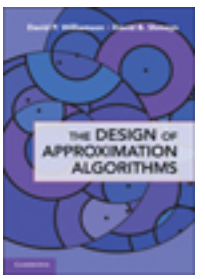

The Design of Approximation Algorithms

David P. Williamson, David B. Shmoys

Book DOI: http://dx.doi.org/10.1017/CBO9780511921735

Online ISBN: 9780511921735

Hardback ISBN: 9780521195270

Chapter

2 - Greedy Algorithms and Local Search pp. 27-56

Chapter DOI: http://dx.doi.org/10.1017/CBO9780511921735.003

Cambridge University Press

CHAPTER 2

Greedy Algorithms and Local Search

In this chapter, we will consider two standard and related techniques for designing algorithms and heuristics, namely, *greedy algorithms* and *local search algorithms*. Both algorithms work by making a sequence of decisions that optimize some local choice, though these local choices might not lead to the best overall solution.

In a greedy algorithm, a solution is constructed step by step, and at each step of the algorithm the next part of the solution is constructed by making some decision that is locally the best possible. In Section 1.6, we gave an example of a greedy algorithm for the set cover problem that constructs a set cover by repeatedly choosing the set that minimizes the ratio of its weight to the number of currently uncovered elements it contains.

A local search algorithm starts with an arbitrary feasible solution to the problem, and then checks if some small, local change to the solution results in an improved objective function. If so, the change is made. When no further change can be made, we have a *locally optimal solution*, and it is sometimes possible to prove that such locally optimal solutions have value close to that of the optimal solution. Unlike other approximation algorithm design techniques, the most straightforward implementation of a local search algorithm typically does not run in polynomial time. The algorithm usually requires some restriction to the local changes allowed in order to ensure that enough progress is made during each improving step so that a locally optimal solution is found in polynomial time.

Thus, while both types of algorithm optimize local choices, greedy algorithms are typically *primal infeasible* algorithms: they construct a solution to the problem during the course of the algorithm. Local search algorithms are *primal feasible* algorithms: they always maintain a feasible solution to the problem and modify it during the course of the algorithm.

Both greedy algorithms and local search algorithms are extremely popular choices for heuristics for NP-hard problems. They are typically easy to implement and have good running times in practice. In this chapter, we will consider greedy and local search algorithms for scheduling problems, clustering problems, and others, including the most famous problem in combinatorial optimization, the traveling salesman problem.

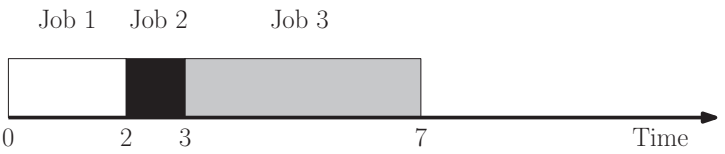

Figure 2.1. An instance of a schedule for the one-machine scheduling problem in which $p_1 = 2, r_1 = 0, p_2 = 1, r_2 = 2, p_3 = 4, r_3 = 1$. In this schedule, $C_1 = 2, C_2 = 3$, and $C_3 = 7$. If the deadlines for the jobs are such that $d_1 = -1$, $d_2 = 1$, and $d_3 = 10$, then $L_1 = 2 - (-1) = 3$, $L_2 = 3 - 1 = 2$, and $L_3 = 7 - 10 = -3$, so that $L_{\max} = L_1 = 3$.

Because greedy and local search algorithms are natural choices for heuristics, some of these algorithms were among the very first approximation algorithms devised; in particular, the greedy algorithm for the parallel machine scheduling problem in Section 2.3 and the greedy algorithm for edge coloring in Section 2.7 were both given and analyzed in the 1960s, before the concept of NP-completeness was invented.

2.1 Scheduling Jobs with Deadlines on a Single Machine

One of the most common types of problems in combinatorial optimization is that of creating a schedule. We are given some type of work that must be done, and some resources to do the work, and from this we must create a schedule to complete the work that optimizes some objective; perhaps we want to finish all the work as soon as possible, or perhaps we want to make sure that the average time at which we complete the various pieces of work is as small as possible. We will often consider the problem of scheduling jobs (the work) on machines (the resources). We start this chapter by considering one of the simplest possible versions of this problem.

Suppose that there are n jobs to be scheduled on a single machine, where the machine can process at most one job at a time, and must process a job until its completion once it has begun processing; suppose that each job j must be processed for a specified p_j units of time, where the processing of job j may begin no earlier than a specified *release date* r_j, $j = 1, \ldots, n$. We assume that the schedule starts at time 0, and each release date is nonnegative. Furthermore, assume that each job j has a specified due date d_j, and if we complete its processing at time C_j, then its *lateness* L_j is equal to $C_j - d_j$; we are interested in scheduling the jobs so as to minimize the maximum lateness, $L_{\max} = \max_{j=1,\ldots,n} L_j$. A sample instance of this problem is shown in Figure 2.1.

Unfortunately, this problem is NP-hard, and in fact, even deciding if there is a schedule for which $L_{\max} \leq 0$ (i.e., deciding if all jobs can be completed by their due date) is strongly NP-hard (the reader unfamiliar with strong NP-hardness can consult Appendix B). Of course, this is a problem that we often encounter in everyday life, and many of us schedule our lives with the following simple greedy heuristic: focus on the task with the earliest due date. We will show that in certain circumstances this is a provably good thing to do. However, we first argue that as stated, this optimization problem is not particularly amenable to obtaining near-optimal solutions. If there were a ρ-approximation algorithm, then for any input with optimal value 0, the algorithm

must still find a schedule of objective function value at most $\rho \cdot 0 = 0$, and hence (given the NP-hardness result stated above) this would imply that P = NP. (There is the further complication of what to do if the objective function of the optimal solution is negative!) One easy workaround to this is to assume that all due dates are negative, which implies that the optimal value is always positive. We shall give a 2-approximation algorithm for this special case.

We first provide a good lower bound on the optimal value for this scheduling problem. Let S denote a subset of jobs, and let $r(S) = \min_{j \in S} r_j$, $p(S) = \sum_{j \in S} p_j$, and $d(S) = \max_{j \in S} d_j$. Let $L^*_{\max}$ denote the optimal value.

Lemma 2.1. *For each subset S of jobs,*

$$L^*_{\max} \geq r(S) + p(S) - d(S).$$

Proof. Consider the optimal schedule, and view this simply as a schedule for the jobs in the subset S. Let job j be the last job in S to be processed. Since none of the jobs in S can be processed before $r(S)$, and in total they require $p(S)$ time units of processing, it follows that job j cannot complete any earlier than time $r(S) + p(S)$. The due date of job j is $d(S)$ or earlier, and so the lateness of job j in this schedule is at least $r(S) + p(S) - d(S)$; hence, $L^*_{\max} \geq r(S) + p(S) - d(S)$. □

A job j is *available* at time t if its release date $r_j \leq t$. We consider the following natural algorithm: at each moment that the machine is idle, start processing next an available job with the earliest due date. This is known as the *earliest due date (EDD) rule*.

Theorem 2.2. *The EDD rule is a 2-approximation algorithm for the problem of minimizing the maximum lateness on a single machine subject to release dates with negative due dates.*

Proof. Consider the schedule produced by the EDD rule, and let job j be a job of maximum lateness in this schedule; that is, $L_{\max} = C_j - d_j$. Focus on the time C_j in this schedule; find the earliest point in time $t \leq C_j$ such that the machine was processing without any idle time for the entire period $[t, C_j)$. Several jobs may be processed in this time interval; we require only that the machine not be idle for some interval of positive length within it. Let S be the set of jobs that are processed in the interval $[t, C_j)$. By our choice of t, we know that just prior to t, none of these jobs were available (and clearly at least one job in S is available at time t); hence, $r(S) = t$. Furthermore, since only jobs in S are processed throughout this time interval, $p(S) = C_j - t = C_j - r(S)$. Thus, $C_j \leq r(S) + p(S)$; since $d(S) < 0$, we can apply Lemma 2.1 to get that

$$L^*_{\max} \geq r(S) + p(S) - d(S) \geq r(S) + p(S) \geq C_j. \tag{2.1}$$

On the other hand, by applying Lemma 2.1 with $S = \{j\}$,

$$L^*_{\max} \geq r_j + p_j - d_j \geq -d_j. \tag{2.2}$$

Adding inequalities (2.1) and (2.2), we see that the maximum lateness of the schedule computed is

$$L_{\max} = C_j - d_j \leq 2L^*_{\max},$$

which completes the proof of the theorem. □

2.2 The k-Center Problem

The problem of finding similarities and dissimilarities in large amounts of data is ubiquitous: companies wish to group customers with similar purchasing behavior, political consultants group precincts by their voting behavior, and search engines group webpages by their similarity of topic. Usually we speak of *clustering* data, and there has been extensive study of the problem of finding good clusterings.

Here we consider a particular variant of clustering, the k-center problem. In this problem, we are given as input an undirected, complete graph $G = (V, E)$, with a *distance* $d_{ij} \geq 0$ between each pair of vertices $i, j \in V$. We assume $d_{ii} = 0$, $d_{ij} = d_{ji}$ for each $i, j \in V$, and that the distances obey the *triangle inequality*: for each triple $i, j, l \in V$, it is the case that $d_{ij} + d_{jl} \geq d_{il}$. In this problem, distances model similarity: vertices that are closer to each other are more similar, whereas those farther apart are less similar. We are also given a positive integer k as input. The goal is to find k clusters, grouping together the vertices that are most similar into clusters together. In this problem, we will choose a set $S \subseteq V$, $|S| = k$, of k *cluster centers*. Each vertex will assign itself to its closest cluster center, grouping the vertices into k different clusters. For the k-center problem, the objective is to minimize the maximum distance of a vertex to its cluster center. Geometrically speaking, the goal is to find the centers of k different balls of the same radius that cover all points so that the radius is as small as possible. More formally, we define the distance of a vertex i from a set $S \subseteq V$ of vertices to be $d(i, S) = \min_{j \in S} d_{ij}$. Then the corresponding radius for S is equal to $\max_{i \in V} d(i, S)$, and the goal of the k-center problem is to find a set of size k of minimum radius.

In later chapters we will consider other objective functions, such as minimizing the *sum* of distances of vertices to their cluster centers, that is, minimizing $\sum_{i \in V} d(i, S)$. This is called the k-median problem, and we will consider it in Sections 7.7 and 9.2. We shall also consider another variant on clustering called correlation clustering in Section 6.4.

We give a greedy 2-approximation algorithm for the k-center problem that is simple and intuitive. Our algorithm first picks a vertex $i \in V$ arbitrarily, and puts it in our set S of cluster centers. Then it makes sense for the next cluster center to be as far away as possible from all the other cluster centers. Hence, while $|S| < k$, we repeatedly find a vertex $j \in V$ that determines the current radius (or in other words, for which the distance $d(j, S)$ is maximized) and add it to S. Once $|S| = k$, we stop and return S. Our algorithm is given in Algorithm 2.1.

An execution of the algorithm is shown in Figure 2.2.

We will now prove that the algorithm is a good approximation algorithm.

