[3] Karmarkar, N., and R.M. Karp, The differencing method of set partitioning, Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, Ca., 1982.

[4] Korf, R.E., Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[5] Korf, R.E., Linear-space best-first search, *Artificial Intelligence*, Vol. 62, No. 1, July 1993, pp. 41-78.

[6] Korf, R.E., From approximate to optimal solutions: A case study of number partitioning, to appear, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995.

7 Conclusions

Limited discrepancy search is an effective algorithm for problems where only part of a tree has to be searched to find a solution, or the problem size prohibits an exhaustive search. We presented an improved version of the algorithm that reduces its time complexity from $O(\frac{d+2}{2}2^d)$ to $O(2^d)$ for searching a complete binary tree of uniform depth d. In practice, however, the improvement is much less due to pruning of the tree, and we measured a factor of six improvement on trees of depth 35. While our improved algorithm always generates fewer nodes than the original, when only a very small fraction of the tree is searched, it is not significantly better than the original. We also showed that the overhead of the improved algorithm compared to depth-first search is a factor of $b/(b-1)$, for a complete search of a b-ary tree. In practice, again due to pruning, this overhead is significantly greater, and we measured a value of 3.5 on our binary trees. Finally, we compared the performance of these algorithms on the NP-Complete problem of number partitioning, as a function of problem difficulty. While depth-first search is the best algorithm for the hardest problem instances, limited discrepancy search is significantly more efficient on easier problems where an exact solution exists.

8 Acknowledgements

Thanks to Matt Ginsberg and Will Harvey for many helpful discussions concerning this research. This work was supported by NSF Grant IRI-9119825, and a grant from Rockwell International.

References

[1] Garey, M.R., and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[2] Harvey, W.D., and M.L. Ginsberg, Limited discrepancy search, to appear, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Canada, Aug. 1995.

The performance of the original OLDS algorithm is even worse in this region, due to its redundant generation of all paths in previous iterations. The node generation ratio between OLDS and ILDS increases with increasing problem size, and peaks at about six in this experiment, for problems of size 35. Our analysis in Section 3 predicts a ratio of $(d+2)/2$, or 18.5 in this case, but again we assumed that all leaf nodes occur at the maximum search depth. Pruning, however, makes ILDS relatively less efficient compared to OLDS, decreasing the performance gap between them.

To the right of the peak, where perfect partitions exist, the situation is very different. As expected, the linear discrepancy searches outperform depth-first search. By searching the leaf nodes in non-decreasing order of the number of "good" moves in their paths from the root, perfect partitions are found sooner on average, and this more than compensates for the additional overhead of these algorithms. This overhead itself is significantly reduced, since only a small fraction of the tree is searched, corresponding to only a few iterations. As the problem size increases, the number of iterations decreases, and hence the gap between ILDS and OLDS quickly becomes insignificant. It is for problems in this region that OLDS was designed.

We have run these experiments on problems of up to 300 numbers. For problems larger than 200, depth-first search begins to outperform LDS again. The reason is that in this region, a perfect partition is found within the first two iterations, and the overhead of starting the second iteration from the root of the tree, as described in Section 5, becomes significant. Replacing ILDS with the RBFS version of the algorithm would remove this effect.

Note that Figure 5 shows the relative performance of these algorithms as a function of problem difficulty. To the left of the peak, there are no perfect partitions, and the entire tree must be searched. Depth-first search is the best algorithm in this case. To the right of the peak, once a perfect partition is found, the search can be terminated, and limited discrepancy search is a better choice. ILDS is never less efficient than OLDS, and is more more efficient when a significant fraction of the tree must be searched, and hence should be chosen over OLDS. Finally, for the hardest problems, occurring near the peak in Figure 5, depth-first search is the algorithm of choice.

