a) Adding second copy of each edge in T^* to form H. Note H is Eulerian. (b) Shorting cutting edges $(\{t1, a\}, \{a, t2\})$, $(\{t2, a\}, \{a, t3\})$, $(\{t3, a\}, \{a, t1\})$ to $\{t1, t2\}, \{t2, t3\}, \{t3, t1\}$ respectively.

2.4 Example: Set Coverage

Next we study a problem that we haven't seen before, *set coverage*. This problem is a killer application for greedy algorithms in approximation algorithm design. The input is a collection $S_1, \ldots, S_m$ of subsets of some ground set U (each subset described by a list of its elements), and a budget k. The goal is to pick k subsets to maximize the size of their union (Figure 3). All else being equal, bigger sets are better for the set coverage problem. But it's not so simple — some sets are largely redundant, while others are uniquely useful (cf., Figure 3).

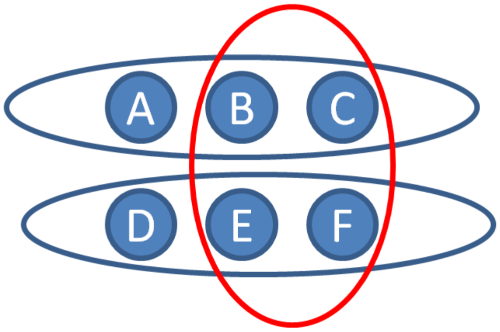

Figure 3: Example set coverage problem. If $k = 2$, we should pick the blue sets. Although the red set is the largest, picking it is redundant.

Set coverage is a basic problem that comes up all the time (often not even disguised). For example, suppose your start-up only has the budget to hire k new people. Each applicant can be thought of as a set of skills. The problem of hiring to maximize the number of distinct skills required is a set coverage problem. Similarly for choosing locations for factories/fire engines/Web caches/artisinal chocolate shops to cover as many neighborhoods as possible. Or, in machine learning, picking a small number of features to explain as much as the data as possible. Or, in HCI, given a budget on the number of articles/windows/menus/etc. that can be displayed at any given time, maximizing the coverage of topics/functionality/etc.

The set coverage problem is NP-hard. Turning to approximation algorithms, the following greedy algorithm, which increases the union size as much as possible at each iteration, seems like a natural and good idea.

Greedy Algorithm for Set Coverage

```
for $i = 1, 2, \ldots, k$: do
    compute the set $A_i$ maximizing the number of new elements covered
      (relative to $\cup_{j=1}^{i-1} A_j$)
return $\{A_1, \ldots, A_k\}$
```

This algorithm can clearly be implemented in polynomial time, so we don't expect it to always compute an optimal solution. It's useful to see some concrete examples of what can go wrong. example.

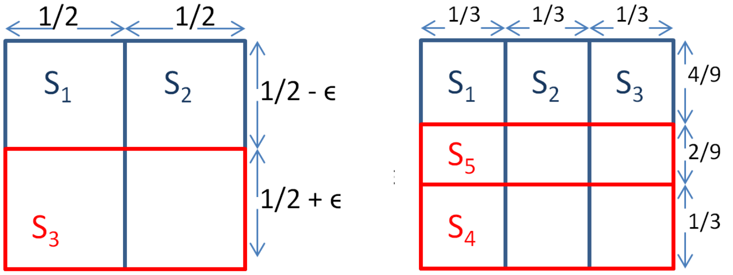

Figure 4: (a) Bad example when $k = 2$ (b) Bad example when $k = 3$.

For the first example (Figure 4(a)), set the budget $k = 2$. There are three subsets. S_1 and S_2 partition the ground set U half-half, so the optimal solution has size $|U|$. We trick the greedy algorithm by adding a third subset S_3 that covers slightly more than half the elements. The greedy algorithm then picks S_3 in its first iteration, and can only choose one of S_1, S_2 in the second iteration (it doesn't matter which). Thus the size of the greedy solution is $\approx \frac{3}{4}|U|$. Thus even when $k = 2$, the best-case scenario would be that the greedy algorithm is a $\frac{3}{4}$-approximation.

