Lecture Notes on Approximation Algorithms – Volume I

Rajeev Motwani*

Department of Computer Science
Stanford University
Stanford, CA 94305-2140.

*Part of this work was supported by NSF Grant CCR-9010517, and grants from Mitsubishi and OTL.

These lecture notes are based on the course CS351 (Dept. of Computer Science, Stanford University) offered during the academic year 1991-92. The notes below correspond to the first half of the course. The second half consists of topics such as *MAX SNP, cliques*, and *colorings*, as well as more specialized material covering topics such as *geometric problems*, *Steiner trees* and *multicommodity flows*. The second half is being revised to incorporate the implications of recent results in approximation algorithms and the complexity of approximation problems. Please let me know if you would like to be on the mailing list for the second half. Comments, criticisms and corrections are welcome, please send them by electronic mail to `rajeev@cs.stanford.edu`.

Contents

Chapter 1

Introduction

SUMMARY: *The notion of approximation algorithm is introduced and some motivation is provided for the issues to be considered later. Basic notation and some elementary concepts from complexity theory are presented. Two measures of goodness for approximation algorithms are contrasted: absolute and relative. Both positive and negative results are described for the following problems: scheduling, bin packing, and the traveling salesman problem.*

A large number of (if not, most of) the optimization problems which are required to be solved in practice are $\mathcal{NP}$-hard. Complexity theory tells us that it is impossible to find efficient algorithms for such problems unless $\mathcal{P} = \mathcal{NP}$, and this is very unlikely to be true. This does not obviate the need for solving these problems. Observe that $\mathcal{NP}$-hardness only means that, if $\mathcal{P} \neq \mathcal{NP}$, we cannot find algorithms which will find exactly the *optimal solution* to *all instances* of the problem in time which is *polynomial* in the size of the input. If we relax this rather stringent requirement, it may still be possible to solve the problem reasonably well.

There are three possibilities for relaxing the requirements outlined above to consider a problem well-solved in practice:

- **[Super-polynomial time heuristics.]** We may no longer require that the problem be solved in *polynomial time*. In some cases there are algorithms which are just barely super-polynomial and run reasonably fast in practice. There are techniques (heuristics) such as branch-and-bound or dynamic programming which are useful from this point of view. For example, the *Knapsack* problem is $\mathcal{NP}$-complete but it is considered "easy" since there is a "pseudo-polynomial" time algorithm for it. (We shall say more about this in Chapter 2.) A problem with this approach is that very few problems are susceptible to such techniques and for most $\mathcal{NP}$-hard problems the best algorithm we know runs in truly exponential time.

- **[Probabilistic analysis of heuristics.]** Another possibility is to drop the requirement that the solution to a problem cater equally to *all input instances*. In some applications, it is possible that the class of input instances is severely constrained and for these instances there is an efficient algorithm which will always do the trick. Consider for example the problem of finding Hamiltonian cycles in graphs. This is $\mathcal{NP}$-hard. However, it can be shown that there is an algorithm which will find a Hamiltonian cycle in "almost every" graph which contains one. Such results are usually derived using a probabilistic model of the constraints on the input instances. It is then shown that certain heuristics will solve the problem with very high probability. Unfortunately, it is usually not very easy to justify the choice of a particular input distribution. Moreover, in a lot of cases, the analysis of algorithms under assumptions about distributions is in itself intractable.

- **[Approximation algorithms.]** Finally, we could relax the requirement that we always find the *optimal solution*. In practice, it is usually hard to tell the difference between an optimal solution and a near-optimal solution. It seems reasonable to devise algorithms which are really efficient in solving $\mathcal{NP}$-hard problems, at the cost of providing solutions which in all cases is guaranteed to be only slightly sub-optimal

In some situations, the last relaxation of the requirements for solving a problem appears to be the most reasonable. This results in the notion of the "approximate" solution of an optimization problem. In this book we will attempt to classify as one of three types all hard optimization problems, from the point of view of approximability. Some problems seem to be extremely easy to approximate, e.g. Knapsack, Scheduling and Bin Packing. Other problems are so hard that even finding very poor approximations can be shown to be $\mathcal{NP}$-hard, e.g. Graph Coloring, TSP and Clique. Finally, there is a class of problems which seem to be of intermediate complexity, e.g. Vertex Cover, Euclidean TSP or Steiner Trees. In some cases we will be able to demonstrate that a problem is provably hard to approximate within some error.

1.1. Preliminaries and Basic Definitions

We first define an $\mathcal{NP}$-hard optimization problem and explore two notions of approximation. The following is a formal definition of a *maximization problem*; a *minimization problem* can be defined analogously.

Definition 1.1: *An* **optimization problem** Π *is characterized by three components:*

- **[Instances]** D*: a set of input instances.*
- **[Solutions]** $S(I)$*: the set of all feasible solutions for an instance* $I \in D$*.*
- **[Value]** f*: a function which assigns a value to each solution, i.e.* $f : S(I) \to \Re$*.*

A **maximization problem** Π *is: given* $I \in D$*, find a solution* $\sigma^I_{opt} \in S(I)$ *such that*

$$\forall \sigma \in S(I),\ f(\sigma^I_{opt}) \geq f(\sigma)$$

We will also refer to the value of the optimal solution as $OPT(I)$*, i.e.* $OPT(I) \triangleq f(\sigma^I_{opt})$*.*

We will abuse our notation a bit by sometimes referring to the optimal solution also as $OPT(I)$. The meaning should be clear from the context. The following example should help to flesh out these definitions.

BIN PACKING (BP): Informally, we are given a collection of items of sizes between 0 and 1. We are required to pack them into bins of unit size so as to minimize the number of bins used. Thus, we have the following minimization problem.

- **[Instances]** $I = \{s_1, s_2, \ldots s_n\}$, such that $\forall i, s_i \in [0,1]$.
- **[Solutions]** A collection of subsets $\sigma = \{B_1, B_2, \ldots B_k\}$ which is a disjoint partition of I, such that $\forall i, B_i \subset I$ and $\sum_{j \in B_i} s_j \leq 1$.
- **[Value]** The value of a solution is the number of bins used, or $f(\sigma) = |\sigma| = k$,

We would like to specify at the outset that an underlying assumption throughout this book will be that the optimization problems satisfy the following two technical conditions. This will be particularly important when we present complexity-theoretic results.

1. The range of f and all the numbers in I have to be integers. Note that we can easily extend this to allow rational numbers since those can be represented as pairs of integers. For example, in the Bin Packing problem we will assume all item sizes are rationals.
2. For any $\sigma \in S(I)$, $f(\sigma)$ is polynomially bounded in the size of any number which appears in I.

It is not very hard to see that the first condition is reasonable since no computer can deal with infinite precision real numbers. As for the second condition, we defer the justification and the motivation to Chapter 2.

We are only going to be concerned with $\mathcal{NP}$-complete optimization problems such as Bin Packing. Some people may find this concept slightly puzzling since normally the notion of $\mathcal{NP}$-completeness is applied to languages or *decision problems*. For example, when we say that Bin Packing is $\mathcal{NP}$-complete, it is understood that we are referring to the problem of deciding whether a given instance I has a solution of value at most K, where K is also specified as a part of the input. Therefore, we define the notion of $\mathcal{NP}$-hardness for optimization problems.

Definition 1.2: *If an $\mathcal{NP}$-hard decision problem Π_1 is polynomially reducible to computing the solution of an optimization problem Π_2, then Π_2 is $\mathcal{NP}$-hard.*

Typically, the problem Π_1 is the decision version of the problem Π_2. In other words, for a maximization problem Π_2, Π_1 is of the form: "Does there exist $\sigma \in S(I)$ such that $f(\sigma) \geq K$?"; however, this is not always the case. In fact, the above definition uses the more general notion of Turing reducibility and this permits a wider applicability of the term $\mathcal{NP}$-hardness. Refer to the book by Garey & Johnson [15] for a discussion of these issues.

Given an $\mathcal{NP}$-hard optimization problem Π, it is clear that we cannot find an algorithm which is guaranteed to compute an optimal solution in polynomial time for all input instances, unless $\mathcal{P} = \mathcal{NP}$. We now relax the requirement of optimality and ask for an approximation algorithm. This is defined as follows.

Definition 1.3: *An* **approximation algorithm** *A, for an optimization problem Π, is a polynomial time algorithm such that given an input instance I for Π, it will output some $\sigma \in S(I)$. We will denote by $A(I)$ the value $f(\sigma)$ of the solution obtained by A.*

A couple of remarks are in order. First, note that we are only interested in polynomial time algorithms and so this is built into the definition of an approximation algorithm. We will abuse notation and use $A(I)$ to denote both the value of the solution and the solution itself.

Consider, for example, the Bin Packing problem. Let DA (Dumb Algorithm) be an algorithm which packs each item into a bin by itself.

Clearly, this is an approximation algorithm for the problem BP. Of course it is not a very good approximation algorithm in the sense that the number of bins it uses need not be close to the optimal number of bins.

Thus, we need some way of comparing approximation algorithms and analyzing the quality of solutions produced by them. Moreover, the "measure of goodness" of an approximation algorithm must somehow relate the optimal solution to the solution produced by the algorithm. Such measures are referred to as *performance guarantees* and the exact choice of such a measure is not obvious *a priori*. We will explore several notions of performance guarantees in what follows.

What do you think is the most natural choice of such a measure?

1.2. Absolute Performance Guarantees

We know that packing a collection of items into the smallest possible number of bins is "impossible". So what is the next best solution that we could obtain? Clearly, this would be a solution which uses at most one extra bin when compared to the optimal solution. In general, it would be desirable to have a solution whose value differs from the optimal by some small constant. This is formalized in the *absolute performance measure.*

Definition 1.4: *An* **absolute approximation algorithm** *is a polynomial time approximation algorithm for* Π *such that for some constant* $k > 0$*,*

$$\forall I \in D, |A(I) - OPT(I)| \leq k$$

This is clearly the best we can expect from an approximation algorithm for any $\mathcal{NP}$-hard problem. But can we find such algorithms? We give below a couple of examples where such algorithms are possible to find.

1.2.1. Absolute Approximation Algorithms

Consider the problem* of coloring the vertices of a graph such that no two adjacent vertices have the same color. The goal is to minimize the number of colors used. The decision version of this problem is $\mathcal{NP}$-hard even when restricted to graphs that are *planar*. We now show that the planar graph coloring problem has an absolute approximation algorithm.

We first present the following theorem about the $\mathcal{NP}$-hardness of the planar graph coloring [15].

Theorem 1.1: *The problem of deciding whether a planar graph is 3-colorable is $\mathcal{NP}$-complete.*

It is also well-known that any planar graph is 5-colorable. In fact, the (in)famous Four Color Theorem for planar maps [2, 3] tells us that every planar graph is 4-colorable.

Consider the following approximation algorithm A for the planar coloring problem. It first checks if the graph is 2-colorable (or, bipartite) and computes the 2-coloring if possible. Otherwise, it just computes the obvious 5-coloring in polynomial time. It follows that A never uses more than 2 extra colors.

Do you know how check if a graph is bipartite?

Theorem 1.2: *Given any planar graph G, the performance of the approximation algorithm A is such that $|A(G) - OPT(G)| \leq 2$.*

Consider now the related problem of edge coloring. Here we have to color the edges of a graph with the smallest possible number of colors such that no two adjacent edges have the same color. The following theorem of Vizing [8] relates the maximum degree Δ to the edge coloring number.

Theorem 1.3: *Every graph needs at least Δ and at most $\Delta + 1$ colors to color its edges.*

*We will not explicitly specify the various components of optimization problems in the rest of the book.

In fact, the proof of Vizing's Theorem gives a polynomial time algorithm to actually find a coloring using $\Delta + 1$ colors. It is therefore amazing that even a very special case of the edge coloring problem is $\mathcal{NP}$-hard, as described in the following theorem of Holyer [26].

Theorem 1.4: *The problem of determining the number of colors needed for a 3-regular planar graph is $\mathcal{NP}$-hard.*

Putting all this together we can construct another absolute approximation algorithm for an $\mathcal{NP}$-hard optimization problem. The algorithm A just colors the input graph using $\Delta + 1$ colors as per Vizing's Theorem.

Theorem 1.5: *The approximation algorithm A has the performance guarantee $|A(G) - OPT(G)| \leq 1$.*

1.2.2. Negative Results for Absolute Approximation

One may conclude from the preceding examples that only a very special type of optimization problem can have an absolute approximation algorithm. These are problems where the value of the optimal solution can easily be pinned down within a small range, and the hardness of the problem lies in determining the exact value of the optimum solution within this range. An absolute approximation algorithm merely uses this information to give a trivial solution. It remains open whether some really interesting problem (i.e. one where the optimum value is not so easily pinned down) has an absolute approximation algorithm. Possibly the best candidate for such a result would be the Bin Packing problem.

But what if there is no such algorithm for Bin Packing? How do we go about proving that such an approximation is impossible? First note that if $\mathcal{P} = \mathcal{NP}$ then we can find the exact optimum for any $\mathcal{NP}$-complete problem. Thus, any hardness or impossibility result must be predicated upon the assumption that $\mathcal{P} \neq \mathcal{NP}$. It turns out that most optimization problems are hard to approximate in the sense that finding an absolute approximation is itself $\mathcal{NP}$-hard. The following two examples will help to illustrate this.

Let us first consider the KNAPSACK problem. An instance of the problem consists of:

- Items $I = \{1, \ldots, n\}$.
- Sizes $s_1, \ldots, s_n$ for each of the corresponding items.
- Profits $p_1, \ldots, p_n$ for each of the corresponding items.
- Knapsack capacity B.

A feasible solution to the problem is a subset $I' \subseteq I$ such that $\sum_{i \in I'} s_i \leq B$. We want to maximize $f(I') = \sum_{i \in I'} p_i$. More informally, we would like to pack some items of differing sizes into a knapsack of fixed capacity, so as to maximize the payoffs obtained from packing each item.

This problem is $\mathcal{NP}$-hard and so it is natural to try for an absolute approximation algorithm for it. Unfortunately, there exists no such algorithm unless there is a polynomial time algorithm which can find an *optimum solution.*

Theorem 1.6: *If $\mathcal{P} \neq \mathcal{NP}$ then no approximation algorithm can solve KNAPSACK with $|A(I) - OPT(I)| \leq k$, for any fixed k.*

Proof: We will prove this by contradiction using a scaling argument. Assume there exists an algorithm A with performance guarantee k which is a positive integer. We will show that this algorithm can be used to construct an optimum solution to any instance of Knapsack, thereby establishing the theorem.

Suppose we are given some instance I of Knapsack. We then construct a new instance I' such that $s_i' = s_i$ and $p_i' = (k+1)p_i$. In other words, we leave everything unchanged except the profits which are scaled up by a factor of $k+1$. It is easy to see that every feasible solution for I is also a feasible solution for I', and *vice versa.* The only difference is that the value of the solution for I' is $k+1$ times the value of the solution for I.

We now run the algorithm A on I' to obtain the solution $A(I')$. This gives us a solution σ for I. Clearly,

$$\begin{aligned} |A(I') - OPT(I')| &\leq k \\ \Rightarrow \quad |(k+1)f(\sigma) - (k+1)OPT(I)| &\leq k \end{aligned}$$

Recall that we are only dealing with integer values here. Upon dividing across by $k+1$ we get

$$\begin{aligned} |f(\sigma) - OPT(I)| &\leq \tfrac{k}{k+1} \\ \Rightarrow \quad |f(\sigma) - OPT(I)| &\leq 0 \end{aligned}$$

This, of course, means that we have found the optimal solution σ.
□

The key ingredient in the proof was the observation that KNAPSACK has a certain scaling property due to the linear dependence of the value function on some numbers in the input. It may seem that this will only be possible when the problem involves numbers in some crucial sense. As the next example shows, we can use "scaling" arguments in purely combinatorial problems which do not have any numerical aspect. But this relies on the notion of "graph products" which implicitly provides us with the required scaling.

Consider the CLIQUE problem. The problem is that of finding the largest clique (or, complete subgraph) in the input graph G. This is an $\mathcal{NP}$-hard problem. Note the problem is essentially the same as the MAXIMUM INDEPENDENT SET (MIS) problem. The following theorem establishes the hardness of approximating the largest clique.

Can you see why MIS and CLIQUE are related?

Theorem 1.7: *If $\mathcal{P} \neq \mathcal{NP}$, then there is no absolute approximation algorithm A for the CLIQUE problem.*

Proof: We first define the m-power of a graph G, say G^m, as follows. Take m copies of G and connect any two vertices which lie in different copies. We leave the proof of the following claim as an exercise.

Claim: The largest clique in G is of size α if and only if the largest clique in G^m is of size $m\alpha$.

Again, let us assume for the purposes of contradiction that the approximation algorithm A gives an absolute error of k. Then we claim that the clique problem can be optimally solved by the following strategy. Run A on G^{k+1}. If the largest clique in G is of size α, then we have that:

$$\begin{array}{rrcl} & |A(G^{k+1}) - OPT(G^{k+1})| & \leq & k \\ \Rightarrow & |A(G^{k+1}) - (k+1)OPT(G)| & \leq & k \end{array}$$

Now it is not very hard to see that given any clique of size β in G^m, we can find a clique of size $\frac{\beta}{m}$ in G in polynomial time. Thus, we can find a clique C in G such that

$$||C| - OPT(G)| \leq \frac{k}{k+1}$$

Since both $|C|$ and $OPT(G)$ are integer-valued, it follows that C must be an optimal clique.
$\square$

1.3. Relative Performance Guarantees

From the preceding section it is clear that, while absolute performance guarantees are the most desirable ones, it is quite unlikely that we can give such guarantees for any interesting class of hard optimization problems. Therefore it seems reasonable to relax the requirement for a "good approximation algorithm". We start by examining the problem of *multiprocessor scheduling* and use it to motivate the definition of *relative performance guarantees.* Interestingly enough, the whole field of approximation algorithms has its roots in the work of Graham [18] in 1966 on the problem of scheduling. In fact, scheduling problems probably have the most well-developed body of work from the point of view of approximation algorithms. In this book, however, we will not be able to cover most of these results and the reader is referred to the survey article by Lawler *et al* [40] for further details.

1.3.1. Multiprocessor Scheduling

Consider the simplest version of the multiprocessor scheduling problem. The input consists of n jobs, $J_1, J_2, \ldots, J_n$. Each job has a corresponding runtime $p_1, \ldots, p_n$, where each p_i is assumed to be rational. The jobs are to be scheduled on m identical machines or processors so as to minimize the finish time. The finish time is defined to be the maximum over all processors of the total run-time of the jobs assigned to that processor. The set of feasible solutions consists of all partitions of the n jobs into m subsets, and the value of a solution is the maximum over all subsets of the total run-time of the subset. The problem is known to be $\mathcal{NP}$-hard even in the case where $m = 2$.

Consider the following algorithm due to Graham which is called the *list scheduling* algorithm. The algorithm considers the n jobs one-by-one, assigning each job to one of the m machines in an online fashion. The rule is to assign the current job to that processor which is (at that point) the least loaded processor. Note that the load on a processor is the total run-time of all the jobs assigned to it.

Theorem 1.8: *Let A denote the list scheduling algorithm. Then, for all input instances I,*

$$\frac{A(I)}{OPT(I)} \leq 2 - \frac{1}{m}$$

Moreover, this bound is tight in that there exists an input instance I^* such that

$$\frac{A(I^*)}{OPT(I^*)} = 2 - \frac{1}{m}$$

Proof: Let us first prove the upper bound on the ratio. Assume, without loss of generality, that after all the jobs have been assigned the machine M_1 has the highest load. Let L denote the total run-time of all the jobs assigned to M_1. Also, let J_j denote the last job to be assigned to this machine.

We claim that every machine has a total load of at least $L - p_j$. This is because when J_j was assigned to M_1, M_1 was the least loaded processor with a load exactly $L - p_j$. It then follows that

$$\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$$

But it is also the case that

$$OPT(I) \geq \frac{\sum_{i=1}^{n} p_i}{m}$$

since some processor must have this much load at the end of the scheduling process. Since $A(I) = L$, we obtain that

$$OPT(I) \geq (L - p_j) + \frac{p_j}{m} = A(I) - \left(1 - \frac{1}{m}\right) p_j$$

Observing that $OPT(I) \geq p_j$ since some processor has to execute the job J_j, we obtain the desired result.

To see that the algorithm actually achieves this ratio, consider the following input instance I^*. Let $n = m(m-1)+1$ and let the first $n-1$ jobs have a run-time of 1 each, while the last job has $p_n = m$. It is easy to see that $OPT(I^*) = m$ while $A(I) = 2m - 1$. This gives the desired lower bound on the ratio. □

The interesting thing to note about this result is that we are measuring the quality of the approximation algorithm in terms of the ratio between the value of its solution and that of the optimal solution. This is exactly what we mean by a relative performance measure. The following definition formalizes this notion.

Definition 1.5: *Let A be an approximation algorithm for an optimization problem Π. The performance ratio $R_A(I)$ of the algorithm A on an input instance I is defined as*

$$R_A(I) = \frac{A(I)}{OPT(I)}$$

in the case where Π is a minimization problem. On the other hand when Π is a maximization problem we define the performance ratio as

$$R_A(I) = \frac{OPT(I)}{A(I)}$$

The ratio is defined differently for maximization and minimization problems so as to have a uniform measure for the quality of the solution produced by A. The ratio is always at least 1 and the algorithm produces a better approximation if the ratio is closer to 1. We now define the worst-case ratio for the algorithm A.

Definition 1.6: *The* **absolute performance ratio***, R_A, of an approximation algorithm A for an optimization problem Π is*

$$R_A = \inf\{r \mid R_A(I) < r, \forall I \in D\}$$

Applying these definitions to the list scheduling algorithm A, we have that $R_A = 2 - \frac{1}{m}$. Actually there is an even better approximation algorithm for the scheduling problem called LPT. This algorithm first orders the jobs by decreasing value of their run-times. After this, the algorithm behaves exactly the same as the list scheduling algorithm. Graham proved the following result for this new algorithm. We leave the proof as an exercise.

Theorem 1.9: *The LPT algorithm has a performance ratio of $R_{LPT} = \frac{4}{3} - \frac{1}{3m}$.*

In some problems, the absolute performance ratio is not the best possible definition of the performance guarantee for an approximation algorithm. This is because there may be input instances where the value of the optimal solution is very small, and the performance of the approximation algorithm differs only slightly from the optimal value. However, the small value of the optimum solution will make the ratio appear to be large. This is unreasonable since on larger instances the ratio is bounded by a small constant. We will see an example of such a problem in the next section. To take care of such anomalies, we will also define an asymptotic performance ratio.

Definition 1.7: *The* **asymptotic performance ratio***, R_A^∞, of an approximation algorithm A for an optimization problem Π is*

$$R_A^\infty = \inf\{r \mid \exists N_0,\ R_A(I) \leq r \textit{ for all } I \in D_\Pi \textit{ with } OPT(I) \geq N_0\}$$

We note that there is no difference between the absolute and asymptotic performance ratios of any approximation algorithm for scheduling. This is due to the scaling property of this problem. The scaling property is that we can multiply all the run-times by any large constant N, thereby scaling up the value of the optimal solution by N, without really changing the problem being solved. On the other hand, we will see that the approximative behavior of the Bin Packing problem changes dramatically when we move from the absolute to the asymptotic ratios. Most $\mathcal{NP}$-complete optimization problems do not have the scaling property.

Before we start proving bounds on the performance ratios of specific algorithms, it is useful to consider how such a bound may be derived in general. Assume without loss of generality that Π is a minimization problem. Then the proof of an upper bound on R_A for any algorithm A can be broken up into two parts. The first part is a proof of a lower bound on the value of $OPT(I)$ in terms of some parameters x. The second stage is to show that we can provide an upper bound on $A(I)$ in terms of x. To obtain the bound on the ratio, we merely eliminate x from these two inequalities. It is reasonably easy to see what the two parts of the proof need to be in the case where Π is a maximization problem and/or when proving a lower bound on R_A.

Can you identify these two parts of the proof in Theorem ?

1.3.2. Bin Packing

Recall the Bin Packing problem defined earlier. This problem is very closely related to the scheduling problem – they are *duals* of each other. Therefore, it is not very surprising that similar ideas crop up in devising approximation algorithms for these two problems.

We first consider the algorithm called *First Fit* or FF. This algorithm goes down the list of items and fits each item into the *first* bin where it will fit. More precisely, let us number the bins according to the time at each the first item was inserted into it. While trying to pack item i, FF successively tries to fit it into the already opened bins in this order. If no open bin has any room for the current item, then it opens a new bin and place item i in it.

Claim: For all instances I, $FF(I) < \lceil 2 \sum s_i \rceil$.

Proof: The proof is based on the observation that at most one bin is more than half empty at the end of the entire packing process. Suppose this is not the case. Let B_i and B_j be two bins which are more than half empty, such that $i < j$. Then the first item placed into bin B_j is of size at most 0.5. But this item would have fit into B_i and FF would not have opened the new bin B_j.

From this we conclude that total size of all the items is at least half of the number of bins used by FF. But the total size of all the items is also a lower bound on the value of the optimal solution. This gives the desired bound. □

Actually, much stronger bounds were obtained for the First Fit algorithm by Johnson *et al* [31] in 1974. They established the following result.

Theorem 1.10: $R^{\infty}_{FF} = 1.7$ *and more precisely we have the following bounds.*

- $\forall I, FF(I) \leq 1.7 OPT(I) + 2$
- $\exists I, FF(I) \geq 1.7(OPT(I) - 1)$

It is fairly easy to see an example where $FF(I) \geq \frac{5}{3} OPT(I)$. Consider the following instance I with $18m$ items. Here ϵ denotes a suitably small constant.

- $6m$ items of size $\frac{1}{7} + \epsilon$.
- $6m$ items of size $\frac{1}{3} + \epsilon$.
- $6m$ items of size $\frac{1}{2} + \epsilon$.

Can you see why there is no better packing?

It is clear that $OPT(I) = 6m$ – the optimal packing puts one item of each type into each bin. On the other hand, FF will distribute the items as follows.

- m bins with 6 items of size $\frac{1}{7} + \epsilon$ each.

- $3m$ bins with 2 items of size $\frac{1}{3}+\epsilon$ each.
- $6m$ bins with 1 item of size $\frac{1}{2}+\epsilon$ each.

A seemingly smarter heuristic is called *Best Fit* or BF. This puts each item into a bin where it fits the best. In other words, if the item fits into a bin which is already open, then it is placed into that bin where the empty space left over (after the current item has been added) is minimized. If no currently open bin can accommodate the current item then a new bin is opened for it. Quite surprisingly, Johnson *et al* showed that the BF algorithm also has an asymptotic performance ratio of 1.7.

