APPROXIMATION ALGORITHMS FOR VARIATIONS OF THE NP-HARD STORED PROGRAM PROBLEM

Boston University
Graduate School
Master's Thesis

Charles H. Connell Jr.
B.A. Hampshire College 1979

April 1984

Submitted in partial fulfillment of the Master's of Arts degree in Computer Science at the Graduate School of Boston University.

Steven Homer: _____ Date: _____

A. J. Kfoury: _____ Date: _____

I am indebted to Steve Homer and Dennis Kfoury for all their time and encouragement, to my father for giving me a love of science, to Boris Khazanov and Gregory Piatetsky-Shapiro for providing an excellent work environment and putting up with my studies, and to all my friends for their support and worthwhile distractions.

APPROXIMATION ALGORITHMS FOR VARIATIONS OF THE NP-HARD STORED PROGRAM PROBLEM

(Order No. _____)

Charles H. Connell Jr.
Boston University, Graduate School, 1984
Major Professor: Steven Homer Professor of: Computer Science

Abstract

This paper will examine variations of a known NP-hard optimization problem which has an absolute approximation algorithm - the Stored Program Problem. I present two main results. First I expand the known problem by finding relaxations of it that still maintain the viability of the approximation algorithm. Then I consider a variation where the known algorithm does not hold and prove that *no* approximation algorithm is possible for this problem.

In the first section I review known results about complexity theory and approximation.

The second section presents the NP-hard problem that this thesis centers around (the Stored Program Problem). I describe the problem, give a known approximation algorithm for it, and prove that the algorithm provides absolute approximation.

In the third section I discuss new results which relax the Stored Program Problem in ways that still allow the approximation algorithm to work. There are few known NP-hard problems with absolute approximation - the extensions to this one expand that set.

The fourth section looks at an on-line modification of the Stored Program Problem and considers if the non-online algorithm can be extended to it. I show that it cannot. I then prove that this On-line Stored Program Problem cannot have any approximation algorithm, even allowing epsilon approximation.

The thesis closes with two appendices that discuss interesting side issues. The first presents

an alternative to the algorithm of section 2. The second presents notes toward a lower bound proof of the algorithm in 2.

TABLE OF CONTENTS

1. Background 4
1.1. NP-hard problems 4
1.2. Fast solutions for NP-hard problems 7
1.3. Approximation algorithms 8
1.4. Absolute approximation algorithms 10
2. The Stored Program Problem 11
2.1. The stored program problem is NP-hard 12
2.2. An absolute approximation algorithm for the stored program problem 12
2.3. Terminology 13
2.4. Proof of 2.2 13
2.5. Additional notes on the stored program problem 14
3. Variations of the Stored Program Problem 16
3.1. SPP may use bins of different sizes 16
3.2. Differing sized bins may be re-ordered in SPP 17
3.3. It is not necessary that each object fit in every bin 18
4. An On-line Version of the Stored Program Problem 20
4.1. On-line algorithm #1 22
4.2. On-line algorithm #2 23
4.3. There is no approximation algorithm for the on-line stored program problem 24
I. A Second Algorithm for the Stored Program Problem 27
I.1. The second algorithm 27
I.2. Analysis of the algorithm 28
II. Notes on a Lower Bound Proof 29

1. BACKGROUND

1.1. NP-hard problems

Complexity theory is concerned with the broad classification of the difficulty of problems. It seeks to distinguish sets of problems which require substantially different degrees of computational resources. Two of the classes of problems that have emerged from complexity theory are *P* and *NP*.

The class *P* contains all problems that can be solved by a Turing machine in time bounded by a polynomial on the length of the problem's statement. *P* is an important group of problems because it captures formally the intuitive notion of problems that can be realistically solved by computers.

The class *NP* is all problems for which a proposed solution can be verified in polynomial time (again this is a polynomial on the length of the problem's statement). This definition may appear to be equivalent to the one given for *P*, however there is an important difference. Problems in *P* can be *solved* in polynomial time. Problems in *NP* only require that a proposed answer can be *checked* to see if it is correct in polynomial time; it may take very long to actually find the answer to be checked. Relating this definition of polynomial time verification to the name "NP", this class of problems can alternatively be defined by a non-deterministic Turing machine. A problem is in *NP* if:

1. Instances of the problem can be encoded as a (possibly infinite) set of strings.
2. The problem can be stated as a yes/no question concerning these strings.
3. We define a language *L* to be the set of strings from above for which the answer to the problem is "yes".
4. There is a non-deterministic Turing machine which accepts exactly the strings in *L*.
5. The time required for the minimum length accepting computation (over all the accepting computations) is bounded by a polynomial on the length of the input string.

These two definitions are equivalent since the non-deterministic Turing machine can be considered to "guess" at an answer, then attempt to verify it in polynomial time.

The relationship between the classes *P* and *NP* is of considerable interest. Any problem in *P* can be restated as a problem asking a yes/no question. When restated this way, all problems in *P* are in *NP*. Any problem that can be solved in polynomial time can have its solution verified in polynomial time. It is widely believed, but not proven, that *NP* contains some additional problems which are outside of *P*. If this is true, there are problems for which an answer can be verified quickly (*NP*), but for which there is no polynomial time way to *find* a solution. There is strong circumstantial evidence that this is the case. A discussion of *P* and *NP* can be found by Garey and Johnson [79].

