Interleaved Depth-First Search *

Pedro Meseguer
Institut d'Investigacio en Intel.ligencia Artificial
Consejo Superior de Investigaciones Cientificas
Campus UAB, 08193 Bellaterra, Spain

Abstract

In tree search, depth-first search (DFS) often uses ordering successor heuristics. If the heuristic makes a mistake ordering a *bad* successor (without goals in its subtree) before *good* ones (with goals in their subtrees), DFS has to unsuccessfully traverse the whole bad subtree before finding a goal. To prevent this useless work, we present a new strategy called *interleaved depth-first search* (IDFS), which searches depth-first several subtrees —called *active*— in parallel. IDFS assumes a single processor on which it *interleaves* DFS on active subtrees. When IDFS finds a mistake, it traverses partially the bad subtree. IDFS does not reexpand nodes and uses a memory amount linear in search depth (with a bounded number of active subtrees). IDFS outperforms DFS if the heuristic improves from the first to the second tree level. Experimental results on hard solvable problems confirm the practical validity of IDFS.

1 Introduction

In tree search with bounded depth, *depth-first search* (DFS) is widely used because it is simple and its space requirement is linear in search depth. Often, DFS is combined with some heuristic which estimates the likelihood of goal existence in subtrees rooted at internal nodes. In this case, DFS is called *ordered* because node successors are ordered from left to right by decreasing heuristic value. We say that an internal node is *good* if the subtree below it has goal nodes, and *bad* otherwise. With this definition, heuristic ordering tries to move good successors to the left and bad successors to the right. Given that DFS starts searching on the left, heuristic ordering tries to speed up goal finding.

'This research is supported by the Spanish CICYT proyect TIC96-0721-C02-02.

Heuristic advice is not perfect and, in occasions, it may be wrong. A heuristic makes a *mistake* when it orders a bad successor before a good one. Ordered DFS *falls into* a mistake when it visits the bad node. If mistakes appear in the ordering of current node successors, ordered DFS falls into them as it progresses from left to right, until it finds a good node. Falling into a mistake at shallow levels of the search tree is quite costly, because DFS is forced to unsuccessfully traverse a large subtree without goals. Falling into a mistake at deep levels is less costly, because DFS has to traverse a smaller subtree. Typically, heuristics provide better advice at deep levels than at shallow levels, so mistakes are more likely to appear at shallow levels, where they are more costly to recover. The presence of mistakes, specially at shallow levels, is a weak point for DFS performance, against which this algorithm has no defense.

In this paper, we present a new approach to prevent — at least partially— ordered DFS falling into mistakes. We call this approach *Interleaved Depth-First Search* (IDFS), and it works as follows. While DFS searches sequentially — from left to right — subtrees at each tree level, IDFS searches in parallel several subtrees at some tree level. Calling *active* those subtrees being searched in parallel, IDFS searches depth-first the current active subtree until it finds a leaf. If it is a goal, search terminates. Otherwise, the state of the current subtree is recorded and added to a fifo queue of active subtrees, from which it will be taken later to resume search at the point it was suspended. IDFS selects another active subtree as the new current subtree and repeats the process. IDFS simulates parallel DFS on active subtrees using a single processor. In this sense, IDFS *interleaves* depth-first search on active subtrees. If the heuristic orders successors with mistakes, IDFS avoids falling completely into them by distributing search among active subtrees, which are expected to include some good subtree.

IDFS provides advantages over DFS in average performance on hard solvable problems. Performance improvement is directly related to mistake occurence: the more

mistakes there are, the higher the performance improvement of IDFS over DFS. If no mistakes occur DFS is the best algorithm. IDFS is meaninful for solvable problems only. Unsolvable problems require both IDFS and DFS to traverse the whole search tree to detect goal absence, so they will expand the same nodes —although in different order— causing no difference in performance[1]. IDFS provides practical advantages over DFS on hard problems, where many mistakes are likely to occur at shallow levels. On easy problems, heuristics make few mistakes and IDFS benefits do not pay off the overhead of searching simultaneously active subtrees. On the other hand, IDFS requires more memory than DFS because it has to store active subtrees. At shallow levels, each active subtree requires almost as much storage as single DFS, so its number should be bounded to allow for the practical applicability of this algorithm.