In the lower bound example for FF it seems that the poor performance is due to the fact that all the small items are placed earlier in the list. A natural modification is to first sort the items in decreasing order of sizes, and then run the FF or BF algorithm. This is quite similar to the LPT modification to the list scheduling algorithm. Let us call the resulting algorithms FFD (First Fit Decreasing) and BFD (Best Fit Decreasing). Once again both algorithms have the same asymptotic ratio of $\frac{11}{9}$.

The proof of the upper bound for FFD or BFD is very involved (over 100 pages long!). However, it is easy to see that the bound of $\frac{11}{9}$ is achieved for the following input instance: $6m$ items of size $\frac{1}{2}+\epsilon$, $6m$ items of size $\frac{1}{4}+2\epsilon$, $6m$ items of size $\frac{1}{4}+\epsilon$, and $12m$ items of size $\frac{1}{4}-2\epsilon$. We leave the proof as an exercise.

Finally, we comment on the difference between the absolute and asymptotic performance ratios for the Bin Packing problem. The following theorem can be proved by using an input instance consisting of only seven items – the proof is again left as an exercise.

Theorem 1.11: $R_{FFD} \geq \frac{3}{2}$

Contrast this result with upper bound of 11/9 on the asymptotic ratio for FFD. This gives an example of an approximation algorithm with very different performance in terms of the two kinds of ratios.

1.3.3. The Traveling Salesman Problem

As a final example to illustrate the notion of performance ratios, we consider the famous problem of TSP. The input instance for TSP consists of a directed graph G with edge lengths $d(i,j)$, for all vertices i and j. Some of the edge lengths may be infinite, so we can assume that the graph is complete without any loss of generality. A feasible solution consists of a tour of the graph which visits every vertex exactly once. The goal is to find a tour of minimum length. We will only consider the symmetric version of the TSP, i.e. where $d(i,j) = d(j,i)$. Thus, we may restrict ourselves to the case of undirected graphs only. At this point we are interested in an even more special case of this problem called ΔTSP.

Definition 1.8: *The Metric Traveling Salesman Problem (ΔTSP) is the special case of the TSP where the input instances satisfy the triangle inequality. More precisely, for all vertices i, j and k,*

$$d(i,k) \leq d(i,j) + d(j,k)$$

Consider the following heuristic for ΔTSP called the *Nearest Neighbor* heuristic or NN. Starting at any vertex, construct a Hamiltonian path by going to the nearest unvisited vertex at each step. Finally, the cycle is completed by returning to the starting vertex. This is a natural heuristic but its performance is very poor as demonstrated by the following result due to Rosenkrantz *et al* [51].

Theorem 1.12: *Let n denote the number of vertices in an instance of ΔTSP. Then, $R^{\infty}_{NN} = \Theta(\log n)$*

However, it turns out that we can do much better by using more complex ideas. In fact, there are several heuristics known to achieve an asymptotic ratio of 2 [51]. Most of the good heuristics for ΔTSP are based on finding an Eulerian tour and then using "short-cuts" to obtain a Hamiltonian tour. We start by reviewing the notion of an Eulerian tour (refer to any standard graph theory book for more details).

Definition 1.9: *Let G be a multigraph. An Eulerian tour in G is a walk that visits every vertex at least once and each edge exactly once.*

Note that in a multigraph every edge can be repeated arbitrarily often. The following theorem characterizes the class of graphs which permit an Eulerian tour. Constructing such a tour in polynomial time is an easy consequence of the proof of this theorem.

Theorem 1.13: *A multigraph G has an Eulerian tour if and only if G is connected and all vertices are of even degree.*

Let us now consider the heuristic for ΔTSP based on the Minimum Spanning Tree (MST) in a weighted graph. The MST heuristic starts off by finding (in polynomial time) any MST for the graph G. It then constructs an Eulerian tour ET from the edges of T (using each edge exactly twice). The Eulerian tour yields a Hamiltonian cycle as follows. Starting at any vertex, visit the vertices in the order in which they are *first* visited in ET.

Algorithm MST:

Input: Graph $G(V, E)$ with distance function d.

Output: A Hamiltonian tour in G.

1. Find a minimum spanning tree T in G.

2. Construct a multigraph T' by making two copies of each edge in T.

3. Find an Eulerian tour ET in T'.

4. Construct a Hamiltonian tour by short-circuiting the Eulerian tour. That is, starting at any vertex, follow the Eulerian tour as long as new vertices are being visited. At any point where the Eulerian tour repeats a vertex, jump directly to the next unvisited vertex. Finally, complete the cycle by returning to the starting vertex.

Theorem 1.14: *The MST heuristic applied to ΔTSP has $R^{\infty}_{MST} = 2$.*

Proof: To prove correctness, it suffices to note that the graph T' is Eulerian since it is connected and all degrees are even.

Given any collection of edges H from G, denote by $d(H)$ the sum of all the edge lengths for the edges in H. We first claim the $d(T) \leq OPT(G)$. This is because any Hamiltonian cycle with an edge removed gives a spanning tree. Thus, we obtain that $d(ET) = d(T') \leq 2 \cdot OPT(G)$. Finally, the short-cut procedure ensures that $A_{MST}(G) \leq d(ET)$. This gives us an upper bound of 2 for the ratio.

Do you see why $A_{MST} \leq d(ET)$?

We leave the construction of an instance where this ratio is achieved by A_{MST} as an (easy) exercise.
□

It turns out that there is a modification to this heuristic which improves the performance ratio substantially. This is the heuristic due to Christofides [9] which we will refer to as CH. The basic idea is to avoid doubling the edges in going from the MST to an Eulerian graph. All we really need to do is to add a collection of edges which will increase the degree of every odd-degree vertex in the MST by exactly 1. This collection of edges is nothing but a matching on the odd-degree vertices.

Why does such a matching always exist?

Recall that a matching for a collection of vertices S in G is a subset of edges from G such that the set of end-points of these edges is exactly S, and each vertex in S has exactly one edge from the matching incident on it. Since G is complete, there exists a matching for every set S. Moreover, using standard results [38], the minimum-weight matching in G for S can be found in polynomial time.

It is relatively easy to modify the MST heuristic to incorporate the ideas presented above. We obtain the following result for Christofides heuristic.

Theorem 1.15: $R^{\infty}_{CH} = 1.5$

Proof: Let M be the minimum weight matching on the set O of odd degree vertices in the MST T. We claim that $d(M) \leq \frac{OPT(G)}{2}$.

To see this, consider the tour X obtained by taking short-cuts in the optimal solution so as to exclude all vertices which are not in O. The claim follows from the observations that $d(X) \leq OPT(G)$ and that the tour on O is the union of two matchings for O (consider the alternate edges in the tour). Thus, one of these two matchings has weight at most half that of the entire tour. Now the Eulerian tour ET is constructed in the graph $T \cup M$ and has weight at most $1.5 \cdot OPT(G)$. This gives the desired result. As usual, we leave as an exercise the construction of an example to show that this bound can be achieved.
□

This last heuristic is the best-known for ΔTSP. Note that the MST heuristic is very efficient since it runs in almost linear time. The heuristic due to Christofides is much more inefficient since finding a minimum weight matching [38] requires time $O(n^3)$. An interesting open problem is to find a *simple* construction of a class of algorithms which allows a smooth trade-off between the running time and the performance ratio. The results of Vaidya [57, 58] on exact and approximate minimum-weight matching (for points in the Euclidean plane) does give a trade-off, but it would seem that better results should be possible. Of course, improving the bound of 1.5 would be a major breakthrough! Another way of looking at the Euclidean TSP problem is: given n points in the plane, embed a Hamiltonian cycle on these points so as to minimize the total length of the embedded cycle. This can now be generalized to the embedding of any graph, and not just the Hamiltonian cycle. Interesting approximation results of this type can be found in the work of Bern *et al* [7] and Hansen [23].

1.3.4. Negative Results for Relative Approximation

We have seen several problems which permit good approximation algorithms under the relative performance measure. However, there are a large number of problems which do not exhibit this behavior. For example, in the GRAPH COLORING, CLIQUE or TSP problems we do not know of any algorithm which provides a performance guarantee that is substantially better than n, the number of vertices in the graph. It is desirable to come up with some explanation for why certain problems are easy to approximate and others are as intractable in their approximating versions as in the optimization version. Unfortunately, the theory of $\mathcal{NP}$-completeness does not provide any insight into this issue. There appears to be no connection between the approximate version of problems which are very closely related in their optimization versions (as all $\mathcal{NP}$-complete problems must be!).

A good example is provided by the problems of VERTEX COVER (VC) and MAXIMUM INDEPENDENT SET (MIS). Given a graph $G(V, E)$, a vertex cover is a set $C \subseteq V$ such that each edge in E has at least one end-point in C. The VC problem is to find a minimum cardinality vertex cover in the input graph G. An independent set in G is a set $I \subseteq V$ such that there are no edges between any pair of vertices in I. The MIS problem is to find a maximum cardinality independent set in G. An independent set is exactly the complement of a clique, so the MIS problem is the same as the CLIQUE problem in the complement graph.

Exercise 1.1: *Show that in every graph G, C is a vertex cover if and only if $I = V \setminus C$ is an independent set. Moreover, C is an optimal solution to VC if and only if $I = V \setminus C$ is an optimal solution to MIS.*

From this one might conclude that approximating VC and MIS are related problems. This is not the case! As we will see in Chapter 4, there is an approximation algorithm for VC with the ratio 2. On the other hand, we do not know of any approximation algorithm for MIS with a ratio significantly better than $|V| = n$. To see why the approximation of VC does not help in approximating MIS, let G be a graph with an optimal VC of size $\frac{n}{2} - 1$. Then we are guaranteed a vertex cover of size at most $n - 2$ by the approximation algorithm. Unfortunately, the complement of this vertex cover gives us an independent set of size only 2, as opposed to the optimal independent set of size $\frac{n}{2} + 1$.

Even though $\mathcal{NP}$-hardness reductions shed little light on the approximative behavior of optimization problems, it turns out we can use the theory of $\mathcal{NP}$-completeness to show that certain kinds of approximation algorithms do not exist unless $\mathcal{P} = \mathcal{NP}$. Let us first define the best possible performance ratio for a given optimization problem.

Definition 1.10: *We define* $R_{MIN}(\Pi)$*, the* **best achievable performance ratio** *for an optimization problem* Π*, as follows*

$$R_{MIN}(\Pi) = \inf\{r \geq 1 \,|\, \exists \textit{ poly-time algorithm } A \textit{ for } \Pi \textit{ with } R_A^\infty \leq r\}$$

The most desirable situation would be to have $R_{MIN} = 1$ for a problem Π. We see in the next few chapters that this can be achieved for problems such as KNAPSACK and BIN PACKING. These are the problems which are very easy to approximate. The next level of problems are those for which we can show that R_{MIN} is bounded, as in the case of ΔTSP. Finally, there are the really hard problems for which R_{MIN} is unbounded. In the rest of this chapter we examine a few problems of the latter type.

Consider the general TSP problem, i.e. without the triangle inequality. The following theorem due to Sahni and Gonzalez [55] shows that this is a really hard problem to approximate. Note that, as usual, the hardness of an approximation problem is predicated upon $\mathcal{P}$ and $\mathcal{NP}$ being different.

Theorem 1.16: *If* $\mathcal{P} \neq \mathcal{NP}$ *then* $R_{MIN}(TSP) = \infty$*.*

Proof: Assume that there is an algorithm A such that $R_A^\infty = K$, for some constant K. The proof idea is to use A to construct a polynomial time algorithm to solve HAM, the Hamiltonian cycle problem. Since the HAM is $\mathcal{NP}$-complete, we get a contradiction if $\mathcal{P} \neq \mathcal{NP}$.

Suppose we have an instance of HAM, i.e. an undirected and unweighted graph $G(V, E)$. We construct an instance I for the TSP as follows. Let H be a complete graph on the vertex set V. The length of an edge in H which is from E is set to 1, and the length of all other edges are set to Kn, where n is the cardinality of V. The following claim is easy to prove.

Claim: If G has a Hamiltonian cyle then $OPT(I) = n$. Otherwise, $OPT(I) \geq (K+1)n - 1$.

Consider what happens when we run the algorithm A on I. If G is Hamiltonian, then $A(I) \leq Kn$. Otherwise, $A(I) \geq OPT(I) \geq (K+1)n - 1$. Thus, the value of the solution found by A tells us whether G is Hamiltonian or not. In effect, we have given a polynomial time reduction from HAM to the approximate version of TSP. This contradicts the fact that HAM is $\mathcal{NP}$-complete, unless $\mathcal{P} = \mathcal{NP}$.
□

So far we have seen results which prove the impossibility of finding absolute approximation algorithms (e.g. for CLIQUE), and the above result shows that for the TSP there is no approximation algorithm with a bounded performance ratio. It is possible to devise algorithms whose performance lies somewhere in between these two kinds of approximation algorithms. For example, we will see shortly an approximation algorithm A for BIN PACKING where $|A(I) - OPT(I)| \leq \mathrm{O}(\log^2 OPT(I))$. Another example is the result of Lipton and Tarjan [42] where it is shown that there is an approximation algorithm A for finding maximum independent sets in *planar* graphs such that

$$|A(I) - OPT(I)| \leq \mathrm{O}\left(\frac{1}{\sqrt{\log\log OPT(I)}}\right) OPT(I)$$

using the planar separator theorem. Notice that such results imply that $R_A^\infty = 1$.

Given the possibility of such intermediate performance guarantees, it becomes interesting to prove impossibility results for such approximation algorithms. It is not very hard to modify the proof of Theorem 1.7 to obtain the following hardness result for CLIQUE. A series of such results have been obtained by Nigmatullin [47] and Kucera [37].

Theorem 1.17: *For all constants ϵ, $K > 0$, there is no approximation algorithm A for the CLIQUE problem such that*

$$|A(I) - OPT(I)| \leq K \cdot OPT(I)^{1-\epsilon}$$

Notice that this theorem does not rule out the possibility that there is some algorithm A for CLIQUE such that $R_A^\infty = 1$. (For example, one could obtain a result similar to that obtained for the case of planar graphs.) We do not know whether this is possible for CLIQUE but believe it to be extremely unlikely. Unfortunately, we do not know of any way of showing that the asymptotic ratio 1 is not achievable, besides showing that $R_{MIN} > 1$ if $\mathcal{P} \neq \mathcal{NP}$. The latter is strictly stronger result.

Assuming that the ratio 1 is not achievable for CLIQUE, we can show that no constant ratio is achievable either. This is a consequence of the following very curious theorem. Basically the result says that if that problem can be approximated within a specific constant factor, then it can be approximated within any constant factor. This is an example of a *self-reducibility* result for the approximation of an optimization problem. This result is usually interpreted as saying that CLIQUE is very hard to approximate as we do not believe that it has $R_{MIN} = 1$.

Theorem 1.18: *For the CLIQUE problem, either* $R_{MIN}(CLIQUE) = \infty$ *or* $R_{MIN}(CLIQUE) = 1$.

The idea behind this theorem is to use a notion of graph product to amplify the size of the optimal clique.

Definition 1.11: *The product of two graphs* $G_1(V_1, E_1)$ *and* $G_2(V_2, E_2)$ *is defined as a graph* $G(V, E)$ *such that*

- $V = V_1 \times V_2$.
- *an edge* $\{(u_1, u_2), (v_1, v_2)\}$ *is in* G *if either* $(u_1, v_1) \in E_1$ *or* $(u_1 = v_1$ *and* $(u_2, v_2) \in E_2)$.

We write $G = G_1 \circ G_2$ *and define* $G^N = G^{N-1} \circ G$.

Note that the product operation is non-commutative. We will need the following lemma whose proof is left as an exercise.

Lemma 1.1: *Let $OPT(G)$ denote the size of the largest clique in G. Then $OPT(G^N) = OPT(G)^N$ for all $N > 0$. Moreover, given any clique of size C in G^N, we can find in polynomial time a clique of size $C' \geq \lceil C^{\frac{1}{N}} \rceil$ in G.*

We are now ready to prove the theorem.

Proof: Assume that $R_{MIN}(CLIQUE) < \infty$. Then there exists an approximation algorithm A for the clique problem which has $R_A^\infty = r$, for a some constant r. We now fix any $\epsilon > 0$ and construct an algorithm A_ϵ such that $R_\epsilon^\infty < 1 + \epsilon$. This would imply the desired result.

The algorithm A_ϵ first chooses N such that $r^{\frac{1}{N}} < 1+\epsilon$. It then runs the algorithm A on G^N. Clearly, A finds a clique of size at least

$$\frac{OPT(G^N)}{r} = \frac{OPT(G)^N}{r}$$

in G^N. By the preceding lemma, this can be used to construct a clique of size at least

$$\frac{OPT(G)}{r^{\frac{1}{N}}} > \frac{OPT(G)}{1+\epsilon}$$

in G.

To see that the algorithm A_ϵ runs in polynomial time, observe that the graph product can be computed in polynomial time and N is a constant.

□

1.4. Discussion

We are following the notation of Garey and Johnson [15] which is now universally accepted. Their book is well-known as a good reference on the theory of $\mathcal{NP}$-completeness. It also provides a great introduction to the area of approximation algorithms, although it is quite a bit outdated in this respect. You could also refer to some of the other standard text-books on combinatorial algorithms [27, 48]. Unfortunately neither of these is up-to-date and they only provide a very cursory description of the work in this area. There are some survey articles on approximation algorithms [13, 30, 35] but again all of these are really old and outdated. A more recent article by Kannan and Korte [32] is much more useful.

Problems

1–1 Using the fact that every planar graph has a vertex of degree at most 5, show that all planar graphs are 5-colorable. How do you find such a coloring in polynomial time?

1–2 Consider the following variant of the traveling salesman problem called the *bottleneck TSP* problem. The goal is to find a Hamiltonian tour of the input graph so as to minimize the length of the *longest* edge in the tour. Assuming that the input graph satisfies the triangular inequality, show that this problem has a polynomial time approximation algorithm with ratio 3.

1–3 Consider the following generalization of the TSP called k-TSP which is defined for any fixed $k > 0$. Notice that the 1-TSP problem is exactly the ΔTSP.

Instance: Complete graph $G(V, E)$, with a distance function $d : E \to \Re^+$ which satisfies the *triangle inequality*.

Feasible Solutions: A collection of k subtours on G such that each subtour starts and ends at v_1, and all other vertex occur exactly once in exactly one of the subtours.

Goal: The objective is to minimize the total length of the k subtours.

Modify the Christofides Heuristic to solve this problem approximately and provide upper and lower bounds on its performance ratio.
(**Hint:** There is a polynomial time algorithm to find a minimum spanning tree T of G such that a specific node v has a specific degree d in T.)

1–4 Define the product of two graphs G_1 and G_2 as $G = G_1 \times G_2$, where G is obtained by replacing each vertex of G_1 by a copy of G_2 and putting all possible edges between two copies which correspond to adjacent vertices. Let G^N be the graph defined by the recurrence relation $G^{i+1} = G^i \times G$. Prove the following two claims.

- Let $OPT(G)$ be the size of the largest clique in G. Then, $OPT(G^N) = OPT(G)^N$.
- Given a clique of size C in G^N, we can construct in polynomial time a clique of size at least $\lceil C^{\frac{1}{N}} \rceil$ in G.

1–5 A *Hamiltonian walk* in a graph $G(V, E)$ is a closed walk that visits each vertex at least once. Let Π denote the optimization problem of finding a minimum length Hamiltonian walk.

a). Show that Π is $\mathcal{NP}$-complete.

b). What can you say about the hardness of approximating Π?

c). Construct (and analyze) the best approximation algorithm you can for Π.

1–6 The Edge-Disjoint Cycle Cover [ECC] problem is to find a collection of cycles in $G(V, E)$ which are edge-disjoint and which include every vertex at least once. Comment on the relationship between the optimization version of finding a cover by the smallest number of cycles and the Hamiltonian walk problem. Analyze the approximability of ECC by presenting positive and negative results.

Chapter 2

Approximation Schemes

SUMMARY: *The concept of an approximation scheme is defined and is illustrated by presenting such schemes for multiprocessor scheduling and the knapsack problem. This definition is then strengthened to that of fully polynomial approximation schemes and illustrated via the knapsack problem. It is observed that the existence of such schemes is intimately related to the existence of pseudo-polynomial time algorithms. The notion of strong $\mathcal{NP}$-completeness is presented and a connection is made with the existence of approximation schemes and pseudo-polynomial time algorithms.*

Recall the result for CLIQUE which states that either $R_{MIN} = 1$ or $R_{MIN} = \infty$ for that problem. We had said that this is a "hardness" result. Why should this not be viewed as an "easiness" result? After all, we have only to find any bounded-ratio approximation algorithm for CLIQUE, and then that can be turned into an approximation algorithm with a ratio arbitrarily close to 1. One reason for viewing this as hardness result is similar to the commonly held view of an $\mathcal{NP}$-completeness result as a hardness result. The existence of a bounded ratio algorithm for CLIQUE would imply a result which seems much too good to be true, given the lack of success in solving the problem so far. Another reason is that not too many problems seem to have algorithms which can achieve a ratio arbitrarily close to 1. We will try to provide some characterization of such problems in this part of the book.

Let us start by formalizing the notion of "having an algorithm which can achieve a ratio arbitrarily close to 1."

Definition 2.1: *An* **approximation scheme** *for an optimization problem* Π *is an algorithm* A *which takes as input both the instance* I *and an error bound* ϵ*, and has the performance guarantee*

$$R_A(I, \epsilon) \leq (1 + \epsilon).$$

Notice that we can view such an algorithm A as a family of algorithms $\{A_\epsilon : \epsilon > 0\}$ such that $R_{A_\epsilon} \leq 1 + \epsilon$. However, the definition of an approximation scheme has the stronger requirement that the entire (infinite) family of algorithms have a finite and uniform representation. This is a very good solution to a hard optimization problem and most people would consider a problem well-solved for all practical purposes if such an algorithm could be found. As we shall see, one can impose even stronger conditions on the approximate solution to a problem. Our convention has been to assume implicitly that an approximation algorithm must run in polynomial time. However, for the sake of tradition, we will make this explicit in the following definition.

Definition 2.2: *A* **polynomial approximation scheme (PAS)** *is an approximation scheme* $\{A_\epsilon\}$ *where each algorithm* A_ϵ *runs in time polynomial in the length of the input instance* I*.*

We would like to emphasize that the above definitions are made in terms of the *absolute* performance ratios, and not the *asymptotic* performance ratio. We will see later that this is a crucial difference. We now provide some examples of problems which permit of a PAS, viz. SCHEDULING and KNAPSACK.

2.1. Approximation Scheme for Scheduling

Recall the multiprocessor scheduling problem: Jobs $J_1, \ldots, J_n$ have run-times of $p_1, \ldots, p_n$. They are to be scheduled on m machines/processors so as to minimize the finish time. We have already seen some approximation algorithms with bounded ratios for this problem. We now present a PAS for this problem due to Graham [18].

Assume that $n > m$, and that the run-times are arranged in non-increasing order (i.e. $i < j$ implies that $p_i \geq p_j$). Note that the latter assumption can be easily fulfilled by sorting the jobs based on their run-times. Consider now the algorithm A_k which is defined for each integer $k \in [0, n]$.

Algorithm A_k:

Input: Runtimes of n jobs $\{p_1, \ldots, p_n\}$ and processor count m.

Output: A feasible schedule.

1. Schedule the first k jobs $J_1, \ldots, J_k$ optimally.

2. Starting with the partial schedule obtained in the previous step, schedule the remaining jobs greedily using the LPT rule.

Recall that the LPT rule picks the next largest unscheduled job and schedules it on a processor which has the least load currently. This algorithm clearly runs in polynomial time. As for its performance ratio, we have the following result due to Graham.

Theorem 2.1: $R_{A_k} \leq 1 + \frac{1-\frac{1}{m}}{1+\lfloor \frac{k}{m} \rfloor}$.

Proof: Let K denote the finish time of the schedule found in Step 1. Clearly, if $A_k(I) = K$ then this algorithm has found an optimal schedule and we are done. Assume now that the finish time of the total schedule is strictly greater than K. Then it must be the case that there is some job J_j with $j > k$ that is finished at time $A_k(I)$. This implies that all processors are busy during the time interval $[0, A_k(I) - p_j]$, since otherwise the job J_j would have been scheduled earlier on. (Notice that once a processor becomes idle, it remains idle till the end of the schedule.) Let $T = \sum_{i=1}^{n} p_i$ be the total run-time of the n jobs. We now conclude that

Why cannot a processor get a job after becoming idle?

$$T \geq m(A_k(I) - p_j) + p_j$$

Since the jobs are arranged in non-increasing order of run-times, we have that

$$T \geq mA_k(I) - (m-1)p_{k+1}$$

Observing that $OPT(I)$ is at least T/m, we have the following inequality.

$$A_k(I) \leq OPT(I) + \left(1 - \frac{1}{m}\right) p_{k+1} \tag{2.1}$$

If we can now show that p_{k+1} cannot be too large in terms of $OPT(I)$*, we will have the desired result. This may be established as follows. Consider the k largest jobs which were scheduled in Step 1. In an optimal schedule, some processor must be assigned at least $1 + \lfloor k/m \rfloor$ of these k jobs. Since each of these has run-time at least as large as J_{k+1}, we conclude that

$$\begin{array}{rrcl} & OPT(I) & \geq & \left(1 + \left\lfloor \frac{k}{m} \right\rfloor\right) p_{k+1} \\ \Rightarrow & p_{k+1} & \leq & \frac{OPT(I)}{1 + \left\lfloor \frac{k}{m} \right\rfloor} \end{array}$$

Combining this with equation 2.1, we have the desired result.
□

We can now extract the promised PAS from the above result. Let A_ϵ, for any $\epsilon > 0$, be the algorithm A_k with k chosen such that the

*The proof of an upper bound on $R_A(I)$ usually consists of two parts. First, there is an upper bound on $A(I)$, possibly in terms of $OPT(I)$ and some parameter X; then there is a lower bound on $OPT(I)$, possibly in terms of $A(I)$ and X. Eliminating X from these two inequalities gives the upper bound.

performance ratio of A_k is at most $1 + \epsilon$. Verify that this will be the case provided $k \geq \frac{1-\epsilon}{\epsilon} m$. However, we have left out one crucial detail in the description of the algorithm A_k. Exactly how does Step 1 get implemented? It is not very hard to see that there is a brute-force algorithm which compute an optimal schedule in time $O(m^k)$, for k jobs on m processors. The running time of this step is polynomially bounded in the length of I for sufficiently small values of m, say for constant m. We have established the following theorem.

