The Journal of Supercomputing, 32, 231–250, 2005

Distributed Unidirectional and Bidirectional Heuristic Search: Algorithm Design and Empirical Assessment

ABDEL-ELAH AL-AYYOUB ayyoub@acm.org
Faculty of Computer Studies, Arab Open University, P.O. Box 18211, Bahrain

Abstract. Since its introduction three decades ago, bidirectional heuristic search did not deliver the expected performance improvement over unidirectional search methods. The problem of search frontiers passing each other is a widely accepted conjecture led to amendments to steer the search using computationally demanding heuristics. The computation cost associated with front-to-front evaluations crippled further investigation and hence bidirectional search was long neglected. However, recent findings demonstrate that the initial conjecture is wrong since the major search effort is spent after the frontiers have already met [7]. In this paper we reconsider bidirectional search by proposing a new generic approach based on cluster computing. The proposed approach is then evaluated and compared with its unidirectional counterparts. The obtained results reveal that cluster computing is a viable approach for distributed heuristic search. Particularly, clustered bidirectional search is capable of solving problems beyond unidirectional search capabilities and in the same time outperforms unidirectional approaches in terms of memory space and execution time.

Keywords: cluster computing, distributed heuristic search, performance evaluation

1. Introduction

A variety of problems in planning, decision-making, theorem proving, expert systems, combinational problems, constraint satisfaction problems, and discrete optimization problems involve extensive computational power to inspect a space of possible alternatives [8, 12, 14]. There are two categories of search algorithms, *brute force* and informed (*heuristic*) search algorithms. The first category evaluates alternatives one after another with no distinction between alternatives which induces huge amounts of computation. Heuristic search, on the other hand, is the standard artificial intelligence technique to alleviate the search load. Heuristic search can be viewed as a path-fining problem in an implicit graph. The search begins by generating the successors of a start node. At the subsequent steps, one of the previously generated nodes is expanded until a goal node is found, assuming that one exists, where the solution path is constructed by following the pointers from the goal node back to the start node. Figure 1 explains this process diagrammatically.

The heuristic search is characterized by the 4-tuple $\langle \sigma, \tau, \xi, f \rangle$, where σ is the start node, τ is the goal node, $\xi = \{g_1, g_2, \ldots, g_k\}$ is the set of generators used to expand nodes, and f is the evaluation function which for each node n gives the estimated cost of the solution path from σ to τ constrained to go through n. This function, referred to as $f(n)$, is composed of a known part $g(n)$ that represents the cost of the path from σ to n plus an estimated part $h(n)$ that represents the cost of the path from n to τ. The latter part, called the heuristic function, is based on application-specific heuristics that are obtained from the problem domain.

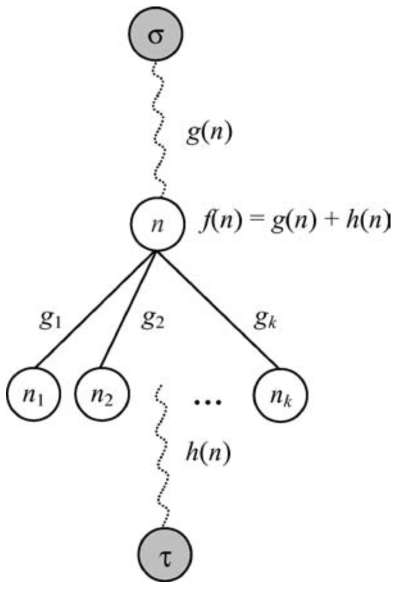

Figure 1. Heuristic search process.

There are a number of heuristic search algorithms proposed in the literature [13, 14]. The main difference between the various heuristic search algorithms lay in their completeness and admissibility [14].

The main objective of heuristic search is to cope with the huge demand on memory space and computing time. Iterative deepening search [8] alleviates the demand on memory space (on the expense of search time) by performing a series of incrementally deepening depth-first searches. Some other search algorithms such as the front-to-front heuristic search [4] puts a lot of emphasis on the heuristic information to reduce the explored search space.

Bidirectional search is another attempt proposed three decades ago [4, 15] to cope with the exponential space and time requirements. This approach maintains two search spaces; one rooted at σ and grows downward and the other rooted at τ and grows upward. When the search frontiers meet in a middle point the algorithm terminates with the solution path. The expected reduction in search effort, see Figure 2, was not observed in the experiments [15], and hence there was a consensus that bidirectional search is afflicted by the so called *missile metaphor*; the problem of search frontiers passing each other without intersecting. However, recent findings blemished this conjecture [7]. The authors showed that the major search effort is spent after the frontiers have already met in order to satisfy the optimality concerns. The work in [7] proposes to reconsider new generic approaches for bidirectional search without the time-consuming front-to-front evaluations. The obtained results are encouraging and hence the authors suggest that bidirectional search be reconsidered as an attractive alterative for unidirectional counterparts.

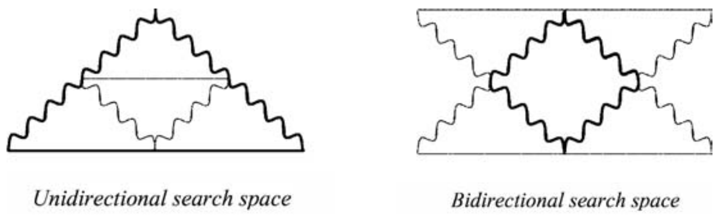

Figure 2. Unidirectional versus bidirectional search space.

Motivated by these findings, this paper reconsiders bidirectional search by proposing a new generic approach based on cluster computing. The paper shows that distributed bidirectional search using the proposed approach can solve faster and also can solve samples that are not solvable using sequential unidirectional or bidirectional searches.

The rest of the paper is organized as follows: In the next section some related work is discussed. Section 3 overviews distributed heuristic search and presents two generic algorithms to carry out distributed unidirectional and bidirectional heuristic search. Performance evaluation of the proposed algorithms is given in Section 4, followed by experimental results in Section 5. Finally, the paper is concluded with a recount of obtained results and some open questions.

2. Related work

Many bidirectional search algorithms have appeared in the literature including the two main traditional approaches, these are front-to-end and front-to-front searches. Both approaches encompass two algorithms searching in both directions. The difference between the two approaches is in the way heuristic values are calculated.

The front-to-end approach [7] employs heuristic evaluation functions that estimates the cost from a given node to the root of the opposite tree. Nodes are expanded alternately from the smaller tree until a common node n between the two search trees is found (the search continues from the upper tree when the two trees are equal in size). The cost of this node is $g_1(n) + g_2(n)$, where g_1 and g_2 are the cost from σ to n and the cost from τ to n; respectively. It should be noticed that even though the two sub-paths are optimal, the concatenated path is not necessarily optimal. Hence, this approach uses a special termination condition [7]. The obtained solution is optimal if the estimated cost $g_1(n) + g_2(n)$ is lower than $f_1(n_1)$ and $f_2(n_2)$ for any n_1 and n_2 in the open lists of the upward and downward searches; respectively. Here, f_1 and f_2 are similarly the evaluation functions for the two searches; the upward and downward searches, respectively.

At a first glance the approach might seem to suffer large overhead due to optimality test. Adding to it the misleading conjecture (the missile metaphor), the front-to-end approach has been neglected [15]. The induced optimality overhead can be simply skipped for faster solutions; however the misleading missile metaphor conjecture lead to proposing amendments to steer the search frontiers towards each other. The front-to-front approach is one alternative.

