Band Search: an Efficient Alternative to Guided Depth-first Search

Lon-Chan Chu and Benjamin W. Wah
University of Illinois at Urbana-Champaign

Abstract

In this paper, we propose a novel search algorithm called **band search** *that generalizes guided depth-first and best-first searches. The search allocates a band of at most W nodes in each level of the search tree for storing active nodes in that level, D priority lists, one for each level, for storing overflow nodes, and D counters, one for each level, for keeping track of and to limit the degree of backtracking allowed. The algorithm has three major features: a) it selects for expansion in a best-first fashion from all nodes in the bands, b) it moves nodes from an overflow list into the corresponding band in a depth-first fashion, and c) it restricts backtracking so that at most W nodes in a level are fully searched before allowing new nodes in this level into the band. The algorithm specializes to be a guided depth-first search (GDFS) when W is one, and a best-first search (BFS) when W is unlimited. By choosing W appropriately, we show empirically that the algorithm can often outperform GDFS and has performance close to that of BFS. Moreover, the algorithm can be used instead of GDFS in iterative searches, such as IDA*, MIDA*, and DFS*. Finally, we identify conditions upon which the algorithm behaves like BFS, and the anomalous behavior when the algorithm performs worse than GDFS.*

Keywords and phrases: Band search, best-first search, guided depth-first search, and iterative search.

1: Introduction

Search, an important and powerful method for solving problems in artificial intelligence and operations research, has been studied extensively. Simple search algorithms often used are best-first search (BFS), depth-first search (DFS), and guided depth-first search (GDFS). BFS expands the least number of nodes for finding the optimal solution, but requires a huge amount of memory that grows exponentially with the problem size. As a result, BFS is only practical for solving small problems. DFS requires memory proportional to the depth of the search tree; however, it may get trapped deep in the search tree. GDFS is a variant of DFS that expands in a depth-first fashion the node with the minimum lower bound.[1] Since it generally outperforms a pure DFS, we consider it instead of DFS in this paper.

More efficient search algorithms have been designed using GDFS as a building block. These include IDA* [3], dynamic-control IDA* (MIDA*) [9], IDA_CR [7], DFS* [8], and iterative refining A* (IRA*) [5]. All these algorithms are iterative and search progressively deeper in the search tree using multiple GDFSs. They have the advantage that they use limited memory, and that the number of nodes expanded is close to that of BFSs.

Instead of searching deeper in successive iterations, *iterative broadening* [1] increases the number of branches searched in each node progressively until the entire tree is searched. This algorithm is suitable for searching trees with large branching degree and for solving decision problems. However, it is not suitable for optimization problems considered in this paper, since it stops whenever it finds a solution.

Band search studied in this paper allocates a band of at most W nodes in each level for storing active nodes in that level, selects for expansion in a best-first fashion from nodes in all the bands in memory, and inserts nodes into the band in a depth-first fashion using nodes expanded previously and stored outside the band. W in this search is called the *bandwidth* of the search. The algorithm is a hybrid between GDFS, where W is one, and BFS, where W is unlimited. The advantage of the algorithm is that it uses a linear amount of space $bigO(D \times W)$, where D is the maximum depth of the search tree. The algorithm, however, inherits the problem associated with GDFSs in that it may be trapped deep in a search tree. Hence, it is efficient for solving problems that do not have deep search trees, such as symmetric traveling-salesperson, production-planning, knapsack, and vertex-cover problems. It should not be

Authors' address: Center for Reliable and High Performance Computing, Coordinated Science Laboratory, 1308 West Main Street, MC255, Urbana, Illinois 61801; E-mail: {l-chu1, b–wah}@uiuc. edu.

Research partially supported by National Aeronautics and Space Administration under contracts NCC 2-481 and NAG 1-613 and by National Science Foundation under grant MIPS 88-10584.

Proceedings of 4th International Conference on Tools with Artificial Intelligence, November 10-13, 1992.

1. The lower-bound values are used by a GDFS to select the node in a level for expansion in a minimization search; for a maximization search, the upper-bound values are used instead.

applied to problems with deep search trees, such as puzzle and maze problems. Instead, for these problems, it can be used to replace GDFS in iterative searches, such as IDA*, MIDA*, and DFS*.

An interesting search algorithm that simulates BFS using a bounded amount of space, $bigO(D^{m+1})$, is Ibaraki's depth-m search [2], where D is the maximum depth of the search tree and m is similar to the bandwidth in band search. Its major drawback is that this bound is too large since it is exponential in m.

In this paper, we study the performance of band search and evaluates it using six benchmarks. Section 2 describes the notation used. In Section 3, we present the band-search algorithm and its properties, identify the conditions when band search behaves like BFS, and discuss possible anomalous behavior when W is increased. In Section 4, we present the classes of search problems for which band search is applicable, and evaluate the performance of band search on these problems.