We next extend this example (Figure 4(b)). Take $k = 3$. Now the optimal solution is S_1, S_2, S_3, which partition the ground set into equal-size parts. To trick the greedy algorithm in the first iteration (i.e., prevent it from taking one of the optimal sets S_1, S_2, S_3), we add a set S_4 that covers slightly more than $\frac{1}{3}$ of the elements and overlaps evenly with S_1, S_2, S_3. To trick it again in the second iteration, note that, given S_4, choosing any of S_1, S_2, S_3 would cover $\frac{1}{3} \cdot \frac{2}{3} \cdot |U| = \frac{2}{9}|U|$ new elements. Thus we add a set S_5, disjoint from S_4, covering slightly more than a $\frac{2}{9}$ fraction of U. In the third iteration we allow the greedy algorithm to pick one of S_1, S_2, S_3. The value of the greedy solution is $\approx |U|(\frac{1}{3} + \frac{2}{9} + \frac{1}{3}\frac{4}{9}) = \frac{19}{27}|U|$. This is roughly 70% of $|U|$, so it is a worse example for the greedy algorithm than the first

Exercise Set #8 asks you to extend this family of bad examples to show that, for all k, the greedy solution could be as small as

$$1 - \left(1 - \frac{1}{k}\right)^k$$

times the size of an optimal solution. (Note that with $k = 2, 3$ we get $\frac{3}{4}$ and $\frac{19}{27}$.) This expression is decreasing with k, and approaches $1 - \frac{1}{e} \approx 63.2\%$ in the limit (since $1 - x$ approaches e^{-x} for x going to 0, recall Figure 5).[4]

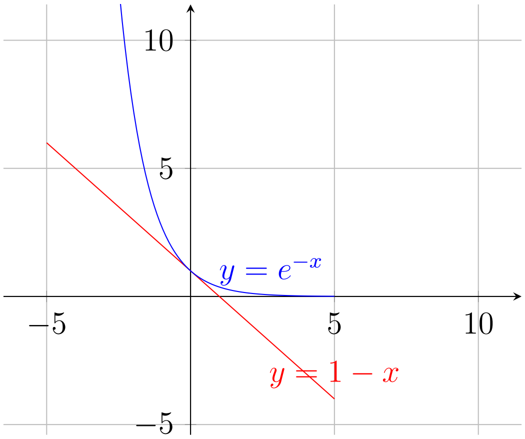

Figure 5: Graph showing $1 - x$ approaching e^{-x} for small x.

These examples show that the following guarantee is remarkable.

Theorem 2.2 *For every $k \geq 1$, the greedy algorithm is a $(1 - (1 - \frac{1}{k})^k)$-approximation algorithm for set coverage instances with budget k.*

Thus there are no worse examples for the greedy algorithm than the ones we identified above. Here's what's even more amazing: under standard complexity assumptions, there is *no* polynomial-time algorithm with a better approximation ratio![5] In this sense, the greedy algorithm is an optimal approximation algorithm for the set coverage problem.

We now turn to the proof of Theorem 2.2. The following lemma proves a sense in which the greedy algorithm makes healthy progress at every step. (This is the most common way to analyze a greedy algorithm, whether for exact or approximate guarantees.)

[4] There's that strange number again!

[5] As k grows large, that is. When k is a constant, the problem can be solved optimally in polynomial time using brute-force search.

Lemma 2.3 *Suppose that the first $i-1$ sets $A_1, \ldots, A_{i-1}$ computed by the greedy algorithm cover ℓ elements. Then the next set A_i chosen by the algorithm covers at least*

$$\frac{1}{k}(OPT - \ell)$$

new elements, where OPT is the value of an optimal solution.

Proof: As a thought experiment, suppose that the greedy algorithm were allowed to pick k new sets in this iteration. Certainly it could cover $OPT - \ell$ new elements — just pick all of the k subsets in the optimal solution. One of these k sets must cover at least $\frac{1}{k}(OPT - \ell)$ new elements, and the set A_i chosen by the greedy algorithm is at least as good. ■

Now we just need a little algebra to prove the approximation guarantee.

