From Approximate to Optimal Solutions: Constructing Pruning and Propagation Rules

Ian P. Gent and **Toby Walsh**[*]
APES Group, Department of Computer Science
University of Strathclyde, Glasgow G1 1XH, Scotland
{ipg,tw}@cs.strath.ac.uk

Abstract

At the heart of many optimization procedures are powerful pruning and propagation rules. This paper presents a case study in the construction of such rules. We develop a new algorithm, Complete Decreasing Best Fit, that finds the optimal packing of objects into bins. The algorithm use a branching rule based on the well known Decreasing Best Fit approximation algorithm. In addition, it includes a powerful pruning rule derived from a bound on the solution to the remaining subproblem. The bound is constructed by using modular arithmetic to decompose the numerical constraints. We show that the pruning rule adds essentially a constant factor overhead to runtime, whilst reducing search significantly. On the hardest problems, runtime can be reduced by an order of magnitude. Finally we demonstrate how propagation rules can be built by adding lookahead to pruning rules. This general approach – optimization procedures built from branching rules based on good approximation algorithms, and pruning and propagation rules derived from bounds on the remaining subproblem – may be effective on other NP-complete problems.

1 Introduction

When presented with a new combinatorial problem, how do we construct an effective optimization procedure? Korf has demonstrated how to convert approximation algorithms for number partitioning into branching rules within optimization procedures [Korf, 1995]. He concluded that this "presents an example of an approach that may be effective on other combinatorial problems. Namely, we took a good polynomial-time approximation algorithm, and made it complete, so that the first solution found is the approximation, and then better solutions are found as long as the algorithm continues to run, eventually finding the optimal solution" [Korf, 1995]. We test this claim for a closely related problem, bin packing using the Decreasing Best Fit approximation algorithm.

[*]Supported by EPSRC award GR/K/65706. We thank the members of the APES group, especially Paul Shaw.

A shortfall of Korf's proposal is that optimization procedures also benefit from pruning rules to terminate unproductive lines of search, and propagation rules to determine when branching decisions are forced. We identify a very general pruning rule derived from a bound on the solution to the remaining subproblem. The bound is constructed by using modular arithmetic to decompose the numerical constraints. Since pruning rules need to be cheap to test, we describe an implementation that adds essentially a constant factor overhead to the runtime whilst significantly reducing the number of nodes explored. This pruning rule is more effective on harder and more constrained problems. We then show how propagation rules can be derived from pruning rules. Finally, we derive similar pruning rules for other domains including number partitioning and knapsack problems.

Our pruning bound is based on reasoning about the numerical constraints of a problem using modular arithmetic. For example, can we pack objects with weights $8, 6, 4, 2, 2$ into two equally sized bins leaving no empty space? The parity bit alone tells us this is impossible. As the sum of the weights is 22 and the two bins are the same size, each bin has 11 units of capacity. It is clearly impossible to pack objects with even weights into bins with an odd capacity and leave no empty space. In this paper, we generalize such reasoning to other bit positions and to non-binary bases.

This research suggests a general strategy for building optimization procedures that may be useful in other NP-complete problems. That is, we construct branching rules from good polynomial-time approximation algorithms, and pruning and propagation rules from bounds on the solution to the remaining subproblem. For combinatorial problems involving numbers, modular arithmetic may be able to decompose the constraints and suggest bounds useful for pruning and propagation.

2 Bin packing

To explore Korf's claim, we chose bin packing. There exist several good polynomial-time approximation algorithms for bin packing but few effective optimization procedures. Bin packing is of practical and theoretical importance. Many businesses like post offices have real bin packing problems to solve that are of economical value [Slatford and Yeadon, 1970]. Problems such as scheduling, processor allocation, fabric cutting and minimization of VLSI circuit size and delay, can be modeled by bin packing problems. Bin packing is NP-complete in the strong sense [Garey and Johnson, 1979].

