The Theory of Computation

Bernard M.E. Moret

Addison-Wesley-Longman, 1998

Solution Manual
Chapter 8, Text Problems

Exercise 8.1

We prove a slightly stronger result, namely that we can solve in polynomial time any instance of *Satisfiability* in which every variable appears at most twice. If a variable appears only once or appears twice as the same literal, then we simply set the literal to true, then remove all satisfied clauses and repeat the process. Thus the only problem of interest arises when every variable appears once uncomplemented and once complemented. But we have seen (in our proof that *3SAT* is NP-complete) that two clauses that share a variable, with one clause including the uncomplemented literal and the other the complemented literal, are logically linked into a single clause. Thus we can process each variable in turn, linking the two clauses in which it appears and eliminating that variable. If the resulting clause contains both a literal and its complement, it is trivially satisfiable—we can eliminate it and process the clauses left over in the fashion just described; otherwise we just replace the two clauses just processed by the new joined clause and repeat our step. The process stops when all clauses are eliminated (success) or when we have created two single-literal clauses for the same variable, one complemented and the other uncomplemented (failure). Clearly, the whole process takes at most quadratic time (a linear number of variables to be processed, each taking up to linear time).

Exercise 8.2

We prove that *Planar NAE3SAT* is in P by reduction to a known problem in P, namely *Planar MaxCut*. Our reduction uses only local replacement and carefully expands the single variable nodes of *Planar NAE3SAT* into chains of nodes for *MaxCut* so as to produce only planar instances of *MaxCut*. Orlova and Dorfman ("Finding the maximum cut in a graph," *Eng. Cybernetics* **10** (1972), 502–506) showed that such instances can be solved in polynomial time by a matching process carried on the dual of the graph. Thus *Planar NAE3SAT* is in P.

An instance of (Nonpolar) *Planar NAE3SAT* is defined as an instance of *NAE3SAT*, except that the graph produced from it according to the following rules must be planar. The graph has one vertex for each variable of the instance and one vertex for each clause of the instance; all variable vertices are connected in a simple cycle and each clause vertex is connected by an edge to variable vertices corresponding to the literals present in the clause. Note that positive and negative literals are treated exactly alike. (Simple) *MaxCut* is defined as follows: given an undirected graph and given a bound (a positive integer), can the vertices of the graph be partitioned into two subsets such that the number of edges with endpoints in both subsets is no smaller than the given bound?

Our transformation is quite elementary. Given an instance of *Planar NAE3SAT* with k clauses, we transform it in polynomial time into an instance of *MaxCut* with $9k$ vertices and no more than $6k$ edges as follows. For each variable x forming a total of n_x literals in the k clauses, we set up a simple cycle of $2n_x$ vertices (and $2n_x$ edges); alternating vertices represent complemented and uncomplemented literals.[1] For each clause, we set up a triangle, where each vertex of the triangle is connected by an edge to the (complement of the) corresponding "literal vertex." Finally, we set the minimum number of edges to be cut to $11k$ This transformation is clearly feasible in polynomial time.

Now, given a satisfying truth assignment, we put all vertices corresponding to true literals on one side of the partition and all others on the other side. Since the truth assignment is valid, each edge between a literal and its complement is cut, thereby contributing a total (for all variables) of $6k$ (i.e., twice the total number of literals) to the cut sum. Since the truth assignment is a solution, each triangle is cut (not all three vertices may be on the same side, as this would correspond to a clause with three false or three true literals), thereby contributing a total of $5k$ to the cut sum. Hence we have a solution to *MaxCut*. Conversely, observe that $11k$ is the maximum attainable cut sum: we cannot do better than cut each clause triangle and each segment between complementary literals. Moreover, the cut sum of $11k$ can only be reached by cutting all triangles and segments. (If all three vertices of a clause triangle are placed on the same side of the partition, at most 4 of the 6 edges associated with the clause can be cut.) Hence a solution to *MaxCut* yields a solution to *NAE3SAT*: cutting the segments ensures a

---

[1]Note that, if $n_x = 1$, this would produce a double edge between two vertices; in this case only, we set up a single edge between the two vertices and lower the required number of cut edges by one. Since this is an elementary adjustment we shall assume in the following that $n_x \geq 2$ rather than qualify every statement to take this special case into account.

valid truth assignment and cutting each triangle ensures a satisfying truth assignment.

Only one point remains: to verify that the instance produced is indeed planar. No crossing can arise from the replacement of the single clause vertices of *Planar NAE3SAT* the triangles of *MaxCut*, as the vertices of the triangles are assigned arbitrarily. The replacement of the single variable vertices of *Planar NAE3SAT* by the simple cycles of *MaxCut* likewise cannot produce non-planar configurations, as the choice of vertex pair for some literal $\tilde{x}$ from among the n_x pairs present in the circle is also arbitrary. The correctness of this transformation completes our proof.

It is interesting to note that rather more transformations from the *3SAT* family of problems to graph problems produce a single edge per variable (with the endpoints intended to represent the two truth value assignments) than produce a single vertex or arbitrarily labelled structure. Thus a potentially more useful definition of *Planar 3SAT* would constrain instances according to a different graph construction, where each variable is in fact represented by a single edge and where the single vertex clause is connected to the proper (i.e., complemented or not) endpoint of the variable's edge—this is the polar version of *Planar 3SAT*. Simplifying the transformation given above to one where a single edge (rather than a simple cycle) is produced for each variable—thus yielding the simple transformation given in the text—shows that *Planar NAE3SAT* remains in P under this new definition.

Exercise 8.3

Exercise 8.4

Exercise 8.5

Exercise 8.6

Exercise 8.7

Exercise 8.8

Exercise 8.9

Exercise 8.10

1. Membership in NP is trivial. The design of a degree-reducing gadget for *Vertex Cover* is surprisingly easy. Given a vertex of degree $d > 3$, it is enough to replace it by a gadget with $2d - 1$ vertices in a line (connected by $2d - 2$ edges), where the d odd vertices are

attaching points; for each such gadget used, the bound on the number of vertices in the cover is increased by $d - 1$. First note that the resulting transformation is easily accomplished in polynomial time and produces graphs with maximum degree 3.

To see that a solution to the original problem implies a solution to the transformed problem is very simple: if the degree d vertex was part of the cover in the original, all d attaching points are part of the cover in the transformed version, whereas, if the vertex was not selected, the $(d - 1)$ even vertices are selected instead. In both cases the bound is respected (the difference remains exactly $(d - 1)$, which is what was added), the gadget is covered, and the d edges connecting the the degree d vertex are either all covered or all uncovered. To see that the converse implication also holds, it suffices to note that, in order to cover the $2d - 2$ edges internal to the gadget, at least $(d - 1)$ vertices of the gadget must be selected; moreover, if only $(d - 1)$ are selected, then they must be the even vertices, which do not cover any of the d edges incident from "outside." Hence a solution to the transformed instance uses a minimum number of the cover vertices allowed by the bound to cover the gadgets themselves; what is left is exactly the bound of the original instance.

Another solution uses a cycle of $2d$ vertices and edges, with the odd-numbered vertices used for connecting to the rest of the graph, plus one more vertex of degree one, connected to one of the even-numbered vertices. The vertex of degree one and its edge serve ensure that, if all odd-numbered vertices are selected for the cover (corresponding to placing the original vertex of degree d in the cover), then one more vertex is needed to cover this additional edge.

