Le mieux est l'ennemi du bien. [The best is the enemy of the good.]

— Voltaire, *La Bégueule* (1772)

Who shall forbid a wise skepticism, seeing that there is no practical question on which any thing more than an approximate solution can be had?

— Ralph Waldo Emerson, *Representative Men* (1850)

We dance round in a ring and suppose,
But the secret sits in the middle and knows.

— Robert Frost, "The Secret Sits" (1942)

Now, distrust of corporations threatens our still-tentative economic recovery; it turns out greed is bad, after all.

— Paul Krugman, "Greed is Bad", *The New York Times* (June 4, 2002)

J

Approximation Algorithms

[Read Chapter 12 first.]

Status: Needs revision.

J.1 Load Balancing

On the future viral-hit online reality game show *Grunt Work*, broadcast every Wednesday at 3am on YouTube Green, the contestants are given a series of utterly pointless tasks to perform. Each task has a predetermined time limit; for example, "Sharpen this pencil for 17 seconds," or "Pour pig's blood on your head and sing The Star-Spangled Banner for two minutes," or "Listen to this 75-minute algorithms lecture." The directors of the show want you to assign each task to one of the contestants, so that the last task is completed as early as possible. When your predecessor correctly informed the directors that their problem is NP-hard, he was immediately fired. "Time is money!" they screamed at him. "We don't need perfection. Wake up, dude, this is the *internet*!"

Less facetiously, suppose we have a set of n jobs, which we want to assign to m machines. We are given an array $T[1..n]$ of non-negative numbers, where $T[j]$ is the running time of job j. We can describe an *assignment* by an array $A[1..n]$, where

$A[j] = i$ means that job j is assigned to machine i. The *makespan* of an assignment is the maximum time that any machine is busy:

$$makespan(A) = \max_i \sum_{A[j]=i} T[j]$$

The *load balancing* problem is to compute the assignment with the smallest possible makespan.

It's not hard to prove that the load balancing problem is NP-hard by reduction from Partition: The array $T[1..n]$ can be evenly partitioned if and only if there is an assignment to two machines with makespan exactly $\sum_i T[i]/2$. A slightly more complicated reduction from 3Partition implies that the load balancing problem is *strongly* NP-hard. If we really need the optimal solution, there is a dynamic programming algorithm that runs in time $O(nM^m)$, where M is the minimum makespan, but that's just horrible.

There is a fairly natural and efficient greedy heuristic for load balancing: consider the jobs one at a time, and assign each job to the machine i with the earliest finishing time $Total[i]$.

```
GreedyLoadBalance(T[1..n], m):
  for i ← 1 to m
      Total[i] ← 0

  for j ← 1 to n
      mini ← arg min_i Total[i]
      A[j] ← mini
      Total[mini] ← Total[mini] + T[j]

  return A[1..m]
```

Theorem J.1. *The makespan of the assignment computed by* GreedyLoadBalance *is at most twice the makespan of the optimal assignment.*

Proof: Fix an arbitrary input, and let *OPT* denote the makespan of its optimal assignment. The approximation bound follows from two trivial observations. First, the makespan of any assignment (and therefore of the optimal assignment) is at least the duration of the longest job. Second, the makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

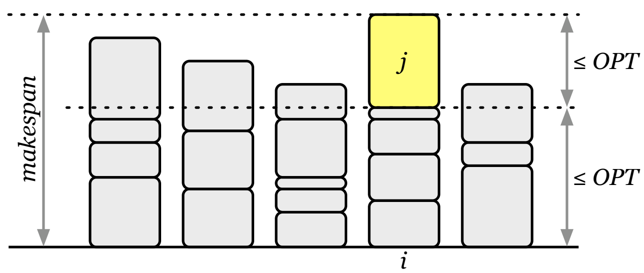
$$OPT \geq \max_j T[j] \qquad \text{and} \qquad OPT \geq \frac{1}{m}\sum_{j=1}^{n} T[j]$$

Now consider the assignment computed by GreedyLoadBalance. Suppose machine i has the largest total running time, and let j be the last job assigned to machine i. Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that

$Total[i] - T[j] \le OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \le Total[k]$ for all k. (Some values $Total[k]$ may have increased since job j was assigned, but that only helps us.) In particular, $Total[i] - T[j]$ is less than or equal to the *average* finishing time over all machines. Thus,

$$Total[i] - T[j] \le \frac{1}{m}\sum_{i=1}^{m} Total[i] = \frac{1}{m}\sum_{j=1}^{n} T[j] \le OPT$$

by our second trivial observation. We conclude that the makespan $Total[i]$ is at most $2 \cdot OPT$. □



Proof that GREEDYLOADBALANCE is a 2-approximation algorithm

GREEDYLOADBALANCE is an *online* algorithm: It assigns jobs to machines in the order that the jobs appear in the input array. Online approximation algorithms are useful in settings where inputs arrive in a stream of unknown length—for example, real jobs arriving at a real scheduling algorithm. In this online setting, it may be *impossible* to compute an optimum solution, even in cases where the offline problem (where all inputs are known in advance) can be solved in polynomial time. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

In our original offline setting, we can improve the approximation factor by sorting the jobs before piping them through the greedy algorithm.

```
SORTEDGREEDYLOADBALANCE(T[1..n], m):
  sort T in decreasing order
  return GREEDYLOADBALANCE(T, m)
```

Theorem J.2. *The makespan of the assignment computed by* SORTEDGREEDYLOADBALANCE *is at most* $3/2$ *times the makespan of the optimal assignment.*

Proof: Let i be the busiest machine in the schedule computed by SORTEDGREEDYLOADBALANCE. If only one job is assigned to machine i, then the greedy schedule is actually optimal, and the theorem is trivially true. Otherwise, let j be the last job assigned to machine i. Since each of the first m jobs is assigned to a unique machine, we must have $j \ge m+1$. As in the previous proof, we know that $Total[i] - T[j] \le OPT$.

In *any* schedule, at least two of the first $m+1$ jobs, say jobs k and ℓ, must be assigned to the same machine. Thus, $T[k]+T[\ell] \le OPT$. Since $\max\{k,\ell\} \le m+1 \le j$, and the jobs are sorted in decreasing order by duration, we have

$$T[j] \le T[m+1] \le T[\max\{k,\ell\}] = \min\{T[k], T[\ell]\} \le OPT/2.$$

We conclude that the makespan $Total[i]$ is at most $3 \cdot OPT/2$, as claimed. □

In fact, neither of these analyses is tight. The exact approximation ratio of GreedyLoadBalance is actually $2-1/m$, and the exact approximation ratio of SortedGreedyLoadBalance is actually $4/3-1/(3m)$.

J.2 Generalities

Consider an arbitrary optimization problem. Let $OPT(X)$ denote the value of the optimal solution for a given input X, and let $A(X)$ denote the value of the solution computed by algorithm A given the same input X. We say that A is an ***$\alpha(n)$-approximation algorithm*** if and only if

$$\frac{OPT(X)}{A(X)} \le \alpha(n) \quad\text{and}\quad \frac{A(X)}{OPT(X)} \le \alpha(n)$$

for all inputs X of size n. The function $\alpha(n)$ is called the ***approximation factor*** for algorithm A. For any given algorithm, only one of these two inequalities will be important. For maximization problems, where we want to compute a solution whose cost is as small as possible, the first inequality is trivial. For maximization problems, where we want a solution whose value is as large as possible, the second inequality is trivial. A 1-approximation algorithm always returns the exact optimal solution.

