Improved Limited Discrepancy Search

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

Abstract

We present an improvement to Harvey and Ginsberg's limited discrepancy search algorithm, which eliminates much of the redundancy in the original, by generating each path from the root to the maximum search depth *only once*. For a complete 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. The overhead of the improved algorithm on a complete b-ary tree is only a factor of $b/(b-1)$ compared to depth-first search. While this constant factor is greater on a heavily pruned tree, this improvement makes limited discrepancy search a viable alternative to depth-first search, whenever the entire tree may not be searched. Finally, we present both positive and negative empirical results on the utility of limited discrepancy search, for the problem of number partitioning.

Limited Discrepancy Search

The best-known search algorithms are breadth-first and depth-first search. Breadth-first search is rarely used in practice, since it requires space that is exponential in the search depth. Depth-first search uses only linear space, and thus is often used for exhaustive searches. For a very large tree, however, exhaustive search is not feasible. In that case, we would like to search as much of the tree as possible in the time available, and return the best solution found. Depth-first search is not necessarily the best choice in this case.

For example, consider a binary tree with costs associated with the edges, where 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 known algorithm to solve this problem exactly is depth-first branch-and-bound, searching the lower-cost child of each node first. If the tree is too large to search completely, we would like to search those leaves that are most likely to be of lowest cost 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 order the children of a node, the expected value of the left branch is $1/3$, and the expected value of the 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 $d/3$. The leaf nodes with the second-lowest expected cost are those with all left branches except for one right branch in their path from the root. The expected cost of these leaves is $(d-1)/3+2/3$. The second and third leaf nodes from the left are in this set, but so is the leaf reached by going right from the root, and then left thereafter. The nodes with the next lowest expected cost are those with exactly two right branches in their path from the root, with an expected cost of $(d-2)/3+4/3$. Thus, depth-first search does not search the leaf nodes in non-decreasing order of expected cost.

Limited discrepancy search (LDS) (Harvey and Ginsberg 1995), does however. A discrepancy is a right branch in a heuristically ordered tree. The first path generated by LDS is the leftmost. 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 has been generated, with the rightmost path being generated last. Figure 1 shows the sets of paths with 0, 1, 2, and 3 discrepancies in a binary tree of depth three.

The first algorithm that searched a tree in this fashion, was called *left-first search* (Basin and Walsh 1993). Their formulation of the algorithm used exponential space, however, and hence is not practical for large searches. (Harvey and Ginsberg 1995) independently discovered a linear-space version of the algorithm, which they called limited discrepancy search, and recognized its more general applicability.

Figure 2 gives a pseudo-code description of a single iteration of Harvey and Ginsberg's original LDS algorithm (OLDS) 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.

LDS can be applied to any search problem where one branch from each node is preferred to its siblings. The simplest extension to a non-binary tree is to treat

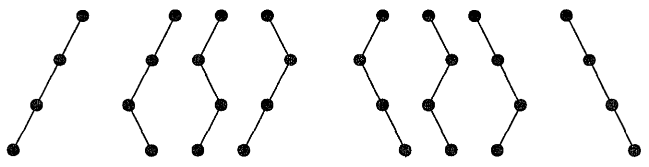
Figure 1: Paths with 0, 1, 2, and 3 discrepancies

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

Figure 2: Pseudo-code for one iteration of original LDS

any branch except the leftmost as a discrepancy. Harvey and Ginsberg showed that if the left branch has a higher probability of containing a solution in its subtree than the right branch, then limited discrepancy search has a higher probability of finding a solution than depth-first search, for a given number of node generations. They also showed that it outperforms depth-first search on a set of job-shop scheduling problems.

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, the iteration for k discrepancies generates all paths with *k or less* right branches. Thus, each iteration regenerates all the paths of all previous iterations. For example, OLDS generates a total of 19 paths on a depth-three binary tree, only 8 of which are unique, as shown in Figure 2 of (Harvey and Ginsberg 1995). As an extreme case, while the rightmost path is the only new path in the last iteration, OLDS regenerates the entire tree on this iteration.

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, exploring only right branches below that node. The modified pseudo code is shown in Figure 3. The depth parameter is the depth to be searched below the current node. It is set to the maximum depth of the tree in each 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. This interior node overhead is analyzed below.

Analytic Comparison of the Algorithms

Here we compare the performance of these two versions of LDS analytically, to see how much is saved

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

Figure 3: Code for one iteration of improved LDS

by the improvement. To do this, we count the number of leaf nodes generated by the two algorithms on a complete binary tree of depth d. Since each iteration of ILDS generates those paths with exactly k discrepancies, each leaf node is generated exactly once, by the iteration corresponding to the number of right branches in its path from the root. Since a binary tree of depth d has 2^d leaf nodes, ILDS generates 2^d leaves. 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 later.