Exactly how large can m be without making the time super-polynomial?

Theorem 2.2: *For fixed m, there is a polynomial approximation scheme for the m-processor scheduling problem.*

Notice that this algorithm is by no means a practical algorithm even for relatively small value of m. The running time is exponential in $1/\epsilon$ and so we cannot ask for ratios arbitrarily close to 1 without excessively increasing the running time. It is instructive to compute the running time of A_ϵ for small values of ϵ. For example, figure out the running time when $m = 10$ and $\epsilon = 0.1$.

We would like to point out that this trade-off between the running-time and the quality of the approximation obtained is an important feature of any approximation scheme. In general, we would like the trade-off to be such that the running time does not increase too fast with a decrease in the performance ratio. We will see this feature in the approximation schemes presented in the next few sections.

2.2. Approximation Scheme for Knapsack

In the KNAPSACK problem we are required to find a subset of the specified items such that the total size of the subset does not exceed the knapsack capacity, while maximizing the sum of the payoffs associated with the items. More formally,

Instances: Set $U = \{u_1, \ldots, u_n\}$ of items where each item u_i has a *size* s_i and a *profit* p_i associated with it. The capacity of the knapsack, B, is also specified as a part of the input.

Solutions: Subset $U' \subseteq U$ such that $\sum_{u_i \in U'} s_i \leq B$.

Value: The value of a solution is the total profit, $\sum_{u_i \in U'} p_i$, of the items packed into the knapsack. The goal is to maximize the net profit.

As usual we assume that the numbers involved in the input instance are non-negative rationals. There is no loss of generality in requiring that each item has size at most B. The following greedy algorithm GA is an obvious approximation algorithm for knapsack. The idea is to consider the items one-by-one in the order decreasing profit to size ratio. Each item is inserted into the knapsack if adding it does not cause the set of current items to exceed the knapsack capacity.

Algorithm GA:

Input: The knapsack size B, the item sizes $\{s_1, \ldots, s_n\}$ and the profits $\{p_1, \ldots, p_n\}$.

Output: Subset of the items of total size at most B.

1. Sort the items in non-increasing order of their profit densities p_i/s_i. At this point, we have that if $i < j$ then $p_i/s_i \geq p_j/s_j$;

2. $U' \leftarrow \emptyset$;

3. **for** $i = 1$ to n **do begin**
if $\sum_{j \in U} s_j \leq B - s_i$ **then** $U \leftarrow U + i$;

end .

Unfortunately, this natural greedy algorithm does not do well in the worst case. For example, consider the case where there are only two items – the first has size 1 and profit 2, while the second has size B and payoff B. Surprisingly, there is a very simple modification to GA which substantially improves its performance. Let MGA be the Modified Greedy Algorithm which picks the better of the solutions provided by GA and the best solution obtained by packing just one item, i.e. the item with the largest profit, into the knapsack. We leave the proof of the following theorem as an easy exercise.

What is R_{GA}?

Theorem 2.3: $R_{MGA} = 2$

In 1975, Sahni [53] came up with a PAS for this problem. The basic idea was quite similar to the one used for the scheduling problem. For all k, $0 \leq k \leq n$, define the algorithm A_k as follows. First the algorithm A_k chooses a subset S of at most k items as being in the knapsack initially. Then it runs the algorithm GA using the remaining items. This process is repeated for all possible choices of the k-set S. This paradigm is generally referred to as *k-enumeration* for obvious reasons. Almost every PAS that has been devised is based on this approach. The application of this idea to knapsack gives the following result.

Theorem 2.4: *For all k, A_k has a performance ration $R_{A_k} \leq 1 + \frac{1}{k}$ and runs in time $\mathrm{O}(kn^{k+1})$.*

Proof: The number of subsets of size at most k is $\mathrm{O}(kn^k)$. For each such subset the amount of work done by A_k is $\mathrm{O}(n)$, implying the bound on the running time. We now turn our attention to the performance ratio for A_k. It can be shown that the bound on the performance ratio is tight, the construction of an input instance for this is left as an exercise.

Let us fix our attention on any one optimal solution, say $X \subseteq U$. If $|X| \leq k$, then it is obvious that A_k will find an optimal solution and we are done. Assume then that $x = |X| > k$. Let the items $Y = \{v_1, \ldots, v_k\} \subset X$ be the items with the largest profits in X. The remaining items $Z = X \setminus Y = \{v_{k+1}, \ldots, v_x\}$ all have smaller profits than the items in Y, and are assumed to be numbered in the order of decreasing profit density p_i/s_i.

The algorithm A_k will try Y as the initial k-set at some point. We are only interested in this one iteration of the algorithm. After initializing the knapsack to Y, the algorithm will greedily try to fit into the knapsack all the remaining items one-by-one in the order of their profit densities. Define m as the index of the first item in the set Z which is not placed into the knapsack by the algorithm A_k, i.e. items v_{k+1}, $\ldots$, v_{m-1} are placed into the knapsack.

The reason that the item v_m did not get placed in the knapsack is that the remaining empty space at that point, say E, was smaller than s_m. At the time when v_m is rejected the knapsack contains the items from Y, the items $v_{k+1}, \ldots, v_{m-1}$ and some items which are not present in the optimal set X. Let G denote the items that are placed into the knapsack *so far* by the greedy stage of A_K. (These are all the items added to the knapsack up to this point that are not in Y.) It is clear that the items in $G \setminus X$ are of total size $\Delta = B - E - \sum_{i=1}^{m-1} s_i$. Moreover, each of these items has profit density at least p_m/s_m since they were considered earlier than v_m by the greedy stage of A_k. It then follows that[†]

$$profit(G) \geq \sum_{i=k+1}^{m-1} p_i + \Delta \frac{p_m}{s_m}$$

We can now write $profit(X)$, the net profit of the optimal solution X, as follows.

$$\begin{aligned}
& \sum_{i=1}^{k} p_i + \sum_{i=k+1}^{m-1} p_i + \sum_{i=m}^{x} p_i \\
\leq \;& profit(Y) + \left(profit(G) - \Delta \frac{p_m}{s_m}\right) + \left(B - \sum_{i=1}^{m-1} s_i\right) \frac{p_m}{s_m} \\
= \;& profit(Y) + profit(G) + E \frac{p_m}{s_m} \\
\leq \;& profit(Y \cup G) + p_m
\end{aligned}$$

Since the solution produced by A_k is a superset of $Y \cup G$, we get $OPT(I) - A(I) \leq p_m$. Noting that there are at least k items with a higher profit than p_m in X, viz. the items in Y, we have that $p_m \leq \frac{profit(X)}{k+1}$. This completes the proof.
□

To obtain a PAS for KNAPSACK, let A_ϵ be the algorithm A_k with $k = \left\lceil \frac{1}{\epsilon} \right\rceil$.

[†]We will use $profit(T)$ and $size(T)$ to denote the total profit and size, respectively, of the items in the subset T.

Corollary 2.1: *There exists a PAS for KNAPSACK where the algorithm A_ϵ runs in time $n^{O(\frac{1}{\epsilon})}$.*

2.3. Fully Polynomial Approximation Schemes

Consider the running times of the algorithm A_ϵ in the PAS presented above for KNAPSACK and SCHEDULING – the running times are really enormous even for reasonable values of ϵ. The definition of fully polynomial approximation schemes is designed to remedy this shortcoming in the definition of a PAS.

Definition 2.3: *A* **fully polynomial approximation scheme** *(FPAS) is an approximation scheme $\{A_\epsilon\}$ where each algorithm A_ϵ runs in time polynomial in the length of the input instance I and $1/\epsilon$.*

Assuming that $\mathcal{P} \neq \mathcal{NP}$, the existence of a FPAS is the best one can hope for in the case of an $\mathcal{NP}$-complete problem. Not surprisingly, there are very few $\mathcal{NP}$-complete problems which permit of an FPAS. One problem for which we can demonstrate such a scheme is KNAPSACK. The basic idea behind the FPAS for KNAPSACK is prototypical of most FPAS that are known.

Let PP be an algorithm for the *exact* solution of KNAPSACK which runs in time $O(n^3 P \log SB)$, where $P = \max_i p_i$ and $S = \max_i s_i$. From this we construct an approximation algorithm A_K for KNAPSACK as follows.

This does not imply that $\mathcal{P} = \mathcal{NP}$ even though the running time appears to be polynomial. Do you see why not?

Algorithm A_K:

Input: Knapsack instance I with the profits p_i, sizes s_i, and knapsack capacity B.

Output: Subset of items of total size at most B.

1. Construct an input instance I_K with new profits $p'_i = \lfloor p_i/K \rfloor$, while leaving everything else unchanged.

2. Run algorithm PP on the instance I_K to obtain a subset of the items S which has total size not exceeding B.

3. **return** S.

We derive an algorithm A_ϵ from A_K by setting $K = \frac{P}{\left(\frac{1}{\epsilon}+1\right)n}$. This gives us an FPAS for KNAPSACK as proved in the following theorem [28]. The running time is polynomial in both the length of the input and the inverse of ϵ.

Theorem 2.5: *The algorithm A_ϵ runs in time $\mathrm{O}\left(\frac{n^3 \log SB}{\epsilon}\right)$ and has $R_{A_\epsilon} \leq 1+\epsilon$.*

Proof: The running time is easily obtained from the above definitions. As for the performance ratio, first observe that

$$OPT(I) - K \cdot OPT(I_K) \leq Kn$$

which implies that

$$OPT(I) - A_K(I) \leq Kn$$

Note also that $OPT(I) \geq P$. We can now derive the desired bound as follows,

$$\begin{aligned} R_{A_\epsilon}(I) &\leq \frac{A_\epsilon(I) + Kn}{A_\epsilon(I)} \\ &= 1 + \frac{Kn}{A_\epsilon(I)} \\ &\leq 1 + \frac{Kn}{OPT(I) - Kn} \\ &\leq 1 + \frac{Kn}{P - Kn} \\ &\leq 1 + \epsilon \end{aligned}$$

□

Basically, this FPAS starts off with a slow but exact algorithm for KNAPSACK, and then trades accuracy for speed by means of a "scaling" technique. Of course, we have yet to specify the algorithm PP; this is dealt with in the next section.

2.4. Pseudo-Polynomial Algorithms

The algorithm PP that we promised in the last section is an example of a *pseudo-polynomial algorithm*. This is a class of algorithms which runs in time that is polynomial in the *size* of the numbers involved in an input instance. Note that the usual definition of the length of an input depends logarithmically in the size of the numbers used. Thus, such algorithms are not really polynomially bounded in the length of the input. In this section we present a brief introduction to these notions. Refer to the book by Garey and Johnson [15] for a more thorough treatment of these concepts. We will also point out the connections between such algorithms and the construction of FPAS.

We start off by defining the notion of a *number problem*. A combinatorial optimization problem consists of two components – a combinatorial component and a numerical component. The former refers to structures which are purely combinatorial in nature, such as graphs and set systems. This component can be thought of as structures which are made up of atoms drawn from some bounded domains; these can be encoded as vectors over a finite domain. The latter can be thought of as numbers which are drawn from some unbounded domain such as integers or rationals. We now define the following two functions which measure the size of the encoding of an input instance assuming some "reasonable" encoding scheme.[‡]

Definition 2.4: *Given any optimization problem* Π *and a reasonable encoding of the input instance for* Π*, we define* $LENGTH(I)$ *and*

[‡]We are presenting only an intuitive development here and the reader should refer to [15] for a more formal treatment.

$MAX\#(I)$ as functions which map the input instances into positive integers. The $LENGTH$ function measures the combinatorial size of the input, and the $MAX\#$ function measures the size of the largest number used in the encoding.

For example, in the case of the KNAPSACK problem, we have $LENGTH(I) = n$ and $MAX\#(I) = \max\{P, S\}$. We can now present a more formal definition of polynomial time and pseudo-polynomial time algorithms.

Definition 2.5: *A* **polynomial time** *algorithm A for Π runs in time which is polynomially bounded in $LENGTH(I)$ and $\log MAX\#(I)$ for each input instance I.*

Definition 2.6: *A* **pseudo-polynomial time** *algorithm A for Π runs in time which is polynomially bounded in $LENGTH(I)$ and $MAX\#(I)$ for each input instance I.*

Thus, our usual definition of an efficient algorithm refers to the former class of algorithms, while the algorithm PP for KNAPSACK belongs to the latter class of algorithms. Let us illustrate the latter definition by providing a pseudo-polynomial algorithm for the PARTITION problem. This problem is $\mathcal{NP}$-complete and is defined as follows. An input instance consists of n positive integers, s_1, ..., s_n, and a bag size B. A feasible solution solution consists of a subset $X \subseteq [1, \ldots, n]$ such that $\sum_{i \in X} s_i = B$. The decision version of this problem is closely related to the BIN PACKING, SCHEDULING and KNAPSACK problems.

Consider the algorithm DP [14] for PARTITION which is based on the paradigm of Dynamic Programming. The basic idea behind DP is to construct the table $T_{i,j}$, for $1 \leq i \leq n$ and $0 \leq j \leq B$. The entries in the table are boolean values such that $T_{i,j} = TRUE$ if and only if there exists $Y \subseteq [1, \ldots, i]$ such that $\sum_{l \in Y} s_l = j$. This table T is constructed in a row-by-row fashion as follows.

Algorithm DP:

Input: Bag size B and item sizes $\{s_1, \ldots, s_n\}$.

Output: Table T.

1. for all $1 \leq i \leq n$ **do** $T_{i,0} \leftarrow TRUE$;

2. for all $0 \leq j \leq B$ **do** $T_{1,j} \leftarrow (j = s_1)$;

3. for $j = 1$ to B **do** $T_{i+1,j} \leftarrow T_{i,j} \vee T_{i,j-s_{i+1}}$;

A simple induction proof establishes that this algorithm will correctly compute the table T. To decide the problem of PARTITION, it suffices to check the entry $T(n, B)$. Actually, the algorithm can be easily modified to solve the search problem of computing a set X. This can be done by storing the set $X_{i,j}$ at the position $T_{i,j}$ such that $X_{i,j} \subseteq [1, \ldots, i]$ and $\sum_{l \in X_{i,j}} s_l = j$. Note that there may not be a unique $X_{i,j}$ but it suffices to store any one set at each position in T. We obtain the following theorem which implies that DP is a pseudo-polynomial time algorithm for PARTITION.

Do you see how to do this?

Theorem 2.6: *The algorithm DP solves the search version of the PARTITION problem in time* $O(n^2 B)$.

We now show how to devise the pseudo-polynomial time algorithm PP for KNAPSACK using some of the ideas from DP. Our goal is now to find a subset $U' \subseteq U$ of total size at most B so as to maximize the $profit(U')$. The obvious modification to DP would be to store a set $X_{i,j}$ at each each table entry $T_{i,j}$ as before, just ensuring that we pick the set of maximum possible profit out of all the sets which are candidates for being $X_{i,j}$. Unfortunately, this does not work and there are two problems which crop up.

Observe the close relation between PARTITION and KNAPSACK.

The first problem is that we get a running time that depends exponentially on the length of B, while the scaling argument of the previous section works only permits the running time of the algorithm PP to depend exponentially on the length of P. Secondly, it is not clear that picking the set $X_{i,j}$ of the largest profit is the right choice of a partial solution. It may be the case that only a lower profit subset of $[1 \ldots i]$ can be extended to an optimal solution.

The way to fix the first problem is to have the columns of T as being in correspondence with the profits of the sets rather than their sizes. Note that the maximum profit of any set is nP, and in particular we have that $OPT(I) \leq nP$.

Definition 2.7: *The boolean table T consists of entries $T_{i,j}$, for $1 \leq i \leq n$ and $0 \leq j \leq nP$, such that $T_{i,j} = TRUE$ if and only if there exists a set $X_{i,j} \subseteq [1, \ldots, i]$ with $size(X_{i,j}) \leq B$ and $profit(X_{i,j}) = j$.*

In this definition, the entry $T_{i,j}$ tells us if there is a subset of the first i items which is a feasible solution to KNAPSACK and has value j.

The second problem that we had mentioned can be easily handled once we have the following fact. The proof is obvious.

Lemma 2.1: *Let X, $Y \subseteq [1, \ldots, i]$ be such that $profit(X) = profit(Y) = j$ and $size(X) \leq size(Y)$. Then, for all $Z \subseteq [i+1, \ldots, n]$, it is the case that $size(X \cup Z) \leq size(Y \cup Z)$.*

This means that if any candidate set $X_{i,j}$ can be extended to an optimal solution, then a candidate set of the smallest size can be so extended. Thus, we can now define the sets $X_{i,j}$ as follows.

Definition 2.8: *Consider any entry $T_{i,j}$ in the table T. If $T_{i,j} = FALSE$ then $X_{i,j} = *$, where $*$ denotes that $X_{i,j}$ is undefined. If $T_{i,j} = TRUE$, then $X_{i,j}$ is defined as the subset $X \subseteq [1, \ldots, i]$ of the smallest size which has size at most B and profit exactly j; when no such set exists the value of $X_{i,j}$ is undefined.*

It is now fairly easy to come up with a strategy for computing the T and X values inductively in a row-by-row fashion.

Algorithm PP:

Input: Knapsack capacity B, item sizes s_i and profits p_i.

Output: The tables T and X.

1. **for all** $1 \leq i \leq n$ **do begin**
$T_{i,0} \leftarrow TRUE$;
$X_{i,0} \leftarrow \emptyset$;

end ;

2. **for all** $j \neq p_1$ **do begin**
$T_{1,j} \leftarrow FALSE$;
$X_{1,j} \leftarrow *$;

end ;

3. $T_{1,p_1} \leftarrow TRUE$;

4. $X_{1,v_1} \leftarrow \{1\}$;

5. **for** $i = 2$ to n **do begin**
for all $1 \leq j \leq nP$ **do begin**
$T_{i+1,j} \leftarrow T_{i,j} \vee T_{i,j-p_{i+1}}$;
$X_{i+1,j}$ is assigned the set of smallest size among $X_{i,j}$ and $X_{i,j-p_{i+1}} \cup \{i+1\}$. If only one of these is defined then the choice is forced; when both are undefined or infeasible, then $X_{i+1,j} = *$.
end ;

end .

Once we have computed the tables X and T, the optimal solution to KNAPSACK can be read off from the column with highest index which has a $TRUE$ entry. The proof of correctness is by means of a simple induction on i and is omitted. The following theorem results.

Theorem 2.7: *Algorithm PP solves KNAPSACK exactly in time* $O(n^3 P \log SB)$.

This result is due to Ibarra and Kim [28], and a more efficient FPAS has been presented by Lawler [39].

2.5. Strong $\mathcal{NP}$-completeness and FPAS

Let us try to better understand the implications of a pseudo-polynomial algorithms for $\mathcal{NP}$-complete problems. Does the existence of such an algorithm imply that $\mathcal{P} = \mathcal{NP}$? The answer is no, because the running time is exponential in $\log MAX\#(I)$ and the length of the input is assumed to be polynomial in both $LENGTH(I)$ and $\log MAX\#(I)$ when we define the notion of $\mathcal{NP}$-completeness. Given this, it seems possible that every $\mathcal{NP}$-complete problem has a pseudo-polynomial time algorithm, even if $\mathcal{P} \neq \mathcal{NP}$. But it is not very hard to see that if $MAX\#(I)$ is polynomially bounded in $LENGTH(I)$, for every instance I of Π, then the existence of a pseudo-polynomial algorithm implies the existence of a polynomial time algorithm. This bound is valid for every "non-number" problem such as CLIQUE or HAM. We conclude that it is impossible to find pseudo-polynomial algorithms for such problems unless $\mathcal{P} = \mathcal{NP}$. This can be formalized as follows [14, 15].

Definition 2.9: *Given any optimization problem* Π*, we define the problem* Π_{poly} *as the problem* Π *restricted to only those instances* I *where* $MAX\#(I)$ *is polynomially bounded in* $LENGTH(I)$.

Definition 2.10: *An optimization problem* Π *is said to be* **strongly $\mathcal{NP}$-complete** *if* Π_{poly} *is* $\mathcal{NP}$*-complete.*

It is clear that all non-number problems are strongly $\mathcal{NP}$-complete. Moreover, it is fairly easy to see that the existence of pseudo-polynomial algorithms is quite unlikely for any number problem which is strongly $\mathcal{NP}$-complete.

Theorem 2.8: *Unless* $\mathcal{P} = \mathcal{NP}$*, a strongly* $\mathcal{NP}$*-complete problem cannot have a pseudo-polynomial algorithm.*

Some examples of strongly $\mathcal{NP}$-complete number problems are BIN PACKING, TSP and SCHEDULING. The standard $\mathcal{NP}$-hardness proof of KNAPSACK uses really large numbers so that does not establish strong $\mathcal{NP}$-completeness for this problem. (Of course, if we believe that $\mathcal{P} \neq \mathcal{NP}$ then KNAPSACK cannot be strongly $\mathcal{NP}$-complete since we have already seen a pseudo-polynomial algorithm for it.) It is not obvious how one may go about establishing strong $\mathcal{NP}$-completeness results. Most reductions for number problems involve really large numbers and thus say nothing about the hardness of Π_{poly}. The following theorem [14, 15] proves to be very helpful in this regard.

Theorem 2.9: *If* Π_1 *is strongly* $\mathcal{NP}$*-hard,* Π_2 *is in* $\mathcal{NP}$*, and there is a* **pseudo-polynomial reduction** *from* Π_1 *to* Π_2*, then* Π_2 *is strongly* $\mathcal{NP}$*-complete.*

Here a pseudo-polynomial reduction is a generalization of the usual notion of polynomial reduction, where it is required that the length of the produced instance is not much smaller than the original instance. The proof of the above theorem is fairly obvious. The only problem is that to apply this theorem we must know of some strongly $\mathcal{NP}$-complete number problem to start with. (Non-number problems are trivially strongly $\mathcal{NP}$-complete, but they are not very useful in the application of this theorem to number problems since reductions from them always seem to involve large numbers.) Luckily, there is a number problem called 3-PARTITION which is known to be strongly $\mathcal{NP}$-complete and this usually plays the role of the satisfiability problem when proving strong $\mathcal{NP}$-completeness results. Once again we urge the reader to refer to the book of Garey and Johnson [15] for a more comprehensive treatment of these ideas.

What does all this have to do with the approximation issue? It turns out that all known FPAS have been derived by the application of a scaling-like technique to a pseudo-polynomial algorithm, just as in the case of KNAPSACK. It seems plausible then to argue that we can find an FPAS for a problem only if it is not strongly $\mathcal{NP}$-complete. This idea is formalized in the following result due to Garey and Johnson [14].

Theorem 2.10: *Let* Π *be an optimization problem which has the property that for all instances* I*,* $OPT(I)$ *is polynomially bounded in the* $LENGTH(I)$ *and* $MAX\#(I)$*. If* Π *has a FPAS, then* Π *has a pseudo-polynomial algorithm.*

Proof: We will only deal with problems where all the numbers involved are positive integers, although the following proof can be generalized to the case of rationals too. Let $\epsilon = 1/MAXOPT$, where $MAXOPT$ is the bound on the value of OPT that is guaranteed in the theorem. Then $1/\epsilon$ is polynomially bounded in $LENGTH(I)$ and $MAX\#(I)$.

Suppose we have a FPAS for Π, and assume without loss of generality that Π is a minimization problem. Then we have an algorithm A_ϵ which finds a solution to any instance of Π such that

$$\begin{aligned} A_\epsilon(I) &\leq (1+\epsilon)OPT(I) \\ \Rightarrow \quad A_\epsilon(I) - OPT(I) &\leq \epsilon \cdot OPT(I) < 1 \end{aligned}$$

The last inequality follows from the choice of ϵ. Since the numbers involved are all integers, this means that A_ϵ finds an optimal solution. Moreover, A_ϵ is a pseudo-polynomial algorithm given our choice of ϵ. $\square$

Corollary 2.2: *Let Π be an integer optimization problem such that $OPT(I)$ is polynomially bounded in the $LENGTH(I)$ and $MAX\#(I)$. If Π is strongly $\mathcal{NP}$-complete , then Π does not have a FPAS.*

Notice that this accounts for the two technical conditions we had imposed on the class of optimization problems we are dealing with in this book. It is possible to find optimization problems which violate the conditions of the corollary, but such problems do not arise naturally in practice. At this point we have a fairly complete characterization of problems which have a FPAS. Since most interesting problems are strongly $\mathcal{NP}$-complete, we may as well forget about constructing FPAS for them. It is still possible to construct PAS for such problems, e.g. the one we saw for SCHEDULING. However, we do not even know how to construct a PAS for a large class of strongly $\mathcal{NP}$-complete problems. It is therefore quite natural to look for negative results about the existence of PAS. The following theorem of Garey and Johnson proves to be quite useful for this purpose; the proof is trivial.

Definition 2.11: *Let* Π *be an optimization problem. The decision problem* Π_K *is the problem of deciding, for a given instance* I*, whether* $OPT(I) \leq K$*.*

Theorem 2.11: *Let* Π *be an integer optimization problem. Suppose that the decision problem* Π_K *is* $\mathcal{NP}$*-hard for some constant* K*. Then, unless* $\mathcal{P} = \mathcal{NP}$*, there is no PAS for* Π *and, in particular, there does not exist any approximation algorithm* A *for* Π *with* $R_A < 1 + 1/K$*.*

Let us see how this theorem applies to specific problems. Consider the COLORING problem of finding a vertex coloring of a graph with the minimum number of colors. It is well known that checking that a graph is 3-colorable is $\mathcal{NP}$-hard. This implies that there is no PAS for coloring, and that no algorithm can guarantee a ratio better than 4/3. Similarly, the problem of deciding if an instance of BIN PACKING has a solution with 2 bins is $\mathcal{NP}$-hard – this is exactly the PARTITION problem. This implies that BIN PACKING does not have a PAS and that no algorithm can guarantee a ratio better than 3/2.

At this point a discerning reader may start to protest at what seems like a contradiction in that we have already seen an algorithm for BIN PACKING which has a ratio much better than 3/2. But note that we are talking only about *absolute* performance ratios in this section, whereas the approximation algorithms for BIN PACKING seen earlier (such as FFD or BFD) guaranteed a good *asymptotic* performance ratio. In fact, most people had assumed that strong $\mathcal{NP}$-completeness even implied that no Asymptotic PAS/FPAS could be be devised for the problem, unless $\mathcal{P} = \mathcal{NP}$. It was therefore a big shock when Vega and Lueker [59] presented an Asymptotic PAS for Bin Packing in 1981. This shock was compounded when Karmakar and Karp [34] transformed this result into an Asymptotic FPAS for BIN PACKING. These two results will be our next topic of discussion.