2. A solution to the degree reduction in *Maximum Cut* can be developed along much the same lines, but needs a small amendment. To replace a vertex of degree $d > 3$, we can try to use a simple cycle of length $2d$, where odd vertices serve as attaching points, and increase the bound by $2d$, thereby effectively requiring that all gadget edges be cut. This cut can be accomplished by coloring all attaching points with the same color and all others with the opposite color. Thus a solution to the original problem immediately implies a solution to the transformed problem. Unfortunately, there could be unforeseen cuts that still work. If the original problem has a vertex of degree $2k$ where k of the neighbors are placed on one side of the partition and the other k on the other side and the k neighbors on the same side are next to each other in the cycle, we could partition the gadget to cut all $2k$ edges leading out of it to the neighbors and still manage to cut $4k - 2$

edges inside the gadget, leading to a cut of $6k - 2$ edges, whereas the original graph had only k cut edges, a difference of $5k - 2$—exceeding the $4k$ increase in the bound and thus invalidating our transformation for $k > 2$. But what works once works twice: we can connect the gadget vertices of degree 2 in a cycle of their own by adding another d vertices of degree 2, thereby creating a second, "inner" cycle; the bound is correspondingly increased by an additional $2d$ (this turns the vertices of degree 2 into vertices of degree 4, higher than allowed; but our gadget works for degree 4 vertices without modification, so we can use it later to take care of these vertices). Now a vertex of degree $2k$ in the same circumstances gives rise to a gadget where we can cut all outside edges and all gadget edges, except for 2 per cycle, for a total cut of $8k - 4 + 2k$ (not accounting for the contribution of the gadgets used to replace the vertices of degree 4 introduced by the construction); since the original had a cut of k, the difference is $9k - 4$, which exceeds the difference in bounds of $4d = 8k$ when $k > 4$. At this point, we can add a third inner cycle in the same style; in general, a simple computation shows that a vertex of degree $2k$ or $2k + 1$ will require $n = \lceil k/2 \rceil$ nested cycles, with a corresponding bound increase of $4nk$ or $2n(2k + 1)$, which remains safely polynomial. Once the nested cycles have been introduced, all vertices of degree 4 are replaced by our "first-level" gadget.

We can avoid the nesting altogether and use a simple cycle of $2d$ vertices as originally intended if it is used in a context where all edges connected to the original vertex of degree d must be cut—in which case the problem described above cannot arise; such a context is precisely that arising out of our original transformation from *NAE3SAT* to *MaxCut*: the clauses vertices all have degree 3 and thus need not be replaced, while and all edges incident upon the truth-setting vertices must be cut.

In both *VC* and *MaxCut*, it is possible that the transformed instance admits "weird" solutions, where the gadget is not treated as described above. Those solutions arise whenever the original instance had a loose bound—where many fewer than B vertices would constitute a cover or where many more than B edges could be cut. In those cases, however, we can easily "migrate" elements of the solution into the gadgets to obtain a new solution where the gadgets are used as intended. In both *VC* and *MaxCut*, the latter problem can be avoided by using a transformation from a *SAT* problem; the simplest such (even simpler than above) is to use, in both cases, an extension of the truth-setting component to provide separate attaching points for each clause literal. The original truth-setting component in each transformation was a single edge per variable; for variable x_i, use instead $2k_i + 1$ edges in a line, where k_i is the larger of the number of occurrences of x_i and the number of occurrences of $\overline{x}_i$, and increase the bound for *VC* by k and the bound for *MaxCut* by $2k$. The resulting instances have exactly the same structure as those produced in the original transformation, but the degree of every vertex is at most 3.

Exercise 8.11

Exercise 8.12

An instance of the *Minimum Complete Research Program* problem is given by a set X of "open problems," a partial order on X, and a positive integer $B \leq |X|$. The question is whether there exists a subset $X' \subseteq X$, $|X'| \leq B$, and a "complexity classification" function $c\colon X' \to \{\text{easy, hard}\}$ that would resolve all the open problems, i.e., such that c can be extended to a well-defined total function on X by using the partial order:

- $x < y$ and $c(x) = \text{hard} \Rightarrow c(y) = \text{hard}$
- $x < y$ and $c(y) = \text{easy} \Rightarrow c(x) = \text{easy}$

We prove this problem NP-complete (it is obviously in NP) by transforming *Vertex Cover* to it. Given an instance of *VC*, a graph $G = (V, E)$ and a positive integer $K \leq |V|$, we produce an instance of *MCRP* as follows. Let $l = |V| + K$. For each vertex $v \in V$, we create $l + 2$ vertices, $v_1, \ldots, v_l, v', v''$, and $l + 1$ arcs, (v_i, v') for $1 \leq i \leq l$, and (v', v''). For each edge, $(v, w) \in E$, we create l vertices, $e_1, \ldots, e_l$ and $2l$ arcs, (v'', e_i) and (w'', e_i), for $1 \leq i \leq l$. The bound on the size of the research program is set to l. This transformation is clearly feasible in polynomial time.

Now, if we have a vertex cover, $U \subseteq V$, of size K or less, we can obtain a complete research program of size l or less by classifying each node v' $(v \in U)$ hard, all nodes v'' $(v \notin U)$ easy, and all nodes w' $(w \notin U)$ easy. (It follows that all remaining w'' nodes and all e_i nodes get classified as easy and all v_i nodes as hard.) Conversely, a complete research program of size l or less must yield a vertex cover of size K or less. By construction, the e_i nodes can only be classified as easy and the v_i nodes as hard; moreover, none of these nodes can be part of the l nodes. If the research program contains only one of v' and v'' for some v, then it contains v'' with classification easy. The rest follows.

A somewhat easier transformation from *VC* proceeds as follows: for each vertex v, set up three problems v, v', v'' with $v' < v$ and $v' < v''$; for each edge e, set up one problem e, easier than both its endpoints; and set the bound to $|V| + K$. The correctness of the transformation is easily verified.

Exercise 8.13

Exercise 8.14

We want to prove Vizing's theorem: the chromatic index of a graph is either the maximum degree of the graph or one more than that. That the chromatic index has to be at least equal to the degree of the graph is obvious, so we need only prove that it need not exceed that degree by more than one.

Let d be the maximum degree of the graph. We prove the theorem by induction on d. The basis is clearly true for $d = 0$: such a graph has no edge at all, so that its chromatic index is 0. The proof does not involve any deep reasoning, but it has a lot of case analysis.

Assume then that the theorem holds for all graphs of degree up to some d and let us consider an arbitrary graph G of degree $d + 1$. Remove from this graph enough edges to reduce its degree to d, for instance in the following manner: start at any vertex of degree $d + 1$; if one of its neighbors is a vertex of degree $d + 1$, remove the edge between the two, otherwise remove any incident edge; then move on to any remaining vertex of degree $d + 1$. By inductive hypothesis, the resulting subgraph G' can be edge-colored with d or $d + 1$ colors. Now consider adding back (or, equivalently, coloring) the removed edges. In particular, consider coloring edge $\{u, v\}$; we proceed to show that we can color this edge from our set of $d + 2$ colors. We can then move on to any remaining uncolored edge and eventually color the whole graph, thereby completing our proof.

Denote by $d(w)$ the set of colors (between 1 and $d + 2$) *not* used among the colored edges incident upon w. Obviously, if we have $d(u) \cap d(v) \neq \emptyset$, we can assign to the new edge any color from the intersection. Consider then the cases where $d(u)$ and $d(v)$ do not share any color. By hypothesis, we have $d(u) \geq 2$ and $d(v) \geq 2$, because at least on edge incident upon each is as yet uncolored and each can have at most $d + 1$ incident edges. For any neighbor w of u, we must also have $d(w) \geq 1$, again because no vertex has degree exceeding $d + 1$; in case the edge $\{u, w\}$ is not yet colored, then we must in fact have $d(w) \geq 2$. Thus we can think of assigning to each colored edge $\{u, w_i\}$ incident upon u an alternate color picked from $d(w_i)$, a color that differs from the color used for the edge.