Theorem 2.3. *Algorithm 2.1 is a 2-approximation algorithm for the k-center problem.*

```
Pick arbitrary i ∈ V
S ← {i}
while |S| < k do
    j ← arg max_{j∈V} d(j, S)
    S ← S ∪ {j}
```

Algorithm 2.1. A greedy 2-approximation algorithm for the k-center problem.

Proof. Let $S^* = \{j_1, \ldots, j_k\}$ denote the optimal solution, and let r^* denote its radius. This solution partitions the nodes V into clusters $V_1, \ldots, V_k$, where each point $j \in V$ is placed in V_i if it is closest to j_i among all of the points in S^* (and ties are broken arbitrarily). Each pair of points j and j' in the same cluster V_i are at most $2r^*$ apart: by the triangle inequality, the distance $d_{jj'}$ between them is at most the sum of d_{jj_i}, the distance from j to the center j_i, plus $d_{j_i j'}$, the distance from the center j_i to j' (that is, $d_{jj'} \leq d_{jj_i} + d_{j_i j'}$); since d_{jj_i} and $d_{j'j_i}$ are each at most r^*, we see that $d_{jj'}$ is at most $2r^*$.

Now consider the set $S \subseteq V$ of points selected by the greedy algorithm. If one center in S is selected from each cluster of the optimal solution S^*, then every point in V is clearly within $2r^*$ of some selected point in S. However, suppose that the algorithm selects two points within the same cluster. That is, in some iteration, the algorithm selects a point $j \in V_i$, even though the algorithm had already selected a point $j' \in V_i$ in an earlier iteration. Again, the distance between these two points is at most $2r^*$. The algorithm selects j in this iteration because it is currently the furthest from the points already in S. Hence, all points are within a distance of at most $2r^*$ of some center already selected for S. Clearly, this remains true as the algorithm adds more centers in subsequent iterations, and we have proved the theorem. The instance in Figure 2.2 shows that this analysis is tight. □

We shall argue next that this result is the best possible; if there exists a ρ-approximation algorithm with $\rho < 2$, then P = NP. To see this, we consider the *dominating set problem*, which is NP-complete. In the dominating set problem, we are given a graph $G = (V, E)$ and an integer k, and we must decide if there exists a set $S \subseteq V$ of size k such that each vertex is either in S, or adjacent to a vertex in S. Given an instance

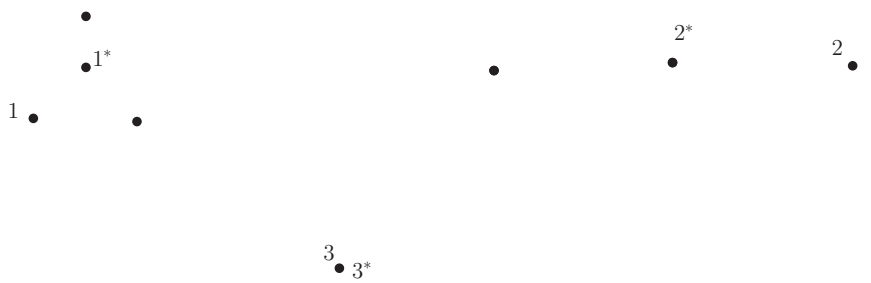

Figure 2.2. An instance of the k-center problem where $k = 3$ and the distances are given by the Euclidean distances between points. The execution of the greedy algorithm is shown; the nodes 1, 2, 3 are the nodes selected by the greedy algorithm, whereas the nodes 1^*, 2^*, 3^* are the three nodes in an optimal solution.

of the dominating set problem, we can define an instance of the k-center problem by setting the distance between adjacent vertices to 1, and nonadjacent vertices to 2: there is a dominating set of size k if and only if the optimal radius for this k-center instance is 1. Furthermore, any ρ-approximation algorithm with $\rho < 2$ must always produce a solution of radius 1 if such a solution exists, since any solution of radius $\rho < 2$ must actually be of radius 1. This implies the following theorem.

Theorem 2.4. *There is no α-approximation algorithm for the k-center problem for $\alpha < 2$ unless* P = NP.

2.3 Scheduling Jobs on Identical Parallel Machines

In Section 2.1, we considered the problem of scheduling jobs on a single machine to minimize lateness. Here we consider a variation on that problem, in which we now have multiple machines and no release dates, but our goal is to minimize the time at which all jobs are finished. Suppose that there are n jobs to be processed, and there are m identical machines (running in parallel) to which each job may be assigned. Each job $j = 1, \ldots, n$, must be processed on one of these machines for p_j time units without interruption, and each job is available for processing at time 0. Each machine can process at most one job at a time. The aim is to complete all jobs as soon as possible; that is, if job j completes at a time C_j (presuming that the schedule starts at time 0), then we wish to minimize $C_{\max} = \max_{j=1,\ldots,n} C_j$, which is often called the *makespan* or *length* of the schedule. An equivalent view of the same problem is as a load-balancing problem: there are n items, each of a given weight p_j, and they are to be distributed among m machines; the aim is to assign each item to one machine so to minimize the maximum total weight assigned to one machine.

This scheduling problem has the property that even the simplest algorithms compute reasonably good solutions. In particular, we will show that both a local search algorithm and a very simple greedy algorithm find solutions that have makespan within a factor of 2 of the optimum. In fact, the analyses of these two algorithms are essentially identical.

Local search algorithms are defined by a set of local changes or local moves that change one feasible solution to another. The simplest local search procedure for this scheduling problem works as follows: Start with any schedule; consider the job ℓ that finishes last; check whether or not there exists a machine to which it can be reassigned that would cause this job to finish earlier. If so, transfer job ℓ to this other machine. We can determine whether to transfer job ℓ by checking if there exists a machine that finishes its currently assigned jobs earlier than $C_\ell - p_\ell$. The local search algorithm repeats this procedure until the last job to complete cannot be transferred. An illustration of this local move is shown in Figure 2.3.

In order to analyze the performance of this local search algorithm, we first provide some natural lower bounds on the length of an optimal schedule, $C^*_{\max}$. Since each job must be processed, it follows that

$$C^*_{\max} \geq \max_{j=1,\ldots,n} p_j. \tag{2.3}$$

On the other hand, there is, in total, $P = \sum_{j=1}^{n} p_j$ units of processing to accomplish, and only m machines to do this work. Hence, on average, a machine will be assigned

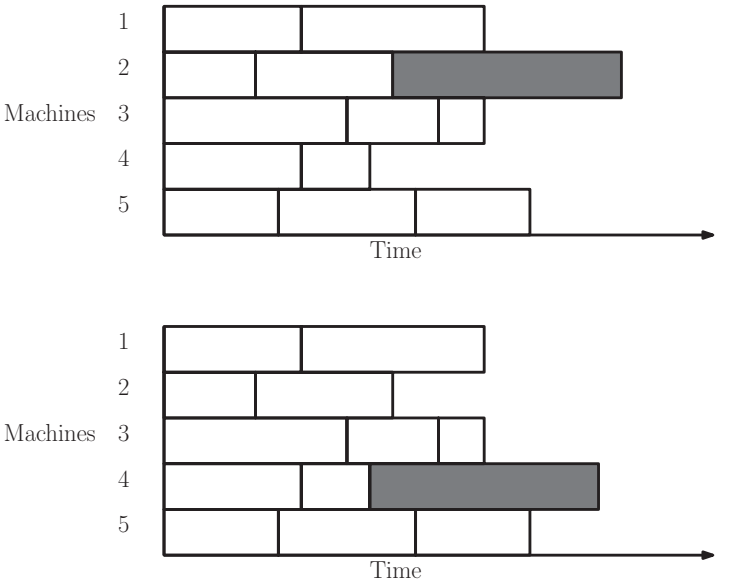

Figure 2.3. An example of a local move in the local search algorithm for scheduling jobs on parallel machines. The gray job on machine 2 finishes last in the schedule on the top, but the schedule can be improved by moving the gray job to machine 4. No further local moves are possible after this one since again the gray job finishes last.

P/m units of work, and consequently, there must exist one machine that is assigned at least that much work. Thus,

$$C^*_{\max} \geq \sum_{j=1}^{n} p_j/m. \tag{2.4}$$

Consider the solution produced by the local search algorithm. Let ℓ be a job that completes last in this final schedule; the completion time of job ℓ, C_ℓ, is equal to this solution's objective function value. By the fact that the algorithm terminated with this schedule, every other machine must be busy from time 0 until the start of job ℓ at time $S_\ell = C_\ell - p_\ell$. We can partition the schedule into two disjoint time intervals, from time 0 until S_ℓ, and the time during which job ℓ is being processed. By (2.3), the latter interval has length at most $C^*_{\max}$. Now consider the former time interval; we know that each machine is busy processing jobs throughout this period. The total amount of work being processed in this interval is mS_ℓ, which is clearly no more than the total work to be done, $\sum_{j=1}^{n} p_j$. Hence,

$$S_\ell \leq \sum_{j=1}^{n} p_j/m. \tag{2.5}$$

By combining this with (2.4), we see that $S_\ell \leq C^*_{\max}$. But now, we see that the length of the schedule before the start of job ℓ is at most $C^*_{\max}$, as is the length of the schedule afterward; in total, the makespan of the schedule computed is at most $2C^*_{\max}$.

Now consider the running time of this algorithm. This local search procedure has the property that the value of $C_{\max}$ for the sequence of schedules produced, iteration by iteration, never increases (it can remain the same, but then the number of machines that achieve the maximum value decreases). One natural assumption to make is that when

we transfer a job to another machine, then we reassign that job to the machine that is currently finishing earliest. We will analyze the running time of this variant instead. Let $C_{\min}$ be the completion time of a machine that completes all its processing the earliest. One consequence of focusing on the new variant is that $C_{\min}$ never decreases (and if it remains the same, then the number of machines that achieve this minimum value decreases). We argue next that this implies that we never transfer a job twice. Suppose this claim is not true, and consider the first time that a job j is transferred twice, say, from machine i to i', and later then to i^*. When job j is reassigned from machine i to machine i', it then starts at $C_{\min}$ for the current schedule. Similarly, when job j is reassigned from machine i' to i^*, it then starts at the current $C'_{\min}$. Furthermore, no change occurred to the schedule on machine i' in between these two moves for job j. Hence, $C'_{\min}$ must be strictly smaller than $C_{\min}$ (in order for the transfer to be an improving move), but this contradicts our claim that the $C_{\min}$ value is non-decreasing over the iterations of the local search algorithm. Hence, no job is transferred twice, and after at most n iterations, the algorithm must terminate. We have thus shown the following theorem.

Theorem 2.5. *The local search procedure for scheduling jobs on identical parallel machines is a 2-approximation algorithm.*

In fact, it is not hard to see that the analysis of the approximation ratio can be refined slightly. In deriving the inequality (2.5), we included job ℓ among the work to be done *prior to the start of job* ℓ. Hence, we actually derived that

$$S_\ell \leq \sum_{j \neq \ell} p_j/m,$$

and hence the total length of the schedule produced is at most

$$p_\ell + \sum_{j \neq \ell} p_j/m = \left(1 - \frac{1}{m}\right) p_\ell + \sum_{j=1}^{n} p_j/m.$$

By applying the two lower bounds (2.3) and (2.4) to these two terms, we see that the schedule has length at most $\left(2 - \frac{1}{m}\right) C^*_{\max}$. Of course, the difference between this bound and 2 is significant only if there are very few machines.

