SEQUENTIAL AND PARALLEL BRANCH-AND-BOUND SEARCH UNDER LIMITED-MEMORY CONSTRAINTS *

NIHAR R. MAHAPATRA† AND SHANTANU DUTT‡

Abstract. Branch-and-bound (B&B) best-first search (BFS) is a widely applicable method that requires the least number of node expansions to obtain optimal solutions to combinatorial optimization problems (COPs). However, for many problems of interest, its memory requirements are enormous and can far exceed the available memory capacity on most systems. To circumvent this problem, a number of limited-memory search methods have been proposed that are either based purely on depth-first search (DFS) or combine BFS with DFS. We survey and compare previous sequential and parallel limited-memory search methods, and discuss their suitability for solving different types of COPs. We also propose a new limited-memory search method, *iterative extrapolated-cost bounded search* (IES*), that performs a sequence of cost-bounded depth-first searches from the root node of the search space. In this method, cost bounds for successive iterations are set to an estimated optimal-solution cost obtained by extrapolating from search experience in previous iterations. We provide accurate and fast, approximate methods suitable for extrapolating the optimal-solution cost for lower-bound cost functions with a range of growth rates. Finally, we propose an efficient approach to parallelizing IES* that is applicable, with minor modifications, to other iterative cost-bounded DFS methods like IDA* and DFS*. An important feature of this approach is the asynchronous execution of the different iterations of IES* by processors to minimize idling. We provide a method for determining cost bounds independently in different processors; these cost bounds vary from processor to processor, and decrease from an initial larger value to the true cost bound for the iteration. Further, to minimize unnecessary node expansions that can occur because of the asynchronous operation and because of the initial loose upper bounds, we propose an efficient load balancing technique. This technique distributes work of earlier iterations with higher priority among processors. As a result, different processors are likely to execute IES* iterations that are as close to each other as possible, and also the individual cost bounds for the same IES* iteration in different processors approach the true cost bound rapidly. This decreases the possibility of unnecessary work in parallel IES*, thus leading to an efficient parallelization.

Key words. Branch-and-bound, best-first search, depth-first search, extrapolation, iterative search, least-square curve fitting, limited-memory search, load balancing, parallel branch-and-bound.

1. Introduction.

Branch-and-bound (B&B) is a widely used, general method for solving combinatorial optimization problems (COPs) [32]. Given a COP $\mathcal{P}$, B&B is used to find a least-cost solution to $\mathcal{P}$.[1] Its operation can be viewed as a search through a tree $\mathcal{T}$ of subproblems in which the original problem $\mathcal{P}$ occurs at the *root*—in this paper we restrict our atten-

*S. Dutt was supported by NSF grant MIP-9210049, and this research was done while N. Mahapatra was a Ph.D. student at the University of Minnesota.

†Dept. of Electrical & Computer Eng., State University of New York at Buffalo, Buffalo, NY 14260. Email:mahapatr@eng.buffalo.edu

‡Dept. of Electrical Eng. & Comp. Sc., University of Illinois at Chicago, Chicago, IL 60607. Email:dutt@eecs.uic.edu

[1]For simplicity, we assume $\mathcal{P}$ is a minimization problem; B&B can equally well be used to solve maximization problems.

tion to tree search spaces which is a realistic model for problem solving in practice [39]. The children of a given subproblem are those obtained from it by breaking it up into smaller problems (*branching*). This branching is such that the optimal solution to an internal subproblem node u is the least expensive among the optimal solutions to the child subproblem nodes of u. The leaves of $\mathcal{T}$ represent solutions to $\mathcal{P}$. Further, each subproblem node is associated with a cost f' that is a lower bound on the cost f of an optimal solution to that subproblem. We assume that f' is nondecreasing along any root-leaf path of $\mathcal{T}$. Consequently, any B&B search algorithm that has expanded or branched from all nodes u with cost less than C^* without finding a solution, and has found at least one solution node v with cost C^*, can conclude that v is an optimal solution.

Different B&B methods differ in the order they explore the search space. Best-first search (BFS) B&B (also referred to as A*) expands nodes in best- or least-cost first order among candidate nodes for expansion. Candidate nodes include all nodes that have been generated but not yet expanded and are stored in an *OPEN* list which is usually implemented as a heap to allow fast selection of the best node. As a result, BFS expands all nodes with cost less than the optimal-solution cost C^* and some nodes with cost equal to C^* before it finds an optimal solution. This is the minimum number of nodes (up to tie-breaking among nodes with cost C^*) that any B&B search algorithm guaranteeing optimality must expand [4]. Hence these nodes are referred to as *essential nodes*.

Many hard COPs have a worst-case complexity that is exponential in the input size, and their average-case complexities are also frequently very high, for example, a polynomial of degree three or higher. Since BFS stores all unexpanded, generated nodes, its memory requirement is similarly high, and as a result, it runs out of main memory on many problems; secondary storage access can be very slow, and it may or may not be available, especially, to processors of a parallel machine. Although the fact that memory chip capacities quadruple every three years does help offset this problem to some extent [1], there are two main reasons why alternative search methods that use only limited memory are needed. First, in many applications, there is a certain time period in which finding optimal solutions to as large a problem as possible is useful. Since processor speeds increase at the same rate as memory capacities [1], assuming a constant amount of computation per node expansion, the time for BFS to run out of memory remains more or less the same from year to year. If memory is exhausted much before the useful time period, then this prevents the possibility of solving larger problems. For instance, a good BFS implementation for solving the 15-puzzle problem can exhaust main memory on a 64MB workstation in under ten minutes [33]. Second, due to technological and economic progress, the complexity of systems such as airplane traffic, transportation systems, manufacturing systems, and VLSI chips, and hence the size of COPs associated with them, is continually increasing. For

example, the quadrupling of processor speeds every three years mentioned above is due in good part to higher levels of integration, which implies a corresponding increase in the number of devices per chip. This dramatically increases the time required to solve VLSI-CAD related COPs used in placement, routing, and floorplan optimization of these chips. As a result, the memory requirement for solving COPs of interest using BFS increases at a rate much higher than increase in memory capacity.

Due to the above reasons, limited-memory search methods are of great importance. These methods fall into two categories. Methods in the first category use solely depth-first search (DFS) and include depth-first B&B (DFBB) [21, 20], iterative-deepening A* (IDA*) [13], DFS* [37], MIDA* [38], and IDA*_CR [35]. To perform DFS, these methods employ a stack which stores information regarding nodes on some root-leaf subpath in the search space, where level 0 of the stack corresponds to the root node, and level i to the ith descendent of the root along that path. Each level also has information to keep track of the unexplored children of the corresponding node. Thus the space complexity of these methods is only $\Theta(bd)$, where b is the average branching factor and d is the depth of the search-space tree. Consequently, typically only a small fraction of the available memory is used. Methods in the second category, in contrast, attempt to make use of all available memory by first performing BFS, and then switching over to DFS at the tip nodes of the generated search subspace when there is only enough memory for the DFS stack. Examples of such methods are MREC [36], MA* [3], SMA* [33], and RA* [8]. In the next section, we describe the above methods and discuss their suitability for different types of COPs.