A third notion from complexity theory that is of importance is the idea of **polynomial reducibility**. Problem A is polynomially reducible to problem B if:

1. Both problems can be stated as yes/no questions.
2. For each problem, there is a set of strings that encodes instances of the problem.
3. Language L_a is the set of strings encoding problem A for which the answer to A is "yes". Language L_b is the set of strings encoding problem B for which the answer to B is "yes".
4. There is a polynomial time function f which maps strings in L_a to strings in L_b.
5. A string x is in L_a if and only if $f(x)$ is in L_b

The notion of polynomial reducibility is important because it provides a way of comparing the complexity of different problems. If a problem is polynomially reducible to another, then we know that the second is at least as hard as the first. This is intuitively clear. Assume there is an f as above transforming problem A to problem B. Given a method for solving B, we can use it to solve A. We take an instance of A and use f to transform it to an instance of B. We then solve problem B. Since f runs in polynomial time, this gives a method for

solving A that is within a polynomial of the time for solving B. If two problems can be polynomially reduced to each other, we know that they are equivalently hard.

Polynomial reducibility has been used to define two sets of especially hard problems. The first is the set of problems which are in *NP* and are sufficiently complex that no problem in *NP* is harder. This set is called **NP-complete**. Formally, if a problem is in *NP* and all problems in *NP* can be polynomially reduced to it then the problem is NP-complete.

We also define a larger set known as **NP-hard** problems. These are problems which may be in *NP* or may be harder than *NP*, but for which it has been shown that all *NP* problems are polynomially reducible to them. Sahni and Horowitz [78] provide an excellent discussion of these concepts.

It should be pointed out again that it is not known that all NP-complete and NP-hard problems are actually hard. It has not been proved that *P* <> *NP*. If *P* does equal *NP* then all NP-complete and some NP-hard problems are easy. It is not likely that this is the case however.

Problems that are NP-complete have the form of **decision problems**. That is, there is some question which requires a yes/no answer. Problems that are NP-hard are often **optimization problems**. These problems require an answer to a question such as, "What is the smallest (or largest) number that satisfies the following problem?". My thesis will be concerned with the latter group, NP-hard optimization problems.

In practical terms, the important fact about a problem that is NP-hard is that it is outside of the set *P* (again assuming that *P* <> *NP*). Using this assumption, since *P* includes all problems that are realistically solvable, NP-hard problems are not realistically solvable. The

running time of algorithms known to solve them is exponential.[1] For all but the smallest problem instances their solutions take longer than anyone would want to wait. There are many real problems that fall into this category however - practical ways to cope with them are necessary.

1.2. Fast solutions for NP-hard problems

Any algorithm that is to be of practical value must return an answer within a reasonable amount of time. If the running time of an algorithm is not bounded by some fairly small polynomial, its running time will be unacceptable. We make this restriction on algorithms that we consider from here on. For the remainder of this thesis an **algorithm** is an algorithm that runs in time bounded by a polynomial on the length of its input, and the degree of the polynomial should be small.

Two types of fast algorithms have been devised to cope with the hard problems described above.

First, it may be possible to design a algorithm which has a very good chance of getting the correct answer to a decision or optimization problem. If an algorithm can get the right solution 99% of the time, it will be of significant practical value. Such algorithms may return an arbitrarily bad answer in the cases where they are not correct, but possibly this happens for only a small fraction of problem instances. This type of algorithm, which is known to find the right answer almost all the time, is called a **probabilistically good algorithm.**

Another approach to hard problems is to find an algorithm that offers no particular promise about how often it will get the exactly right answer, but instead is guaranteed to find an answer within a specified error. It may be the case that such an algorithm never returns an

---

[1] A typical running time function for an NP-hard problem is 2^n. It is clear that for even a problem with input length 100 the time to compute this is very large.

exact answer, but if its solution is always within a certain percentage of the exact solution, it also can be of practical value. We call these algorithms **approximation algorithms.** It is clear that this approach applies only to optimization problems. There is no meaning to finding a yes/no answer to a decision problem within a certain percentage error; the answer is either right or wrong. Later sections of this thesis will examine some of these approximation algorithms for NP-hard problems.

1.3. Approximation algorithms

Approximation algorithms such as those discussed above can offer various guarantees about their performance. There are generally two types of guarantees that these algorithms promise to achieve. First, they may always return an answer that is within a given percentage of the optimal answer. As an example, if an algorithm promises to be within 20% of the best answer for a minimization problem,[2] and the optimum is 100, then the algorithm will always return a solution that is between 100 and 120. Formally, there is some ϵ and some n such that for all problem instances where the solution is larger than n the following holds: $|approx\_solut - opt\_solut| \; / \; opt\_solut <= \epsilon$. Clearly, the smaller ϵ an approximation algorithm can promise, the better it is.

The statement that the guarantee is required only for problems larger than some n is to remove a difficulty with very small answers. If the optimal answer to a problem is 3 and the approximate solution is 2, then $\epsilon = 0.33$. This may obscure the fact however that the approximation algorithm performs very well and returns a solution with $\epsilon < 0.01$ for all large problems.

For maximization problems we additionally require that ϵ be less than 1. This is necessary because without this restriction all maximization problems have a trivial approximation algorithm. The algorithm just returns the value 0, assuring $\epsilon = 1$ for all instances.

---

[2] A minimization problem is an optimization problem where we want the solution to be as small as possible, and a maximization problem is one where we want the answer to be as large as possible.

