Diverse Depth-First Search in Satisificing Planning

Akihiro Kishimoto
Tokyo Institute of Technology

Rong Zhou
Palo Alto Research Center

Tatsuya Imai
Tokyo Institute of Technology

Introduction

In satisficing planning where suboptimal plans are accepted, many planners use greedy best-first search (GBFS). Despite recent advances in automatic heuristic function generation, GBFS often suffers from performance degradation caused by inaccurate state evaluations. Diverse best-first search (DBFS) (Imai and Kishimoto 2011) avoids plateaus of search due to such inaccuracies by occasionally selecting states to expand that appear unpromising. Imai and Kishimoto showed that this approach outperforms the Fast Downward planner (Helmert 2006) with the best configurations based on GBFS for satisficing planning.

Although DBFS has been shown to be effective, many hard planning problems remain unsolved. One drawback of DBFS is memory, beause DBFS is a best-first search algorithm and as such it must save all the open and closed states in memory, which can severely limit the scalability of DBFS.

The Diverse Depth-First Search Algorithm

Our new Diverse Depth-First Search (DDFS) algorithm integrates the best features of DBFS and Korf's recursive best-first search (RBFS) algorithm (Korf 1993). As a depth-first search algorithm, DDFS uses far less memory than DBFS. But like DBFS, DDFS probabilistically selects a node to expand with a non-minimum h-value, in order to avoid search plateaus caused by inaccurate heuristic estimates.

While DBFS has to maintain the open and closed lists, DDFS only needs a stack to expand nodes in depth-first order. DDFS uses as a subroutine a greedy-search version of Korf's RBFS algorithm, which we call *Greedy Recursive Best-First Search* (GRBFS). DDFS controls the "greediness" of GRBFS by limiting the number of leaf node expansions in GRBFS.

DDFS consists of two alternating search modes, *greedy* mode and *diverse* mode. In greedy search mode, DDFS selects a node to expand with the minimum h-value as in GRBFS. The amount of search performed per greedy (or diverse) search is limited by l_{max}, the maximum number of leaf nodes to be expanded in a single run of GRBFS. A similar limit, called node-expansion quota, is used in DBFS. The reason only *leaf* node expansions are counted toward DDFS' expansion quota is to ensure enough work is performed to cover new search grounds instead of simply re-generating previously explored regions.

When DDFS switches from its greedy search mode to diverse mode, it selects a node that may have a higher h-value than the minimum. The selection of such a node is controlled by parameters P and T, which are introduced in (Imai and Kishimoto 2011). P determines the maximum depth d used to probabilistically select the stack depth to which DDFS backtracks from the current depth. Historically, GBFS variants tend to search deeply by greedily expanding nodes with small h-values. To encourage diversity, DDFS is designed to periodically visit regions of the search space that are closer to the root node and that have not been explored enough to avoid search plateaus. DDFS backtracks to a node n at depth d with probability $p[d]$, which is a function based on the sum of $T^{h(s)}$, where s is n's successor and $h(s)$ is the h-value of s or its h-value saved in the transposition table (to be described later). $T^{h(s)}$ is used to strike a balance between exploitation and exploration, according to the heuristic function. Like DBFS, DDFS assigns a smaller exploration probability to a successor s with a larger $h(s)$. After a backtrack node is determined, one of its successor s is chosen to start a new round of GRBFS.

GRBFS uses the *h-threshold*, a technique adapted from the original RBFS, to control the amount of exploration as follows. Let n.th be the h-threshold of node n. GRBFS continues exploring the search space rooted at n, if the h-values of the leaf nodes do not exceed n.th and GRBFS has not expanded more than l_{max} leaf nodes. The successor $best$ with the minimum h-value (more precisely, the minimum backed-up h-value) is selected and $best$.th is updated to be the minimum h-value of all the descendants of $best$. GRBFS stops exploring the subtree rooted at $best$ as soon as the minimum h-value node changes from $best$ to either another successor of n or another successor of n's ancestor. This technique was invented by Korf in his original RBFS algorithm, except that here we use only the (backed-up) h-value instead of the (backed-up) f-value.

Unlike RBFS, GRBFS is enhanced with a transposition table (TT) that is a large cache that maps a state to its corresponding hash table entry. TTs are designed to improve search efficiency by storing previous search effort, as explained in (Reinefeld and Marsland 1994). More specifically, the TT entry for node n keeps the minimum h-value of all the leaf nodes in the subtree rooted at n. TT requires only a fixed size of memory and plays an essential role in reducing (1) duplicate search effort caused by transpositions in the search space and (2) useless re-expansions of internal nodes due to the iterative-deepening nature of GRBFS.