The front-to-front approach uses *wave-shaping* [4] to force the search fronts to meet. Examples of algorithms are BHFFA [4], BHFFA2 [3], and d-node retargeting [16] which can be efficient in terms of the number of generated nodes. Yet, these algorithms are computationally demanding as they have to compare the heuristic estimates between all nodes in one search frontier against all nodes in the other, which results in huge computational overhead. The time needed to select a node for expansion is proportional to the product of the sizes of the two search trees; which is a very time consuming activity though it can be reduced to the size of the opposite tree with the use of appropriate data structures, yet still high overhead.

In order to cope with the large node selection overhead, a number of techniques have been proposed in the literature to restrict comparison to only a small number of "promising" nodes in the opposite tree. Premier search [11] is one example which showed reasonable improvement on node selection time while maintaining solution quality. The idea is to store only one search frontier, instead of both frontiers, so the opposing search targets this frontier (the perimeter) as a more visible goal with more accurate heuristic estimate (recall that the perimeter nodes are in the middle way between a node in the opposing search tree and the actual goal).

This paper's approach is based on this improved bidirectional search. First, two perimeters are generated in parallel, one surrounding the goal node and the other surrounding the start node. Heuristic values in the subsequent stages are calculated from a search frontier to the opposite perimeter. Before we go into the details of the proposed distributed approach we will discuss some of the known distributed heuristic search approaches that have been discussed in the literature.

Due to the wide applicability and to the computation-intensive nature of heuristic search, distributed computing approaches have been investigated thoroughly [5] in an attempt to benefit from today's high performance parallel computing. A common formulation of parallel heuristic search is called *distributed tree search* [5], where the search space is partitioned into "disjoint" portions that are then distributed across individual processors. Each processor works on its portion of the search space independently or in coordination with other processors until a goal is found and a certain solution quality test is met.

Another approach, referred to as *parallel window search* [17], well-suited for parallel depth-first search. This approach distributes the iteration space to the available processors, and each processor then searches its iteration sub-space independently.

The two approaches (parallel window search and distributed tree search) can be combined so each iteration of the a parallel window search is completed in distributed tree search manner, or each part of the tree in distributed tree search is completed in parallel window search manner. This combined approach was tested on Sequent Balance 21000, Intel iPSC Hypercube, and BBN Butterfly using the 15-puzzle and shown to achieve reasonable speed up [10, 18].

The first approach (distributed tree search) is more suitable for parallelizing bidirectional search. Hence, we concentrate on this approach in developing distributed unidirectional and bidirectional algorithms well-suited for cluster computing. We argue that distributed tree search with perimeter based-bidirectional search is a viable alternative that outperforms its traditional approaches. In the subsequent sections we present and support this claim.

3. Distributed heuristic search

In this section we present two generic approaches for distributed heuristic search, one for unidirectional and another for bidirectional heuristic search. In both approaches, hosts perform independent searches beginning with distinct start nodes and searching through their sub-trees until one of the hosts encounters a goal node. Hosts might replicate each others' work, however the associated high communication cost that would be needed to avoid search replication is prohibitive. Eliminating replicas in the search spaces requires that all hosts have access to each others' search spaces, an extremely costly scenario in distributed environments, though hashing and transition tables might seem practical solutions. Hence, in the two approaches for distributed heuristic search that will be presented next, replicated sub-search is partially overlooked in favor of lower communication costs. The effect of this on the solution quality will also be measured.

3.1. Distributed unidirectional heuristic search

The distributed unidirectional heuristic search proposed in this paper can be characterized by the 5-tuple $\langle \Sigma, \tau, \xi, f, A \rangle$, where $\Sigma = \{n_1, n_2, \ldots, n_u\}$ is the set of u start nodes and A is a sequential heuristic search algorithm used to carry out the independent parallel searches. Figure 3 illustrates a generic approach for realizing unidirectional heuristic search in a distributed manner; this approach is referred to by *Uni_DHS*.

The available hosts in Uni_DHS generate/pick their share from Σ in a verity of mapping methods. We may have $e \geq 1$ hosts from the available p hosts perform breadth-first searches

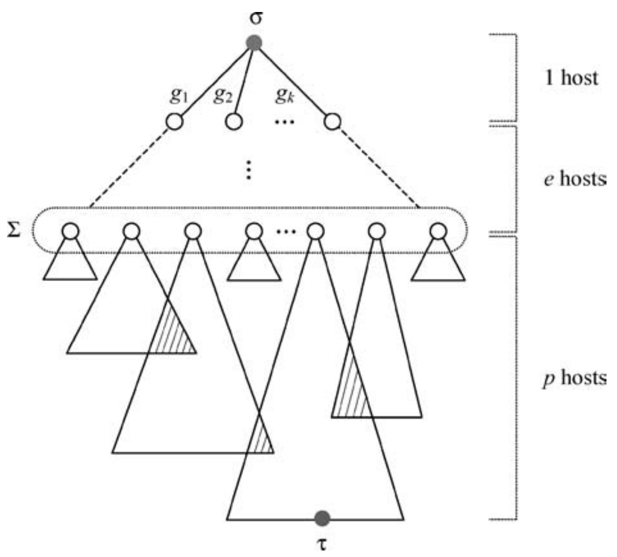

Figure 3. Diagrammatical representation of Uni_DHS.

to generate Σ and then broadcast subsets from Σ to the relevant hosts. The value e is decided empirically and depends on the network speed. Low speed networks perform better with low e values in favor of reduced communication overhead. Generally, the mapping can be expressed by the unified function

$$\rho : \prod \rightarrow 2^{\Sigma}$$

where $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$ is the set of p hosts in the cluster. The function $\rho(\pi_i)$ identifies the subset of start nodes from Σ assigned to the host π_i, $1 \leq i \leq p$. This unified mapping covers a wide range of load distribution methods that can be used in Uni_DHS as well as the bidirectional approach that will be presented next. For example, an even distribution method can be represented by $\rho(\pi_i) = \{n_j | (i-1)\lceil u/p \rceil < j \leq \min(i\lceil u/p \rceil, u)\}$, $1 \leq i \leq p$ and $u = |\Sigma|$. This instance of ρ achieves reasonably balanced load while maintaining minimum replicated work as the adjacent start nodes are likely to generate overlapping search spaces. Hence assigning these adjacent start nodes to the same host would minimize the overall search replication as far as the 15-puzzle is concerned. It should be noticed that this might not be the case in other problem domains.

In Uni_DHS each host maintains its open and closed lists independently. Once a host encounters a goal node, it broadcasts this node and signals a termination request. At this point, if the solution path is acceptable then all hosts terminate their searches. A sufficient condition for an optimal solution path up to a goal node τ is that $f(\tau)$ is the best among all nodes in all open lists in the independent searches. This requires all hosts compare the found goal with their local open lists to see if better nodes exist. If this is the case then the search should continue until an optimal solution is found. This process for ensuring the quality of the obtained solutions can be achieved through two distributed tasks; namely *termination-request* ($\mathcal{T}_{req}$) and *termination-reply* ($\mathcal{T}_{rep}$). These tasks will be explained in the sequel.