Another natural algorithm to compute a schedule is a greedy algorithm that assigns the jobs as soon as there is machine availability to process them: whenever a machine becomes idle, then one of the remaining jobs is assigned to start processing on that machine. This algorithm is often called the *list scheduling algorithm*, since one can equivalently view the algorithm as first ordering the jobs in a list (arbitrarily), and the next job to be processed is the one at the top of the list. Another viewpoint, from the load-balancing perspective, is that the next job on the list is assigned to the machine that is currently the least heavily loaded. It is in this sense that one can view the algorithm as a greedy algorithm. The analysis of this algorithm is now quite trivial; if one uses this schedule as the starting point for the local search procedure, that algorithm would immediately declare that the solution cannot be improved! To see this, consider a job ℓ that is (one of the jobs) last to complete its processing. Each machine is busy until $C_\ell - p_\ell$, since otherwise we would have assigned job ℓ to that other machine. Hence, no transfers are possible.

Theorem 2.6. *The list scheduling algorithm for the problem of minimizing the makespan on m identical parallel machines is a 2-approximation algorithm.*

It is not hard to obtain a stronger result by improving this list scheduling algorithm. Not all lists yield the same schedule, and it is natural to use an additional greedy rule that first sorts the jobs in non-increasing order. One way to view the results of Theorems 2.5 and 2.6 is that the relative error in the length of the schedule produced is entirely due to the length of the last job to finish. If that job is short, then the error is not too big. This greedy algorithm is called the *longest processing time rule*, or LPT.

Theorem 2.7. *The longest processing time rule is a 4/3-approximation algorithm for scheduling jobs to minimize the makespan on identical parallel machines.*

Proof. Suppose that the theorem is false, and consider an input that provides a counterexample to the theorem. For ease of notation, assume that $p_1 \geq \cdots \geq p_n$. First, we can assume that the last job to complete is indeed the last (and smallest) job in the list. This follows without loss of generality: any counterexample for which the last job ℓ to complete is not the smallest can yield a smaller counterexample, simply by omitting all of the jobs $\ell + 1, \ldots, n$; the length of the schedule produced is the same, and the optimal value of the reduced input can be no larger. Hence, the reduced input is also a counterexample.

So we know that the last job to complete in the schedule is job n. If this is a counterexample, what do we know about $p_n (= p_\ell)$? If $p_\ell \leq C^*_{\max}/3$, then the analysis of Theorem 2.6 implies that the schedule length is at most $(4/3)C^*_{\max}$, and so this is not a counterexample. Hence, we know that in this purported counterexample, job n (and therefore all of the jobs) has a processing requirement strictly greater than $C^*_{\max}/3$. This has the following simple corollary. In the optimal schedule, each machine may process at most two jobs (since otherwise the total processing assigned to that machine is more than $C^*_{\max}$).

However, we have now reduced our assumed counterexample to the point where it simply cannot exist. For inputs of this structure, we have the following lemma.

Lemma 2.8. *For any input to the problem of minimizing the makespan on identical parallel machines for which the processing requirement of each job is more than one-third the optimal makespan, the longest processing time rule computes an optimal schedule.*

This lemma can be proved by some careful case checking, and we defer this to an exercise (Exercise 2.2). However, the consequence of the lemma is clear; no counterexample to the theorem can exist, and hence the theorem must be true. □

In Section 3.2, we will see that it is possible to give a polynomial-time approximation scheme for this problem.

2.4 The Traveling Salesman Problem

In the *traveling salesman problem*, or TSP, there is a given set of cities $\{1, 2, \ldots, n\}$, and the input consists of a symmetric n by n matrix $C = (c_{ij})$ that specifies the cost of traveling from city i to city j. By convention, we assume that the cost of traveling

from any city to itself is equal to 0, and costs are nonnegative; the fact that the matrix is symmetric means that the cost of traveling from city i to city j is equal to the cost of traveling from j to i. (The *asymmetric traveling salesman problem*, where the restriction that the cost matrix be symmetric is relaxed, has already made an appearance in Exercise 1.3.) If we instead view the input as an undirected complete graph with a cost associated with each edge, then a feasible solution, or *tour*, consists of a Hamiltonian cycle in this graph; that is, we specify a cyclic permutation of the cities or, equivalently, a traversal of the cities in the order $k(1), k(2), \ldots, k(n)$, where each city j is listed as a unique image $k(i)$. The cost of the tour is equal to

$$c_{k(n)k(1)} + \sum_{i=1}^{n-1} c_{k(i)k(i+1)}.$$

Observe that each tour has n distinct representations, since it does not matter which city is selected as the one in which the tour starts.

The traveling salesman problem is one of the most well-studied combinatorial optimization problems, and this is certainly true from the point of view of approximation algorithms as well. There are severe limits on our ability to compute near-optimal tours, and we start with a discussion of these results. It is NP-complete to decide whether a given undirected graph $G = (V, E)$ has a Hamiltonian cycle. An approximation algorithm for the TSP can be used to solve the Hamiltonian cycle problem in the following way: Given a graph $G = (V, E)$, form an input to the TSP by setting, for each pair i, j, the cost c_{ij} equal to 1 if $(i, j) \in E$, and equal to $n + 2$ otherwise. If there is a Hamiltonian cycle in G, then there is a tour of cost n, and otherwise each tour costs at least $2n + 1$. If there were to exist a 2-approximation algorithm for the TSP, then we could use this algorithm to distinguish graphs with Hamiltonian cycles from those without any: run the approximation algorithm on the new TSP input, and if the tour computed has cost at most $2n$, then there exists a Hamiltonian cycle in G, and otherwise there does not. Of course, there is nothing special about setting the cost for the "non-edges" to be $n + 2$; setting the cost to be $\alpha n + 2$ has a similarly inflated consequence, and we obtain an input to the TSP of polynomial size provided that, for example, $\alpha = O(2^n)$. As a result, we obtain the following theorem.

Theorem 2.9. *For any $\alpha > 1$, there does not exist an α-approximation algorithm for the traveling salesman problem on n cities, provided* P $\neq$ NP. *In fact, the existence of an $O(2^n)$-approximation algorithm for the TSP would similarly imply that* P = NP.

But is this the end of the story? Clearly not. A natural assumption to make about the input to the TSP is to restrict attention to those inputs that are *metric*; that is, for each triple $i, j, k \in V$, we have that the triangle inequality

$$c_{ik} \leq c_{ij} + c_{jk}$$

holds. This assumption rules out the construction used in the reduction for the Hamiltonian cycle problem above; the non-edges can be given cost at most 2 if we want the triangle inequality to hold, and this value is too small to yield a nontrivial nonapproximability result. We next give three approximation algorithms for this *metric traveling salesman problem*.

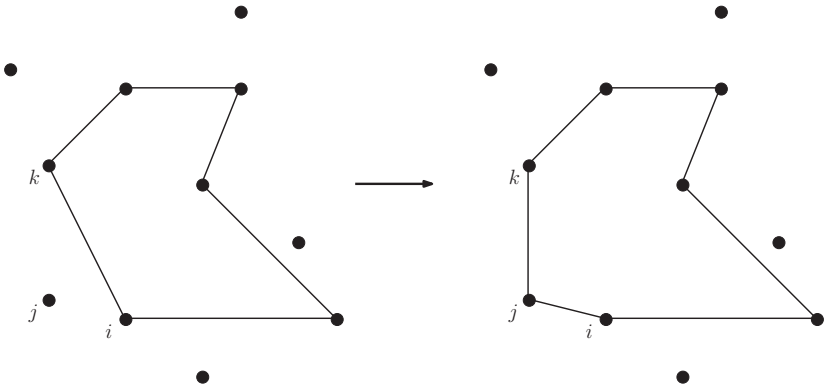

Figure 2.4. Illustration of a greedy step of the nearest addition algorithm.

Here is a natural greedy heuristic to consider for the traveling salesman problem; this is often referred to as the *nearest addition algorithm*. Find the two closest cities, say, i and j, and start by building a tour on that pair of cities; the tour consists of going from i to j and then back to i again. This is the first iteration. In each subsequent iteration, we extend the tour on the current subset S by including one additional city, until we include the full set of cities. In each iteration, we find a pair of cities $i \in S$ and $j \notin S$ for which the cost c_{ij} is minimum; let k be the city that follows i in the current tour on S. We add j to S, and insert j into the current tour between i and k. An illustration of this algorithm is shown in Figure 2.4.

The crux of the analysis of this algorithm is the relationship of this algorithm to Prim's algorithm for the minimum spanning tree in an undirected graph. A *spanning tree* of a connected graph $G = (V, E)$ is a minimal subset of edges $F \subseteq E$ such that each pair of nodes in G is connected by a path using edges only in F. A *minimum spanning tree* is a spanning tree for which the total edge cost is minimized. Prim's algorithm computes a minimum spanning tree by iteratively constructing a set S along with a tree T, starting with $S = \{v\}$ for some (arbitrarily chosen) node $v \in V$ and $T = (S, F)$ with $F = \emptyset$. In each iteration, it determines the edge (i, j) such that $i \in S$ and $j \notin S$ is of minimum cost, and adds the edge (i, j) to F. Clearly, this is the same sequence of vertex pairs identified by the nearest addition algorithm. Furthermore, there is another important relationship between the minimum spanning tree problem and the traveling salesman problem.

Lemma 2.10. *For any input to the traveling salesman problem, the cost of the optimal tour is at least the cost of the minimum spanning tree on the same input.*

Proof. The proof is quite simple. For any input with $n \geq 2$, start with the optimal tour. Delete any one edge from the tour. The result is a spanning tree (albeit a very special one), and this costs no more than the optimal tour. But the minimum spanning tree must cost no more than this special tree. Hence, the cost of the minimum spanning tree is at most the cost of the optimal tour. □

By combining these observations, we can obtain the following result with just a bit more work.