2: Notation

In this paper, we focus on state-space tree searches that repeatedly decompose a problem into smaller subproblems until either a solution is found or infeasibility is proved. Each subproblem is represented by a node in the search tree. Without loss of generality, we consider minimization problems in our discussion. Maximization problems can be transformed into minimization ones by negating their objectives.

A node i in the search tree is associated with an upper bound u_i and a lower bound ι_i. Let x_i be the best solution in the subtree rooted at node i. Hence, $\iota_i \le x_i \le u_i$. In a monotone admissible heuristic search [6], ι_i satisfies the following properties: (1) $\iota_i \le x_i$; (2) $\iota_i = x_i$ if node i is a leaf solution, and (3) $\iota_i \le \iota_j$ if node i is an ancestor of node j. The best solution found so far during the course of search is called the *incumbent* and is denoted as z. Any node i with $\iota_i \ge z$ is obviously inferior and can be pruned.

Nodes in a search tree can be classified into three types. Search nodes with lower bounds less than the optimal-solution value are called *essential nodes*; these nodes must all be expanded when the search terminates in order to prove optimality of the solution found. Nodes with lower bounds larger than the optimal-solution value are called *non-essential nodes*; these nodes do not have to be expanded in order to prove optimality. Nodes with lower bounds equal to the optimal-solution value are called *semi-essential nodes*; depending on the order that nodes are searched, these nodes may or may not be expanded when the optimal solution is found.

A search algorithm is *optimal* if it does not expand any semi-essential or non-essential nodes before finding the optimal solution. One such algorithms is *OPTD*, which knows ahead of time the optimal-solution value and searches in a depth-first fashion, assuming all lower bounds are unique. It is optimal because it expands only essential nodes and uses a limited amount of memory bounded by the depth of the search tree. The algorithm is useful only as a benchmark, as it is not feasible to know the optimal-solution value before the search starts.

3: Band Search

Band search is designed to take the best features of both BFS and GDFS. It is characterized by a band of at most W active nodes in each level, where nodes inside the bands are selected for expansion in a best-first fashion. We denote band search as BS(W), where W is its bandwidth. In this paper, we only consider the case where W is fixed during the search, leaving the study of the dynamic case to future work. In this section, we describe the band-search algorithm, study its properties, and identify the conditions for optimality and anomaly. We focus only on tree searches, although the algorithm can be adapted to graph searches as well.

3.1: Algorithm

Band search allocates a) B_i, a band of at most W nodes in level i of the search tree for storing active nodes in level i; b) F_i, a priority list in level i of the search tree for storing overflow nodes that cannot fit in B_i; and c) C_i, a counter in level i for keeping track of and for limiting the degree of backtracking allowed in level i. Each counter acts like a quota that allows at most W nodes in this level to be searched completely before allowing new nodes in this level or above[2] into the band. Note that the counters can only be incremented or reset to zero.

Initially, all counters are reset to zeroes, the root of the search tree is placed in B_0, C_0 is incremented to one, and the incumbent z is at ∞ (assuming a minimization problem). The algorithm has five steps.

a) *Selection Phase*. It selects for expansion in a best-first fashion from all the nodes in the bands. Assume that this node is in level $d-1$.

b) *Expansion Phase*. It deletes the node selected in step (a) from B_{d-1} and expands the node into descendents

2. In our discussion, the search tree is oriented with the root at the top. A node is above another node when the former is closer to the root than the latter. It also means that the level number for the former is smaller than that of the latter. We also assume that the root is at level 0.

in level d. Note that C_{d-1} is not decremented when a node in level $d-1$ is expanded.

c) *Pruning Phase*. A descendent from the node expanded, if any, is pruned when its heuristic value is worse than the incumbent z. If the descendent is a feasible solution and its value is better than the incumbent, then the incumbent z is updated, and nodes in the bands and the overflow lists with heuristic values worse than the new incumbent are pruned. The counters are not changed in this case.

d) *Insertion Phase*. If C_d is less than W, descendent not pruned in step (b) are inserted into B_d, and C_d is incremented. If C_d is already at W, then the algorithm checks whether this descendent has a lower bound better than any active node in B_d. If there is, then the active node in B_d with the worst heuristic value is deleted from B_d and inserted into F_d, the overflow list in level d, and the descendent is inserted into B_d. In other cases, the descendent is inserted into F_d. This step is repeated for all descendents not pruned in step (c).

e) *Reorganization Phase*. If the band is not empty, then it goes to step (a). If the band is empty and there are non-empty overflow lists, then it first resets all counters, finds the deepest non-empty overflow list (say, in level d_{ne}), moves W nodes from $F_{d_{ne}}$ (or all nodes if $F_{d_{ne}}$ has less than W nodes) into $B_{d_{ne}}$, and sets $C_{d_{ne}}$ to the number of nodes moved before going to step (a). Note that only nodes in level d_{ne} are moved, although there may be non-empty overflow lists in levels above d_{ne}. Similar to backtracking in GDFS, band search only backtracks to the deepest level with a non-empty overflow list. This step is necessary for bounding the number of nodes in the overflow lists. If the bands and all the overflow lists are empty, then the algorithm terminates with the incumbent found.