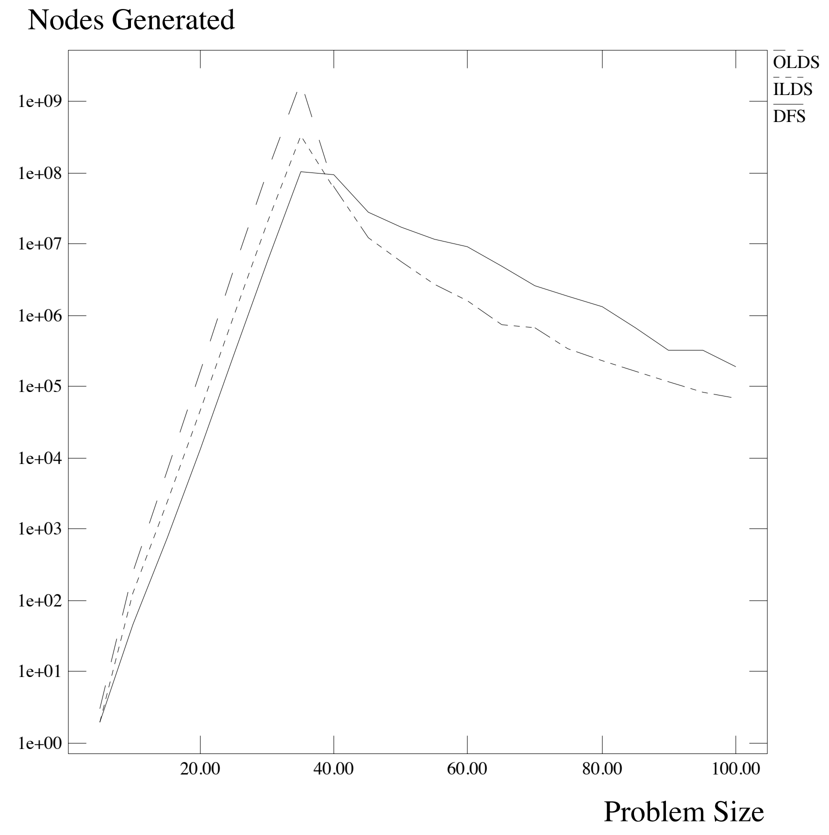

Figure 5: Nodes Generated to Optimally Partition 10-Digit Integers

The relative performance of the algorithms is different in different parts of the graph. To the left of the peak, perfect partitions don't exist, and the entire tree must be searched. As expected, depth-first search is the most efficient algorithm in this region, since it generates each node only once.

Our improved ILDS algorithm is the next best choice in this region, but is less efficient than depth-first search, since it must generate interior nodes multiple times. The measured node generation ratio between these two algorithms is roughly a constant factor of 3.5. While our analysis in Section 4 predicts a factor of two overhead, it assumes that all leaf nodes are at the maximum search depth. Pruning increases the overhead of ILDS relative to DFS, since leaf nodes of the pruned tree occur at shallower depths, and hence are generated multiple times by ILDS, as if they were interior nodes.

Whenever we reach a node in which the largest number is greater than or equal to the sum of all the other numbers, the optimal thing to do is to place the largest number in one subset and all the others in the other subset. Thus, we prune the tree below any such node. Finally, if a perfect partition is found, the entire search is terminated at that point. The actual partition is easily recovered by some additional bookkeeping.

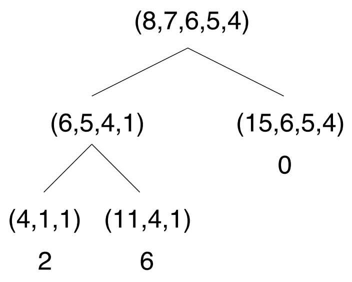

Figure 4: Tree Generated to Optimally Partition (4,5,6,7,8)

For our experiments, we chose random integers uniformly distributed from zero to 10 billion, which have 10 digits of precision. We varied the number of numbers being partitioned from 5 to 100, in increments of 5, and for each data point we averaged 100 different trials. Figure 5 shows our results. The horizontal axis represents the number of numbers being partitioned, or the problem size, and the vertical axis represents the number of nodes generated to optimally solve problems of the given size, on a logarithmic scale. The three different lines correspond to depth-first search (DFS), the original limited discrepancy search (OLDS), and our improved version (ILDS).

The first thing to notice about this graph is that for all three algorithms, the difficulty of the problem initially increases with increasing problem size, reaches a peak, and then decreases. For problem sizes to the left of the peak, perfect partitions don't exist. Thus, to find an optimal partition, the entire tree must be searched. The larger the problem size, the larger the tree, and the more nodes that are generated.

To the right of the peak, perfect partitions do exist, and as soon as one is found, the search is terminated. As the problem size increases, the density of perfect partitions increases as well, making them easier to find, and hence reducing the total number of nodes generated. The peak in problem difficulty occurs where the probability that a perfect partition exists is one-half.

algorithm, as we will see in the experiments below.