Especially for problems where exact optimization is NP-hard, we have little hope of completely characterizing the optimal solution. The secret to proving that an algorithm satisfies some approximation ratio is to find a useful function of the input that provides both lower bounds on the cost of the optimal solution and upper bounds on the cost of the approximate solution. For example, if $OPT(X) \ge f(X)/2$ and $A(X) \le 5f(X)$ for *any* function f, then A is a 10-approximation algorithm. Finding the right intermediate function can be a delicate balancing act.

J.3 Greedy Vertex Cover

Recall that the *vertex color* problem asks, given a graph G, for the smallest set of vertices of G that cover every edge. This is one of the first NP-hard problems introduced in the first week of class. There is a natural and efficient greedy heuristic[1] for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

[1]A *heuristic* is an algorithm that doesn't work.

```
GREEDYVERTEXCOVER(G):
  C ← ∅
  while G has at least one edge
      v ← vertex in G with maximum degree
      G ← G \ v
      C ← C ∪ v
  return C
```

Obviously this algorithm doesn't compute the optimal vertex cover—that would imply P=NP!—but it does compute a reasonably close approximation.

Theorem J.3. *GreedyVertexCover is an $O(\log n)$-approximation algorithm.*

Proof: For all i, let G_i denote the graph G after i iterations of the main loop, and let d_i denote the maximum degree of any node in G_{i-1}. We can define these variables more directly by adding a few extra lines to our algorithm:

```
GREEDYVERTEXCOVER(G):
  C ← ∅
  G_0 ← G
  i ← 0
  while G_i has at least one edge
      i ← i + 1
      v_i ← vertex in G_{i−1} with maximum degree
      d_i ← deg_{G_{i−1}}(v_i)
      G_i ← G_{i−1} \ v_i
      C ← C ∪ v_i
  return C
```

Let $|G_{i-1}|$ denote the number of edges in the graph G_{i-1}. Let C^* denote the optimal vertex cover of G, which consists of OPT vertices. Since C^* is also a vertex cover for G_{i-1}, we have

$$\sum_{v \in C^*} \deg_{G_{i-1}}(v) \geq |G_{i-1}|.$$

In other words, the *average* degree in G_i of any node in C^* is at least $|G_{i-1}|/OPT$. It follows that G_{i-1} has at least one node with degree at least $|G_{i-1}|/OPT$. Since d_i is the maximum degree of any node in G_{i-1}, we have

$$d_i \geq \frac{|G_{i-1}|}{OPT}$$

Moreover, for any $j \geq i-1$, the subgraph G_j has no more edges than G_{i-1}, so $d_i \geq |G_j|/OPT$. This observation implies that

$$\sum_{i=1}^{OPT} d_i \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{i-1}|}{OPT} \;\geq\; \sum_{i=1}^{OPT} \frac{|G_{OPT}|}{OPT} \;=\; |G_{OPT}| \;=\; |G| - \sum_{i=1}^{OPT} d_i.$$

In other words, the first *OPT* iterations of GREEDYVERTEXCOVER remove at least half the edges of G. Thus, after at most $OPT \lg|G| \le 2\,OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GREEDYVERTEXCOVER computes a vertex cover of size $O(OPT \log n)$. □

So far we've only proved an *upper bound* on the approximation factor of GREEDYVERTEXCOVER; perhaps a more careful analysis would imply that the approximation factor is only $O(\log\log n)$, or even $O(1)$. Alas, no such improvement is possible. For any integer n, a simple recursive construction gives us an n-vertex graph for which the greedy algorithm returns a vertex cover of size $\Omega(OPT \cdot \log n)$. Details are left as an exercise for the reader.

J.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: ***minimum set cover*** and ***minimum hitting set***. The input for both of these problems is a *set system* $(X, \mathcal{F})$, where X is a finite *ground set*, and $\mathcal{F}$ is a family of subsets of X. A *set cover* of a set system $(X, \mathcal{F})$ is a subfamily of sets in $\mathcal{F}$ whose union is the entire ground set X. A *hitting set* for $(X, \mathcal{F})$ is a subset of the ground set X that intersects every set in $\mathcal{F}$.

An undirected graph can be cast as a set system in two different ways. In one formulation, the ground set X contains the vertices, and each edge defines a set of two vertices in $\mathcal{F}$. In this formulation, a vertex cover is a hitting set. In the other formulation, the *edges* are the ground set, the *vertices* define the family of subsets, and a vertex cover is a set cover.

Here are the natural greedy algorithms for finding a small set cover and finding a small hitting set. Essentially the same analysis as for our greedy algorithm for vertex cover implies that GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover, and that GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set. These are the best approximation ratios possible (up to lower order terms) unless P=NP.

```
GREEDYSETCOVER(X, 𝓕):
  𝓒 ← ∅
  while X ≠ ∅
      S ← arg max_{S∈𝓕} |S ∩ X|
      X ← X \ S
      𝓒 ← 𝓒 ∪ {S}
  return 𝓒
```

```
GREEDYHITTINGSET(X, 𝓕):
  H ← ∅
  while 𝓕 ≠ ∅
      x ← arg max_{x∈X} |{S ∈ 𝓕 | x ∈ S}|
      𝓕 ← 𝓕 \ {S ∈ 𝓕 | x ∈ S}
      H ← H ∪ {x}
  return H
```

The similarity between these two algorithms is no coincidence. For any set system $(X, \mathcal{F})$, there is a *dual* set system $(\mathcal{F}, X^*)$ defined as follows. For any element $x \in X$ in the ground set, let x^* denote the subfamily of sets in $\mathcal{F}$ that contain x:

$$x^* = \{S \in \mathcal{F} \mid x \in S\}.$$

Finally, let X^* denote the collection of all subsets of the form x^*:

$$X^* = \{x^* \mid x \in S\}.$$

As an example, suppose X is the set of letters of alphabet and $\mathcal{F}$ is the set of last names of student taking CS 473 this semester. Then X^* has 26 elements, each containing the subset of CS 473 students whose last name contains a particular letter of the alphabet. For example, $\mathtt{m}^*$ is the set of students whose last names contain the letter $\mathtt{m}$.

There is a natural bijection between the ground set X and the dual set family X^*. It is a tedious but instructive exercise to prove that the dual of the dual of any set system is isomorphic to the original set system—$(X^*, \mathcal{F}^*)$ is essentially the same as $(X, \mathcal{F})$. It is also easy to prove that a set cover for any set system $(X, \mathcal{F})$ is also a hitting set for the dual set system $(\mathcal{F}, X^*)$, and therefore a hitting set for any set system $(X, \mathcal{F})$ is isomorphic to (or more simply, "is") a set cover for the dual set system $(\mathcal{F}, X^*)$.

J.5 Vertex Cover Redux

The greedy approach doesn't always lead to the best approximation algorithms. Consider the following obviously stupid heuristic for vertex cover:

```
DumbVertexCover(G):
  C ← ∅
  while G has at least one edge
      uv ← any edge in G
      G ← G \ {u, v}
      C ← C ∪ {u, v}
  return C
```

The minimum vertex cover—in fact, *every* vertex cover—contains at least one of the two vertices u and v chosen inside the while loop. It follows immediately that DumbVertexCover is a 2-approximation algorithm! Surprisingly, this "obviously stupid" algorithm is essentially the best polynomial-time approximation algorithm known.