Let c_1 be a color in $d(v)$; by our assumption, it cannot be a color in $d(u)$, so that there must be an edge incident upon u that is colored c_1; denote this edge $\{u, w_1\}$ and denote the alternate color assigned to this edge by c_2—thus we have both $c_2 \neq c_1$ and $c_2 \notin d(w_1)$. Now either c_2 is in $d(u)$ or it is not. If it is, we are done: we recolor edge $\{u, w_1\}$ with color c_2, leaving us free to color edge $\{u, v\}$ with color c_1.

If it is not, we start an inductive process. If c_2 is not in $d(u)$, then there is some vertex, call it w_2, such that the edge $\{u, w_2\}$ is colored with c_2. Let c_3 be the alternate color that was assigned to this edge. Again, either c_3 is in $d(u)$ or it is not. Again, if it is, we are done, this time with a slightly longer chain: we recolor edge $\{u, w_2\}$ with color c_3, then recolor edge $\{u, w_1\}$ with color c_2, leaving us free to color edge $\{u, v\}$ with color c_1. If it is not, then there must be some vertex w_3 such that edge $\{u, w_3\}$ is colored with c_3. The process must either terminate at some stage i with c_i in $d(u)$, in which case we successively recolor all edges involved, freeing us to color $\{u, v\}$ with color c_1, or it terminates at some stage i because we have a "loop"—that is, because c_{i+1}, while not in $d(u)$, is equal to an earlier c_j, in which case we must have $j < i$. Thus we have just this last case to handle: c_{i+1} is not in $d(u)$ and $c_{i+1} = c_j$ for some $j < i$. We now distinguish two cases: $j = 1$ and $j > 1$.

If j equals 1, i.e., if c_{i+1} equals c_1, then vertex w_i cannot be vertex v, because edge $\{u, v\}$ is not (yet) colored, whereas edge $\{u, w_i\}$ already is. Consider then the three distinct vertices u, v, and w_i. Let c be come color in $d(u)$; we know that c_1 is a color in both $d(v)$ and $d(w_i)$. Look at the subgraph of G that uses only edges colored with c or c_1. In this graph, every vertex has degree at most 2 and any vertex of degree 2 has incident upon it one edge colored c and one edge colored c_1; a path in this graph has edges of alternating colors. Two subcases arise: if there is no path from u to v in this graph, follow any maximal (i.e., not a subset of another) path or cycle from a; any such path or cycle must alternate edge colors, starting with a c_1 edge, moving to a c edge, etc. (Of course, there may be must one edge or even no edge at all.) Exchange the colors on this path or cycle; the result is still a legal edge-coloring in G' and now avoids c_1 at u, leaving us free to use it for coloring edge $\{u, v\}$. Otherwise, there is no path from w_i to one (or both) of u and v (because c_1 is used at neither w_i nor v), so follow any maximal path or cycle from w_i and exchange the colors c and c_1 in this path or cycle. Then the color c is no longer used at w_i, so that we can recolor the edge $\{u, w_i\}$ with color c and thus use color c_1 to color edge $\{u, v\}$.

If j is larger than 1, we have a comparable situation: vertex w_i cannot be vertex w_{j-1} (otherwise we would have two edges from u assigned the same color), and thus we can consider the three distinct vertices u, w_i, and w_{j-1}. Let c be come color in $d(u)$; we know that c_j is a color in both $d(w_i)$ and $d(w_{j-1})$. Again, we have two subcases (is u connected to w_{j-1} in the 2-colored subgraph) and again we follow a maximal path or cycle and exchange all colors along it to free up a color at the beginning of the path.

Exercise 8.15

We prove the problem NP-complete by transforming *MaxCut* to it. We simply let A, the matrix of our problem, be the adjacency matrix of the the graph of *MaxCut* and set the bound K for our problem to $2(|E| - 2B)$, where B is the bound in the instance of *MaxCut*. Given a partition cutting at least B edges, we weight vertices on one side with 1, vertices on the other side with -1; the product $f(i)f(j)$ is 1 whenever both endpoints of the edge are on the same side of the partition, -1 otherwise. Since at least B edges are cut, at least B such products equal -1; there are $|E|$ edges in all, so at most $|E| - B$ are not cut and contribute 1 each and at least B are cut and contribute -1 each, for a total not exceeding $|E| - 2B$. The factor of 2 comes from the fact that the full matrix (symmetric because the graph is undirected) is used. Since we only produce small numbers (0, 1, and K, all linear in the size of the input) and since *MaxCut* (a numberless problem) is itself strongly NP-complete, it follows that *Matrix Cover* is strongly NP-complete.

Exercise 8.16

Exercise 8.17

We prove the problem to be NP-complete with just 2 currencies, then show how to add any fixed number of currencies without affecting the result.

Note that proving it complete with 2 currencies is a stronger result than proving it complete for a floating number of currencies; we can do this last trivially by mapping *VC* as follows: for each vertex there will be a box; for each edge there will be a currency; each box contains one unit of currency for each currency that corresponds to an edge incident to the corresponding box; and the requirement is one unit of each currency—i.e., cover each edge.

We transform *1in3SAT* to our problem restricted to two currencies. Our transformation is essentially the same as for *Partition*; indeed, one could view the transformation as one from *Partition*, but a form of it where the *number* of items on each side of the partition is known in advance. Recall our transformation (in the text) from *1in3SAT* to *Partition*. Each literal gives rise to one object, hence $2n$ objects in all (plus the enforcer object); the size of an object is determined by the identity of the literal and the clauses in which it appears; let the size of object i be denoted by $s(i)$. We saw that the sum of the sizes of the $2n$ elements is an odd number, call it S; we made this number even by adding an enforcer with appropriate size—let us call the even value thus obtained S'.

We create an instance of our problem with $2n$ safe deposit boxes. Each box corresponds to a literal and contains two currencies; specifically, box i contains $s(i)$ units of currency 1 and $2S - s(i)$ units of currency 2. We set the upper bound, k, on the number of boxes to be opened to n and set the target to be $\frac{1}{2}S'$ units of currency 1 and $2nS - \frac{1}{2}S'$ units of currency 2.

Note that, in order to reach the bound on currency 2, we must open at least n of the $2n$ boxes (each box contributes at most $2S$ and we must reach a total larger than $2(n-1)S$); since $k = n$, this means that we must open exactly n boxes. But then we must open n boxes such that the sum of the $s(i)$ for these boxes is at least $\frac{1}{2}S'$ (currency 1) and at most $\frac{1}{2}S'$ (to achieve the total for currency 2), i.e., such that the sum of the $s(i)$ for these boxes is exactly $\frac{1}{2}S'$, which, as we have seen in the transformation to *Partition*, corresponds exactly to the false literals in a solution to *1in3SAT*.

To add a new currency, just place one unit of that currency in each box and set the requirement for that currency to n—we know that exactly n boxes will be opened in any solution, so this new currency does not alter the solution in any way.

Exercise 8.18

Exercise 8.19

Exercise 8.20

In all three cases, the straightforward multiplication technique works.