In order to simplify the presentation of the algorithm Uni_DHS, we will use the following notation: $\beta[\Delta, \lambda]$ denotes broadcasting the data set Δ to a subset of hosts λ from Π, $\alpha[\Delta, i]$ denotes receiving the data set Δ broadcasted by the host π_i, and $\Gamma[n, \tau, \xi, f, A]$ denotes performing $\langle n, \tau, \xi, f \rangle$ using the algorithm A. Using this notation the two tasks $\mathcal{T}_{req}$ and $\mathcal{T}_{rep}$ are outlined in Figures 4 and 5 respectively.

The above algorithms for termination request-and-reply sequence use the notation $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$ to denote the search space generated by the host π_k

```
Task $\mathcal{T}_{req}[k, n, f(n)]$
    … a termination request issued by $\pi_k$.
    initiate $\beta[\{k, n, f(n)\}, \Pi]$ then wait until all termination responses are received.
    if any of the received responses is denial, then
        signal termination denial to all hosts in $\Pi$.
        resume $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$.
    else
        signal termination approval to all hosts in $\Pi$.
End $\mathcal{T}_{req}$
```

Figure 4. Distributed termination-request.

```
Task $\mathcal{T}_{rep}[k, i]$
    ... a termination reply executed by $\pi_k$ upon request from $\pi_i$.
    perform $\alpha[\{n, f(n)\}, i]$.
    if $f(n)$ is optimal on the space of $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$, then
        signal a termination approval back to the host $\pi_i$.
        wait until the termination status is received from the host $\pi_i$.
        if termination denial, resume $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$; otherwise exit.
    else
        signal back a termination denial to the host $\pi_i$.
        resume $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$.
End Task $\mathcal{T}_{rep}$
```

Figure 5. Distributed termination-reply.

starting with $\rho(\pi_k)$. Even though this notation suggests that elements of $\rho(\pi_k)$ originate disjoint search trees, a practical implementation would add $\rho(\pi_k)$ to a common open list and consequently the host π_k would maintain only one common replication-free search space.

The algorithm Uni_DHS which uses the termination request-and-reply sequence is given in Figure 6. The algorithm initially starts with an open list containing $\rho(\pi_k)$ and then progressively searches using the search algorithm A until one of the two events happen: a goal node is encountered or a termination request is received.

Signaling termination requests and responds in the algorithm Uni_DHS can be achieved through interrupts or periodic tests once per iteration of $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$. An iteration involves selecting and then expanding a node, which could be too soon for another termination request-and-reply sequence; hence a threshold on the number of iterations might be used. Note that interrupts or periodic tests may not work for other types of problem domains; yet work fine in the 15-puzzle.

Using the above described termination request-and-reply sequence it can be easily verified that Uni_DHS terminates with an optimal solution if there is one. An inherently parallel version of the Uni_DHS can be obtained if the optimality concerns are disregarded. Obviously only empirical knowledge can tell which of these versions is the most effective in terms of performance/solution-quality trade-off assessment. This will be investigated in the coming sections.

```
Algorithm Uni_DHS
    ... executed by host $\pi_k$, $1 \le k \le p$.
    initiate $\Gamma[m, \tau, \xi, f, A]$ for all $m \in \rho(\pi_k)$ until either
      -  a goal node is found, or
      -  a termination request is received from another host.
    if a goal node, say $n$, is found, then
        execute $\mathcal{T}_{req}[k, n, f(n)]$.
    else if a termination request is received from another host, say $\pi_i$, then
        execute $\mathcal{T}_{rep}[k, i]$.
End Uni_DHS.
```

Figure 6. Distributed unidirectional heuristic search algorithm.

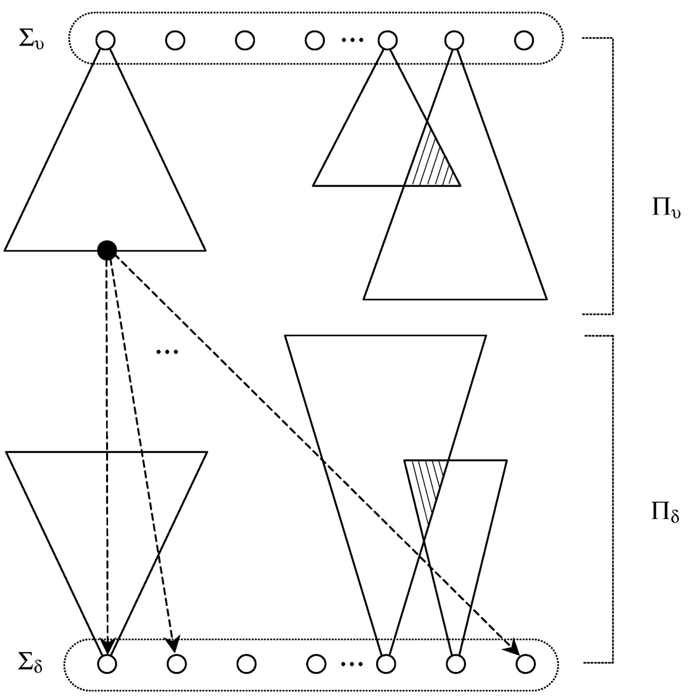

Figure 7. Diagrammatical representation of Bi_DHS.

3.2. *Distributed bidirectional heuristic search*

The distributed bidirectional heuristic search algorithm, abbreviated *Bi_DHS*, consists of two sets of distributed unidirectional heuristic searches running in opposite directions. These are downward and upward searches. The distributed bidirectional heuristic search can be characterized by the 6-tuple $(\Sigma_\upsilon, \Sigma_\delta, \sigma, \tau, \xi, f, A)$, where $\Sigma_\upsilon = \{n_1, n_2, \ldots, n_u\}$ and $\Sigma_\delta = \{m_1, m_2, \ldots, m_d\}$ are the set of start nodes in upward-search and downward-search, respectively. These sets are used as starting nodes for distributed heuristic search in each direction and also as perimeters [11] for the opposite direction to which the search is steered. Figure 7 illustrates a generic approach for carrying out bidirectional heuristic search in a distributed manner.

The major differences between Uni_DHS and Bi_DHS are related to the way the function f is computed, the termination condition, and the quality of the obtained solutions. In calculating $f(n)$ for some newly generated node n, two exact (g_1 and g_2 below) and one accurate estimates are used. A host in the downward-search calculates $f_1(n)$ for any node n in as follows:

$$f_1(n) = g_1(n) + \min_{m \in \sum_\delta} (h(n, m) + g_2(m)) \tag{1}$$

```
Algorithm Bi_DHS
    ... downward-search executed by the host $\pi_k$ in $\Pi_\upsilon$.
    execute $\Gamma[v, \tau, \xi, f, A]$ for all $v \in \rho(\pi_k)$ until one of the following events takes place
      - a node in $\Sigma_\delta$ is encountered,
      - a promising node is found, or
      - a communication request is received from a host $\pi_i$ in $\Pi_\delta$.
    if a node $n \in \Sigma_\delta$ is found, then
        execute $\mathcal{T}_{req}[k, n, f(n)]$.
    else if a goal node is being sent by a host $\pi_i$ in $\Pi_\delta$, then
        execute $\mathcal{T}_{rep}[k, i]$.
    else if a promising node $m$ is found in the space generated by $\Gamma[v, \tau, \xi, f, A]$ for some
    $v \in \rho(\pi_k)$, then
        perform $\beta[\{m\}, \Pi_\delta]$
    else if a promising node $m$ from a host $\pi_i$ in $\Pi_\delta$ is being communicated such that $m$ is
    in the search space generated by $\Gamma[v, \tau, \xi, f, A]$ for some $v \in \rho(\pi_k)$, then
        perform $\alpha[\{m\}, i]$.
        execute $\mathcal{T}_{req}[k, m, f(m)]$.
End Bi_DHS.
```

Figure 8. Distributed bidirectional heuristic search algorithm.

and a host in the upward-search calculates $f_2(n)$ any node n in as follows:

$$f_2(n) = g_2(n) + \min_{m \in \sum_\upsilon} (h(n, m) + g_1(m)). \tag{2}$$

Here we extend the functions h to include the relative node from which the cost is calculated, e.g. the cost $h(n, m)$ is the estimated distance from n to m. Furthermore we assume that the same heuristic function will be used in both searches; yet this restriction is needless as different heuristics can be used. Of course admissibility [14] in both directions is a must to ensure optimal solutions.