While a number of tie-breaking strategies can be used when GRBFS finds more than one successor with the minimum h-value, our implementation randomly selects one of the best successors to diversify the greedy search.

Preferred operators (aka helpful actions) (Hoffmann and Nebel 2001; Helmert 2006) are incorporated into our DDFS implementation using a dedicated stack. Furthermore, a separate transposition table is used to explore the search space induced by the preferred operators. As in (Imai and Kishimoto 2011), when DDFS with preferred operators performs the search, a node is chosen randomly from one of the two stacks (i.e., one for all the applicable operators and the other for only the preferred operators). A search starting from the chosen node is then performed until a leaf node is expanded. These steps are repeated until DDFS finds a plan or proves that no solution exists.

Experimental Results

The performance of DDFS and DBFS was evaluated by running experiments on 1,932 planning instances in 48 domains from the International Planning Competitions 1-7 on a dual quad-core 2.33 GHz Xeon E5410 machine with 6 MB L2 cache with 16 GB memory. A single core was used in all the experiments. Algorithms are implemented on top of Fast Downward with the FF heuristic, which is the best configuration in (Imai and Kishimoto 2011).

We used the best available parameters P and T for DBFS and DDFS by running a number of preliminary experiments. DBFS used $T = 0.5$ and $P = 0.1$; whereas DDFS used $T = 0.99$ and $P = 0.1$.

The size of the TT was set to 0.5 GB (approximately 13.1 million TT entries) for each DBFS search (i.e., one generating only preferred operators and the other with all applicable operators), resulting in a total TT size of 1 GB memory. Additionally, a hash table preserving preferred operators used at most about 0.6 GB memory.

We ran the experiments with time limits of 30 minutes and 4 hours per planning instance, respectively, with a memory limit of 2 GB using the best available seeds obtained by preliminary experiments. We observed that both DDFS and DBFS were robust to different seeds, as shown in (Imai and Kishimoto 2011). The difference in the number of solved problems between the best and worst seeds was 21 for DDFS and 22 for DBFS. The number was reduced to 13 for DDFS and 10 for DBFS, if only the difference between the best and average seeds was considered.

Table shows the number of solved instances by DBFS and DDFS. The number inside the parentheses in the DBFS column indicates the number of instances DBFS was unable to solve because it used up the 2 GB memory. On the other hand, the number inside the parentheses in the DDFS column is the number of such instances DDFS solved while DBFS ran out of memory.

	30 minutes		4 hours	
	DBFS	**DDFS**	**DBFS**	**DDFS**
Total (1,932)	1,740 (115)	**1,755** (24)	1,762 (136)	**1,800** (40)

Table 1: The number of instances solved by DBFS and DDFS.

The table shows that DDFS outperforms DBFS in the number of solved instances. With a time limit of 30 minutes, DBFS exhausted memory when trying to solve 115 instances. Of these 115 instances, DDFS solved 24. This highlights the linear-space advantage of depth-first search, which gives DDFS a significant scalability boost over DBFS in solving hard planning problems. Moreover, when the time limit was set to four hours, the scalability advantage of DDFS can be observed more clearly: DDFS solved 47 instances DBFS could not solve. Of the 47 instances, DBFS ran out of memory in trying to solve 40 instances. Nine instances were solved only by DBFS and the remaining 1,753 instances were solved by both algorithms. In solving four instances, DBFS required 1.6–2GB memory, which is larger than the preset memory size for the tables needed by DDFS. This indicates that the additional 40 instances that only DDFS was able to solve are the difficult ones in the test suite.

We observed that DDFS tends to expand more nodes than DBFS. This is not surprising because DDFS often needs to re-expand interior nodes due to its iterative-deepening characteristics. However, their difference in the number of node evaluations was much smaller than the number of node expansions. Since state evaluation is by far the most computationally expensive part in satisficing planning, DDFS' node reexpansion overhead did not have a significant impact on its performance.

Traditionally, greedy search based on depth-first strategy tends to find plans of lesser quality than their best-first counterparts. Interestingly, this is not the case for DDFS. In fact, the average plan quality for DDFS is 0.2 *% better* than that of DBFS. This underscores the importance of search diversification in depth-first search, which improves not only problem coverage but also the quality of the plan.

References

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Imai, T., and Kishimoto, A. 2011. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In *Proceedings of AAAI 2011*, 985–991.

Korf, R. E. 1993. Linear-space best-first search. *Artificial Intelligence* 62(1):41–78.

Reinefeld, A., and Marsland, T. 1994. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16(7):701–710.