Approximation Algorithms

1. Computational Complexity Again

Before we discuss approximation algorithms, we need to revisit the topic of computational complexity. Recall that

P is the collection of all sets recognizable (in fact decidable) by deterministic Turing machines in polynomial time.

NP is the collection of all sets recognizable (in fact decidable) by nondeterministic Turing machines in polynomial time. (NP comes from *n*ondeterministic *p*olynomial time.)

A set L is **NP-complete** if
1. it is in NP
2. every language L' in NP is polynomial-time reducible to it

All of the above classifications deal with set recognition, which means we are asking whether a given string (because we have to encode the problem into inputs that can be written on a Turing machine tape) does or does not belong to a certain set. These are all therefore decision problems, with yes or no answers. For example, given a string that is a representation of a graph, is there or is there not a Hamiltonian circuit in this graph? The Hamiltonian circuit problem is NP-complete.

The importance of NP-complete problems is that if any NP-complete problem were ever found to have a deterministic polynomial solution, that is, if it were found to belong to P, then every NP problem would belong to P and P = NP. (NP-complete problems are "at least as hard as" any NP problem.) Since there are so many NP-complete problems and no polynomial solution has ever been found for any of them after much investigation, the general belief at this point is that $P \subset NP$. However, there's no proof of this, that is, it's never been proved that for some NP-complete problem a deterministic polynomial solution cannot exist.

Instead of just a yes/no answer, we are often interested in optimization problems. For example, if there is a Hamiltonian circuit in a given graph, what's the circuit with the smallest length? (This is the Traveling Salesman problem.) Strictly speaking, the Traveling Salesman problem, although it has no known deterministic polynomial solution, is not an NP problem, much less an NP-complete problem, because it's an optimization problem, not a decision problem. But this optimization problem can be recast as a decision problem D in the following way: Does this graph have a Hamiltonian circuit of length $\leq k$? We've picked an arbitrary k as a bound on the length of the circuit, and this problem is a decision problem that is also NP-complete.

We could sneak up on the answer to the general optimization problem by asking the decision problem D over and over for smaller and smaller values of k, but that's not the point. The point is that the general optimization problem is at least as hard to solve as the bounded decision problem D. Suppose we had a deterministic polynomial solution for the general optimization problem. Then we surely have a polynomial solution to the bounded decision problem D. In our example, this means that we take a representation of a graph, apply this hypothetical solution

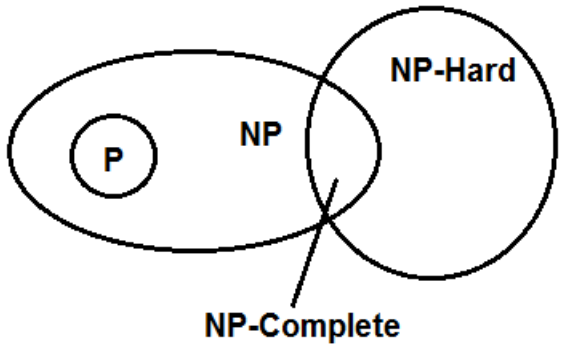
algorithm for the general optimization problem, and get an answer in polynomial time of, say 12. So this graph has a Hamiltonian circuit of length 12 and there's no shorter circuit in this graph. For the bounded decision problem, if k is ≤ 11, the answer is no, if k ≥ 12, the answer is yes. This solves the bounded decision problem in polynomial time, which is a contradiction because that problem is NP-complete. (Although our example is a minimization problem, the same idea works for maximization problems; just change the direction of the inequality.)

A problem H is **NP-hard** if every NP problem L is reducible to H in polynomial time, meaning that if you could solve H in polynomial time, then you would have a solution for any NP problem with only polynomial time additional overhead. Therefore NP-hard problems are "at least as hard as" any NP problem, i.e., an NP-hard problem satisfies property 2 of NP-completeness. To show that a problem H is NP-hard, you only have to find one NP-complete problem L* that is polynomially reducible to it because any NP problem L is polynomially reducible to L*. So the Traveling Salesman problem (the general optimization problem) is NP-hard because the NP-complete bounded decision problem D is polynomially reducible to it. In fact, in this case D is trivially reducible to the general optimization problem because there is virtually no additional work to solve D once you know the answer to the general optimization problem.