We consider a general bin packing problem in which the bins can have different capacities. This allows us to reason about the subproblems encountered during search when bins may be partially full. In addition, as we show in Sec. 9, many other problems like number partitioning can be seen as instances of such general bin packing problems. A bin packing problem consists of a set of objects S and a set of bins B. Each object $s \in S$ has a weight w_s, and each bin $b \in B$ has a capacity c_b. The difference between the sum of the bin capacities and the sum of the weights is the spare capacity, *sc*. The aim is to partition the objects between the bins so that the sum of weights in each bin is less than or equal to its capacity. All weights and capacities are integers. If all the bins have the same capacity, this is the NP-complete problem SR1 from [Garey and Johnson, 1979].

Decreasing Best Fit (DBF) is one of the best polynomial-time approximation algorithms for bin packing. Best Fit puts the next object into the fullest bin that will accommodate it without exceeding the capacity. DBF simply sorts the objects into decreasing order of their weight before calling Best Fit. Objects with the largest weights are thereby packed first. DBF is guaranteed asymptotically to use no more than 11/9 times the optimal number of bins, compared to Best Fit which can use 17/10 times the optimal [Garey and Johnson, 1979].

3 A branching rule

To develop an optimization procedure for bin packing, we follow Korf's methodology, and convert the Decreasing Best Fit approximation algorithm into a branching rule within the Complete Decreasing Best Fit (CDBF) optimization procedure. CDBF computes a lower bound on the number of bins required. If all the bins have the same capacity, c then we need at least $\lceil \sigma/c \rceil$ bins where σ is the sum of the weights. Search begins with the number of available bins equal to this lower bound. If the search tree is exhausted before a packing is found, the number of available bins is incremented. This usually relaxes the constraints sufficiently to make packing easy.

The objects are sorted into decreasing order of their weight and packed into bins by the branching rule. The first choice of the branching rule is that made by the Decreasing Best Fit approximation algorithm. That is, it packs the next object into the fullest bin that accommodates it. To make the optimization procedure complete, we must decide what the branching rules does on backtracking. A natural generalization is to order bins by how full they are. The branching rule thus tries to pack the next object into the fullest bin that accommodates it, and on backtracking, tries bins in increasing order of their unused capacity. A branch terminates successfully if all the objects are packed into bins. A branch terminates unsuccessfully and forces backtracking when the next object cannot be packed into any bin. That is, when the weight of the next object exceeds the largest unused capacity. To avoid duplicating search, the branching rule does not pack an object into more than one bin with the same unused capacity. We therefore never open more than one new bin for any object. As a consequence, the first object is always packed into the first bin, the second object into the first or the second bin, *etc.* In the worst case, when packing n objects into b bins, CDBF explores all $b^n/b!$ different packings.

4 A pruning rule

A bin packing problem imposes constraints on how the weights in the bins add up. We decompose these constraints into constraints on how the bit positions add up. To do this, we take the modulus of the weights and capacities to a set of bases. Often the bases will be all the powers of 2 since, as we show in Sec. 5, we can then use simple logical operations on the binary representation. However, other bases can be used at little extra cost.

The constraint on each bin is that the sum of the weights is no more than the capacity of the bin. To reason about how the spare capacity is distributed across the bins, we construct a set 1^{sc} containing sc "dummy" objects, each with a weight of 1. We now have to find an exact packing of $S \cup 1^{sc}$ so that the sum of the weights in each bin is strictly equal to the capacity of the bin. That is, for each $b \in B$,

$$\sum_{s \in b} w_s = c_b$$

To focus on different bit positions, we take the modulus of both sides of this equation to the base m,

$$(\sum_{s \in b} w_s) \bmod m = c_b \bmod m$$

And sum over the bins,

$$\sum_{b \in B} \left((\sum_{s \in b} w_s) \bmod m \right) = \sum_{b \in B} (c_b \bmod m) \quad (1)$$

Unfortunately, we cannot use this equation directly as we do not know which objects will be put in each bin. We can, however, give an upper bound for the left hand side of (1) assuming the worst possible partition of weights for the mod sum. To derive the bound, we use the fact that for any integers a and b,

$$(a \bmod m) + (b \bmod m) \geq (a + b) \bmod m$$

Applying this repeatedly to the left hand side of (1) gives

$$\sum_{b \in B} \left(\sum_{s \in b} (w_s \bmod m) \right) \geq \sum_{b \in B} (c_b \bmod m)$$