The number of leaves generated by OLDS is more complex. First, we need to count the number of paths with k discrepancies. There is one path with no 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, there are $\binom{d}{k}$ branches with k discrepancies, the number of ways of choosing k right branches out of d branches.

To search a tree of depth d, $d+1$ iterations of OLDS are needed, 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, etc. In general, let x be the number of paths generated by OLDS in a complete search to depth d.

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

Writing the same equation in reverse order,

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

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

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

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

Since the sum of the binomial coefficients is 2^d,

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

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

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

Interior Node Overhead of ILDS

While ILDS generates each leaf at the maximum 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 uniform tree of branching factor b and depth d.

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

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

where the approximation is valid for large d.

For LDS in a b-ary tree, any branch other than 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.

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

Next consider a node at depth $d-2$, two levels above the leaves, 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, 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

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

This is the same as the number of nodes generated by a depth-first iterative-deepening search, which is a series of depth-first searches starting at depth one and increasing to depth d (Korf 1985). In other words,

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

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

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

On a binary tree, this is a only 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 of three instead of two. We'll see this effect in our experiments. In any case, our improvement to LDS makes it a viable alternative to depth-first search, whenever the entire tree may not be searched. In contrast, OLDS is practical only if a tiny fraction of the tree is searched.

A Small Additional Improvement

Another view of limited discrepancy search is that it is a best-first search, where the cost of a node is the number of discrepancies in its path from the root. If we implement such a search using recursive best-first search (RBFS) (Korf 1993), each iteration starts with the last path of the previous iteration. This saves the overhead of generating the first path of each iteration from the root node, and is significant when a solution is found very early in the second iteration. We observed this effect in very large number partitioning problems.

Experiments on Number Partitioning

To compare the relative performance of OLDS, ILDS, and depth-first search, we implemented these algorithms on the NP-complete problem of number partitioning. Given a set of integers, the two-way partitioning problem is to divide them into two mutually exclusive and collectively exhaustive 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 will have a difference of one, since the total sum can't be evenly divided. Below we present experiments with two different partitioning algorithms.

Complete Karmarkar-Karp Algorithm

The best method known for finding optimal partitions of large problem instances[1] is the Complete Karmarkar-Karp (CKK) algorithm (Korf 1995), based on a heuristic of (Karmarkar and Karp 1982). First the numbers are sorted in decreasing order. Then, a binary tree is searched, with each node representing a set of numbers to be partitioned, and the root corresponding to the original numbers. Figure 4 shows the tree that results from partitioning the numbers (4, 5, 6, 7, 8).

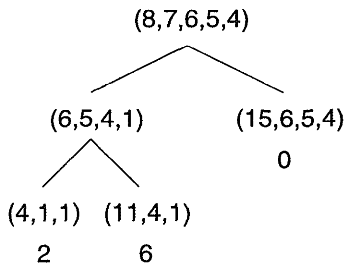

Figure 4: CKK tree to partition (4,5,6,7,8)

The right branch of each node keeps 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 15, which is then treated like any other number.

The left branch of each node keeps the two largest numbers apart in different subsets. This is done by replacing them by their difference. For example, assigning the 8 and 7 to different subsets is equivalent to assigning their difference of 1 to the subset that contains the 8, since we can subtract 7 from both subsets without effecting the final subset difference. Thus, we replace the 7 and 8 with 1, which is inserted in the sorted order, and then treated like any other number. In general, replacing the 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 final subset difference. The Karmarkar-Karp heuristic always replaces the two largest numbers with their difference, corresponding to the left-most path in Figure 4.

When we reach a node where the largest number is greater than or equal to the sum of all the others, the best we can do is to place the largest number in a separate subset from the others. Thus, we prune the tree below any such node. The leaves of Figure 4 show examples of such pruning, and the numbers below them represent the final subset differences. Finally, if a perfect partition is found, the entire search is terminated at that point, as illustrated by the right child of the root in Figure 4. The actual subsets themselves are easily constructed by some additional bookkeeping.

[1]See also (Bright, Kasif, and Stiller 1994) however.