This paper is organized as follows. In Section 2 we describe related approaches. In Section 3 we provide a detailed explanation of the IDFS algorithm, analyzing its performance in Section 4. In Section 5 we give experimental results of this algorithm on different search problems, showing its practical applicability. Finally, in Section 6 we summarize the contributions of this work.

2 Related Work

In the context of binary search trees, the idea of mistakes was introduced in the *limited discrepancy search* algorithm (LDS) [Harvey and Ginsberg, 1995]. Given a heuristic to order successors, a discrepancy is not to follow the heuristic preference at some node, that is, to select the right successor as the next node. LDS visits first the leftmost leaf of the tree, following a path without discrepancies. If it is not a goal, LDS visits leaves with at most one discrepancy in their paths, visiting first paths with discrepancies at early levels, where the heuristic is supossed to be less accurate. If no goal is found, LDS visits leaves with at most two discrepancies in their paths, etc. The proccess continues until it reaches the maximum number of discrepancies and the whole tree is searched. LDS space requirement is linear in search depth and it has to reexpand nodes previously visited. From the analysis given in [Harvey and Ginsberg, 1995], LDS outperforms DFS on easy problems.

This approach has been enhanced by the *improved limited discrepancy search* algorithm (ILDS) [Korf, 1996]. At the iteration for *k* discrepancies, LDS generates all paths with *k* or less discrepancies. But all paths with less than *k* discrepancies have already been visited in previous iterations. This is corrected by ILDS, which at each iteration generates only those paths with exactly *k* discrepancies.

[1] In fact, IDFS will require more time than DFS due to overhead of switching active subtrees.

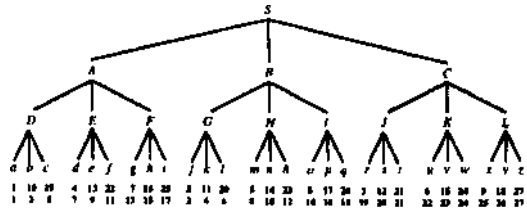

Figure 1: Example search tree. First line is pure IDFS leaf visiting order; second line is limited IDFS leaf visiting order, as explained in the text.

However, different from LDS, ILDS visits first paths with discrepancies at deep levels. Regarding results on number partitioning, DFS outperforms ILDS when no perfect partition exists. On problems with perfect partitions, DFS outperforms ILDS using a simple heuristic, while ILDS outperforms DFS when using the KK heuristic.

The efficiency of parallel DFS is analyzed in [Rao and Kumar, 1993]. Regarding search on ordered-bounded binary trees, parallel DFS expands no more nodes on the average than sequential DFS. On easy problems, those in which heuristics give good advice at early levels, parallel DFS offers no advantadge over sequential DFS. On hard problems, parallel DFS obtains substantial speedup over sequential DFS.

3 Interleaved Depth-First Search

3.1 Pure IDFS

Pure IDFS interleaves search among all successor subtrees of any internal node at any level of the search tree. IDFS searches depth-first in a subtree until it finds a leaf. If it is a goal, search terminates. Otherwise, it stores the search state of those subtrees traversed by the leaf path, and switches to the next subtree at the earliest level of the search tree. Successors of a node are ordered, from left to right, by decreasing value of some heuristic estimation of good node likelihood. Successor list is circular, so the next successor of the rightmost is the leftmost.

