

7 Approximation Algorithms (October 4, 6, and 11)

7.1 Load Balancing

On the future smash hit reality-TV game show *Grunt Work*, scheduled to air Thursday nights at 3am (2am Central) on ESPN5, 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 over your head and sing The Star-Spangled Banner for two minutes”, or “Listen to this algorithms lecture for 75 minutes”. The directors of the show want you to assign each task to one of the contestants, so that the last task is completed in the shortest amount of time. 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 *television!*”

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 $T[j]$ represents the running time of job j . Our assignment will be specified 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:

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

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 with the earliest finishing time.

```

GREEDYLOADBALANCE( $T[1..n], m$ ):
  for  $i \leftarrow 1$  to  $m$ 
     $Total[i] \leftarrow 0$ 
  for  $j \leftarrow 1$  to  $n$ 
     $min \leftarrow 1$ 
    for  $i \leftarrow 2$  to  $m$ 
      if  $Total[i] < Total[min]$ 
         $min \leftarrow i$ 
     $A[j] \leftarrow min$ 
     $Total[min] \leftarrow Total[min] + T[j]$ 
  return  $A[1..m]$ 

```

Theorem: *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

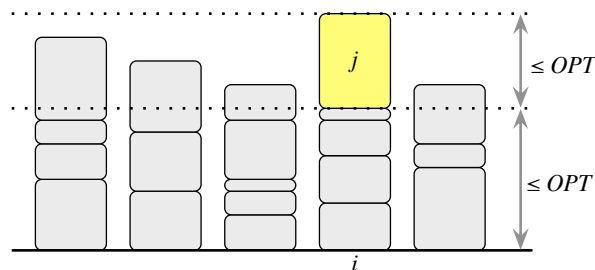
makespan of any assignment is at least the total duration of all the jobs divided by the number of machines.

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

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 job i . Our first trivial observation implies that $T[j] \leq OPT$. To finish the proof, we must show that $Total[i] - T[j] \leq OPT$. Job j was assigned to machine i because it had the smallest finishing time, so $Total[i] - T[j] \leq Total[k]$ for all k . (Some of the 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,

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

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



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. In particular, there is *no* online load balancing algorithm that achieves an approximation factor better than 2. The study of online algorithms could easily fill an entire one-semester course (alas, not this one).

Even though GREEDYLOADBALANCE is the best possible *online* load balancing algorithm, 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: *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 processor, we must have $j \geq m + 1$. As in the previous proof, we know that $Total[i] - T[j] \leq OPT$.

In the optimal assignment, at least two of the first $m + 1$ jobs, say jobs k and ℓ , must be scheduled on the same processor. Thus, $T[k] + T[\ell] \leq OPT$. Since $\max\{k, \ell\} \leq m + 1 \leq j$, and the jobs are sorted in decreasing order by direction, we have

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

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

7.2 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 α -**approximation algorithm** if

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

for all inputs X . Generally, 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. The approximation factor α may be either a constant or a function of the input size.

7.3 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 heuristic¹ for computing a small vertex cover: mark the vertex with the largest degree, remove all the edges incident to that vertex, and recurse.

```

GREEDYVERTEXCOVER( $G$ ):
   $C \leftarrow \emptyset$ 
  while  $G$  has at least one edge
     $v \leftarrow$  vertex in  $G$  with maximum degree
     $G \leftarrow G \setminus v$ 
     $C \leftarrow C \cup v$ 
  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: 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:

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

```

GREEDYVERTEXCOVER( $G$ ):
   $C \leftarrow \emptyset$ 
   $G_0 \leftarrow G$ 
   $i \leftarrow 0$ 
  while  $G_i$  has at least one edge
     $i \leftarrow i + 1$ 
     $v_i \leftarrow$  vertex in  $G_{i-1}$  with maximum degree
     $d_i \leftarrow \deg_{G_{i-1}}(v_i)$ 
     $G_i \leftarrow G_{i-1} \setminus v_i$ 
     $C \leftarrow C \cup v_i$ 
  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 $OPT \lg|G| \leq 2OPT \lg n$ iterations, all the edges of G have been removed, and the algorithm terminates. We conclude that GREEDYVERTEXCOVER computes a vertex cover of size $O(OPT \log n)$. \square