If a deterministic polynomial solution is ever found for any NP-hard problem, then there's a polynomial solution for some NP-complete problem, so again, P = NP.

So what is the difference between an NP-complete problem and an NP-hard problem? An NP-hard problem might not satisfy property 1 of NP-completeness, that is, it might not be in NP. (Note the confusing terminology in that an NP-hard problem need not be in NP.) Hence, it may or may not be a decision problem (even if it is a decision problem, it might not belong to NP). The NP-hard Traveling Salesman problem is, as noted earlier, an optimization problem, not a decision problem.

From the definitions, a problem is NP-complete if it is both in NP and is NP-hard. So below is a picture of what we SUSPECT is true.



2. What Is an Approximation Algorithm?

So let's say you have an optimization problem that is NP-hard. You are pretty sure you won't find a polynomial solution (or if you do, you'll certainly be rich and famous!). But all is not lost.

It may be that a solution algorithm that has exponential (or worse) running time is still useful, for one of two reasons. If your input size n is guaranteed to be very small, then 2^n may still be acceptable. Or, while the worst-case or even the average-case running time is exponential, it may be that the inputs of interest have characteristics that reduce the running time to an acceptable level.[1,2]

Failing these happy occurrences, the next best thing is to look for an approximation algorithm. Intuitively, a good approximation algorithm A for your problem would have two characteristics.

i) A would have provably polynomial-bounded running time.
ii) A would produce answers that are guaranteed to be "close to" the optimum result.

Of course we have to have some definition for "close to" – and how close is "good enough"? Let's let R* denote the optimum result and let R be the result produced by algorithm A. Remember that we have no efficient (polynomial) way to compute R*. But even if the value of R* is unknown to us, we can find some relationship between R and R*. If our problem is a maximization problem, then $0 < R \leq R^*$. If our problem is a minimization problem, then $0 < R^* \leq R$. One definition of "close" could be the ratio R*/R for maximization problems or R/R* for minimization problems. Both of these ratios are ≥ 1, and the closer the value is to 1, the better the approximation result.

As for the elusive R* value, because we can't readily compute it, we have to look for a lower bound on R* for a minimization problem (the optimal solution can't be less than this) or an upper bound on R* for a maximization problem (the optimal solution can't be more than this). [Recall in the competitive analysis example for amortization we worked with a completely hypothetical optimum solution OPT.]

Approximation algorithms are rather ad-hoc, depending greatly on the particulars of the problem to be solved, and there are several possible approaches to developing them. We have already discussed two algorithm design models for optimization problems, namely dynamic

---

[1] The simplex method to solve linear programming problems (which are problems seeking an optimum solution to a system of linear inequalities) had this feature. The worst-case running time was known to be exponential, but it was still a very practical algorithm because the worst case just didn't seem to occur very often. In 1984, an Indian mathematician named Narendra Karmarkar, working for Bell Laboratories, actually found a polynomially-bounded algorithm for linear programming. Obviously linear programming is not an NP-hard problem, or else this discovery would say that P = NP, but if you require integer solutions to all the variables in the problem, then it is an NP-hard problem.

[2] Remember from the group presentation on SAT (satisfiability of Boolean expressions) that, while the general problem is NP-complete, many instances of that problem do have polynomial solutions, and that research group has identified some characteristics that result in efficient runtime using appropriate algorithms.

programming and greedy algorithms, so perhaps one of these approaches could be used for an approximation algorithm. We'll do one example problem where a greedy approach is useful.

3. The Bin-Packing Problem

The sequence $s_1, \ldots, s_n$ represents the sizes of a set of n items. For each item i, $1 \le i \le n$, $0 \le s_i \le 1.0$. There are n bins available in which to store the n items; each bin has capacity 1. What is the minimum number of bins required to store all n objects?