The same stupid idea can be extended to approximate the minimum hitting set for any set system $(X, \mathcal{F})$; the resulting approximation factor is equal to the size of the largest set in $\mathcal{F}$.

J.6 Lightest Vertex Cover: LP Rounding

Now consider a generalization of the minimum vertex cover problem in which every vertex v of the input graph has a non-negative weight $w(v)$, and the goal is to compute a vertex cover C with minimum total weight $w(C) = \sum_{v \in C} w(v)$. Both the greedy and the stupid approximation algorithms can perform arbitrarily badly in this setting. Let G be a path of length 2, where the endpoints have weight 1 and the interior vertex has weight

10^{100}. The lightest vertex cover has weight 2, but the greedy vertex cover has weight 10^{100}, and the stupid vertex cover has weight $10^{100}+1$.

But we can recover a 2-approximation algorithm by casting the problem as an instance of ***integer linear programming***. An integer linear program is just a linear program with the added constraint that all variables are integers. Almost any of the NP-hard problems considered in the previous lecture note can be cast as integer linear programs, which implies that integer linear programming is NP-hard. In particular, the following integer linear program encodes the lightest vertex cover problem. Each feasible solution to this ILP corresponds to some vertex cover; each variable $x(v)$ indicates whether that vertex cover contains the corresponding vertex v.

$$\boxed{\begin{array}{lll} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & x(v)\in\{0,1\} & \text{for every vertex } v \end{array}}$$

Let OPT denote the weight of the lightest vertex cover; OPT is also the optimal objective value for this integer linear program.

Now we're going to do something weird. Consider the linear program obtained from this ILP by removing the integrality constraint; this linear program is usually called the ***linear relaxation*** of the integer linear program.

$$\boxed{\begin{array}{lll} \text{minimize} & \displaystyle\sum_v w(v)\cdot x(v) & \\ \text{subject to} & x(u)+x(v)\geq 1 & \text{for every edge } uv \\ & 0\leq x(v)\leq 1 & \text{for every vertex } v \end{array}}$$

Like all linear programs, this one can be solved in polynomial time; let x^* denote its optimal solution, and let $\widetilde{\boldsymbol{OPT}}=\sum_v w(v)\cdot x^*(v)$ denote the denote the optimal objective value. Unfortunately, the optimal solution may not be integral; the "indicator" variables $x^*(v)$ may be rational numbers between 0 and 1. Thus, x^* is often called the optimal ***fractional*** solution. Nevertheless, this solution is useful for two reasons.

First, because every feasible solution to the original ILP is also feasible for its linear relaxation, the optimal fractional solution is a lower bound on the optimal integral solution: $\boldsymbol{OPT}\geq\widetilde{\boldsymbol{OPT}}$. As we've already seen, finding lower bounds on the optimal solution is the key to deriving good approximation algorithms.

Second, we can derive a good approximation to the lightest vertex cover by ***rounding*** the optimal fractional solution. Specifically, for each vertex v, let

$$x'(v)=\begin{cases}1 & \text{if } x^*(v)\geq 1/2,\\ 0 & \text{otherwise.}\end{cases}$$

For every edge uv, the constraint $x^*(u)+x^*(v) \geq 1$ implies that $\max\{x^*(u), x^*(v)\} \geq 1/2$, and therefore either $x'(u) = 1$ or $x'(v) = 1$. Thus, x' is the indicator function of a vertex cover. On the other hand, we have $x'(v) \leq 2x^*(v)$ for every vertex v, which implies that

$$\sum_v w(v) \cdot x'(v) \;\leq\; 2\sum_v w(v) \cdot x^*(v) \;=\; 2 \cdot \widetilde{OPT} \;\leq\; 2 \cdot OPT.$$

So we've just described a simple polynomial-time 2-approximation algorithm for lightest vertex cover: Compute the optimal fractional solution, and round it as described above.

J.7 Randomized LP Rounding

We can also round the optimal fractional solution *randomly* by interpreting each fractional value $x^*(v)$ as a probability. For each vertex v, independently set

$$x'(v) = \begin{cases} 1 & \text{with probability } x^*(v), \\ 0 & \text{otherwise.} \end{cases}$$

Our usual argument from linearity of expectation immediately implies that

$$\mathrm{E}\left[\sum_v w(v) \cdot x'(v)\right] \;=\; \sum_v w(v) \cdot x^*(v) \;=\; \widetilde{OPT} \;\leq\; OPT.$$

Unfortunately, x' is not necessarily the indicator function of a vertex cover. The probability that any particular edge uv is uncovered is

$$\Pr[x'(u) = 0 \wedge x'(v) = 0] \;=\; (1 - x^*(u))(1 - x^*(v)).$$

Thanks to the constraint $x^*(u) = x^*(v) \geq 1$, the expression $(1 - x^*(u))(1 - x^*(v))$ is minimized when $x^*(v) = x^*(u) = 1/2$. It follows that each edge uv is covered with probability at least $3/4$; equivalently, we expect x' to cover $3/4$ of the edges of the graph.

To construct an actual vertex cover, we can run the randomized rounding algorithm several times in succession, and take the union of the resulting vertex sets.

```
ApproxLightestVertexCover(G):
  C ← ∅
  x* ← optimal fractional solution
  for i ← 1 to 2 lg n
      for all vertices v
          with probability x*(v)
              C ← C ∪ {x}
  return C
```

Because the random choices in each iteration of the main loop are independent, we can simplify the algorithm by changing the rounding criterion as follows, where $N = 2\lg n$:

$$x'(v) = \begin{cases} 1 & \text{with probability } 1-(1-x^*(v))^N, \\ 0 & \text{otherwise,} \end{cases}$$

Let C be the set of vertices returned by this algorithm. Each edge uv is left uncovered by C with probability $(1-x^*(u))^N(1-x^*(v))^N \le 1/4^N = 1/n^4$. It follows that C is a vertex cover with probability at least $1-1/n^2$. On the other hand, linearity of expectation immediately implies that the expected size of C is exactly $N \cdot \widetilde{OPT} \le 2\lg n \cdot OPT$.

If we want to *guarantee* a vertex cover, we can modify the main loop in APPROXLIGHTEST-VERTEXCOVER to continue running until every edge is covered. The loop will terminate after $O(\log n)$ iterations with high probability, and the expected size of the resulting vertex cover is still at most $O(\log n) \cdot OPT$. Thus, at least in expectation, we obtain the same $O(\log n)$ approximation ratio (at least in expectation) as our original greedy algorithm.

This randomized rounding strategy can be generalized to the lightest *set* cover problem, where each subset in the input family has an associated weight, and the goal is to find the lightest collection of subsets that covers the ground set. The resulting rounding algorithm computes a set cover whose expected weight is at most $O(\log|\mathcal{F}|)$ times the weight of the lightest set cover, matching the performance of the greedy algorithm for unweighted set cover. The dual of this algorithm is a randomized $O(\log|X|)$-approximation algorithm for weighted hitting set (where every item in the ground set has a weight).

★★★

Derandomization: Greedy set cover with prices with proof via LP duality

J.8 Traveling Salesman: The Bad News

The ***traveling salesman problem***[2] problem asks for the shortest Hamiltonian cycle in a weighted undirected graph. To keep the problem simple, we can assume without loss of generality that the underlying graph is always the complete graph K_n for some integer n; thus, the input to the traveling salesman problem is just a list of the $\binom{n}{2}$ edge lengths.