But the nested summation sums over all the weights in all the bins. Hence,

$$\sum_{s \in S \cup 1^{sc}} (w_s \bmod m) \geq \sum_{b \in B} (c_b \bmod m)$$

That is,

$$\sum_{s \in S \cup 1^{sc}} (w_s \bmod m) - \sum_{b \in B} (c_b \bmod m) \quad \geq \quad 0$$

Now, if $s \in 1^{sc}$ then $w_s = 1$ and $(1 \bmod m) = 1$ for all m. Thus,

$$sc + \sum_{s \in S} (w_s \bmod m) - \sum_{b \in B} (c_b \bmod m) \quad \geq \quad 0 \qquad (2)$$

This is our general pruning bound. The first two terms of the LHS of (2) represents the maximum contributions mod m that we can expect from the spare capacity and S. The third term represents the capacities mod m that we need these contributions to reach. If at any time, the LHS is less than zero, the packing of the bins is impossible. We can then prune search.

As an example, consider again the bin packing problem from the introduction. This problem has no spare capacity. Mod 2, we have objects with zero weight (i.e. the weights provide no parity bits), and two bins each with a capacity of 1 (i.e. each bin needs a parity bit). The LHS of (2) is 0-2. Since this is less than zero, we cannot pack the objects into the two bins.

As a second example, can objects with weights 650, 540, 390, 260 and 130 be packed into two bins of capacity 1000? Consider this problem mod 128. The objects contribute weights mod 128 of 10, 28, 6, 4 and 2 respectively. In addition, we have 30 units of spare capacity. However, each bin needs 1000 mod 128 = 104. The objects cannot therefore be packed into the bins. The pruning bound reflects this; the LHS of (2) is $30 + (10 + 28 + 6 + 4 + 2) - (104 + 104)$. As this is less than zero, we cannot pack the objects into the bins.

5 Implementation

To implement checking the bound efficiently, we make three critical observations. First, if the moduli used are all powers of a given base (e.g. m, m^2, m^3 ...), many computations from one power of the base can be reused in the next higher power. For example, with powers of 10, $(a \bmod 1000) > (b \bmod 1000)$ if either the 100's digit of a is larger than that of b, or if the 100's digits of the numbers are the same and $(a \bmod 100) > (b \bmod 100)$. Similar reuse of calculated values can be made for other operations such as longhand addition and subtraction.

Second, the LHS of (2) changes only slightly when an object is assigned to a bin. In fact, detailed analysis shows that if we put an object s into a bin b, the value of the LHS of (2) does not change at all if $(w_s \bmod m) \leq (c_b \bmod m)$, and is otherwise simply reduced by m. It is more efficient therefore to initialise the values of the LHS of (2) at the start of search, and merely compute the change to these values at each node. To restore the values on backtracking, we use one bit for each modulus m indicating if the LHS of (2) was decremented.

These two observations combine together. We initially find the base m expansion of the weights and capacities. Then at each node, we perform the subtraction $c_b - w_s$ longhand in base m. In doing this, the values of the LHS of (2) in each power of m are decremented when there is a borrow in that digit position, since a borrow is necessary when $(w_s \bmod m) > (c_b \bmod m)$. The complexity of the operations performed at each node is thus the same as that of the subtraction $c_b - w_s$, which has to be performed in any case. The only other operation is that of decrementing the LHS of (2) and incrementing it on backtracking. The LHS of (2) is always a multiple of m so we store the value divided by m and decrement it and increment it by 1 instead of m. Since this value is a number bounded above by n, these decrements and increments take less than $O(\log n)$ time. In practice n is small and it may take just a single machine operation. We can thus implement checking the bound in all powers of a given base with a constant factor overhead compared to CDBF without modular pruning. Our experiments support this observation.