Since practical COPs solved by B&B are usually very hard, parallel processing has been used as a means to making them tractable [2, 5, 6, 7, 11, 12, 16, 19, 22, 23, 24]. However, most of this work has concentrated on parallelization of pure BFS in which sufficient memory is assumed to be available [2, 5, 6, 7, 16, 22, 23, 24]. Other related work concerns parallelization of ordinary DFS (i.e., without using lower-bound costs) [9, 15, 17, 18, 19, 27, 28, 34]. Very little research in parallel limited-memory B&B search has been attempted [8, 19, 29, 30, 31]. In the next section, we discuss and compare previous sequential and parallel limited-memory search methods. Then in Sec. 3, we present a new sequential limited-memory search method that can greatly improve upon previous methods, and in the following section we discuss its efficient parallelization. Finally, we conclude in Sec. 5.

2. Previous work. In this section, we describe previous sequential and parallel limited-memory search methods.

2.1. Sequential limited-memory search methods. We first describe pure DFS-based methods like DFBB, IDA*, and DFS*, then discuss combined BFS-DFS-based methods like MREC and MA*, and finally compare the two types of methods.

2.1.1. Pure DFS-based methods. DFBB starts with an upper bound on the optimal-solution cost and explores the state space in a DFS order [20]. Whenever a new solution is found that is better than the best one known so far, the upper bound is revised to the cost of that solution. Any node with cost equal to or exceeding the current bound is pruned. The algorithm terminates when the entire state space has been searched in this manner, and returns the best solution found during the search as an optimal solution. The overhead of DFBB with respect to BFS, which expands the least number of nodes of any search algorithm, is that it may expand nonessential nodes. The key to reducing the nonessential work of DFBB is to find tight upper bounds or good solutions early in the search. To increase the possibility of finding good solutions early, the children of a node are explored in a best-first order during DFS from the node [39]. Another possible approach to quickly locating good solutions is to search those parts of the state space first that appear promising based on the results of a search conducted in a simpler abstract search space to which the original problem has been mapped [10]. An application-dependent approach to quickly finding tight upper bounds is to execute an upper-bounding algorithm on the partial solutions represented by low-cost nodes encountered at relatively deeper levels during the search; this algorithm should be executed more frequently earlier in the search and less frequently later. DFBB is particularly suitable for solving COPs that have state spaces with high solution density, such as TSP [37].

IDA* is another DFS-based algorithm that performs a series of cost-bounded depth-first searches from the root node, pruning nodes when their cost exceeds the cost threshold for that iteration [13]. The cost threshold for the first iteration is equal to the root-node cost, and the threshold for each succeeding iteration is the minimum cost of all nodes that were generated but not expanded in the previous iteration. Thus, if $c_0 < c_1 < c_2 < \ldots$ represent costs in increasing order that one or more nodes in the search space have, where c_0 is the root-node cost, then IDA* performs cost-bounded depth-first search with a cost bound of c_i in the ith iteration. IDA* terminates when it finds a solution, which it then returns as an optimal solution. The drawback of IDA* is it that may have a high re-expansion overhead, which was analyzed in [37] as follows. Let V_i denote the set of nodes with cost c_i, $i \geq 0$, let $|V_0| = 1$, and let c_d denote the optimal-solution cost. Assume that $|V_{i+1}| = b|V_i|$, where $b \geq 1$ is called the *heuristic branching factor* [37]. Also, assume that BFS expands all nodes with cost c_d. Then the number of nodes expanded by BFS and IDA*, respectively, are:

$$W_{\mathrm{BFS}} = \sum_{i=0}^{i=d} |V_i| = \sum_{i=0}^{i=d} b^i = \frac{b^{d+1} - 1}{b - 1}, \quad \text{for} \quad b > 1,$$

and

$$W_{\mathrm{IDA}*} = \sum_{i=0}^{i=d} \sum_{j=0}^{j=i} |V_j| = \sum_{i=0}^{i=d} \frac{b^{i+1}-1}{b-1}, \quad \text{for} \quad b > 1$$
$$= \frac{b}{b-1}\frac{b^{d+1}-1}{b-1} - \frac{d+1}{b-1} \approx \frac{b}{b-1} W_{\mathrm{BFS}}, \quad \text{for} \quad b > 1.$$

The worst case for IDA* occurs when $b = 1$ and it expands just one new node in every iteration. In this case:

$$W_{\mathrm{IDA}*} = \sum_{i=1}^{i=W_{\mathrm{BFS}}} i = \Theta(W_{\mathrm{BFS}}^2), \quad \text{for} \quad b = 1.$$

Thus IDA* is suitable only for COPs with relatively large heuristic branching factors, such as the 15-puzzle problem for which $b \approx 6$ [37].

To reduce the re-expansion overhead of IDA*, an IDA*-type algorithm, DFS*, has been proposed in [37] that increases cost bounds more generously from iteration to iteration so that $B > 1$ times as many nodes expanded in the previous iteration are expanded in the current one. Due to the non-minimal cost-bound increases, the first solution found in DFS* will not necessarily be optimal. To determine an optimal solution, when a solution is found during an iteration, the remaining search space of that iteration is explored using DFBB search. It is shown in [37] that the optimal value of B that minimizes the worst-case complexity of DFS* is two—the worst case occurs when the cost bound for the penultimate iteration of DFS* is one less than the optimal-solution cost. Further, it is shown that in the worst case using this value of B, DFS* expands four times as many nodes as BFS. It can be also shown that for this optimal value of B, in the best case, when the cost bound for the last iteration of DFS* is equal to the optimal-solution cost, DFS* expands twice as many nodes as BFS. To achieve this optimal value of B, the cost interval between root-node cost and a finite, reasonably good upper bound on the optimal-solution cost, $maxcost$, is divided into a fixed number of equal-sized cost ranges, and a counter associated with each cost range. Each counter is used to keep track of the number of nodes expanded in a given iteration in the corresponding cost range, and thus the cost-wise distribution of nodes is determined. Then from this an average heuristic branching factor is computed and used to estimate the cost bound for the next iteration that will result in the expansion of twice as many nodes. Note that if no good upper bound on the optimal-solution cost is available, or if the upper bound is loose (in an extreme case, the costs of all essential nodes lie inside the first few cost ranges), then suitable cost bounds for successive iterations cannot be determined, and so DFS* cannot be applied. Also, if the cost-wise distribution of nodes in the search space does not fit well the heuristic-branching-factor model on which DFS* is based, cost bounds determined in DFS* may be much smaller or larger than

desired, resulting in excessive re-expansion or nonessential work overhead, respectively. Two algorithms similar to DFS*, MIDA* [38] and IDA*_CR [35], have been proposed independently.