1. Given a claimed algorithm $\mathcal{A}$ that guarantees a distance K approximation, we solve an arbitrary instance $S, \mathcal{C} \subseteq 2^S$ of the set cover problem exactly as follows. We produce the new instance $S', \mathcal{C}' \subseteq 2^{S'}$ where S' is the disjoint union of $(K+1)$ copies of S and, similarly, $\mathcal{C}'$ is the disjoint union of $(K+1)$ copies of $\mathcal{C}$. Note that the minimum cover for this new instance is exactly $(K+1)$ times the size of the minimum cover for the original instance. We apply $\mathcal{A}$ to this new instance. Since the cover it returns has at most K subsets more than the minimum cover, $\mathcal{A}$ in fact must have solved one of the copies optimally.
2. Given a claimed algorithm $\mathcal{A}$ that guarantees a distance K approximation, we solve an arbitrary instance $V, \mathcal{C}$ of *2SAT* (where V denotes the set of variables and $\mathcal{C}$ the set of clauses) exactly as follows. We produce the new instance $V', \mathcal{C}'$, where V' is the disjoint union of $(K+1)$ copies of V and $\mathcal{C}'$ is the disjoint union of $(K+1)$ copies of $\mathcal{C}$. Note that the maximum number of clauses satisfiable in this new instance, call it M, is exactly $(K+1)$ times the maximum number satisfiable in the original instance, call it m; that is $M = (K+1)m$.

We apply $\mathscr{A}$ to this new instance. Since it returns a truth assignment that satisfies at least $M - K$ of the clauses and since no more than m clauses can be satisfied in each copy, it must have produced a truth assignment that satisfies exactly m clauses in one of the copies.

3. Given a claimed algorithm $\mathscr{A}$ that guarantees a distance K approximation, we solve an arbitrary instance $G = (V, E)$ exactly as follows. We produce a new instance $G' = (V', E')$ which consists simply of $(K + 1)$ copies of G. Note that G' will be bipartite if and only if each of the $(K + 1)$ copies is bipartite; indeed, an optimal solution for G' has exactly $(K + 1)$ times the size of an optimal solution for G. We apply $\mathscr{A}$ to G'; since it returns a subset with at most K vertices more than in the optimal solution, it must have found a minimal subset for at least one of the copies.

Exercise 8.21

1. In order to "multiply" the problem here, we need to do more than merely copy the problem over and over, since the result would not even have an identification tree of any kind. We want more classes, because we want to feed to the claimed approximation algorithm a suitably larger instance. But we need all new classes tied to existing ones somehow, since all will be in the same tree. In particular, we need tests to distinguish not only between new classes in a "copy," but also tests to distinguish between "copies." Assume that the claimed algorithm guarantees a distance K approximation and let k be the smallest power of 2 such that $2^p = k > K$. We can solve an arbitrary instance $C, \mathscr{T}$ (where C is the set of classes and $\mathscr{T}$ the set of tests) exactly by using $\mathscr{A}$ as follows.
   We create a new instance $C', \mathscr{T}'$. C' is the disjoint union of $2^p = k$ copies of C; that is, if C equals $\{a, b \ldots, z\}$, then C' equals $\{a_1, a_2, \ldots, a_k, b_1, \ldots, b_k, \ldots, z_1, \ldots, z_k\}$. $\mathscr{T}'$ is a single copy of $\mathscr{T}$ where each class has been replaced by its k copies, plus p additional tests, call them $T_1, \ldots, T_p$, as follows.

$$T_1 = \{a_1, \ldots, a_{k/2}, b_1, \ldots, b_{k/2}, \ldots, z_1, \ldots, z_{k/2}\}$$

$$T_2 = \{a_1, a_2, \ldots, a_{k/4}, a_{1+k/2}, a_{2+k/2}, \ldots, a_{3k/4},\\ b_1, b_2, \ldots, b_{k/4}, b_{1+k/2}, b_{2+k/2}, \ldots, b_{3k/4}, \ldots, z_1, \ldots, z_{3k/4}\}$$

$$\ldots$$

$$T_p = \{a_1, a_3, a_5, \ldots, a_{k-1}, b_1, b_3, b_5, \ldots, b_{k-1}, \ldots, z_1, z_3, \ldots, z_{k-1}\}$$

Note that these p tests among themselves distinguish any copy from any other copy, but cannot distinguish any two elements in the same copy. We claim that an optimal identification tree for this new instance uses these p tests in the first p levels (all complete), then has $2^p = k$ copies of the optimal identification tree for the original instance hanging from level p; if the total path length of the optimal identification tree for the original instance with n objects was m, the total path length of this new tree is $M = km + knp$. (Equivalently, we can use the optimal identification tree for the original instance and replace each of its leaves by a complete tree of p levels using tests $T_1, \ldots, T_p$ to distinguish between the 2^p copies of each class associated with that leaf. Yet other optimal trees exist, but they all share the property that all p tests, $T_1, \ldots, T_p$, appear on every path from the root.) A simple, but tedious, induction proof verifies that claim.

Now we have the desired effect. Applying $\mathcal{A}$ to the new instance produces a tree with total path length not exceeding $M + K$; since every class exists in 2^p copies and since only the p tests T_i can distinguish between copies, all of them must appear on every path, so that we can assume that they make up the top p levels of the tree. (If they do not, we can move them up the tree until they do.) Now, each of the k subtrees hanging from level p is a copy of the original instance and its optimal solution requires a subtree of total path length m. Since the total path length of the tree created by $\mathcal{A}$ does not exceed $M + K = km + knp + K$ and since the top p levels account for the knp term, the k copies have a total cost of $km + K$, with $K < k$; since each copy costs at least m, one copy must cost exactly m and thus has been solved optimally.

2. The problem here is that a straightforward replication of the graph into $k + 1$ copies will produce a disconnected graph, one that does not have any spanning tree. We can either correct that problem by introducing additional edges to link the copies (somewhat difficult because of the degree bound) or avoid it altogether by scaling the edge costs rather than the entire graph. If we multiply the cost of each edge in the original graph by $k + 1$ we multiply the cost of the optimal solution by the same factor. Denote the cost of the optimal solution in the original problem by c; the cost of the optimal solution in the scaled problem is then $(k + 1)c$. The answer returned on the scaled problem by an approximation algorithm with guaranteed distance of k has cost at most $(k + 1)c + k$. In terms of the original problem, then the cost of the solution returned (the exact same tree, but now with

original edge costs) is at most $c + k/(k+1)$; since that cost must be an integer, it is in fact at most c and thus must be optimal.

3. We again need to be careful in "multiplying" the problem. Mere copies would not work, since colors would be reusable from copy to copy. We still want copies, but must connect them so as to prevent reuse in other copies of any color used in a copy. The solution is to connect the copies in the analog of a complete graph—a complete graph forces a different color for each of its vertices and our similar construction will force a different set of colors for each copy. In other words, we treat each copy like a vertex of some higher-level graph, which we make completely connected. This operation can be viewed as multiplying the complete graph on n vertices (for n copies) by the graph in hand, with the result that each vertex of the first graph is replaced by a copy of the second. Wherever the first graph had an edge $\{u, v\}$, we connect every vertex of the copy corresponding to u to every vertex of the copy corresponding to v.

   Denote this operation by $G' = K_n[G]$, where G is our original graph and K_n denotes, as usual, the complete graph on n vertices. We claim that, if the chromatic number of G is $\chi(G)$, then the chromatic number of G' is $\chi(G') = n\chi(G)$. Observe that no color used in one copy of G can be used in any other copy, since every vertex of one copy of G is connected to all vertices of all other copies (due to the fact that our pattern graph is a complete graph). But, if that's the case, whatever legal coloring is used for one copy (and it must take at least $\chi(G)$ colors) uses its own set of colors, so that the minimum number of colors needed for G' is $d\chi(G)$; it is easy to see that these many colors suffice.

   Now that we have established this claim, the proof follows the usual lines. Given a claimed approximation algorithm $\mathscr{A}$ with constant distance guarantee d, we solve an arbitrary instance $G = (V, E)$ of chromatic number as follows. We create the new graph $G' = K_{d+1}[G]$ and run $\mathscr{A}$ on it; it returns a coloring with at most $\chi(G') + d = (d+1)\chi(G) + d$ colors. Now, we know that each copy uses a different set of colors and that each copy requires at least $\chi(G)$ colors; so one of the copies must have been colored with exactly $\chi(G)$ colors.