The algorithm requires nodes in all the bands as well as nodes in each individual band to be indexed by their heuristic values. On the other hand, nodes in each overflow list are indexed by their heuristic values only.

The algorithm resembles a beam search [10] which focuses on a band (or beam) of nodes in each level. The major difference, however, is that a beam search traverses nodes in a breadth-first fashion, whereas a band search indices nodes in all the bands by their heuristic values and traverses them in a best-first fashion.

To illustrate band search of bandwidth W, we show an example of BS(2) for exploring the search tree shown in Figure 3.1. The feasible-solution nodes of the search tree are doubly circled, and the intermediate nodes are represented by single circles. Numbers inside the circles represent both the node numbers and the corresponding lower-bound or feasible-solution values. The optimal-solution value is 11.

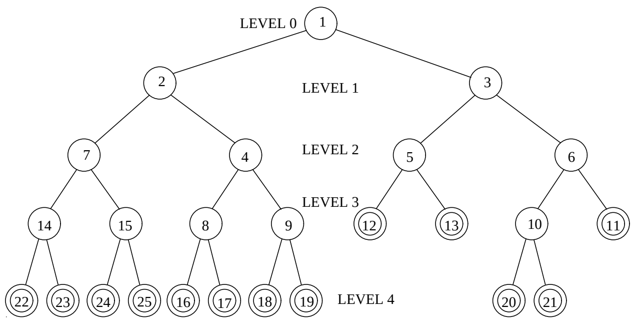

Figure 3.1. A search tree for illustrating the operations of the band search.

Table 3.1. Trace of BS(2) for the search tree in Figure 3.1.

ITER.	1	2	3	4	5	6	7	8	9	10
STEPS USED	a,b d	a,b d	a,b d	a,b d	a,b c	a,b c	a,b c,e	a,b c,d	a,b c	a,b c,e
B_0	$\cancel{1}$									
F_0										
C_0	1	1	1	1	1	1	$\cancel{1}$ 0	0	0	0
B_1		$\cancel{2}$, 3	$\cancel{3}$							
F_1										
C_1	0	2	2	2	2	2	$\cancel{2}$ 0	0	0	0
B_2			4, 7	$\cancel{4}$, 5	$\cancel{5}$		6, 7	$\cancel{6}$, 7	$\cancel{7}$	
F_2				6, 7	6, 7	6, 7	$\cancel{6}$, $\cancel{7}$			
C_2	0	0	2	2	2	2	$\cancel{2}$ 2	2	2	2
B_3					8, 9	$\cancel{8}$, 9	$\cancel{9}$		10	$\cancel{10}$
F_3										
C_3	0	0	0	0	2	2	$\cancel{2}$ 0	1	1	1
z	∞	∞	∞	∞	∞	12	12	12	11	11

The steps involved in the example are shown in Table 3.1. There are 10 iterations of the algorithm. Initially, the root is expanded into nodes 2 and 3; both of which are included in B_1, resulting in C_1 to be 2. This is the starting point for iteration 2, which chooses node 2 for expansion, as this is the node with the minimum lower bound. In iteration 3, expanding node 3 results in nodes 5 and 6 in level 2. Since node 5 is better than node 7 in B_2, node 7 is moved from B_2 to F_2, and node 5 is inserted into B_2. Node 6 is worse than all the nodes in B_2 and is inserted into F_2. In iteration 5, node 5 is expanded, as this is the node with the minimum lower bound. This results in the update of the incumbent z from ∞ to 12. In iteration 7, all the bands are empty after node 9 is expanded. As a result, all the counters are reset, and at most 2 nodes in the deepest non-empty overflow list (nodes 6 and 7 in F_2) are moved to the corresponding band. C_2 is then incremented to 2. The steps carried out in the other iterations are straightforward. The expansion sequence for BS(2) is, therefore,

$$\text{BS(2)} \Rightarrow 1, 2, 3, 4, 5, 8, 9, 6, 7, 10.$$

For comparison, the expansion sequences for BFS and GDFS are, respectively,

$$\text{BFS} \Rightarrow 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,$$

$$\text{GDFS} \Rightarrow 1, 2, 4, 8, 9, 7, 14, 15, 3, 5, 6, 10.$$

In this example, BS(2) and BFS expand different sequences of nodes, but they are both optimal because they expand only essential nodes. Further, we observe that BS(2) performs better than GDFS.