2.1.2. Combined BFS-DFS-based methods. Next, we discuss previous combined BFS-DFS limited-memory B&B methods. The MREC algorithm performs BFS until memory is left only for a DFS stack, after which it executes IDA* from the stored frontier nodes in the *OPEN* list [36]. To reduce re-expansion overhead in MREC, one can use DFS* instead of instead of IDA*. We call this algorithm BFS-DFS*. The MA* algorithm similarly explores the search space like BFS until all memory is exhausted, except that in addition to storing the frontier nodes in *OPEN*, it also stores the expanded nodes in a *CLOSED* list.[2] Subsequently, MA* prunes a node of highest cost in *OPEN*, updates the cost of its parent to the minimum of its children's values, and places the parent node back in *OPEN*. Thus at any point, MA* may have partially expanded nodes. The idea behind MA* is that it explores the most promising path at any given instant while retracting nodes along least promising paths. Since nodes costs are nondecreasing along any path, different paths will be explored to different depths before being retracted, and then they will be explored to a greater depth later when they again becoming relatively promising, and so on. Therefore the number of nodes expanded by MA* and MREC are likely to be comparable. However, MA* is not as suitable for parallelization as MREC. This is because if the search space is split across processors and dynamic load balancing is employed, retraction in a processor may require access to a parent node sitting on another processor. The algorithms RA* [8] and SMA* [33] are similar to MA*.

2.1.3. Comparison of DFS-based and BFS-DFS-based methods. We now compare pure DFS-based methods and combined BFS-DFS based methods, and in particular DFS* and BFS-DFS* which are the best representative examples. Let m denote the memory capacity and assume that both m and W_{BFS}, which we denote for simplicity by W, are powers of two. The advantage of BFS-DFS based methods over DFS-based methods is that they make use of all available memory and hence incur less re-expansion overhead. Specifically, BFS-DFS* avoids the re-expansion of the m frontier nodes that it stores. Therefore, while $W_{\mathrm{DFS*}} = 1 + 2 + \ldots + m + 2m + \ldots + W = 2W$ (since both m and W are powers of two), $W_{\mathrm{BFS-DFS*}} = m + (2m - m) + (4m - m) + \ldots + (W - m) = 2W - m(1 + \log(\frac{W}{m}))$. Pure DFS-based methods have two main advantages over combined BFS-DFS-based methods [37, 14]. First, a child node in the former can be generated by making simple changes to the state of its

[2] Actually, MREC also stores the expanded nodes, but with the purpose of pruning duplicate nodes that might be encountered while searching a state-space graph. Since we are considering only tree search spaces, storing the complete expanded state space is unnecessary in MREC.

parent node, and the parent node's state can be reconstructed by undoing those changes. To allow this reconstruction, the modifications made to the parent-node state are stored at the level of the stack corresponding to the child node. Node generation in combined BFS-DFS-based methods includes time for memory allocation and the time to make a copy of the parent's state, modified appropriately. This extra time overhead in BFS-DFS* is proportional to the product of the number of nodes generated m and the size of a state representation. Second, unlike DFS-based methods, combined BFS-DFS-based methods incur an $O(m \log m)$ time overhead to insert and retrieve the m nodes stored during the initial BFS part of the search. These two advantages are significant only in applications for which node generation and heap insertion/retrieval time dominate or are comparable to lower-bound cost computation time. This is the case in COPs such as the 15-puzzle. In other applications, such as TSP, for which node expansion (especially lower-bound computation) is more expensive, BFS-DFS based methods are preferable.

2.2. Parallel limited-memory search methods. Next we discuss previous parallel implementations of limited-memory search methods on MIMD machines. In all these implementations, the basic parallelization consists of having each processor search a disjoint part of the search-space tree.

2.2.1. Parallel DFBB algorithms. In the parallel DFBB (PDFBB) algorithm of [19], initially the root node of the search space is with one processor which performs DFBB search from it. Work gets distributed from this to other processors as idle processors request work from their neighbors. The donor processor grants half of the untried alternatives (branches) at each level of its local stack to the recipient processor. In this manner, eventually all processors receive work and perform DFBB on their local stacks, obtaining work from neighbors whenever they become idle. Every processor stores the cost of the best solution it has found in a variable *best_soln*, and broadcasts any improvements in it to other processors, so that the global best-solution cost is known to all processors and can be used for pruning. A termination detection algorithm is used to detect the completion of the parallel DFBB search. There are two main drawbacks to this scheme. First, there is an initial idle time during which work gets distributed from one processor to all other processors. Second, because all work originates at one processor and this work is recursively split and distributed to other processors, there can be a wide disparity in the amount of work that processors near the origin processor and those far from it have. This will cause frequent idling of distant processors which in turn will trigger frequent work distribution from the origin to other processors, leading to a higher work-transfer overhead.

To circumvent the above problems, in the parallel DFBB algorithm of [31], called FDFBB, the root node is broadcast to all other processors, and

then subsequently all processors perform a replicated breadth-first search until there are enough nodes for every processor (anywhere between 100-3000 nodes depending upon node-expansion granularity). Then all processors pick different but equal number of nodes generated after breadth-first search, and perform DFBB search from those nodes. The startup time for DFBB search in this algorithm is the time to generate the initial nodes for every processor using a replicated sequential breadth-first search on them. Therefore the time saved compared to PDFBB is that required for work distribution from the origin processor in PDFBB, and is more for larger diameter machines. However, since the purpose of the startup phase in this algorithm is to generate enough nodes so that the total work can be more evenly divided among processors before the parallel DFBB phase and so that little load balancing is required during the latter phase, the startup overhead (as a percentage of the execution time time) can be considerable, especially for small- to medium-sized problems. Another feature of the parallel DFBB algorithm of [31] is that during work transfer, nodes are transferred instead of a split stack. This avoids the overhead of stack splitting and also incurs less overhead since node storage is more compact (generally of fixed size) than stack storage. Therefore this parallel DFBB algorithm is referred to as Fixed-size DFBB or FDFBB [31]. Performance results indicate that FDFBB performs much better than PDFBB

2.2.2. Parallel IDA* algorithms. In the parallel IDA* (PIDA*) algorithm of [29], a startup phase similar to that of PDFBB is executed, after which processors perform successive iterations of IDA* synchronously. In each iteration, PIDA* behaves similar to PDFBB, performing work transfer in the form of a split stack from a processor that has work to one that is idling. The completion of an iteration is detected by a termination-detection algorithm. A parallel IDA* algorithm that executes the different iterations of IDA* asynchronously is given in [30, 31]; this algorithm is called AIDA* for asynchronous IDA*. AIDA* executes a startup phase similar to FDFBB, after which processors perform IDA* on their starting nodes. When a processor completes an iteration of IDA*, it seeks work from other processors (such as the processors in its row and column in a 2-D torus). If no work is available, the processor proceeds with the next iteration of IDA*. So processors execute IDA* iterations asynchronously, because of which the first solution found need not be an optimal one—the solution may have been found by a processor executing an iteration beyond the one in which the optimal solution lies. Again as in FDFBB, work transfer in AIDA* is in the form of nodes and not split stacks. Performance results presented in [31] for 15 puzzle problems with $W \approx 10^9$ reveal that AIDA* performs much better than PIDA* and has an efficiency of approximately 70% on 1024 processors of a 2-D torus system. We will point out how IDA* and related algorithms like DFS* can be more efficiently parallelized in Sec. 4 when we discuss the parallelization of our new

limited-memory search algorithm that also performs iterative cost-bounded searches like IDA*.