Exercise 8.22

The method remains essentially unchanged, but the multiplication factor may need to be much larger.

In the case of the three problems of Exercise 8.20, the size of the optimal solution (the cardinality of a set cover, the number of satisfied clauses in

2SAT, and the number of vertices removed to make the graph bipartite) is trivially no larger than the size of the input. Thus the multiplier, which remains, as before, one more than the largest tolerable difference—here one more than the largest possible optimal solution—is (more or less) the size of the input, so that the multiplication, while no longer by a constant, is by a linear factor and thus remains feasible within polynomial time. The result immediately follows.

In the case of the three problems of Exercise 8.21, we have to be a bit more careful. The last one (chromatic number) is similar to the three just discussed, since the chromatic number of a graph cannot exceed its number of vertices. But the other two have potentially much larger solutions, since they involve separate weights. In the case of of the bounded-degree MST, the optimal solution is bounded by $|V| \cdot w_{\max}$, where $w_{\max}$ is the largest weight assigned to an edge, but the input is only of order $|V| + |E| \log w_{\max}$. Thus we cannot afford to multiply the instance by a factor on the order of $|V| \cdot w_{\max}$. The decision tree problem suffers from the same drawback. However, recall that both problems are in fact strongly NP-complete, so that we can consider only instances where the weights are polynomial functions of the number of vertices or classes and still have an NP-hard problem. The multiplication of the instance now works as before, since the factor, while much larger than constant, is still a polynomial.

Exercise 8.23

The key insight in the proof is to realize that the set of edges chosen by the algorithm forms a matching of the graph—i.e., no two edges chosen by the algorithm share a vertex. Since each edge must be covered and since the set of edges examined by the algorithm forms a matching, the minimum vertex cover must include at least one vertex for each such edge; but the cover found by the algorithm includes exactly two vertices for each such edge and thus has at most twice as many vertices as the optimal cover.

Exercise 8.24

As in the case of *VC*, a deceptively simple iterative algorithm actually works. The algorithm is very simple indeed: start from any cut at all—even from the empty cut into $(V, \emptyset)$ if so desired. Denote the current cut by $(S, V - S)$; while moving one vertex from S to $V - S$ or from $V - S$ to S increases the number of cut edges, carry out the move and update S appropriately. Once no more improvement can be made with the transfer of a single vertex, stop and return the current cut. This is clearly a polynomial-time algorithm—even with a very na ve implementation, each step takes at most $|V|^2$ time and, since we have $|E|$ edges in all and must improve the cut by at least 1 at each step, the number of steps is bounded by $|E|$, so that even a na ve implementation must run in $O(|E| \cdot |V|^2)$ time.

We claim the the cut returned by this algorithm cuts as least half as many edges as the optimal cut. Denote an optimal cut by $(T, V - T)$; let $S_1 = S \cap T$, $S_2 = S - T$, $S_3 = T - S$, and $S_4 = V - (S \cup T)$. Then we can write the approximate solution as $(S_1 \cup S_2, S_3 \cup S_4)$ and the optimal solution as $(S_1 \cup S_3, S_2 \cup S_4)$. Let us consider what we know about the number of edges between vertices in S_i and vertices in S_j, for $1 \leq i \neq j \leq 4$, call it n_{ij}. By construction the approximate solution cannot be improved by moving any single vertex across the cut; thus, in particular, if we consider a vertex in S_1, the number of edges connecting it to other vertices in S_1 and S_2 is no larger than the number of edges connecting it to vertices in S_3 and S_4. Summing this inequality over all vertices in S_1, we get

$$2n_{11} + n_{12} \leq n_{13} + n_{14}$$

Of course, the exact same reasoning can be made about the other three subsets, so we get in all

$$\begin{cases} 2n_{11} + n_{12} \leq n_{13} + n_{14} \\ 2n_{22} + n_{12} \leq n_{23} + n_{24} \\ 2n_{33} + n_{34} \leq n_{13} + n_{23} \\ 2n_{44} + n_{34} \leq n_{14} + n_{24} \end{cases}$$

Each of these numbers is at least 0, so we can drop the first term in each inequality and write

$$\begin{cases} n_{12} \leq n_{13} + n_{14} \\ n_{12} \leq n_{23} + n_{24} \\ n_{34} \leq n_{13} + n_{23} \\ n_{34} \leq n_{14} + n_{24} \end{cases}$$

Combining the four inequalities yields

$$n_{12} + n_{34} \leq n_{13} + n_{14} + n_{23} + n_{24}$$

Note that the right-hand side is exactly the number of edges cut in the approximate solution; the left-hand side is a subset of the edges cut by the optimal solution—we need to add in $n_{14} + n_{23}$. We write

$$n_{14} + n_{23} \leq n_{13} + n_{14} + n_{23} + n_{24}$$

which is obviously true (again because no number is negative) and add it to our previous inequality to yield

$$n_{12} + n_{14} + n_{23} + n_{34} \leq 2(n_{13} + n_{14} + n_{23} + n_{24})$$

or, in shorthand

$$\text{optimal} \leq 2 \cdot \text{approximate}$$

which proves the result.

Exercise 8.25

Clearly, if the optimal solution has no more than k objects in it, our algorithm will discover the optimal solution. Assume then that the optimal solution has over $n > k$ objects and consider the subset of k objects formed by the k most valuable objects in the optimal solution. This subset is one of the $\left(\binom{n}{k}\right)$ subsets examined by our algorithm; we claim that its greedy completion (through ordering by value density) yields a solution of value no less than $\frac{k}{k+1}$ times the optimal.

Denote the value of our subset of k objects by V and its size by S; denote the value of the optimal solution by $\hat{V}$. We know that the $n - k$ other elements in the optimal solution (each of value not exceeding that of the least valuable element in our subset) take up no more than $B - S$ space, so that at least one of these objects must have size at most $\frac{B-S}{n-k}$. The greedy completion attempts to pack the remaining $B - S$ space with the items that give the best value for their size; it will differ from the optimal solution only if it finds remaining objects of higher value density than the $n - k$ present in the optimal solution. In the worst case, one such object is found and its inclusion precludes that of any other object; in view of our previous comment, that means that less than $\frac{B-S}{n-k}$ space is left—or, equivalently, that the included object has size at least $(B - S)\frac{n-k-1}{n-k}$. Since its value density is as high as any in the $n - k$ other objects in the optimal solution, our greedy completion must have a value at least $\frac{n-k-1}{n-k}$ that of the optimal completion (which is, in our notation, $\hat{V} - V$). Overall, then, our solution (the greedy completion of the best k elements of the optimal solution) has value at least $V + \frac{n-k-1}{n-k}(\hat{V} - V)$, which we can simplify to $\frac{V+(n-k-1)\hat{V}}{n-k}$ Its ratio to the optimal solution is therefore at least $\frac{n-k-1}{n-k} + \frac{V}{(n-k)\hat{V}}$; since V represents the k best items of n items worth $\hat{V}$ in all, we must have $V \geq \frac{k}{n}\hat{V}$, so that our ratio is at least $1 - \frac{1-\frac{k}{n}}{n-k} = 1 - \frac{1}{n}$. The value n is not known in advance, but we have assumed $n > k$, so that we can write that the ratio is at least $1 - \frac{1}{k}$, as desired.