Algorithms of the above type are called **epsilon approximation algorithms** and are by far the most common type of approximation algorithm.

A second, better, kind of approximation algorithm gives a solution that is always within an additive constant of the optimum. Continuing the example above, if an algorithm promises to be within 5 of optimum for a minimization problem, and the optimum is 100, the algorithm will find an answer between 100 and 105. It is clear that this is a much stronger algorithm. As solution values grow large an epsilon approximation algorithm may be farther and farther from the true answer. If the optimum for some problem is 1000, the above epsilon approximation algorithm will find an answer between 1000 and 1200. An algorithm of this second type will provide a solution between 1000 and 1005. Formally, there is some k such that $|approx\_solut - opt\_solut| <= k$ for all problem instances. An algorithm that can provide this guarantee is an **absolute approximation algorithm.** Section 2 will examine a problem with this kind of approximation algorithm.

Not all NP-hard problems have the approximation algorithms described above. For some problems it can be shown that unless $P = NP$ there cannot be *any* approximation algorithms of either type. Section 4 contains such a result. For other problems it may be shown that one of these kinds of algorithms is not possible (assuming $P <> NP$). In other cases some restrictions on the possible approximation algorithms can be found. For example, Garey and Johnson [76a] show that unless $P = NP$ there is no epsilon approximation for general vertex coloring with $\epsilon < 1$.[3]

In many cases results such as these are not known. It is often an open question whether a problem can have approximation algorithms and, if so, of what strength.

---

[3] This is a minimization problem so $\epsilon >= 1$ is allowed.

1.4. Absolute approximation algorithms

The problem that this thesis centers around is an NP-hard maximization problem for which an absolute approximation algorithm is known. The number of such problems is very small. My research found only two others: vertex coloring for planar graphs, and edge coloring for regular graphs (where every vertex has the same degree).

Minimum vertex coloring for planar graphs (an NP-hard problem) is in this class by the following demonstration. (Horowitz and Sahni [78])

- The number of colors needed to color a planar graph so that no edge is incident to two vertices with the same color is at most 4.
- A polynomial time algorithm can determine if the number of colors needed for a particular instance is 0, 1 or 2.
- We construct an algorithm which checks (in polynomial time) if the number of colors needed by an instance is 0, 1 or 2. If the answer is one of these numbers, the algorithm returns that answer. If it is not 0, 1 or 2, the algorithm returns a 3.
- The answer returned is either correct or wrong by at most 1.

Determining the minimum number of colors for the edges of a regular graph so that no vertex terminates two edges of the same color is also NP-hard. It is also solvable by an absolute approximation algorithm though in the following way. (Leven and Galil [83] and Johnson [84])

- The number of colors needed to color the edges of a regular graph of degree k is either k or $k+1$.
- The degree of a regular graph can be found in polynomial time.
- The algorithm finds the degree of the graph, k, and returns it as the answer.
- The solution returned is either correct or off by 1.

In subsequent sections I will discuss a third known problem in this class, the Stored Program Problem.

2. THE STORED PROGRAM PROBLEM

The stored program problem is described by Horowitz and Sahni [78]. Assume that we have two disk drives each with a storage capacity of L, and a number of programs to be stored P_1 ... P_n. P_i has a size l_i. We want to store as many of the programs as possible on the disks, with the restriction that a program must be entirely on one disk.

It is immediately obvious that this problem has many re-statements other than a concern for storing programs on disks. The most obvious is bin packing. Since the problem does not allow a program to be split among the disks, it is natural to restate it using physical objects. Consider each disk to be a bin of size L, and each program P_i to be an object of size l_i. We ask for the maximum number of objects that can be stored in the two bins. Throughout this paper I will refer to the problem as the "stored program problem" or "SPP" but discuss it in terms of "bins" and "objects" rather than "disk drives" and "programs".

One difference should be pointed out right away about the relationship between the stored program problem and more common bin packing problems. In standard bin packing problems we have a set of objects and a large number of bins. The goal is to pack *all* of the objects into as few bins as possible. We assume that there are as many bins as needed to do this. The value to be optimized is a minimum for the number of bins used. The stored program problem is quite different. There is a fixed number of bins (2 for now) and we want to put as many objects as possible in them. We try to find a maximum on the number of objects. The difference is significant.

There is no known absolute approximation algorithm for standard bin packing. The best algorithms that are known provide epsilon approximation. For the stored program problem there is an absolute approximation algorithm (2.2).

2.1. The stored program problem is NP-hard

The stored program problem is NP-hard. This can be demonstrated by showing that a known NP-hard problem is polynomially reducible to the stored program problem. The following demonstration is due primarily to Horowitz and Sahni [78], with a few changes by myself.