Our third observation does not affect the theoretical complexity but does make a practical implementation more efficient. A natural set of bases to use is all powers of 2. This is the set of bases used in the experiments reported here. This choice allows us to take advantage of the binary representation of numbers in computers. For example, finding the value of an integer modulus a power of 2 is very cheap. In addition, we do not need to perform the subtraction longhand since the bits of $(a \textbf{ xor } b \textbf{ xor } a - b)$ indicate whether a borrow was necessary into a given bit position when computing $a - b$.

6 Experiments

As in previous experimental studies [McGeoch, 1986], we pack objects with pseudo-random integer weights into bins that are twice their maximum size. We generate n objects each with a weight drawn uniformly and randomly from $(0, l]$ and pack into bins of capacity $2l$. Exploratory tests with this model show large variation in problem difficulty. For example, the worst case for a sample of 1,000 bin packing problems with 20 objects of size 2^{11} took 3,522,573 nodes whereas 90% of problems needed just 1 branch. The data suggests that problem difficulty is very dependent on the spare capacity.

We therefore modify the model to generate problems with a predetermined spare capacity. We use two additional parameters: a lower bound on the number of bins, d, and a spare capacity, sc. The new model constructs bin packing problems which, if they pack into d bins of capacity $2l$, leave a spare capacity of sc. We generate $n-1$ objects with weights randomly and uniformly distributed on $(0, l]$. Let w be $2ld - sc - \sigma$. This is the weight the nth and final object would need to have to give the required spare capacity. If $w \in (0, l]$ then we assign the nth object this weight. If not, we throw away the $n-1$ objects and start again. We experimented with a variety of spare capacities, but found that performance was broadly similar with weights in the range $(0, l]$ and spare capacity sc as with weights in the range $(0, l/sc]$ and spare capacity 0. In the rest of the paper we restrict attention to problems with no spare capacity.

To determine the overhead modular pruning adds to CDBF in practice, Fig. 1 gives a scatter plot of nodes searched to find the optimal packing against CPU time with and without modular pruning. We use 100 problems at each value of n from 8 to 20 and random 10-bit weights which leave no spare capacity when packed into $n/4$ bins. In this and subsequent experiments, we prune with bases that are powers of 2. CDBF was coded into Common Lisp and run on a network of identical DEC Alpha 300LX's with 125MHz processors. This graph supports our claim that pruning adds a constant factor overhead to runtime. Regression for cpu seconds per node suggests that the overhead is about a factor of 5.4 for $n = 8$ but declines to 2.4 at $n = 20$. The nearly linear nature of this graph also supports the practice of using nodes searched as a proxy for cpu time.

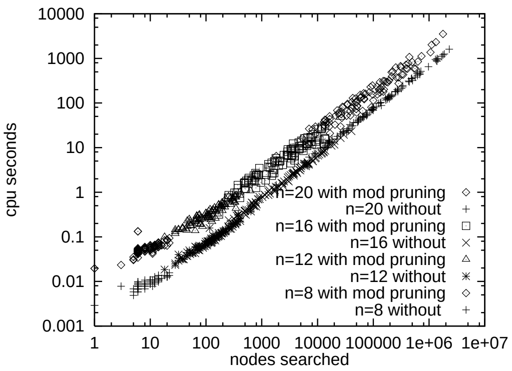

Figure 1: Nodes searched against CPU time

The pruning rule can reduce search significantly. Fig. 2 shows the mean nodes searched to find the optimal packing against $\log_2(l)$. We again generate bin packing problems with 8 to 20 objects which have no spare capacity when packed into $n/4$ bins. We use 1,000 problems at each value of n and l. As in other NP-complete problem classes [Cheeseman *et al.*, 1991], there is a phase transition as the constrainedness varies. At small $\log_2(l)$, almost all problems pack into $n/4$ bins easily. At large $\log_2(l)$, almost all problems require an extra bin. The hardest bin packing problems tend to occur in the phase transition inbetween. We see similar behavior to Fig. 2 in the median and other percentiles of performance.