Exercise 8.26

Exercise 8.27

The perfect illustration of this theorem is *Knapsack*, where the feasibility of a solution is determined by the set of sizes and the bag size, the cost of a solution is determined by the set of weights (values), the optimal cost of any instance is easily bounded above by the product of the number of items by the value of the most valuable item, and a pseudo-polynomial algorithm exists that runs in time bounded by a polynomial in the number of items and the cost of the optimal solution. We used all these facts in deriving a fully polynomial-time approximation scheme for *Knapsack*, essentially by truncating the numbers representing the weights, solving the resulting problem exactly using the pseudo-polynomial algorithm, and showing that the resulting solution's cost in the original problem could be made arbitrarily close (in ratio) to the optimal by appropriate choice of truncation width.

Assume a maximization problem (minimization is the obvious symmetric) and assume that the cost function is non-negative when the q_is are non-negative. (This rather reasonable assumption in fact implies that the cost function is monotone non-decreasing.) Also assume that the optimal cost is no less than the largest cost-setting parameter, q. Let t be the truncation length; how to set it will soon become apparent. Then let $\hat{q}_i = \lfloor q_i/2^t \rfloor$; we have

$$\frac{q_i}{2^t} \geq \hat{q}_i \geq \frac{q_i}{2^t} - 1.$$

Let f^* be the optimal solution for the problem and let $\hat{f}$ be the optimal solution for the truncated problem. We get

$$c(q_i, f^*) \geq c(q_i, \hat{f}) \geq 2^t \cdot c(\hat{q}_i, \hat{f}) \geq 2^t \cdot c(\hat{q}_i, f^*) \geq c(q_i, f^*) - 2^t \cdot c(1, f^*).$$

The first inequality is due to the optimality of f^* in the original problem; the second derives from the cost function's monotonicity and linearity; the third is due to the optimality of $\hat{f}$ in the truncated problem; the last again comes from the monotonicity and linearity of the cost function. An immediate consequence is that

$$c(q_i, f^*) - c(q_i, \hat{f}) \leq 2^t \cdot c(1, f^*),$$

so that the performance ratio of our algorithm is bounded as follows:

$$\frac{c^* - \hat{c}}{c^*} \leq \frac{2^t \cdot c(1, f^*)}{c^*}.$$

But, by assumption, $c^* \geq q$ and $c(1, f^*) \leq p_1(|I|, 1)$ for some polynomial p_1, so that we get

$$\frac{c^* - \hat{c}}{c^*} \leq \frac{2^t \cdot p_1(|I|, 1)}{q},$$

and this last bound is the desired ε. Thus we set the truncation length to

$$t = \left\lfloor \log \frac{q \cdot \varepsilon}{p_1(|I|, 1)} \right\rfloor$$

Now our pseudo-polynomial algorithm runs in time $p_3(|I|, q)$, so that, on the truncated problem, it will run in time $p_3(|I|, \hat{q})$, that is, $p_3(|I|, \frac{p_1(|I|,1)}{\varepsilon})$; since $p_1(|I|, 1)$ is polynomial in $|I|$, our algorithm runs in time polynomial in $|I|$ and $1/\varepsilon$, as desired, and we are done.

Exercise 8.28

Recall that $len(I) = |I|$ is the length of the instance in a reasonable encoding, $f(I)$ is the value of the optimal solution for I, and $max(I)$ is the size of the instance in unary encoding.

A problem is p-simple if there exists a bivariate polynomial p such that instance I can be solved in time $p(len(I), f(I))$. The theorem we are to prove states that an NPO problem such that there exists polynomials q_1 and q_2 with $f(I) = O(q_1(max(I))$ and $max(I) = O(g_2(len(I), f(I)))$ is p-simple if and only if it can be solved in pseudopolynomial time.

Both parts fall to simple substitution.

If the problem can be solved in polynomial time, there exists some polynomial q such that instance I can be solve in $O(q(max(I))$ time. But then it can be solved in $O(q(g_2(len(I), f(I)))$ time; writing $p = q \cdot g_2$, we conclude that instance I can be solved in $O(p(len(I), f(I)))$ time, as desired.

Conversely, if instance I can be solved in $p(len(I), f(I))$ time, then, by assumption, it can be solved in $p(len(I), g_1(max(I)))$ time. But we have $max(I) \geq len(I)$ for all I, so we can write

$$p(len(I), g_1(max(I))) \leq p(max(I), g_1(max(I)))$$

so that there exists some polynomial q such that

$$p(max(I), g_1(max(I))) \leq q(max(I))$$

and thus instance I can be solved in $q(max(I))$ time, which is pseudopolynomial time.

Exercise 8.29

Exercise 8.30

Exercise 8.31

Exercise 8.32

We need to exhibit PTAS reductions to and from *Max3SAT*: one will establish membership of *MaxCut* in OptNP and the other will establish completeness. Recall that a PTAS reduction needs three distinct functions: a conventional mapping of instances f, a reverse mapping of solutions g, and a mapping of approximation ratios h.

To develop a PTAS reduction from *MaxCut* to *Max3SAT*, we set up a correspondence in which vertices become variables and encode every edge of the graph with a collection of clauses, in usch a way that every cut edge gives rise to a fixed proportion of satisfied clauses in that collection and every uncut edge gives rise to a lower number of satisfied clauses; in order to be able to define g, we also need to ensure that various truth assignments to the variables leave essentially no choice in the structure of the satisfied clauses. Given vertices u and v and edge $\{u, v\}$, we will have variables u and v and clauses $\{u, v\}$ and $\{\overline{u}, \overline{v}\}$. Since these are not in 3SAT form, we can use our standard expansion with one additional variable (a single one for the entire transformation), call it α, and thus obtain four clauses per edge, namely $\{u, v, \alpha\}$, $\{u, v, \overline{\alpha}\}$, $\{\overline{u}, \overline{v}, \alpha\}$, and $\{\overline{u}, \overline{v}, \overline{\alpha}\}$. Now observe that, if the edge $\{u, v\}$ is cut in a *MaxCut* solution, the corresponding truth assignment gives opposite truth values to u and v and thus satisfies all four clauses, whereas, if the edge is not cut, the corresponding truth assignment gives identical truth values to u and v and thus satisfies three of the four clauses. Conversely, in order to satisfy all four clauses, it must be the case that u and v are assigned opposite truth values; any other assignment satisfies three of the four clauses.

Thus we have a reduction from *MaxCut* to *Max3SAT* that starting from an instance $G = (V, E)$, produces an instance with $|V| + 1$ variables and $4|E|$ clauses and is such that a solution that cuts k edges gives rise to a truth assignment that satisfies $3|E| + k$ clauses and vice versa (any truth assignment handles clauses in blocks of 4 and satisfies either all 4 or just 3 in each block). In particular, if the optimal solution to an instance I of *MaxCut* has $\hat{k}$ edges cut, then the optimal solution to the instance $f(I)$ of *Max3SAT* has $3|E| + \hat{k}$ satisfied clauses and vice versa; also, if we obtain an approximation for $f(I)$ that is within a ratio $\epsilon < 0$ of optimal (i.e., it satisfies $\epsilon(3|E| + \hat{k})$ clauses), then the corresponding partition for I cuts

$\lfloor \epsilon \hat{k} \rfloor$ edges, meaning that the precision requirements are identical. Thus we have our f, g, and h, as desired, and we have proved that *MaxCut* is in OptNP.