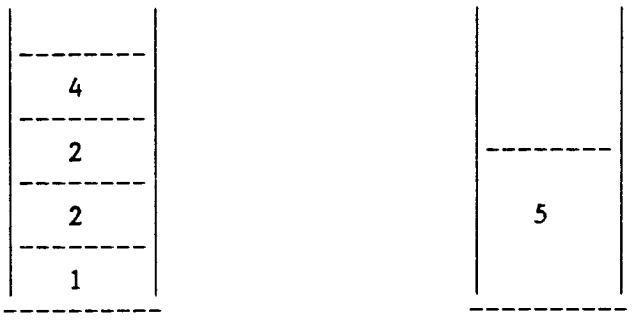
- The partition problem is to choose a subset S', if there is one, from a set of integers S such that $\Sigma\ S' = \Sigma\ S-S'$. This problem is NP-hard [Garey and Johnson 79].
- Let $\{a_1, a_2 \ldots a_n\}$ be an instance of the partition problem.
- We transform the partition problem to the stored program problem by the following method:
  * Set L (disk or bin capacity) of SPP equal to $(\Sigma\ S)\ /\ 2$ (from the partition problem). If $(\Sigma\ S)\ /\ 2$ is not an integer there is no partition in this instance of the partition problem and the problem is solved.
  * Set $l_1, l_2 \ldots l_n$ of the stored program problem equal to $a_1, a_2 \ldots a_n$ of the partition problem.
- The partition problem has a partition if the optimal solution to the SPP is that all of the programs will fit on the disks.

2.2. An absolute approximation algorithm for the stored program problem

Horowitz and Sahni [78] present the following approximation algorithm for SPP.

- Sort the objects by non-decreasing size (ascending order).
- Put the first object in the first bin, put the next object on top of it, and the next on top of it, etc. until the first bin is full.
- Put the next object from the sorted list as the first item in the second bin, then continue adding objects in order from the list until no more will fit.

An example will illustrate the application of this algorithm. Consider objects with sizes 1, 2, 2, 4, 5, 6 and bins of size 10. An optimal packing could fit all the objects; placing 1, 2, 2 and 5 in the first bin, and 4 and 6 in the second bin. The algorithm presented above would pack the objects as shown.



This algorithm will always provide a packing which uses the greatest possible number of objects, or one less than that. It is an absolute approximation algorithm for an NP-hard problem. A proof of this claim is below.

2.3. Terminology

I will use the following terminology in the proof.

- The objects are labeled $O_1 \ldots O_n$ after being sorted. So O_1 has no object smaller than it, O_n has no object larger than it. Let l_i denote the size of O_i.
- The objects that the algorithm packs into the first bin will be considered $O_1 \ldots O_i$, and the objects in the second bin as $O_j \ldots O_k$.
- Let the empty space that is left at the top of the first bin (after the algorithm has run) be S_1 and the empty space in the second bin be S_2.

2.4. Proof of 2.2

Algorithm 2.2 provides absolute approximation within 1 for the (2 Bin) Stored Program Problem. The proof that follows is a variation by myself of the one presented by Horowitz and Sahni [78].

- Assume that we also have a larger bin of size $2L$.
- Observe that the number of objects that can be packed into this single larger bin is greater than or equal to the number that can be packed into the original two bins. (Fact #1)

- Observe that the problem of packing the greatest number of objects in the single large bin is solvable by placing objects in by non-decreasing size.

- Take the objects that are packed into the two smaller bins by algorithm 2.2 and put them (in the same order) into the single large bin. The last object placed in the large bin is now O_k.

- It may be the case that another object, O_{k+1}, will fit on top of O_k in the large bin. If this happens it may be that algorithm 2.2 did not provide an optimal packing and that O_{k+1} could have fit in the original bins also.

- It cannot be the case however that *two* more objects will fit into the larger bin; O_{k+1} and O_{k+2} cannot fit on top of O_k in the bin of size $2L$.

  * Assume that they can. Then the sum of the unused spaces on top of the bins from 2.2 is greater than or equal to the sum of the sizes of the two extra objects; $S_1 + S_2 >= l_{k+1} + l_{k+2}$.

  * If the sum of the unused spaces in the original bins is greater than or equal to the sum of two additional objects, then the larger of those spaces could have held the smaller of the objects. This is restated formally below.

  * $S_1+S_2 >= l_{k+1}+l_{k+2}$ --> $MAX(S_1, S_2) >= l_{k+1}$.

  * But algorithm 2.2 stopped because no more objects would fit, so the assumption must be wrong.

- If two more objects cannot fit into the single large bin, two more objects cannot fit into the two original bins (by Fact #1).

2.5. Additional notes on the stored program problem

The above algorithm and proof complete the central presentation of this section. There are two further points of interest that deserve note however.

There is another, slightly different, approximation algorithm for the stored program problem which also provides an absolute approximation guarantee of 1. The algorithm has the disadvantage that it delivers an optimal packing in fewer cases than the algorithm of 2.2, but it has a very simple proof. This second algortihm is due to Gregory Piatetsky-Shapiro in a variation of proof 2.4, and for interested readers is given in Appendix I.

It is also interesting to note a set of variations on SPP. We may create an unending set of

different SPPs by changing the number of bins. The number of bins in the SPP is not an input to the problem, but new NP-hard problems may be created by fixing that number at something other than 2. In general we can have a K-bin stored program problem, where $K >= 2$. The algorithm of 2.2 will deliver a packing within $K - 1$ of optimal for any K bins, and the proof of 2.4 applies in the obvious way to these problems.

It is my conjecture that this algorithm provides a lower bound on the approximation of the K-bin stored program problem. That is, unless $P = NP$, no polynomial time algorithm can deliver a solution guaranteed to be closer than $K - 1$ for a K bin stored program problem. I did some work on proving this conjecture and was unable to do so. Appendix II however contains some preliminary results from that effort which may be of help for other research.

3. VARIATIONS OF THE STORED PROGRAM PROBLEM

As mentioned in section 1 the set of NP-hard problems for which absolute approximation algorithms are known is very small.