3. Iterative extrapolated-cost bounded search (IES*). Here we propose an algorithm that has the potential of yielding significant improvements over previous limited-memory search methods. The algorithm performs iterative cost-bounded DFS like DFS*, estimating in each iteration the optimal-solution cost from its search experience in previous iterations, the estimates improving with successive iterations, and sets cost bounds to these estimated optimal-solution cost levels. The effectiveness of the algorithm depends upon the accuracy of the estimations which are obtained by extrapolating from current knowledge of the search space. Hence we call this algorithm *Iterative Extrapolated-Cost Bounded Search* (IES*). Below we first describe how these extrapolations are performed, and then explain their use in IES*.

3.1. Basis for solution-cost extrapolation. Consider a path $(u_1, u_2, \ldots, u_m)$ of maximum length in the search space from an internal node u_1 to a leaf or solution node u_m, such that u_i is a least- or best-cost child of u_{i-1}, for $1 < i \leq m$. We call this path a *best path*. Further, we refer to $(u_1, u_2, \ldots, u_l)$ as a *best subpath* at a given time if u_l is a leaf node of the search subspace explored at that time. Note that the cost of u_{i-1} is a lower-bound estimate of the cost of the best solution reachable from it. This best solution is likely to be a leaf of the subtree rooted at its best child u_i (or some other equal-cost child, if there are multiple best children). Consequently, we may say that $(f'(u_1), f'(u_2), \ldots, f'(u_m))$ represents an increasingly better sequence of estimates of the cost of the solution node u_m in which the last estimate $f'(u_m)$ is exact. The monotonicity of f' along an internal-node-leaf path arises from the fact that every internal node represents a partial solution, and with each node closer to the solution, a level of "uncertainty" or ambiguity with regard to the optimal solution reachable is removed as the partial solution increases by a unit size. For example, while solving TSP, with each node closer to the solution, the partial tour increases by the addition of one more edge and city [5]. Due to this diminishing uncertainty, the lower bound estimate f' improves as we move towards a solution node.

For simplicity, we use $f'(x)$ to mean $f'(u_x)$, where x is the position of node u_x on the best path. Further, we use $f'_{(1)}(x)$ to denote the slope of the $f'(x)$ versus x curve or the first derivative of $f'(x)$ with respect to x, and similarly $f'_{(2)}(x)$ to denote the second derivative of $f'(x)$ with respect to x. From our above discussion, $f'_{(1)}(x) \geq 0$. IES* is applicable to problems with either one of the following additional characteristics: (1) $f'_{(2)}(x) < 0$ (so that the increments in $f'(x)$ decrease along a best path), and either (a) solution nodes occur at a fixed depth D (this is the case in problems such as TSP in which solution nodes lie at a depth equal to the number of

cities [5, 6]), or (b) solution nodes may occur at any depth (as for example in the case of MIP [24]), but occur where the increments in $f'(x)$ become vanishingly small or where $f'_{(1)}(x) = 0$ and the $f'(x)$ curve becomes flat. In the former case, we estimate the solution cost by extrapolating the $f'(x)$ curve to the depth D, and in the latter case by extrapolating it to the point where $f'_{(1)}(x) = 0$. (2) $f'_{(2)}(x) \geq 0$ and solution nodes occur at a fixed depth D. We estimate solution nodes in this case as in 1(a). In the following, we consider only case 1; extrapolation in case 2 can be obtained analogous to case 1(a), except for minor differences which we will note. To allow reasonably good extrapolation, the best subpath from which an extrapolated solution cost is to be determined should be of sufficient length, say, of length at least three or four; we call such a best subpath a *sufficient best subpath*. Before we describe specific methods of extrapolation, we first briefly describe below how extrapolation is used in IES*.

3.2. Use of extrapolated costs in IES*. Initially, IES* performs cost-bounded DFS, increasing cost bounds minimally as in IDA*, until it has completed an iteration in which it first explored a sufficient best subpath. In each succeeding iteration, it selects among sufficient best subpaths encountered in the previous iteration that subpath which will yield (or is very likely to yield) the least extrapolated solution-cost estimate. To keep the overhead of extrapolation low, we perform this selection using a fast and approximate, but reliable, extrapolation approach. We refer to the selected best subpath as the *least best subpath*. Then a more accurate extrapolation is performed using the least best subpath, and the extrapolated solution-cost estimate obtained is used as the cost bound for the next iteration. When in an iteration it finds a solution, it explores the remaining search space using DFBB search. The best solution found in this last iteration of IES* (in which DFBB is performed) is returned as an optimal solution and the algorithm terminates. Next, we first describe the more accurate extrapolation approach used to determine cost bounds for the different iterations of IES*, and then describe the approximate extrapolation method used to select the least best subpath.

3.3. Accurate extrapolations via least-square curve fitting. We perform extrapolation by curve fitting using the least-squares method [26]. In this method, the curve $y(x)$ to be fitted to the set of points $\{(1, f'(u_1)), (2, f'(u_2)), \ldots, (m, f'(u_m))\}$ defined by a best subpath $(u_1, u_2, \ldots, u_m)$ consists of a linear combination of some n known functions $\phi_j(x)$ (to be discussed shortly) as below.

$$y(x) = \sum_{j=0}^{n-1} a_j \phi_j(x), \quad x \geq 1$$

The parameters a_j are determined so that the sum R of the squares of the deviations of $y(x)$ from the data points is minimized, where R is given by:

$$R = \sum_{i=1}^{m} [f'(u_i) - y(i)]^2 .$$

We minimize R by setting the partial derivatives $\frac{\partial R}{\partial a_l}$ of R with respect to the undetermined coefficients a_l, $l = 0, 1, \ldots, n-1$, equal to zero, to obtain the following set of n simultaneous equations in the n unknown a_l's.

$$\sum_{j=0}^{n-1} \left[\sum_{i=1}^{m} \phi_j(i)\phi_l(i) \right] a_j = \sum_{i=1}^{m} f'(u_i)\phi_l(i), \quad l = 0, 1, \ldots, n-1.$$

These equations can be solved for the a_l's using Gaussian elimination in $\Theta(n^3)$ time. The more appropriate the $\phi_j(x)$'s, and the more the number of functions n and the number of data points m, more accurate will be the extrapolation. The first two variables are under our control. We discuss the issue of appropriate $\phi_j(x)$'s to use in the next paragraph. The number of functions is related to the $\phi_j(x)$'s used, since the more appropriate the functions used, the less will be the number of them required to obtain a given accuracy in extrapolation. Further, in applications with higher node-expansion granularity, we can use more number of functions to increase accuracy, since extrapolation overhead will be relatively smaller. When node-expansion granularity is very small, we can use the approximate extrapolation method to be described shortly. Normally, with appropriate functions, $n = 3$ or 4 and $m \geq 3$ or 4 should suffice for reasonably good extrapolation, so that curve fitting is not very compute intensive.