3.2: Properties of Band Search

In this section, we study the properties of band search BS(W) and its memory requirement. The following theorem shows the limiting cases of BS(W). BS(1) is equivalent to GDFS, and BS(∞) is equivalent to BFS. When W is one, each band has only one node. Initially, the search progresses to the first leaf of the search tree, and siblings of the nodes expanded are placed in the corresponding overflow lists. At the instant when all the bands are empty, the node in the deepest overflow list is moved to the corresponding band, and the search progresses from then on as GDFS.
When W is unlimited, all search nodes generated can be kept in the bands in the corresponding levels, and all overflow lists are empty. Since nodes in the bands are expanded in a best-first fashion, the selection of active nodes for expansion is, therefore, best first. □

For some cases, band search behaves like BFS even when the bandwidth is limited. There exists a critical value W_{BFS} so that, for all $W \geq W_{BFS}$ and a given problem instance, BS(W) expands the same set of nodes as BFS (assuming all nodes have unique lower bounds). The sequence of nodes expanded, however, may be different, as illustrated in Figure 3.1.

The following lemma shows that band search BS(W) uses a bounded amount of memory, assuming a maximum branching degree b and a maximum depth D. For $1 \leq i \leq D$,

$$|\,B_i\,| \;\leq\; W \tag{3.1a}$$

$$|\,F_i\,| \;\leq\; W \times (b-1) \tag{3.1b}$$

The proof of Eq. (3.1a) follows from the definition of band search. To prove Eq. (3.1b), note that for level d, at most W active nodes can be included in B_d, and that no nodes in F_d can be moved to B_d until the subtrees rooted at the nodes in B_d are searched. The maximum number of nodes that can be generated by the parents of active nodes in B_d is bounded by $W \times b$. Since W of them are included in B_d, there are at most $W \times (b-1)$ active nodes in F_d. □

The following theorem says that the maximum space used by band search is bounded by the product of W, its bandwidth, D, the maximum depth, and b, the maximum branching degree. N_{active}, the total number of nodes in all the bands and overflow lists, is bounded as follows.

$$N_{active} \leq W \times D \times b. \tag{3.3}$$

From Lemma 3.1, we have

$$N_{active} \leq \sum_{i=1}^{D} (\,|\,B_i\,| + |\,F_i\,|\,)$$

$$\leq W \times D + W \times (b-1) \times D = W \times D \times b. \qquad \square$$

Hence, the memory used in a band search is at most W times the maximal amount needed in a GDFS.

3.3: Performance Analysis

Band search BS(W) in general takes less time to find the optimal solution when W is large, although anomalies may happen. In this section, we study the optimality conditions and illustrate the anomalous behavior of band search. The empirical evaluation of band search is shown in Section 4.

Define the *first band* as the set of nodes that are included in the bands before the bands become empty for the first time. Theorem 3.3 shows a sufficient condition of the optimality of band search in terms of the first band. If the optimal solution is generated by a node in the first band and that all lower bounds are unique, then band search is optimal in that it expands only essential nodes. Within the band, selection of active nodes for expansion is best first. As a result, when the optimal solution is generated by a node in the first band, then the incumbent will prevent any non-essential nodes from being expanded. Moreover, since all lower bounds are unique, there are no semi-essential nodes in the search tree. □

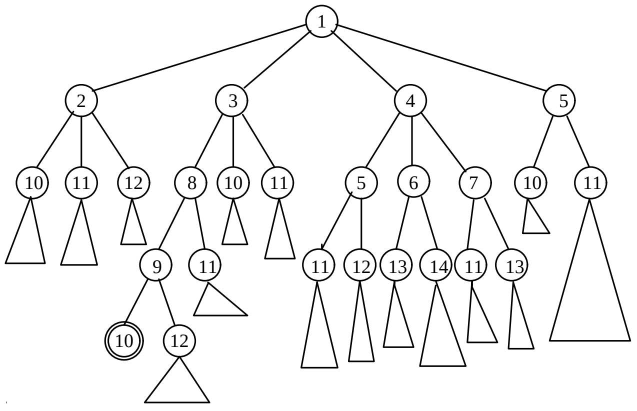

Figure 3.2. A search tree with bandwidth anomaly.

The following corollary show the necessary and sufficient conditions of the optimality of band search.

Corollary 3.1. *Band search is optimal if and only if there is at least one essential node in any of the bands at all time during the search, and all lower bounds are unique. The proof follows from the fact that only nodes in the bands are selected for expansion in a best-first fashion.* □

When the bandwidth W increases, band search generally expends less time and behaves like BFS. However, the improvement in performance is not monotonic with respect to increases in bandwidth. In order words, BS(W_1) may perform worse than BS(W_2) when $W_1 > W_2$. This phenomenon is called *bandwidth anomaly* and is illustrated in the following example. Consider the search tree shown in Figure 3.2, applying BS(2) and BS(3) results in the following expansion sequences.

$BS(2) \Rightarrow$ 1, 2, 3, 8, 9, 10, 4, 5, 5, 6, 7, stop.

$BS(3) \Rightarrow$ 1, 2, 3, 4, 5, 6, 7, 11, 11, 12, ..., get trapped.

BS(3) gets trapped in a large subtree; consequently, it expends more time than BS(2). It is easy to extend this example to show that BS(W_1) may perform worse than BS(W_2) for any arbitrary $W_1 > W_2$.