Theorem 2.11. *The nearest addition algorithm for the metric traveling salesman problem is a 2-approximation algorithm.*

Proof. Let $S_2, S_3, \ldots, S_n = \{1, \ldots, n\}$ be the subsets identified at the end of each iteration of the nearest addition algorithm (where $|S_\ell| = \ell$), and let $F = \{(i_2, j_2), (i_3, j_3), \ldots, (i_n, j_n)\}$, where (i_ℓ, j_ℓ) is the edge identified in iteration $\ell - 1$ (with $i_\ell \in S_{\ell-1}$, $\ell = 3, \ldots, n$). As indicated above, we also know that $(\{1, \ldots, n\}, F)$ is a minimum spanning tree for the original input, when viewed as a complete undirected graph with edge costs. Thus, if OPT is the optimal value for the TSP input, then

$$\text{OPT} \geq \sum_{\ell=2}^{n} c_{i_\ell j_\ell}.$$

The cost of the tour on the first two nodes i_2 and j_2 is exactly $2c_{i_2 j_2}$. Consider an iteration in which a city j is inserted between cities i and k in the current tour. How much does the length of the tour increase? An easy calculation gives $c_{ij} + c_{jk} - c_{ik}$. By the triangle inequality, we have that $c_{jk} \leq c_{ji} + c_{ik}$ or, equivalently, $c_{jk} - c_{ik} \leq c_{ji}$. Hence, the increase in cost in this iteration is at most $c_{ij} + c_{ji} = 2c_{ij}$. Thus, overall, we know that the final tour has cost at most

$$2 \sum_{\ell=2}^{n} c_{i_\ell j_\ell} \leq 2\,\text{OPT},$$

and the theorem is proved. □

In fact, this algorithm can be viewed from another perspective. Although this new perspective deviates from viewing the algorithm as a "greedy" procedure, this approach ultimately leads to a better algorithm. First, we need some graph-theoretic preliminaries. A graph is said to be *Eulerian* if there exists a permutation of its edges of the form $(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_0)$; we will call this permutation a *traversal* of the edges, since it allows us to visit every edge exactly once. A graph is Eulerian if and only if it is connected and each node has even degree (where the degree of a node v is the number of edges with v as one of its endpoints). Furthermore, if a graph is Eulerian, one can easily construct the required traversal of the edges, given the graph.

To find a good tour for a TSP input, suppose that we first compute a minimum spanning tree (for example, by Prim's algorithm). Suppose that we then replace each edge by two copies of itself. The resulting (multi)graph has cost at most 2 OPT and is Eulerian. We can construct a tour of the cities from the Eulerian traversal of the edges, $(i_0, i_1), (i_1, i_2), \ldots, (i_{k-1}, i_k), (i_k, i_0)$. Consider the sequence of nodes, $i_0, i_1, \ldots, i_k$, and remove all but the first occurrence of each city in this sequence. This yields a tour containing each city exactly once (assuming we then return to i_0 at the end). To bound the length of this tour, consider two consecutive cities in this tour, i_ℓ and i_m. We have omitted $i_{\ell+1}, \ldots, i_{m-1}$ because these cities have already been visited "earlier" in the tour. However, by the triangle inequality, the cost of the edge c_{i_ℓ, i_m} can be upper bounded by the total cost of the edges traversed in the Eulerian traversal between i_ℓ and i_m, that is, the total cost of the edges $(i_\ell, i_{\ell+1}), \ldots, (i_{m-1}, i_m)$. In total, the cost of the tour is at most the total cost of all of the edges in the Eulerian graph, which is at most 2 OPT. Hence, we have also analyzed this *double-tree algorithm*.

Theorem 2.12. *The double-tree algorithm for the metric traveling salesman problem is a 2-approximation algorithm.*

This technique of "skipping over" previously visited cities and bounding the cost of the resulting tour in terms of the total cost of all the edges is sometimes called *shortcutting*.

The bigger message of the analysis of the double-tree algorithm is also quite useful; if we can efficiently construct an Eulerian subgraph of the complete input graph, for which the total edge cost is at most α times the optimal value of the TSP input, then we have derived an α-approximation algorithm as well. This strategy can be carried out to yield a 3/2-approximation algorithm.

Consider the output from the minimum spanning tree computation. This graph is certainly not Eulerian, since any tree must have nodes of degree one, but it is possible that not many nodes have odd degree. Let O be the set of odd-degree nodes in the minimum spanning tree. For any graph, the sum of its node degrees must be even, since each edge in the graph contributes 2 to this total. The total degree of the even-degree nodes must also be even (since we are adding a collection of even numbers), but then the total degree of the odd-degree nodes must also be even. In other words, we must have an even number of odd-degree nodes; $|O| = 2k$ for some positive integer k.

Suppose that we pair up the nodes in O: $(i_1, i_2), (i_3, i_4), \ldots, (i_{2k-1}, i_{2k})$. Such a collection of edges that contain each node in O exactly once is called a *perfect matching* of O. One of the classic results of combinatorial optimization is that given a complete graph (on an even number of nodes) with edge costs, it is possible to compute the perfect matching of minimum total cost in polynomial time. Given the minimum spanning tree, we identify the set O of odd-degree nodes with even cardinality, and then compute a minimum-cost perfect matching on O. If we add this set of edges to our minimum spanning tree, we have constructed an Eulerian graph on our original set of cities: it is connected (since the spanning tree is connected) and has even degree (since we added a new edge incident to each node of odd degree in the spanning tree). As in the double-tree algorithm, we can shortcut this graph to produce a tour of no greater cost. This algorithm is known as *Christofides' algorithm*.

Theorem 2.13. *Christofides' algorithm for the metric traveling salesman problem is a 3/2-approximation algorithm.*

Proof. We want to show that the edges in the Eulerian graph produced by the algorithm have total cost at most $\frac{3}{2}$ OPT. We know that the minimum spanning tree edges have total cost at most OPT. So we need only show that the perfect matching on O has cost at most OPT/2. This is surprisingly simple.

First observe that there is a tour on just the nodes in O of total cost at most OPT. This again uses the shortcutting argument. Start with the optimal tour on the entire set of cities, and if for two cities i and j, the optimal tour between i and j contains only cities that are not in O, then include edge (i, j) in the tour on O. Each edge in the tour corresponds to disjoint paths in the original tour, and hence by the triangle inequality, the total length of the tour on O is no more than the length of the original tour.

Now consider this "shortcut" tour on the node set O. Color these edges red and blue, alternating colors as the tour is traversed. This partitions the edges into two sets, the red set and the blue set; each of these is a perfect matching on the node set O.

In total, these two edge sets have cost at most OPT. Thus, the cheaper of these two sets has cost at most OPT/2. Hence, there is a perfect matching on O of cost at most OPT/2. Therefore, the algorithm to find the minimum-cost perfect matching must find a matching of cost at most OPT/2, and this completes the proof of the theorem. □

Remarkably, no better approximation algorithm for the metric traveling salesman problem is known. However, substantially better algorithms might yet be found, since the strongest negative result is as follows.

Theorem 2.14. *Unless* P = NP*, for any constant* $\alpha < \frac{220}{219} \approx 1.0045$*, no* α*-approximation algorithm for the metric TSP exists.*

We can give better approximation algorithms for the problem in special cases. In Section 10.1, we will see that it is possible to obtain a polynomial-time approximation scheme in the case that cities correspond to points in the Euclidean plane and the cost of traveling between two cities is equal to the Euclidean distance between the corresponding two points.

2.5 Maximizing Float in Bank Accounts

In the days before quick electronic check clearing, it was often advantageous for large corporations to maintain checking accounts in various locations in order to maximize float. The *float* is the time between making a payment by check and the time that the funds for that payment are deducted from the company's banking account. During that time, the company can continue to accrue interest on the money. Float can also be used by scam artists for *check kiting*: covering a deficit in the checking account in one bank by writing a check against another account in another bank that also has insufficient funds – then a few days later covering this deficit with a check written against the first account.

We can model the problem of maximizing float as follows. Suppose we wish to open up to k bank accounts so as to maximize our float. Let B be the set of banks where we can potentially open accounts, and let P be the set of payees to whom we regularly make payments. Let $v_{ij} \geq 0$ be the value of the float created by paying payee $j \in P$ from bank account $i \in B$; this may take into account the amount of time it takes for a check written to j to clear at i, the interest rate at bank i, and other factors. Then we wish to find a set $S \subseteq B$ of banks at which to open accounts such that $|S| \leq k$. Clearly we will pay payee $j \in P$ from the account $i \in S$ that maximizes v_{ij}. So we wish to find $S \subseteq B$, $|S| \leq k$, that maximizes $\sum_{j \in P} \max_{i \in S} v_{ij}$. We define $v(S)$ to be the value of this objective function for $S \subseteq B$.

A natural greedy algorithm is as follows: we start with $S = \emptyset$, and while $|S| < k$, find the bank $i \in B$ that most increases the objective function, and add it to S. This algorithm is summarized in Algorithm 2.2.

We will show that this algorithm has a performance guarantee of $1 - \frac{1}{e}$. To do this, we require the following lemma. We let O denote an optimal solution, so that $O \subseteq B$ and $|O| \leq k$.

$S \leftarrow \emptyset$
while $|S| < k$ **do**
  $i \leftarrow \arg\max_{i \in B} v(S \cup \{i\}) - v(S)$
  $S \leftarrow S \cup \{i\}$
return S

Algorithm 2.2. A greedy approximation algorithm for the float maximization problem.

Lemma 2.15. *Let S be the set of banks at the start of some iteration of Algorithm 2.2, and let $i \in B$ be the bank chosen in the iteration. Then*

$$v(S \cup \{i\}) - v(S) \geq \frac{1}{k}(v(O) - v(S)).$$

To get some intuition of why this is true, consider the optimal solution O. We can allocate shares of the value of the objective function $v(O)$ to each bank $i \in O$: the value v_{ij} for each $j \in P$ can be allocated to a bank $i \in O$ that attains the maximum of $\max_{i \in O} v_{ij}$. Since $|O| \leq k$, some bank $i \in O$ is allocated at least $v(O)/k$. So after choosing the first bank i to add to S, we have $v(\{i\}) \geq v(O)/k$. Intuitively speaking, there is also another bank $i' \in O$ that is allocated at least a $1/k$ fraction of whatever wasn't allocated to the first bank, so that there is an i' such that $v(S \cup \{i'\}) - v(S) \geq \frac{1}{k}(v(O) - v(S))$, and so on.

Given the lemma, we can prove the performance guarantee of the algorithm.

Theorem 2.16. *Algorithm 2.2 gives a $\left(1 - \frac{1}{e}\right)$-approximation algorithm for the float maximization problem.*

Proof. Let S^t be our greedy solution after t iterations of the algorithm, so that $S^0 = \emptyset$ and $S = S^k$. Let O be an optimal solution. We set $v(\emptyset) = 0$. Note that Lemma 2.15 implies that $v(S^t) \geq \frac{1}{k} v(O) + \left(1 - \frac{1}{k}\right) v(S^{t-1})$. By applying this inequality repeatedly, we have

$$\begin{aligned}
v(S) &= v(S^k) \\
&\geq \frac{1}{k} v(O) + \left(1 - \frac{1}{k}\right) v(S^{k-1}) \\
&\geq \frac{1}{k} v(O) + \left(1 - \frac{1}{k}\right) \left(\frac{1}{k} v(O) + \left(1 - \frac{1}{k}\right) v(S^{k-2})\right) \\
&\geq \frac{v(O)}{k} \left(1 + \left(1 - \frac{1}{k}\right) + \left(1 - \frac{1}{k}\right)^2 + \cdots + \left(1 - \frac{1}{k}\right)^{k-1}\right) \\
&= \frac{v(O)}{k} \cdot \frac{1 - \left(1 - \frac{1}{k}\right)^k}{1 - \left(1 - \frac{1}{k}\right)} \\
&= v(O) \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \\
&\geq v(O) \left(1 - \frac{1}{e}\right),
\end{aligned}$$

where in the final inequality we use the fact that $1 - x \leq e^{-x}$, setting $x = 1/k$. □

To prove Lemma 2.15, we first prove the following.