6 Experiments on Number Partitioning

To evaluate the performance of LDS in a new domain, and to compare the relative performance of OLDS, ILDS, and DFS, we implemented all three algorithms for the problem of number partitioning. Given a set of integers, the two-way partitioning problem is to divide them into two subsets, so that the sums of the numbers in each subset are as nearly equal as possible. For example, given the numbers $(4, 5, 6, 7, 8)$, if we divide them into the subsets $(4, 5, 6)$ and $(7, 8)$, the sum of the numbers in each subset is 15, and the difference between the subset sums is zero. In addition to being optimal, this is also a perfect partition. If the sum of all the numbers is odd, a perfect partition has a difference of one. Number partitioning is NP-Complete[1].

The algorithm we implemented to find an optimal partition is described in detail in [6], and is based on a polynomial-time heuristic due to Karmarkar and Karp[3]. First we sort the numbers in decreasing order. Then, we search a binary tree, where each node represents a set of numbers to be partitioned, with the root corresponding to the original numbers. Figure 4 shows the tree that results from partitioning the numbers $(4, 5, 6, 7, 8)$. The numbers below the leaf nodes represent the differences of the final subset sums.

The right branch of each node represents a decision to keep the two largest numbers together in the same subset. This is done by replacing them with their sum. For example, the right branch from the root in Figure 4 replaces 8 and 7 by their sum of 15. The sum is then treated like any other number.

The left branch of each node represents a decision to separate the two largest numbers in different subsets. This is done by replacing them by their difference in the sorted order. For example, if the 8 and 7 are in different subsets, this is equivalent to assigning their difference of 1 to the subset that contains the 8. This is because we can subtract 7 from both subsets without effecting the final subset difference. Thus, we remove the 7 and 8, and replace them with 1, which is inserted in the sorted order, and henceforth treated like any other number. In general, replacing the two largest numbers by their difference is better than replacing them with their sum, since differencing reduces the size of the remaining numbers, making it easier to minimize the resulting subset difference.

For example, for a binary tree, this is only a factor of two, and the ratio decreases with increasing branching factor.

Our analysis is for a tree of uniform depth d, and is optimistic. With any pruning, the overhead of ILDS increases. For example, if all the nodes are pruned one level above the maximum depth, then in a binary tree ILDS generates $4 \cdot 2^d - 2^d = 3 \cdot 2^d$ nodes, while DFS generates only $2 \cdot 2^d - 2^d = 2^d$ nodes, for an overhead factor of three instead of two. We will see this effect in our experiments in Section 6.

5 A Small Additional Improvement

Consider a deep tree where a solution is found on the second leaf node from the left. Depth-first search will find this node after generating $d + 1$ nodes, where d is the depth of the tree. ILDS, however, will have to generate $2d$ nodes, since it starts each iteration from the root of the tree. Thus, ILDS generates almost twice as many nodes as depth-first search in this case. The solution to this problem is instead of starting from the root each time, start each iteration from the last path of the previous iteration.

To see how this is done, think of LDS as a best-first search, where the cost of a node is the number of discrepancies in its path from the root. A standard best-first search with this cost function would execute LDS without generating any node more than once, but would require exponential space. Recursive best-first search (RBFS), however, executes a best-first search, with an arbitrary cost function, using space that is only linear in the search depth, at the cost of generating some nodes more than once[5]. RBFS saves only the current search path, and the cost of all siblings of the nodes on this path. Using the discrepancy cost function, after generating the leftmost path, the leaf node is assigned a value of infinity. All the siblings of this path will have one discrepancy, and RBFS will start exploring these paths in order of their distance from the leftmost leaf. After completing the second iteration, it will begin the third iteration starting with the last path of the second iteration. The reader is referred to [5] for full details on this algorithm.

This implementation of LDS saves the work of generating the initial path from the root to a leaf in each iteration, a savings of roughly d^2 nodes in a tree of depth d. While this savings is quite small, it could be important in cases where a solution is found very early in the second iteration of the

Now consider a node at depth $d-1$, one level above the leaves, and assume that its path from the root has k discrepancies. Its left child also has k discrepancies, but its right child has $k+1$ discrepancies. Therefore, its children are generated on two different iterations of ILDS, and the parent must be generated twice. There b^{d-1} such nodes.