Pure IDFS is better explained with the example of Figure 1, a ternary tree of depth 3. IDFS interleaves search among subtrees at level 1, *A, B* and *C* in turn, switching from one to the next after reaching a leaf. When searching *A,* IDFS interleaves search among its successors D, *E* and F, switching from one to the next each time subtree *A* is searched. When searching *D,* IDFS visits its leaf successors a, 6 and c in turn, one each time subtree *D* is searched. The same process occurs for subtrees *B* and *C*. IDFS execution is as follows. It expands 5, *A, D* and visits a, a non-goal leaf. IDFS stops visiting subtrees *D* and A_j stores their states and switches to *B,* the next subtree at level 1, where IDFS repeats the process: it expands *B, G* and visits ;, a non-goal leaf. IDFS stops visiting subtrees *G* and *B,* stores their states and switches to (7,

```
IDFS(s)
1  if s is a leaf then return success or failure;
2  if s is not expanded then generate successors(s);
3  if successors(s) is not empty then
4     s' ← get_first(successors(s));
5     result ← IDFS(s');
6     case result of
7        continue: add_last(successors(s), s');
8        success:  return result;
9        failure:  do nothing;
10 if successors(s) is empty then return failure;
11 else if s is the initial node goto 3;
12 else return continue;
```

Figure 2: Pure IDFS algorithm.

the next subtree at level 1, where IDFS repeats the process: it expands C, *J* and visits r, a non-goal leaf. IDFS stops visiting subtrees *J* and C, stores their states and switches to *A*. Now, IDFS resumes search on *A,* moving to the next successor *E*. This process is repeated again and again until a goal is found or the whole tree is searched. The first line in Figure 1 indicates the order in which pure IDFS visits leaves in the example search tree. The algorithm for pure IDFS appears in Figure 2. It is a function that takes node *s* as input and can return:

success: a goal has been found,
failure: no goal found, tree exhausted,
continue: no goal found, tree not exhausted.

When this function returns *continue,* it modifies the state of *s* as a side-effect, to record the search progress in the subtree rooted at *s*.

In its current implementation, pure IDFS requires an amount of memory exponential in search depth, which renders it unapplicable. We have presented pure IDFS for clarity purposes. In the next subsection we introduce the practical version of IDFS.

3.2 Limited IDFS

Limited IDFS interleaves search among a limited number of successor subtrees at some levels of the search tree. Limited IDFS distinguishes between two kinds of levels: *parallel* levels, where search is interleaved among some of its subtrees, and *sequential* levels, where search is sequential. For simplicity, we will assume that parallel levels start at level 1 and they are consecutive. Given a node with successors in a parallel level, we call *active* those successors (and their subtrees) on which IDFS interleaves search. Limited IDFS performs as pure IDFS on active subtrees at parallel levels, and as DFS on sequential levels until it reaches a leaf. If it is a goal, search terminates. Otherwise, it stores current subtrees and switches to the next active subtree at the earliest parallel level. As in the pure case, active successors are

```
IDFS(s)
1  if s is not expanded then
2     generate successors(s);
3     active(s) ← get_k_first(successors(s));
4  if active(s) is not empty then
5     s' ← get_first(active(s));
6     if level(s') + 1 is a parallel level
7        then result ← IDFS(s');
8        else result ← IDFS_seq(s');
9     case result of
10       continue: add_last(active(s), s');
11       success:  return result;
12       failure:  if successors(s) is not empty then
13          add_last(active(s), get_first(successors(s)));
14 if active(s) is empty then return failure;
15 else if s is the initial node goto 3;
16 else return continue;

IDFS_seq (s)
1  if s is a leaf then return success or failure;
2  if s is not expanded then generate successors(s);
3  if successors(s) is not empty then
4     s' ← get_first(successors(s));
5     result ← IDFS_seq(s');
6     case result of
7        continue: add_first(successors(s), s');
8        success:  return result;
9        failure:  do nothing;
10 if successors(s) is empty then return failure;
11 else return continue;
```

Figure 3: Limited IDFS algorithm.