As was expected, the effectiveness of the pruning rule increases as $\log_2(l)$ increases and we have more bits with which to prune. For example at $n = 20$ and $\log_2(l) = 45$ mean cpu time was reduced from 274 to 246 seconds with mod pruning, despite the overhead. The effectiveness of the pruning rule also increases as the number of objects being packed increases. Fig. 2 supports our claim that modular pruning reduces search significantly. On the harder problems, runtimes can reduce by an order of magnitude even in the phase transition region. For example, at $n = 20$ and $\log_2(l) = 10$, one problem took 1,258 cpu seconds without modular pruning but only 80 seconds with it. The pruning rule cannot increase the number of nodes searched, so where cpu time is greater it is by no more than the implementation overhead.

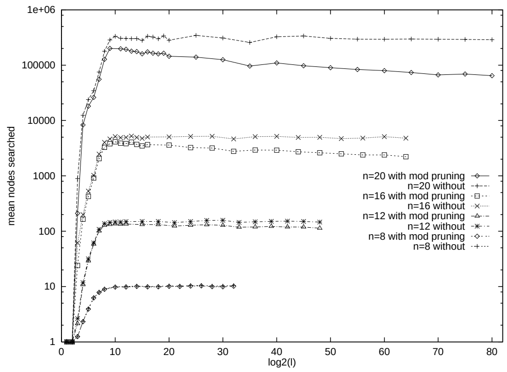

Figure 2: Mean number of nodes searched against the size of the weights

In conclusion, the pruning rule adds essentially a constant factor overhead to but reduces search significantly. On the hardest problems, runtime can be reduced by an order of magnitude. For problems larger than considered here, the search tree may become too large to explore exhaustively. On such problems, we might consider a search strategy like ILDS [Korf, 1996]. This systematically explores decisions made against the branching heuristic. Unlike depth-first search, ILDS can undo branching mistakes high in the search tree at little cost, as well as taking advantage of pruning and propagation rules.

7 Multiple mod pruning

If we multiply the weights and capacities by some constant then we get an equivalent bin packing problem. However, the modulus of the numbers may now offer new pruning opportunities. To test this hypothesis, Fig. 3 plots median and 99th percentile in nodes searched with pruning bounds derived by multiplying the weights and capacities by 1, 3, 5, and 7. We use 100 problems $n = 20$ for varying l. Fig. 3 shows that multiple modular pruning offers further reductions in search over the regular bound. By adding pruning possibilities we can only reduce the number of nodes searched. These reductions in search can result in shorter runtimes on harder problems than single modular pruning or no modular pruning.

8 A propagation rule

Propagation rules determine when branching decisions are forced. For example, the Davis Putnam satisfiability procedure [Davis and Putnam, 1960] has the unit propagation rule. This assigns truth values to variables in unit clauses as these values are forced. This rule contributes significantly to the effectiveness of the Davis Putnam procedure. Propagation rules can often be constructed by adding look-ahead to a suitable pruning rule.

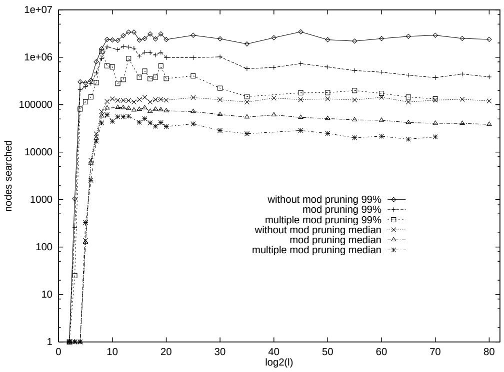

Figure 3: Modular pruning with the multiples, $\{1, 3, 5, 7\}$

For instance, the Davis Putnam procedure also has an empty clause pruning rule. This prunes search when an empty clause is generated. The unit propagation rule is merely a one-step look-ahead on top of the empty clause pruning rule. Suppose we have a unit clause, l. Looking ahead, if we instantiate l to $False$, then the empty clause pruning rule would fire. We therefore assign l to $True$. But this is precisely the unit clause propagation rule: if we have a unit clause, l then we assign l to $True$.