The downward-search in the algorithm Bi_DHS presented in Figure 8 executes on the p hosts, one copy per host (the upward-search is similar and is skipped for brevity). The p hosts are divided into two groups one works top-down $\Pi_\upsilon = \{\pi_1, \pi_2, \dots, \pi_{\lceil p/2 \rceil}\}$ and the other works bottom-up $\Pi_\delta = \{\pi_{\lceil p/2 \rceil+1}, \pi_{\lceil p/2 \rceil+2}, \dots, \pi_p\}$.

In traditional bidirectional search methods, the two searches (upward-search and downward-search) have access to each others' search spaces, hence termination conditions can be evaluated in a straightforward manner: once a common node is encountered in the two searches, then a solution is found. However, in Bi_DHS hosts have no access to each others' search spaces, thus traditional termination tests are unachievable in distributed bidirectional heuristic search.

The termination condition in the algorithm Bi_DHS is based on the set Σ_δ (respectively Σ_υ) and a subset of nodes received from Π_δ(respectively Π_υ) in the upward-search (respectively downward-search). For instance in the upward-search, if a node in Σ_δ is encountered or a node received from a host in Π_δ that already exists in the upward-search, then a solution path is obtained. This path is optimal if no other node in both the upward and the downward searches has better f-value. Any host in Π_υ can broadcast to all nodes in Π_δ promising nodes it comes across. Of course, a threshold based on the evaluation function f can used to identify promising nodes that should be sent to the opposite tree. Similar processing will take place in the downward-search.

For a reasonable load balancing, the size of the two sets Σ_υ and Σ_δ should be multiple integrals of the number of hosts in the relevant direction. Again, the function ρ can be used to distribute the set of start nodes (the perimeters) to the hosts. A tree-based parallel model will be sufficient to generate and distribute the two perimeters. In this model, duplicate nodes may appear which may cause redundant search efforts, but this is inconsequential when large number of hosts is available. The empirical results presented in the next section confirm this observation.

4. Framework for assessing the performance of Uni_DHS and Bi_DHS

In this section we present a framework for assessing the performance of the proposed algorithms. In this assessment we focus on four fundamental measures, these measures are *execution time*, *memory space*, *solution quality* and *search effectiveness*. The execution time is measured by the actual execution time taken until a goal is found and a termination request succeeds. In this section all termination requests are acceptable which means the termination request-and-reply sequence is relaxed and hence the first obtained solution will abort the search. The effect of this relaxation will be exposed on the solution quality as we will see next.

The execution time can be estimated in terms of average counts on the major search activities such as *node expansion*, *heuristic distance calculation*, *node duplicate checking*, and so on (these parameters can also be used to give indication on the search effectiveness as well). This method has been used in [1] to develop machine-independent performance assessment models for unidirectional and bidirectional sequential heuristic search algorithms. These models are partially applicable in assessing the performance of Uni_DHS and Bi_DHS with the exception that communication overhead has to be accounted for. The next section portrays the average counts on the major activities performed by the two algorithms Uni_DHS and Bi_DHS.

The memory space can be solely expressed by the number of nodes generated by the algorithm until a goal node is found. In parallel search, there are two ways to record this count: *average* and *accumulate* number of nodes generated by hosts in the cluster. The difference between these two can easily be noticed if we think of the cluster as a single computer with multiple processing units or as separate machines with single processors. The latter image is a common one and hence the information in all of the subsequent figures are based on the averages.

The solution quality is quantified by the ratio of the actual and the obtained solution path lengths [2]. Hence, the term $(w - l)/l$ shows the increase in the length of the solution path as a percentage of the optimal length, where w is the length of the obtained solution path and l is the optimal length. This percentage is close to zero when the length of the obtained solution path is close to the optimal length.

A better indication on the search quality should take into account the amount of search effort spent on obtaining a solution path. One such measure is the search effectiveness [2], which is given by $b \cdot w/c$ where b is the average branching factor of the search graph and c

is the size of the generated search space. The search effectiveness gives indication on how much the search effort was directed towards the goal. Larger ratios mean more focused search.

The above measures depend basically on the following factors: the *problem domain*, the *underlying machine* and the employed *heuristic functions*. The answer to the obvious question “how effective was the search to find a solution of this quality?” can be provided if we have correct estimates on b, w, and c. Such values for a given heuristic search algorithm can be obtained by finding averages over all possible problems in the problem domain. Experimental results on this issue are presented in the next section.

5. Experimental results on the 15-puzzle

The two distributed search approaches (Uni_DHS and Bi_DHS) have been tested on a cluster of 16 dual processor Intel Xeon with 512 MB RAM hosts hocked together by two types of Gigabit networks: Ethernet and Myrinet.

The problem domain is the 15-puzzle [9] which is still computationally intensive problem for the most powerful computing systems. The heuristic function used in the experiments is called *Linear-conflict* [6]. Each of the two algorithms has been tested against 528 samples selected from the 15-puzzle space with optimal solution paths ranging from 20 to 74 moves. The samples are randomly selected from the space with 10 samples at depth 20, 10 at depth 21, and so on. Some samples at large depth exhausted the physical memory of the cluster and hence they were excluded from the experiments.

On each sample, the algorithms are executed until a solution is found or the physical memory is consumed in one of the hosts, triggering an automatic self-termination. Upon successful completion, the raw data are automatically collected and recorded in the associated databases. In what follows we summarize the obtained results.

The actual execution times for the two approaches using 4, 8 and 16 hosts are shown in Figures 9–12. The four figures show execution times for disjoints sets of samples, ranging from easy to very hard. Figure 9 suggests that increasing the number of hosts searching on easy problems does not necessarily speedup the search. This is expected because, for easy problems, one host finds the solution rather quickly and even before other hosts have had a chance to start searching. This is true for both Uni_DHS and Bi_DHS. In other words, both algorithms become more of a sequential type as the samples become easier to solve.

This observation discontinues for middle and hard problems (Figures 10 and 11) in both Uni_DHS and Bi_DHS and again the same trend continues for some samples in the class of very hard problems, however, only in Bi_DHS this time (Figure 12). The reason behind this could be the increase in the number of Ethernet collisions. We believe that the overall delay introduced due to Ethernet collisions outweighs the expected decrease in execution time.