Not surprisingly, given its similarity to the Hamiltonian cycle problem, it's quite easy to prove that the traveling salesman problem is NP-hard. Let G be an arbitrary undirected graph with n vertices. We can construct a length function for K_n as follows:

$$\ell(e) = \begin{cases} 1 & \text{if } e \text{ is an edge in } G, \\ 2 & \text{otherwise.} \end{cases}$$

[2] This is sometimes bowdlerized into the traveling sales*person* problem. That's just silly. Who ever heard of a traveling salesperson sleeping with the farmer's child?

Now it should be obvious that if G has a Hamiltonian cycle, then there is a Hamiltonian cycle in K_n whose length is exactly n; otherwise every Hamiltonian cycle in K_n has length at least $n+1$. We can clearly compute the lengths in polynomial time, so we have a polynomial time reduction from Hamiltonian cycle to traveling salesman. Thus, the traveling salesman problem is NP-hard, even if all the edge lengths are 1 and 2.

There's nothing special about the values 1 and 2 in this reduction; we can replace them with any values we like. By choosing values that are sufficiently far apart, we can show that even approximating the shortest traveling salesman tour is NP-hard. For example, suppose we set the length of the 'absent' edges to $n+1$ instead of 2. Then the shortest traveling salesman tour in the resulting weighted graph either has length exactly n (if G has a Hamiltonian cycle) or has length at least $2n$ (if G does not have a Hamiltonian cycle). Thus, if we could approximate the shortest traveling salesman tour within a factor of 2 in polynomial time, we would have a polynomial-time algorithm for the Hamiltonian cycle problem.

Pushing this idea to its limits us the following negative result.

Theorem J.4. *For any function $f(n)$ that can be computed in time polynomial in n, there is no polynomial-time $f(n)$-approximation algorithm for the traveling salesman problem on general weighted graphs, unless P=NP.*

J.9 Traveling Salesman: The Good News

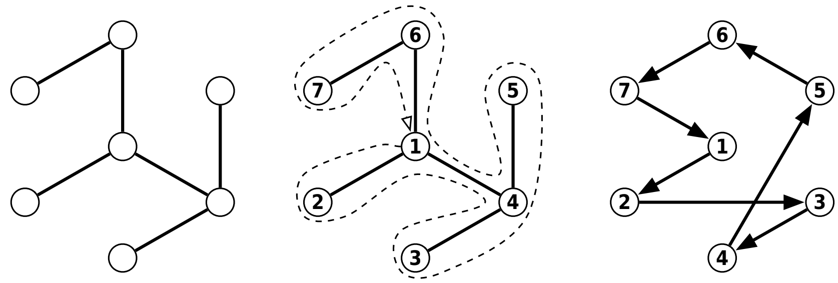
Even though the general traveling salesman problem can't be approximated, a common special case can be approximated fairly easily. The special case requires the edge lengths to satisfy the so-called *triangle inequality*:

$$\ell(u,w) \le \ell(u,v) + \ell(v,w) \quad \text{for any vertices } u, v, w.$$

This inequality is satisfied for *geometric graphs*, where the vertices are points in the plane (or some higher-dimensional space), edges are straight line segments, and lengths are measured in the usual Euclidean metric. Notice that the length functions we used above to show that the general TSP is hard to approximate do not (always) satisfy the triangle inequality.

With the triangle inequality in place, we can quickly compute a 2-approximation for the traveling salesman tour as follows. First, we compute the minimum spanning tree T of the weighted input graph; this can be done in $O(n^2 \log n)$ time (where n is the number of vertices of the graph) using any of several classical algorithms. Second, we perform a depth-first traversal of T, numbering the vertices in the order that we first encounter them. Because T is a spanning tree, every vertex is numbered. Finally, we return the cycle obtained by visiting the vertices according to this numbering.

Theorem J.5. *A depth-first ordering of the minimum spanning tree gives a 2-approximation of the shortest traveling salesman tour.*



A minimum spanning tree T, a depth-first traversal of T, and the resulting approximate traveling salesman tour.

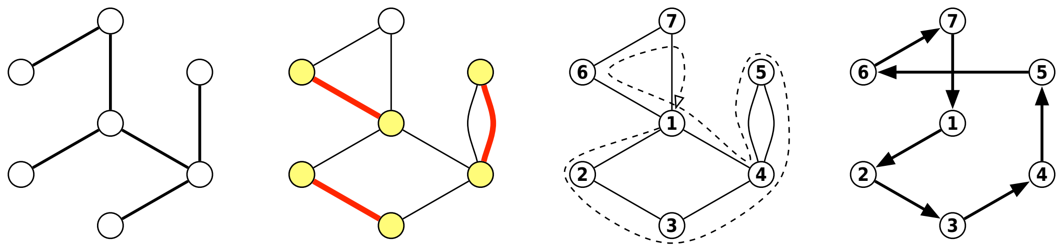
Proof: Let *OPT* denote the cost of the optimal TSP tour, let *MST* denote the total length of the minimum spanning tree, and let A be the length of the tour computed by our approximation algorithm. Consider the 'tour' obtained by walking through the minimum spanning tree in depth-first order. Since this tour traverses every edge in the tree exactly twice, its length is $2 \cdot MST$. The final tour can be obtained from this one by removing duplicate vertices, moving directly from each node to the next *unvisited* node.; the triangle inequality implies that taking these shortcuts cannot make the tour longer. Thus, $A \leq 2 \cdot MST$. On the other hand, if we remove any edge from the optimal tour, we obtain a spanning tree (in fact a spanning *path*) of the graph; thus, $MST \geq OPT$. We conclude that $A \leq 2 \cdot OPT$; our algorithm computes a 2-approximation of the optimal tour. □

We can improve this approximation factor using the following algorithm discovered by Nicos Christofides in 1976. As in the previous algorithm, we start by constructing the minimum spanning tree T. Then let O be the set of vertices with *odd* degree in T; it is an easy exercise (hint, hint) to show that the number of vertices in O is even.

In the next stage of the algorithm, we compute a *minimum-cost perfect matching* M of these odd-degree vertices. A perfect matching is a collection of edges, where each edge has both endpoints in O and each vertex in O is adjacent to exactly one edge; we want the perfect matching of minimum total length. A minimum-cost perfect matching can be computed in polynomial time using maximum-flow techniques, which are described in a different lecture note.

Now consider the multigraph $T \cup M$; any edge in both T and M appears twice in this multigraph. This graph is connected, and every vertex has even degree. Thus, it contains an *Eulerian circuit*: a closed walk that uses every edge exactly once. We can compute such a walk in $O(n)$ time with a simple modification of depth-first search. To obtain the final approximate TSP tour, we number the vertices in the order they first appear in some Eulerian circuit of $T \cup M$, and return the cycle obtained by visiting the vertices according to that numbering.

Theorem J.6. *Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a (3/2)-approximation of the shortest traveling salesman tour.*



A minimum spanning tree T, a minimum-cost perfect matching M of the odd vertices in T, an Eulerian circuit of $T \cup M$, and the resulting approximate traveling salesman tour.

Proof: Let A denote the length of the tour computed by the Christofides heuristic; let OPT denote the length of the optimal tour; let MST denote the total length of the minimum spanning tree; let MOM denote the total length of the minimum odd-vertex matching.

The graph $T \cup M$, and therefore any Euler tour of $T \cup M$, has total length $MST + MOM$. By the triangle inequality, taking a shortcut past a previously visited vertex can only shorten the tour. Thus, $A \leq MST + MOM$.