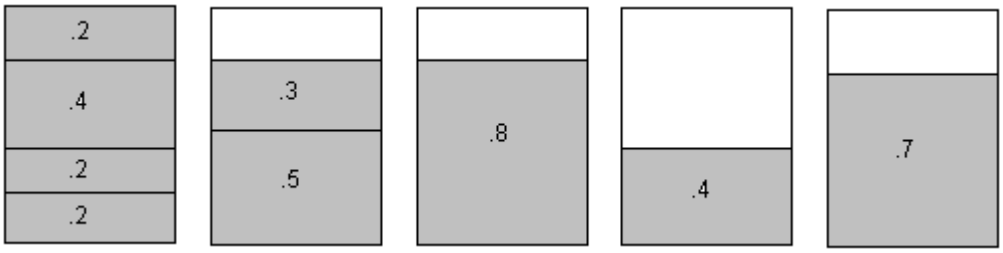
The solution to this problem obviously has an upper bound as we could just put each of the n items into its own bin. But we want a minimum solution. This optimization problem is known to be NP-hard, so there's no known deterministic polynomial-time solution. (There is a brute-force, very inefficient solution – just try all possible ways of packing the n items in the n bins and pick out the one that uses the fewest bins.) The best we can do is try to find an approximation algorithm.

First Fit:

This is the way bin-packing often works for carry-on on an airplane. You put your stuff into the first bin you come to in which it will fit (because you are greedy). You don't start using a new bin until all the previous bins are too full.

Here is an example for the sequence

0.2, 0.2, 0.4, 0.5, 0.8, 0.3, 0.4, 0.2, 0.7



(5 bins required)

The key part of the pseudocode would be something like this:

```
j is index for bins
i is index for items
   for j = 1..n
      used[j] = 0       //all bins are initially empty
   for i = 1..n         //pack the items into bins
      j = 1
      while (used[j] + s_i) > 1  //not enough room in bin j
         j = j + 1
      end while
      bin[i] = j
      used[j] = used[j] + s_i
   end for
```

In the worst case, each item gets stored in its own bin and the body of the while loop will be executed

0 times for i = 1
1 time for i = 2
2 times for i = 3
…
n – 1 times for i = n

for a total of 1 + 2 + … + (n – 1) = (n-1)n/2. The initialization is $\Theta(n)$, so the worst-case running time for this algorithm is $\Theta(n^2)$. It's a polynomial algorithm, satisfying condition (i). Now, how good is it?

First, we can make an observation about the optimal solution R*. The governing principle is: the fuller we can fill each used bin, the fewer bins we need. If we could completely fill every bin used except the last one, that would be

$$\left\lceil \sum_{i=1}^{n} s_i \right\rceil \text{ bins}$$

so

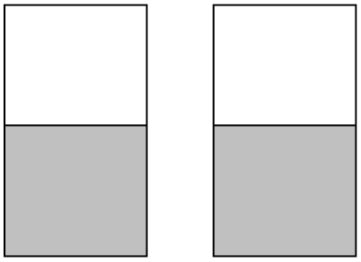
$$R^* \geq \left\lceil \sum_{i=1}^{n} s_i \right\rceil$$

and we have found a lower bound for R*.

On the other hand, let R_{FF} be the result from the First Fit algorithm. Assuming more than one bin is needed, there can be at most one used bin that is less than or equal to half full: If bin_k and bin_m are both less than or equal to half full, where k < m, then whatever was put into bin_m is of size ≤ 0.5 and would have been put into the earlier bin_k instead. So if the total size of the items fills more than half of every bin except one, then the number of bins need be no more than twice this total size, so that

$$R_{FF} \leq \left\lceil 2\sum_{i=1}^{n} s_i \right\rceil$$

(To see this, note that if each bin were exactly half full, which we just saw is impossible, it would take exactly twice as many bins as the sum of the sizes;



$\Sigma s_i = ½ + ½ = 1$ but takes 2 bins

in reality, the bins are more tightly packed, requiring fewer bins.)