We can construct a propagation rule based on the pruning rule for bin packing in an identical fashion. In each base, the propagation rule packs the object with largest weight in that modulus into one of the bins, and tests the pruning bound. If the bound rejects every bin, we prune search. If the bound rejects all but one bin, we pack the object into the relevant bin. This adds an inter-node cost of $O((n+d)\log(l)^2)$ if we use all powers of 2 as bases, as we do here. Fig. 4 shows the median and 99th percentile in nodes searched with and without this propagation rule. We again use 100 problems at $n = 20$. Despite the larger overheads, the reductions in search can result in shorter runtimes on the harder problems compared to CDBF either with or without modular pruning.

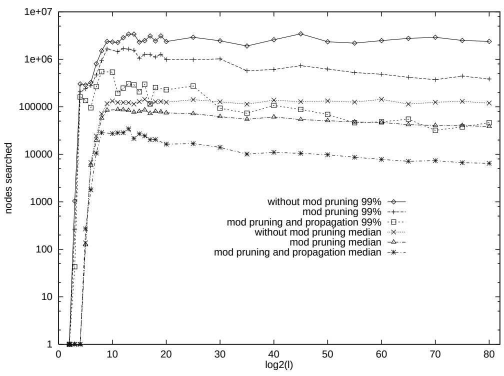

Figure 4: Modular pruning and propagation

9 Other applications

We now describe some other application domains.

9.1 Subset sum

Given a set S, is there a subset whose weights have a given target sum t? Subset sum is problem SP13 in [Garey and Johnson, 1979]. It is equivalent to bin packing into two bins with capacities t and $\sigma - t$ where σ is the sum of the weights. The pruning bound becomes,

$$\sum_{s \in S}(w_s \bmod m) - t \bmod m - (\sigma - t) \bmod m \quad \geq \quad 0$$

For example, is there a subset of 17, 12, 9 and 4 with sum 23? Consider the numbers mod 4. We have 1, 0, 1 and 0 with which to meet the target sums of 23 mod 4 and 19 mod 4 (that is, 3 and 3). It is therefore impossible to find a subset with the required sum. This is confirmed by the bound, with the LHS being less than zero.

9.2 Number partitioning

Given a set S is there a partition of S into k sets whose weights have an equal sum? For $k = 2$ this is SP12 in [Garey and Johnson, 1979]. Number partitioning is equivalent to bin packing into k equal bins with capacities σ/k, where σ is the sum of the weights, assumed to be a multiple of k. To construct a pruning bound that applies in the middle of search, we assume that we have constructed a partial partition of S, that the partial partitions have weights with sums s_i, and that $s_i \leq \sigma/k$. Let S' be the numbers still to be partitioned. The remaining problem is equivalent to bin packing into k bins with capacities $\sigma/k - s_i$. This gives the pruning boumd,

$$\sum_{s \in S'}(w_s \bmod m) - \sum_{i=1}^{k}(\frac{\sigma}{k} - s_i) \bmod m \quad \geq \quad 0$$

Note that the first term is less than $|S|.m$ and the second is less than $k.m$. For a fixed number of objects, we are therefore more likely to break the bound for large k. This agrees with our intuitions. The more bins we have, the more likely we will be able to deduce using modular arithmetic that we cannot fill one of the bins. A similar bound can be derived for inexact number partitioning problems in which the sum of the weights is not necessarily a multiple of k, and the sums of the partitions is within some $\Delta > 0$. To construct a bound here, we add to S an extra set of "dummy" objects with unit weight.

9.3 Knapsack problems

In the 0/1 knapsack problem, we have a set of objects S, each object $s \in S$ has a weight w_s and a value v_s. We also have a target value v and a knapsack which can contain a weight w. Is it possible to put objects in the knapsack to reach the target value without exceeding the weight? This is problem MP9 in [Garey and Johnson, 1979]. For simplicity, consider the special case where the weights of the objects equal their values. Let σ be the sum of the weights. Then we wish to find a subset of

S with weight at least v but no more than w. This is equivalent to packing into two bins, with capacities w and $\sigma - w$, and with a spare capacity of $w - v$.

10 Related Work

Korf converted the Greedy and the Karmarkar-Karp approximation algorithms for number partitioning into branching rules for optimization procedures [Korf, 1995]. The pruning rules Korf used are the analogues of the simple pruning rule in CDBF without modular pruning.