In conclusion, we would like to offer some observations about the above development. Consider the SCHEDULING problem. It is a scalable problem, which implies that for every approximation algorithm it must be the case that $R_A = R_A^\infty$. We know of a PAS, and therefore an Asymptotic PAS, for this problem, but it is clear that there cannot be even an Asymptotic FPAS (without leading to a FPAS and therefore a pseudo-polynomial algorithm) for SCHEDULING. This is in contrast to BIN PACKING which does not have a PAS, but does have an Asymptotic FPAS!

2.6. Discussion

Sahni [54] gives general techniques for constructing PAS and FPAS. For constructing a PAS, the technique is k-enumeration whose applications have been demonstrated above. The techniques for FPAS are rounding/scaling and interval partitioning, some aspects of which were seen above and are further demonstrated in the algorithms for BIN PACKING that follow. An interesting result of Korte and Schrader [36] shows that essentially the only way to construct PAS/FPAS is by means of these techniques. This result is proved in the context of independence systems but appear to be reasonably powerful in their application.

Problems

2–1 Consider the KNAPSACK problem defined in class. (Find a subset of n items, with total size at most K, which maximizes the total value.)

a). Consider the Greedy Algorithm (GA). It first sorts the items in decreasing order of their density $d_i = \frac{v_i}{s_i}$. Then it considers the items in this order and greedily adds an item to the current knapsack if the resulting size is at most K. Finally, it compares the solution so obtained with the one in which only the maximum value item is placed in the knapsack – choosing the better of these two possible solutions. Show that $R_{GA} = 2$.

b). Construct (and analyze) a PAS (polynomial time approximation scheme) for the KNAPSACK problem using GA.

Chapter 3

Bin Packing

SUMMARY: *Approximation schemes are presented for BIN PACKING, including a PAS due to Vega and Lueker, and an FPAS due to Karmakar and Karp. It is shown that the latter can be modified into an approximation algorithm whose absolute error is bounded by a polylogarithmic function in the value of the optimal solution.*

It is clear from the preceding discussion that we cannot expect to find any approximation schemes for BIN PACKING, unless $\mathcal{P} = \mathcal{NP}$. However, we had said that the hardness result for BIN PACKING does not preclude the existence of *asymptotic* approximation schemes. For the sake of completeness, we give a formal definition of such schemes.

Definition 3.1: *An* **Asymptotic PAS (APAS)** *is a family of algorithms* $\{A_\epsilon \mid \epsilon > 0\}$ *such that each* A_ϵ *runs in time polynomial in the length of the input and* $R^\infty_{A_\epsilon} \leq 1 + \epsilon$.

Definition 3.2: *An* **Asymptotic FPAS (AFPAS)** *is a family of algorithms* $\{A_\epsilon \mid \epsilon > 0\}$ *such that each* A_ϵ *runs in time polynomial in the length of the input and* $1/\epsilon$, *while* $R^\infty_{A_\epsilon} \leq 1 + \epsilon$.

The first result that we present is due to Vega and Lueker [59]. They provide an APAS for BIN PACKING which runs in *linear time* and has $A_\epsilon(I) \leq (1 + \epsilon) \cdot OPT(I) + 1$. The running time is linear in the $LENGTH(I)$ but turns out to be severely exponential in ϵ. Note that the reason this is an APAS, and not a PAS, is the additive error term of 1 in this bound. The basic techniques used in this result were similar to those used earlier for other problems such as Knapsack [54]. These may be summarized as follows:

- Elimination of "small" items.
- Interval Partitioning or Linear Grouping.
- Rounding of "Fractional" Solutions.

We then present the modification of this result due to Karmakar and Karp [34] which led to an AFPAS for BIN PACKING. They gave an approximation scheme with a performance guarantee similar to the one described above; the running time was improved to $\mathrm{O}\left(\frac{n\log n}{\epsilon^8}\right)$. In fact, a variation of their ideas leads to a stronger result. This was the construction of an approximation algorithm A which is fully polynomial and has the performance guarantee

$$A(I) \leq OPT(I) + \mathrm{O}(\log^2 OPT(I))$$

At this point there is no reason to believe that we cannot devise an (asymptotic) approximation algorithm which runs in polynomial time and guarantees that $A(I) \leq OPT(I) + \mathrm{O}(1)$. This is a major open problem!

We now derive the results described above. Our presentation combines the ideas of both Vega and Lueker, and Karmakar and Karp, as there is a considerable overlap in the basic tools used by them. The basic approach used in both results is as follows. We first define a restricted version of the problem where all items are reasonably large in size, and the item sizes can only take on a few distinct values. This version of the BIN PACKING problem turns out to reasonably easy to solve. We then provide a reduction from the original problem instance to a restricted problem instance in two steps. The first step is to eliminate "small" items – it is shown that given any packing of the remaining items, the small items can be added in without increasing the number of bins used significantly. The second step is to divide the item sizes into m intervals, and replace all items in the same interval by items of the same size. It turns out that this affects the value of the optimal solution only marginally. In the next few sections, we consider each of these ingredients in turn and finally show how they can all be tied together to give the APAS/AFPAS.

3.1. Asymptotic Approximation Scheme

The input to the BIN PACKING problem consists of a set of n items, where the size of the i^{th} item is s_i. We will assume that each item size is a rational number in the interval $(0, 1]$.

Definition 3.3: *For any instance* $I = \{s_1, \ldots, s_n\}$*, let* $SIZE(I) = \sum_{i=1}^{n} s_i$ *denote the total size of the* n *items, and let* $OPT(I)$ *denote the minimum number of unit size bins needed to pack the items.*

We now give two inequalities relating these quantities. The proof of the first lemma is obvious. The second lemma follows from a result given in Chapter 1 which showed that the First Fit algorithm will always find a solution that uses at most $2 \cdot SIZE(I) + 1$ bins. This is a constructive result in that there is a linear time algorithm which guarantees the bound of Lemma 3.2.

Lemma 3.1: $SIZE(I) \leq OPT(I) \leq |I| = n.$

Lemma 3.2: $OPT(I) \leq 2 \cdot SIZE(I) + 1.$

We will represent an instance I as an *ordered* list of items, writing $I = s_1 s_2 \ldots s_n$ such that $1 \geq s_1 \geq s_2 \geq \cdots \geq s_n > 0$.

Definition 3.4: *Let* $I_1 = x_1 x_2 \ldots x_n$ *and* $I_2 = y_1 y_2 \ldots y_n$ *be two instances of equal cardinality. The instance* I_1 *is said to* **dominate** *the instance* I_2*, or* $I_1 \succeq I_2$*, if it is the case that* $x_i \geq y_i$*, for all* i*.*

The following lemma is easily proved by noting that any feasible packing of I_1 gives a feasible packing of I_2, using the same number of bins.

Lemma 3.3: *Let I_1 and I_2 be two instances of equal cardinality such that $I_1 \succeq I_2$. Then, $SIZE(I_1) \geq SIZE(I_2)$ and $OPT(I_1) \geq OPT(I_2)$.*

We define a restricted version of the BIN PACKING problem as follows. Suppose that the item sizes in I take on only m distinct values. Now the instance I can be represented as a multi-set of items which are drawn from these m types of items.

Definition 3.5: *Suppose that we are given m distinct item sizes $V = \{v_1, \ldots, v_m\}$, such that $1 \geq v_1 > v_2 > \cdots > v_m > 0$, and an instance I of items whose sizes are drawn only from V. Then, we can represent I as multi-set $M_I = \{n_1 : v_1, n_2 : v_2, \ldots, n_m : v_m\}$, where n_i is a non-negative integer denoting the number of items in I which have size v_i.*

It follows that $|M_I| = \sum_{i=1}^{m} n_i = n$, $SIZE(M_I) = \sum_{i=1}^{m} n_i v_i = SIZE(I)$ and $OPT(M_I) = OPT(I)$. We now define the restricted version of the BIN PACKING problem called RBP.

Definition 3.6: *For all $0 < \delta < 1$ and positive integers m, the problem $RBP[\delta, m]$ is defined as the BIN PACKING problem restricted to instances where the item sizes take on at most m distinct values and each item size is at least as large as δ.*

In the next section we show how to approximately solve RBP via a linear programming formulation.

3.1.1. Restricted Bin Packing

Assume that δ and m are fixed independently of the input size n. The input instance for $RBP[\delta, m]$ is a multiset $M = \{n_1 : v_1, n_2 :$

$v_2, \ldots, n_m : v_m\}$ such that $1 \geq v_1 > v_2 > \cdots > v_m \geq \delta$. Let $n = |M| = \sum_{i=1}^{n} n_i$. In the following discussion we will assume that the underlying set V for M is fixed. Note that, given M, it is trivial to determine V and verify that M is a valid instance of $RBP[\delta, m]$.

Consider a packing of some subset of the items in M into a unit size bin B. We can denote this by a multiset $B = \{b_1 : v_1, b_2 : v_2, \ldots, b_m : v_m\}$ such that b_i is the number of items of size v_i that are packed into B. More concisely, having fixed V, we can denote the packing in B by the m-vector $B = (b_1, \ldots, b_m)$ of non-negative integers. We will say that two bins packed with items from M are of the same type if the corresponding packing vectors are identical.

Definition 3.7: *A* **bin type** T *is an m-vector $(T_1, \ldots, T_m)$ of non-negative integers such that $\sum_{i=1}^{m} T_i v_i \leq 1$.*

Having fixed the set V, the collection of possible bin types is fully determined and is finite. Let $T^1, \ldots, T^q$ denote the set of all legal bin types with respect to V. Here q, the number of distinct types, is a function of δ and m. We bound the value of q as follows. *Do you see why?*

Lemma 3.4: *Let $k = \lfloor \frac{1}{\delta} \rfloor$. Then,*

$$q(\delta, m) \leq \binom{m+k}{k}$$

Proof: Notice that each type vector $T^t = (T_1^t, \ldots, T_m^t)$ has the property that, for all i, $T_i^t \geq 0$ and $\sum_{i=1}^{m} T_i^t v_i \leq 1$. It follows that $\sum_{i=1}^{m} T_i^t \leq k$, since we have a lower bound of δ on the values in V. Thus, each type vector corresponds to a way of choosing m non-negative integers whose sum is at most k. This is the same as choosing $m+1$ non-negative integers whose sum is exactly k. The number of such choices is an upper bound on the value of q. A standard counting argument now gives the desired bound.
□

Consider any feasible solution x to an instance M of $RBP[\delta, m]$. Each packed bin in this solution can be classified as belonging to one of the $q(\delta, m)$ possible types of packed bins. The solution x can therefore be specified completely by providing the number of bins of each of the q types.

Definition 3.8: *A feasible solution x to an instance M of $RBP[\delta, m]$ is a q-vector of non-negative integers, say $x = (x_1, \ldots, x_q)$, where x_t denotes the number of bins of type T^t used in x.*

Notice that not all q-vectors correspond to a feasible solution. A feasible solution must guarantee, for each i, that exactly n_i items of size v_i are packed in the various copies of the bin types. The feasibility condition can be phrased as a series of linear equations as follows.

$$\forall i \in \{1, \ldots, m\},\ \sum_{t=1}^{q} x_t T_i^t = n_i$$

Let the matrix A be a $q \times m$ matrix whose t^{th} row is the type vector T^t, and $\vec{n} = (n_1, \ldots, n_m)$ denote the multiplicities of the various item sizes in the input instance M. Then the above set of equations can be concisely expressed as $\vec{x}.A = \vec{n}$. The number of bins used in the solution x is simply $\vec{x}.\vec{1} = \sum_{t=1}^{q} x_t$, where $\vec{1}$ denotes all-ones vector. In fact, we have proved the following lemma.

Lemma 3.5: *The optimal solution to an instance M of $RBP[\delta, m]$ is exactly the solution to the following* integer linear program ILP(M)

$$\begin{aligned} &\textit{minimize} && \vec{x}.\vec{1} \\ &\textit{subject to} && \\ & && \vec{x} \geq 0 \\ & && \vec{x}.A \geq \vec{n} \end{aligned}$$

Notice that we have replaced the equations by inequalities, but that does not affect the validity of the lemma since a packing of a superset of M can always be converted into a packing of M using the same number of bins. It is also worth noting that the matrix A is not determined *a priori* but depends on the instance M.

How easy is it to obtain this integer program? Note that the number of constraints in ILP(M) is exponentially large in terms of δ and m. However, we are going to assume that both δ and m are constants which are fixed independently of the length of the input, which is n. Thus, obtaining ILP(M) requires time linear in n, given any instance M of cardinality n.

How about solving ILP? Recall that the integer programming problem is $\mathcal{NP}$-complete in general [15]. However, there is an algorithm due to Lenstra [41, 20, 56] which solves any integer linear program in time linear in the number of constraints, provided the number of variables is fixed. This is exactly the situation in ILP: the number of variables q is fixed independent of n, as is the number of constraints which is $q + m$. Thus, we can solve ILP exactly in time which is independent of n. (A more efficient algorithm for approximately solving ILP will be described in a later section.) The following theorem results. Here $f(\delta, m)$ is some constant which depends only on δ and m.

Theorem 3.1: *Any instance of* $RBP[\delta, m]$ *can be solved in time* $\mathrm{O}(n + f(\delta, m))$.

3.1.2. Eliminating Small Items

In this section we present the second ingredient of the APAS devised by Vega and Lueker. It is shown that if we have a packing of all items except those whose sizes are bounded from above by δ, then it is possible to obtain a packing of all items which is not much worse in its use of bins. This is summarized in the following lemma; the rest of this section is devoted to the proof of this lemma.

Lemma 3.6: *Fix some constant* $\delta \in (0, \frac{1}{2}]$. *Let* I *be an instance of BIN PACKING and suppose that all items of size greater than* δ *have been packed into* β *bins. Then it is possible to find in linear time a packing for* I *which uses at most* $\max\{\beta, (1 + 2\delta) \cdot OPT(I) + 1\}$ *bins.*

Proof: The basic idea is to start with the packing of the "large" items and to use the greedy algorithm First Fit to pack the "small" items into the empty space in the β bins. The implementation of this scheme is not very important. For example, we could start by numbering the β bins in an arbitrary fashion. Then the FF algorithm can be run as usual using this ordering to decide where each small item will be placed. If at some point the small items do not fit into any of the currently available bins, then a new bin is initiated.

The best case is where the small items can all be greedily packed into the β bins which were open initially. Clearly, the lemma is valid in that case. Suppose now that some new bins were required for the small items. We claim that at the end of the entire process each of the bins used for packing I has at most δ empty space in it, with the possible exception of at most one bin.

To see this, consider the case where there are two bins with more than δ wasted space. Let these bins be B_i and B_j, with $i < j$ under the ordering defined by FF. It cannot be the case that either of these two bins is from the set of β bins which were available initially. Otherwise, we would have packed some small item into that bin before opening any new bin, contradicting the assumption that new bins were required by FF. On the other hand, if both bins are new bins then we would have packed at least one of the items from B_j into B_i before the bin B_j was opened.

Let $\beta' > \beta$ be the total number of bins used by FF. We are guaranteed that all the bins, except one, are at least $1-\delta$ full. This implies that $SIZE(I) \geq (1-\delta)(\beta'-1)$. But we know that $SIZE(I) \leq OPT(I)$, implying that

$$\beta' \leq \frac{1}{1-\delta} OPT(I) + 1 \leq (1+2\delta) \cdot OPT(I) + 1$$

and we have the desired result.
□

3.1.3. Linear Grouping

The final ingredient needed for the APAS is called Interval Partitioning or Linear Grouping. This is a technique for converting an instance I of BIN PACKING into an instance M of $RBP[\delta, m]$, for an appropriate choice of δ and m, without changing the value of the optimal solution too much. Let us assume for now that all the items in I are of size at least δ, for some choice of $\delta \in (0, \frac{1}{2}]$. All that remains is to show how to obtain an instance where the item sizes take on only m different values. First, let us fix some parameter k, a non-negative integer to be specified later. We now show how to convert an instance of $RBP[\delta, n]$ into an instance of $RBP[\delta, m]$, for $m = \lfloor n/k \rfloor$.

Definition 3.9: *Given an instance I of $RBP[\delta, n]$ and a parameter k, let $m = \lfloor n/k \rfloor$. Define the groups of items $G_i = s_{(i-1)k+1} \ldots s_{ik}$, for $i = 1, \ldots, m$, and let $G_{m+1} = s_{mk+1} \ldots s_n$.*

Here the group G_1 contains the k largest items in I, G_2 contains the next k largest items and so on. The following fact is an easy consequence of these definitions.

Fact 3.1: $G_1 \succeq G_2 \succeq \cdots \succeq G_m$.

From each group G_i we can obtain a new group of items H_i by increasing the size of each item in G_i to that of the largest item in that group. The following fact is also obvious.

Definition 3.10: *Let $v_i = s_{(i-1)k+1}$ be the largest item in group G_i. Then the group H_i is a group of $|G_i|$ items, each of size v_i. In other words, $H_i = v_i v_i \ldots v_i$ and $|H_i| = |G_i|$.*

Fact 3.2: $H_1 \succeq G_1 \succeq H_2 \succeq G_2 \succeq \cdots \succeq H_m \succeq G_m$.

The entire point of these definitions is to obtain two instances of $RBP[\delta, m]$ such that their optimal solutions bracket the optimal solution for I. These instances are defined as follows.

Definition 3.11: *Let the instance* $I_{LO} = H_2H_3 \ldots H_{m+1}$ *and* $I_{HI} = H_1H_2H_3 \ldots H_{m+1}$.

Note that I_{LO} is an instance of $RBP[\delta, m]$. Moreover, it is easy to see that $I \preceq I_{HI}$. We now present some properties of these three instances.

Lemma 3.7:

$$OPT(I_{LO}) \leq OPT(I) \leq OPT(I_{HI}) \leq OPT(I_{LO}) + k$$

$$SIZE(I_{LO}) \leq SIZE(I) \leq SIZE(I_{HI}) \leq SIZE(I_{LO}) + k$$

Proof: First, observe that

$$I_{LO} = H_2H_3 \ldots H_m H_{m+1} \preceq G_1G_2 \ldots G_{m-1}X,$$

where X is the any set of $|H_{m+1}|$ items from G_m. The right hand side of this inequality is a subset of I, and this gives us that $OPT(I_{LO}) \leq OPT(I)$ and $SIZE(I_{LO}) \leq SIZE(I)$, using Lemma 3.3.

Observe now that $I_{HI} = H_1 I_{LO}$. Given any packing of I_{LO}, we can obtain a packing of I_{HI} which uses at most k extra bins. (Just pack each item in H_1 in a separate bin.) This implies that $OPT(I_{HI}) \leq OPT(I_{LO}) + k$ and $SIZE(I_{HI}) \leq SIZE(I_{LO}) + k$. Since $I \preceq I_{HI}$, by using Lemma 3.3 we get the remaining part of the desired result.
□

It is worth noting that the result presented in this lemma is constructive. There is an $O(n \log n)$ time algorithm which constructs the instances I_{LO} and I_{HI}, and given an optimal packing of I_{LO} it is possible to construct a packing of I which meets the guarantee of the above lemma.

Do you see how to actually find I_{LO} and I_{HI}, as well as convert their packing into a packing of I, within the stated time bound?

3.1.4. APAS for Bin Packing

We now put together all these ingredients and obtain the APAS. The algorithm A_ϵ, for any $\epsilon \in (0, 1]$, takes as input an instance I of BIN PACKING consisting of n items.

Algorithm A_ϵ:

Input: Instance I consisting of n item sizes $\{s_1, \ldots, s_n\}$.

Output: A packing into unit-sized bins.

1. $\delta \leftarrow \frac{\epsilon}{2}$

2. Set aside all items of size smaller than δ, obtaining an instance J of $RBP[\delta, n']$ with $n' = |J|$.

3. $k \leftarrow \left\lceil \frac{\epsilon^2}{2} n' \right\rceil$

4. Perform linear grouping on J with parameter k. Let J_{LO} be the resulting instance of $RBP[\delta, m]$ and $J_{HI} = H_1 \cup J_{LO}$, with $|H_1| = k$ and $m = \left\lfloor \frac{n'}{k} \right\rfloor$.

5. Pack J_{LO} optimally using Lenstra's algorithm on $ILP(J_{LO})$.

6. Pack the k items in H_1 into at most k bins.

7. Obtain a packing of J using the same number of bins as in steps 6 and 7, by replacing each item in J_{HI} by the corresponding (smaller) item in J.

8. Using FF, pack all the small items set aside in step 2, using new bins only if necessary.

How many bins does A_ϵ use in the worst case? Observe that we have packed the items in J_{HI}, hence the items in J, into at most $OPT(J_{LO}) + k$ bins. Consider the now the value of k in terms of the optimal solution. Since all items have size at least $\epsilon/2$ in J, it must be the case that $SIZE(J) \geq \epsilon n'/2$. This implies that

$$k \leq \frac{\epsilon^2 n'}{2} + 1 \leq \epsilon \cdot SIZE(J) + 1 \leq \epsilon \cdot OPT(J) + 1$$

Using Lemma 3.7, we obtain that J is packed into a number of bins not exceeding

$$OPT(J_{LO}) + k \leq OPT(J) + \epsilon \cdot OPT(J) + 1 \leq (1 + \epsilon) \cdot OPT(J) + 1$$

Finally, Lemma 3.6 implies that, while packing the small items at the last step, we use a number of bins not exceeding

$$\max\{(1+\epsilon)\cdot OPT(J)+1,\ (1+\epsilon)\cdot OPT(I)+1\} \leq (1+\epsilon)\cdot OPT(I)+1$$

since $OPT(J) \leq OPT(I)$. We have obtained the following theorem.

Theorem 3.2: *The algorithm A_ϵ finds a packing of I into at most $(1+\epsilon)\cdot OPT(I)+1$ bins in time $c(\epsilon)n\log n$, where $c(\epsilon)$ is a constant depending only on ϵ.*

For the running time, note that the only really expensive step in the algorithm is the one where we solve ILP using Lenstra's algorithm. As we observed earlier, that this requires time linear in n, although it may be severely exponential in δ and m which are functions of ϵ.

3.2. Asymtotic Fully Polynomial Scheme

Our next goal is to convert the preceding APAS into an AFPAS. The reason that the above scheme is not fully polynomial is the use of the algorithm for integer linear programming which requires time exponential in $1/\epsilon$. We now a describe a technique for getting rid of this step via the construction of a "fractional" solution to the restricted bin packing problem, and a "rounding" of this to a feasible solution which is not very far from optimal. This is based on the ideas due to Karmakar and Karp.

3.2.1. Fractional Bin Packing and Rounding

Consider again the problem $RBP[\delta, m]$. By the preceding discussion, any instance I of this problem can be formulated as the integer linear program $ILP(I)$.

minimize $\vec{x}.\vec{1}$

subject to

$$\vec{x} \geq 0$$

$$\vec{x}.A = \vec{n}$$

Notice that we are now using equality in the last constraint. Recall that A is a $q \times m$ matrix, $\vec{x}$ is a q-vector and $\vec{n}$ is an m-vector. The bin types matrix A, as well as $\vec{n}$, are determined by the instance I.

Consider now the linear programming relaxation of $ILP(I)$. This system $LP(I)$ is exactly the same as $ILP(I)$, except that we now relax the requirement that $\vec{x}$ be an integer vector. Recall that $SIZE(I)$ is the total size of the items in I, and that $OPT(I)$ is the value of the optimal solution to $ILP(I)$ as well as the smallest number of bins into which the items of I can be packed.

Definition 3.12: *$LIN(I)$ is the value of the optimal solution to $LP(I)$, the linear programming relaxation of $ILP(I)$.*

What does a non-integer solution to $LP(I)$ mean? The value of x_i is a real number which denotes the number of bins of type T^i which are used in the optimal packing. One may interpret this as saying that items can be "broken up" into fractional parts, and these fractional parts can then be packed into fractional bins. This in general would give us a solution of value $SIZE(I)$, but keep in mind that the constraints in $LP(I)$ do not allow any arbitrary "fractionalization". The constraints require that in any fractional bin, the items packed therein must be the same fraction of the original items. Thus, this solution does capture some of the features of the original problem. We will refer to the solution of $LP(I)$ as a *fractional bin packing*.

To analyze the relationship between the fractional and integral solutions to any instance we will have to use some basic facts from the theory of linear programming. The uninitiated reader is referred to any standard text-book for a more complete treatment, e.g. see the book by Papadimitriou and Steiglitz [48].

Consider the system of linear equations implicit in the constraint* $\vec{x}.A = \vec{n}$. Here we have m linear equations in q variables, where q is

*We will ignore the non-negativity constraints for now as they do not bear upon the following discussion.

much larger than m. This is an overconstrained system of equations. Let us assume that $rank(A) = m$; it is easy to modify the following analysis when $rank(A) < m$. Assume, without loss of generality, that the first m rows of A form a basis, i.e. they are linearly independent. The following are standard observations from linear programming theory.

Make sure you see how to handle the case where the rank is smaller.

Definition 3.13: *A* **basic feasible solution** *to LP is a solution* $\vec{x}^*$ *such that only the entries corresponding to the basis of* A *are non-zero. In other words,* $x_i^* = 0$ *for all* $i > m$.

Fact 3.3: *Every LP has an* **optimal** *solution which is a basic feasible solution.*

We can now derive the following lemma which relates $LIN(I)$ to both $SIZE(I)$ and $OPT(I)$.

Lemma 3.8: *For all instances* I *of* $RBP[\delta, m]$,

$$SIZE(I) \leq LIN(I) \leq OPT(I) \leq LIN(I) + \frac{m+1}{2}$$

Proof: The value of any solution $\vec{x}$ to $LP(I)$ is $\sum_{i=1}^{q} x_i$. It is easy to see that the total number of times the item j is packed in any fractional solution is exactly n_j, given the constraint of A. This implies the first inequality. The second inequality follows from the observation that an optimal solution to $ILP(I)$ is also a feasible solution to $LP(I)$.