Next consider a node at depth $d-2$, two levels above the leaves, again with k discrepancies in its path from the root. Its leftmost grandchild has k discrepancies as well, its rightmost grandchild has $k+2$ discrepancies, and its two remaining grandchildren each have $k+1$ discrepancies. Thus, the parent must be generated three times, once during each of these different iterations, and there are b^{d-2} such nodes.

In general, a node at depth $d-n$ is generated $n+1$ times, and there are b^{d-n} such nodes. Thus, the total number of nodes generated by ILDS in a complete b-ary tree of depth d, $ILDS(b,d)$, is

$$ILDS(b,d) = b^d + 2b^{d-1} + 3b^{d-2} + \cdots + (d-1)b^2 + db$$

Interestingly, this is the same as the number of nodes generated by a depth-first iterative-deepening search, which is a series of depth-first search iterations starting at depth one and increasing to depth d[4]. In other words,

$$ILDS(b,d) = DFS(b,1) + DFS(b,2) + \cdots + DFS(b,d)$$

$$\approx b\frac{b}{b-1} + b^2\frac{b}{b-1} + \cdots + b^d\frac{b}{b-1}$$

$$= \frac{b}{b-1}(b + b^2 + \cdots + b^d) \approx \frac{b}{b-1}(1 + b + b^2 + \cdots + b^d)$$

$$= \frac{b}{b-1}\frac{b^{d+1}-1}{b-1} \approx \frac{b}{b-1}\frac{b^{d+1}}{b-1} = \frac{b}{b-1}b^d\frac{b}{b-1} = b^d\left(\frac{b}{b-1}\right)^2$$

The ratio of the total number of nodes generated by ILDS to the total number of nodes generated by depth-first search is

$$\frac{ILDS(b,d)}{DFS(b,d)} = \frac{b^d\left(\frac{b}{b-1}\right)^2}{b^d\frac{b}{b-1}} = \frac{b}{b-1}$$

Since the sum of the binomial coefficients is 2^d, the number of leaf nodes,

$$2x = (d+2)2^d \qquad \text{or} \qquad x = \frac{d+2}{2}2^d$$

which is the asymptotic time complexity of OLDS, since the interior node generations don't affect the asymptotic complexity. Since the complexity of ILDS is only $O(2^d)$, the performance of OLDS could be as much as a factor of $(d+2)/2$ worse than that of ILDS on a binary tree. For example, on a binary tree of uniform depth 30, which contains about a billion leaves, OLDS would generate 16 times as many leaf nodes as ILDS.

This is a worst-case scenario, however, for two reasons. The first is that OLDS was designed for very large trees, in which case only the first few iterations could be performed, dramatically reducing the relative inefficiency of the original algorithm. Secondly, in most applications, such as constraint-satisfaction problems or branch-and-bound, there is a great deal of pruning, meaning that most branches terminate before the maximum depth is reached. Terminal nodes above the maximum depth are generated multiple times by ILDS, and hence the relative inefficiency of OLDS compared to ILDS is also reduced. This will be seen in the empirical results presented in Section 6.

4 Interior Node Overhead of ILDS

While ILDS generates each leaf node at the maximum search depth exactly once, it is still less efficient than depth-first search, because it generates interior nodes multiple times. Here we analyze this overhead. Assume a complete search to a uniform depth d, on a tree with branching factor b.

First, consider the number of nodes generated by depth-first search, $DFS(b,d)$, which is just the total number of nodes in the tree. Since the number of nodes at depth d is b^d, the total number of nodes is

$$DFS(b,d) = 1 + b + b^2 + \cdots + b^d = \frac{b^{d+1}-1}{b-1} \approx \frac{b^{d+1}}{b-1} = b^d\frac{b}{b-1}$$

For limited discrepancy search in a b-ary tree, we assume that any branch that is not the leftmost is a discrepancy. The leaf nodes at depth d are each generated exactly once by ILDS, in the iteration corresponding to the number of right branches in their path from the root, and there are b^d such nodes.

do this by counting the number of leaf nodes generated by the two algorithms. Assume we are searching a complete binary tree of uniform depth d. Since each iteration of ILDS generates those paths with exactly k discrepancies, each leaf node is generated exactly once, by the iteration in which k is equal to the number of right branches in its path from the root. Since a complete binary tree to depth d has 2^d leaf nodes, ILDS generates 2^d leaf nodes. This is also the asymptotic time complexity of the algorithm, since the interior node generations don't affect the asymptotic complexity, as we'll see in Section 4.