7.4 Set Cover and Hitting Set

The greedy algorithm for vertex cover can be applied almost immediately to two more general problems: *set cover* and *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. GREEDYSETCOVER finds a set cover whose size is at most $O(\log|\mathcal{F}|)$ times the size of smallest set cover. GREEDYHITTINGSET finds a hitting set whose size is at most $O(\log|X|)$ times the size of the smallest hitting set.

```

GREEDYSETCOVER( $X, \mathcal{F}$ ):
 $\mathcal{C} \leftarrow \emptyset$ 
while  $X \neq \emptyset$ 
     $S \leftarrow \arg \max_{S \in \mathcal{F}} |S \cap X|$ 
     $X \leftarrow X \setminus S$ 
     $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$ 
return  $\mathcal{C}$ 

```

```

GREEDYHITTINGSET( $X, \mathcal{F}$ ):
 $H \leftarrow \emptyset$ 
while  $\mathcal{F} \neq \emptyset$ 
     $x \leftarrow \arg \max_{x \in X} |\{S \in \mathcal{F} \mid x \in S\}|$ 
     $\mathcal{F} \leftarrow \mathcal{F} \setminus \{S \in \mathcal{F} \mid x \in S\}$ 
     $H \leftarrow H \cup \{x\}$ 
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^* :

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

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 473G this semester. Then X^* has 26 elements, each containing the subset of CS 473G students whose last name contains a particular letter of the alphabet. For example, m^* is the set of students whose last names contain the letter m .

There is a direct one-to-one correspondence 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 a set cover for the dual set system (\mathcal{F}, X^*) .

7.5 Vertex Cover, Again

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

```

DUMBVERTEXCOVER( $G$ ):
 $C \leftarrow \emptyset$ 
while  $G$  has at least one edge
     $(u, v) \leftarrow$  any edge in  $G$ 
     $G \leftarrow G \setminus \{u, v\}$ 
     $C \leftarrow C \cup u, v$ 
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!

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

7.6 Traveling Salesman: The Bad News

The *traveling salesman problem*² 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:

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

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: *For any function $f(n)$ that can be computed in time $O(\text{poly}(n))$, there is no polynomial-time $f(n)$ -approximation algorithm for the traveling salesman problem on general weighted graphs, unless $P=NP$.*

7.7 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*:

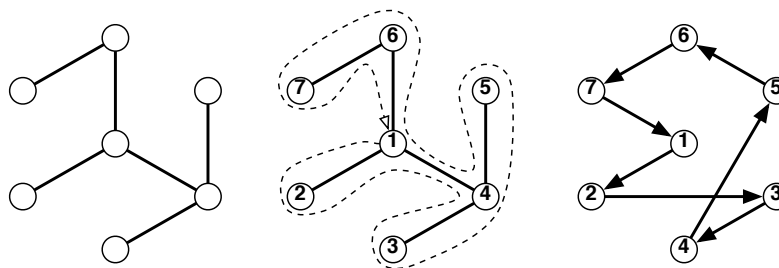
$$\ell(u, w) \leq \ell(u, v) + \ell(v, w) \text{ for any vertices } u, v, w.$$

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

²This is sometimes bowdlerized into the traveling salesperson problem. Sorry, no. Who ever heard of a traveling salesperson sleeping with the farmer's child?

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.



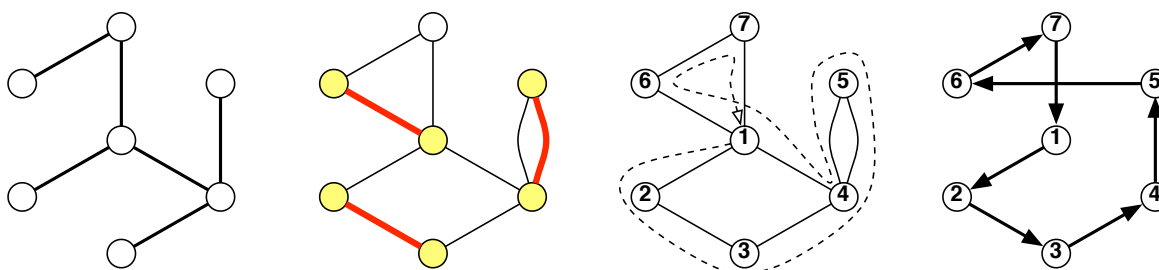
A minimum spanning tree T , a depth-first traversal of T , and the resulting approximate traveling salesman tour.

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 traveling salesman tour.

We can improve the approximation factor even further 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. Later in the semester, we will see an algorithm to compute M in polynomial time.

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.



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.

Theorem: Given a weighted graph that obeys the triangle inequality, the Christofides heuristic computes a $(3/2)$ -approximation of the minimum 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. \square