Homework 4 for CMSC 858E

Due 10/09/2018

Note: problem and page number are for the PDF version provided on course website. In case the problem/section/page numbers are not the same, the corresponding pages are also included in this pdf.

Problem 1

Solve problem 2.1, on page 57 in *The Design of Approximation Algorithms* .

Problem 2

Solve problem 2.16 on page 62 in *The Design of Approximation Algorithms*

Problem 3

Please design a 2-approximation algorithm for max-cut. (You need to prove that your algorithm is a 2-approximation)

Problem 4

Solve problem 3.1 on page 77 in *The Design of Approximation Algorithms*

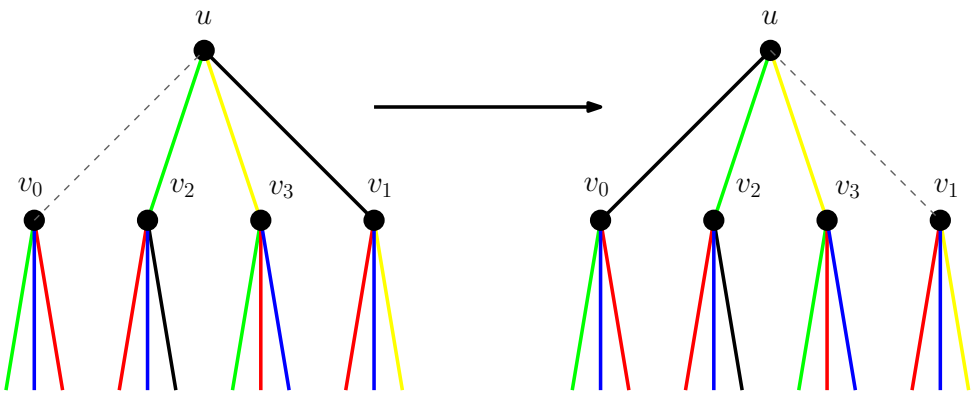

Figure 2.11: The fan sequence from Figure 2.9. We start by shifting the uncolored edge to (u, v_1). Now both v_1 and v_3 lack black. The color blue can be the color c_u that u lacks but that v_1 and v_3 do not lack.

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 have 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 and every edge of color c_u with color c in the component containing u; call this *path recoloring*. Afterwards, 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.

Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys.
To be published by Cambridge University Press

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 [285]. 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 [144] 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 [145].

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 [236]. The earliest due date rule given in Section 2.1 is from a 1955 paper of Jackson [176], 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 [199] 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 [257]. The double-tree algorithm from that section is folklore, while Christofides' algorithm is due, naturally enough, to Christofides [73]. The hardness result of Theorem 2.14 is due to Papadimitriou and Vempala [242].

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 [213] 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 [249] in 1957, Gale [122] in 1968, and Edmonds [97] in 1971; matroids were first defined by Whitney [286].

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 [114]; 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 [115] in 1978. The local search algorithm for finding

Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys.
To be published by Cambridge University Press

constant number of pieces can fit in each bin; hence, we can obtain an optimal packing for the input I' by the dynamic programming algorithm of Section 3.2. This packing for I' can then be used to get a packing for the ungrouped input, and then extended to include all of the small pieces with at most $(1+\epsilon)\,\mathrm{OPT}(I)+1$ bins, as we show below.

Theorem 3.12: *For any $\epsilon > 0$, there is a polynomial-time algorithm for the bin-packing problem that computes a solution with at most $(1+\epsilon)\,\mathrm{OPT}(I)+1$ bins; that is, there is an APTAS for the bin-packing problem.*

Proof. As we discussed earlier, the algorithm will open $\max\{\ell, (1+\epsilon)\,\mathrm{OPT}(I)+1\}$ bins, where ℓ is the number of bins used to pack the large pieces. By Lemma 3.11, we use at most $\mathrm{OPT}(I')+k \leq \mathrm{OPT}(I)+k$ bins to pack these pieces, where $k = \lfloor \epsilon\,\mathrm{SIZE}(I) \rfloor$, so that $\ell \leq \mathrm{OPT}(I)+\epsilon\,\mathrm{SIZE}(I) \leq (1+\epsilon)\,\mathrm{OPT}(I)$, which completes the proof. □

It is possible to improve both the running time of this general approach, and its performance guarantee; we shall return to this problem in Section 4.6.

Exercises

3.1 Consider the following greedy algorithm for the knapsack problem. We initially sort all the items in order of nonincreasing ratio of value to size so that $v_1/s_1 \geq v_2/s_2 \geq \cdots \geq v_n/s_n$. Let i^* be the index of an item of maximum value so that $v_{i^*} = \max_{i \in I} v_i$. The greedy algorithm puts items in the knapsack in index order until the next item no longer fits; that is, it finds k such that $\sum_{i=1}^{k} s_i \leq B$ but $\sum_{i=1}^{k+1} s_i > B$. The algorithm returns either $\{1, \ldots, k\}$ or $\{i^*\}$, whichever has greater value. Prove that this algorithm is a 1/2-approximation algorithm for the knapsack problem.

3.2 One key element in the construction of the fully polynomial-time approximation scheme for the knapsack problem was the ability to compute lower and upper bounds for the optimal value that are within a factor of n of each other (using the maximum value piece that fits in the knapsack to get the lower bound). Use the result of the previous exercise to derive a refined approximation scheme that eliminates one factor of n in the running time of the algorithm.

3.3 Consider the following scheduling problem: there are n jobs to be scheduled on a single machine, where each job j has a processing time p_j, a weight w_j, and a due date d_j, $j = 1, \ldots, n$. The objective is to schedule the jobs so as to maximize the total weight of the jobs that complete by their due date. First prove that there always exists an optimal schedule in which all ontime jobs complete before all late jobs, and the ontime jobs complete in an earliest due date order; use this structural result to show how to solve this problem using dynamic programming in $O(nW)$ time, where $W = \sum_j w_j$. Now use this result to derive a fully polynomial-time approximation scheme.

3.4 Instead of maximizing the total weight of the ontime set of jobs, as in the previous exercise, one could equivalently minimize the total weight of the set of jobs that complete late. This equivalence is only valid when one thinks of optimization, not approximation, since if only one job needs to be late then our approximation for the minimization problem can make only a small error, whereas for the maximization problem the situation is quite different. Or even worse, suppose that all jobs can complete on time. The good news is that there is

Electronic web edition. Copyright 2010 by David P. Williamson and David B. Shmoys.
To be published by Cambridge University Press