The reason for bandwidth anomaly is that a band search with a larger bandwidth has more nodes of similar quality, but not the same set of nodes as in BFS, to select for expansion. This increases the chance of selecting a node with a small lower-bound value, but leads the search to a large subtree with no solution. Similar anomalies have been discovered in sequential DFSs [4].

Fortunately, as empirically observed, bandwidth anomalies do not happen frequently, especially when the bandwidth is large and the lower bounds are generally consistent with the quality of the solutions found. Lower bounds are consistent between two nodes, *i* and *j*, in the search tree when the lower bound of *i* is smaller than that of *j*, and the best solution obtained by searching *i* is also small than that of *j*. Anomalies may happen in problems that have abundant inconsistent lower bounds, such as puzzle and maze problems. The following corollary states the necessary condition for bandwidth anomaly.

Corollary 3.2. *If bandwidth anomaly happens, than at some particular time the bands contain only non-essential or semi-essential nodes, and there remain essential nodes to be searched. The proof follows from the fact that the semi-essential and non-essential nodes selected for expansion do not lead to the optimal solution.* □

4: Experimental Results

In this section, we evaluate the performance of band search. The test problems used include 20-city symmetric traveling salesperson (sTSP), 80-object knapsack (KS), 18-plant production planning (PP), 40-vertex vertex cover (VC), 40×40 maze, and 15-puzzle problems. Due to space limitations, the details of these problems are not described here. These problems are classified according to whether there are polynomial-time algorithms for computing upper bounds and for finding feasible solutions. For a number of problems with polynomial-time heuristics, such as KS, sTSP, PP, and VC problems, GDFS has been reported to be the best search algorithm [8]. For these problems, we compare the performance of band search against that of GDFS. For problems without polynomial-time heuristics, we compare the performance of IDA* as well as DFS* between using GDFS and band search as its primitive search mechanism. Problems evaluated in this class include the 15-puzzle problem (of which the best algorithm reported was IDA* [3]), and the maze problem (of which the best algorithm reported was DFS* [8]).

In the following figures, we report the empirical behavior of band search for various benchmark problems. The scatter plots of the execution times for various bandwidths in solving 50 instances of sTSP, PP, KS, and VC are shown in Figure 4.1. The scatter plot of the execution times of DFS* employing band search of various bandwidths for solving 50 random instances of the 40×40 maze problem is shown in Figure 4.2. Also, Figure 4.3 shows the scatter plot of IDA* employing band search of various bandwidths for solving 50 instances of the 15-puzzle problem.