To show completeness, we now need a PTAS reduction from *Max3SAT* to *MaxCut*. We already have, by transitivity, a polynomial-time transformation from the first to the second (viewed as decision problems) through the intermediate problem of *NAE3SAT*. Does this compound reduction have the desired properties of a PTAS reduction? The transformation from *3SAT* to *NAE3SAT* uses, for an instance of n variables and m clauses, $2n + 24m$ variables and produces $32m$ clauses. For each original clause, 32 clauses are produced and all of them are satisfied exactly when the original is; when the original clause is not satisfied, 2 of the 8 intermediate clauses produced are not satisfied and thus 2 of the 24 final clauses are not satisfied. So far, so good: since the numbers are fixed, no problem can arise. Finally, our reduction from *NAE3SAT* to *MaxCut* has similar properties: for an instance with n' variables and m' clauses, it produces a graph with $2n' + 3m'$ vertices and $n' + 6m'$ edges. Each satisfied clause corresponds to a cut of 5 edges and all n' variable edges are always cut; when a clause is not satisfied, only 3 edges are cut. Again, all values are fixed, so that the compound transformation is a PTAS reduction, with the identity for h and the obvious g.

Exercise 8.33

Exercise 8.34

Exercise 8.35

Exercise 8.36

Exercise 8.37

Exercise 8.38

Exercise 8.39

Solution Manual
Chapter 8, Additional Problems

Additional Problem 1

Denote by $scriptE_i$ the event that the ith clause is not satisfied. Since each clause has exactly k literals, the probability that a given clause is not satisfied is simply $p = 2^{-k}$. Satisfaction of one clause can depend only on other clauses that share a variable and there are at most $k \cdot 2^k/(ke)$ clauses that can share a literal with a given clause; thus we have $d + 1 = 2^k/e$. We can then verify that we have $p(d + 1) \leq 1/e$:

$$p(d + 1) = 2^{-k} \cdot 2^k/e = 1/e$$

By Lovasz' local lemma, we thus must have

$$\Pr[\bigcap \overline{\mathscr{E}_i}] > 0$$

which means that the probability that all clauses are simultaneously satisfied by a randomly chosen assignment is nonzero. Hence the instance is satisfiable.

Additional Problem 2

Since the associated decision problem is (trivially) strongly NP-complete, there can be no fully polynomial-time approximation scheme for the optimization problem. One can also prove that no polynomial-time approximation algorithm for *Decision Tree* can exist unless P = NP.

In order to "multiply" the problem here, we need to do more than merely copy the problem over and over, since the result would not even have an identification tree of any kind. We want more classes, because we want to feed to the claimed approximation algorithm a suitably larger instance. But we need all new classes tied to existing ones somehow, since all will be in the same tree. In particular, we need tests to distinguish not only between new classes in a "copy," but also tests to distinguish between "copies." Assume that the claimed algorithm guarantees a distance K approximation and let k be the smallest power of 2 such that $2^p = k > K$. We can solve an arbitrary instance $C, \mathscr{T}$ (where C is the set of classes and $\mathscr{T}$ the set of tests) exactly by using $\mathscr{A}$ as follows.

We create a new instance C', $\mathcal{T}'$. C' is the disjoint union of $2^p = k$ copies of C; if $C = \{a, b \ldots, z\}$, then $C' = \{a_1, a_2, \ldots, a_k, b_1, \ldots, b_k, \ldots, z_1, \ldots, z_k\}$. $\mathcal{T}'$ is a single copy of $\mathcal{T}$ where each class has been replaced by its k copies, plus p additional tests, call them $T_1, \ldots, T_p$, as follows.

$$T_1 = \{a_1, \ldots, a_{k/2}, b_1, \ldots, b_{k/2}, \ldots, z_1, \ldots, z_{k/2}\}$$

$$T_2 = \{a_1, a_2, ..., a_{k/4}, a_{1+k/2}, a_{2+k/2}, ..., a_{3k/4}, b_1, b_2, ..., b_{k/4}, b_{1+k/2}, b_{2+k/2}, ..., b_{3k/4}, \ldots, z_1, ..., z_{3k/4}\}$$

$$\ldots$$

$$T_p = \{a_1, a_3, a_5, \ldots, a_{k-1}, b_1, b_3, b_5, \ldots, b_{k-1}, \ldots, z_1, z_3, \ldots, z_{k-1}\}$$

Note that these p tests among themselves distinguish any copy from any other copy, but cannot distinguish any two elements in the same copy. We claim that an optimal identification tree for this new instance uses these p tests in the first p levels (all complete), then has $2^p = k$ copies of the optimal identification tree for the original instance hanging from level p; if the total path length of the optimal identification tree for the original instance with n objects was m, the total path length of this new tree is $M = km + knp$. (Equivalently, we can use the optimal identification tree for the original instance and replace each of its leaves by a complete tree of p levels using tests $T_1, \ldots, T_p$ to distinguish between the 2^p copies of each class associated with that leaf. Yet other optimal trees exist, but they all share the property that all p tests, $T_1, \ldots, T_p$, appear on every path from the root.) A simple, but tedious, induction proof verifies that claim. Now we have the desired effect. Applying $\mathcal{A}$ to the new instance produces a tree with total path length not exceeding $M + K$; since every class exists in 2^p copies and since only the p tests T_i can distinguish between copies, all of them must appear on every path, so that we can assume that they make up the top p levels of the tree. (If they do not, we can move them up the tree until they do.) Now, each of the k subtrees hanging from level p is a copy of the original instance and its optimal solution requires a subtree of total path length m. Since the total path length of the tree created by $\mathcal{A}$ does not exceed $M + K = km + knp + K$ and since the top p levels account for the knp term, the k copies have a total cost of $km + K$, with $K < k$; since each copy costs at least m, one copy must cost exactly m and thus has been solved optimally.

Clearly, the algorithm given cannot be optimal: we can run it in time $O(n^2 \cdot \log n)$ easily, so that its optimality would imply $\mathrm{P} = NP$, an unlikely outcome. One can fairly easily construct examples where the heuristic could construct trees with an external path length arbitrarily larger than optimal, in absolute values; it is much harder to construct an example where the ratio of approximate to optimal does not asymptotically reach 1; much harder yet to construct an example where the ratio can grow arbitrarily large. However, the latter is the correct worst-case behavior of the algorithm ($O(\log n/ \log\log n)$), as proved in Garey & Graham (*Acta Informatica* **3**, 74, pp. 347–355). It can nevertheless be shown that the algorithm is in the mean asymptotically optimal (see Moret, *ACM Computing Surveys* **14**, 4, pp. 595–623, for details).

Whether a polynomial-time approximation algorithm with a constant ratio guarantee exists remains an open problem.

Additional Problem 3

We shall show that, given an ε-approximation for *DCC*, we could decide the *HC* problem. Thus, suppose that we have an approximation algorithm for *DCC* such that the size of the optimal cover, $S*$, and that of the approximate solution, $\hat{S}$, satisfy $\hat{S}/S* \leq \varepsilon$. Given a graph, $G = (V, E)$ for the *HC* problem, we construct a new graph, G', on which we run our ε-approximation; depending on the size of the *DCC* obtained, we answer yes or no to the *HC* problem. We construct G' as follows.

Pick a vertex, $v \in V$, and let U_v be the set of vertices adjacent to v; call n_v its cardinality, i.e., $U_v = \{u_1, \ldots, u_{n_v}\}$. G' will consist of n_v unconnected subgraphs, $G_1, \ldots, G_{n_v}$. Let k be the smallest integer larger than ε, $k = \lfloor \varepsilon \rfloor + 2$. Then each G_i is composed of k isomorphic copies of the original G, together with a specific interconnection pattern, which links all k copies in a cycle. Specifically, G_i has extra edges $\{u_{i,j}, v_{j+1}\}$, where $u_{i,j}$ is the i-th vertex of U_v in the j-th isomorphic copy of G and v_{j+1} is the picked vertex v in the $j+1$-th isomorphic copy of G, and the extra edge $\{v_k, u_{i,1}\}$.