To see the last inequality, fix I and let $\vec{y}$ be some basic feasible solution to $LP(I)$. Since $\vec{y}$ has at most m non-zero entries, it uses only m different types of bins. Rounding up the value of each component of $\vec{y}$ will increase the number of bins by at most m, and will yield a solution to ILP. The bound promised in the lemma is slightly stronger and may be observed as follows. Define the vectors $\vec{w}$ and $\vec{z}$ as below.

$$\forall i, \quad w_i = \lfloor y_i \rfloor$$

$$\forall i, \quad z_i = y_i - w_i$$

The vector $\vec{w}$ is the integer part of the solution and $\vec{z}$ is the fractional part. Let J denote the instance of $RBP[\delta, m]$ which consists of the items not packed in the (integral) solution specified by $\vec{w}$. The vector $\vec{z}$ gives a fractional packing of the items in J, such that each of the m bin types is used a number of times which is a fraction less than 1. It is easy to see that $\vec{z}$ is an optimal fractional packing for J. It follows that

Prove that $\vec{z}$ is indeed an optimal fractional packing of J.

$$SIZE(J) \leq LIN(J) \leq \sum_{i=1}^{m} z_i \leq m$$

By Lemma 3.2 we know that

$$OPT(J) \leq 2 \cdot SIZE(J) + 1$$

It is also obvious that $OPT(J) \leq m$, since rounding each non-zero z_i up to 1 gives a feasible packing of J. Thus, we have that

$$\begin{aligned} OPT(J) &\leq \min\{m, 2 \cdot SIZE(J) + 1\} \\ &\leq SIZE(J) + \min\{m - SIZE(J), SIZE(J) + 1\} \\ &\leq SIZE(J) + \frac{m+1}{2} \end{aligned}$$

We needed to bound $OPT(I)$ in terms of $LIN(I)$ and m, and this may be done as follows

$$\begin{aligned} OPT(I) &\leq OPT(I - J) + OPT(J) \\ &\leq \sum_{i=1}^{m} w_i + \left(SIZE(J) + \frac{m+1}{2}\right) \\ &\leq \sum_{i=1}^{m} w_i + LIN(I) + \frac{m+1}{2} \\ &= \sum_{i=1}^{m} w_i + \sum_{i=1}^{m} z_i + \frac{m+1}{2} \\ &= LIN(I) + \frac{m+1}{2} \end{aligned}$$

This completes the proof.
□

It is not very hard to see that all of the above is constructive. More precisely, given the solution to $LP(I)$, we can construct in linear time a solution to I such that the bound from the above theorem is met. The only problem is that it is not obvious that we can solve the linear program in fully polynomial time, even though there exist polynomial time algorithms for linear programming [33], unlike the general problem of integer programming. The reason is that the number of variables is still exponential in $1/\epsilon$. All we have achieved is that we no longer need to solve an integer program.

Karmakar and Karp showed how to get around this problem by resorting to the Ellipsoid method of Grötschel, Lovász and Schrijver [19, 20, 56]. In this method, it is possible to solve a linear program with an exponential number of constraints in time which is polynomial in the number of variables and the number sizes, given a *separation oracle*. A separation oracle takes any proposed solution vector $\vec{x}$ and either guarantees that it is a feasible solution, or provides any one constraint which is violated by it. Karmakar and Karp gave an efficient construction of a separation oracle for $LP(I)$. This would result in a polynomial time algorithm for $LP(I)$ if it had a small number of variables, even if it has an exponential number of constraints. Unfortunately, our situation is exactly the reverse: we have a small number of constraints and an exponential number of variables. However, it is possible to get around this problem by considering the *dual linear program* for $LP(I)$. This has the desired features of a small number of variables, and its optimal solution corresponds exactly to the optimal solution of $LP(I)$.

One important detail is that that it is impossible to solve $LP(I)$ exactly in fully polynomial time. However, it can be solved within an additive error of 1 in fully polynomial time. Moreover, the implementation of the separation oracle is in itself an approximation algorithm. The idea behind this is due to Gilmore and Gomory [17] who observed that, in the case of an infeasible proposed solution, a violated constraint can be computed via the solution of a knapsack problem. Since this is $\mathcal{NP}$-complete, one must resort to the use of an approximation scheme for KNAPSACK. Due to all this the solution of the dual is not exact but a close approximation. This was used by Karmakar and Karp to obtain an approximate lower bound to the original problem's optimal value. Having devised the procedure for efficiently computing an approximate lower bound, they then used this to actually construct an approximate solution.

This algorithm is rather formidable and the details are omitted as it is outside the scope of this book. The following theorem results.

Theorem 3.3: *There is a fully polynomial time algorithm A for solving an instance I of* $RBP[\delta, m]$ *such that* $A(I) \leq LIN(I) + \frac{m+1}{2} + 1$.

3.2.2. AFPAS for Bin Packing

We are now ready to present the AFPAS for BIN PACKING. We will need the following variant of Lemma 3.7. The proof is almost the same and is left as an exercise.

Lemma 3.9: *Using the linear grouping scheme on an instance I of* $RBP[\delta, n]$*, we obtain an instance* I_{LO} *of* $RBP[\delta, m]$ *and a group* H_1 *such that, for* $I_{HI} = H_1 I_{LO}$*,*

$$LIN(I_{LO}) \leq LIN(I) \leq LIN(I_{HI}) \leq LIN(I_{LO}) + k$$

The basic idea behind the AFPAS of Karmakar and Karp is very similar to that used in the APAS. We first eliminate all the small items, and then apply linear grouping to the remaining items. The resulting instance of $RBP[\delta, m]$ is then formulated as an ILP, and the solution to the corresponding relaxation LP is computed using the Ellipsoid method. The fractional solution is then rounded to an integer solution. The small items are then added into the resulting packing exactly as before.

Algorithm A_ϵ**:**

Input: Instance I consisting of n item sizes $\{s_1, \ldots, s_n\}$.

Output: A packing into unit-sized bins.

1. $\delta \leftarrow \frac{\epsilon}{2}$.

2. Set aside all items of size smaller than δ, obtaining an instance J of $RBP[\delta, n']$ with $n' = |J|$.

3. $k \leftarrow \left\lceil \frac{\epsilon^2 n'}{2} \right\rceil$

4. Perform linear grouping on J with parameter k. Let J_{LO} be the resulting instance of $RBP[\delta, m]$ and $J_{HI} = H_1 \cup J_{LO}$, with $|H_1| = k$ and $m = \left\lfloor \frac{n'}{k} \right\rfloor$.

5. Pack the k items in H_1 into at most k bins.

6. Pack J_{LO} using the Ellipsoid method and rounding the resulting fractional solution.

7. Obtain a packing of J using the same number of bins as used for J_{HI}, by replacing each item in J_{HI} by the corresponding (smaller) item in J.

8. Using FF, pack all the small items set aside in step 2, using new bins only if necessary.

Theorem 3.4: *The approximation scheme $\{A_\epsilon : \epsilon > 0\}$ is an AFPAS for BIN PACKING such that*

$$A_\epsilon(I) \leq (1+\epsilon) \cdot OPT(I) + \frac{1}{\epsilon^2} + 3$$

Proof: The running time is dominated by the time required to solve the linear program, and we are guaranteed that this is fully polynomial.

The number of bins used to pack the items in J_{LO} is easily seen to be at most

$$(LIN(J_{LO}) + 1) + \frac{m+1}{2} \leq OPT(I) + \frac{1}{\epsilon^2} + 2$$

given the preceding lemmas and the choice of m. The number of bins used to pack the items in H_1 is at most k, which in turn can be bounded as follows using the observation that $OPT(J) \geq SIZE(J) \geq \epsilon n'/2$.

$$k \leq \left\lceil \frac{n'\epsilon^2}{2} \right\rceil \leq \epsilon \cdot OPT(J) + 1 \leq \epsilon \cdot OPT(I) + 1$$

Thus, the total number of bins used to pack the items in J cannot exceed

$$(1+\epsilon) \cdot OPT(I) + \frac{1}{\epsilon^2} + 3$$

The number of bins used after the addition of the small items can be bounded using Lemma 3.6. This gives the desired result.
□

3.3. Near-Absolute Approximation

We conclude by presenting a technique of Karmakar and Karp which gives an approximation algorithm with an error that is bounded by a slowly increasing function of $OPT(I)$. This result is a step towards devising an absolute approximation algorithm for BIN PACKING. In fact, Johnson [29] had observed that the Vega-Lueker scheme could be modified to construct an approximation algorithm with a performance bounded by $OPT(I) + \mathrm{O}\left(OPT(I)^{1-\delta}\right)$, for some positive constant δ, by letting the value of ϵ depend on the instance I. Here we will present a new technique which leads to a performance bounded by $OPT(I) + \mathrm{O}\left(\log^2 OPT(I)\right)$.

The new technique is a variant of linear grouping called *geometric grouping*. To motivate this, let us first try to pinpoint the exact sources of sub-optimality in the preceding AFPAS. This scheme depends on the grouping parameter k, which leads to an instance with m different item sizes, where $m \approx n/k$. There are two main sources of error in this scheme. The first is in the solution of the restricted bin packing problem, where the rounding error depends on the number of item sizes m. Then there is an error due to the replacement of the original instance by a discretized instance consisting of at most m different item sizes. This last error is roughly the value of k. Since the small items are easily handled, we can assume that the value of $SIZE(I)$ is at least δn. Thus, we cannot choose the value of k to be greater than $\epsilon\delta n$. It is then clear that choosing δ close to ϵ and $k \leq \epsilon^2 n$, we will get the desired approximation.

The way to improve this error is to reduce the value of k closer to a constant. But then the value of m will increase correspondingly and we would not have gained anything. The key insight of Karmakar and Karp was that it is not really necessary to pay the penalty of an error of m in rounding the fractional solution. Recall that the solution to $LP(I)$ was broken up into an integral part $\vec{w}$ and a fractional part $\vec{z}$. After packing some of the items as specified by $\vec{w}$, the remaining items were thought of as an instance J whose optimal fractional solution was exactly the solution specified by $\vec{z}$. The error of m came from the brute force solution of the instance J. The new idea was to iterate the approximation algorithm on this instance J. This seems like a very natural idea but the problem with implementing it is that we are not guaranteed that the iterated process will terminate. Consider the linear program defined by J, one of its optimal solutions is exactly the vector $\vec{z}$. Thus, iterating the process could keep giving us the same solution $\vec{z}$ for J, whose integral part is zero.

It is for this reason that Karmakar and Karp introduced the technique of geometric grouping. Their approach is to use a different grouping, even a different parameter k, at each stage of the iteration. Moreover, the exact form of the grouping is heavily dependent on the distribution of the item sizes in the instance. Thus, a new grouping would be used for J, guaranteeing that the value of m decreases by a constant fraction. This would imply a speedy termination. We give a more formal, and less intuitive, description of these ideas below.

Fix some instance I of BIN PACKING and consider the following definitions.

Definition 3.14: *Let $\delta = x_n$ be the size of the smallest item in I, and define $\Delta = \left\lfloor \log_2 \frac{1}{\delta} \right\rfloor$.*

Definition 3.15: *Denote by I^r the instance of BIN PACKING which* *consists of the items in I whose sizes lie in the interval $\left(\frac{1}{2^{r+1}}, \frac{1}{2^r}\right]$, for $r = 0, 1, \ldots, \Delta$.*

This a geometric partitioning of the items in I into sets of items of roughly the same size, i.e. within a factor of two. The formal definition of *geometric grouping* is as follows.

Definition 3.16: *The geometric grouping of I, with parameter k, is obtained by applying linear grouping to each instance I^r using the parameter k_r, where $k_r = k2^r$. Let I^r_{LO} and $I^r_{HI} = G^r_1 \cup I^r_{LO}$ be the outcome of the linear groupings, with $|G^r_1| = k_r$. Then the outcome of the geometric grouping consists of the instances I_{LO} and $I_{HI} = G_1 \cup I_{LO}$, which are defined as follows.*

$$I_{LO} = \cup_r I^r_{LO}$$

$$I_{HI} = \cup_r I^r_{HI}$$

$$G_1 = \cup_r G^r_1$$

Notice that we are now defining $I_{HI} = G_1 \cup I_{LO}$, instead of $I_{HI} = H_1 \cup I_{LO}$ as before. This could well have been done in the earlier arguments without affecting the analysis in any way. The following lemma corresponds to Lemmas 3.7 and 3.9 that were proved for linear grouping.

Lemma 3.10:

$$OPT(I_{LO}) \leq OPT(I) \leq OPT(I_{HI}) \leq OPT(I_{LO}) + k\Delta$$

$$LIN(I_{LO}) \leq LIN(I) \leq LIN(I_{HI}) \leq LIN(I_{LO}) + k\Delta$$

$$SIZE(I_{LO}) \leq SIZE(I) \leq SIZE(I_{HI}) \leq SIZE(I_{LO}) + k\Delta$$

Proof: The proof is very similar to the proof of Lemma 3.7. It is easy to show that the following inequalities hold.

$$OPT(I_{LO}) \leq OPT(I) \leq OPT(I_{HI}) \leq OPT(I_{LO}) + OPT(G_1)$$

We can easily show a similar series of inequalities for LIN and $SIZE$.

Now notice that $G_1 = \cup_r G_1^r$ and that each G_1^r consists of k_r items of size at most $1/2^r$ each. Clearly, the items in each G_1^r can be packed into k bins. Thus, we obtain that $SIZE(G_1^r) \leq LIN(G_1^r) \leq OPT(G_1^r) \leq k$. Summing over all r, this implies that

$$SIZE(G_1) \leq LIN(G_1) \leq OPT(G_1) \leq k\Delta$$

Plugging in these bounds gives the desired result.
□

The next lemma is the crucial one – it shows that the number of distinct item sizes after the geometric grouping is much less than the size of the original input. Here we denote the number of distinct item sizes in any instance X by $m(X)$.

Lemma 3.11: $m(I_{LO}) \leq \frac{2}{k} \cdot SIZE(I) + \Delta$

Proof: Observe that

$$SIZE(I^r) \geq |I^r| . \frac{1}{2^{r+1}} \geq (m(I_{LO}^r) - 1) \cdot (k2^r) \cdot \frac{1}{2^{r+1}}$$

Upon rearranging, we obtain that

$$m(I_{LO}^r) \leq \frac{2}{k} \cdot SIZE(I^r) + 1$$

Summed over all r, this gives the desired result.
□

We are now ready to describe the overall algorithm. This algorithm is parametrized by the two values δ and k, these will be specified later.

Algorithm $A(\delta, k)$:

Input: Instance I consisting of n item sizes $\{s_1, \ldots, s_n\}$.

Output: A packing into unit-sized bins.

1. Discard all items of size smaller than δ, obtaining an instance J of $RBP[\delta, n']$ with $n' = |J|$.

2. while $SIZE(J) > 1 + \frac{k}{k-1} \ln \frac{1}{\delta}$ **do begin**

Perform geometric grouping with parameter k to get J_{LO} and $J_{HI} = G_1 \cup J_{LO}$.
Pack G_1 into $k\Delta$ bins, by putting each item into a separate bin.
Solve J_{LO} via the LP formulation and let $\vec{y}$ be the optimal basic feasible solution obtained.
Define $\vec{w}$ and $\vec{z}$ as the integral and fractional parts, respectively, of the solution $\vec{y}$. Pack a subset of the items in J_{LO} as per the vector $\vec{w}$, and from this obtain a packing of the corresponding items in J.
Redefine J to be the items left over, i.e. those items whose packing is specified by the fractional part $\vec{z}$.

end ;

3. Pack the remaining items into at most $1 + \frac{k}{k-1} \ln \frac{1}{\delta}$ bins.

4. Using FF, pack all the small items set aside in step 1, using new bins only if necessary.

How much time will this algorithm take? Assume that we will choose k to be a large constant. Let the t^{th} iteration start with the instance J_t and end with an instance J_{t+1}. By Lemma 3.11 we know that $m(J_{t+1}) = \mathrm{O}(SIZE(J_t)/k)$. Moreover, we know that $SIZE(J_{t+1}) \leq m(J_t)$ since the fractional solution $\vec{z}$ uses each of the m basic bin types at most once. From this it is easy to see that $m(J_t) \leq \frac{SIZE(J_1)}{k^{t-1}}$. We conclude that the number of iterations is bounded by $\mathrm{O}(\log SIZE(J))$ or $\mathrm{O}(\log n)$. Since each iteration and every other step runs in fully polynomial time, we have that the entire algorithm runs in fully polynomial time.

We present only an overview of the analysis of the number of bins used; the reader is referred to the original paper for complete details. Note that the main source of error is the brute force packing of G_1

into $k\Delta$ bins in each iteration. Since the number of iterations has been bounded above, we obtain that the total error in the packing is $O(k\Delta \log SIZE(I))$. Suppose we choose $\delta = 1/SIZE(I)$. Then we have that the total error is $O(\log^2 SIZE(I))$. This gives us the following theorem.

Theorem 3.5: *The algorithm $A(\delta, k)$, for $k > 2$ and $\delta = 1/SIZE(I)$, will take an instance I of BIN PACKING and in fully polynomial time produce a solution such that*

$$A(I) \leq OPT(I) + O\left(\log^2 OPT(I)\right)$$

3.4. Discussion

There are several variants of the bin packing problem, all of which are $\mathcal{NP}$-complete. In most of these cases, it is reasonably easy to come up with bounded ratio approximations. These variants can be classified under four different headings.

- packings in which the number of items per bin is bounded
- packings in which certain items cannot be packed into the same bin
- packings in which there are constraints (e.g. partial orders) on the way in which the items are packed
- dynamic packings in which items may be added and deleted

There are also some generalizations of the basic packing problem. Some examples are variable-sized bins and multi-dimensional bin packing. We refer the reader to the survey article by Coffman, Garey and Johnson [12] for further details. It is possible to devise approximation schemes for some of these cases, generally based on the ideas described here. An example is the approximation scheme for the case of variable-sized bins due to Murgolo [45]. Several open problems remain, most notably in the case of on-line bin packing and multi-dimensional bin packing. There is a big gap between the upper and lower bound on the achievable ratios for multi-dimensional bin packing – it is exponential in the dimension.

Problems

3–1 Consider the VECTOR PACKING problem which is a multi-dimensional version of the BIN PACKING problem.

Instance: A list of d-dimensional vectors $I = \{\vec{x}_1, \ldots, \vec{x}_n\}$ such that each component of each vector belongs to the interval $(0, 1]$.

Feasible Solution: A packing of these vectors into bins, where a bin can hold a collection of vectors if and only if the sum of these vectors is dominated by the all-ones vectors, i.e. each component of the sum is at most 1.

Goal: Minimize the number of bins used.

Using the techniques of Vega-Lueker (or, Karmakar-Karp), provide a polynomial time algorithm with a performance ratio of $d + \epsilon$. Notice that in the case of $d = 1$ this will reduce to the result of Vega-Lueker.

Chapter 4

Vertex Cover and Set Cover

SUMMARY: *Some problems are considered for which it is possible to attain a bounded ratio, without being able to have* $R_{MIN} = 1$ *in either the absolute or the asymptotic sense. These are a class of covering problems – vertex cover for graphs and hypergraphs. For the unweighted vertex cover problem in graphs, several algorithms are described which achieve a ratio of 2. Similar bounds are then obtained for the weighted version of this problem. The set cover problem turns out to be much harder to approximate and only a logarithmic performance ratio is obtained for it.*

We have seen several problems which can approximated to any degree, i.e. have $R_{MIN} = 1$ in either the absolute or the asymptotic sense. Now we turn our attention to problems for which we can attain some bounded ratio, without being able to push this ratio all the way down to 1. In most such cases the exact value of R_{MIN} is hard to pin down precisely, all we can say is that it is bounded from above by some constant. It would be great to find matching lower bounds on the value of R_{MIN}, but such bounds are hard to obtain.

A *vertex cover* of a graph is a set of vertices which contains at least one end-point of each edge. As we have seen earlier, this is closely related to cliques and independent sets. It will be convenient to view an edge in a graph as subset of the vertex set. This is justified since the graph is undirected. It enables us to unify the treatment of graphs and hypergraphs.

VERTEX COVER (VC)

- **[Instance]** Graph $G(V, E)$.
- **[Feasible Solutions]** A subset $C \subseteq V$ such that for all $e = \{u, v\} \in E$, $e \cap C \neq \emptyset$.
- **[Value]** The value of a solution is the size of the cover $|C|$, and the goal is to minimize it.

This problem is one of the standard $\mathcal{NP}$-complete problems [15]. As a matter of fact, the problem remains $\mathcal{NP}$-complete even when the graph is planar [16]. There are more general versions of this problem where we allow G to be a hypergraph, or associate weights with vertices.

WEIGHTED VERTEX COVER (WVC)

- **[Instance]** Graph $G(V, E)$ and a positive integer weight function $w : V \to \mathcal{Z}^+$ on the vertices.
- **[Feasible Solutions]** A subset $C \subseteq V$ such that for all $e = \{u, v\} \in E$, $e \cap C \neq \emptyset$.
- **[Value]** The value of a solution is the weight of the cover $w(C)^*$, and the goal is to minimize it.

SET COVER (SC)

- **[Instance]** Set $V = \{v_1, v_2, \ldots, v_n\}$ and a family of sets $E = \{e_1, e_2, \ldots, e_m\}$, such that each $e_i \subseteq V$.
- **[Feasible Solutions]** A subset $C \subseteq V$ such that for all $e_i \in E$, $e_i \cap C \neq \emptyset$.

*We will use the natural generalization of the weight function to a subset of its domain, i.e. $w(C) \triangleq \sum_{v \in C} w(v)$.

- **[Value]** The value of a solution is the size of the cover $|C|$, and the goal is to minimize it.

Notice that the last problem is exactly the vertex covering problem for a hypergraph. There is a natural generalization of the SET COVER problem to WEIGHTED SET COVER, but we will not deal with that problem in this book. It is obvious that all these generalizations of VC are also $\mathcal{NP}$-complete.

We present some observations about a vertex cover of a graph. The first of these was posed as an exercise in Chapter 2. The second follows from the observation that each edge in a matching has to be covered by a distinct vertex in C.

Fact 4.1: *A set $C \subseteq V$ is a vertex cover for the graph $G(V, E)$ if and only if the complement set $\bar{C} = V - C$ is an independent set of vertices in the graph G. Moreover, C is a* minimum vertex cover *for G if and only if $\bar{C}$ is a* maximum independent set *of vertices in G.*

Fact 4.2: *Let $M \subseteq E$ be a matching, or an independent set of edges, in G. Then G cannot contain a vertex cover of size smaller than $|M|$.*

We will use the following notation with regard to any input graph $G(V, E)$.

- $n = |V|$ and $m = |E|$.
- $\Gamma(v) \triangleq \{\, u \in V \mid \{u, v\} \in E \,\}$ will denote the set of neighbors of a vertex $v \in V$.
- $d_v \triangleq |\Gamma(v)|$ will denote the degree of the vertex v.
- $\Delta(G)$ will denote the maximum degree in the graph G.
- For any set $U \subseteq V$, the induced graph $G[U] = (U, E[U])$ will consist of the vertices U and the edges in E which are incident only on vertices in U.

We will refer to any instance of the WVC problem as (G, w), where w denotes the weight function. This will be referred to as the instance G if choice of the weight function is clear from the context, e.g. if the graph is unweighted.

Definition 4.1: *Given any instance (G, w) of WVC, $C^*(G, w)$ will denote an optimal solution to the problem. This will be abbreviated to C^*_G if weight function w is known from the context, and to C^* if the graph G is also fixed by the context. The value of the optimal solution will be denoted by $c^* = w(C^*)$.*

In the following sections we will present several approximation algorithms for the above problems. We will be considering nearly a half-dozen algorithms each of which is based on a distinct idea. One reason for this overly extensive coverage of the various algorithms is that some of the ideas appear to be extremely novel and may be exportable to other problems. Moreover, as we will see later, even a small improvement in the best-known approximation ratio for VC will have profound implications. It is curious, therefore, that we have several different algorithms which all achieve the same ratio (asymptotically 2) but there appears to be no way of improving this ratio at the present time.

4.1. Approximating Vertex Cover

We suggest that the uninitiated reader spend some time on trying to devise heuristic algorithms for the vertex cover problem before reading any further. It is probable that you will come up with most of the simple and natural heuristics described below.

The most natural heuristic is a greedy algorithm which repeatedly picks an edge that has not yet been covered, and places one of its endpoints in the current covering set – call this Algorithm $G1$. Let $G(V, E)$ be any instance of the unweighted vertex cover problem.

Did you think of this? Does it achieve a bounded ratio?

Algorithm G1:

Input: Unweighted graph $G(V, E)$.

Output: Vertex cover C.

1. $C \leftarrow \emptyset$;

2. while $E \neq \emptyset$ **do begin**
Pick any edge $e \in E$ and choose an end-point v of e;
$C \leftarrow C + v$;
$E \leftarrow E \setminus \{e \in E \mid v \in e\}$;

end ;

3. return C.

We leave it as an exercise to show that this algorithm always outputs a vertex cover. We claim that algorithm does not achieve any bounded ratio. To see this, consider the bipartite graph $B(L, R, E)$ depicted in Fig. 4.1. The vertex set L consists of r vertices. The vertex set R is further sub-divided into r sets called $R_1, \ldots, R_r$. Each vertex in R_i has an edge to i vertices in L and no two vertices in R_i have a common neighbor in L; thus, $|R_i| = \lfloor r/i \rfloor$. (It is possible that not all vertices of L have a neighbor in a particular R_i.) It follows that each vertex in L has degree at most r and each vertex in R_i has degree i. The total number of vertices n is $\Theta(r \log r)$.

Consider now the behavior of the greedy algorithm on the graph B. Suppose that (out of sheer bad luck) the algorithm considers all the edges out of R_r first, choosing their end-point in R as the vertex to be placed in the cover. Then it picks all the edges out of R_{r-1}, choosing their end-points in R for the cover C; and, so on. Therefore the vertex cover chosen is $C = R$. But L is itself a vertex cover since the graph is bipartite. It follows that the ratio achieved by this algorithm is no better than $|R|/|L| = \Omega(\log n)$.

Is the vertex cover problem any easier on a bipartite graph?