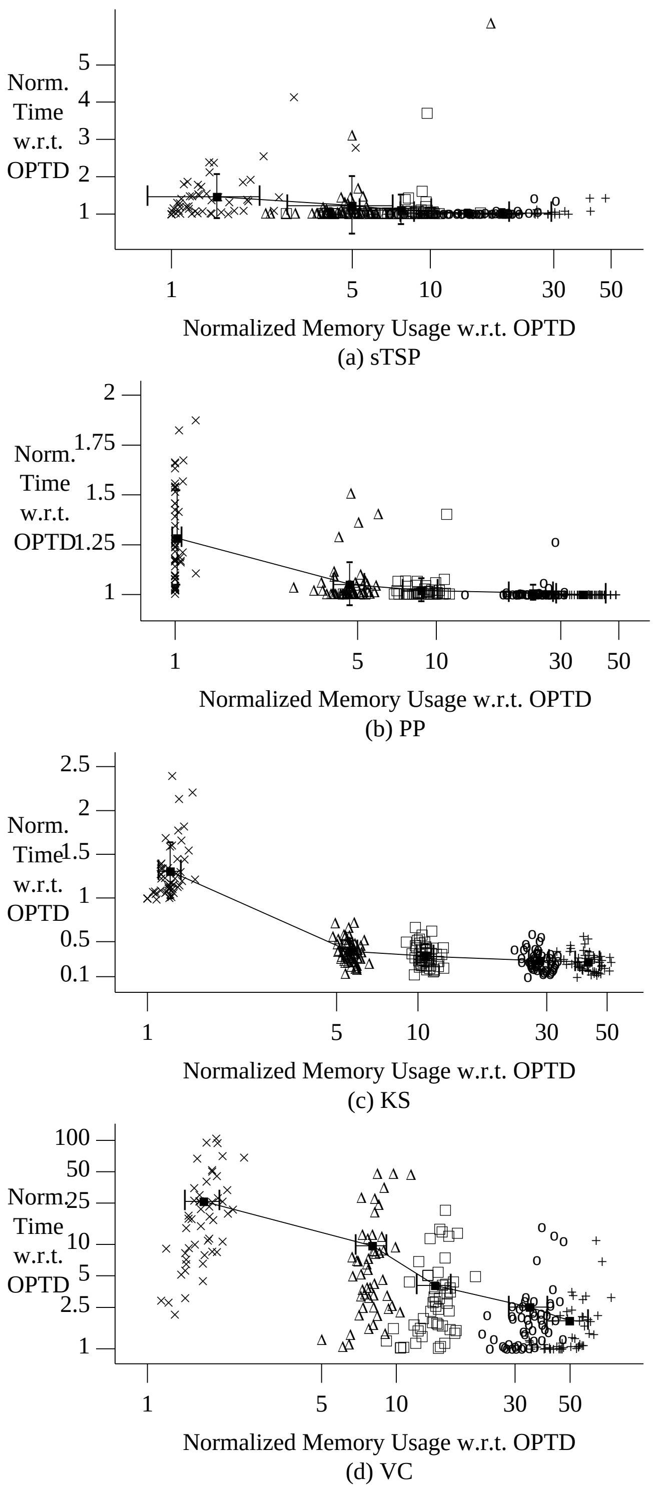

Figure 4.1. Scatter plots of execution times versus memory usages of band searches of various bandwidths for solving 50 instances of (a) sTSP, (b) PP, (c) KS, and (d) VC, where execution times and memory usages are normalized with respect to those of OPTD. [× for GDFS; Δ for BS(5); □ for BS(10); ○ for BS(30); + for BS(50)]

In these graphs, the execution time (*resp*., memory used) for each instance is normalized by the execution time (*resp*., memory used) of OPTD, since the raw data of different instances may vary over several orders of magnitude. Recall that OPTD is a benchmark search algorithm that knows the optimal-solution value before the search begins. Note that the axis representing memory used is logarithmic. In all these graphs, the averages of the normalized times are connected by a solid line, and the vertical and horizontal bars indicate the standard deviations around the averages.

The scatter plots of normalized speedups depicted in Figures 4.1 thru 4.3 are useful for showing the way that experimental data are clustered. However, the average values obtained from these scattered data may be misleading. An average value may be biased heavily since speedups (having values between 1 and ∞ with respect to OPTD) carry more weight in the average than slowdowns (having values less than 1 with respect to OPTD). The same problem happens when the raw data on execution time are used, where the execution times for different instances may vary by several orders of magnitude, and the average execution time is biased by a few instances with large execution times.

One way to cope with this problem of disproportionate weighting is to take one search algorithm as a reference (say, REF) and normalize individual execution times (say, for problem instance i) of the search algorithm being tested (say, S) so that the sum of normalized times of S and that of REF is 1. That is,

$$\tau_{S,i}^{S,REF} \triangleq \frac{T_{S,i}}{T_{S,i} + T_{REF,i}}, \tag{4.1a}$$

$$\tau_{REF,i}^{S,REF} \triangleq \frac{T_{REF,i}}{T_{S,i} + T_{REF,i}}, \tag{4.1b}$$

where $T_{S,i}$ and $T_{REF,i}$ are, respectively, the completion times of algorithms S and REF for problem instance i. The superscripts in τ in Eq. (4.1) indicate, respectively, the search algorithm under consideration and the reference search algorithm. The subscripts in τ in Eq. (4.1) indicate the search algorithm being normalized and the problem instance. The advantage of using this normalization method is that each problem instance carries equal weight in computing the final average value, as

$$\tau_{S,i}^{S,REF} + \tau_{REF,i}^{S,REF} = 1, \tag{4.2a}$$

$$\bar{\tau}_{S}^{S,REF} + \bar{\tau}_{REF}^{S,REF} = \frac{1}{n} \sum_{1=1}^{n} \left(\tau_{S,i}^{S,REF} + \tau_{REF,i}^{S,REF} \right) = 1, \tag{4.2b}$$

where $\bar{\tau}$ is the average normalized execution time, and n is the number of problem instances tested. Note that execution times normalized are always between 0 and 1.

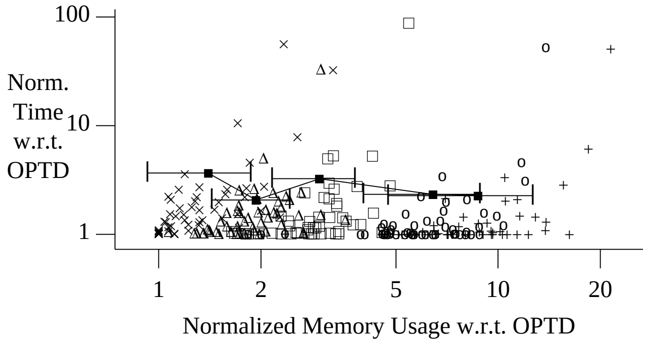

Figure 4.2. Scatter plots of execution times versus memory usages of DFS* employing band search as a primitive search for solving 50 random instances of the 40×40 maze problem, where execution times and memory usages are normalized with respect to those of OPTD; standard deviations of the normalized times are not shown in the graph, as they are relatively large. [× for DFS*; Δ for DFS*_BS(5); □ for DFS*_BS(10); ○ for DFS*_BS(30); + for DFS*_BS(50)]