In this experiment timers are used to control the rate of promising node transmission so that the rate decreases as the number of hosts increases. The reason behind the increase in the number of Ethernet collisions as the number of hosts increases is not due to the increase in the number of promising nodes exchanged. Rather, its due to the more likelihood of timers

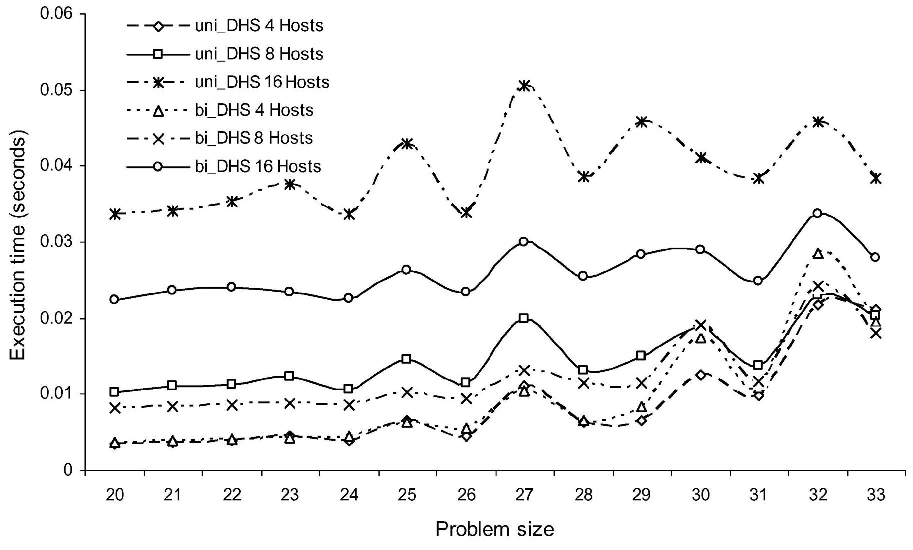

Figure 9. Average execution time on easy samples.

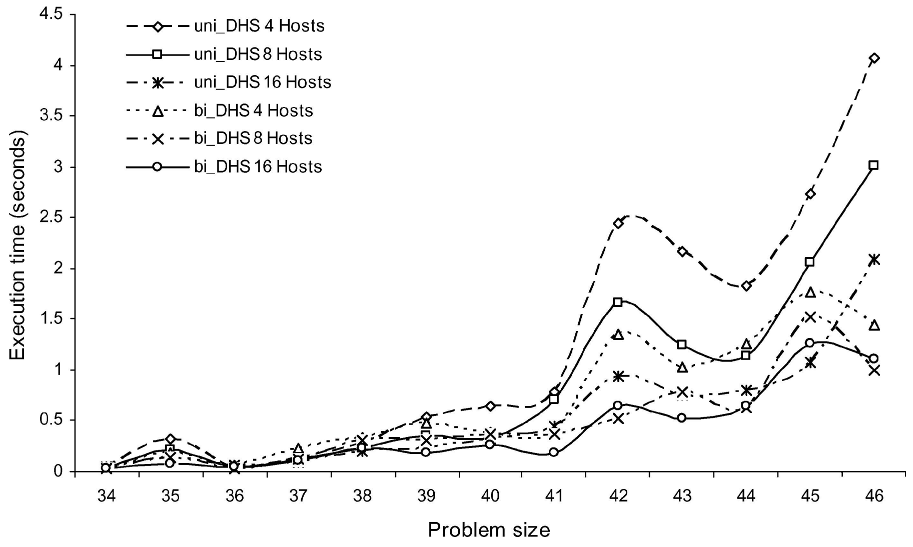

Figure 10. Average execution time on middle samples.

being synchronized enough that many hosts try to send their promising nodes at almost the same instant. An algorithm to ensure that two hosts do not try to send a promising node at the same time is needed. However, this is difficult to achieve especially since the time at which hosts start searching is out of control. As hosts typically send promising nodes every

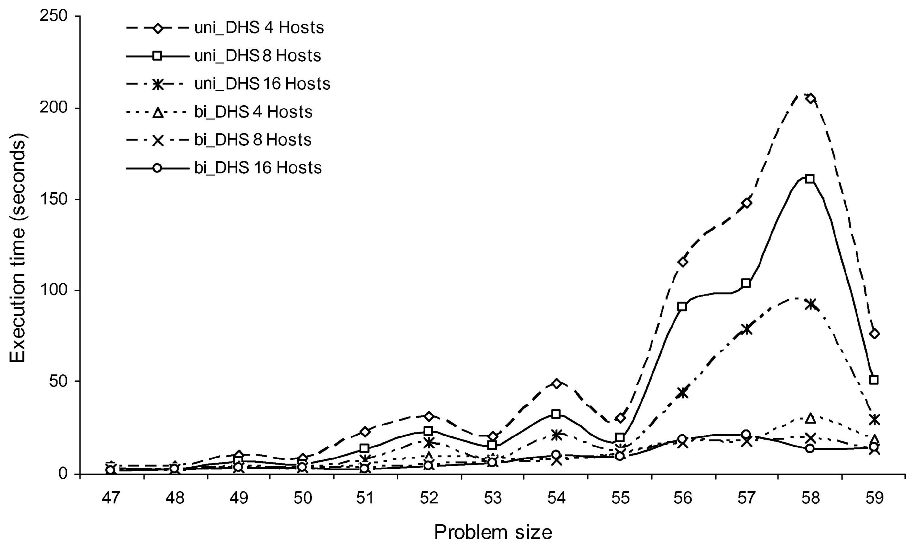

Figure 11. Average execution time on hard samples.

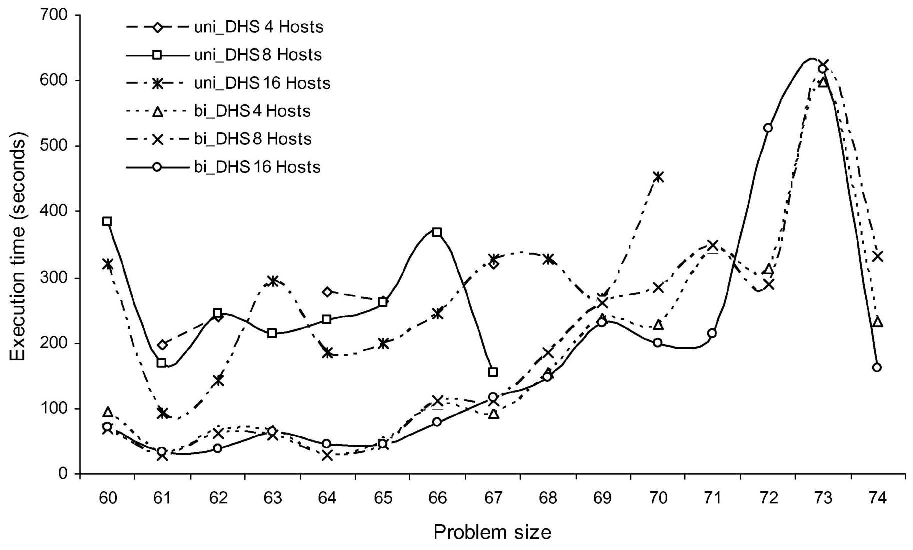

Figure 12. Average execution on very hard samples.

tens of milliseconds or so, it is then difficult to devise a synchronization algorithm that can synchronize timers within that resolution, especially since when we only have access to the Ethernet at the MPI level only. The Bi_DHS has been tested more than once for some samples and the execution time fluctuated wildly. This supports the above explanation.