How do we achieve a better ratio than this? Let us try the obvious strategy of modifying the Algorithm $G1$ to be less arbitrary in its choice of vertices to be included in the cover. A natural modification is to

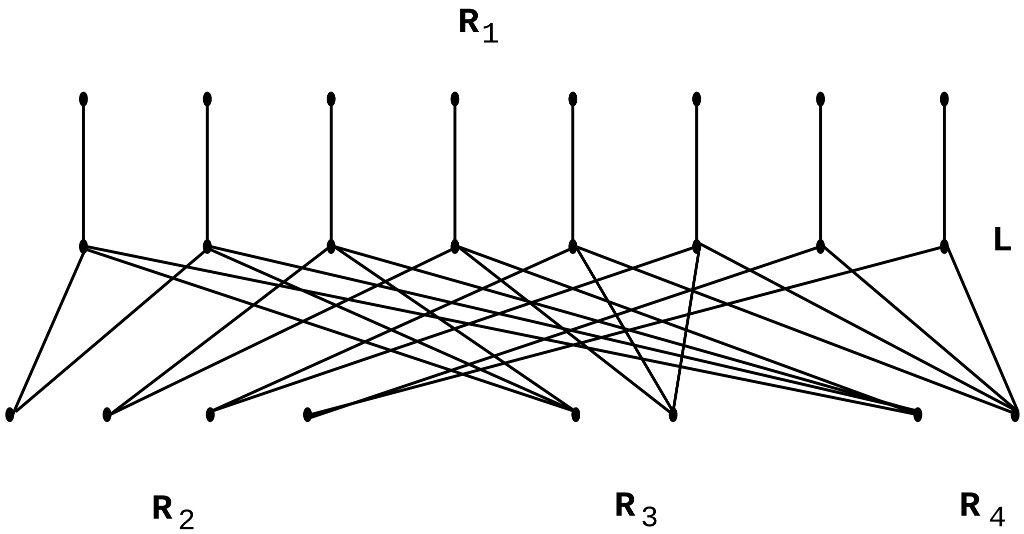

Figure 4.1: *The graph $B(L,R,E)$ with $r = 8$. Only the sets R_1, R_2, R_3 and R_4 are shown; the remaining sets R_5, R_6, R_7 and R_8 consist of one vertex each.*

repeatedly choose vertices which are incident to the largest number of *currently* uncovered edges – call this Algorithm $G2$.

Algorithm G2:

Input: Unweighted graph $G(V,E)$.

Output: Vertex cover C.

1. $C \leftarrow \emptyset$;

2. while $E \neq \emptyset$ **do begin**

Pick a vertex $v \in V$ of maximum degree in the *current* graph;
$C \leftarrow C + v$;
$E \leftarrow E \setminus \{e \mid v \in e\}$;

end ;

3. return C.

Let us consider the behavior of this algorithm on the graph $B(L, R, E)$. It should be easy to see that $G2$ could also output R as a vertex cover. It could choose vertices from R_r at the very first stage. After this, it could choose vertices from R_{r-1}. In general, it would choose the highest degree vertices from R at each stage. It is very surprising that a seemingly much more intelligent heuristic does no better than the rather simple-minded heuristic $G1$. However, as we will see later, this algorithm is not totally useless. It will be shown that it always achieves the ratio $O(\log n)$ for the much more general problem of set cover [30, 43], and hence also for vertex cover.

We now describe a different heuristic which achieves a bounded ratio for the vertex cover problem. The basic idea is to modify $G1$ by placing *both* end-points of some uncovered edge into C. Most people find the fact that this algorithm performs better than $G1$ and $G2$ to be very counter-intuitive at first. The surprisingly good performance of this algorithm can be better understood by considering an alternate description. Pick any *maximal* matching M in the graph $G(V, E)$. Place both end-points of each edge in M into the cover. We call this Algorithm MM. Note that a matching is maximal if it is not contained in any larger matching. It can be computed greedily – repeatedly choose an edge not incident on any currently matched vertex.

It may help to consider the behavior of this algorithm on the graph $B(L,R,E)$.

Algorithm MM:

Input: Unweighted graph $G(V, E)$.

Output: Vertex cover C.

1. Pick any maximal matching $M \subseteq E$ in G;

2. $C \leftarrow \{ v \mid v \text{ is matched in } M \}$;

3. return C.

Now let us try to see why this algorithm does well, contrary to our "intuition". Recall Fact 4.2 which gives a lower bound on the size of the optimal vertex cover in terms of the size of any matching. Algorithm MM can be viewed as first finding a lower bound on the optimal solution, and then constructing a solution which is *provably* within a small constant factor of this lower bound. Really, the goal of every approximation algorithm is exactly this – find a solution and a lower bound proof simultaneously. The "counter-intuitive" behavior of most approximation algorithms can be explained via the observation that it is trying to *prove* near-optimality. We now analyze the performance of MM; this result is due to Gavril [16].

Theorem 4.1: *Algorithm MM always computes a vertex cover in the input graph G. Moreover, $R_{MM} = 2$.*

Proof: The fact that M is a maximal matching implies that all edges in $E \setminus M$ are such that at least one of their end-points is matched in M. Otherwise, that edge could be added to M to provide a larger matching, contradicting the assumption that M is maximal. This implies that every edge in E has at least one end-point that is matched and hence that C is a vertex cover.

To see that the ratio is 2, consider the edges in M. To cover these edges we need at least $|M|$ vertices, since no two of them share a vertex. This implies that the optimal vertex cover has size at least $|M|$. The cover C contains exactly $2 \cdot |M|$ vertices.
□

Exercise 4.1: *Show that there exist input graphs for which the performance of MM is no better than a ratio of 2.*

Exercise 4.2: *Show that using a maximum matching instead of a maximal matching does not improve the worst-case performance of MM.*

Another algorithm which achieves a ratio of 2 for this problem is due to Savage [52]. This algorithm, which we call DFS, is as simple as the one outlined above. The basic idea is to find a depth-first search tree in the graph G. The cover C is then the set of non-leaf nodes in the tree. We leave the analysis of this algorithm as an exercise.

Exercise 4.3: *Show that the DFS algorithm always finds a vertex cover, and that its performance ratio is 2.*

This is asymptotically the best upper bound we have for $R_{MIN}(VC)$. Of course, it is entirely possible that one can find an approximation scheme for VC, but this is considered unlikely. We will provide some evidence for this later on. We conclude by describing a simple randomized algorithm due to Pitt [50] which also achieves the ratio 2 for VC, albeit in the expected sense.[†] One good reason for studying this algorithm is that it can be easily generalized to the case of weighted vertex cover to yield a simple approximation algorithm with an expected performance ratio of 2.

Once again, we suggest that the reader spend some time trying to think of randomized heuristics for the vertex cover problem. The most natural such heuristic is to consider the vertices in a random order, placing each vertex into the cover if it is incident on a currently uncovered edge. Unfortunately, this performs very poorly. To see this, consider the performance of this heuristic on the star graph, viz. a graph with one vertex of degree $n-1$ connected to $n-1$ vertices of degree 1 each. A more reasonable heuristic is a randomized version of $G2$. The idea is that instead of choosing the maximum degree vertex in the residual graph, pick a vertex at random such that the probability that any particular vertex is chosen is proportional to the number of uncovered edges incident on it. We leave it as an exercise to show that even this heuristic will not guarantee an expected ratio which is bounded. (*Hint:* Consider its performance on our old friend, the graph $B(L, R, E)$.)

[†] We generalize the notion of a performance ratio to randomized algorithms in the obvious manner. The expected ratio of a randomized approximation algorithm RA on a fixed input I is defined as $R_{RA}(I) \triangleq \frac{\mathrm{Exp}[RA(I)]}{OPT(I)}$, where $\mathrm{Exp}[RA(I)]$ denotes the expected value of RA's output. The expected performance ratio $R_{RA}(\Pi)$ is defined exactly as in the case of deterministic algorithms.

Quite surprisingly, a simple modification of $G1$ turns out to be the right algorithm. The idea is to consider the edges in some arbitrary (but fixed) order. If the edge currently under consideration is not already covered, pick one of its end-points uniformly at random and add it to the cover. We will refer to this new randomized algorithm as Algorithm RA.

Algorithm RA:

Input: Unweighted graph $G(V, E)$.

Output: Vertex cover C.

1. Order the edges in E arbitrarily;

2. while $E \neq \emptyset$ **do begin**

Pick the next edge $e = \{u, v\} \in E$;
Flip a fair coin to choose x uniformly from $\{u, v\}$;
$C \leftarrow C + x$;
$E \leftarrow E \setminus \{e \in E \mid x \in e\}$;

end ;

3. return C.

Before we formally analyze this algorithm, it is worthwhile to try to understand at an intuitive level why this algorithm should perform well. Observe that Algorithm $G1$ added an arbitrary end-point to the cover, while Algorithm MM added both end-points to the cover. One would expect that this randomized algorithm would have an intermediate performance, but it turns out to do as well as MM in the expected sense. One reason is that it avoids making the wrong choice of an end-point for an uncovered edge, unlike Algorithm $G1$. Moreover, a higher degree vertex has more chances of being chosen by a random coin flip.

Compare the behavior of RA and $G1$ on $B(L, R, E)$.

Theorem 4.2: *Algorithm RA always outputs a vertex cover and* $R_{RA} = 2$.

Proof: It is easy to verify that this algorithm will always output a vertex cover.

Prove this for yourself!

Let us fix an input graph $G(V, E)$, the order in which the edges are to be examined and some optimal cover $C^* \subseteq V$. Suppose that this algorithm outputs a cover C with t vertices in it. Clearly, this algorithm examines exactly t edges and flips as many coins in the course of its execution. Let us define the outcome of a coin flip as being *good* if it causes some vertex $v \in C^*$ to enter the cover C. Note that every edge has at least one end-point in C^* and so each coin flip is good with probability at least a half.

But the number of good coin flips cannot exceed $c^* = |C^*|$, since by then all the vertices of C^* are in C and every edge in G must be covered by C. Thus, the total number of coin flips t is stochastically dominated by the number of unbiased coin flips needed to obtain c^* good coin flips. It follows that the expected number of coin flips needed is no more than $2 \cdot c^*$. This implies the desired bound on the expected value of the performance ratio.
$\square$

4.2. Approximating Weighted Vertex Cover

We now turn our attention to the weighted version of the vertex cover problem. Let us start by considering all the obvious heuristics for this problem; as usual readers are urged to try and think of their own heuristics before proceeding any further.

Consider first the simple greedy heuristic which considers the vertices in increasing order of their weights, placing each vertex in the cover if it is incident on an edge which is currently uncovered. This heuristic becomes identical to Algorithm $G1$ when restricted to the case of unweighted graphs. Therefore it is not very surprising that it has a very poor performance ratio. In fact, its performance is much worse than that of $G1$, as illustrated by the following example. Consider the star graph where the vertex of degree $n-1$ has weight 2 and the degree one vertices all have weight 1 each. It is easy to see that the cover chosen by our heuristic will have weight $n-1$, as opposed to the optimal cover which has weight 2.

Another obvious heuristic is a simple generalization of Algorithm $G2$ which was presented in the previous section. The basic idea here is to choose at each stage a vertex for with the smallest possible ratio of weight to current degree. We will refer to this as Algorithm $WG2$.

Algorithm WG2:

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. $C \leftarrow \emptyset$;

2. while $E \neq \emptyset$ **do begin**

Pick any $v \in V$ which minimizes $\frac{w(v)}{d_v}$ with respect to the current graph;

$C \leftarrow C + v$;

$E \leftarrow E \setminus \{e \,|\, v \in e\}$;

end ;

3. return C.

It is easy to see that this is a generalization to the weighted case of the heuristic $G2$ from the previous section. As such, it cannot be expected to have a performance ratio better than that of $G2$, i.e. $\mathrm{O}(\log n)$. However, this is still a very natural heuristic and not without any merit. In fact, Chvatal [10] has shown that it achieves this ratio, and no better, for the much more general problem of weighted hypergraph covering or weighted set covering.

In the following sections we present several different approximation algorithms for WVC, all of these achieve the ratio 2. The first is a simple randomized algorithm due to Pitt [50]. In the subsequent sections we describe two deterministic approximation algorithms for WVC. One is a simple intuitive algorithm which is not very efficient, while the other achieves efficiency at the cost of being more mystifying. Some amount of history is in order at this point. The very first approximation algorithm for WVC was implicit in the work of Nemhauser and Trotter [46]. This algorithm was made explicit by Hochbaum [25]. Hochbaum [24] had devised an approximation algorithm for the set cover problem with a performance ratio equal to the size of the largest set. This implied a factor of 2 approximation for vertex cover. Both these results made extensive use of a linear programming formulation. The first purely combinatorial analysis was due to Bar-Yehuda and Even [4] and this was followed by the algorithm of Clarkson [11]. All of these algorithms have essentially the same performance ratio, i.e. asymptotically equal to 2. Some of these algorithms, e.g. the one due to Hochbaum [25], achieve a performance ratio of $2 - f(n)$, where $f(n)$ is a decreasing function of n. The best such algorithm is due to Bar-Yehuda and Even [6], as well as Monien and Speckenmeyer [44], and it achieves a ratio of $2 - \mathrm{O}\left(\frac{\log\log n}{\log n}\right)$. This marginal improvement turns out to be quite crucial as it leads to some strong results for graph coloring which will be presented later. The first deterministic algorithm we present is derived from the work of Nemhauser and Trotter, and the second is the one due to Clarkson. Finally, we will describe the algorithm of Bar-Yehuda and Even, and show how it encompasses most of the algorithms mentioned above.

4.2.1. A Randomized Approximation Algorithm

In this section we generalize the randomized algorithm RA described earlier to the weighted case. The basic idea remains the same, the only difference is in the bias of the coin flip made at each stage. We choose an end-point of the edge in consideration with probability inversely proportional to its weight. Notice that Algorithm WRA becomes exactly the Algorithm RA when restricted to unweighted graphs.

Algorithm WRA:

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. Order the edges in E arbitrarily;

2. while $E \neq \emptyset$ **do begin**
Pick the next edge $e = \{u, v\} \in E$;
Choose x randomly from $\{u, v\}$ such that
$\text{Prob}[x = u] = \frac{w(v)}{w(u)+w(v)}$;
$C \leftarrow C + x$;
$E \leftarrow E - e$;

end ;

3. return C.

Notice that this algorithm captures the advantages of Algorithm $WG2$ without allowing the possibility of consistently choosing the wrong end-point at each stage. In particular, high degree vertices have a higher chance of being chosen and at each stage the coin flip is biased in favor of the lower weight vertex. Thus, it is the ratio of the weight to the current degree which determines the chances of a vertex being selected at each stage. The following theorem is due to Pitt [50].

Theorem 4.3: $\text{Exp}[WRA(G, w)] \leq 2 \cdot c^*(G, w)$, *and this bound is tight.*

The rest of this section is devoted to the proof of this theorem. Let us fix the input instance (G, w), the order in which the edges are to be examined and some optimal weighted vertex cover $C^* \subseteq V$. Suppose that we now execute Algorithm WRA and obtain the cover $C \subseteq V$. Here C is a random subset of V and its distribution is totally determined *prior* to the execution of the algorithm.

Definition 4.2: *For each vertex v, define the random variable X_v as follows.*

$$X_v = \begin{cases} w(v) & \text{if } v \in C \\ 0 & \text{otherwise} \end{cases}$$

Let $e_v = \mathrm{Exp}[X_v]$ *denote its expected value.*

These denote the actual and expected contributions of the vertex v to the cover C. Again, the distribution of X_v and the value of e_v is totally determined prior to the execution of the algorithm. The value of WRA's output and its expected value can then be expressed as follows.

$$\begin{aligned} w(C) &= \sum_{v \in V} X_v \\ \mathrm{Exp}[w(C)] &= \sum_{v \in V} e_v \end{aligned}$$

Our goal is to appropriately generalize the analysis of Algorithm RA. There the idea was to consider vertices in C which were from C^*, and to show that these vertices formed a significant fraction of the vertices in C. We make use of a similar argument here.

Definition 4.3: *Let* $\widehat{C} = C^* \cap C$ *denote the vertices from the optimal cover which were also chosen by* WRA.

Since C^* was fixed prior to the execution of the algorithm, it is clear that $w(\widehat{C}) = \sum_{v \in C^*} X_v$, and hence that $\mathrm{Exp}[w(\widehat{C})] = \sum_{v \in C^*} e_v$. Moreover, since $\widehat{C} \subseteq C^*$, it follows that

$$\mathrm{Exp}[w(\widehat{C})] \leq w(C^*)$$

In the following lemma we show that the expected weight of the output C is at most twice the expected weight of $\widehat{C}$, and this combined with the above inequality implies the result in the theorem.

Lemma 4.1: $\mathrm{Exp}[w(C)] \leq 2 \cdot \mathrm{Exp}[w(\widehat{C})]$

Proof: The proof is best described in terms of a game played on the input graph. Suppose that each vertex $v \in V$ has e_v dollars as its initial capital. This capital is fixed *prior* to the execution of the algorithm. The total money supply in the graph initially is exactly the expected weight of the algorithm's output.

We assume for now that there exists a global strategy under which each vertex can distribute its capital to the incident edges such that each edge gets exactly the same amount of money from both its end-points. Having collected this money from its end-points, each edge returns partitions it equally among its end-points which actually belong to the optimal cover C^*. Thus, if both end-points belong to C^* then the edge just returns the amount it had received from each; on the other hand, if only one end-point belonged to C^* then it gets back twice the amount of money it had initially handed over to this edge.

It is not very hard to see that at the end of these transactions each vertex in C^* has at most doubled its fortune, while each vertex not in C^* has become bankrupt. From this it follows that the total money supply in G is at most twice the *initial* money supply in the control of the vertices from C^*. Recalling that each vertex v started off with a sum of money equal to e_v, we can interpret this as

$$\sum_{v \in V} e_v \leq 2 \cdot \sum_{v \in C^*} e_v$$

and this is equivalent to the statement of the lemma.

The only thing left to show is that the global strategy for distributing the capital to the edges exists. Why should such a strategy exist? Consider any vertex $v \in V$. This will be in the vertex cover if one of its incident edges chooses to place it there. Thus, the expected contribution of v to the vertex cover's weight is made up of the contributions due its incident edges selecting it to be in the cover. Moreover, each edge contributes an equal amount for both its end-point, given the choice of bias of the coin flips. (The preceding argument merely uses this "distribution strategy" to relate the weight of the cover C to that of the optimal cover in an obvious manner.)

We formalize the distribution strategy as follows. Call an edge *critical* if it is not yet covered by the time Algorithm WRA considers it. It is the critical edges which will cause a coin flip and the addition of a vertex to C. We say that a vertex u is *chosen* by a coin flip for the (critical) edge $\{u, v\}$ if the coin flip causes v to be added to C.

Definition 4.4: *For each vertex u, and each $v \in \Gamma(u)$, define the ran-*

dom variable

$$X_{u,v} = \begin{cases} w(u) & \textit{if } u \textit{ is chosen for } C \textit{ due to the critical edge } \{u,v\} \\ 0 & \textit{otherwise} \end{cases}$$

Notice that for each edge $\{u, v\}$, exactly one of $X_{u,v}$ and $X_{v,u}$ is non-zero. Further, for each vertex u at most one of its incident edges can "choose" it and so we have the following.

$$X_u = \sum_{v \in \Gamma(u)} X_{u,v}$$

From this we conclude that

$$e_u = \mathrm{Exp}[X_u] = \sum_{v \in \Gamma(u)} \mathrm{Exp}[X_{u,v}]$$

While at most one of the $X_{u,v}$'s can be non-zero for each u, every one of the expectations of these could be non-zero since the expectation is taken over all possible random choices made by WRA. Finally, we claim that $\mathrm{Exp}[X_{u,v}] = \mathrm{Exp}[X_{v,u}]$ for all edges $\{u, v\}$. This claim implies the existence of the desired distribution strategy. This is because each vertex u give can give a sum of money equal to $\mathrm{Exp}[X_{u,v}]$ to the edge $\{u, v\}$, and then each edge will receive the same amount of money from both its end-points.

To validate the claim we observe that the choice of the bias in each coin flip ensures symmetry between the expected contribution of the critical edges' two end-points. More formally,

$$\begin{aligned} \mathrm{Exp}[X_{u,v}] &= w(u) \times \mathrm{Prob}[\{u,v\} \text{ is critical and chooses } u] \\ &= w(u) \times \mathrm{Prob}[\{u,v\} \text{ is critical}] \times \frac{w(v)}{w(u)+w(v)} \\ &= w(v) \times \mathrm{Prob}[\{u,v\} \text{ is critical}] \times \frac{w(u)}{w(u)+w(v)} \\ &= w(u) \times \mathrm{Prob}[\{u,v\} \text{ is critical and chooses } v] \\ &= \mathrm{Exp}[X_{v,u}] \end{aligned}$$

□

4.2.2. The Nemhauser Trotter Algorithm

Nemhauser and Trotter considered an integer programming formulation of the WVC problem. There is a variable for each vertex which takes on values in $\{0, 1\}$; each edge creates a constraint that the variables associated with its end-points have sum at least 1. Any feasible solution to this set of constraints corresponds naturally to a vertex cover of the graph $G(V, E)$. The objective function is simply the weighted sum of the variables, where the weights are exactly the weights of the corresponding vertices. They showed that the optimal solution to the LP-relaxation of this problem has the semi-integral property. In other words, the basic feasible optimal solution to the corresponding linear programming relaxation would assign values to the variables which would be drawn from the set $\{0, \frac{1}{2}, 1\}$. The linear program's optimal solution could be interpreted as a *fractional* vertex cover – the value of a variable denoted the fraction of the corresponding vertex which should be placed in the cover. The constraints require that, for each edge, the total fraction of its end-points in a fractional cover should exceed 1. It then follows that the semi-integral solution can be used to obtain an approximation to the optimal *integral* cover by placing a vertex in the cover if the corresponding variable was non-zero. It is not very hard to see that the linear program can be solved via a maximum flow computation, and in the unweighted case it can be solved via a maximum matching algorithm.

We now present a combinatorial interpretation of this process and obtain an approximation algorithm which does not refer to the linear programming formulation. (Actually, the results of Nemhauser and Trotter were much more general but this has no bearing on the approximation of the WVC problem.)

Definition 4.5: *A 2-cover of a graph $G(V, E)$ is a* multiset $S \subseteq V$ *such that for every edge $e \in E$, $|e \cap S| \geq 2$.*

Essentially, a 2-cover is a multiset of vertices such that each edge has either both its end-points or at least two copies of one end-point in the multiset. We may assume, without loss of generality, that each vertex occurs at most twice in a 2-cover since we can throw away any further copies of a vertex without destroying the property of being a 2-cover. Observe that doubling the value of each variable in a semi-integral solution to the above linear program will yield a 2-cover. The relation between 2-covers and vertex covers is made explicit by the following fact; its proof is left as an exercise.

Definition 4.6: *Let $\bar{S}$ denote the set underlying the multiset S, i.e. the set obtained by retaining exactly one copy of each element of S.*

Fact 4.3: *If S is a 2-cover for G then $\bar{S}$ is a vertex cover for G.*

We define the weight of a 2-cover in the obvious way – it is the sum of the weights of the vertices in the 2-cover, with each weight being multiplied by the multiplicity of the corresponding vertex. An *optimal 2-cover* is a 2-cover of minimum weight. Notice that an optimal 2-cover will never have more than two copies of any vertex. Therefore, the set underlying a 2-cover will have total weight at least half that of the 2-cover itself. Moreover, taking a vertex cover and making two copies of each vertex in the cover will yield a 2-cover of at most twice the weight. We have proved the following lemma.

Lemma 4.2: *The weight of an optimal 2-cover is at most twice the weight of an optimal vertex cover.*

Therefore, to find a 2-approximation to the optimal weighted vertex cover in G, it suffices to find an optimal weighted 2-cover in G. It turns out that an optimal 2-cover can be found in polynomial time. The basic idea behind this is to consider a bipartite version of the input graph $G(V, E)$. In the bipartite graph there are two copies of each vertex in V of the same weight. one on each side of the bipartition. Each edge of E creates two edges in the bipartite graph.

Definition 4.7: *Let $G(V, E)$ be a graph and w a weight function on its vertices. Define the bipartite graph $B_G(L, R, F)$ such that*

- $L = \{v_i^L \mid v_i \in V\}$
- $R = \{v_i^R \mid v_i \in V\}$
- $F = \{\{v_i^L, v_j^R\}, \{v_j^L, v_i^R\} \mid \{v_i, v_j\} \in E\}$
- $w(v_i^L) = w(v_i^R) = w(v_i)$

Given any set of vertices $U \subseteq V$ in G, we will denote the copies of these vertices in L as U^L and the copies in R as U^R; thus, $L = V^L$ and $R = V^R$. Further, the vertices in G corresponding to a set of vertices $S \subseteq L \cup R$ from B will be denoted by S_G; thus, $L_G = R_G = (L \cup R)_G = V$.

Lemma 4.3: *Any vertex cover of B_G can be converted into a 2-cover of G of equal weight.*

Proof: Let $C \subseteq L \cup R$ be a vertex cover of B_G. We can construct a multiset $C' \subseteq V$ from C by replacing each vertex from $L \cup R$ by a copy of the corresponding vertex in V. Note that the underlying set for C' is exactly C_G. For each edge $e \in E$ we had placed two edges in F and both these edges must have at least one end-point each in C. This implies that e has either both end-points or two copies of one of its end-points in C'. It follows that C' is a 2-cover of G; its weight is trivially equal to the weight of C.
$\square$

The next lemma proves the converse of Lemma 4.3.

Lemma 4.4: *Any 2-cover C of G can be converted into a vertex cover of B_G of equal weight.*

Proof: Recall that we are only considering 2-covers which have at most two copies of each vertex. Let $C_1 = \bar{C}$ denote the set underlying the multiset C, and define $C_2 = C \setminus C_1$ as the set of vertices in C which occur twice. Let $C' = C_1^L \cup C_2^R$ consist of the vertices from L which correspond to the vertices in C_1, as well as the vertices from R which correspond to the vertices in C_2. Clearly, the sets C and C' have equal weights.

We now show that C' is a vertex cover of B_G. Consider any edge $e = \{u, v\} \in E$. If both u and v are in C, then both edges corresponding to e in F are covered as both u^L and v^L are in C'. Otherwise assume, without loss of generality, that u occurs twice in C. This means that both u^L and u^R are in C' and once again both edges corresponding to e in F are covered.
□

These two lemmas have shown that finding an optimal 2-cover of G is equivalent to finding an optimal vertex cover of B_G. Before we show how the latter can be done in polynomial time, let us summarize the Nemhauser-Trotter algorithm as follows.

Algorithm NT:

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. Compute the graph $B(L, R, F)$ from the input $G(V, E)$;

2. Compute an optimal weighted vertex cover C_B^* for B;

3. **return** $C = (C_B^*)_G = \{v \in V \mid v^L \in C_B^* \text{ or } v^R \in C_B^*\}$.

The preceding lemmas imply that C is a vertex cover for the graph G and its weight is at most twice that of the optimal weighted vertex cover for G. We have the following theorem.

Can you show that the bound given in the theorem is tight?