By the triangle inequality, the optimal tour of the odd-degree vertices of T cannot be longer than OPT. Any cycle passing through of the odd vertices can be partitioned into two perfect matchings, by alternately coloring the edges of the cycle red and green. One of these two matchings has length at most $OPT/2$. On the other hand, both matchings have length at least MOM. Thus, $MOM \leq OPT/2$.

Finally, recall our earlier observation that $MST \leq OPT$.

Putting these three inequalities together, we conclude that $A \leq 3 \cdot OPT/2$, as claimed. □

Four decades after its discovery, Christofedes' algorithm is the best approximation algorithm known for metric TSP, although better approximation algorithms are known for interesting special cases. In particular, there have been several recent breakthroughs for the special case where the underlying metric is determined by shortest-path distances in an *unweighted* graph G; this special case is often called the ***graphic*** traveling salesman problem. The best approximation ratio known for this special case is $7/5$, from a 2012 algorithm of Sebő and Vygen.

J.10 k-center Clustering

The *k-center clustering* problem is defined as follows. We are given a set $P = \{p_1, p_2, \ldots, p_n\}$ of n points in the plane[3] and an integer k. Our goal to find a collection of k circles that collectively enclose all the input points, such that the radius of the largest circle is as

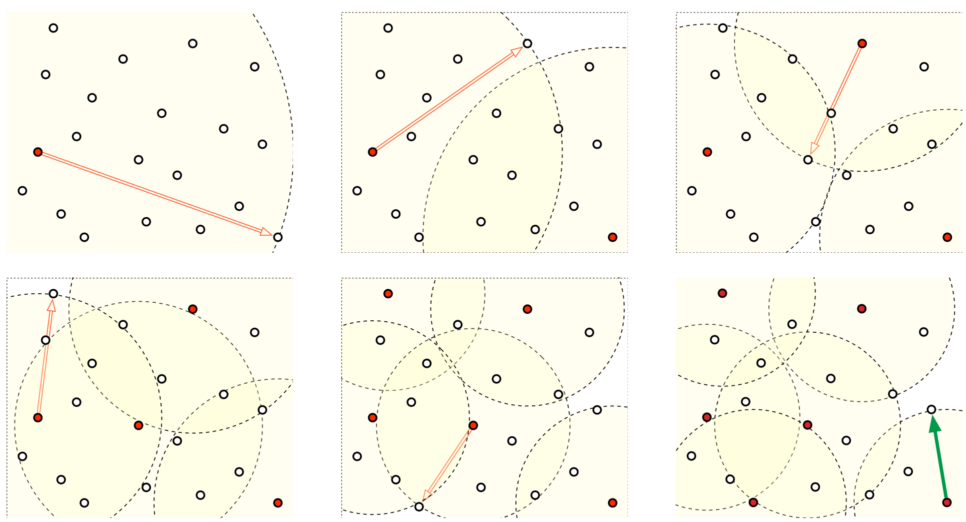
[3]The k-center problem can be defined over *any* metric space, and the approximation analysis in this section holds in any metric space as well. The analysis in the *next* section, however, does require that the points come from the Euclidean plane.

small as possible. More formally, we want to compute a set $C = \{c_1, c_2, \ldots, c_k\}$ of k *center points*, such that the following cost function is minimized:

$$cost(C) := \max_i \min_j |p_i c_j|.$$

Here, $|p_i c_j|$ denotes the Euclidean distance between input point p_i and center point c_j. Intuitively, each input point is assigned to its closest center point; the points assigned to a given center c_j comprise a *cluster*. The distance from c_j to the farthest point in its cluster is the *radius* of that cluster; the cluster is contained in a circle of this radius centered at c_j. The k-center clustering cost $cost(C)$ is precisely the maximum cluster radius.

This problem turns out to be NP-hard, even to approximate within a factor of roughly 1.8. However, there is a natural greedy strategy, first analyzed in 1985 by Teofilo Gonzalez[4], that is guaranteed to produce a clustering whose cost is at most twice optimal. Choose the k center points one at a time, starting with an arbitrary input point as the first center. In each iteration, choose the input point that is farthest from any earlier center point to be the next center point.



The first six iterations of Gonzalez's k-center clustering algorithm.

In the pseudocode below, d_i denotes the current distance from point p_i to its nearest center, and r_j denotes the maximum of all d_i (or in other words, the cluster radius) after the first j centers have been chosen. The algorithm includes an extra iteration to compute the final clustering radius r_k (and the next center c_{k+1}).

[4]Teofilo F. Gonzalez. Clustering to minimize the maximum inter-cluster distance. *Theoretical Computer Science* 38:293-306, 1985.

```
GonzalezKCenter(P, k):
  for i ← 1 to n
      d_i ← ∞
  c_1 ← p_1
  for j ← 1 to k
      r_j ← 0
      for i ← 1 to n
          d_i ← min{d_i, |p_i c_j|}
          if r_j < d_i
              r_j ← d_i;  c_{j+1} ← p_i
  return {c_1, c_2, ..., c_k}
```

GonzalezKCenter clearly runs in $O(nk)$ time. Using more advanced data structures, Tomas Feder and Daniel Greene[5] described an algorithm to compute exactly the same clustering in only $O(n \log k)$ time.

Theorem J.7. *GonzalezKCenter computes a 2-approximation to the optimal k-center clustering.*

Proof: Let OPT denote the optimal k-center clustering radius for P. For any index i, let c_i and r_i denote the ith center point and ith clustering radius computed by GonzalezKCenter.

By construction, each center point c_j has distance at least r_{j-1} from any center point c_i with $i < j$. Moreover, for any $i < j$, we have $r_i \geq r_j$. Thus, $|c_i c_j| \geq r_k$ for all indices i and j.

On the other hand, at least one cluster in the optimal clustering contains at least two of the points $c_1, c_2, \ldots, c_{k+1}$. Thus, by the triangle inequality, we must have $|c_i c_j| \leq 2 \cdot OPT$ for some indices i and j. We conclude that $r_k \leq 2 \cdot OPT$, as claimed. □

♥J.11 Approximation Schemes

With just a little more work, we can compute an arbitrarily close approximation of the optimal k-clustering, using a so-called ***approximation scheme***. An approximation scheme accepts both an instance of the problem and a value $\varepsilon > 0$ as input, and it computes a $(1+\varepsilon)$-approximation of the optimal output for that instance. As I mentioned earlier, computing even a 1.8-approximation is NP-hard, so we cannot expect our approximation scheme to run in polynomial time; nevertheless, at least for small

[5]Tomas Feder* and Daniel H. Greene. Optimal algorithms for approximate clustering. *Proc. 20th STOC*, 1988. Unlike Gonzalez's algorithm, Feder and Greene's faster algorithm does not work over arbitrary metric spaces; it requires that the input points come from some $\mathbb{R}^d$ and that distances are measured in some L_p metric. The time analysis also assumes that the distance between any two points can be computed in $O(1)$ time.

values of k, the approximation scheme will be considerably more efficient than any exact algorithm.

Our approximation scheme works in three phases:

1. Compute a 2-approximate clustering of the input set P using GonzalezKCenter. Let r be the cost of this clustering.
2. Create a regular grid of squares of width $\delta = \varepsilon r/2\sqrt{2}$. Let Q be a subset of P containing one point from each non-empty cell of this grid.
3. Compute an *optimal* set of k centers for Q. Return these k centers as the approximate k-center clustering for P.

The first phase requires $O(nk)$ time. By our earlier analysis, we have $r^* \le r \le 2r^*$, where r^* is the optimal k-center clustering cost for P.