Therefore

$$R_{FF} \le \left\lceil 2\sum_{i=1}^{n} s_i \right\rceil \le 2\left\lceil \sum_{i=1}^{n} s_i \right\rceil \le 2R^*$$

and First Fit gives a solution where the ratio of R_{FF}/R^* is no more than 2. (It has actually been proved that the ratio is slightly more than 1.7.)

Best Fit:

A variation on First Fit is to put item i in the bin that has the smallest space left that is greater than or equal to s_i (and the lowest index for such a bin if there is a tie). In other words, squeeze it into the bin with the closest fit. It has been proven that the worst case for this algorithm is about the same as First Fit.

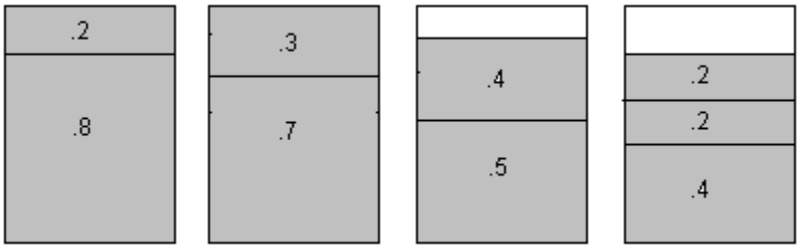
First Fit Decreasing:

The First Fit strategy seems to work worst when small items are stored early, leaving big items to stuff in somewhere at the end. This observation leads to a third variation where the items are first sorted by decreasing size. Of course you have to know what all the item sizes are ahead of time to be able to sort them. Remember in the list access problem, we scorned this possibility when we developed the Move-To-Front algorithm because the list access sequence was supposed to be arbitrary. Likewise we could say here that the sequence of item sizes is also arbitrary, but it seems more likely (at least if the items are "physical entities") that we might actually know what all the sizes are ahead of time.

The cost to sort the n sizes is at worst $\Theta(n^2)$, followed by the worst-case $\Theta(n^2)$ performance of the First Fit algorithm, so First Fit Decreasing (FDD) is a polynomial algorithm.

Consider our previous example, where the input items have been sorted in decreasing order:

0.8, 0.7, 0.5, 0.4, 0.4, 0.3, 0.2, 0.2, 0.2



(4 bins required. This happens to be the optimum solution for this set of values because

$$\sum_{i=1}^{n} s_i = 3.7, \text{ so } \left\lceil \sum_{i=1}^{n} s_i \right\rceil = 4$$

but that's just a coincidence.)

So let us assume that

$$s_1 \geq s_2 \geq \ldots \geq s_n$$

and see what we can say about R_{FFD} compared to R*. Because this is a minimization problem, $R_{FFD} \geq R^*$, so more bins may be needed.

Result 1: All of the items placed by FFD in bins with index beyond R*, that is, extra bins needed by FFD that are not needed in the optimum solution, have size ≤ 1/3.

proof: To prove this, we only need to show that the first item i stored in an extra bin has size ≤ 1/3 because none of the remaining items to be stored are any larger. Assume instead that $s_i > 1/3$. Then none of the items previously stored (in the non-extra bins) were any smaller, so their size is all > 1/3 and no more than 2 such items could be stored in each bin.

Just before item i is inserted in the first extra bin, there is some k, $0 \leq k \leq R^*$, where the first k bins each hold one item and the rest of the R* - k bins hold two items. This seems intuitive because large items are stored first, but for a more formal proof, suppose there are two bins, bin_x and bin_y with x < y such that bin_x has two items of size x_1 and x_2, with $x_1 \geq x_2$, and bin_y has one item of size y_1. If $y_1 > x_1$, then item y_1 would have come before x_1 in the sequence and been stored before x_1 was stored, so it would be in bin_x (or an earlier bin). Therefore $x_1 \geq y_1$. Also x_2 has been stored and s_i has not, so $x_2 \geq s_i$. Adding these two inequalities results in

$$x_1 + x_2 \geq y_1 + s_i$$

which means that s_i would fit in bin_y and would not have to go into an extra bin.