heuristically ordered, and the next active sucessor of the rightmost is the leftmost. In the example of Figure 1, assuming that the only parallel level is level 1 and the number of active subtrees is 2, limited IDFS works as follows. It expands 5, taking *A* and *B* as active subtrees. It expands *A, D* and visits a, a non-goal leaf. It stops searching *A,* stores its state and switches to *B,* the next active successor at level 1, where it repeats the same process: it expands B, *G* and visits j, a non-goal leaf. IDFS stops searching B, stores its state and switches to *A,* where it resumes depth-first search visiting 6, another non-goal leaf. IDFS stops searching *A* and resumes search on B_f where it visits *k*. The same process goes on until an active subtree, say *A,* is exhausted (assuming that no goal has been found). Then, *A* is replaced by *C*. The whole process terminates after finding a goal or exhausting the complete search tree. The second line in Figure 1 indicates the order in which this execution of limited IDFS visits leaves in the example search tree. The algorithm for limited IDFS appears in Figure 3. It is worth noting that limited IDFS does not reexpand nodes and, if the number of active subtrees is bounded to a number independent from the problem size, limited IDFS requires a memory space linear in search depth.

4 Efficiency Analysis

In the following we provide a probabilistic efficiency analysis of limited IDFS versus ordered DFS. Following the approach of [Rao and Kumar, 1993], efficiency is evaluated by the average number of leaves visited by each algorithm. We work on a search tree of depth d and uniform branching factor 6 containing goal nodes. For simplicity, we assume that no pruning is done. At the first level, there are b subtrees ordered from left to right, and we denote as γ_i the probability of goal existence in the i-th subtree.

Regarding DFS, the probability of finding a goal in the subtree i is the probability of goal existence at subtree t\ under the condition that no goal has been found in subtrees before i, that is,

$$\gamma_i(1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{i-1})$$

If the problem has solution, the probability that DFS finds a goal in the last subtree depends only on probabilities that no goal has been found in previous subtrees. This probability is,

$$(1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{b-1})$$

S(d) is the average number of leaves visited by DFS in a tree of depth d with goals. *F(d)* is the number of leaves visited by DFS in a tree of depth d without goals (it is equal to b^d). *S(d)* is,

$$\begin{aligned} S(d) &= S_1(d-1)\gamma_1 + \\ &\quad [S_2(d-1)+F(d-1)]\gamma_2(1-\gamma_1) + \\ &\quad [S_3(d-1)+2F(d-1)]\gamma_3(1-\gamma_1)(1-\gamma_2) + \\ &\quad \dots \\ &\quad [S_b(d-1)+(b-1)F(d-1)] \\ &\qquad (1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{b-1}) \\ &= \hat{S}(d-1) + F(d-1)\Gamma = \hat{S}(d-1) + b^{d-1}\Gamma \end{aligned} \tag{1}$$

where summand i-th is the expected number of visited leaves by DFS if it finds the goal in the i-th subtree and not before. Expressions $S(d-1)$ and T are,

$$\begin{aligned} \hat{S}(d-1) &= S_1(d-1)\gamma_1 + S_2(d-1)\gamma_2(1-\gamma_1) + \dots + \\ &\quad S_b(1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{b-1}) \end{aligned} \tag{2}$$

$$\begin{aligned} \Gamma &= \gamma_2(1-\gamma_1) + 2\gamma_3(1-\gamma_1)(1-\gamma_2) + \dots + \\ &\quad (b-1)(1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{b-1}) \\ &= (1-\gamma_1) + (1-\gamma_1)(1-\gamma_2) + \dots + \\ &\quad (1-\gamma_1)(1-\gamma_2)\dots(1-\gamma_{b-1}) \end{aligned} \tag{3}$$

Regarding IDFS, we assume that level 1 is the only parallel level and all its subtrees are active. *I(d)* is the average number of leaves visited by IDFS in a tree of depth d with goals. *I(d)* is,

$$I(d) = b\ min_i\{S_i(d-1)\};\quad i = 1,\dots,b$$

Using (1), we get

$$S_i(d-1) = \hat{S}_i(d-2) + F(d-2)\,\Gamma^i$$

where $\hat{S}_i(d-2)$ and Γ^i are given by (2) and (3) respectively with the following substitutions,

1. γ_j, the probability of goal existence in subtree j at first level of the search tree, is substituted by γ_j^i, the probability of goal existence in subtree j at second level inside subtree i at first level, and
2. $S_j(d-2)$, the average number of visited leaves in subtree j with depth $d-2$, is substituted by $S_j^i(d-2)$, the average number of visited leaves in subtree j at second level inside subtree i at first level.