The appropriate $\phi_j(x)$'s are those that best capture the behavior of the lower-bound function $f'(x)$. We already know two characteristics of $f'(x)$, viz., $f'_{(1)}(x) \geq 0$ and $f'_{(2)}(x) < 0$ (in case 2, $f'_{(2)}(x) \geq 0$). Therefore, we focus on those $\phi_j(x)$'s which have *different* signs for their first and second derivatives with respect to x—the *correct* signs for the first and second derivatives of $a_j\phi_j(x)$ will then be automatically obtained by the Gaussian elimination procedure which determines the a_j's and their signs. Examples of such $\phi_j(x)$'s include $\frac{1}{\log^j(1+x)}$, $\frac{1}{x^{j/2}}$, $\frac{1}{x^j}$, and $\frac{1}{e^{jx}}$, which in that order are suitable for $f'(x)$'s with increasing growth rates (in case 2, the reciprocals of these functions will be suitable); note that for all these $\phi_j(x)$'s, a_j will be negative so that $a_j\phi_j(x)$ has the correct sign. One can also form a versatile or more general-purpose curve fitting function by combining functions with different growth rates, for example, by combining the five functions: $\phi_0(x) = 1$, $\phi_1(x) = \frac{1}{\log(1+x)}$, $\phi_2(x) = \frac{1}{x^{1/2}}$, $\phi_3(x) = \frac{1}{x}$, and $\phi_4(x) = \frac{1}{e^x}$. If the behavior of $f'(x)$ for a given lower-bound function and application is known from prior experience, then appropriate $\phi_j(x)$'s can be used. Otherwise, we perform a "binary search" for the best fitting $\phi_j(x)$ as follows.

First, we pick a set of k candidate $\phi_j^l(x)$'s, $1 \leq l \leq k$, that are appropriate for $f'(x)$'s with different growth rates, and arrange them in order of their growth rates. For example, for $k = 4$, we can choose: $\phi_j^1(x) = \frac{1}{\log^j(1+x)}$, $\phi_j^2(x) = \frac{1}{x^{j/2}}$, $\phi_j^3(x) = \frac{1}{x^j}$, $\phi_j^4(x) = \frac{1}{e^{jx}}$, which in that order represent increasingly higher growth-rate families of functions. During the initial IDA* part of IES* when we first encounter a sufficient best subpath, we explore further along the corresponding best path until we have extended the subpath by a reasonable length, say, by 10-20 nodes. Then we determine a best fit for the first half of the points of the extended subpath using $\phi_j^{k/2}(x)$ that lies in the middle of the ordering of the $\phi_j^l(x)$'s. If the extrapolated value of $\phi_j^{k/2}(x)$ at the tail of the extended subpath exceeds the actual value, then we try to find a best fit using $\phi_j^{k/4}(x)$ (a lower growth-rate family of functions); otherwise, we consider $\phi_j^{3k/4}(x)$ (a higher growth-rate family of functions). In this manner, we recursively explore the $\phi_j^l(x)$'s until we find one that best predicts (has the least error at) the tail of the extended subpath. This takes $\Theta(n^3 \log k)$ time. If the heuristic branching factor is reasonably large, it is better to underestimate the optimal-solution cost than to overestimate it, to avoid too much non-essential work in the last iteration of IES*. Therefore, in these cases, we try to find a $\phi_j^l(x)$ that not only minimizes the error in prediction, but also underestimates the actual value. Note that this extended search along a best subpath is performed only once to determine an appropriate $\phi_j^l(x)$ to use. Subsequently, we use this best $\phi_j^l(x)$ to fit best subpaths and compute extrapolated costs.

3.4. Fast, approximate extrapolations. We now describe an approximate extrapolation method that is suitable for finding the least best subpath to which the more accurate extrapolation method described above is to be applied, and that may be used instead of the latter when the node-expansion granularity is very small. In this method, we assume simply that the slope $f'_{(1)}(x)$ of the lower-bound estimate $f'(x)$ along a best subpath $(u_1, u_2, \ldots, u_m)$ either decreases toward zero in a geometric series ($s_g = \frac{f'(u_2)-f'(u_1)}{2-1} = f'(u_2) - f'(u_1), s_g r_g = f'(u_3) - f'(u_2), s_g r_g^2 = f'(u_4) - f'(u_3), \ldots$) or arithmetic series ($s_a = f'(u_2) - f'(u_1), s_a + r_a = f'(u_3) - f'(u_2), s_a + 2r_a = f'(u_4) - f'(u_3), \ldots$) with increasing x depending upon whether the observed decrease in slope is drastic (close to exponential) or gradual (close to linear), where $r_g < 1$ and $r_a < 0$ are the common ratio and common difference of the two series, respectively (in case 2, $r_g \geq 1$ and $r_a \geq 0$). Which assumption is more appropriate for a lower-bound function is either known from past experience or is determined in the beginning of IES* in the same manner as the appropriate $\phi_j(x)$'s to use as described earlier. Here we just discuss the case in which along a best subpath the slope varies in a geometric series. An average r_g that takes the actual values of all the $f'(u_i)$'s of the path into account is given by the geometric

mean:

$$\bar{r}_g = \left[\prod_{i=2}^{m-1} \frac{f'(u_{i+1}) - f'(u_i)}{f'(u_i) - f'(u_{i-1})} \right]^{1/(m-2)}$$

and an average s_g also by the geometric mean:

$$\bar{s}_g = \left[\frac{\prod_{i=2}^{m} f'(u_i) - f'(u_{i-1})}{\bar{r}_g^{(m-2)(m-1)/2}} \right]^{1/(m-1)}.$$

The extrapolated solution cost can then be readily computed as follows. In applications where solution nodes occur at a fixed depth D (case 1(a)), the estimated solution cost is $f'(u_i) + \frac{\bar{s}_g(1-\bar{r}_g^{(D-D(u_i))})}{1-\bar{r}_g}$, where $D(u_i)$ is the depth at which u_i occurs. In applications where solution nodes may occur at any depth, but at a point where increments in $f'(x)$ becomes vanishingly small (case 1(b)), the estimated solution cost is equal to the asymptotic value of $f(x)$ as x approaches ∞. This asymptotic value is $f'(u_i) + \frac{\bar{s}_g}{1-\bar{r}_g}$. Thus extrapolations can be made using the above method with a small constant overhead per node of a best subpath.

3.5. Fruitfully utilizing available memory. Next, we consider the issue of fruitfully utilizing the available memory. In Sec. 2.1.3, we had discussed the reduction in re-expansion overhead obtained by storing m nodes, and the overheads of generating and inserting/deleting these nodes in *OPEN*. Clearly, there is an optimal value of m (within the available memory capacity) which gives us the maximum savings, and this value depends upon the node-expansion granularity and the size of the state representation of a node. Therefore in IES*, we initially execute BFS until the optimal number of nodes are formed, and then perform cost-bounded search as described above from these nodes. Further, these nodes need not be sorted in any order, so that node retrieval takes constant time.

Actually, the reduction in re-expansion overhead can be increased beyond just avoiding the re-expansion of the m nodes stored, as follows. When we perform cost-bounded search from a stored node u in any given iteration of IES*, we keep track of the cost of the best or least-expensive child v of leaf nodes of the subtree explored under the node u. Node v is the best unexplored descendent of u and its cost represents a more tighter lower-bound estimate for the true cost of u. Therefore we update u's cost to $f'(v)$. Thus the cost of stored nodes will (most likely) increase with every iteration of IES*. As a result, during an iteration of IES*, some stored nodes may have costs exceeding the cost bound for that iteration. We can avoid performing cost-bounded search from these nodes, since they do not have any unexplored descendent node within the cost bound. This type of bound strengthening will be most useful in IDA* since it has minimally spaced successive cost bounds.