So if $s_i > 1/3$, we have one item in each of the first k bins and 2 items in the next R* - k bins. The items $s_{k+1} \ldots s_{i-1}$ do not fit into any of the first k bins or FFD would have put them there. So the k items in the first k bins are too big to share a bin with $s_{k+1} \ldots s_{i-1}$, and this will be true for any solution, including the optimal solution. That leaves, in the optimal solution, $s_{k+1} \ldots s_{i-1}$ items to fit into the remaining R* - k bins, and we know from FFD that there are 2(R*- k) such items. Each is of size > 1/3, so no more than two per bin, and if $s_i > 1/3$, there's no place for the optimal solution to put it. Contradiction, so $s_i \leq 1/3$.

Result 2: For any sequence $s_1, \ldots, s_n$, FFD puts at most R* - 1 items in extra bins.

proof: Assume that FFD puts ≥ R* items in extra bins. In particular, consider $t_1, \ldots, t_{R^*}$ as the sizes of the first R* items stored in extra bins. Also, let $w_1, \ldots, w_{R^*}$ be the total final amount stored by FFD in each bin j, $1 \leq j \leq R^*$. If $w_i + t_i \leq 1$, FFD would have stored t_i in bin i, so $w_i + t_i > 1$. Then

$$\sum_{i=1}^{n} s_i \geq \sum_{i=1}^{R^*} w_i + \sum_{i=1}^{R^*} t_i \quad \text{because there could be more than R* items in extra bins}$$

$$= \sum_{i=1}^{R^*} (w_i + t_i) > R^* \quad \text{(adding up R* items, each greater than 1)}$$

but we know that

$$\sum_{i=1}^{n} s_i \le R^*$$

because all the items fit in R* bins. Contradiction, so FFD puts < R* items in extra bins.

Result 3: $R_{FFD} \le \frac{4}{3}R^* - \frac{1}{3}$

proof: From the two previous results, FFD stores at most R*-1 items in extra bins, each of size ≤ 1/3. Thus, at least 3 items can go in each extra bin, so there can be at most (R*- 1)/3 extra bins needed. Thus, the total number of bins needed by FFD is

$$R_{FFD} \le R^* + (R^* - 1)/3 = (4R^* - 1)/3 = (4/3)R^* - 1/3 = (12/9)R^* - 1/3$$

A little history of upper bounds:
$R_{FFD} \le (11/9)R^* + 4$ (1973) – proof took 100 pages
$R_{FFD} \le (11/9)R^* + 3$ (1985)
$R_{FFD} \le (11/9)R^* + 1$ (1991)
$R_{FFD} \le (11/9)R^* + 7/9$ (1997) – conjecture, not complete proof
$R_{FFD} \le (11/9)R^* + 6/9$ (2007) – proof took 30 pages and 1 year; this is a tight upper bound, that is, there is an example where the equality holds (so the 6/9 can never be further reduced).

Note that these results do not represent improved algorithms, as there were new algorithms that made marginal improvements over Strassen's algorithm, only improved proofs for the existing First Fit Descending algorithm.

Applications:

The bin-packing problem has many applications, such as file storage in fixed-size disk sectors, music storage on a set of DVDs, job scheduling on a number of identical processors, cutting patterns from stock (2-D bin packing) – the goal is to minimize the number of disk sectors, DVDs, processors, or stock. A paper published in 2007 in a journal called Anesthesia and Analgesia is titled "Improving Operating Room Efficiency by Applying Bin-Packing and Portfolio Techniques to Surgical Case Scheduling"**.**

Lots of research has been done on different algorithms for solving bin-packing problems and related problems with varying constraints, such as certain items can't go into the same bin.