Lemma 2.17. *For the objective function v, for any $X \subseteq Y$ and any $\ell \notin Y$,*

$$v(Y \cup \{\ell\}) - v(Y) \leq v(X \cup \{\ell\}) - v(X).$$

Proof. Consider any payee $j \in P$. Either j is paid from the same bank account in both $X \cup \{\ell\}$ and X, or it is paid by ℓ from $X \cup \{\ell\}$ and some other bank in X. Consequently,

$$v(X \cup \{\ell\}) - v(X) = \sum_{j \in P} \left(\max_{i \in X \cup \{\ell\}} v_{ij} - \max_{i \in X} v_{ij} \right) = \sum_{j \in P} \max \left\{ 0, \left(v_{\ell j} - \max_{i \in X} v_{ij} \right) \right\}. \tag{2.6}$$

Similarly,

$$v(Y \cup \{\ell\}) - v(Y) = \sum_{j \in P} \max \left\{ 0, \left(v_{\ell j} - \max_{i \in Y} v_{ij} \right) \right\}. \tag{2.7}$$

Now since $X \subseteq Y$ for a given $j \in P$, $\max_{i \in Y} v_{ij} \geq \max_{i \in X} v_{ij}$, so that

$$\max \left\{ 0, \left(v_{\ell j} - \max_{i \in Y} v_{ij} \right) \right\} \leq \max \left\{ 0, \left(v_{\ell j} - \max_{i \in X} v_{ij} \right) \right\}.$$

By summing this inequality over all $j \in P$ and using the equalities (2.6) and (2.7), we obtain the desired result. □

The property of the value function v that we have just proved is one that plays a central role in a number of algorithmic settings, and is often called *submodularity*, though the usual definition of this property is somewhat different (see Exercise 2.10). This definition captures the intuitive property of decreasing marginal benefits: as the set includes more elements, the marginal value of adding a new element decreases.

Finally, we prove Lemma 2.15.

Proof of Lemma 2.15. Let $O - S = \{i_1, \ldots, i_p\}$. Note that since $|O - S| \leq |O| \leq k$, then $p \leq k$. Since adding more bank accounts can only increase the overall value of the solution, we have that

$$v(O) \leq v(O \cup S),$$

and a simple rewriting gives

$$v(O \cup S) = v(S) + \sum_{j=1}^{p} \left[v(S \cup \{i_1, \ldots, i_j\}) - v(S \cup \{i_1, \ldots, i_{j-1}\}) \right].$$

By applying Lemma 2.17, we can upper bound the right-hand side by

$$v(S) + \sum_{j=1}^{p} \left[v(S \cup \{i_j\}) - v(S) \right].$$

Since the algorithm chooses $i \in B$ to maximize $v(S \cup \{i\}) - v(S)$, we have that for any j, $v(S \cup \{i\}) - v(S) \geq v(S \cup \{i_j\}) - v(S)$. We can use this bound to see that

$$v(O) \leq v(O \cup S) \leq v(S) + p[v(S \cup \{i\}) - v(S)] \leq v(S) + k[v(S \cup \{i\}) - v(S)].$$

This inequality can be rewritten to yield the inequality of the lemma, and this completes the proof. □

This greedy approximation algorithm and its analysis can be extended to similar problems in which the objective function $v(S)$ is given by a set of items S, and is monotone and submodular. We leave the definition of these terms and the proofs of the extensions to Exercise 2.10.

2.6 Finding Minimum-Degree Spanning Trees

We now turn to a local search algorithm for the problem of minimizing the maximum degree of a spanning tree. The problem we consider is the following: given a graph $G = (V, E)$ we wish to find a spanning tree T of G so as to minimize the maximum degree of nodes in T. We will call this the *minimum-degree spanning tree problem*. This problem is NP-hard. A special type of spanning tree of a graph is a path that visits all nodes of the graph; this is called a *Hamiltonian path*. A spanning tree has maximum degree two if and only if it is a Hamiltonian path. Furthermore, deciding if a graph G has a Hamiltonian path is NP-complete. Thus, we have the following theorem.

Theorem 2.18. *It is* NP*-complete to decide whether or not a given graph has a minimum-degree spanning tree of maximum degree two.*

For a given graph G, let T^* be the spanning tree that minimizes the maximum degree, and let OPT be the maximum degree of T^*. We will give a polynomial-time local search algorithm that finds a tree T with maximum degree at most $2\,\mathrm{OPT} + \lceil \log_2 n \rceil$, where $n = |V|$ is the number of vertices in the graph. To simplify notation, throughout this section we will let $\ell = \lceil \log_2 n \rceil$.

The local search algorithm starts with an arbitrary spanning tree T. We will give a local move to change T into another spanning tree in which the degree of some vertex has been reduced. Let $d_T(u)$ be the degree of u in T. The local move picks a vertex u and tries to reduce its degree by looking at all edges (v, w) that are not in T but if added to T create a cycle C containing u. Suppose $\max(d_T(v), d_T(w)) \leq d_T(u) - 2$. For example, consider the graph in Figure 2.5, in which the edges of the tree T are shown in bold. In this case, the degree of node u is 5, but those of v and w are 3. Let T' be the result of adding (v, w) and removing an edge from C incident to u. In the example, if we delete edge (u, y), then the degrees of u, v, and w will all be 4 after the move. The conditions ensure that this move provides improvement in general; the degree of u is reduced by one in T' (that is, $d_{T'}(u) = d_T(u) - 1$) and the degrees of v and w in T' are not greater than the reduced degree of u; that is, $\max(d_{T'}(v), d_{T'}(w)) \leq d_{T'}(u)$.

The local search algorithm carries out local moves on nodes that have high degree. It makes sense for us to carry out a local move to reduce the degree of any node, since it is possible that if we reduce the degree of a low-degree node, it may make possible another local move that reduces the degree of a high-degree node. However, we do not know how to show that such an algorithm terminates in polynomial time. To get a polynomial-time algorithm, we apply the local moves only to nodes whose degree is relatively high. Let $\Delta(T)$ be the maximum degree of T; that is, $\Delta(T) = \max_{u \in V} d_T(u)$.

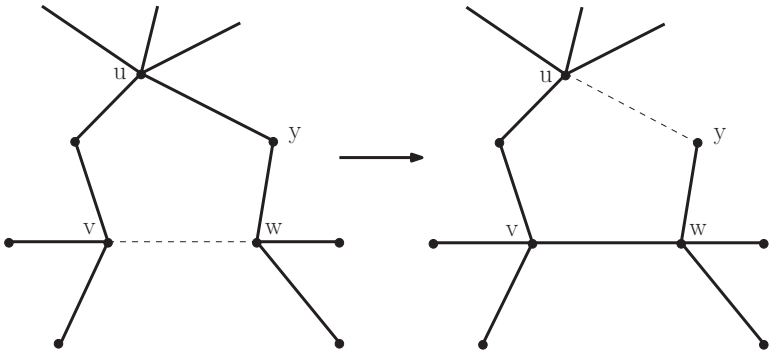

Figure 2.5. Illustration of a local move for minimizing the maximum degree of a spanning tree. The bold solid lines are in the tree, and the dashed lines are graph edges not in the tree.

The algorithm picks a node in T that has degree at least $\Delta(T) - \ell$ and attempts to reduce its degree using the local move. If there is no move that can reduce the degree of any node having degree between $\Delta(T) - \ell$ and $\Delta(T)$, then the algorithm stops. We say that the algorithm has found a *locally optimal* tree. By applying local moves only to nodes whose degree is between $\Delta(T) - \ell$ and $\Delta(T)$, we will be able to show that the algorithm runs in polynomial time.

We now need to prove two things. First, we need to show that any locally optimal tree has maximum degree at most $2\,\mathrm{OPT} + \ell$. Second, we need to show that we can find a locally optimal tree in polynomial time. For most approximation algorithms, the proof that the algorithm runs in polynomial time is relatively straightforward, but this is often not the case for local search algorithms. In fact, we usually need to restrict the set of local moves in order to prove that the algorithm converges to a locally optimal solution in polynomial time. Here we do this by restricting the local moves to apply only to nodes with high degree.

Theorem 2.19. *Let T be a locally optimal tree. Then $\Delta(T) \leq 2\,\mathrm{OPT} + \ell$, where $\ell = \lceil \log_2 n \rceil$.*

Proof. We first explain how we will obtain a lower bound on OPT. Suppose that we remove k edges of the spanning tree. This breaks the tree into $k + 1$ different connected components. Suppose we also find a set of nodes S such that each edge in G connecting two of the $k + 1$ connected components is incident on a node in S. For example, consider the graph in Figure 2.6 that shows the connected components remaining after the bold edges are deleted, along with an appropriate choice of the set S. Observe that any spanning tree of the graph must have at least k edges with endpoints in different components. Thus, the average degree of nodes in S is at least $k/|S|$ for any spanning tree, and $\mathrm{OPT} \geq k/|S|$.

Now we show how to find the set of edges to remove and the set of nodes S so that we can apply this lower bound. Let S_i be the nodes of degree at least i in the locally optimal tree T. We claim that for each S_i, where $i \geq \Delta(T) - \ell + 1$, there are at least $(i - 1)|S_i| + 1$ distinct edges of T incident on the nodes of S_i, and after removing these edges, each edge that connects distinct connected components is incident on a node of S_{i-1}. Furthermore, we claim there exists an i such that $|S_{i-1}| \leq 2|S_i|$, so that the value

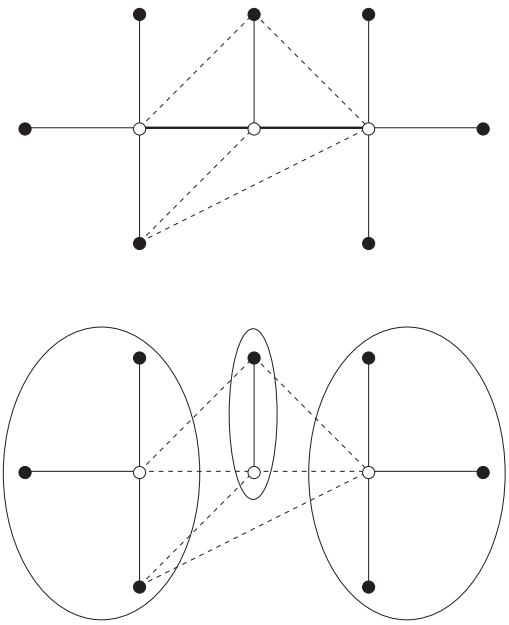

Figure 2.6. Illustration of lower bound on OPT. The vertices in S have white centers. If the bold edges are deleted, every potential edge for joining the resulting connected components has an endpoint in S.

of OPT implied by removing these edges with $S = S_{i-1}$ is

$$\text{OPT} \geq \frac{(i-1)|S_i|+1}{|S_{i-1}|} \geq \frac{(i-1)|S_i|+1}{2|S_i|} > (i-1)/2 \geq (\Delta(T)-\ell)/2.$$

Rearranging terms proves the desired inequality.