To obtain an analytical expression for $S_i(d-1)$, we take two approximations: (i) at levels below 1, the average number of visited leaves in good subtrees does not depend on the subtree position, that is, $S_j^i(k) = S(k)$, $k < d-1$, and (ii) probabilities of goal existence in subtree j at second level inside subtree i at first level, γ_j^i, remain constant at any level of subtree i. Approximation (i) may introduce errors on the order of $S(d-2)$ (that is, $O(\frac{b^{d-1}}{b-1})$), which are not very important in $S(d)$, on the order of $O(\frac{b^{d+1}}{b-1})$). Approximation (ii) assumes that Γ^i remains constant for every node at every level inside subtree i, which is not true in general. The exact value of (3) inside subtree i varies at each expanded node, and it can be higher or lower than Γ^i, so we take the value computed at the root of subtree i as an estimation of the exact values. From approximation (i), it is easy to see that $\hat{S}_i(k) = S(k)$, $k < d-1$. Using (1) to compute $S(d-2), S(d-3), \dots, S(1)$, we get,

$$S_i(d-1) = \textstyle\sum_{j=0}^{d-2} F(j)\,\Gamma^i = \frac{b^{d-1}-1}{b-1}\,\Gamma^i$$

Therefore,

$$I(d) = \tfrac{b^d-b}{b-1}\ min_i\{\Gamma^i\} \approx \tfrac{b}{b-1} b^{d-1}\ min_i\{\Gamma^i\}$$

IDFS outperforms DFS when $I(d) < S(d)$, that is,

$$min_i\{\Gamma^i\} < \tfrac{b-1}{b}\Gamma + \tfrac{b-1}{b^d}\hat{S}(d-1)$$

For medium b and d the second summand can be eliminated. So IDFS outperforms DFS when,

$$min_i\{\Gamma^i\} < \tfrac{b-1}{b}\Gamma \tag{4}$$

Condition (4) is given in terms of probabilities which depend on the quality of the heuristic used to order successors. In fact, there is a *inverse relation* between heuristic quality and the value of Γ. The better the heuristic is, the lower the number of mistakes it causes in which DFS will fall. In the best case, DFS will not fall into any mistake when $\gamma_1 = 1$, which corresponts to the minimum of (3). The probability of DFS falling into mistakes increases as Γ increases. Γ is function of probabilities computed at level 1, while Γ^i is function of probabilities computed at level 2, of which (4) takes the *best* one among subtrees of level 1. In heuristic terms, Γ

depends on the quality of the heuristic at level 1, while T^1 depends on the best quality of the heuristic at level 2. Given a problem and a heuristic, 1DFS outperforms DFS if there is a significant difference of heuristic quality between level 1 and level 2, enough to satisfy expression (4). Observe that the best T^i should be better than T, to overcome the factor $\frac{b-1}{b}$, Otherwise, if heuristic quality is similar in both levels, DFS is the algorithm of choice.

In general, heuristic quality improves with tree depth. This quality improvement increases with problem difficulty. On easy problems, heuristics are able to give good advice at shallow levels, leaving no room for significant improvement as they move to deeper levels. On hard problems, heuristic advice is not very good at shallow levels, and it improves as it goes into deeper levels. Hard problems are good candidates to satisfy condition (4), for which we believe that IDFS is a suitable algorithm.

In the previous analysis we assume that all subtrees of first level are active. If we suppose that a goal exists in the first *k* subtrees, we take these subtrees as active and the whole analysis can be repeated. Expressions (1) (2) and (3) are truncated at the k-th subtree (instead of including contributions until the 6-th subtree), expression (4) remains invariant and the discussion about heuristic quality is restricted to the first *k* subtrees.