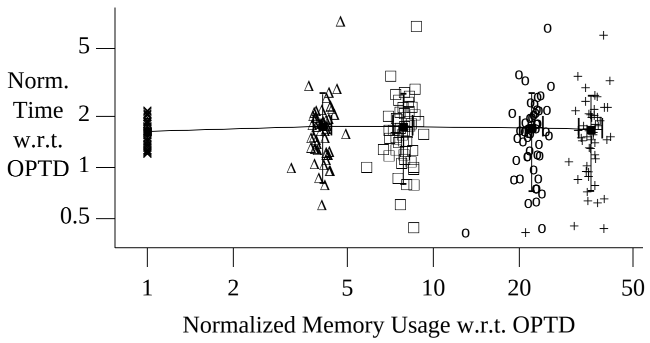

Figure 4.3. Scatter plots of execution times versus memory usages of IDA* employing band search as a primitive search for solving 50 random instances of the 15-puzzle problem where execution times and memory usages are normalized with respect to those of OPTD. [× for IDA*; Δ for IDA*_BS(5); □ for IDA*_BS(10); ○ for IDA*_BS(30); + for IDA*_BS(50)]

The ratio of the total sums of the normalized execution times based on the normalization method stated in Eq. (4.1) is, therefore,

$$P_S^{REF} \triangleq \sum_{i=1}^{n} \tau_{REF,i}^{S,REF} \Big/ \sum_{i=1}^{n} \tau_{S,i}^{S,REF} = \overline{\tau}_{REF}^{S,REF} \Big/ \overline{\tau}_{S}^{S,REF}. \quad (4.3)$$

The speedup of S with respect to REF based on raw execution times is

$$Q_S^{REF} \triangleq \frac{1}{n} \sum_{i=1}^{n} \frac{T_{REF,i}}{T_{S,i}} = \frac{1}{n} \sum_{i=1}^{n} \frac{\tau_{REF,i}^{S,REF}}{\tau_{S,i}^{S,REF}}. \quad (4.4)$$

Table 4.1 shows the average performance of those shown in Figures 4.1 thru 4.3. Note that the values

Table 4.1. Average performance of band search of various bandwidths for solving 50 random instances of each of the sTSP, PP, KS, VC, 40-by-40 maze, and 15-puzzle problems.

Problem	Algorithm S	$\overline{\tau}_S^{S,GDFS}$	P_S^{GDFS}	Q_{OPTD}^{S}	Q_S^{GDFS}
sTSP	GDFS	0.500	1.00	1.46	1.00
	BS(5)	0.453	1.21	1.22	1.28
	BS(10)	0.440	1.27	1.10	1.38
(20;1-50)	BS(30)	0.428	1.34	1.02	1.43
	BS(50)	0.428	1.34	1.02	1.43
PP	GDFS	0.500	1.00	1.28	1.00
	BS(5)	0.453	1.21	1.05	1.23
	BS(10)	0.447	1.24	1.02	1.26
(18;1-50)	BS(30)	0.444	1.25	1.01	1.27
	BS(50)	0.442	1.26	1.00	1.28
KS	GDFS	0.500	1.00	1.31	1.00
	BS(5)	0.232	3.31	0.39	3.81
	BS(10)	0.205	3.88	0.33	4.49
(80;1-50)	BS(30)	0.180	4.56	0.28	5.32
	BS(50)	0.172	4.81	0.27	5.63
VC	GDFS	0.500	1.00	25.93	1.00
	BS(5)	0.293	2.41	9.70	6.12
	BS(10)	0.187	4.35	4.05	11.55
(40;1-50)	BS(30)	0.135	6.41	2.51	15.58
	BS(50)	0.114	7.77	1.87	18.11
Problem	Algorithm S	$\overline{\tau}_S^{S,DFS*}$	P_S^{DFS*}	Q_S^{OPTD}	Q_S^{DFS*}
Maze	DFS*	0.500	1.00	3.73	1.00
	BS(5)	0.449	1.23	2.11	2.78
	BS(10)	0.454	1.20	3.29	3.25
(40;1-50)	BS(30)	0.423	1.36	2.36	3.39
	BS(50)	0.417	1.40	2.31	3.45
Problem	Algorithm S	$\overline{\tau}_S^{S,IDA*}$	P_S^{IDA*}	Q_S^{OPTD}	Q_S^{IDA*}
Puzzle	IDA*	0.500	1.00	1.63	1.00
	BS(5)	0.497	1.01	1.75	1.05
	BS(10)	0.493	1.03	1.74	1.10
(15;*)	BS(30)	0.482	1.07	1.71	1.17
	BS(50)	0.475	1.11	1.67	1.22

shown in Table 4.1 for Q_{BS}^{OPTD} are shown as lines in Figures 4.1 thru 4.3.

From the graphs in Figures 4.1 thru 4.3 and Table 4.1, we find that band search expands less nodes in general for finding optimal solutions, although bandwidth anomalies may happen. We notice that band search performs very well on the average for small bandwidths, and that its performance for large bandwidths is more stable, showing a *monotonic* improvement in performance and a smaller chance for bandwidth anomalies to occur. For the various problems evaluated, we have the following observations.

a) For problems where GDFS performs well, such as the sTSP and PP problems, band search performs better and has performance close to that of OPTD.