We turn to the proofs of the claims. We first show that there must exist some $i \geq \Delta(T) - \ell + 1$ such that $|S_{i-1}| \leq 2|S_i|$. Suppose not. Then clearly $|S_{\Delta(T)-\ell}| > 2^\ell |S_{\Delta(T)}|$ or $|S_{\Delta(T)-\ell}| > n|S_{\Delta(T)}| \geq n$ since $|S_{\Delta(T)}| \geq 1$. This is a contradiction, since any S_i can have at most n nodes.

Now we show that there are at least $(i-1)|S_i|+1$ distinct edges of T incident on the nodes of S_i, and after removing these edges, any edge connecting different connected components is incident on a node of S_{i-1}. Figure 2.6 gives an example of this construction for $i = 4$. Each edge that connects distinct connected components after removing the edges of T incident on nodes of S_i either must be one of the edges of T incident on S_i, or must close a cycle C in T containing some node in S_i. Because the tree is locally optimal, it must be the case that at least one of the endpoints has degree at least $i-1$, and so is in S_{i-1}. In removing the edges in T incident on nodes in S_i, there are at least $i|S_i|$ edges incident on nodes in S_i, since each node has degree at least i. At most $|S_i|-1$ such edges can join two nodes in S_i since T is a spanning tree. Thus, there are at least $i|S_i| - (|S_i|-1)$ distinct edges of T incident on the nodes S_i, and this proves the claim. □

Theorem 2.20. *The algorithm finds a locally optimal tree in polynomial time.*

Proof. To prove that the algorithm runs in polynomial time, we will use a *potential function* argument. The idea of such an argument is that the function captures the current state of the algorithm, and that we can determine upper and lower bounds on this function for any feasible solution, as well as a lower bound on the amount that the function must decrease after each move. In this way, we can bound the number of moves possible before the algorithm must terminate, and of course, the resulting tree must therefore be locally optimal.

For a tree T, we let the potential of T, $\Phi(T)$, be $\Phi(T) = \sum_{v \in V} 3^{d_T(v)}$. Note that $\Phi(T) \leq n3^{\Delta(T)}$, and so the initial potential is at most $n3^n$. On the other hand, the lowest possible potential is for a Hamiltonian path, which has potential $2 \cdot 3 + (n-2)3^2 > n$. We will show that for each move, the potential function of the resulting tree is at most $1 - \frac{2}{27n^3}$ times the potential function previously.

After $\frac{27}{2}n^4 \ln 3$ local moves, the conditions above imply that the potential of the resulting tree is at most

$$\left(1 - \frac{2}{27n^3}\right)^{\frac{27}{2}n^4 \ln 3} \cdot (n3^n) \leq e^{-n \ln 3} \cdot (n3^n) = n,$$

using the fact that $1 - x \leq e^{-x}$. Since the potential of a tree is greater than n, after $O(n^4)$ local moves there must be no further local moves possible, and the tree must be locally optimal.

We must still prove the claimed potential reduction in each iteration. Suppose the algorithm reduces the degree of a vertex u from i to $i-1$, where $i \geq \Delta(T) - \ell$, and adds an edge (v, w). Then the increase in the potential function due to increasing the degree of v and w is at most $2 \cdot (3^{i-1} - 3^{i-2}) = 4 \cdot 3^{i-2}$, since the degree of v and w can be increased to at most $i-1$. The decrease in the potential function due to decreasing the degree of u is $3^i - 3^{i-1} = 2 \cdot 3^{i-1}$. Observe that

$$3^\ell \leq 3 \cdot 3^{\log_2 n} \leq 3 \cdot 2^{2 \log_2 n} = 3n^2.$$

Therefore, the overall decrease in the potential function is at least

$$2 \cdot 3^{i-1} - 4 \cdot 3^{i-2} = \frac{2}{9}3^i \geq \frac{2}{9}3^{\Delta(T)-\ell} \geq \frac{2}{27n^2}3^{\Delta(T)} \geq \frac{2}{27n^3}\Phi(T).$$

Thus, for the resulting tree T' we have that $\Phi(T') \leq (1 - \frac{2}{27n^3})\Phi(T)$. This completes the proof. □

By slightly adjusting the parameters within the same proof outline, we can actually prove a stronger result. Given some constant $b > 1$, suppose we perform local changes on nodes of degree at least $\Delta(T) - \lceil \log_b n \rceil$. Then it is possible to show the following.

Corollary 2.21. *The local search algorithm runs in polynomial time and results in a spanning tree T such that $\Delta(T) \leq b\,\mathrm{OPT} + \lceil \log_b n \rceil$.*

In Section 9.3, we will prove a still stronger result: we can give a polynomial-time algorithm that finds a spanning tree T with $\Delta(T) \leq \mathrm{OPT} + 1$. Given that it is NP-hard to determine whether a spanning tree has degree exactly OPT, this is clearly the best possible result that can be obtained. In the next section, we give another result of this type for the edge coloring problem. In Section 11.2, we will show that there are

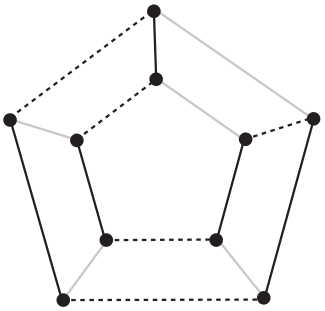

Figure 2.7. A graph with a 3-edge-coloring.

interesting extensions of these results to the case of spanning trees with costs on the edges.

2.7 Edge Coloring

To conclude this chapter, we give an algorithm that has the elements of both a greedy algorithm and a local search algorithm: it attempts to make progress in a greedy way, but when blocked it makes local changes until progress can be made again.

The algorithm is for the problem of finding an *edge coloring* of a graph. An undirected graph is *k-edge-colorable* if each edge can be assigned exactly one of k colors in such a way that no two edges with the same color share an endpoint. We call the assignment of colors to edges a *k-edge-coloring*. For example, Figure 2.7 shows a graph with a 3-edge-coloring. An analogous notion of *vertex coloring* will be discussed in Sections 5.12, 6.5, and 13.2.

For a given graph, we would like to obtain a k-edge-coloring with k as small as possible. Let Δ be the maximum degree of a vertex in the given graph. Clearly, we cannot hope to find a k-edge-coloring with $k < \Delta$, since at least Δ different colors must be incident to any vertex of maximum degree. Note that this shows that the coloring given in Figure 2.7 is optimal. On the other hand, consider the example in Figure 2.8, which is called the *Petersen graph*; it is not too hard to show that this graph is not 3-edge-colorable, and yet it is easy to color it with four colors. Furthermore, the following has been shown.

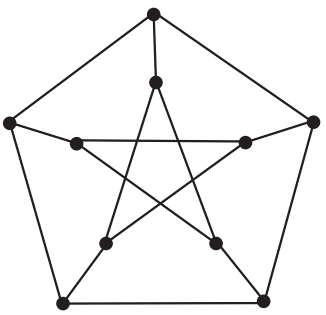

Figure 2.8. The Petersen graph. This graph is not 3-edge-colorable.

Theorem 2.22. *For graphs with $\Delta = 3$, it is* NP*-complete to decide whether the graph is 3-edge-colorable or not.*

In this section, we give a polynomial-time algorithm that will find a $(\Delta + 1)$-edge-coloring for any graph. Given the NP-completeness result, it is clearly the best we can hope to do unless P = NP.

We give the algorithm and its analysis in the proof of the theorem below. We repeatedly find an uncolored edge (u, v) and attempt to color it with one of the $\Delta + 1$ colors. If no color is available such that coloring (u, v) would result in a $(\Delta + 1)$-edge-coloring, then we show that it is possible to locally change some of the edge colors in such a way that we can correctly color (u, v).

Theorem 2.23. *There is a polynomial-time algorithm to find a $(\Delta + 1)$-edge-coloring of a graph.*

Proof. Our algorithm will start with a completely uncolored graph. In each iteration of the algorithm we will take some uncolored edge and color it. This will be done in such a way that the algorithm maintains a *legal* coloring of the graph at the beginning of each iteration of the main loop; that is, for any vertex v in the graph, no two colored edges incident on v have the same color, and at most $\Delta + 1$ distinct colors have been used. Clearly this is true initially, when the entire graph is uncolored. We show that this invariant is maintained throughout the course of each iteration. In the argument that follows, we say that a vertex v *lacks* color c if an edge of color c is not incident on v.

We summarize the algorithm in Algorithm 2.3, and now explain it in detail. Let (u, v_0) be the selected uncolored edge. We then construct a sequence of edges $(u, v_0), (u, v_1), \ldots$ and colors $c_0, c_1, \ldots$. We will use this sequence to do some local recoloring of edges so that we can correctly color the uncolored edge. Note that since we are maintaining a legal coloring of $\Delta + 1$ colors, and the maximum degree is Δ, each vertex must lack at least one of the $\Delta + 1$ colors. To build this sequence of edges, consider the current vertex v_i, starting initially with v_0; if v_i lacks a color that u also lacks, then we let c_i be this color, and the sequence is complete. If not, then choose the color c_i arbitrarily from among those that v_i lacks; note that u will not lack this color. If this color has not appeared in the sequence $c_0, c_1, \ldots$ to this point, then we let v_{i+1} be the vertex such that (u, v_{i+1}) is the edge incident to u of color c_i, and this extends our sequence (sometimes called a *fan sequence*) one edge further. If we have that $c_i = c_j$, where $j < i$, then we also stop building our sequence. (See Figure 2.9 for an example of this construction.)

We need to argue that this process terminates either in the case that u and v_i lack the same color c_i, or that we have $c_i = c_j$ for some $j < i$. Let d be the number of edges incident to u that are currently colored. Suppose the sequence reaches vertex v_d without terminating. If there is no color c_d that both u and v_d lack, then any color that v_d lacks is one that u does not lack, which must be one of the colors on the edges $(u, v_1), \ldots, (u, v_{d-1})$. Hence, the color we choose for c_d must be the same as one of the previous colors $c_0, \ldots, c_{d-1}$.

Suppose we complete the sequence because we find some c_i that both u and v_i lack. This case is easy: we recolor edges (u, v_j) with color c_j for $j = 0, \ldots, i$; call this *shifting recoloring*. This situation and the resulting recoloring are depicted in

```
while G is not completely colored do
    Pick uncolored edge (u, v_0)
    i ← −1
    repeat // Build fan sequence
        i ← i + 1
        if there is a color v_i lacks and u lacks then
            Let c_i be this color
        else
            Pick some color c_i that v_i lacks
            Let v_{i+1} be the edge (u, v_{i+1}) of color c_i
    until c_i is a color u lacks or c_i = c_j for some j < i
    if u and v_i lack color c_i then
        Shift uncolored edge to (u, v_i) and color (u, v_i) with c_i
    else
        Let j < i be such that c_i = c_j
        Shift uncolored edge to (u, v_j)
        Pick color c_u that u lacks
        Let c = c_i
        Let E' be edges colored c or c_u
        if u and v_j in different connected components of (V, E') then
            Switch colors c_u and c in component containing u of (V, E')
            Color (u, v_j) with color c
        else // u and v_i in different components of (V, E')
            Shift uncolored edge to (u, v_i)
            Switch colors c_u and c in component containing u of (V, E')
            Color (u, v_i) with color c
```

Algorithm 2.3. A greedy algorithm to compute a $(\Delta + 1)$-edge-coloring of a graph.