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

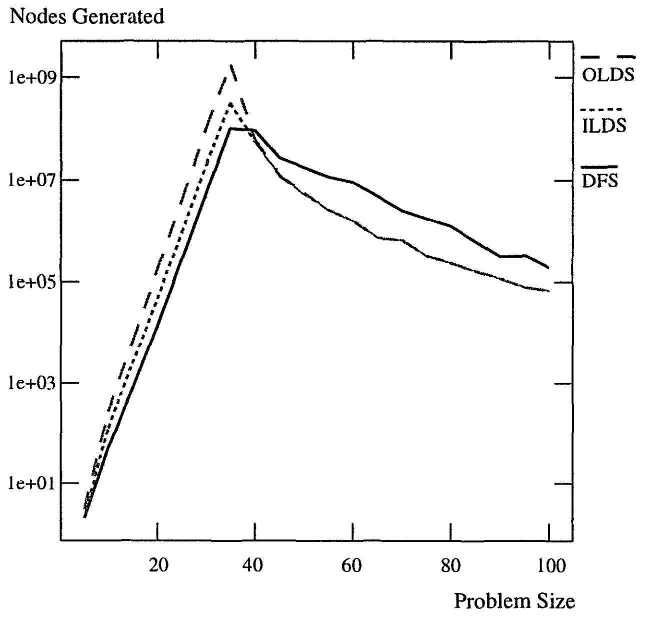

Figure 5: CKK performance on 10-digit integers

Notice that for all three algorithms, the problem difficulty initially increases with increasing problem size, reaches a peak at 35, and then decreases. For small problems, perfect partitions don't exist, and to find an optimal partition the entire tree must be searched. The larger the problem, the larger the tree, and the more nodes that are generated.

For large problems, however, 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 also increases, making them easier to find, and reducing the number of nodes generated. The peak in problem difficulty occurs where the probability of a perfect partition is about one-half. See (Gent and Walsh 1996) for more detail on this phenomenon.

The relative performance of the algorithms is different in different parts of the graph. To the left of the peak, the entire tree must be searched, and DFS is the best choice, since it generates each node only once.

Our improved ILDS algorithm is the next best choice in this region. The measured node generation ratio between ILDS and DFS is roughly a constant factor of 3.5. While our previous analysis 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.

The performance of the original OLDS algorithm is even worse in this region, The node generation ratio between OLDS and ILDS increases with increasing problem size, and peaks at about six, for problems of size 35. Our analysis predicts a ratio of $(d+2)/2$, or 18.5 in this case, but that assumes that all leaf nodes occur at the maximum search depth. Pruning, however, decreases the performance gap between ILDS and OLDS.

To the right of the peak, where perfect partitions exist, the situation is very different. As expected, both OLDS and ILDS outperform DFS. By searching the leaf nodes in non-decreasing order of the number of differencing steps in their paths from the root, perfect partitions are found sooner on average, and this more than compensates for the additional node generation overhead. The overhead itself is also reduced, since only a few iterations are needed. As the problem size increases, the number of iterations decreases, and the gap between ILDS and OLDS becomes insignificant.

ILDS is never less efficient than OLDS, and is much more efficient when a significant fraction of the tree must be searched. For the hardest problems, occurring near the peak in Figure 5, DFS is the algorithm of choice. For all three algorithms, the time per node generation is roughly the same, so that the ratio of node generations is also the ratio of running times.

A Simpler Algorithm and Negative Result

Here we present a search space for which LDS performs worse than DFS. A simpler complete algorithm for number partitioning is to sort the numbers in decreasing order, and then search a binary tree, where at each node the left branch assigns the next number to the subset with the smaller sum, and the right branch assigns it to the subset with the larger sum. A branch is pruned when the sum of the remaining unassigned numbers is less than or equal to the difference of the current subset sums. Figure 6 shows the tree that results from partitioning the set $(4,5,6,7,8)$ using this algorithm, where the numbers outside the parentheses are the current subset sums, and the numbers inside the parentheses remain to be assigned. The reason this tree is larger than that in figure 4 is that pruning is less effective for this algorithm.

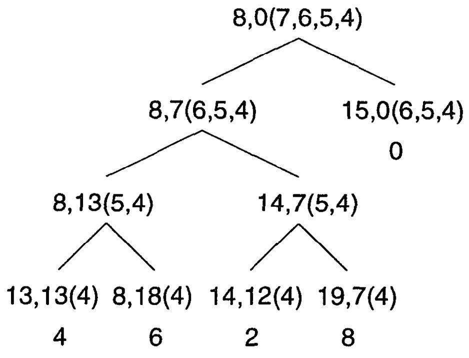

Figure 6: Simpler tree to partition (4,5,6,7,8)

As in the case of CKK, we can apply LDS to search these trees as well. Since this simple algorithm is considerably less efficient than CKK, in order to gather sufficient data we chose uniformly distributed random eight-digit integers, from 0 to 100 million. For the same reason, we only considered the improved ILDS version in these experiments. Figure 7 shows the results, in a form identical to that of Figure 5. Each data point is the average of 100 random problem instances.