However, its improvement over GDFS is small because GDFS already performs very well for these problems.

b) For the KS problem, dominance pruning is applied so that node *i* dominates node *j* in the search tree when both nodes have considered the same set of objects, and the total profit (*resp*., total weight) of objects assigned for node *i* is larger (*resp*., smaller) than that of node *j*. We find that band search performs better than GDFS and even OPTD. The reason for this is that band search allows more nodes at any level to be considered for selection, allowing dominance relations to be applied more frequently.

c) For the VC problem, GDFS does not perform well, and band search improves the performance dramatically, while incurring limited overheads in memory usage. The deviations of normalized times are not shown in Figure 4.1d, as they are relatively large. Note that the time axis in Figure 4.1d is logarithmic.

d) From Figures 4.2 and 4.3 and the results in Table 4.1, we find that the improvements due to band search for the maze and the puzzle problems are less than those for the sTSP, KS, PP, and VC problems. This happens because successive searches, except the last one, in IDA* and DFS* employ pruning by thresholds, regardless of whether GDFS or band search is used. As a result, the performance difference between using band search versus GDFS is only apparent but is minor in the last iteration in IDA* and DFS*. Moreover, the thresholds used in the last iteration of IDA* and DFS* for solving the puzzle and the maze problems either are very close to the optimal solution or are exactly at the optimal solution. This reduces the benefit of band search, which keeps nodes in memory and expands them in a best-first fashion. Band search will work well when an accurate estimation of thresholds in the last iteration is difficult; this is the case when MIDA* [9] is applied for solving combinatorial optimization problems.

5: Conclusion

In this paper, we propose a novel search algorithm called band search, which is designed as an alternative to GDFS. We present the algorithm, study its properties, and illustrate its anomalous behavior. We show that the algorithm performs better than GDFS and that it can be used as an alternative to GDFS in iterative searches, such as IDA*, MIDA*, DFS*, and IDA_CR.

We have only studied band search with fixed bandwidth. An extension will be to change its bandwidth, either statically or dynamically, as a function of the level number, problem size, and memory size. Such an extension allows band search to adapt dynamically to memory available. For instance, given a memory constraint, band search can start as a BFS. When memory is below a given threshold, it reduces the bandwidth and continues the search. This process can be repeated until the optimal solution is solved. When more memory becomes available, the search process can resume with a larger bandwidth. We plan to study the performance of this extension in the future.

Acknowledgements

The second author would like to thank Prof. H. Tanaka and funding from Fujitsu Limited for their support at the University of Tokyo, upon which part of this paper was written.

References

[1] M. L. Ginsberg and W. D. Harvey, ‘‘Iterative Broadening,’’ *Proc. National Conf. on Artificial Intelligence*, pp. 216-220, AAAI, 1990.

[2] T. Ibaraki, ‘‘Depth-m Search in Branch-and-Bound Algorithms,’’ *Int’l. J. of Computer and Information Sciences*, vol. 7, no. 4, pp. 315-373, Plenum Press, 1978.

[3] R. E. Korf, ‘‘Depth-First Iterative Deepening: An Optimal Admissible Tree Search,’’ *Artificial Intelligence*, vol. 27, pp. 97-109, North-Holland, 1985.

[4] G. J. Li and B. W. Wah, ‘‘Computational Efficiency of Combinatorial OR-Tree Searches,’’ *IEEE Trans. on Software Engineering*, vol. 16, no. 1, pp. 13-31, IEEE, Jan. 1990.

[5] G.-J. Li and B. W. Wah, ‘‘Parallel Iterative Refining A*: An Efficient Search Scheme for Solving Combinatorial Optimization Problems,’’ *Proc. Int’l Conf. on Parallel Processing*, pp. 608-615, Pennsylvania State Univ. Press, University Park, PA, Aug. 1991.

[6] N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, McGraw-Hill, New York, NY, 1971.

[7] U. K. Sarkar, P. P. Chakrabarti, S. Ghose, and S. C. De Sarkar, ‘‘Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds,’’ *Artificial Intelligence*, vol. 50, pp. 207-221, Elsevier Science Publishers, 1991.

[8] N. Rao Vempaty, V. Kumar, and R. E. Korf, ‘‘Depth-First vs Best-First Search,’’ *Proc. National Conf. on Artificial Intelligence*, AAAI, Anaheim, CA, July 1991.

[9] B. W. Wah, *MIDA*: An IDA* Search with Dynamic Control,* Research Report CRHC-91-09, Center for Reliable and High Performance Computing, Coordinated Science Laboratory, Univ. of Illinois, Urbana, IL, April 1991.

[10] P. H. Winston, *Artificial Intelligence,* Second Edition, Addison-Wesley, Reading, MA, 1984.