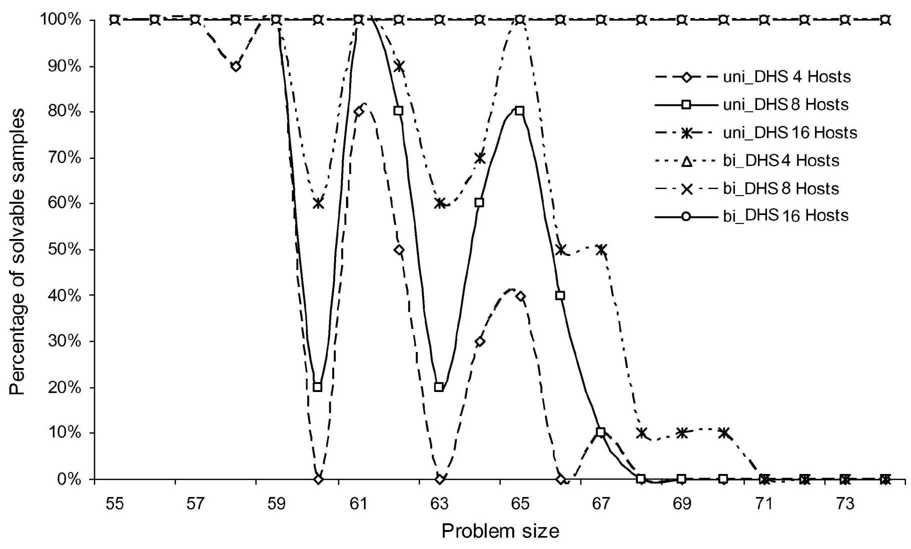

Figure 13. Percentages of solvable samples (hard to very hard categories).

The discontinuous plotting in Figure 12 are due to the samples that could not be solved using Uni_DHS before the physical memory is consumed. Bi_DHS was able to solve all samples using 4, 8, or 16 hosts. Figure 13 shows the percentages of samples solved from hard to very hard categories. Uni_DHS was able to solve small percentage of the hard and very hard samples. This fact has been taken into consideration when averaging samples for comparisons. The parameters for unsolvable samples are excluded from the averages in order to avoid ambiguous interpretations.

The gained speedup for the two approaches is shown in Figure 14. Both algorithms run much faster than their sequential counterparts. There are sample categories where Bi_DHS runs over 30 times faster using 16 hosts and 20 times faster using only 8 hosts, which is not common in parallel computing terms. This observation supports the assertion that wider perimeter might end up in a faster search. Therefore, the gained speedup is ascribed not only to the larger number of hosts incorporated in the search but also to the induced wider perimeter. This assertion becomes more evident if we look at the speedup gain in Uni_DHS which is significantly less than Bi_DHS. Thus, confirming the claim made in this paper about the effectiveness of combining perimeter search with distributed bidirectional approaches. The discontinuous curves in Figure 14 are due to the fact that sequential unidirectional and bidirectional searches where unable to solve hard and very hard sample categories; and hence the speedup on these samples cannot be calculated.

The effect of relaxing quality control in both Uni_DHS and Bi_DHS is depicted in Figure 15. This figure shows the quality measure as discussed in the previous section. Uni_DHS offers higher quality solutions (a maximum of 2% loss) compared to Bi_DHS (a maximum of 14% loss). Yet another observation that merits being mentioned here is

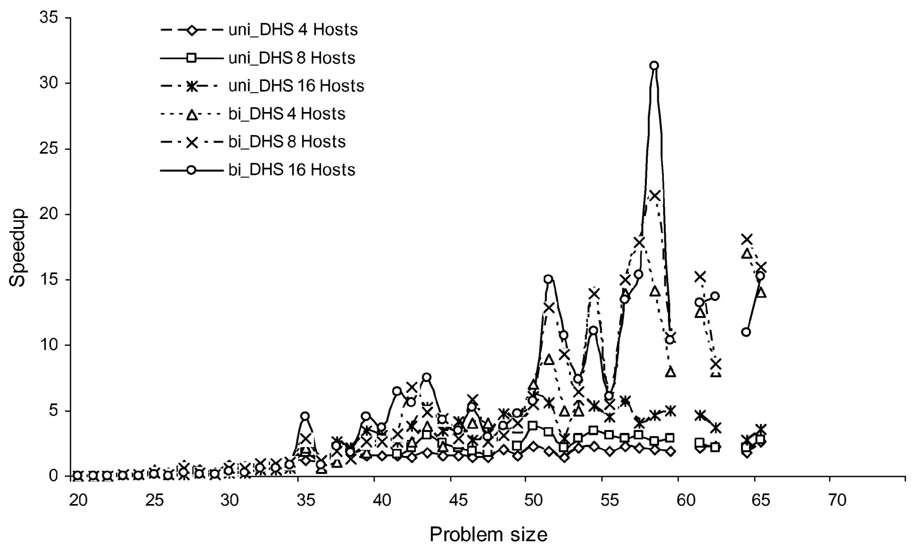

Figure 14. Speedup gain.

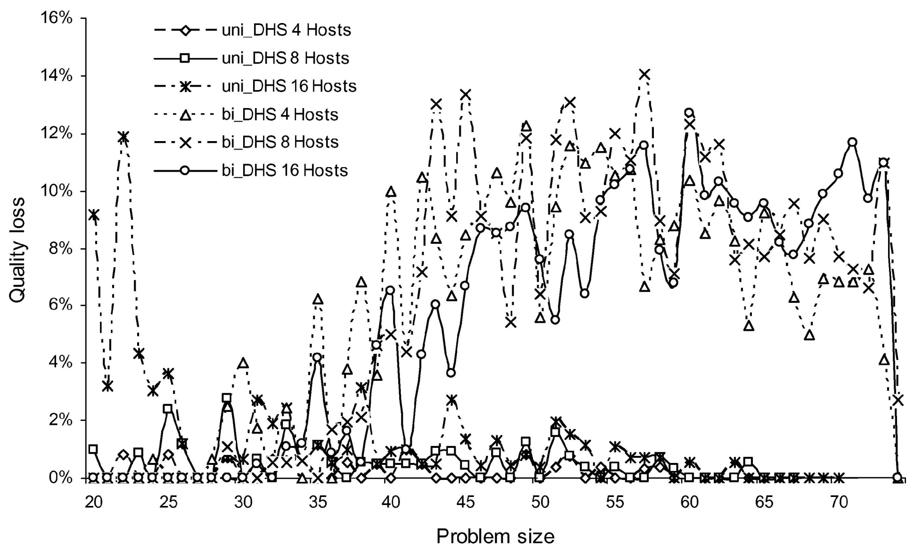

Figure 15. The effect of relaxing quality control.

that adding more hosts doesn't negatively affect the solution quality. This is clear in the figure with the exception of Uni_DHS using 16 hosts to solve easy problems. There are instances where the 16 hosts Bi_DHS obtains better solutions than on 4 or 8 hosts. The explanation for this behavior could be the wider perimeter when using larger number of hosts.