The number of leaves generated by the original LDS algorithm is more complex. To count them, we need to count the number of paths with k discrepancies. There is one path with zero discrepancies, the leftmost one. There are d paths with one discrepancy, since the single right branch could occur at any level in the tree. In general, the number of different paths with k discrepancies is $\binom{d}{k}$, the number of ways of choosing k right branches out of a total of d branches.

A total of $d+1$ iterations of LDS are needed to completely search a tree of depth d, since the number of discrepancies can range from 0 to d. The single path in the zeroth iteration is generated $d+1$ times, once in every iteration. The d paths in the first iteration are each generated d times, once in each iteration except the first. In general, if x is the number of paths, or leaf nodes, generated by OLDS in a complete search to depth d, then,

$$x = (d+1)\binom{d}{0} + d\binom{d}{1} + (d-1)\binom{d}{2} + \cdots + 2\binom{d}{d-1} + 1\binom{d}{d}$$

Writing the same terms in reverse order,

$$x = 1\binom{d}{d} + 2\binom{d}{d-1} + 3\binom{d}{d-2} + \cdots + d\binom{d}{1} + (d+1)\binom{d}{0}$$

Since $\binom{d}{k} = \binom{d}{d-k}$ for any k, adding the two equations together gives

$$2x = (d+2)\binom{d}{0} + (d+2)\binom{d}{1} + (d+2)\binom{d}{2} + \cdots + (d+2)\binom{d}{d-1} + (d+2)\binom{d}{d}$$

Factoring out $(d+2)$ gives

$$2x = (d+2)\left(\binom{d}{0} + \binom{d}{1} + \binom{d}{2} + \cdots + \binom{d}{d-1} + \binom{d}{d}\right)$$

```
LDS (NODE, K, DEPTH)
   If NODE is a leaf, return
   If (DEPTH > K)
      LDS (left-child(NODE), DEPTH-1, K)
   If (K > 0)
      LDS (right-child(NODE), DEPTH-1, K-1)
```

Figure 3: Pseudo-code for a Single Iteration of Improved LDS

2 Improved Limited Discrepancy Search

The main drawback of the original formulation of LDS, OLDS, is that it generates some leaf nodes more than once. In particular, an iteration with k discrepancies generates all those paths with *k or less* right branches. Thus, each iteration repeats all the work of all previous iterations. For example, OLDS generates a total of 19 different paths for a depth-three binary tree, only 8 of which are unique[2]. As an extreme example, while the rightmost path is the only new path in the last iteration, OLDS regenerates the entire tree on this iteration. This is unnecessary.

Given a maximum search depth, the algorithm can be modified so that each iteration generates only those paths with *exactly k* discrepancies. This is done by keeping track of the remaining depth to be searched, and if it is less than or equal to the number of discrepancies, only right branches are explored. The modified pseudo code is shown in Figure 3. The depth parameter is the remaining depth to be searched below the current node. It is set to the maximum depth of the tree in the call on the root node. Every leaf node at the maximum depth is generated exactly once by this improved version of LDS, ILDS. Leaf nodes above the maximum depth, however, as well as interior nodes, are generated more than once.

3 Analytic Comparison of the Algorithms

We now compare the performance of these two versions of LDS analytically, to determine how much is saved by the improved version of the algorithm. We discrepancies in a binary tree of depth three. Figure 2 gives a pseudo-code description of a single iteration of LDS on a binary tree. Its arguments are a node, and the number of discrepancies k for that iteration. This function is called once for each iteration, with k ranging from zero to the maximum tree depth, terminating if a goal is found.

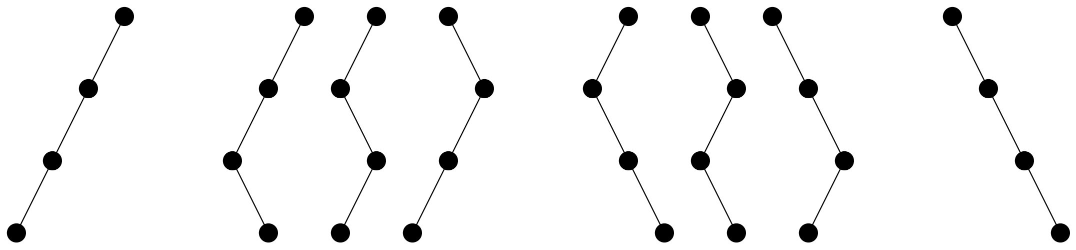
Figure 1: Paths with 0, 1, 2, and 3 Discrepancies in a Depth 3 Binary Tree