3.6. Comparison with DFS*. Here we compare IES* to the best previous limited-memory search method DFS*. Recall from Sec. 2.1.1 that IDA* performs well only when the heuristic branching factor b is high, otherwise it incurs significant re-expansion overhead. DFS* was proposed to reduce this overhead by artificially increasing b when it is small to an optimal B value by appropriately adjusting cost bounds for successive iterations. However, as stated in Sec. 2.1.1, it still incurs appreciable overheads of 100% to 300% due to re-expansions (in iterations preceding the last iteration) and non-essential work (in the last iteration). Although IES* also perform cost-bounded DFS like DFS*, but since IES* attempts to set cost bounds as close as possible to the optimal-solution cost level right from the beginning, it requires fewer iterations, and hence incurs less re-expansion overhead, and performs less non-essential work in the last iteration. Recall also from Sec. 2.1.1 that DFS* is effective only when a reasonable upper bound on the optimal-solution cost is available and when the problem search-space fits the heuristic-branching-factor model reasonably well. IES*, on the other hand, is applicable to problems with characteristics identified in Sec. 3.1, which are not very restrictive, and is effective when the behavior of $f'(x)$ can be predicted reasonably closely by good extrapolation methods as described earlier. In future research, we will compare DFS* and IES* by applying them to solve several real and randomly-generated problems.

4. Parallel IES*. In this section, we describe our approach to the parallelization of IES*; this approach can be applied with minor modifications to parallelize other cost-bounded search algorithms like IDA* and DFS*.

4.1. Parallel startup phase. Initially, the root node is broadcast to all processors. Then in $\log P$ steps all P processors execute a parallel startup phase that we developed in [5, 6] to obtain their unique starting nodes. The startup phase is briefly described as follows. At the beginning of step i, $0 \leq i < P$, there are 2^i disjoint processor sets, and each processor in a set has a copy of the same node which is unique to that set. Then during step i, processors in a set execute BFS to form two nodes, and then the set is split into two equal smaller sets with each smaller set picking up one of the newly generated nodes. Thus after $\log P$ steps every processor will have its unique node. The startup phase entails no communication and processors execute their steps asynchronously. Since the starting nodes for processors are generated using local BFS in the different processor sets, they are of good quality. To further improve the static load balance at the end of the startup phase, we can generate more than two nodes (a multiple of two) in each step and distribute them equitably among the smaller processor sets. We prove in [6] that, under reasonable assumptions, our parallel startup phase performs optimal load balance across processors.

Subsequently, each processor executes BFS until it has the optimal

number of nodes which fruitfully utilize its memory as discussed in Sec. 3.5. Then, each processor asynchronously executes the different iterations of IES* on its stored nodes, interacting with other processors primarily for load balancing; we will shortly discuss how load balancing is performed and how cost bounds for successive iterations in different processors are determined. When a processor first finds a solution, it broadcasts it to all other processors. All processors thenceforth perform DFBB search, broadcasting any new better solutions found to other processors. The best solution found in this last iteration of IES* is returned as the optimal solution.

4.2. Load balancing. Due to the asynchronous execution of parallel IES*, different processors may be executing different iterations. Indeed, due to load balancing, a processor may have different sets of nodes on which a different iteration needs to be executed next—we will see later that not more than two such sets of nodes are likely to be present in a processor. Each processor stores these different sets of nodes in separate *OPEN* lists. We denote by $OPEN_p(i)$ the *OPEN* list in processor p that has nodes on which iteration i is to be executed next. Moreover, to facilitate load balancing, a "load value" equal to the size of the search subspace explored in iteration $(i-1)$ from a node in $OPEN_p(i)$ is stored with the node. Also, processor p stores the cumulative load value $w_p(i)$ associated with the nodes in $OPEN_p(i)$; if a node in $OPEN_p(i)$ is currently being processed, then $w_p(i)$ will continuously decrease to reflect the number of descendents of the node that have been explored in iteration i that were also explored in iteration $(i-1)$. We define $W_p(i) = \sum_{j=1}^{i} w_p(j)$, and $\lambda_p = \min(j)$ such that $W_p(j) > 0$.

Ideally, we would like the startup phase to result in perfect load balance so that even when all processors execute IES* independently, they will be performing the same iteration. Since this is unlikely, we should let processors proceed asynchronously with their iterations in order to maximize processor utilization. However, processors that are ahead of others may end up expanding too many nonessential nodes if the optimal solution node is in the search subspace of the latter processors. Therefore we need a load balancing scheme to ensure that processors are not only kept busy, but kept busy executing iterations that are as close to each other as possible. For this, we employ an efficient, recipient-initiated near-neighbor quantitative load balancing scheme we developed in [5, 6], modifying it appropriately so that nodes on which earlier iterations are to be executed are processed with higher priority than those on which later iterations are to be executed. Simply described, the scheme is applied as below.

Each processor p reports any significant percentage changes (say, a 10% change) in the loads $w_p(i)$ for its different *OPEN* lists to its neighbors. Whenever it notices that for some neighbor q, $\lambda_p \geq \lambda_q$ and $W_p(\lambda_q)$ is less than a certain a threshold, it requests nodes from the neighbor q with

the least λ_q, and among all such neighbors, the one with the maximum $w_q(\lambda_q)$. The reason work is requested whenever a processor is *about to* process rather than when it *has started* processing nodes of lower priority than neighboring processors, is to avoid low priority work during the time it takes to procure the required higher priority work. The threshold used for requesting work depends upon the node-expansion granularity and is larger for low granularity problems.

On receiving the work request, processor q grants a fraction of its $w_q(\lambda_q)$ load by transferring an appropriate set of nodes (whose cumulative load value equals the desired fraction of $w_q(\lambda_q)$) from $OPEN_q(\lambda_q)$ to processor p. If processor p does not have space to store the received nodes, it gives away an equal number of lower priority nodes to processor q. In the unlikely case that processor p has begun processing a low priority node before it receives higher priority nodes from q, it suspends its former work and processes the higher priority nodes received. To enable a processor to suspend low priority work and pursue higher priority work in this manner, we reserve space for multiple stacks in each processor instead of just for one stack as in sequential IES*, so that suspended work can be later resumed. Since, as we shortly argue, all processors are likely to be executing the same iteration, reserving space for three stacks will suffice. Note that when processor p receives work as above, it in turn can grant a fraction of the received higher priority work to its other neighbors that are also processing low priority nodes. As a result of such load balancing, high priority work gets distributed throughout the parallel system, and all processors are very likely to be executing the same iteration of IES*.