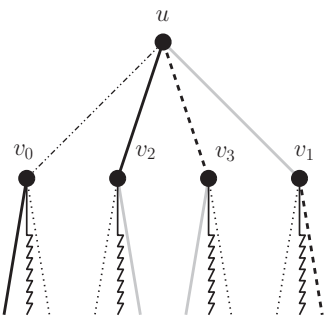

Figure 2.9. An example of building the fan sequence. Edge (u, v_0) is uncolored. Vertex v_0 lacks gray, but u does not lack gray due to (u, v_1), so c_0 is gray. Vertex v_1 lacks black, but u does not lack black due to (u, v_2), so c_1 is black. Vertex v_2 lacks the dashed color, but u does not lack dashed due to (u, v_3), so c_2 is dashed. Vertex v_3 lacks black, so c_3 is black, and the sequence repeats a color.

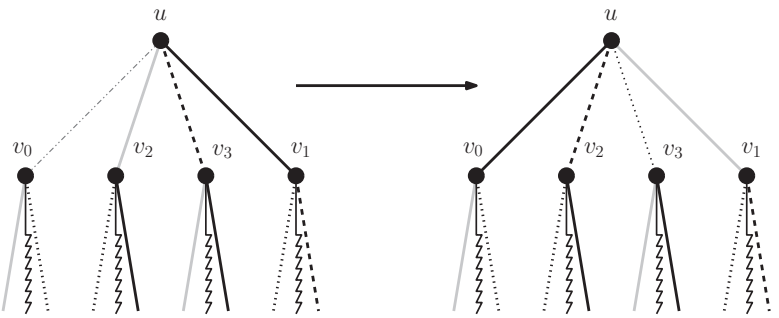

Figure 2.10. A slightly different fan sequence and its recoloring. As before, edge (u, v_0) is uncolored. Vertex v_0 lacks black, but u does not lack black due to (u, v_1), so c_0 is black. Vertex v_1 lacks gray, but u does not lack gray due to (u, v_2), so c_1 is gray. Vertex v_2 lacks dashed, but u does not lack dashed due to (u, v_3), so c_2 is dashed. Vertex v_3 lacks dotted, and u also lacks dotted, and thus c_3 is dotted. Therefore, we shift colors as shown and color the edge (u, v_3) with dotted.

Figure 2.10. In effect, we shift the uncolored edge to (u, v_i) and color it with c_i since both u and v_i lack c_i. The recoloring is correct since we know that each v_j lacks c_j and for $j < i$, c_j was incident on u previously via the edge (u, v_{j+1}), which we now give another color.

Now consider the remaining case, where we complete the sequence because $c_i = c_j$ for some $j < i$. This situation and its recoloring are given in Figure 2.11. We shift the uncolored edge to (u, v_j) by recoloring edges (u, v_k) with color c_k for $0 \leq k < j$ and uncoloring edge (u, v_j); this is correct by the same argument as above. Now v_i and v_j lack the same color $c = c_i = c_j$. We let c_u be a color that u lacks; by our selection (and the fact that we did not fall in the first case), we know that both v_i and v_j do not lack c_u.

Consider the subgraph induced by taking all of the edges with colors c and c_u; since we have a legal coloring, this subgraph must be a collection of paths and simple cycles. Since each of u, v_i, and v_j has exactly one of these two colors incident to it, each is an endpoint of one of the path components, and since a path has only two endpoints, at least one of v_i and v_j must be in a different component than u. Suppose that v_j is in a different component than u. Suppose we recolor every edge of color c with color c_u

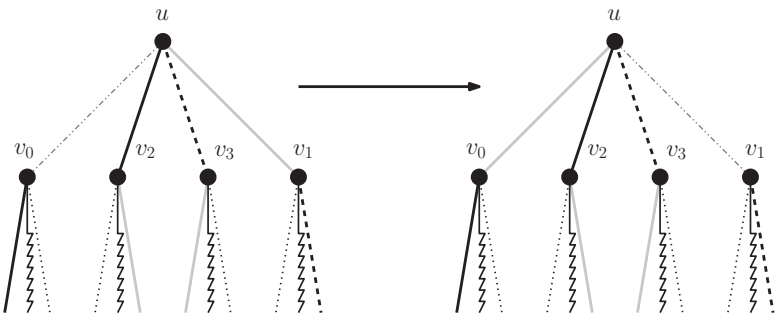

Figure 2.11. The fan sequence from Figure 2.9. We start by shifting the uncolored edge to (u, v_1). Now both v_1 and v_3 lack black. The dotted color can be the color c_u that u lacks but that v_1 and v_3 do not lack.

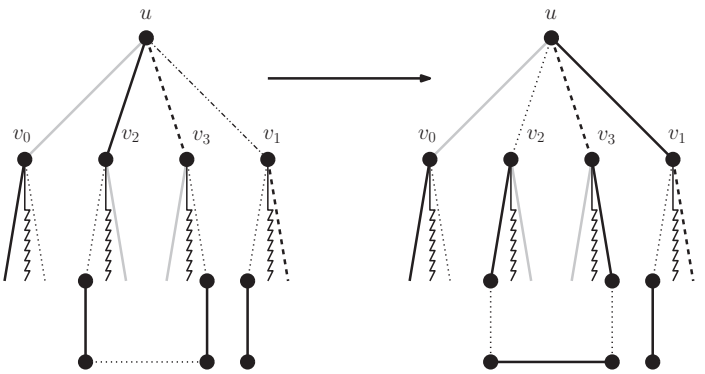

Figure 2.12. The example of Figure 2.11 continued, now showing the components of black and dotted edges containing u, v_1, and v_3. Since u and v_3 are at endpoints of the same black/dotted path, we switch the colors black and dotted on this path, then color (u, v_1) black.

and every edge of color c_u with color c in the component containing u; call this *path recoloring*. Afterward, u now lacks c (and this does not affect the colors incident to v_j at all), and so we may color the uncolored edge (u, v_j) with c. See Figure 2.12 for an example. Finally, suppose that u and v_j are endpoints of the same path, and so v_i must be in a different component. In this case, we can apply the previous shifting recoloring technique to first uncolor the edge (u, v_i). We then apply the path recoloring technique on the u-v_j path to make u lack c; this does not affect any of the colors incident on v_i, and it allows us to color edge (u, v_i) with c.

Clearly we color a previously uncolored edge in each iteration of the algorithm, and each iteration can be implemented in polynomial time. □

Exercises

2.1 The *k-suppliers* problem is similar to the k-center problem given in Section 2.2. The input to the problem is a positive integer k, and a set of vertices V, along with distances d_{ij} between any two vertices i, j that obey the same properties as in the k-center problem. However, now the vertices are partitioned into *suppliers* $F \subseteq V$ and *customers* $D = V - F$. The goal is to find k suppliers such that the maximum distance from a supplier to a customer is minimized. In other words, we wish to find $S \subseteq F$, $|S| \leq k$, that minimizes $\max_{j \in D} d(j, S)$.

(a) Give a 3-approximation algorithm for the k-suppliers problem.
(b) Prove that there is no α-approximation algorithm for $\alpha < 3$ unless P = NP.

2.2 Prove Lemma 2.8: show that for any input to the problem of minimizing the makespan on identical parallel machines for which the processing requirement of each job is more than one-third the optimal makespan, the longest processing time rule computes an optimal schedule.

2.3 We consider scheduling jobs on identical machines as in Section 2.3, but jobs are now subject to *precedence constraints*. We say $i \prec j$ if in any feasible schedule, job i must be completely processed before job j begins processing. A natural variant on the list

scheduling algorithm is one in which whenever a machine becomes idle, then any remaining job that is *available* is assigned to start processing on that machine. A job j is available if all jobs i such that $i \prec j$ have already been completely processed. Show that this list scheduling algorithm is a 2-approximation algorithm for the problem with precedence constraints.

2.4 In this problem, we consider a variant of the problem of scheduling on parallel machines so as to minimize the length of the schedule. Now each machine i has an associated speed s_i, and it takes p_j/s_i units of time to process job j on machine i. Assume that machines are numbered from 1 to m and ordered such that $s_1 \geq s_2 \geq \cdots \geq s_m$. We call these *related* machines.

(a) A *ρ-relaxed decision procedure* for a scheduling problem is an algorithm such that given an instance of the scheduling problem and a deadline D either produces a schedule of length at most $\rho \cdot D$ or correctly states that no schedule of length D is possible for the instance. Show that given a polynomial-time ρ-relaxed decision procedure for the problem of scheduling related machines, one can produce a ρ-approximation algorithm for the problem.

(b) Consider the following variant of the list scheduling algorithm, now for related machines. Given a deadline D, we label every job j with the slowest machine i such that the job could complete on that machine in time D; that is, $p_j/s_i \leq D$. If there is no such machine for a job j, it is clear that no schedule of length D is possible. If machine i becomes idle at a time D or later, it stops processing. If machine i becomes idle at a time before D, it takes the next job of label i that has not been processed, and starts processing it. If no job of label i is available, it looks for jobs of label $i + 1$; if no jobs of label $i + 1$ are available, it looks for jobs of label $i + 2$, and so on. If no such jobs are available, it stops processing. If not all jobs are processed by this procedure, then the algorithm states that no schedule of length D is possible.
Prove that this algorithm is a polynomial-time 2-relaxed decision procedure.

2.5 In the *minimum-cost Steiner tree problem*, we are given as input a complete, undirected graph $G = (V, E)$ with nonnegative costs $c_{ij} \geq 0$ for all edges $(i, j) \in E$. The set of vertices is partitioned into *terminals* R and *nonterminals* (or *Steiner vertices*) $V - R$. The goal is to find a minimum-cost tree containing all terminals.

(a) Suppose initially that the edge costs obey the triangle inequality; that is, $c_{ij} \leq c_{ik} + c_{kj}$ for all $i, j, k \in V$. Let $G[R]$ be the graph induced on the set of terminals; that is, $G[R]$ contains the vertices in R and all edges from G that have both endpoints in R. Consider computing a minimum spanning tree in $G[R]$. Show that this gives a 2-approximation algorithm for the minimum-cost Steiner tree problem.

(b) Now we suppose that edge costs do not obey the triangle inequality, and that the input graph G is connected but not necessarily complete. Let c'_{ij} be the cost of the shortest path from i to j in G using input edge costs c. Consider running the algorithm above in the complete graph G' on V with edge costs c' to obtain a tree T'. To compute a tree T in the original graph G, for each edge $(i, j) \in T'$, we add to T all edges in a shortest path from i to j in G using input edge costs c. Show that this is still a 2-approximation algorithm for the minimum-cost Steiner tree problem on the original (incomplete) input graph G. G' is sometimes called the *metric completion* of G.