LDS can be applied to any tree-search problem where one branch from each node is preferred to that of its siblings. The simplest extension to a non-binary tree is to treat any branch except the leftmost as a single discrepancy. In Harvey and Ginsberg's analysis of the algorithm[2], they consider the left branch to have a higher probability of containing a solution in its subtree than the right branch, and show that limited discrepancy search has a higher probability of finding a solution than depth-first search, for a given number of node generations. They also show experimentally that it outperforms depth-first search in a constraint-satisfaction scheduling task.

```
LDS (NODE, K)
   If NODE is a leaf, return
   LDS (left-child(NODE), K)
   If (K > 0) LDS (right-child(NODE), K-1)
```

Figure 2: Pseudo-code for a Single Iteration of Original LDS

is not feasible. In that case, we would like to search as much of the tree as possible in the time available, and then return the best solution found. Depth-first search is not necessarily the best choice in this case.

For example, consider the following tree-search problem. There are costs associated with the edges of a binary tree, and the cost of a node is the sum of the edge costs from the root to that node. We want to find a leaf node of lowest cost. The best algorithm to exactly solve this problem is depth-first branch-and-bound, with node ordering. The idea of node ordering is that at each node, we search the lower-cost child first. If the tree is too large to search exhaustively, we would like to search those leaves that are most likely to have the lowest costs first.

Assume that the edge costs are independent random variables uniformly distributed from zero to one. The expected value of the minimum of two such variables is $1/3$, and the expected value of the maximum is $2/3$. Thus, if we reorder the tree, the expected value of a left branch is $1/3$, and the expected value of a right branch is $2/3$. The leaf node with the lowest expected total cost is the leftmost. If d is the depth of the tree, its expected cost is $d/3$. This is also the first leaf node visited by depth-first search. The second leaf node from the left has $d-1$ left branches, and one right branch in its path from the root, and hence an expected cost of $(d-1)/3+2/3$. This is also true of the third leaf node from the left. The fourth leaf from the left, however, has an expected cost of $(d-2)/3+4/3$, since it has two right branches in its path from the root. In general, the leaf nodes with the second-lowest expected value, after the leftmost leaf, are those with all left branches except for one right branch, in their path from the root. This includes, for example, the path that goes right from the root, and then left thereafter. The leaf nodes with the next lowest cost are those with all left branches except for two right branches. Thus, depth-first search does not search the leaf nodes in non-decreasing order of total expected cost.

Limited discrepancy search (LDS), invented by Harvey and Ginsberg[2], does search the leaf nodes in non-decreasing order of cost. A discrepancy corresponds to a right branch in an ordered tree. The first path generated by LDS is the leftmost path. Next, it generates those paths with at most one right branch from the root to the leaf. The next set of paths generated by LDS are those with at most two right branches, etc. This continues until every path in the tree has been generated, with the rightmost path being generated last. Figure 1 shows the sets of paths with 0, 1, 2, and 3

Improved Limited Discrepancy Search

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu

July 26, 1995

Abstract

We present an improvement to Harvey and Ginsberg's limited discrepancy search algorithm. Our version eliminates much of the redundancy in the original algorithm, generating each search path from the root to the maximum search depth only once. For a uniform-depth binary tree of depth d, this reduces the asymptotic complexity from $O(\frac{d+2}{2}2^d)$ to $O(2^d)$. The savings is much less in a partial tree search, or in a heavily pruned tree. We also show that the overhead of the improved algorithm on a uniform-depth b-ary tree is only a factor of $b/(b-1)$ compared to depth-first search. This constant factor is greater on a heavily pruned tree. Finally, we present empirical results showing the utility of limited discrepancy search, as a function of problem difficulty, on the NP-Complete problem of number partitioning.

1 Introduction: Limited Discrepancy Search

The best-known tree-search algorithms are breadth-first and depth-first search. Breadth-first search is rarely used in practice, because it requires space that is exponential in the search depth. Depth-first search, however, uses only linear space, and hence is often the algorithm of choice for trees that are to be searched exhaustively. For a very large tree, however, exhaustive search