4.3. Determining cost bounds for different iterations. Next, we explain how cost bounds for successive iterations are determined in different processors. Recall from Sec. 3.2 that in sequential IES*, we determined first the least best subpath in an iteration using a fast, approximate extrapolation method, and then computed the actual cost bound for the next iteration from this best subpath using an accurate extrapolation method. Since in parallel IES* processors proceed with the next iteration without waiting for the current iteration to complete in other processors, cost bounds have to be determined differently. Specifically, our approach is as follows. Let $c_p(i)$ denote the cost bound for iteration i in processor p, let $t_p(i)$ denote the time when processor p comes to know that all iteration $(i-1)$ nodes in all processors have been processed (i.e., when it comes to know that iteration $(i-1)$ is complete and there are only iteration j, where $j > (i-1)$, nodes with processors), and define $t(i) = \max(t_p(i))\ \forall\ p$. We will shortly discuss how p comes to know when iteration $(i-1)$ is complete.

The cost bound $c_p(i)$ is initialized to $+\infty$ and has a "tentative" dynamic value until time $t_p(i)$. At any time before $t_p(i)$, whenever a processor communicates with other processors, say, when it communicates load information to neighbors as described earlier, it piggybacks onto the message all

its cost bounds for iterations $j > (i-1)$ that have been set to finite values from their initial value of $+\infty$. Also, at any time before $t_p(i)$, $c_p(i)$ is set to the minimum extrapolated solution cost obtained by extrapolating along sufficient best paths encountered while processing iteration $(i-1)$ nodes in processor p till that time. For this we use a fast, approximate extrapolation method that gives a higher (but not much different) extrapolated cost compared to that obtained from an accurate method. Apart from the above updates, $c_p(i)$ is revised to any piggybacked $c_q(i)$ value of smaller magnitude that it receives from another processor q. Between times $t(i-1)$ and $t(i)$, we run a delay-optimal and message-efficient termination detection algorithm that we developed in [25] to detect and inform all processors of the completion of iteration $(i-1)$. We use this algorithm to simultaneously determine the processor p that has the minimum cost bound for iteration i and also the corresponding best subpath. Note that this best subpath in p will be the same as the least best subpath (or one yielding the same extrapolated cost) determined in iteration $(i-1)$ of sequential IES*. Processor p is notified of this, and it then uses an accurate extrapolation method on its least best subpath to compute the true cost bound $c(i)$ for iteration i, and broadcasts it to all other processors. All processors then revise their iteration i cost bounds to this final value.

4.4. Summary. We can summarize the operation of parallel IES* as follows. After an initial parallel startup phase, processors execute different iterations of IES* asynchronously. Initially, the cost bounds for any iteration i differ from processor to processor and are greater than the true cost bound for that iteration. These cost bounds are independently determined by processors from their own search experience in the previous iteration and from cost-bound information received from other processors that communicate with them (say, for load balancing). As search experience increases and more and more information becomes available, these cost bounds decrease toward the true cost bound. The cost bounds stabilize at the true value when the previous iteration $(i-1)$ is complete in all processors. During the time that cost bounds take to stabilize, any processor pursuing iteration i with a greater than true cost bound may end up exploring more nodes than necessary (nodes with cost exceeding the true cost bound, some of which may be nonessential). However, because of prioritized load balancing, any iteration $(i-1)$ nodes will be quickly distributed and processed before iteration i nodes are processed. This means that the time gap between the beginning of iteration i in any processor and the completion of iteration $(i-1)$ in all processors (when the true cost bound for iteration i becomes known to all processors) will be very small. This greatly reduces the likelihood of exploring unnecessary nodes in iteration i, and by extension, in any other iteration. Thus the number of nodes expanded by parallel IES* will be similar to that expanded by sequential IES* minus the savings in re-expansion overhead obtained by storing m

nodes per processor in all P processors; from Sec. 2.1.3, this saving will be $mP(1 + \log(\frac{W}{mP}))$.

4.5. Comparison with AIDA*. As noted in the beginning of this section, our approach to the parallelization of IES* can be applied with minor modifications to parallelize other cost-bounded search algorithms. Here we compare our approach to that adopted in the best previous parallel cost-bounded search algorithm AIDA* [30, 31] discussed in Sec. 2.2.2. Although, like parallel IES*, AIDA* also executes cost-bounded search iterations asynchronously, there are some significant differences. First, the startup phase in AIDA* is completely replicated and executed sequentially in all P processors thus taking $\Theta(P)$ time as opposed to the $\Theta(\log P)$ time that our parallel startup phase takes. Moreover, the starting nodes in AIDA* are generated from breadth-first search as opposed to local BFS in our algorithm, so that the starting nodes in the former case are likely to be of much more disparate quality, leading to a poorer initial static load balance. Next, processors in AIDA* proceed with the next iteration when they do not find nodes of the current iteration in the few processors they request work from. Thus, essentially, processors always move from one iteration to the next, even though work in earlier iterations might be available at processors that they do not request work from. This can lead to some processors racing ahead of others, and such processors are likely to perform non-essential work. Furthermore, the cost bounds for an iteration are determined by different processors independently from their local search experience and hence will be different. Consequently, processors with bad search spaces will likely set much larger cost bounds than that in the sequential case, and hence these processors will end up performing non-essential work. Finally, our load balancing algorithm is much more sophisticated and efficient in that it performs anticipatory, prioritized load balancing taking into account load values of nodes.

5. Conclusions. In this paper, we surveyed previous DFS-based and BFS-DFS based limited-memory search methods and discussed their suitability for different types of COPs. We proposed a new method, iterative extrapolated-cost bounded search (IES*), that can dramatically reduce the re-expansion overheads incurred in previous cost-bounded search methods like DFS*. We also developed an efficient asynchronous approach to parallelizing IES* that can also be applied to other cost-bounded search methods. We discussed how our approach addresses the problems associated with that used in the best previous parallel cost-bounded search algorithm AIDA*. Our parallel IES* algorithm incorporates a parallel startup phase, a method for independently determining cost bounds in different processors, and a prioritized load balancing technique to reduce initial startup time and unnecessary node expansions arising from asynchronous operation. In future research, we will study the performance of IES* by applying it to solve several real and randomly-generated problems in both sequential

and parallel settings.

REFERENCES

[1] G.S. ALMASI AND A. GOTTLIEB, *Highly Parallel Computing*, Benjamin/Cummings, Redwood City, CA, 1994.

[2] S. ANDERSON AND M.C. CHEN, *Parallel Branch-and-Bound Algorithms on the Hypercube*, Proc. Second Conference on Hypercube Multiprocessors, pp. 309–317, 1987.

[3] P.P. CHAKRABARTI, S.. GHOSE, A. ACHARYA, AND S.C. DE SARKAR, *Heuristic search in restricted memory*, Artificial Intelligence, **Vol. 41**, pp. 197–221, 1989.

[4] R. DECHTER AND J. PEARL, *Generalized best-first search strategies and the optimality of A**, Journal of the ACM, **Vol. 32**, pp. 505–536, 1985.

[5] S. DUTT AND N.R. MAHAPATRA, *Parallel A* Algorithms and their Performance on Hypercube Multiprocessors*, Seventh Int'l Par. Proc. Symp., pp. 797–803, Apr. 1993.

[6] S. DUTT AND N.R. MAHAPATRA, *Scalable Load Balancing Strategies for Parallel A* Algorithms*, Journal of Parallel and Distributed Computing, **Vol.22, No.3**, pp. 488–505, Sep. 1994.