The second phase can be implemented in $O(n)$ time using a hash table, or in $O(n \log n)$ time by standard sorting, by associating approximate coordinates $(\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor)$ to each point $(x, y) \in P$ and removing duplicates. The key observation is that the resulting point set Q is significantly smaller than P. We know P can be covered by k balls of radius r^*, each of which touches $O(r^*/\delta^2) = O(1/\varepsilon^2)$ grid cells. It follows that $|Q| = O(k/\varepsilon^2)$.

Let $T(n, k)$ be the running time of an *exact* k-center clustering algorithm, given n points as input. If this were a computational geometry class, we might see a "brute force" algorithm that runs in time $T(n, k) = O(n^{k+2})$; the fastest algorithm currently known[6] runs in time $T(n, k) = n^{O(\sqrt{k})}$. If we use this algorithm, our third phase requires $(k/\varepsilon^2)^{O(\sqrt{k})}$ time.

It remains to show that the optimal clustering for Q implies a $(1 + \varepsilon)$-approximation of the optimal clustering for P. Suppose the optimal clustering of Q consists of k balls $B_1, B_2, \dots, B_k$, each of radius $\tilde{r}$. Clearly $\tilde{r} \le r^*$, since any set of k balls that cover P also cover any subset of P. Each point in $P \setminus Q$ shares a grid cell with some point in Q, and therefore is within distance $\delta\sqrt{2}$ of some point in Q. Thus, if we increase the radius of each ball B_i by $\delta\sqrt{2}$, the expanded balls must contain every point in P. We conclude that the optimal centers for Q gives us a k-center clustering for P of cost at most $r^* + \delta\sqrt{2} \le r^* + \varepsilon r/2 \le r^* + \varepsilon r^* = (1 + \varepsilon)r^*$.

The total running time of the approximation scheme is $O(nk + (k/\varepsilon^2)^{O(\sqrt{k})})$. This is still exponential in the input size if k is large (say $\sqrt{n}$ or $n/100$), but if k and ε are fixed constants, the running time is linear in the number of input points.

♥J.12 An FPTAS for Subset Sum

An approximation scheme whose running time, for any fixed ε, is polynomial in n is called a ***polynomial-time approximation scheme*** or ***PTAS*** (usually pronounced *"pee*

[6] R. Z. Hwang, R. C. T. Lee, and R. C. Chan. The slab dividing approach to solve the Euclidean p-center problem. *Algorithmica* 9(1):1–22, 1993.

taz"). If in addition the running time depends only polynomially on ε, the algorithm is called a ***fully*** polynomial-time approximation scheme or *FPTAS* (usually pronounced "*eff pee taz*"). For example, an approximation scheme with running time $O(n^2/\varepsilon^2)$ is an FPTAS; an approximation scheme with running time $O(n^{1/\varepsilon^6})$ is a PTAS but not an FPTAS; and our approximation scheme for k-center clustering is not a PTAS.

The last problem we'll consider is the SubsetSum problem: Given a set X containing n positive integers and a target integer t, determine whether X has a subset whose elements sum to t. The lecture notes on NP-completeness include a proof that SubsetSum is NP-hard. As stated, this problem doesn't allow any sort of approximation—the answer is either True or False.[7] So we will consider a related optimization problem instead: Given set X and integer t, find the subset of X whose sum is as large as possible but no larger than t.

We have already seen a dynamic programming algorithm to solve the decision version SubsetSum in time $O(nt)$; a similar algorithm solves the optimization version in the same time bound. Here is a different algorithm, whose running time does not depend on t:

```
SubsetSum(X[1..n], t):
  S_0 ← {0}
  for i ← 1 to n
      S_i ← S_{i−1} ∪ (S_{i−1} + X[i])
      remove all elements of S_i bigger than t
  return max S_n
```

Here $S_{i-1} + X[i]$ denotes the set $\{s + X[i] \mid s \in S_{i-1}\}$. If we store each S_i in a sorted array, the ith iteration of the for-loop requires time $O(|S_{i-1}|)$. Each set S_i contains all possible subset sums for the first i elements of X; thus, S_i has at most 2^i elements. On the other hand, since every element of S_i is an integer between 0 and t, we also have $|S_i| \le t+1$. It follows that the total running time of this algorithm is $\sum_{i=1}^{n} O(|S_{i-1}|) = O(\min\{2^n, nt\})$.

Of course, this is only an estimate of worst-case behavior. If several subsets of X have the same sum, the sets S_i will have fewer elements, and the algorithm will be faster. The key idea for finding an approximate solution quickly is to 'merge' nearby elements of S_i—if two subset sums are nearly equal, ignore one of them. On the one hand, merging similar subset sums will introduce some error into the output, but hopefully not too much. On the other hand, by reducing the size of the set of sums we need to maintain, we will make the algorithm faster, hopefully significantly so.

Here is our approximation algorithm. We make only two changes to the exact algorithm: an initial sorting phase and an extra Filtering step inside the main loop.

[7] Do, or do not. There is no 'try'. (Are old one thousand when years you, alphabetical also in order talk will you.)

```
FILTER(Z[1..k], δ):
  SORT(Z)
  j ← 1
  Y[j] ← Z[i]
  for i ← 2 to k
      if Z[i] > (1+δ)·Y[j]
          j ← j+1
          Y[j] ← Z[i]
  return Y[1..j]
```

```
APPROXSUBSETSUM(X[1..n], k, ε):
  SORT(X)
  R_0 ← {0}
  for i ← 1 to n
      R_i ← R_{i−1} ∪ (R_{i−1} + X[i])
      R_i ← FILTER(R_i, ε/2n)
      remove all elements of R_i bigger than t
  return max R_n
```

Theorem J.8. *APPROXSUBSETSUM returns a* $(1+\varepsilon)$*-approximation of the optimal subset sum, given any* ε *such that* $0 < \varepsilon \leq 1$.

Proof: The theorem follows from the following claim, which we prove by induction:

$$\boxed{\text{For any element } s \in S_i \text{, there is an element } r \in R_i \text{ such that } r \leq s \leq r \cdot (1+\varepsilon n/2)^i.}$$

The claim is trivial for $i = 0$. Let s be an arbitrary element of S_i, for some $i > 0$. There are two cases to consider: either $x \in S_{i-1}$, or $x \in S_{i-1} + x_i$.

(1) Suppose $s \in S_{i-1}$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s \leq r' \cdot (1+\varepsilon n/2)^{i-1}$. If $r' \in R_i$, the claim obviously holds. On the other hand, if $r' \notin R_i$, there must be an element $r \in R_i$ such that $r < r' \leq r(1+\varepsilon n/2)$, which implies that

$$r < r' \leq s \leq r' \cdot (1+\varepsilon n/2)^{i-1} \leq r \cdot (1+\varepsilon n/2)^i,$$

so the claim holds.

(2) Suppose $s \in S_{i-1} + x_i$. By the inductive hypothesis, there is an element $r' \in R_{i-1}$ such that $r' \leq s - x_i \leq r' \cdot (1+\varepsilon n/2)^{i-1}$. If $r' + x_i \in R_i$, the claim obviously holds. On the other hand, if $r' + x_i \notin R_i$, there must be an element $r \in R_i$ such that $r < r' + x_i \leq r(1+\varepsilon n/2)$, which implies that

$$\begin{aligned} r < r' + x_i \leq s &\leq r' \cdot (1+\varepsilon n/2)^{i-1} + x_i \\ &\leq (r - x_i) \cdot (1+\varepsilon n/2)^i + x_i \\ &\leq r \cdot (1+\varepsilon n/2)^i - x_i \cdot ((1+\varepsilon n/2)^i - 1) \\ &\leq r \cdot (1+\varepsilon n/2)^i. \end{aligned}$$

so the claim holds.