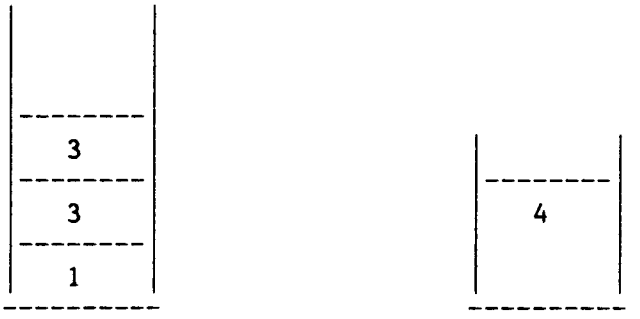
I find this set of problems interesting because it appears to be close to the border of the difference between problems in P and problems in $NP - P$ (I assume this set is not empty). Just what is it about a problem that prevents it from being solved quickly, when it is always possible to "almost solve" it quickly? What about these problems allows us to guarantee that we can get an almost perfect answer, but prevents us from knowing the exact answer without exhaustive search? I was also interested to see if I could enlarge the set of problems that have this property - NP-hard optimization problems with polynomial time approximation algorithms that guarantee an answer within an additive constant of the optimum.

While I have not shed any light on the *essential* difference between NP-hard problems with absolute approximation algorithms and those without, I have somewhat expanded the set of known NP-hard problems that have absolute approximation. I found a number of variations to the stored program problem which relax the original problem definition while maintaining the "quality" of the approximation algorithm. The three variations that I found make use of what the proof in 2.4 does *not* say.

3.1. SPP may use bins of different sizes

It is not necessay that the bins in the stored program problem be of the same size. The proof of 2.4 requires that $S_1 + S_2 < l_{k+1} + l_{k+2}$. This relation does not mention the total size of each bin, only the unused space in them after application of algorithm 2.2.

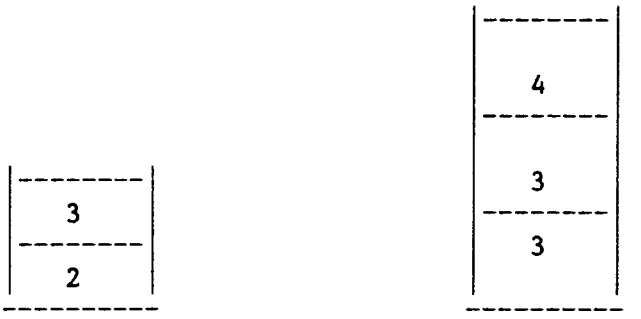
As an example, consider a list of objects with sizes 1, 3, 3, 4, 4 and bins of sizes 10 and 5. The algorithm of 2.2 will pack 4 objects, while an optimal packing would use all 5. The packing from 2.2 is shown below.

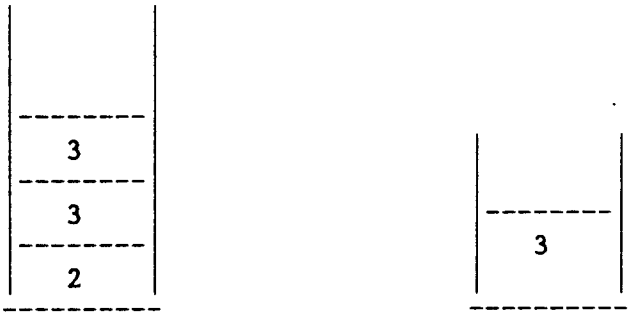


3.2. Differing sized bins may be re-ordered in SPP

Directly following from the observation above is the fact that the order of different sized bins does not affect the algorithm. Again $S_1 + S_2 < I_{k+1} + I_{k+2}$.

An example has object sizes 2, 3, 3, 3, 4 and bin sizes 5, 10. Algorithm 2.2 will yield an optimal packing. If we reverse the order of the bins, the algorithm will no longer deliver an optimal packing, but it will pack the first four objects and meet the algorithm's guarantee. 2.2 applied to the two different bin orders is shown below.





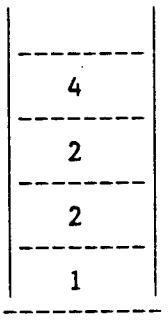
The irrelevancy of the order of the bins is striking when we consider a stored program problem with more than two bins. If we take the problem to have 4 bins, algorithm 2.2 promises to deliver a packing that is within 3 of the optimum. It may first appear that it will be possible to construct a set of objects and a set of 4 bins, such that in one ordering of bins the packing is optimal and in another ordering the result is bad (greater than 3 less than optimal). However a reading of proof 2.4 shows that this is not possible.

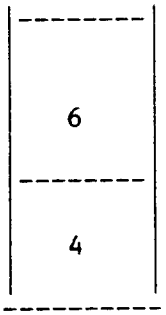
3.3. It is not necessary that each object fit in every bin

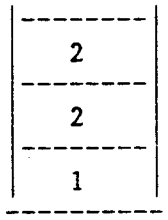
Algorithm 2.2 also holds if each object does not fit in every bin.

Proof 2.4 never counted the objects that were in a bin after the algorithm ran. It considered only how many more might fit. It is necessary only to state again that the proof establishes that $S_1 + S_2 < I_{k+1} + I_{k+2}$.