Proof of Theorem 2.2: Let $g_i = |\cup_{j=1}^{i} A_i|$ denote the number of elements covered by the greedy solution after i iterations. Applying Lemma 2.3, we get

$$g_k = (g_k - g_{k-1}) + g_{k-1} \geq \frac{1}{k}(OPT - g_{k-1}) + g_{k-1} = \frac{OPT}{k} + \left(1 - \frac{1}{k}\right) g_{k-1}.$$

Applying it again we get

$$g_k \geq \frac{OPT}{k} + \left(1 - \frac{1}{k}\right)\left(\frac{OPT}{k} + \left(1 - \frac{1}{k}\right) g_{k-2}\right) = \frac{OPT}{k} + \left(1 - \frac{1}{k}\right)\frac{OPT}{k} + \left(1 - \frac{1}{k}\right)^2 g_{k-3}.$$

Iterating, we wind up with

$$g_k \geq \frac{OPT}{k}\left[1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \cdots + \left(1 - \frac{1}{k}\right)^{k-1}\right].$$

(There are k terms, one per iteration of the greedy algorithm.) Recalling from your discrete math class the identity

$$1 + z + z^2 + \cdots + z^{k-1} = \frac{1 - z^k}{1 - z}$$

for $z \in (0,1)$ — just multiply both sides by $1 - z$ to verify — we get

$$g_k \geq \frac{OPT}{k} \cdot \frac{1 - (1 - \frac{1}{k})^k}{1 - (1 - \frac{1}{k})} = OPT\left[1 - \left(1 - \frac{1}{k}\right)^k\right],$$

as desired. ■

2.5 Influence Maximization

Guarantees for the greedy algorithm for set coverage and various generalizations were already known in the 1970s. But just over the last dozen years, these ideas have taken off in the data mining and machine learning communities. We'll just mention one representative and influential (no pun intended) example, due to Kempe, Kleinberg, and Tardos in 2003.

Consider a "social network," meaning a directed graph $G = (V, E)$. For our purposes, we interpret an edge (v, w) as "v influences w." (For example, maybe w follows v on Twitter.)

We next posit a simple model of how an idea/news item/meme/etc. "goes viral," called a "cascade model."[6]

- Initially the vertices in some set S are "active," all other vertices are "inactive." Every edge is initially "undetermined."
- While there is an active vertex v and an undetermined edge (v, w):
  - with probability p, edge (v, w) is marked "active," otherwise it is marked "inactive;"
  - if (v, w) is active and w is inactive, then mark w as active.

Thus whenever a vertex gets activated, it has the opportunity to active all of the vertices that it influences (if they're not already activated). Note that once a vertex is activated, it is active forevermore. A vertex can get multiple chances to be activated, corresponding to the number of its influencers who get activated. See Figure 6. In the example, note that a vertex winds up getting activated if and only if there is a path of activated edges from v to it.

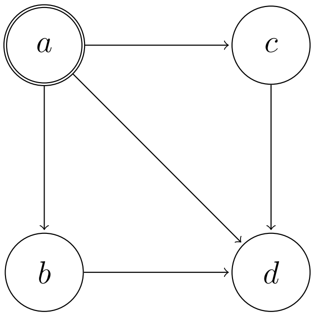

Figure 6: Example cascade model. Initially, only a is activated. b (and similarly c) can get activated by a with probability p. d has a chance to get activated by either a, b or c.

The *influence maximization problem* is, given a directed graph $G = (V, E)$ and a budget k, to compute the subset $S \subseteq V$ of size k that maximizes the expected number of active vertices at the conclusion of the cascade, given that the vertices of S are active at the beginning.

[6] Such models were originally proposed in epidemiology, to understand the spread of diseases.

(The expectation is over the coin flips made for the edges.) Denote this expected value for a set S by $f(S)$.

There is a natural greedy algorithm for influence maximization, where at each iteration we increase the function f as much as possible.

Greedy Algorithm for Influence Maximization

```
$S = \emptyset$
**for** $i = 1, 2, \ldots, k$: **do**
    add to $S$ the vertex $v$ maximizing $f(S \cup \{v\})$
return $S$
```

The same analysis we used for set coverage can be used to prove that this greedy algorithm is a $(1-(1-\frac{1}{k})^k)$-approximation algorithm for influence maximization. The greedy algorithm's guarantee holds for every function f that is "monotone" and "submodular," and the function f above is one such example (it is basically a convex combination of set coverage functions). See Problem Set #4 for details.