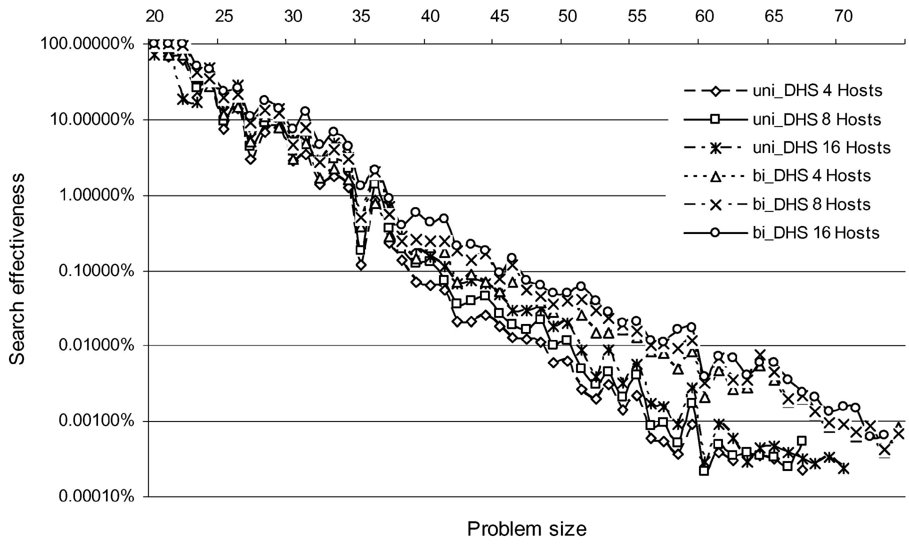

Figure 16. Search effectiveness.

Figure 16 gives indication on the search effectiveness for the two heuristic search approaches as defined in the previous section. The search effectiveness decreases for harder problems. The figure also suggests that Bi_DHS is more effective than Uni_DHS. This observation agrees with what is known about sequential unidirectional and bidirectional search approaches [1, 2].

The last set of findings is presented in Figures 17—20. In these figures we can see that Bi_DHS expands and generates lesser nodes than Uni_DHS. Of course all these qualities translate into faster execution and lower memory consumption.

The use of perimeters induces more emphasis on heuristic information. Bi_DHS has two perimeters towards which the two search frontiers are steered through repeated heuristic distance calculation. The information in Figure 20 demonstrate the fact that Bi_DHS consumes more time in assuring accurate heuristic estimates for each newly generated node. As it can be seen from the above results the consumed time paid off.

6. Epilogue

The two distributed search algorithms presented in this paper are based on cluster computing approaches. The first approach uses distributed tree search; while the second approach combines distributed tree search, perimeter search, and bidirectional search. The second approach provides faster and more effective search yet with lower solution quality than unidirectional approach. Table 1 summarizes the results of applying different performance assessment parameters on the two approaches. The information in the table is obtained by averaging all sample categories. These averages add little new

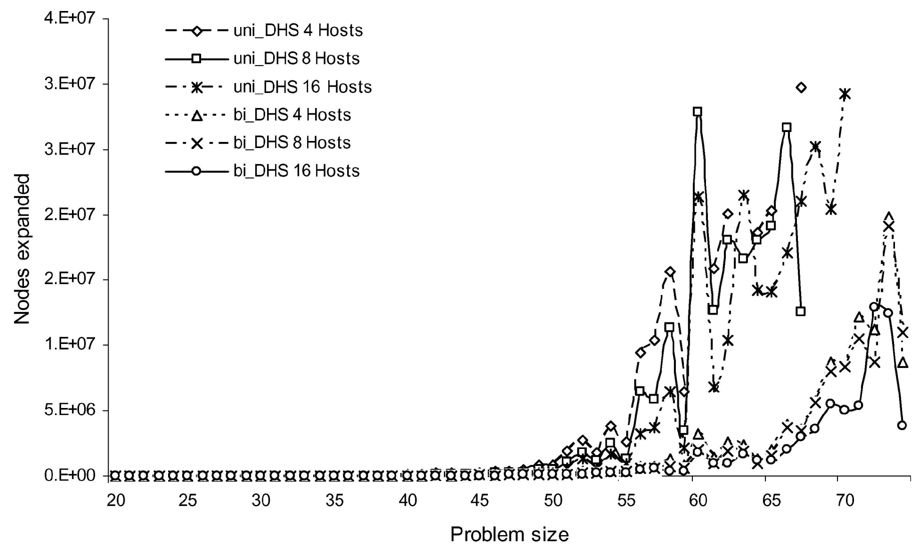

Figure 17. Average number of expanded nodes.

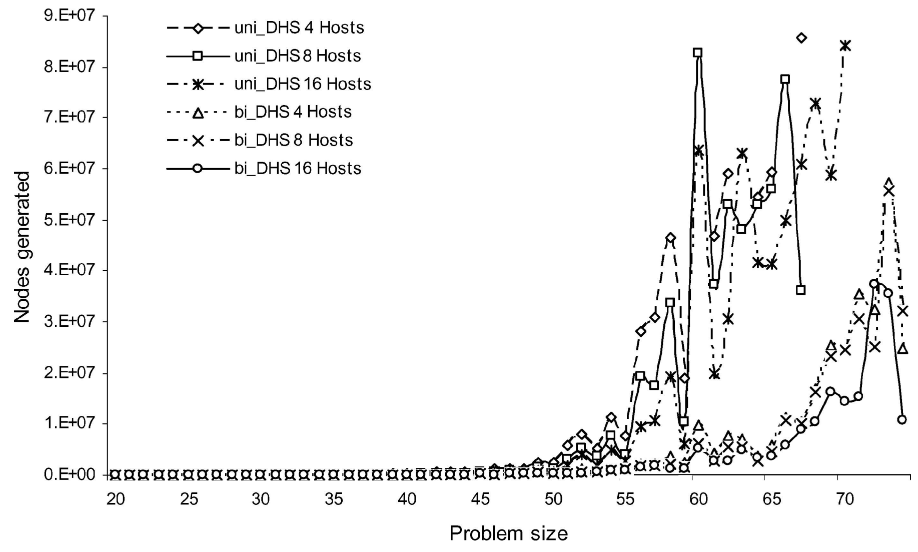

Figure 18. Average size of generated search space.

observations over the previous figures, yet they give an overview on the algorithms' behavior.

Some issues that need more investigation include determining the optimal number of start nodes each host should get. This relates to developing smart instances of the function ρ that distribute the set of start nodes so the overall replicated work is reduced. The reader

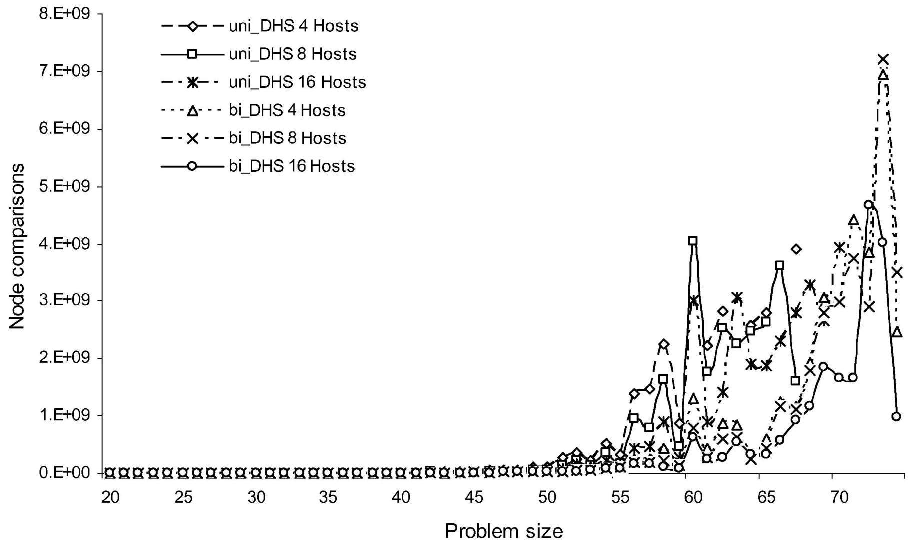

Figure 19. Average number of node duplicate checks.