McGeoch has performed extensive experiments on the First Fit, Best Fit, Decreasing First Fit and Decreasing Best Fit approximation algorithms [McGeoch, 1986], using objects with integer weights and bins of capacity $2^{30}-1$. She observed a 'critical region' in which Decreasing First Fit gave packings with a large amount of empty space. As these packings tended to occur when there was a statistical excess of objects with large weights, this region may become less important as n increases and the distribution of weights tends to become more uniform.

Bin packing has a polynomial-time asymptotic approximation scheme [Papadimitriou, 1994]. This uses modular arithmetic to reduce the grain size of the weights. To approximate within ϵ, each weight w is replaced by $\lceil w/q \rceil$ where q is the quantum size of weights, $\lfloor \epsilon c \rfloor$ and c is the bin capacity. Knapsack problems have been proposed as the basis of a public-key cryptosystem [Merkle and Hellman, 1978]. The receiver deciphers the encoded message by multiplying the weights of the knapsack problem by a secret key and taking the modulus using a second secret key. This gives a knapsack instance that can be rapidly solved. Brickell (personal communication cited in [Lagarias and Odlyzko, 1985]) suggested that difficult and 'dense' cryptographic knapsack problems with many objects and small weights might be solved by converting them to 'low-density' problems through one or more modular multiplications. [Bright *et al.*, 1994] describes a parallel method for solving knapsack problems. Modular arithmetic may still be of use in such methods for eliminating parts of the search space.

11 Conclusions

The main contributions of this paper are new pruning and propagation rules for bin packing and some closely related problems. The pruning rule uses a bound constructed by decomposing the numerical constraints with modular arithmetic. We incorporated this rule into a new optimization procedure, Complete Decreasing Best Fit. We showed that the pruning rule adds essentially a constant factor overhead to runtime whilst reducing search significantly. On the hardest problems, runtime can be reduced by an order of magnitude. Finally, we demonstrated how propagation rules can be built by adding lookahead to pruning rules. There are many directions for future research. For example, how do we identify good sets of bases and multiples for pruning?

What contributions might this research make beyond bin packing? First, the pruning bound can be applied to many other partition problems like subset sum, number partitioning, and knapsack problems. Second, modular arithmetic may be useful in constructing pruning bounds and propagation rules in other combinatorial problems involving numerical constraints. Third, new propagation rules may be developed by adding lookahead to existing pruning rules. Fourth, this paper presents an example of a general methodology for building optimization procedures in new domains. That is, we construct a branching rule based on a good approximation algorithm, and pruning and propagation rules derived from a bound on the solution to the remaining subproblem. This approach may be effective on other NP-complete problems.

References

[Bright *et al.*, 1994] J. Bright, S. Kasif, and L. Stiller Exploiting algebraic structure in parallel state space search In *Proceedings of AAAI-94*, pages 1341–1346, 1994.

[Cheeseman *et al.*, 1991] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *Proceedings of 12th IJCAI*, pages 331–337, 1991.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.

[Garey and Johnson, 1979] M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W H Freeman, 1979.

[Korf, 1995] R.E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of 14th IJCAI*, pages 266–272, 1995.

[Korf, 1996] R. Korf. Improved limited discrepancy search. In *Proceedings of AAAI-96*, pages 288–291, 1996.

[Lagarias and Odlyzko, 1985] J.C. Lagarias and A.M. Odlyzko. Solving low-density subset sum problems. *Journal of the Association for Computing Machinery*, 12(1):229–246, 1985.

[McGeoch, 1986] C. C. McGeoch. *Experimental Analysis of Algorithms.* PhD thesis, Carnegie Mellon University, 1986. Also available as CMU-CS-87-124.

[Merkle and Hellman, 1978] R.C. Merkle and M.E. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Transactions on Information Theory*, 24:525–530, 1978.

[Papadimitriou, 1994] C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Slatford and Yeadon, 1970] J.M. Slatford and N.B. Yeadon. Report on the packing efficiency of parcel containers. Technical Report 12, Parcel and Bulk Mail Mechanisation Branch, Design and Development Division, Post Office, London EC1, 1970.