An example is the following: the objects are 1, 2, 2, 4, 6 and the bins are 10, 5. An optimal packing can use all the objects. Algorithm 2.2 puts the first four objects in the first bin and the fifth is not packed. The algorithm's guarantee holds however because the packing is within 1 of optimal. Two packings of this instance are shown below, first by the algorithm and then an optimal one.


4
2
2
1


6
4


2
2
1

4. AN ON-LINE VERSION OF THE STORED PROGRAM PROBLEM

The next variation of the stored program problem changes considerably. I will examine an on-line version which I believe to be an original problem. Assume there is a conveyer belt that is bringing the objects to the bins. The problem is to pack the greatest ***sequence*** of objects. That is, we can take the first 7 objects, or the first 10, but we cannot take the 2nd and the 5th and the 9th. Only the object at the end of the conveyer belt can be seen before it arrives. We also assume that there is a **staging area** which can be used to organize the objects that have arrived before packing them. The objects can be packed only once - they cannot be moved around after they are in the bins. This is the On-line Stored Program Problem (OLSPP).

There are several observations to make about this problem before considering algorithms for it. The first is that this is quite a natural problem. On-line versions of standard bin packing (where the goal is to minimize the number of bins) have been defined and discussed in the literature [Garey and Johnson]. It is also clear that this problem could correspond to some "real world" situations which may be just as described in the problem statement.

Second, I note that the problem is NP-hard. This is shown formally.

Proposition:

The On-line Stored Program Problem is NP-hard.

Proof:

- The On-line Stored Program Problem is to determine the longest initial sequence of objects that can be packed in the bins.
- It is an NP-complete problem to decide if an arbitrary set of objects can fit completely into an arbitrary set of bins. This is the "standard" bin packing problem [Garey and Johnson 79].
- Solving OLSPP includes discovering the answer to a series of NP-complete decision problems. When we know the longest initial sequence that can be packed, we also

know that all sequences shorter than it will fit, and that all sequences longer will not.

- Any standard bin packing decision problem can be polynomially reduced to OLSPP.
    * Make the set of objects in the bin packing problem some initial sequence in an instance of OLSPP. Make the bins the same in both problems.
- Solving OLSPP will determine if the initial sequence of objects corresponding to the standard bin packing problem will fit in the bins.
- An NP-complete problem is polynomially reducible to OLSPP. OLSPP is NP-hard.

The third observation to be made about OLSPP is that this problem is quite different from the Stored Program Problem considered initially. This may not be immediately obvious, but some reflection will show it to be true. In SPP we were able to pick and choose among all the objects to find the greatest set that could be packed. Obviously we chose the smallest objects. In OLSPP we must pack an entire sequence of objects, starting with the first, even if some of them are large.

Finally, OLSPP includes a staging area because it seemed natural to have such a feature, and unusually restrictive not to. One may certainly formulate a problem without one - where the objects must be taken off the conveyer belt and immediately placed in some bin. This problem should probably allow the additional freedom of moving objects around after they are in the bins. I do not consider this possible variation further however.

My approach will be as follows. First I investigate whether the algorithm from 2.2 can be used for OLSPP. I conclude that it cannot. Noting why it fails I then examine another algorithm that appears to hold promise of working well for this on-line problem. We see that it too does not work, and that it suggests some basic difficulty about the problem. I conclude that unless $P = NP$ no approximation algorithm is possible for OLSPP, and prove this result.

4.1. On-line algorithm #1

The most obvious candidate algorithm for the on-line stored program problem is an extension of the one presented in 2.2. We should consider applying it to each sequence of objects. This suggests the following algorithm:

- As an object arrives at the end of the conveyer belt sort its size with the sizes of the objects already in the staging area.
- See if the sorted list has two partitions such that the sum of the sizes in the first partition is less than or equal to the size of the first bin, and the sum of the second partition is less than or equal to the second bin. (We "test" algorithm 2.2 to see if it can pack the objects.)
    * If the above is true, take the object off the conveyer belt and add it to the staging area.
    * If the above is not true, do not take the object and proceed to pack the bins with the objects currently in the staging area - using algorithm 2.2. They will fit in the bins because the algorithm was tested with them.

This algorithm seems to hold promise because it tests algorithm 2.2 for every sequence of objects until it finds a sequence that it cannot pack, and then it packs all the objects prior to that. Consider the following case. The bins are sized 10, 10 with objects 3, 1, 3, 2, 4, 5, 2 (arriving in that order). Our algorithm will test 2.2 on each sequence, accepting an object when 2.2 delivers a packing. It will accept the first 6 objects, then reject the last one - because 2.2 cannot pack it with the others. When the last is rejected the algorithm uses 2.2 to pack the first six. In this case we are within 1 of optimal, since an optimal solution can pack all 7 objects.

The picture painted above is too rosy however. Consider the next case: bins 10, 10 and objects 5, 5, 3, 6, 1. The algorithm accepts the first 3, then fails to find a packing when the fourth is included, so it proceeds to pack only the first three. An optimal packing can pack all the objects however - our algorithm fails by 2 objects.

A particularly bad case has bins of size 10, 10 again and objects 5, 1, 6, 5, 1, 1, 1. In this

case the algorithm accepts only the first 3 (because the first 4 cannot be packed by the method in 2.2). This is quite a mistake however since all the objects can be placed in the two bins.