2.6 Prove that there can be no α-approximation algorithm for the minimum-degree spanning tree problem for $\alpha < 3/2$ unless P = NP.

2.7 Suppose that an undirected graph G has a Hamiltonian path. Give a polynomial-time algorithm to find a path of length at least $\Omega(\log n/(\log \log n))$.

2.8 Consider the local search algorithm of Section 2.6 for finding a minimum-degree spanning tree, and suppose we apply a local move to a node whenever it is possible to do so; that is, we don't restrict local moves to nodes with degrees between $\Delta(T) - \ell$ and $\Delta(T)$. What kind of performance guarantee can you obtain for a locally optimal tree in this case?

2.9 As given in Exercise 2.5, in the Steiner tree problem we are given an undirected graph $G = (V, E)$ and a set of terminals $R \subseteq V$. A Steiner tree is a tree in G in which all the terminals are connected; a nonterminal need not be spanned. Show that the local search algorithm of Section 2.6 can be adapted to find a Steiner tree whose maximum degree is at most $2\,\mathrm{OPT} + \lceil \log_2 n \rceil$, where OPT is the maximum degree of a minimum-degree Steiner tree.

2.10 Let E be a set of items, and for $S \subseteq E$, let $f(S)$ give the value of the subset S. Suppose we wish to find a maximum value subset of E of at most k items. Furthermore, suppose that $f(\emptyset) = 0$, and that f is *monotone* and *submodular*. We say that f is *monotone* if for any S and T with $S \subseteq T \subseteq E$, then $f(S) \leq f(T)$. We say that f is *submodular* if for any $S, T \subseteq E$, then

$$f(S) + f(T) \geq f(S \cup T) + f(S \cap T).$$

Show that the greedy $(1 - \frac{1}{e})$-approximation algorithm of Section 2.5 extends to this problem.

2.11 In the *maximum coverage problem*, we have a set of elements E, and m subsets of elements $S_1, \ldots, S_m \subseteq E$, each with a nonnegative weight $w_j \geq 0$. The goal is to choose k elements such that we maximize the weight of the subsets that are covered. We say that a subset is covered if we have chosen some element from it. Thus, we want to find $S \subseteq E$ such that $|S| = k$ and that we maximize the total weight of the subsets j such that $S \cap S_j \neq \emptyset$.

(a) Give a $(1 - \frac{1}{e})$-approximation algorithm for this problem.
(b) Show that if an approximation algorithm with performance guarantee better than $1 - \frac{1}{e} + \epsilon$ exists for the maximum coverage problem for some constant $\epsilon > 0$, then every NP-complete problem has an $O(n^{O(\log \log n)})$ time algorithm. (Hint: Recall Theorem 1.13.)

2.12 A *matroid* $(E, \mathcal{I})$ is a set E of ground elements together with a collection $\mathcal{I}$ of subsets of E; that is, if $S \in \mathcal{I}$, then $S \subseteq E$. A set $S \in \mathcal{I}$ is said to be *independent*. The independent sets of a matroid obey the following two axioms:

- If S is independent, then any $S' \subseteq S$ is also independent.
- If S and T are independent, and $|S| < |T|$, then there is some $e \in T - S$ such that $S \cup \{e\}$ is also independent.

An independent set S is a *base* of the matroid if no set strictly containing it is also independent.

(a) Given an undirected graph $G = (V, E)$, show that the forests of G form a matroid; that is, show that if E is the ground set, and $\mathcal{I}$ the set of forests of G, then the matroid axioms are obeyed.

(b) Show that for any matroid, every base of the matroid has the same number of ground elements.
(c) For any given matroid, suppose that for each $e \in E$, we have a nonnegative weight $w_e \geq 0$. Give a greedy algorithm for the problem of finding a maximum-weight base of a matroid.

2.13 Let $(E, \mathcal{I})$ be a matroid as defined in Exercise 2.12, and let f be a monotone, submodular function as defined in Exercise 2.10 such that $f(\emptyset) = 0$. Consider the following local search algorithm for finding a maximum-value base of the matroid: First, start with an arbitrary base S. Then consider all pairs $e \in S$ and $e' \notin S$. If $S \cup \{e\} - \{e'\}$ is a base, and $f(S \cup \{e'\} - \{e\}) > f(S)$, then set $S \leftarrow S \cup \{e'\} - \{e\}$. Repeat until a locally optimal solution is reached. The goal of this problem is to show that a locally optimal solution has value at least half the optimal value.

(a) We begin with a simple case: Suppose that the matroid is a *uniform matroid*; that is, $S \subseteq E$ is independent if $|S| \leq k$ for some fixed k. Prove that for a locally optimal solution S, $f(S) \geq \frac{1}{2}\,\mathrm{OPT}$.
(b) To prove the general case, it is useful to know that for any two bases of a matroid, X and Y, there exists a bijection $g : X \to Y$ such that for any $e \in X$, $S - \{e\} \cup \{g(e)\}$ is independent. Use this to prove that for any locally optimal solution S, $f(S) \geq \frac{1}{2}\,\mathrm{OPT}$.
(c) For any $\epsilon > 0$, give a variant of this algorithm that is a $(\frac{1}{2} - \epsilon)$-approximation algorithm.

2.14 In the *edge-disjoint paths* problem in directed graphs, we are given as input a directed graph $G = (V, A)$ and k source-sink pairs $s_i, t_i \in V$. The goal of the problem is to find edge-disjoint paths so that as many source-sink pairs as possible have a path from s_i to t_i. More formally, let $S \subseteq \{1, \ldots, k\}$. We want to find S and paths P_i for all $i \in S$ such that $|S|$ is as large as possible and for any $i, j \in S$, $i \neq j$, P_i and P_j are edge-disjoint ($P_i \cap P_j = \emptyset$).

Consider the following greedy algorithm for the problem. Let ℓ be the maximum of $\sqrt{m}$ and the diameter of the graph (where $m = |A|$ is the number of input arcs). For each i from 1 to k, we check to see if there exists an s_i-t_i path of length at most ℓ in the graph. If there is such a path P_i, we add i to S and remove the arcs of P_i from the graph.

Show that this greedy algorithm is an $\Omega(1/\ell)$-approximation algorithm for the edge-disjoint paths problem in directed graphs.

2.15 Prove that there is no α-approximation algorithm for the edge coloring problem for $\alpha < 4/3$ unless P = NP.

2.16 Let $G = (V, E)$ be a bipartite graph; that is, V can be partitioned into two sets A and B, such that each edge in E has one endpoint in A and the other in B. Let Δ be the maximum degree of a node in G. Give a polynomial-time algorithm for finding a Δ-edge-coloring of G.

Chapter Notes

As discussed in the introduction to this chapter, greedy algorithms and local search algorithms are very popular choices for heuristics for discrete optimization problems. Thus, it is not surprising that they are among the earliest algorithms analyzed for a performance guarantee. The greedy edge coloring algorithm in Section 2.7 is from a 1964 paper due to Vizing [284]. To the best of our knowledge, this is the earliest

polynomial-time algorithm known for a combinatorial optimization problem that proves that its performance is close to optimal, with an additive performance guarantee. In 1966, Graham [142] gave the list scheduling algorithm for scheduling identical parallel machines found in Section 2.3. To our knowledge, this is the first appearance of a polynomial-time algorithm with a relative performance guarantee. The longest processing time algorithm and its analysis is from a 1969 paper of Graham [143].

Other early examples of the analysis of greedy approximation algorithms include a 1977 paper of Cornuejols, Fisher, and Nemhauser [83], who introduce the float maximization problem of Section 2.5, as well as the algorithm presented there. The analysis of the algorithm presented follows that given in Nemhauser and Wolsey [232]. The earliest due date rule given in Section 2.1 is from a 1955 paper of Jackson [174], and is sometimes called *Jackson's rule*. The analysis of the algorithm in the case of negative due dates was given by Kise, Ibaraki, and Mine [195] in 1979. The nearest addition algorithm for the metric traveling salesman problem given in Section 2.4 and the analysis of the algorithm are from a 1977 paper of Rosenkrantz, Stearns, and Lewis [255]. The double-tree algorithm from that section is folklore, while Christofides' algorithm is due, naturally enough, to Christofides [73]. The hardness result of Theorem 2.9 is due to Sahni and Gonzalez [257] while the result of Theorem 2.14 is due to Papadimitriou and Vempala [239].

There is an enormous literature on the traveling salesman problem. For book-length treatments of the problem, see the book edited by Lawler, Lenstra, Rinnooy Kan, and Shmoys [210] and the book of Applegate, Bixby, Chvátal, and Cook [9].

Of course, greedy algorithms for polynomial-time solvable discrete optimization problems have also been studied for many years. The greedy algorithm for finding a maximum-weight base of a matroid in Exercise 2.12 was given by Rado [246] in 1957, Gale [121] in 1968, and Edmonds [96] in 1971; matroids were first defined by Whitney [285].

Analysis of the performance guarantees of local search algorithms has been relatively rare, at least until some work on facility location problems from the late 1990s and early 2000s that will be discussed in Chapter 9. The local search algorithm for scheduling parallel machines given in Section 2.3 is a simplification of a local search algorithm given in a 1979 paper of Finn and Horowitz [113]; Finn and Horowitz show that their algorithm has performance guarantee of at most 2. The local search algorithm for finding a maximum-value base of Exercise 2.13 was given by Fisher, Nemhauser, and Wolsey [114] in 1978. The local search algorithm for finding a minimum-degree spanning tree in Section 2.6 is from 1992 and can be found in Fürer and Raghavachari [118].

Now to discuss the other results presented in the chapter: The algorithm and analysis of the k-center problem in Section 2.2 is due to Gonzalez [141]. An alternative 2-approximation algorithm for the problem is due to Hochbaum and Shmoys [163]. Theorem 2.4, which states that getting a performance guarantee better than 2 is NP-hard, is due to Hsu and Nemhauser [172]. Theorem 2.22, which states that deciding if a graph is 3-edge-colorable or not is NP-complete, is due to Holyer [170].

The k-supplier problem of Exercise 2.1, as well as a 3-approximation algorithm for it, were introduced in Hochbaum and Shmoys [164]. The list scheduling variant for problems with precedence constraints in Exercise 2.3 is due to Graham [142]. The idea

of a ρ-relaxed decision procedure in Exercise 2.4 is due to Hochbaum and Shmoys [165]; the 2-relaxed decision procedure for related machines in that exercise is due to Shmoys, Wein, and Williamson [265]. Exercise 2.7 was suggested to us by Nick Harvey. Exercise 2.9 is due to Fürer and Raghavachari [118]. Exercises 2.10 and 2.11 are due to Nemhauser, Wolsey, and Fisher [233]. The hardness result of Exercise 2.11 is due to Feige [107]; Feige also shows that the same result can be obtained under the assumption that $P \neq NP$. Kleinberg [199] gives the greedy algorithm for the edge-disjoint paths problem in directed graphs given in Exercise 2.14. Kőnig [201] shows that it is possible to Δ-edge-color a bipartite graph as in Exercise 2.16.