5 Experimental Results

A binary CSP is defined by a finite set of variables taking values on finite domains under a set of binary constraints. CSPs are search tree problems with fixed depth —the number of variables— for which several ordering heuristics exist, so they are adequate to test IDFS performance. We have taken *forward checking (FC),* a DFS based algorithm, substituting its DFS structure by IDFS, obtaining FC_{idfs}. We have tested *FC* and FC_{idfs} on a variety of random CSP instances. A random binary CSP [Prosser, 1996] is defined by $\langle n, m, p_1, p_2 \rangle$, where *n* is the number of variables, *m* is the number of values for each variable, *p* is the proportion of existing constraints, and •*pi* is the proportion of forbidden value pairs between two constrained variables. Constrained variables and their forbidden value pairs are randomly selected.

In CSPs, heuristics to order successors are value ordering heuristics. We have used the *highest support* heuristic for value ordering, and the *lowest support* heuristic for variable ordering [Larrosa and Meseguer, 1995] [2]. They are computed incrementally, and the consistency checks performed in their computation are included in the results.

We have tested *FC* and FC_{idfs} (both using the same heuristics) on difficult solvable instances in the left side

[2] In that paper, both heuristics were considered under the generic name of "lowest support". Current names represent more faithfully the heuristic criterion behind each one.

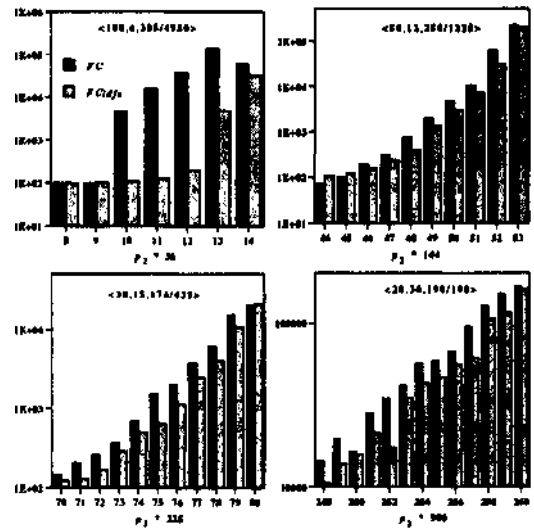

Figure 4: Visited nodes by *FC* and FC_{idfs} for four random CSP classes

of the peak of computational effort (inside the "mushy region" [1] [Smith and Dyer, 1996], where solvable and unsolvable instances coexist), for the following problem classes ((n, *rn,pi)),*

$$\text{(a)}\quad \langle 100, 6, 305/4950 \rangle \qquad \text{(b)}\quad \langle 50, 12, 250/1225 \rangle$$
$$\text{(c)}\quad \langle 30, 15, 174/435 \rangle \qquad \text{(d)}\quad \langle 20, 30, 190/190 \rangle$$

where constraint density varies from very low (a) to totally connected (d). For all experiments, FC_{idfs} takes level 1 as the only parallel level, with 2 active subtrees for class (a), 4 active subtreees for classes (b) and (c), and 9 active subtrees for class (d). Results are contained in Figures 4, and 5, with p_2 as the varying parameter. Each point is the average over 100 solvable instances. Regarding the number of visited nodes (Figure 4), on the left side of the plots problems are relatively easy, and both *FC* and FC_{idfs} visit a similar number of nodes. Going to the right, problems become more difficult and the number of visited nodes increases, but the increment for *FC* is much higher than the increment for FC_{idfs} (note the logarithmic scale). In this part, FC_{idfs} outperforms *FC* in a factor varying from 2 to 100. The rightmost point corresponds to the peak (or the point previous to the peak when the density of solvable problems at the peak is too low). At this point problems are very difficult and the number of visited nodes is similar for both algorithms. Regarding the number of consistency checks (Figure 5), we observe a similar picture with a linear scale. On easy problems at the left, both algorithms perform a similar number of checks. When problems become harder, the number of consistency checks increases, but *FC* performs more consistency checks than FC_{idfs}. In this part, FC_{idfs} outperforms *FC* in a factor from 1.1