It is clear what the problem is. The current algorthim rejects the first object that it cannot pack by 2.2, even if there are large spaces left in the bins. If there is truly no way (by any ordering) to pack the object that is rejected, then the algorithm is correct in rejecting it - the problem states that an entire sequence must be packed. The trouble comes when we falsely reject an object (that could be packed in another way) and thereby reject small objects that come after it.

4.2. On-line algorithm #2

We can overcome the problem of large objects "hiding" small objects by packing the large objects first. That suggests the following:

- As each object arrives at the top of the conveyer belt, sort its size in *descending* order with the other objects in the staging area.
- Attempt to partition this list of objects for the bins by the following method.
    * Take the objects from largest to smallest and put them in a partition where they fit the worst (leave the most space).
- If the above partitioning succeeds, move the object to the staging area and go back to step 1.
- If the partitioning fails do not take the object, but instead pack the objects already in the staging area by the worst fit method already tested on them.

This algorithm works optimally for the following cases. Bins of size 10 and 10, objects of size:

- 2, 7, 3, 4, 1, 8 --- first 5 are packed
- 5, 5, 3, 6, 1 --- all are packed
- 5, 1, 6, 5, 1, 1, 1 --- all are packed

- 5, 1, 7, 6, 1, 2 --- first 3 are packed

While the above results are all optimal it is not surprising to discover that the algorithm is not optimal in all cases. As mentioned above, the problem is NP-hard.

The following case proves that the algorithm is not optimal: Bins of size 10 and 10, objects of sizes 6, 5, 4, 3, 2. Our algorithm will put the 6 in the first bin, the 5 in the second, then the 4 on top of the 5, and the 3 on top of the 6. It will then fail to pack the 2 and deliver a packing of only the first 4 objects.

We are not surprised that this algorithm does not provide an optimal solution. How badly does it fail though? Perhaps it provides some kind of approximation guarantee. It does not, and as we will see can perform arbitrarily badly.

4.3. There is no approximation algorithm for the on-line stored program problem

The reasonable algorithms presented above, and their failure, suggest that there might be something particularly difficult about this problem. There is.

Theorem:

If P <> NP there is no epsilon or absolute approximation algorithm for the On-line Stored Program Problem.

Terminology: The objects are labeled in the order they arrive. O_1 is the first object on the conveyer belt, O_2 the next, etc.

Proof:

- Assume *P <> NP*. Under this assumption, no polynomial time algorithm can provide an optimal answer in all cases to the on-line stored program problem.
- Since all instances are not solved optimally by any polynomial time algorithm, for any algorithm there is some set of instances where the algorithm packs objects $O_1 \ldots O_i$ when it is possible also to pack O_{i+1}.

- Let A be any polynomial time approximation algorithm. In the set of instances on which A fails (does not pack optimally), there is a non-empty subset where an optimal packing will leave some space in a bin.

    * The above is true because there cannot be a polynomial time algorithm for OLSPP which correctly packs all cases that leave some space in a bin.

    * Suppose there were. We can use this algorithm to also pack instances where an optimal packing completely fills the bins.

    * Take an instance where the optimal packing completely fills the bins and add a very small amount to one bin. The instance is now one where an optimal packing will leave some space. If we are restricting ourselves to integer values, multiply the objects and bins by 10 and add 1 to one bin.

    * We are assuming we have an algorithm which can pack this new problem optimally. We use it, then remove the small amount that was added to one bin. There will be this small amount unpacked because we manufactured this case (above) to make this true.

    * We now have an optimal solution to the original instance. We have used an algorithm which can always pack (optimally) cases that leave some space to optimally pack any case that does not leave some space.

    * But this is not possible because we would then have solved all cases of an NP-hard problem in polynomial time, rejecting our assumption that $P <> NP$.

    * So, unless $P = NP$, there can be no polynomial time algorithm which optimally packs all OLSPP instances that leave some space in the optimal solution.

- So, for any polynomial time algorithm there is some instance I which is not packed optimally by it, and an optimal packing would leave unused space. Such an instance for each algorithm is key to the proof.

- For any algorithm, it failed to deliver an optimal packing on instance I because it looked at object O_{i+1} at the end of the conveyer belt and concluded that it could not be packed (when in fact it could have). By the definition of the problem, the algorithm could not "see" objects that may have been beyond O_{i+1}.

- We take such instance I, which we have for any algorithm, and modify it so that there are many small objects beyond O_{i+1} that fit in the unused space. The number of such small objects is unbounded.

    * If we restrict ourselves to integers for all problem values, we can still add any number of small objects beyond O_{i+1} by multiplying the objects and bins of the problem by 10 (or 100, or 1000) and adding objects of size 1 beyond O_{i+1}.

- The number of objects that could have been packed, but were not, by any polynomial time algorithm is unbounded. There is no ϵ and n such that

$|approx\_solut - opt\_solut| \; / \; opt\_solut <= \epsilon < 1$ for all problems with *opt_solut* larger than *n*.

- There is therefore no polynomial time algorithm that delivers an absolute or an epsilon approximation for the on-line stored program problem.

Appendix I

I. A SECOND ALGORITHM FOR THE STORED PROGRAM PROBLEM

I mentioned in 2 that there was another algorithm for the stored program problem. I include it here for two reasons: the proof that it works is beautifully simple, and it may aid another researcher with the lower bound proof discussed in Appendix II. This algorithm is due to Gregory Piatetsky-Shapiro in a variation of my variation of Horowitz's proof (2.4). The analysis following the proof is by myself.