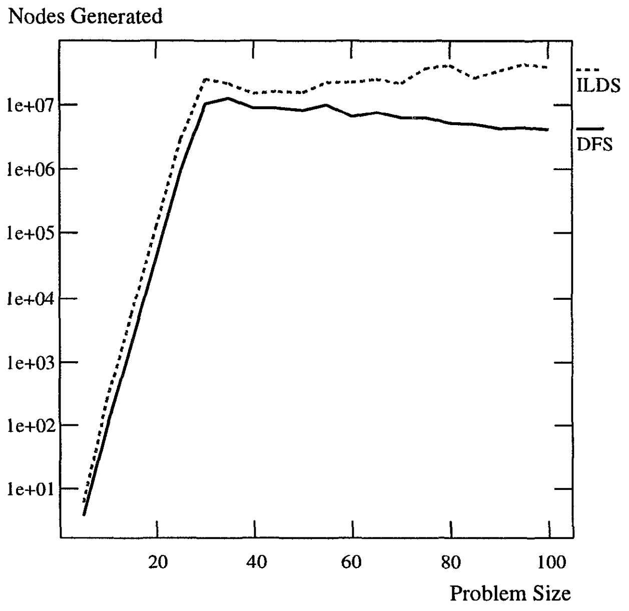

Figure 7: Simple algorithm on 8-digit integers

Again we see that for small problems without perfect partitions, DFS outperforms ILDS, as expected. However, for large problems with perfect partitions, DFS continues to outperform ILDS. This is the opposite of what we found on the CKK trees. Furthermore, the performance gap is larger than the overhead of ILDS, and increases with increasing problem size.

One explanation for this anomaly is as follows. The heuristic of placing each number in the subset with the smaller sum is much weaker than the differencing heuristic of Karmarkar and Karp. This is shown by the fact that CKK dramatically outperforms our simple complete algorithm, particularly when perfect partitions exist (Korf 1995). Since LDS is heavily guided by the heuristic that recommends the left branches over the right, its performance relative to DFS should degrade with a weaker heuristic. This doesn't explain why it would perform worse than DFS however.

We believe that the main weakness of LDS is that it is a dilettante. On a big tree, it explores a large number of paths to the frontier that are widely spaced. On any path, once all the discrepancies have been exhausted, it makes only a single probe below that point, always staying to the left. Since each node represents a subproblem, it makes only one attempt at these subproblems, then goes on to another subproblem, rather than working further on the given one. With the simple partitioning algorithm, the decisions made at the top of the tree probably don't matter much, but diligence in trying a large number of alternatives near the bottom of the tree may be important. As a result, LDS performs worse than DFS on these trees. The same argument could also be applied to the CKK trees, but we believe that the extra power of the differencing heuristic, at all levels of the tree, overcomes this weakness.

In contrast to LDS, however, the weakness of DFS is its excessive diligence. Once it sets its mind on a subproblem represented by a node, it doggedly pursues that subproblem until it either finds a solution or proves that no solution exists, before considering any other subproblems. This drawback of DFS was the original motivation behind the development of alternatives such as LDS. Clearly, neither algorithm is ideal for all problems, but what is required is a balance between diligence and dilettantism, a balance that is difficult to identify and most likely problem specific.

Conclusions

Limited discrepancy search (LDS) is often an effective algorithm for problems where only part of a tree must be searched to find a solution. We presented an improved version of the algorithm (ILDS) that reduces its time complexity from $O(\frac{d+2}{2}2^d)$ to $O(2^d)$ for a complete search a uniform binary tree of depth d. In practice, 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 ILDS always generates fewer nodes than OLDS, 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 ILDS compared to depth-first search (DFS) is only 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 binary trees in our experiments. Thus, ILDS is a viable alternative to DFS, whenever the entire tree may not be searched. In contrast, OLDS is only practical when an insignificant fraction of the tree will be searched.

We evaluated the performance of LDS on two different problem spaces for number partitioning. In both cases, DFS outperforms LDS on the hardest problem instances. For problems with many perfect solutions, DFS outperforms LDS in the simple space, while LDS outperforms DFS in the more efficient search space. The stronger the ordering heuristic, the better LDS should perform relative to DFS. The weakness of LDS is dilettantism, whereas the weakness of DFS is excessive diligence. An ideal algorithm should strike the right balance between these two properties.

Acknowledgements

Thanks to Matt Ginsberg and Will Harvey for helpful discussions concerning this research, and to Max Chickering and Weixiong Zhang for comments on an earlier draft. This work was supported by NSF Grant IRI-9119825, and a grant from Rockwell International.

References

Basin, D.A., and T. Walsh, Difference unification, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-93)*, Chamberry, France, Aug. 1993, pp. 116-122.

Bright, J., S. Kasif, and L. Stiller, Exploiting algebraic structure in parallel state space search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 1994, pp. 1341-1346.

Gent, I., and T. Walsh, Phase transitions and annealed theories: number partitioning as a case study, *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, Budapest, Aug. 1996.

Harvey, W.D., and M.L. Ginsberg, Limited discrepancy search, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, Aug. 1995, pp. 607-613.

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.

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

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

Korf, R.E., From approximate to optimal solutions: A case study of number partitioning, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, Montreal, 1995, pp. 266-272.