Now let $s^* = \max S_n$ and $r^* = \max R_n$. Clearly $r^* \leq s^*$, since $R_n \subseteq S_n$. Our claim implies that there is some $r \in R_n$ such that $s^* \leq r \cdot (1+\varepsilon/2n)^n$. But r cannot be bigger than r^*, so $s^* \leq r^* \cdot (1+\varepsilon/2n)^n$. The inequalities $e^x \geq 1 + x$ for all x, and $e^x \leq 2x + 1$ for all $0 \leq x \leq 1$, imply that $(1+\varepsilon/2n)^n \leq e^{\varepsilon/2} \leq 1 + \varepsilon$. □

Theorem J.9. *ApproxSubsetSum runs in $O((n^3 \log n)/\varepsilon)$ time.*

Proof: Assuming we keep each set R_i in a sorted array, we can merge the two sorted arrays R_{i-1} and $R_{i-1} + x_i$ in $O(|R_{i-1}|)$ time. FilterIn R_i and removing elements larger than t also requires only $O(|R_{i-1}|)$ time. Thus, the overall running time of our algorithm is $O(\sum_i |R_i|)$; to express this in terms of n and ε, we need to prove an upper bound on the size of each set R_i.

Let $\delta = \varepsilon/2n$. Because we consider the elements of X in increasing order, every element of R_i is between 0 and $i \cdot x_i$. In particular, every element of $R_{i-1} + x_i$ is between x_i and $i \cdot x_i$. After Filtering, at most one element $r \in R_i$ lies in the range $(1+\delta)^k \leq r < (1+\delta)^{k+1}$, for any k. Thus, at most $\lceil \log_{1+\delta} i \rceil$ elements of $R_{i-1} + x_i$ survive the call to Filter. It follows that

$$\begin{aligned}
|R_i| &= |R_{i-1}| + \left\lceil \frac{\log i}{\log(1+\delta)} \right\rceil \\
&\leq |R_{i-1}| + \left\lceil \frac{\log n}{\log(1+\delta)} \right\rceil && [i \leq n] \\
&\leq |R_{i-1}| + \left\lceil \frac{2 \ln n}{\delta} \right\rceil && [e^x \leq 1 + 2x \text{ for all } 0 \leq x \leq 1] \\
&\leq |R_{i-1}| + \left\lceil \frac{n \ln n}{\varepsilon} \right\rceil && [\delta = \varepsilon/2n]
\end{aligned}$$

Unrolling this recurrence into a summation gives us the upper bound $|R_i| \leq i \cdot \lceil (n \ln n)/\varepsilon \rceil = O((n^2 \log n)/\varepsilon)$.

We conclude that the overall running time of ApproxSubsetSum is $O((n^3 \log n)/\varepsilon)$, as claimed. □

Exercises

1. (a) Prove that for any set of jobs, the makespan of the greedy assignment is at most $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.

   (b) Describe a set of jobs such that the makespan of the greedy assignment is exactly $(2 - 1/m)$ times the makespan of the optimal assignment, where m is the number of machines.

   (c) Describe an efficient algorithm to solve the minimum makespan scheduling problem *exactly* if every processing time $T[i]$ is a power of two.

2. (a) Find the smallest graph (minimum number of edges) for which GreedyVertex-Cover does not return the smallest vertex cover.

   (b) For any integer n, describe an n-vertex graph for which GreedyVertexCover returns a vertex cover of size $OPT \cdot \Omega(\log n)$.

3. (a) Find the smallest graph (minimum number of edges) for which DUMBVERTEXCOVER does not return the smallest vertex cover.
   (b) Describe an infinite family of graphs for which DUMBVERTEXCOVER returns a vertex cover of size $2 \cdot OPT$.

4. Let R be a set of rectangles in the plane, with horizontal and vertical edges. A *stabbing set* for R is a set S of points such that every rectangle in R contains at least one point in S. The *rectangle stabbing problem* asks for the smallest stabbing set of a given set of rectangles.
   (a) Prove that the rectangle stabbing problem is NP-hard.
   (b) Describe and analyze an efficient approximation algorithm for the rectangle stabbing problem. What is the approximation ratio of your algorithm?

5. Consider the following heuristic for constructing a vertex cover of a connected graph G: return the set of non-leaf nodes in any depth-first spanning tree of G.
   (a) Prove that this heuristic returns a vertex cover of G.
   (b) Prove that this heuristic returns a 2-approximation to the minimum vertex cover of G.
   (c) Describe an infinite family of graphs for which this heuristic returns a vertex cover of size $2 \cdot OPT$.

6. Suppose we have n pieces of candy with weights $W[1..n]$ (in ounces) that we want to load into boxes. Our goal is to load the candy into as many boxes as possible, so that each box contains at least L ounces of candy. Describe an efficient 2-approximation algorithm for this problem. Prove that the approximation ratio of your algorithm is 2. *[Hint: First consider the case where every piece of candy weighs less than L ounces.]*

7. Suppose we want to route a set of N calls on a telecommunications network that consists of a cycle of n nodes, indexed in cyclic order from 0 to $n-1$. Each call can be routed either clockwise or counterclockwise around the cycle from its source node to its destination node. Our goal is to route the calls so as to minimize the overall load on the network. The load L_i on any edge $(i, (i+1) \bmod n)$ is the number of calls routed through that edge, and the overall load is $\max_i L_i$. Describe and analyze an efficient 2-approximation algorithm for this problem.

8. The *linear arrangement* problem asks, given an n-vertex directed graph as input, for an ordering $v_1, v_2, \ldots, v_n$ of the vertices that maximizes the number of forward edges: directed edges $v_i \rightarrow v_j$ such that $i < j$. Describe and analyze an efficient 2-approximation algorithm for this problem. (Solving this problem exactly is NP-hard.)

9. A *three-dimensional matching* in an undirected graph G is a collection of vertex-disjoint triangles (cycles of length 3) in G. A three-dimensional matching is *maximal* if it is not a proper subgraph of a larger three-dimensional matching in the same graph.

   (a) Let M and M' be two arbitrary maximal three-dimensional matchings in the same underlying graph G. Prove that $|M| \le 3 \cdot |M'|$.

   (b) Finding the *largest* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   (c) Finding the *smallest maximal* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

10. Consider the following optimization version of the PARTITION problem. Given a set X of positive integers, our task is to partition X into disjoint subsets A and B such that $\max\left\{\sum A, \sum B\right\}$ is as small as possible. Solving this problem exactly is clearly NP-hard.

    (a) Prove that the following algorithm yields a (3/2)-approximation.

```
GreedyPartition(X[1..n]):
  a ← 0
  b ← 0
  for i ← 1 to n
      if a < b
          a ← a + X[i]
      else
          b ← b + X[i]
  return max{a, b}
```

    (b) Prove that the approximation ratio 3/2 cannot be improved, even if the input array X is sorted in *non-decreasing* order. In other words, give an example of a sorted array X where the output of GREEDYPARTITION is 50% larger than the cost of the optimal partition.

    (c) Prove that if the array X is sorted in *non-increasing* order, then GREEDYPARTITION achieves an approximation ratio of at most 4/3.