I.1. THE SECOND ALGORITHM

In this algorithm we go, in a sense, immediately to its proof. We do the following:

- Construct a single large bin that is the height of all the original bins combined.
- Put as many objects as possible in the constructed bin filling it in ascending order.
- Remove any object that crosses the original bin boundries.

An example to illustrate uses bins with sizes 10 and 5, and objects of sizes 1, 3, 3, 4, 4. Construct a single bin of size 15, fill it with all the objects, then remove the first '4' because it cross an original boundry. Our packing is 1, 3, 3 in the 10 bin, and 4 in the 5 bin.

The algorithm obviously packs within $K - 1$ objects of optimal (where K is the number of bins). No algorithm can pack more objects into the original bins than will fit in the larger constructed bin. There are $K - 1$ boundries in the larger bin. At most $K - 1$ objects are removed.

I.2. ANALYSIS OF THE ALGORITHM

In application this algorithm does not work quite as well as the one of section 2.2 because it provides a packing that is less than optimal in cases where the one from 2.2 is optimal. There are no cases where the reverse is true.

We first consider the second claim. Piatetsky's algorithm gives an optimal packing when an ascending order sum of the objects has partitions that exactly match the bin sizes in the problem. In this case the algorithm of 2.2 will also give an optimal packing.

However it is easy to construct an example where algorithm 2.2 is optimal when this one is not. An example is bins of size 10 and 10 with objects 1, 3, 3, 4, 4. Piatetsky's algorithm would pack 1, 3, 3 in the first bin, and a 4 in the second bin. Algorithm 2.2 will pack all the objects.

In theoretical considerations however the observation above is of little consequence and Piatetsky's algorithm has great appeal due to its simplicity.

Appendix II

II. NOTES ON A LOWER BOUND PROOF

In section 2 I remarked that different versions of the stored program problem could be created with any fixed number of bins. I stated that the approximation algorithm given there, and its proof, could be extended to these other problems with the guarantee that the value derived would be within $K - 1$ of optimal (where K is the number of bins). I conjectured that this result provides a lower bound on approximation for these problems. That is, unless $P = NP$ no polynomial algorithm can guarantee a solution closer than $K - 1$ to optimal.

Below are some observations from my work on a proof of this conjecture. They may prove helpful for further research.

1) For any SPP we can simplify it by removing from the list the largest objects that make the list larger than the size of the bins. No algorithm can pack a list that has a larger size than the total of the bins, so there is no point in considering such a list. We assume, as we did in section 2, that the list of objects is sorted. Choose the largest k such that $\Sigma\ l_1 ... l_k$ <= sum of the bins.[4] All objects with a subscript greater than k may be discarded.

We do not hurt an optimal packing by removing these extra, largest, objects. If there is an optimal packing that uses a larger object, say O_{k+1}, (and leaves out O_i which is smaller), then there is also an optimal packing that substitutes O_i for O_{k+1}.

2) When an approximation algorithm fails by n objects, we can assume it failed because it could not pack the n largest objects in the optimal list. That is, if it is possible to pack k objects, but an algorithm packs only $k-2$, then we can assume that it did not pack O_{k-1} and O_k.

---

[4] l_i is the size of object O_i.

If the algorithm had, in fact, packed O_{k-1} and O_k we can swap those objects for some smaller ones that were left out. Thus, we can "normalize" all algorithms for SPP by having them fail on the largest objects in the optimal list. This convention may make it easier to compare competing algorithms and analyze why it is hard to improve on the $K-1$ bound.

References

Garey, M. and Johnson, D. "Approximation Algorithms for Bin Packing Problems: A Survey".

Garey, M. and Johnson, D. 1976a "The Complexity of Near-Optimal Graph Coloring". JACM, vol 23, no 1, pp 43-49.

Garey, M. and Johnson, D. 1976b "Approximation Algorithms for Combinatorial Problems: An Annotated Bibliography" in *Algorithms and Complexity: Recent Results and New Directions.* J. Traub (ed.). Academic Press. pp 41-52.

Garey, M. and Johnson, D. 1979 *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman and Company.

Holyer, I. 1980 "The Computational Complexity of Graph Theory Problems". Ph.D. thesis. University of Cambridge.

Holyer, I. 1981 "The NP-completeness of Edge Coloring". SIAM Journal of Computation, vol 10, pp 718-720.

Horowitz, E. and Sahni, S. 1978 *Fundamentals of Computer Algorithms.* Computer Science Press.

Johnson, D. 1984 Personal conversation about the edge coloring problem. His remarks about the absolute approximation possible for minimum edge coloring were based on proofs by Holyer [1980] and [1981] and Leven and Galil [1983].

Leven, D. and Galil, Z. 1983 "NP-completeness of Finding the Chromatic Index of Regular Graphs". Journal of Algorithms, vol 4, pp 35-44.

Piatetsky-Shapiro, G. 1983 Personal conversation about the algorithm and proof in Appendix II.

Sahni, S. 1977 "General Techniques for Combinatorial Approximation". Operations Research, vol 25, no 6, pp 920-936.

Sahni, S. and Gonzalez, T. 1976 "P-Complete Approximation Problems". JACM, vol 23, no 3, pp 555-565.

Sahni, S. and Horowitz, E. 1978 "Combinatorial Problems: Reducibility and Approximation". Operations Research, vol 26, no 4, pp 718-759.