Theorem 4.4: $R_{NT} = 2$

We now briefly sketch an algorithm for finding an optimal weighted vertex cover in a bipartite graph. The WVC problem restricted to bipartite graphs is polynomially solvable via a reduction to the maximum flow problem [38]. This works by constructing a *directed* network from B_G. Introduce a source s into B_G with an edge going to each vertex in L of capacity equal to the weight of that vertex. Similarly, introduce a sink t with an edge coming in from each vertex of R of capacity equal to the weight of that vertex. Direct each edge in F from L to R and make its capacity infinite.

The minimum (s,t)-cut in the resulting network can be computed via a maximum flow computation. Moreover, the minimum cut must be finite since the net flow out of the source is finite. Thus, no edge of F can cross that cut. In other words, it cannot be the case for any edge (v_i^L, v_j^R) that v_i^L lies on the side of s and v_i^R on the side of t. Thus the set of vertices from L lying on the side of t, together with set of vertices from R lying on the side of s, forms a vertex cover for B_G. Further, the capacity of the cut is exactly equal to the weight of this vertex cover. Similarly, it is also fairly easy to see that any vertex cover implies a cut of capacity equal to the weight of the vertex cover. Thus, the vertex cover determined by the min-cut must be a minimum weight vertex cover.

4.2.3. Clarkson's Algorithm

Consider once again the greedy algorithm $WG2$ proposed earlier for the WVC problem. The basic idea in this algorithm was to keep track of the ratio between the weight and the *current* degree of a vertex, and at each stage it selected the vertex with smallest value of this ratio. This seemed like the right thing to do at an intuitive level as we would like to minimize the average increase in weight of the vertex cover per edge being covered. Unfortunately, this algorithm does not achieve any bounded ratio. But can anything be salvaged from this intuitively attractive heuristic? The answer is yes, and this is exactly the algorithm proposed by Clarkson [11]. His modified greedy algorithm (MGA) follows basically the same approach, except that the weights of the vertices are also modified as the algorithm progresses. (Recall that Algorithm $WG2$ was only modifying the degrees of the vertices to account for the edges which were already covered.) In the following description of Algorithm MGA ignore for now the edge cost function

EC, this is merely an artifact of the algorithm's analysis. We will use $W(v)$ and $D(v)$ to denote the current weight and degree of the vertex v at each point in the execution.

Algorithm MGA:

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. for all $v \in V$ **do** $W(v) \leftarrow w(v)$;

2. for all $v \in V$ **do** $D(v) \leftarrow d_v$;

3. for all $e \in E$ **do** $EC(e) \leftarrow 0$;

4. $C \leftarrow \emptyset$;

5. while $E \neq \emptyset$ **do begin**
Pick a vertex $v \in V$ for which $\frac{W(v)}{D(v)}$ is minimized;
$C \leftarrow C + v$; $V \leftarrow V - v$;
$W(v) \leftarrow 0$;
for all edges $e = \{u, v\} \in E$ **do begin**
$E \leftarrow E - e$;
$W(u) \leftarrow W(u) - \frac{W(v)}{D(v)}$; $D(u) \leftarrow D(u) - 1$;
$EC(e) \leftarrow \frac{W(v)}{D(v)}$;
end ;

end ;

6. return C.

This algorithm differs from $WG2$ only in that each time a vertex is placed in the cover, each of its neighbors has its weight reduced by an amount equal to the ratio of the selected vertex's current weight and degree. This is exactly the cost of covering the edge between the two vertices and the value of EC reflects this cost. (Note that the value of

EC is never used by the algorithm.) This may seem counter-intuitive in that we are actually increasing the likelihood of picking a vertex whose neighbor has just been included in the cover. But an approximation algorithm is not trying to pick an optimal solution. Instead it tries to pick a solution which is *provably* not very far from the optimum, The reduction in the weights of the neighbors can be viewed as an attempt to ensure that the "error" made by this algorithm is small. In fact, this reduction in the weights is exactly what will enable us to argue that the algorithm's output is not too far from the optimal.

The argument presented below proceeds as follows. The edge cost $EC(e)$ is viewed as the cost of covering the edge e. The algorithm assigns costs to the edges in a manner which guarantees that each vertex in the cover partitions its weight amongst the incident edges, and each edge gets assigned the same weight from both its end-points. Thus, the weight of the cover being produced is at most twice the net cost of the edges. Under any such choice of the edge cost function, it can be easily seen that an optimal cover must have weight at least as large as the total of the edge costs. It should now be clear that the "counter-intuitive" part of the algorithm is actually a device for ensuring that the cover being produced does not stray too far from the optimal cover. For a further discussion on this point, refer to the article by Gusfield and Pitt [22].

Fact 4.4: *For all vertices $v \in V$ and edges $e \in E$*

$$W(v) \geq 0$$

$$EC(e) \geq 0$$

at all times during the execution of the algorithm.

Proof: The second inequality is obvious since the only modification to the edge costs is the addition of a positive value. As for the first inequality, notice that the current weight of a vertex is reduced only when some other vertex (in fact, its neighbor) is selected. But this implies that the selected vertex has a smaller weight to degree ratio, and the result of subtraction must be non-negative.
□

The next fact is easy to verify from the description of the algorithm.

Fact 4.5: *For all vertices $v \in V$*

$$w(v) = W(v) + \sum_{u \in \Gamma(v)} EC(u,v)$$

at all times during the execution of the algorithm.

The next fact follows from the description of the algorithm and Fact 4.5.

Fact 4.6: *At the end of the algorithm's execution*

$$\forall v \in C,\ \ w(v) = \sum_{u \in \Gamma(v)} EC(u,v) \tag{4.1}$$

$$\forall v \notin C,\ \ w(v) \geq \sum_{u \in \Gamma(v)} EC(u,v) \tag{4.2}$$

From the above facts we conclude the following lemma which relates the weight of MGA's output to the book-keeping variables of edge costs.

Lemma 4.5: $w(C) \leq 2 \cdot \sum_{e \in E} EC(e)$

Proof: Observe that by the equation (4.1)

$$w(C) = \sum_{v \in C} w(v) = \sum_{v \in C} \sum_{u \in \Gamma(v)} EC(u,v)$$

Each edge in E is counted at most twice in the last expression, implying the desired result.
□

The next step is to relate the edge costs to the value of the optimal solution.

Lemma 4.6: $\sum_{e \in E} EC(e) \leq c^* = w(C^*)$

Proof: First, observe that

$$\sum_{e \in E} EC(e) \leq \sum_{v \in C^*} \sum_{u \in \Gamma(v)} EC(u,v)$$

since the second expression must count each edge at least once, as C^* is a vertex cover. Using Fact 4.6, we now have the desired result. □

Putting together these two lemmas, we have the result that the weight of C is at most twice optimal. It is fairly easy to see that the entire algorithm can be efficiently implemented using standard data structures. We have the following theorem; showing that the bound of 2 on the performance ratio is the best possible is left as an exercise.

Theorem 4.5: *Algorithm MGA runs in time* $\mathrm{O}(m \log n)$ *time and has* $R_{MGA} = 2$.

4.3. Improved Vertex Cover Approximations

In this section we present some algorithms which marginally improve the approximation ratio for WVC (and VC). These algorithms do not achieve a ratio which is better than 2 in the asymptotic sense. Their performance ratios are of the type $2 - f(n)$, where $f(n)$ is some decreasing function of n. The function $f(n)$ could be $1/\sqrt(n)$ or $1/\Delta$, where Δ is the maximum degree in the input graph. The best such result is due to Bar-Yehuda and Even [6], and Monien and Speckenmeyer [44]; they achieve a ratio of $2 - \frac{\log \log n}{2 \log n}$. (This improvement may seem very minor but it leads to a significant improvement in the approximation ratio for the graph coloring problem which will be considered in a subsequent chapter.) For example, on graphs with at most 10^{12} vertices the ratio achieved is 1.9. We will present the version of this result due to Bar-Yehuda and Even.

In the following sections we will develop the main ideas behind the this result in three parts. First we will return to the Nemhauser-Trotter algorithm and show that it allows us to restrict ourselves to approximating WVC on graphs where the optimal solution has a large weight. Then we will present a "Local-Ratio Theorem" which allows us to strip off a certain kind of subgraph H from the input graph without adversely affecting the approximability of the remaining graph. Finally, we will show that in a graph without small odd cycles, the vertex cover can be well approximated provided the optimum solution is of large weight. The removal of odd cycles is performed by using the Local-Ratio Theorem.

4.3.1. The Nemhauser-Trotter Algorithm Revisited

We take a fresh look at the Nemhauser-Trotter algorithm presented in Section 4.2.2 and conclude that it suffices to be able to approximate WVC on instances where the value of the optimal solution is at most $w(V)/2$.

Recall that the optimal weighted vertex cover in $B(L, R, F)$ was C_B^*, and that this could be computed in polynomial time. Let C_0 contain the vertices $v \in V$ such that both v^L and v^R are in this optimal cover, and let V_0 be the vertices $v \in V$ such that only one of v^L and v^R is in the optimal cover.

$$C_0 \triangleq \left\{v \in V \mid \{v^L, v^R\} \subseteq C_B^*\right\}$$

$$V_0 \triangleq \left\{v \notin C_0 \mid \{v^L, v^R\} \cap C_B^* \neq \emptyset\right\}$$

The following theorem is a re-statement of the results of Nemhauser and Trotter, and we provide a different proof from theirs. The first two parts of the theorem are referred to as the *local optimality conditions.*

Theorem 4.6: *The sets C_0 and V_0 produced by Algorithm NT have the following properties.*

1. *If $D \subseteq V_0$ is a vertex cover for $G[V_0]$, then $C = D \cup C_0$ is a vertex cover for G.*
2. *There exists some optimal cover C^* for G such that $C_0 \subseteq C^*$.*

3. *The optimal solution for $G[V_0]$ has weight at least half as much as the total weight of the vertices in V_0, i.e.* $c^*(G[V_0], w) \geq \frac{w(V_0)}{2}$.

Proof:

1. We already know that $C_0 \cup V_0$ is a vertex cover for G. In fact, this is exactly the vertex cover which is produced by Algorithm NT. The set V_0 only covers the edges in G which have at least one end-point in V_0. Therefore, C_0 is a vertex cover for $G[V \setminus V_0]$ and it is clear that $C = C_0 \cup D$ covers all edges in $E[V \setminus V_0] \cup E[V_0]$.

   Consider any edge $\{x, y\}$ in E such that $x \in V_0$ and $y \notin V_0$. This is the only type of edge which could create a problem. Our choice of x and y implies that only one of x^L and x^R is contained in C_B^*. Assume, without loss of generality, that x^L is the one contained in C_B^*. Then the edge $\{y^L, x^R\}$ in B could only have been covered by C_B^* by having $y^L \in C_B^*$. Since $y \notin V_0$, it must be the case that $y \in C_0$. But this implies that $C = C_0 \cup D$ covers the edge $\{x, y\}$ and we are done.

2. Let S be some optimal cover for G. We claim that $C^* = C_0 \cup (S \cap V_0)$ is also an optimal cover for G; this will validate the second part of the theorem since $C_0 \subseteq C^*$. Observe that $S \cap V_0$ is a vertex cover of $G[V_0]$ and so, by the previous part of the theorem, we have that C^* is a vertex cover of G.

   To see the optimality of C^*, first observe that $C_B = (V_0 \cup C_0 \cup S)^L \cup (S \cap C_0)^R$ is a vertex cover for B. Consider any edge $\{x^L, y^R\}$ in B. It is covered by C_B if $x \in C_0 \cup V_0 \cup S$ or if $y \in S \cap C_0$. Assume then that neither of these two conditions is met. Since $x \notin C_0 \cup V_0$, the cover C_B^* must have covered the edges $\{x^L, y^R\}$ and $\{y^L, x^R\}$ by containing both y^L and y^R. Then it must be the case that $y \in C_0$. Since $y^R \notin C_0 \cap S$, we then have that $y \notin S$. But this gives a contradiction, since we now have both $x, y \notin S$ and the edge $\{x, y\}$ is not covered by the vertex cover S of G.

   Given that C_B is a vertex cover of B, it follows that its weight cannot be less then that of the optimal cover C_B^*.

$$\begin{array}{rrcl} & w(C_B^*) & \leq & w(C_B) \\ \Rightarrow & w(V_0) + 2 \cdot w(C_0) & \leq & w(V_0) + w(C_0) + w(S) - w(S \cap V_0) \\ \Rightarrow & w(C_0) & \leq & w(S) - w(S \cap V_0) \\ \Rightarrow & w(C_0) + w(S \cap V_0) & \leq & w(S) \end{array}$$

This implies that the $w(C^*) \leq w(S)$, and thus that C^* is an optimal cover for G which contains C_0.

3. Let D^* be an optimal cover for $G[V_0]$. Then, by the first part of the theorem, $C_0 \cup D^*$ is a vertex cover for G. It follows then that $(C_0 \cup D^*)^L \cup (C_0 \cup D^*)^R$ is a vertex cover for B. But this must have weight at least as large as that of the optimal cover C_B^*, which is exactly $w(V_0) + 2 \cdot w(C_0)$. Therefore,

$$2 \cdot (w(C_0) + w(D^*)) \geq w(V_0) + 2 \cdot w(C_0)$$

which implies that $2 \cdot w(D^*) \geq w(V_0)$.

□

Let us try to understand the implications of this theorem. It shows us how to compute, using a single max-flow computation, a subset V_0 of the vertices such that if we can compute an optimal vertex cover D^* in $G[V_0]$, then we can compute an optimal vertex cover in the graph G. In fact, this optimal vertex cover of G is nothing but $C_0 \cup D^*$, where the set C_0 is also provided by Algorithm NT. Of course, if we merely get an approximation within a ratio r of the optimal cover of $G[V_0]$, then that combined with C_0 also gives us an approximation within the ratio r for the entire graph G. We have established the following corollary.

Make sure you see why the first two parts of theorem imply these claims.

Corollary 4.1: *Let (G, w) be an instance of WVC. Algorithm NT computes subsets $C_0, V_0 \subseteq V$ such that if $D \subseteq V_0$ is an r-approximation to the optimal weighted vertex cover in $G[V_0]$, then $C_0 \cup V_0$ is an r-approximation to the optimal weighted vertex cover in G. Moreover, the optimal solution for $G[V_0]$ has value at least half as large as $w(V_0)$.*

By this corollary, we only need to worry about finding an approximation algorithm for instances of WVC where the value of the optimal solution is at least half of the total weight of the vertices. Notice that a trivial 2-approximation for $G[V_0]$ is simply the set of all vertices in that graph, which is V_0. This gives us an approximation algorithm A for WVC with ratio 2. The algorithm A is exactly the algorithm of Nemhauser and Trotter!

4.3.2. A Local Ratio Theorem

We are now going to describe a new technique for obtaining an approximation algorithm for WVC, this is due to Bar-Yehuda and Even [4, 6]. We first show that any partition of the weight function gives two instances of WVC whose optimal solutions yield an optimal solution for original instance.

Lemma 4.7: *Let $G(V, E)$ be a graph and w, w_0 and w_1 be any three weight functions on the vertex set of G, such that for all $v \in V$*

$$w(v) \geq w_0(v) + w_1(v)$$

If C^*, C_0^* and C_1^* are optimal weighted vertex covers for the instances (G, w), (G, w_0) and (G, w_1), then

$$w(C^*) \geq w_0(C_0^*) + w_1(C_1^*)$$

Proof:

$$\begin{aligned} w(C^*) &= \sum_{v \in C^*} w(v) \\ &\geq \sum_{v \in C^*} (w_0(v) + w_1(v)) \\ &= w_0(C^*) + w_1(C^*) \\ &\geq w_0(C_0^*) + w_1(C_1^*) \end{aligned}$$

The last inequality follows from the observation that C^* is a vertex cover for G and so its weight with respect to any weight function cannot be smaller than the weight of the optimal cover with respect to that weight function.

□

We apply this lemma as follows. Let $H(V_H, E_H)$ be any fixed graph. Suppose we find an *induced* subgraph of G isomorphic to H. We can determine the weight function w_1 such that it is non-zero only on the vertices in that subgraph, and the weight function w_0 is obtained by subtracting w_1 from w. By the lemma, finding optimal solutions with respect to the new instances gives us an optimal solution for the original instance. In fact, we will show that for a suitable choice of H we can make strong claims about the approximative behavior also. Let Algorithm A be any approximation algorithm for WVC. Our approach will be to run Algorithm A on the instance (G, w_0), and handle the instance (G, w_1) separately. Let us formalize the decomposition as the following algorithm which is parametrized by the choice of H.

Algorithm LOCAL(H):

Input: Graph $G(V, E)$ with weights w. It is assumed that H and A have been fixed in advance.

Output: Vertex cover C.

1. Find a set of vertices $U \subseteq V$ such that the induced subgraph $G[U]$ is isomorphic to H;

2. $\delta \leftarrow \min_{v \in U} w(v)$;

3. Choose the weight function w_0 as follows:

$$\forall v \in V,\ w_0(v) = \begin{cases} w(v) - \delta & \text{if } v \in U \\ w(v) & \text{otherwise} \end{cases}$$

4. Run Algorithm A on instance (G, w_0) to obtain a vertex cover C for G;

5. return C.

At this point there are several questions which may arise in the readers mind. How do we choose H and what is this Algorithm A? We defer the answers to these questions. We first show that for any choice of H and A, the quality of the approximation produced by Algorithm $LOCAL(H)$ can be well characterized. Notice that we are not worrying about the instance corresponding to (H, δ) which is subtracted off from (G, w). We are merely studying the ratio to the optimal of the cover C which is produced by the invocation of Algorithm A on the instance (G, w_0). Observe also that (H, δ) is really an instance of VC since all the vertex weights are identically δ.

Definition 4.8: *Given any fixed graph H, let $|V_H| = n_H$ and c_H^* be the size of an optimal (unweighted) vertex cover of H. Define the local ratio for H as $r_H = \frac{n_H}{c_H^*}$.*

For example, if the graph H is a cycle on $2k+1$ vertices then $n_H = 2k+1$, $c_H^* = k+1$ and $r_H = 2 - \frac{1}{k+1}$. The following theorem bounds the approximative ratio of the cover C produced by Algorithm $LOCAL(H)$.

Theorem 4.7: $R_{LOCAL(H)}(G, w) \leq \max\{r_H,\ R_A(G, w_0)\}$

Proof: Let r be the larger of r_H and $R_A(G, w_0)$. Also, let c_0^* denote the value of the optimal solution for the instance (G, w_0). Since $|C \cap V'| \leq |V'| = n_H$, we have that

$$\begin{aligned}
w(C) &\leq w_0(C) + \delta n_H \\
&\leq R_A(G, w_0) \cdot c_0^* + \delta r_H c_H^* \\
&\leq r \cdot (c_0^* + \delta c_H^*) \\
&\leq r \cdot c^*(G, w)
\end{aligned}$$

The last inequality can be obtained from the preceding lemma as follows. Observe that the value of the optimal solution for (H, δ) is simply

δ times the size of the optimal unweighted vertex cover for H, i.e. δc^*_H. We claim that this is also the weight of the optimal solution for the instance (G, w_1), where $w_1 = w - w_0$ is non-zero only on vertices in U. This is because the optimal cover for (G, w_1) can be obtained by augmenting the optimal unweighted cover of $G[U]$ by all the vertices in $V \setminus U$, which are of weight 0. It follows that w, w_0 and w_1 satisfy the premise of the lemma.
□

The last theorem is also referred to as the *Local Ratio Theorem.* It is not very hard to see that this idea can be generalized to any class of graphs, rather than just one graph H. Let $\mathcal{H}$ denote a family of graphs. We define $r_{\mathcal{H}} = \max\{r_H \mid H \in \mathcal{H}\}$. We now present a new algorithm called $MLOCAL(\mathcal{H})$. The basic idea is to enumerate all induced subgraphs of G which are isomorphic to some graph in $\mathcal{H}$, and apply the operation of reducing the weights exactly as in Algorithm $LOCAL(H)$. Finally, all vertices of weight 0 are set aside, and Algorithm A is applied to the remaining graph.

Algorithm MLOCAL($\mathcal{H}$):

Input: Graph $G(V, E)$ with weights w. It is assumed that $\mathcal{H}$ and A have been fixed in advance.

Output: Vertex cover C.

1. for all $v \in V$ **do** $w_0(v) \leftarrow w(v)$;

2. for all $U \subseteq V$ such that $G[U]$ is isomorphic to some $H \in \mathcal{H}$ **do**
  begin
    $\delta \leftarrow \min\{w_0(v) \mid v \in U\}$;
    for all vertices $v \in U$ **do** $w_0(v) \leftarrow w_0(v) - \delta$;

end ;

3. $C_1 \leftarrow \{v \in V \mid w_0(v) = 0\}$;

4. $V_1 \leftarrow V \setminus C_1$;

5. Run Algorithm A on instance $(G[V_1], w_0)$ to obtain a vertex cover C_2 for $G[V_1]$;

6. return $C = C_1 \cup C_2$.

The exact implementation of the Step 2 is deliberately left unspecified. The intent is that the iterations be applied by using some enumeration of all induced subgraphs of G isomorphic to graphs in $\mathcal{H}$. The exact ordering in the enumeration is irrelevant and can be chosen to make the algorithm more efficient. One way to do this is to enumerate all sets $U \subseteq V$ such that $|U| \leq \max\{n_H \mid H \in \mathcal{H}\}$, and then to check if $G[U]$ is isomorphic to some graph in $\mathcal{H}$. This can be done in polynomial time provided the number of vertices in the graphs in $\mathcal{H}$ is fixed independent of $|V|$. We will see later that for a well-structured class of graphs $\mathcal{H}$ we can relax this requirement.

At the end of Step 2, it will be the case that in every induced subgraph isomorphic to a graph in $\mathcal{H}$, at least one vertex will have the w_0-weight equal to 0. This means that every such subgraph will have at least one vertex in C_1. We conclude that the remaining graph $G[V_1]$ cannot have any subgraph isomorphic to a graph in $\mathcal{H}$. It is now clear why this algorithm is useful: for an appropriate choice of $\mathcal{H}$ it will be easier to guarantee that a near-optimal cover can be easily found in $G[V_1]$. In other words, Algorithm A has to perform well only on inputs which do not have any subgraphs isomorphic to graphs in $\mathcal{H}$. Notice that there are two ways in which we are constrained in the choice of $\mathcal{H}$: it must have $r_{\mathcal{H}} \leq 2$ and the enumeration in the Step 2 should be easy to perform.

As for the quality of the approximation produced by this algorithm, we present the following result called the *Local Ratio Corollary.* The proof is by a simple induction on the number of iterations in the Step 2, using the Local Ratio Theorem on each iteration. We leave the proof as an exercise.

Corollary 4.2: *Using any family $\mathcal{H}$ and any approximation algorithm A*

- $R_{MLOCAL(\mathcal{H})}(G, w) \leq \max\{r_{\mathcal{H}}, R_A(G[V_1], w_0)\}$.

- $G[V_1]$ *does not have any subgraphs isomorphic to graphs in* $\mathcal{H}$.

We now present several applications of this result. Consider first the case where the family $\mathcal{H}$ contains only the graph H, and H is simply an edge. It is then the case that $r_{\mathcal{H}} = 2$. Applying the above corollary, we can show that $R_{LOCAL(\mathcal{H})} = 2$. Moreover, the graph $G[V_1]$ must be empty since it cannot have any induced subgraphs isomorphic to an edge. There is no need for an Algorithm A in this case. The following is an equivalent description of the resulting algorithm. This is exactly the linear time approximation algorithm with ratio 2 that was devised by Bar-Yehuda and Even [4].

Algorithm MLOCAL($EDGE$**):**

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. while $\exists e \in E$ with both end-points of non-zero weight **do begin**
  Pick an edge $\{u, v\} \in E$ with both end-points of non-zero weight;
  $\delta \leftarrow \min\{w(u), w(v)\}$;
  $w(u) \leftarrow w(u) - \delta$;
  $w(v) \leftarrow w(v) - \delta$;

end ;

2. return $C = \{v \in V \mid w(v) = 0\}$.

Amazingly enough, almost every approximation algorithm (except Algorithm NT) can be viewed as some version of $MLOCAL(\mathcal{H})$. For another example, consider Algorithm MGA described in the previous section. This can be thought of as a generalization of $MLOCAL(EDGE)$ which picks several copies of the graph H (which is an edge) simultaneously, all sharing a common vertex. The choice of this common vertex is such that it is possible to subtract an equal amount of weight from all other edges without making their new weights negative. Clearly, this is merely an implementation detail and has no bearing on the ratio achieved.

Another application of the Local Ratio Theorem is in improving the performance ratio of the algorithm devised by Hochbaum [24]. Her algorithm was based on the following novel idea. First, run Algorithm NT to obtain an instance $G[V_0]$ with an optimal solution of weight at least half of $w(V_0)$. Suppose now that we can color the input graph G with k colors. Let $I \subseteq V_0$ be the color class which has the largest weight. Output $C = V_0 - I$ as the vertex cover. It is clear that C is a vertex cover since each color class is an independent set, and the complement of any independent set is a vertex cover. Moreover, by our choice of I, $w(I) \geq w(V_0)/k$ and this implies that

$$\frac{w(C)}{w(C^*)} \leq \frac{w(V_0) - w(I)}{w(V_0)/2} \leq 2 - \frac{2}{k}$$

Prove that any graph is Δ-colorable.

Since any graph can be colored with Δ colors, it follows that we have obtained an approximation algorithm with a performance ratio of $2-\frac{2}{\Delta}$. In the special case of planar graphs, we can improve the ratio to 1.5 by noting that every planar graph can be 4-colored [2, 3].

We now observe that the approximate graph coloring algorithm of Wigderson [60] (which we will see in a later chapter) will color a graph using at most $2\sqrt{n}$ colors provided it is triangle-free. This helps in improving the algorithm of Hochbaum, in conjunction with the use of the Local Ratio Corollary. The idea is to choose $\mathcal{H}$ containing only one graph, viz. the triangle graph. Now we run $MLOCAL$ on the input graph G to obtain a triangle-free graph. Next we run Algorithm NT to obtain a graph which is both triangle-free and has an optimal solution of value at least half of the total weight of the graph. At this point we can run Wigderson's algorithm to obtain a coloring using $k = 2\sqrt{n}$ colors. This implies an approximation ratio of $2 - \frac{1}{\sqrt{n}}$. The details of the analysis are fairly straightforward.