    ♥(d) Prove that if the array X is sorted in *non-increasing* order, then GREEDYPARTITION achieves an approximation ratio of exactly 7/6.

11. The *chromatic number* $\chi(G)$ of a graph G is the minimum number of colors required to color the vertices of the graph, so that every edge has endpoints with different colors. Computing the chromatic number exactly is NP-hard.

    Prove that the following problem is also NP-hard: Given an n-vertex graph G, return any integer between $\chi(G)$ and $\chi(G)+31337$. *[Note: This does not contradict the possibility of a constant **factor** approximation algorithm.]*

12. Let $G = (V, E)$ be an undirected graph, each of whose vertices is colored either red, green, or blue. An edge in G is *boring* if its endpoints have the same color, and *interesting* if its endpoints have different colors. The *most interesting 3-coloring* is the 3-coloring with the maximum number of interesting edges, or equivalently, with the fewest boring edges. Computing the most interesting 3-coloring is NP-hard, because the standard 3-coloring problem is a special case.

    (a) Let $zzz(G)$ denote the number of boring edges in the most interesting 3-coloring of a graph G. Prove that it is NP-hard to approximate $zzz(G)$ within a factor of $10^{10^{100}}$.

    (b) Let $wow(G)$ denote the number of interesting edges in the most interesting 3-coloring of G. Suppose we assign each vertex in G a *random* color from the set $\{\text{red}, \text{green}, \text{blue}\}$. Prove that the expected number of interesting edges is at least $\frac{2}{3} wow(G)$.

    (c) Prove that with high probability, the expected number of interesting edges is at least $\frac{1}{2} wow(G)$. *[Hint: Use Chebyshev's inequality. But wait... How do we know that we* ***can*** *use Chebyshev's inequality?]*

13. The KNAPSACK problem can be defined as follows. We are given a finite set X, each of whose elements has a non-negative *size* and a non-negative *value*, along with an integer *capacity* c. Our task is to determine the maximum total value among all subsets of X whose total size is at most c. This problem is NP-hard. Specifically, the optimization version of SUBSETSUM is a special case, where each element's value is equal to its size.

    Determine the approximation ratio of the following polynomial-time approximation algorithm. Prove your answer is correct.

```
APPROXKNAPSACK(X, c):
  return max{GREEDYKNAPSACK(X, c), PICKBESTONE(X, c)}
```

```
GREEDYKNAPSACK(X, c):
  Sort X in decreasing order by the ratio value/size
  S ← 0; V ← 0
  for i ← 1 to n
      if S + size(x_i) > c
          return V
      S ← S + size(x_i)
      V ← V + value(x_i)
  return V
```

```
PICKBESTONE(X, c):
  Sort X in increasing order by size
  V ← 0
  for i ← 1 to n
      if size(x_i) > c
          return V
      if value(x_i) > V
          V ← value(x_i)
  return V
```

14. In the *bin packing* problem, we are given a set of n items, each with weight between 0 and 1, and we are asked to load the items into as few bins as possible, such that the total weight in each bin is at most 1. It's not hard to show that this problem is

NP-hard; this question asks you to analyze a few common approximation algorithms. In each case, the input is an array $W[1..n]$ of weights, and the output is the number of bins used.

```
NextFit(W[1..n]):
  b ← 0
  Total[0] ← ∞
  for i ← 1 to n
      if Total[b] + W[i] > 1
          b ← b + 1
          Total[b] ← W[i]
      else
          Total[b] ← Total[b] + W[i]
  return b
```

```
FirstFit(W[1..n]):
  b ← 0
  for i ← 1 to n
      j ← 1; found ← False
      while j ≤ b and found = False
          if Total[j] + W[i] ≤ 1
              Total[j] ← Total[j] + W[i]
              found ← True
          j ← j + 1
      if found = False
          b ← b + 1
          Total[b] = W[i]
  return b
```

(a) Prove that NextFit uses at most twice the optimal number of bins.

(b) Prove that FirstFit uses at most twice the optimal number of bins.

♥(c) Prove that if the weight array W is initially sorted in decreasing order, then FirstFit uses at most $(4 \cdot OPT + 1)/3$ bins, where OPT is the optimal number of bins. The following facts may be useful (but you need to prove them if your proof uses them):

- In the packing computed by FirstFit, every item with weight more than $1/3$ is placed in one of the first OPT bins.
- FirstFit places at most $OPT - 1$ items outside the first OPT bins.

15. Given a graph G with edge weights and an integer k, suppose we wish to partition the vertices of G into k subsets $S_1, S_2, \ldots, S_k$ so that the sum of the weights of the edges that cross the partition (that is, have endpoints in different subsets) is as large as possible.

    (a) Describe an efficient $(1 - 1/k)$-approximation algorithm for this problem.

    (b) Now suppose we wish to minimize the sum of the weights of edges that do *not* cross the partition. What approximation ratio does your algorithm from part (a) achieve for the new problem? Justify your answer.

16. The lecture notes describe a (3/2)-approximation algorithm for the metric traveling salesman problem, which asks for the minimum-cost Hamiltonian cycle in a graph whose edge weights satisfy the triangle inequality. Here, we consider computing minimum-cost Hamiltonian *paths*. Our input consists of a graph G whose edges have

weights that satisfy the triangle inequality. Depending upon the problem, we are also given zero, one, or two endpoints.

(a) If our input includes zero endpoints, describe a (3/2)-approximation to the problem of computing a minimum cost Hamiltonian path.

(b) If our input includes one endpoint u, describe a (3/2)-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u.

♥(c) If our input includes two endpoints u and v, describe a (5/3)-approximation to the problem of computing a minimum cost Hamiltonian path that starts at u and ends at v.

17. Suppose we are given a collection of n jobs to execute on a machine containing a row of p identical processors. The ***parallel scheduling problem*** asks us to schedule these jobs on these processors, given two arrays $T[1..n]$ and $P[1..n]$ as input, subject to the following constraints:

    - When the ith job is executed, it occupies a contiguous interval of $P[i]$ processors for exactly $T[i]$ seconds.
    - No processor works on more than one job at a time.

    A valid schedule specifies a non-negative starting time and an interval of processors for each job that meets these constraints. Our goal is to compute a valid schedule with the smallest possible *makespan*, which is the earliest time when all jobs are complete.

    (a) Prove that the parallel scheduling problem is NP-hard.

    (b) Describe a polynomial-time algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs. That is, if the minimum makespan is M, your algorithm should compute a schedule with make-span at most $3M$. You may assume that p is a power of 2. *[Hint: p is a power of 2.]*

    (c) Describe an algorithm that computes a 3-approximation of the minimum makespan of a given set of jobs in $O(n \log n)$ time. Again, you may assume that n is a power of 2.

18. Consider the greedy algorithm for the metric traveling salesman problem: start at an arbitrary vertex u, and at each step, travel to the closest unvisited vertex.

    (a) Prove that the approximation ratio for this algorithm is $O(\log n)$, where n is the number of vertices. *[Hint: Argue that the kth least expensive edge in the tour output by the greedy algorithm has weight at most* $\mathrm{OPT}/(n-k+1)$*; try $k=1$ and $k=2$ first.]*

    ♥(b) Prove that the approximation ratio for this algorithm is $\Omega(\log n)$. That is, describe an infinite family of weighted graphs such that the greedy algorithm returns a

Hamiltonian cycle whose weight is $\Omega(\log n)$ times the weight of the optimal TSP tour.