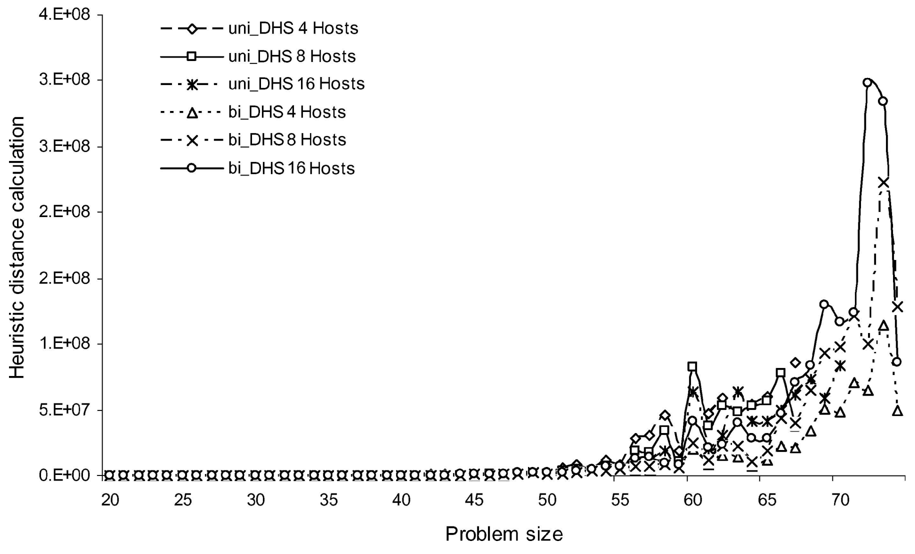

Figure 20. Average number of times heuristic distance is calculated.

should not be deceived by the information in the row "Replication percentage" in Table 1. These are percentages of replicas found in each host and not in the overall search (inter-node replication cannot be measured). The small percentages in all cases indicate that the huge amount or node duplicate checks (notice the large numbers in the node comparison

Table 1. Overall averages

	Uni_DHS			Bi_DHS		
Parameter	4 Hosts	8 Hosts	16 Hosts	4 Hosts	8 Hosts	16 Hosts
Execution time	18.59	13.47	8.28	3.68	3.05	2.94
Speed up	1.26	1.70	2.53	3.60	4.36	4.79
Node expansion	1,466,521	936,426	550,103	160,273	112,768	85,831
Node generation	4,368,415	2,790,478	1,640,090	478,695	336,947	256,209
Heuristic distance	4,368,415	2,790,478	1,640,090	957,390	1,347,788	2,049,673
Node comparison	204,221,411	127,759,932	71,621,193	48,249,260	31,599,130	23,031,989
Duplicate nodes found	1,725,778	1,096,411	637,774	181,302	126,643	95,800
Replication percentage	0.85%	0.86%	0.89%	0.38%	0.40%	0.42%
Obtained path length	40	40	40	42	42	42
Quality loss	0.24%	0.60%	1.74%	5.42%	5.07%	4.07%
Search effectiveness	8.27%	10.05%	8.56%	9.33%	11.88%	13.32%
Solvable hard samples	35%	46%	56%	100%	100%	100%

row) might be needless. Of course overlooking duplicates might lead to higher loss in the solution quality. Again this area needs more empirical knowledge.

Finally, identifying promising nodes and the frequency of broadcasting them need additional heuristic knowledge that depends on the characteristics of the hardware, the communication network, and the number of hosts participating in the search.

7. Future work

The major limiting factor on the performance of Uni_DHS and Bi_DHS is the inter-node replicated search efforts. One obvious way to lessen the amount of replicated work is to have the hosts send nodes to each other. However, the performance gains using this approach are limited at best. Additionally, this approach undoubtedly introduces an overhead that may backfire and reduce the performance.

A less obvious way is a clever distribution of the start nodes. Future work includes experimentation using different distribution functions in order to reduce replicated search efforts. The distribution functions obviously depend on the employed heuristic function.

The more the amount of replicated work is minimized in the early stages of the search, the lesser it will be in later stages of the search. Therefore, a technique for avoiding replicated work in the early stages of the search may be advantageous, even if it were a costly one. This is especially true for hard samples. One such technique is to have each host run a number of best-first searches after the breadth-first search executes and before the main best-first search is executed. The best-first searches are used to prune the paths that will result in replicated work in the early stages of the search. For a host π_i to prune paths that will result in replicating the work of another host π_j it must run a best-first search using the start nodes of host π_j to determine which nodes should be put on its closed list before the main best-first search is executed.

Many questions must be answered before this technique can be fully implemented, including: How many best-first searches should each host execute and for which hosts? What is the strategy used to terminate the best-first searches and how does it affect the solution quality?

Acknowledgments

The author would like to the thank the anonymous referees for their comments and suggestions which improved the paper. Many thanks also go to Nayef Abu-Ghazaleh and to the Department of Computer Science at State University of New York at Binghamton for allowing him do the tests on their cluster.

References

1. A. Al-Ayyoub and F. Masoud. Heuristic search revisited. *Journal of Systems and Software*, 55(2):103–113, 2000.
2. A. Al-Ayyoub and F. Masoud. Search quality and effectiveness for intelligent systems. In *Proceedings of the* 8th *International Conference on Intelligent Systems*, Denver, Colorado, USA, pp. 146–149, June 24–26, 1999.
3. D. DeChampeaux. Bidirectional heuristic search again. *Journal of the ACM*, 30(1): 22–32, 1983.
4. D. DeChampeaux and L. Sint. An improved bidirectional heuristic search algorithm. *Journal of the ACM*, 24(2):177–191, 1977.
5. A. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization. Technical Report Number 93-11. Department of Computer Science and Engineering, The University of Minnesota, 1993.
6. O. Hansson, A. Mayer, and M. Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Science*, 63(3):207–227, 1992.
7. H. Kaindl and G. Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.
8. R. Krof. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
9. R. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
10. V. Kumar and V. Rao. Scalable parallel formulation of depth-first search. In V. Kumar, P. Gopalakishnan, and L. Kanal, eds. *Parallel Algorithms for Machine Intelligence and Vision*, Springer-Verlag, New York, 1990.
11. G. Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.
12. N. Nadal. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, eds. *Search in Artificial Intelligence*, Springer-Verlag, New York, 1988.
13. N. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, 1980.
14. J. Pearl. *Heuristic-Intelligence Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading MA, 1984.
15. I. Phol. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.
16. G. Politowski and I. Pohl. D-node Retargeting in Bidirectional Heuristic Search. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, Menlo Park, CA, pp. 274–277, 1984.
17. C. Powley and R. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI–13(5):466–477,1991.
18. V. Rao and V. Kumar. Parallel depth-first search, Part I: implementation. *International Journal of Parallel Programming*, 16(6):479–499, 1987.