[7] J. ECKSTEIN, *Parallel branch-and-bound algorithms for general mixed integer-programming on the CM-5*, SIAM Journal on Optimization, 1994.

[8] M. EVETT, J. HENDLER, A. MAHANTI, AND D. NAU, *PRA*: A memory-limited heuristic search procedure for the Connection Machine*, Proc. 3rd Symp. on the Frontiers of Mass. Par. Computation, pp. 145–149, 1990.

[9] C. FERGUSON AND R. KORF, *Distributed tree search and its application to alpha-beta pruning*, In Proc. 1988 National Conf. Artificial Intelligence, Aug. 1988.

[10] R.C. HOLTE, C. DRUMMOND AND M.B. PEREZ, *Searching with abstractions: A unifying framework and new high-performance algorithm*, Proc. 10th Canadian Conf. on AI, pp. 263–270, 1994.

[11] S.-R. HUANG AND L.S. DAVIS, *Parallel Iterative A* Search: An Admissible Distributed Heuristic Search Algorithm*, Proc. Eleventh Int'l Joint Conf. on Artificial Intelligence, pp. 23–29, 1989.

[12] R.M. KARP AND Y. ZHANG, *A Randomized Parallel Branch-and-Bound Procedure*, J. of the ACM, pp. 290–300, 1988.

[13] R.E. KORF, *Depth-first iterative deepening: An optimal admissible tree search*, Artificial Intelligence, **Vol. 27**, pp. 97–109, 1985.

[14] R.E. KORF, Linear-space best-first search, Artificial Intelligence, **Vol. 62**, pp. 41–78, 1993.

[15] V. KUMAR AND V.N. RAO, *Parallel depth first search, part II: Analysis*, International Journal of Parallel Programming, **Vol. 16, No. 6**, pp. 501–519, Dec. 1987.

[16] V. KUMAR, K. RAMESH AND V.N. RAO, *Parallel Best-First Search of State-Space Graphs: A Summary of Results*, Proc. 1988 Nat'l Conf. Artificial Intell., 1988.

[17] V. KUMAR, V.N. RAO, AND K. RAMESH, *Parallel depth first search on the ring architecture*, Proc. of the 1988 International Conference on Parallel Processing, **Vol. 3**, pp. 128–32, University Park, PA, Aug. 15–19, 1988.

[18] V. KUMAR AND V.N. RAO, *Load Balancing on the Hypercube Architecture*, Proc. Hypercubes, Concurrent Comp., Appli., Mar 1989.

[19] V. KUMAR AND V.N. RAO, *Scalable parallel formulations of depth-first search*, in Kumar, Gopalakrishnan, Kanal, editors, Parallel Algorithms for Machine Intelligence and Vision, Springer, pp. 1–41, 1990.

[20] V. KUMAR, *Branch-and-bound search*, in S. C. Shapiro, editor, Encyclopedia of Artificial Intelligence, pp. 1468–1472, Wiley-Interscience, New York, 2nd edition, 1992.

[21] E.L. LAWLER AND D.E. WOOD, *Branch-and-bound methods: A survey*, Operations Research, **Vol. 14**, pp. 699–719, 1966.

[22] R. LULING AND B. MONIEN, *Load Balancing for Distributed Branch-and-Bound Algorithms*, Sixth Int'l Par. Proc. Symp., pp. 543–548, 1992.

[23] N.R. MAHAPATRA AND S. DUTT, *New Anticipatory Load Balancing Strategies for Parallel A* Algorithms*, American Mathematical Society's Proc. in the DIMACS Series in Discrete Mathematics and Theoretical Computer Science, **Vol. 22**, pp. 197–232, 1995.

[24] N.R. MAHAPATRA AND S. DUTT, *Random seeking: A general, efficient, and informed randomized scheme for dynamic load balancing*, Proc. Tenth International Parallel Processing Symposium, pp. 881–885, Honolulu, Hawaii, Apr. 15–19, 1996.

[25] N.R. MAHAPATRA AND S. DUTT, *An efficient delay-optimal distributed termination detection algorithm*, to be submitted to Journal of Parallel and Distributed Computing.

[26] S. NAKAMURA, *Applied numerical methods in C*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[27] C. POWLEY, C. FERGUSON, AND R.E. KORF, *Depth-first heuristic search on a SIMD machine*, Artificial Intelligence, **Vol. 60, No. 2**, pp. 199–242, Apr. 1993.

[28] V.N. RAO AND V. KUMAR, *Parallel depth first search, part I: Implementation*, International Journal of Parallel Programming, **Vol. 16, No. 6**, pp. 479–499, Dec. 1987.

[29] V.N. RAO, V. KUMAR, AND K. RAMESH, *A parallel implementation of iterative deepening A**, Proc. Fifth National Conference on Artificial Intelligence (AAAI-87), pp. 878–882, 1987.

[30] A. REINEFELD AND V. SCHNECKE, *AIDA* - Asynchronous parallel IDA**, Tenth Canadian Conf. on Artificial Intelligence (AI-94), Banff, Canada, May 1994.

[31] A. REINEFELD AND V. SCHNECKE, *Work-load balancing in highly parallel depth-first search*, Proceedings of the Scalable High-Performance Computing Conference, pp. 773–780, Knoxville, TN, May 1994. *Parallel depth first search, part I: Implementation*, International Journal of Parallel Programming, **Vol. 16, No. 6**, pp. 479–499, Dec. 1987.

[32] E. RICH, *Artificial Intelligence*, McGraw Hill, New York, pp. 78–84, 1983.

[33] S. RUSSELL, *Efficient memory-bounded search methods*, Procs. of the 10th European Conf. on Artificial Intelligence (ECAI-92), Vienna, Austria, 1992.

[34] V.A. SALETORE AND L.V. KALE, *Consistent linear speedups to a first solution in parallel state-space search*, Proc. Eighth National Conference on Artificial Intelligence (AAAI-90), **Vol. 2**, pp. 227–233, Boston, MA, Jul. 29 - Aug. 3, 1990.

[35] 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, 1991.

[36] A.K. SEN AND A. BAGCHI, *Fast recursive formulations for best-first search that allow controlled use of memory*, Proceedings 11th Int'l Joint Conf. on Artificial Intelligence (IJCAI-89), pp. 297–302, Detroit, MI, Aug. 1989.

[37] N.R. VEMPATY, V. KUMAR, AND R.E. KORF, *Depth-first vs best-first search*, Proc. 9th Nat'l Conf. Artificial Intelligence (AAAI-91), pp. 434–440, Anaheim, CA, Jul. 1991.

[38] B.W. WAH, *MIDA*, an IDA* search with dynamic control*, Technical report UILU-ENG-91-2216 CRHC-91-9, Center for Reliable and High Performance Computing Coordinated Research Lab, College of Eng., Univ. of Illinois at Urbana Champagne-Urbana, IL, 1991.

[39] W. ZHANG AND R.E. KORF, *Performance of linear-space search algorithms*, Artificial Intelligence, **Vol. 79, No. 2**, pp. 241–292, 1995.