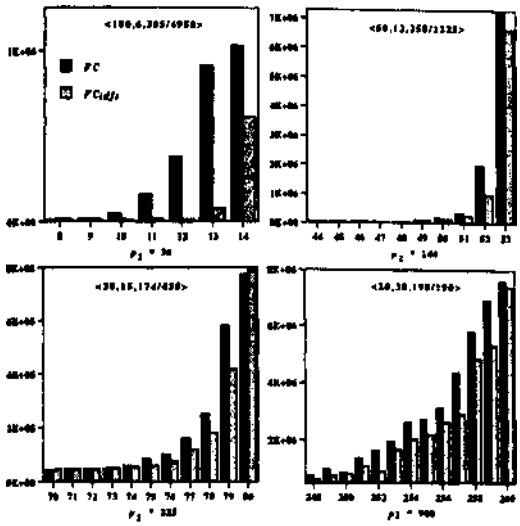

Figure 5: Consistency checks performed by *FC* and FC_{idfs} for four random CSP classes.

to 2. For very difficult problems at the rightmost point, both algorithms perform a similar number of consistency checks. It is worth noting that FC_{idfs} always performs better than or approximately equal to FC, but never significantly worse.

These results can be interpreted in full agreement with the efficiency analysis of Section 4. For easy problems, the heuristic is good enough at the first level of the tree to provide almost always the good advice. Mistakes are very rare so 1DFS provides no advantages over DFS. When problems become harder, more and more mistakes occur. For these problems, the heuristic can improve when it is computed at the second level of the tree, with respect to the same heuristic computed at the first level. It seems to do so because IDFS outperforms DFS on these problems, in both number of visited nodes and consistency checks. For the hardest problems, going down a single level in the tree is not enough to improve significantly the heuristic quality. This is in agreement with the experimental results, since IDFS and DFS behave similarly on the hardest instances.

6 Open Issues and Conclusions

Experimental results confirm the efectiveness of IDFS over DFS on hard solvable problems. However, some important questions to use IDFS as a general search procedure remains to be answered, such as the depth of parallel levels in the search tree, their distribution (consecutive or not) and their number. Related to this is the number of active subtrees in each parallel level, which could vary from one subtree to another inside the same parallel level. Dynamic selection of these parameters, to adjust the procedure to specific problem characteristics, can also be devised. Besides, given the close relation between IDFS benefits and heuristics, further knowledge about the variation of heuristic quality with search depth is needed. All these issues remain to be investigated in the future.

In [Korf, 1996), Korf concludes: "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." We have conceived IDFS with this aim, trying to keep the right balance between exploration of the search tree and exploitation of heuristic advice, inside a complete algorithm. As in the case of LDS and ILDS, IDFS is a new search strategy which outperforms classical ones on several problem classes, demonstrating how imaginative ways of search can effectively improve the current state of the art in search procedures.

Acknowledgments

I thank Lluis Godo, Javier Larrosa, Carme Torras and the anonimous reviewers for their constructive comments. I thank Josep Lluis Arcos for Latex support. I also thank Romero Donlo for her collaboration on the preparation of this work.

References

[Harvey and Ginsberg, 1995] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proc. of the 14th Int. Joint Conf. on Artificial Intelligence,* pages 607-613, Montreal, Canada, 1995.

[Korf, 1996] Richard E. Korf. Improved limited discrepancy search. In *Proc. of the 13th National Conf on Artificial Intelligence,* pages 286-291, Portland, Oregon, 1996.

[Larrosa and Meseguer, 1995] Javier Larrosa and Pedro Meseguer. Optimization-based heuristics for maximal constraint satisfaction. In *Proc. of the 1th Int. Conf. on Principles and Practice of Constraint Programming,* pages 103-120, Cassis, France, 1995.

[Prosser, 1996] Patrick Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence,* 81:81-109, 1996.

[Rao and Kumar, 1993] V. N. Rao and Vipin Kumar. On the efficiency of parallel backtracking. *IEEE Transactions on Parallel and Distributed Systems,* 4(2):427-437, 1993.

[Smith and Dyer, 1996] Barbara M. Smith and M. E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence,* 81:155-181, 1996.