Now, since the k isomorphic copies are connected in a simple cycle which just replaces the original $\{u_i, v\}$ edge of G, no G_i has a Hamiltonian circuit unless G has one; moreover, if G does not have an *HC*, then the minimal *DCC* for each G_i has at least $k + 1$ cycles (since no two added edges can belong to distinct cycles). Conversely, if G has an *HC*, then this circuit must include one of the $\{u_i, v\}$ edges; then G_i has an *HC* (because each copy has a Hamiltonian path from u_i to v and we can use the additional edges to complete the circuit).

Thus, we now run our *DCC* approximation algorithm on each G_i in turn; if the size of the approximate *DCC* for any of them is no larger than k, we answer yes to the *HC* problem; if the size of the approximate *DCC* is larger than k for all of them, we answer no to the *HC* problem.

Note that there are as many possible Turing reductions for this problem as there are people devising such; the above is only one example—deliberately a bit different from our standard approach to such problems.

Additional Problem 4

1. First note that the sum of the intra-cluster and the inter-cluster dissimilitudes is the constant $\sum_{x,y} d(x, y)$. This shows that an optimal solution for one is an optimal solution for the other and vice-versa (thereby solving part (ii)); it also allows us to Turing-reduce one problem to the other. Thus it is enough to show that one of the two problems is NP-equivalent. Consider then *k-max clustering*; its decision problem can be restricted (by setting $k = 2$) to the known NP-complete problem *Max cut*; thus the optimization version is NP-hard.
2. This was proved above.
3. We show that the existence of any ε-approximate algorithm for *k-min clustering* would allows us to solve in polynomial time a known NP-complete problem, *graph k-coloring*. Specifically, given a graph, $G = (V, E)$, to be colored in k colors, we transform it to a k-min clustering problem as follows. The set of vertices becomes the set of elements; the dissimilitude measure is defined as

$$d(x, y) = \begin{cases} 2\varepsilon n^2 & \text{if } \{x, y\} \in E \\ 1 & \text{otherwise} \end{cases}$$

   Now, if the graph had a k-coloring, we can obtain a clustering of cost no larger than n^2 by putting in cluster i all elements corresponding to vertices colored i; if the graph did not have such a coloring, however, at least two elements corresponding to connected vertices will be in the same cluster, thereby guaranteeing a cost of at least $2\varepsilon n^2$. Our ε-approximate algorithm will return a solution of cost no worse than εn^2 in the first case, no better than $2\varepsilon n^2$ in the second (it cannot do better than optimal), so that we can distinguish the two cases and thus decide graph k-colorability.
4. A suitable choice for the greedy measure is the minimum intra-cluster dissimilitude—which is rather surprising in view of the result of part (iii). However, the choice is perfectly reasonable, since the two

measures are essentially the same as seen in part (ii). It remains to show that such an approximation algorithm has a ratio guarantee of $\frac{k-1}{k}$.

5. There are two reasons. One explains why one of the two problems can be hard to approximate while the other is relatively easy: their optimal values sum to a constant and are generally orders of magnitude apart, so that a small error in one becomes a large error in the other. The second explains why it is *k-min clustering* that is hard: it has the smaller optimal values.

Additional Problem 5

1. This is just a special case of the theorem.
2. Since L is NP-complete, there exists a polynomial time transformation, u, from *Clique* to L, such that $x \in$ *Clique* iif and only if $u(x) \in L$ and $u(x)$ runs in time $q(x)$. The problem, of course, is that u is not q-sparse; however, we can decide all but a subexponential number of instances in polynomial time using $\mathcal{A}$. Thus, we let the subexponential number of "hard" instances stand, but map all polynomially decidable instances to one of two instances. Define the new polynomial time transformation u' as follows:

$$u'(x) = \begin{cases} x_1 & \text{if } \mathcal{A}(u(x)) \text{ runs in time } p \circ q \text{ and } u(x) \in L \\ x_2 & \text{if } \mathcal{A}(u(x)) \text{ runs in time } p \circ q \text{ and } u(x) \notin L \\ u(x) & \text{otherwise} \end{cases}$$

   where $x_1 \in L$ and $x_2 \notin L$. Notice that the total number of different instances produced is subexponential; indeed, u' is $(f \circ q)$-sparse. Thus we can use Berman's theorem with $\max\{q(n), f \circ q(n)\}$ as the function f and u' as the transformation t. We conclude that the complexity of *Clique* (and thus of any NP-complete language) is polynomially related to $\max\{q(n), f \circ q(n)\}$; since q is a polynomial, then this is in turn polynomially related to $\max\{f(n), n\}$, which proves our theorem.
3. If a good approximation algorithm exists for some NP-complete problem, then the complexity of that problem (and thus of all other NP-complete problems) is subexponential, so that we have NP $\subseteq$ SubExp. Further, if the subexponential function is in fact polynomial, then P = NP.

Additional Problem 6

Additional Problem 7

Additional Problem 8

In other words, if *SAT* (or any other NP-complete problem) is solvable at all in randomized polynomial time, it is solvable in randomized polynomial time with one-sided error.

In order to ensure that only "yes" instances can be accepted, we shall resort to testing a truth assignment and accept only if the truth assignment satisfies the instance. In order to create the truth assignment to be tested (which has to be the "right" assignment with probability at least ε), we shall use the BPP algorithm to test the assignment of a truth value to each variable in turn.

Let our BPP solution algorithm for *SAT* be $\mathcal{A}$. This algorithm returns "yes" on "yes" instances with probability at least $1 - \delta$ and returns "no" on "no" instances with probability at least $1 - \delta$; by Theorem 8.29, we can make the δ any chosen constant by running the basic BPP algorithm $\Theta(\log \delta)$ times. In particular, we can run the algorithm on input x a linear number of times and thus make δ as small 2^{-n}, where n is the number of variables in instance x. Such an iterated algorithm, call it $\mathcal{B}$, when working on a "yes" instance, will reject it with probability at most 2^{-n} and, when working on a "no" instance, will accept it with probability at most 2^{-n}.

Now consider some unknown instance x; we shall use algorithm $\mathcal{B}$ to attempt to discover a satisfying truth assignment for x, accepting x only if such an assignment is found. Clearly, such a procedure will reject all "no" instances—we just need to ensure that it runs in polynomial time and accepts a "yes" instance with some minimum probability ε. We consider each variable in turn. Say we tentatively set x_i to true; we reflect that assignment in the instance x producing the new instance $x|_{x_i=1}$ and run algorithm $\mathcal{B}$ on $x|_{x_i=1}$. If the algorithm answers "yes," we permanently set x_1 to true, otherwise we permanently set it to false. Now we proceed the same way with the next variable, until we have assigned all variables. Once all variables are set, we check the resulting truth assignment and accept x if and only if that assignment satisfies x. This last step ensures that all "no" instances are always rejected.

Let us now compute the probability that the truth assignment derived in our procedure is a satisfying truth assignment when x is a "yes" instance. If x is a "yes" instance, then it has at least one certificate—one satisfying truth assignment. In the worst case, it has only one such assignment. In that case, our procedure assigns the wrong value to the first variable with probability at most $2 \cdot 2^{-n} = 2^{1-n}$; that is, it succeeds with probability at least $1 - 2^{1-n}$. The same is true at every step, since the number of iterations used by $\mathcal{B}$ is always the same. Thus our procedure succeeds in assigning all n variables with probability at least $(1 - 2^{1-n})^n$, which is at least $1 - n2^{1-n}$ and thus at least as large as $\frac{1}{4}$ for all $n \geq 3$. (For instances of one and two variables, we can trivially exhaust all truth assignments in polynomial time.)

Additional Problem 9

Additional Problem 10