4.3.3. An Algorithm for Graphs Without Small Odd Cycles

We have seen how Algorithm NT and $MLOCAL(\mathcal{H})$ can be used to obtain several approximation algorithms with performance ratios of 2 or $2 - f(n)$. Bar-Yehuda and Even improved on all previously known performance ratios by combining these two algorithms in a particular fashion. The basic idea is to use $MLOCAL(\mathcal{H})$ to eliminate all odd cycles of small length. Then, by the use of Algorithm NT we guarantee that the graph has a large optimal vertex cover. Finally, a simple algorithm is used to obtain a good approximation in the resulting graph. We now present the latter algorithm.

Definition 4.9: *An instance (G, w) of WVC is said to be k-*proper *if the following conditions are satisfied.*

- $(2k-1)^k \geq n$.
- *G has no odd cycles of length smaller than $2k-1$.*
- $c^*(G, w) \geq \frac{w(V)}{2}$.

For $u, v \in V$, let $d(u, v)$ denote the distance from u to v in $G(V, E)$. The sets D_i represent the collection of vertices in V which are at a distance i from v. These can be determined in linear time by performing a breadth-first search starting at v.

Algorithm C_k finds an approximation to WVC in an instance (G, w) which is k-proper. The basic idea is to fix a vertex v and find sets B_t which contain all vertices at distance at most t from v, such that vertices in B_t are at an even distance from v if and only if t is even. For $t < k$, it is clear that each pair of vertices in B_t have an even length path joining them and they cannot be adjacent without creating an odd cycle of length at most $2k-1$. Since this is not possible for G, we obtain that B_t must be an independent set. Note that $X_t = B_{t-1} \cup B_t$ contains all the vertices at distance at most t from v. We now claim that for $t \leq k$, B_t covers all the edges which have at least one end-point in X_t. The algorithm chooses a value of t for which it can be guaranteed that the weight of $w(B_t)$ is a small fraction of $w(X_t)$. This is done by choosing the smallest value of t for which $w(B_t) \leq (2k-1)w(B_{t-1})$. The only problem is that t is required to be at most k. But if the weight of v is large, and the weight of each subsequent B_t keeps increasing by a factor of at least $2k-1$, it follows that we will exhaust all the vertices in the graph by the time $t = k$. Now the set X_t can be removed from the graph if we ensure that B_t is placed in the cover. The whole process is repeated till all the vertices have been removed.

Algorithm C_k:

Input: Graph $G(V, E)$ and weight function w on V, such that (G, w) is k-proper.

Output: Vertex cover C.

1. $U \leftarrow V$;

2. $C \leftarrow \emptyset$;

3. **while** $U \neq \emptyset$ **do begin**

Pick a vertex $v \in U$ such that $w(v) = \max_{u \in U} w(u)$;
for $0 \leq i \leq k$ **do** $D_i \leftarrow \{w \in V \mid d(v, w) = i\}$;
for $0 \leq t \leq \lfloor \frac{k}{2} \rfloor$ **do** $B_{2t} \leftarrow \cup_{i=0}^{t} D_{2i}$;
for $0 \leq t \leq \lfloor \frac{k-1}{2} \rfloor$ **do** $B_{2t+1} \leftarrow \cup_{i=0}^{t} D_{2i+1}$;
$f \leftarrow \min\{t \mid w(B_t) \leq (2k-1)w(B_{t-1})\}$;
$C \leftarrow C \cup B_f$;
$U \leftarrow U \setminus (B_f \cup B_{f-1})$;

end ;

4. **return** C.

It is obvious that this algorithm can be implemented to run in time polynomial in the size of the input (G, w). We obtain the following result about the output of this algorithm.

Theorem 4.8: *The set C produced by Algorithm C_k is a vertex cover for G and $R_{C_k} \leq 2 - \frac{1}{k}$.*

Proof: We first claim that $f \leq k$. To see this, note that for $t \leq f$, $w(B_t) > (2k-1)w(B_{t-1})$ which implies that

$$w(B_t) > (2k-1)^t w(B_0) = (2k-1)^t w(v)$$

Now, if $f > k$ then we have that $w(B_k) > (2k-1)^k w(v) > |V| \cdot w(v) \geq w(V)$, implying a contradiction.

Next, we claim that B_{f-1} is an independent set. Otherwise, there would be two vertices $x, y \in B_{f-1}$ which are adjacent. But the distance from v to x and y is either both even or both odd, given the definition of B_t. This implies the existence of an odd cycle containing v of length at most $2k-1$, which is not possible given that G is k-proper.

Consider any edge e with at least one end-point incident on $B_{f-1} \cup B_f$. If both end-points of e are in $B_{f-1} \cup B_f$, then B_f covers this edge since B_{f-1} is an independent set. On the other hand, if only one end-point of e lies in $B_{f-1} \cup B_f$, then it must lie in D_f since otherwise the other end-point would be at distance at most f from v and also lie in $B_{f-1} \cup B_f$. It follows that every edge incident on $B_{f-1} \cup B_f$ is covered by B_f. We now conclude that C must be a vertex cover for G.

It remains to bound the weight of this vertex cover. By definition, $w(B_f) \leq (2k-1)w(B_{f-1})$ or

$$w(B_f) \leq \left(1 - \frac{1}{2k}\right)(w(B_f) + w(B_{f-1}))$$

Moreover, at each iteration the set B_f is added to the cover while both sets B_f and B_{f-1} are deleted from the graph. It is now clear that $w(C) \leq \left(1 - \frac{1}{2k}\right) w(V)$.
□

4.3.4. The Overall Algorithm

The overall vertex cover algorithm can now be specified in terms of the algorithms $MLOCAL(\mathcal{H})$, NT and C_k. The following Algorithm

A takes as input any instance (G, w) of WVC. It can be thought of as the algorithm A used by $MLOCAL$, although we present the overall algorithm in a slightly different manner. Let C_r denote the graph which consists of a cycle on r vertices.

Algorithm BE:

Input: Graph $G(V, E)$ and weight function w on V.

Output: Vertex cover C.

1. Let k be the smallest integer such that $(2k-1)^k \geq n$;

2. $\mathcal{H} \leftarrow \{C_{2i+1} \mid 1 \leq i \leq k-1\}$;

3. Run Algorithm $MLOCAL(\mathcal{H})$ on (G, w) to obtain the $C_1 \subseteq V$ and a residual instance $(G[V_1], w_0)$;

4. Run Algorithm NT on $(G[V_1], w_0)$ to obtain the sets $C_0, V_0 \subseteq V$;

5. Run Algorithm C_k on the $(G[V_0], w_0)$ to obtain the cover C;

6. **return** $C \cup C_0 \cup C_1$.

It is fairly easy to see that the entire algorithm runs in polynomial time, provided that $MLOCAL(\mathcal{H})$ can be implemented in polynomial time. If we were to try and compute all possible odd cycles of length upto $2k-1$, the running time of $MLOCAL$ would be super-polynomial. Instead, we present a strategy for enumerating a small number of odd cycles such that, if at least one vertex in each such cycle has its weight reduced to 0, then there will not be any odd cycles of length at most $2k-1$ which contains only vertices of positive weight. Clearly, this is a valid implementation of $MLOCAL(\mathcal{H})$.

To enumerate these odd cycles, pick any node v of non-zero weight and construct a breadth-first tree rooted at that node. Any odd cycle of length $2r+1$ containing v must have two adjacent vertices at distance r from v. This implies that there must be a pair of nodes at level r of the tree which are adjacent. Moreover, any adjacent pair of nodes at level r determine an odd cycle containing v of length $2r+1$. If there exists any such pair of adjacent vertices at any level $l \leq k-1$, compute the nuique odd cycle of length $2l+1$ determined by the tree and these two vertices. Reduce the weights as specified by $MLOCAL$. If there is no such odd cycle containing v, then eliminate v from contention in any future iteration. Now repeat the whole process outlined above.

The claim is that at each iteration at least one vertex is eliminated from consideration as the root of a breadth-first tree, or at least one vertex has its weight reduced to 0 and is also eliminated. It follows that the number of iterations is at most n. Moreover, at the end of these iterations, the graph does not contain any odd cycles of length at most $2k-1$ which do not have vertices of weight 0.

Thus, the running time of the entire algorithm is polynomial in the size of the input. In fact, the running time is dominated by Algorithm NT which uses one max-flow computation.

It is also clear that this algorithm has a performance ratio of $2-\frac{1}{k}$. This can be formally verified by using the results proved in the previous sections for the algorithms NT, $MLOCAL$ and C_k. Note that our choice of k is such that $k = \mathrm{O}\left(\frac{\log n}{\log\log n}\right)$. We have the following result.

Theorem 4.9: *The algorithm BE computes a vertex cover in polynomial time such that $R_A = 2 - \frac{\log\log n}{2\log n}$.*

4.4. Approximating Set Cover

Let $H(V,E)$ be a hypergraph representing an instance of the (unweighted) set covering problem. We generalize the notion of the degree of a vertex to a hypergraph.

Definition 4.10: *For all $v \in V$, d_v is the number of edges in E which contain v. Also, let $d = d(H)$ be the maximum degree in the hypergraph H.*

As usual, a cover $C \subseteq V$ is a collection of vertices of the hypergraph such that each edge in E contains at least one vertex from C.

Definition 4.11: *$\tau(H)$ is the size of a minimum cover of the hypergraph H.*

There is no known constant factor approximation for the minimum cover of a hypergraph. In fact, there is some evidence to the effect that such an approximation is impossible to find in polynomial time. The best known approximation algorithm has a performance ratio of $\mathrm{O}(\log d)$, and this was independently discovered by Johnson [30] and Lovasz [43]. A similar result was achieved for the case of weighted hypergraphs by Chvatal [10]. We will present only the result for unweighted hypergraphs. The algorithm is essentially the greedy algorithm $G2$, as generalized to hypergraphs. We will also refer to this generalized algorithm as $G2$.

Algorithm G2:

Input: Hypergraph $H(V, E)$.

Output: Set cover C.

1. $C \leftarrow \emptyset$;

2. while $E \neq \emptyset$ **do begin**

Pick a vertex $v \in V$ of maximum degree in the *current* hypergraph;
$C \leftarrow C + v$;
$E \leftarrow E \setminus \{e \mid v \in e\}$;

end ;

3. return C.

The following presentation is based on that of Lovasz. We will need some further notation in the course of analyzing $G2$. A fractional cover of a hypergraph is essentially a feasible solution to the LP-relaxation of the integer programming formulation of the covering problem. It is a choice of a fraction of each vertex such that for every edge the total fraction of all its vertices selected is at least 1.

Definition 4.12: *A* fractional cover *of the hypergraph H is a weight function $w : V \to \Re^+$ such that for all edges $e \in E$*

$$\sum_{v \in e} w(v) \geq 1$$

Definition 4.13: *Let $\tau^*(H)$ denote the size of the* optimal *fractional cover of H, i.e.*

$$\tau^* = \min_w \sum_{v \in V} w(v)$$

A matching in a hypergraph is a natural generalization of a matching in a graph, i.e. it is a collection of independent edges. We can further generalize this to the notion of a k-matching, as follows.

Definition 4.14: *A* k-matching *in the hypergraph is a subset $M \subseteq E$ such that each vertex $v \in V$ is contained in at most k edges from M. In other words, it is a sub-hypergraph of degree at most k.*

Definition 4.15: *Let $m_k(H)$ denote the size (number of edges) of a maximum k-matching in the hypergraph H.*

For the sake of brevity, we will omit the dependence of d, τ, τ^* and m_k on the input hypergraph H, assuming that the input H has been fixed. We first present some elementary relations between these quantities. The first of these follows from the observation that every cover of H is also a fractional cover.

Fact 4.7: $\tau^* \leq \tau$

The next fact follows from linear programming duality, but we provide an elementary proof.

Fact 4.8: *For all k, $m_k \leq k\tau^*$.*

Proof: Let M be a maximum cardinality k-matching; then, $|M| = m_k$. Consider any optimal fractional cover w such that $\sum_{v \in V} w(v) = \tau^*$. Now we know that each edge in E, and hence each edge in M, has total weight at least 1 under w. Therefore, for all $e \in M$,

$$\begin{array}{rrcl} & \sum_{v \in e} w(v) & \geq & 1 \\ \Rightarrow & \sum_{e \in M} \sum_{v \in e} w(v) & \geq & |M| = m_k \end{array}$$

But, in the left-hand-side of the last inequality each vertex occurs at most k times. Therefore, we have that

$$\sum_{v \in V} k \cdot w(v) \geq m_k$$

Noting that $\tau^* = \sum_{v \in V} w(v)$, we have the desired result.
□

We are now ready to show that Algorithm $G2$ has a performance ratio of $\mathrm{O}(\log d)$. Suppose that Algorithm $G2$ chooses the vertices v_1, v_2, …, v_t, in that order, to produce a cover of size t. The following lemma bounds the value of t in terms of the matching numbers for H. Pay particular attention to the last term in the series.

Lemma 4.8:

$$t \leq \frac{m_1}{1 \cdot 2} + \frac{m_2}{2 \cdot 3} + \frac{m_3}{3 \cdot 4} + \cdots + \frac{m_{d-1}}{(d-1) \cdot d} + \frac{m_d}{d}$$

Proof: Note that v_1 has maximum degree in H, i.e. degree d, and that $m_d = |E|$. Observe that the number of new edges covered by each successive v_i is a non-increasing function of i. We will refer to the number of new edges covered by any such v_i as its *covering degree*. Let t_r be the number of times that algorithm selects a vertex of covering degree r in the course of its execution. Thus, among the v_i's, the first

t_d of them have covering degree d each, the next t_{d-1} of them have covering degree $d-1$ each, and so on. We conclude the following

$$\begin{aligned} t &= t_d + t_{d-1} + \cdots + t_2 + t_1 \\ |E| &= dt_d + (d-1)t_{d-1} + \cdots + 2t_2 + t_1 \end{aligned}$$

Let $H_i(V, E_i)$ denote the hypergraph defined by the collection of uncovered edges after $t_d + t_{d-1} + \cdots + t_{i+1}$ vertices have been selected by $G2$. Clearly, the maximum degree of each hypergraph H_i is at most i. This implies that E_i is an i-matching in H. Therefore,

$$m_i \geq |E_i|$$

Notice that all the edges of H_i were covered during the last $t_i + t_{i-1} + \cdots + t_2 + t_1$ iterations of $G2$. This gives us the following equation

$$|m_i| \geq |E_i| = it_i + (i-1)t_{i-1} + \cdots + 2t_2 + t_1$$

or that

$$m_i \geq \sum_{j=1}^{i} jt_j$$

Upon suitable algebraic manipulation, this yields the inequality stated in the lemma.
□

We are now ready to prove the main theorem.

Theorem 4.10: $R_{G2} < 1 + \log d$

Proof: From Facts 4.7 and 4.8, we have that

$$m_k \leq k\tau^* \leq k\tau$$

Combining this with the previous lemma, we obtain that

$$\begin{aligned} t &\leq \sum_{i=1}^{d-1} \frac{i\tau}{i(i+1)} + \frac{d\tau}{d} \\ &= \tau\left(\sum_{i=1}^{d} \frac{1}{i}\right) \\ &< (1 + \log d)\tau \end{aligned}$$

In other words,

$$G2(H) < (1 + \log d)OPT(H)$$

implying the desired result.

□

Exercise 4.4: *Show that the above bound on the performance ratio of $G2$ is the best possible.*

4.5. Discussion

Several of the algorithms described above seem to perform operations which are counter-intuitive. A good example is MGA which actually reduces the weights of the neighbors of the vertices already in the cover, thus increasing the likelihood that these neighbors are also selected to be in the cover. See the paper by Gusfield and Pitt [22] for a partial explanation of why such algorithms actually perform better than more intuitive algorithms such as $G2$. This also gives a more unified view of most of the algorithms considered above.

Hochbaum [25] gives bounded ratio approximation algorithms for related problems, viz. independent sets and coloring in *bounded* degree graphs and planar graphs. A result that we did not cover is the approximation algorithm for weighted set cover due to Chvatal [10]. The algorithm is a generalization of the greedy algorithm described above for set cover. The result in the case of set cover may be viewed as bounding the ratio of optimal integral cover and fractional cover for hypergraphs. See the paper by Aharoni, Erdös and Linial for a more general version of this result, i.e. a study of the ratio between the optimal fractional and integral solutions to a class of integer programs. A different version of the set cover was studied by Johnson [30]. Here, as before, the objective is to find a collection of vertices which cover all the edges but the value of a cover is now defined to be the sum of the degrees of the vertices in the cover, rather than the size of the cover. The results obtained are very similar to those described above for the set cover problem.

Problems

4–1 Recall the result proved in an earlier chapter which showed that there is no absolute approximation algorithm for CLIQUE, assuming that $\mathcal{P} \neq \mathcal{NP}$. Prove a similar result for SET COVER.

4–2 Consider the algorithm MGA for WEIGHTED VERTEX COVERING due to Clarkson. Prove the following variation of the result presented in class. Given an unweighted graph $G(V, E)$ with maximum degree Δ such that the optimal vertex cover is of size at most $n/3$,

$$R_{MGA}(G) \leq 2 - \frac{2}{\Delta - 2}$$

4–3 We have seen the greedy algorithm of Lovasz guarantees a $1+\log d$ factor approximation for the SET COVER problem. Prove that this is the best bound possible in that there exist instances where this bound is achieved by the greedy algorithm. Can you prove a similar result for the greedy algorithm on WEIGHTED VERTEX COVER?

4–4 Consider the problem called RECTANGLE COVERING or RC.

Instance: A collection of rectangles $I = \{R_1, \ldots, R_n\}$ in the plane such that each rectangle is aligned with the axes – all sides are horizontal or perpendicular. Note that the rectangles may overlap.

Feasible Solution: A collection of points $P = \{p_1, \ldots, p_m\}$ such that each rectangle in I contains at leas one point from P.

Goal: Minimize $|P|$.

Provide the best approximation algorithm you can for this problem. Can you say anything about the hardness of approximating this problem?

Bibliography

[1] R. Aharoni, P. Erdös and N. Linial, *Optima of dual integer programs*, Combinatorica, 8 (1988), pp. 13–20.

[2] K. Appel and W. Haken, *Every planar map is four colorable, Part I: Discharging*, Illinois Journal of Mathematics, 21 (1977), pp. 429–490.

[3] K. Appel, W. Haken and J. Koch, *Every planar map is four colorable, Part II: Reducibility*, Illinois Journal of Mathematics, 21 (1977), pp. 491–567.

[4] R. Bar-Yehuda and S. Even, *A Linear Time Approximation Algorithm for the Weighted Vertex Cover Problem*, Journal of Algorithms, 2 (1981), pp. 198–203.

[5] R. Bar-Yehuda and S. Even, *On Approximating a Vertex Cover for Planar Graphs*, Proceedings of 14th Annual ACM Symposium on Theory of Computing (1982), pp. 303-309.

[6] R. Bar-Yehuda and S. Even, *A Local-Ratio Theorem for Approximating the Weighted Vertex Cover Problem*, Annals of Discrete Mathematics, 25 (1985), pp. 27–45.

[7] M.W. Bern, H.J. Karloff, P. Raghavan, and B. Schieber, *Fast geometric approximation techniques and geometric embedding problems*, Proceedings of Fifth Annual Symposium on Computational Geometry (1989), pp. 292–301.

[8] J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, North-Holland (1976).

[9] N. Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Technical Report, Graduate School of

Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA (1976).

[10] V. Chvatal, *A Greedy Heuristic for the Set-Covering Problem*, Mathematics of Operations Research, 4 (1979), pp. 233–235.

[11] K.L. Clarkson, *A Modification of the Greedy Algorithm for Vertex Cover*, Information Processing Letters, 16 (1983), pp. 23–25.

[12] E.G. Coffman, M.R. Garey and D.S. Johnson, *Approximation algorithms for bin packing – an updated survey*, in Algorithm Design for Computer System Design (ed. G. Ausiello, M. Lucertini and P. Serafini), Springer-Verlag (1984).

[13] M.R. Garey and D.S. Johnson, *Approximation algorithms for combinatorial problems: an annotated bibliography*, in Algorithms and Complexity: New Directions and Recent Results (ed. J.F. Traub), Academic Press (1976).

[14] M.R. Garey and D.S. Johnson, *Strong NP-completeness Results: Motivations, Examples and Implications*, Journal of the ACM, 25 (1978), pp. 499–508.

[15] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company (1979).

[16] F. Gavril, *cited in [15], page 134.*

[17] P.C. Gilmore and R.E. Gomory, *A linear programming approach to the cutting-stock problem*, Operations Research, 9 (1961), pp. 849–859.

[18] R.L. Graham, *Bounds for certain multiprocessing anomalies*, Bell Systems Technical Journal, 45 (1966), pp. 1563–1581.

[19] M. Grötschel, L. Lovász and A. Schrijver, *The ellipsoid method and its consequences in combinatorial optimization*, Combinatorica, 1 (1981), pp. 169–197.

[20] M. Grötschel, L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag (1987).

[21] L. Guibas, *Personal communication*, 1992.

[22] D. Gusfield and L. Pitt, *Understanding approximations for node cover and other subset selection algorithms*, Technical Report YaleU/DCS/TR-308, Department of Computer Science, Yale University, 1984.

[23] M.D. Hansen, *Approximation Algorithms for Geometric Embeddings in the Plane with Applications to Parallel Processing Problems*, Proceedings of 30th Annual Symposium on Foundations of Computer Science (1989), pp. 604–611.

[24] D.S. Hochbaum, *Approximation Algorithms for Set Covering and Vertex Cover Problems*, SIAM Journal on Computing, 11 (1982), pp. 555-556.

[25] D.S. Hochbaum, *Efficient Bounds for the Stable Set, Vertex Cover and Set Packing Problems*, Discrete Applied Mathematics, 6 (1983), pp. 243-254.

[26] I. Holyer, *The NP-completeness of edge coloring*, SIAM Journal of Computing, 10 (1981), pp. 718–720.

[27] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press (1978).

[28] O.H. Ibarra and C.E. Kim, *Fast approximation algorithms for the knapsack and sum of subset problems*, Journal of the ACM, 22 (1975), pp. 463–468.

[29] D.S. Johnson, *The NP-completeness column: an ongoing guide*, Journal of Algorithms, 3 (1982), pp. 288–300.

[30] D.S. Johnson, *Approximation algorithms for combinatorial problems*, Journal of Computer and System Sciences, 9 (1974), pp. 256–278.

[31] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey and R.L. Graham, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing, 3 (1974), pp. 299–325.

[32] R. Kannan and B. Korte, *Approximative Combinatorial Algorithms*, Mathematical Programming 1984 (ed. R.W. Cottle, M.L. Kelmanson and B. Korte), pp. 195–248.

[33] N. Karmakar, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.

[34] N. Karmakar and R.M. Karp, *An Efficient Approximation Scheme For The One-Dimensional Bin Packing Problem*, Proceedings of 23rd Annual Symposium on Foundations of Computer Science (1982), pp. 312–320.

[35] R.M. Karp, *The fast approximate solution of hard combinatorial problems*, Proceedings of 6th Southeastern Conference on Combinatorics, Graph Theory and Computing, Utilitas Mathematics (1975), pp. 15–31.

[36] B. Korte and R. Schrader, *On the existence of fast approximation schemes*, Nonlinear Programming, 4 (1980), pp. 415–437.

[37] L. Kucera, *The complexity of clique finding algorithms*, unpublished manuscript.

[38] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart & Winston (1976).

[39] E.L. Lawler, *Fast Approximation Algorithms for Knapsack Problems*, Proceedings of 18th Annual Symposium on Foundations of Computer Science (1977), pp. 206–213.

[40] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, *Sequencing and Scheduling: Algorithms and Complexity*, in Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory (1990).

[41] H.W. Lenstra, *Integer programming with a fixed number of variables*, Mathematics of Operations Research, 8 (1983), pp. 538–548.

[42] R.J. Lipton and R.E. Tarjan, *Applications of a planar separator theorem*, Proceedings of the 18th Annual Symposium on Foundations of Computer Science (1977), pp. 162–170.

[43] L. Lovász, *On the Ratio of Optimal Integral and Fractional Covers*, Discrete Mathematics, 13 (1975), pp. 383–390.

[44] B. Monien and E. Speckenmeyer, *Ramsey Numbers and an Approximation Algorithm for the Vertex Cover Problem*, Acta Informatica, 22 (1985), pp. 115–123.

[45] F.D. Murgolo, *An efficient approximation scheme for variable-sized bin packing*, SIAM Journal on Computing, 16 (1987), pp. 149–161.

[46] G.L. Nemhauser and L.E. Trotter, Jr., *Vertex Packing: Structural Properties and Algorithms*, Mathematical Programming, 8 (1975), pp. 232-248.

[47] R.G. Nigmatullin, *Complexity of the approximate solution of combinatorial problems*, Soviet Mathematical Doklady, 16 (1975), pp. 1199–1203.

[48] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall (1982).

[49] V.Th. Paschos, *A Theorem on the Approximation of Set Cover and Vertex Cover*, to appear in Eleventh Conference on Foundations of Software Technology and Theoretical Computer Scienceai (FSTTCS 11), New Delhi (India), 1991.

[50] L. Pitt, *A Simple Probabilistic Approximation Algorithm for Vertex Cover*, Technical Report YaleU/DCS/TR-404, Department of Computer Science, Yale University, 1985.

[51] D.J. Rosenkrantz, R.E. Stearns and P.M. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing, 6 (1977), pp. 563–581.

[52] C. Savage, *Depth First Search and the Vertex Cover Problem*, Information Processing Letters, 14 (1982).

[53] S. Sahni, *Approximate algorithms for the 0/1 knapsack problem*, Journal of the ACM, 22 (1975), pp. 115-124.

[54] S. Sahni, *General Techniques for Combinatorial Approximation*, Operations Research, 25 (1977), pp. 920–936.

[55] S. Sahni and T. Gonzalez, *P-complete approximation problems*, Journal of the ACM, 23 (1976), pp. 555-565.

[56] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons (1986).

[57] P.M. Vaidya, *Geometry helps in matching*, SIAM Journal on Computing, 18 (1989), pp. 1201–1225.

[58] P.M. Vaidya, *Approximate minimum weight matching on points in k-dimensional space*, Algorithmica (1990).

[59] W. Fernandez de la Vega and G.S. Lueker, *Bin Packing can be solved within* $1 + \epsilon$ *in Linear Time*, Combinatorica, 1 (1981), pp. 349–355.

[60] A. Wigderson, *Improving the Performance Guarantee for Approximate Graph Coloring*, Journal of the ACM, 30 (1983), pp. 729–735.