# PRA*: Massively Parallel Heuristic Search

Matthew Evett* James Hendler† Ambuj Mahanti‡ Dana Nau§

University of Maryland

**Abstract**

In this paper we describe a variant of A* search designed to run on the massively parallel, SIMD Connection Machine (CM-2). The algorithm is designed to run in a limited memory by use of a retraction technique which allows nodes with poor heuristic values to be removed from the open list, until such time as they may need reexpansion, more promising paths having failed. Our algorithm, called PRA* (for Parallel Retraction A*), is designed to maximize use of the Connection Machine's memory and processors. In addition, the algorithm is guaranteed to return an optimal path when an admissable heuristic is used. Results comparing PRA* to Korf's IDA* for the fifteen-puzzle show significantly fewer node expansions for PRA*. In addition, empirical results show significant parallel speedups, indicative of the algorithm's design for high processor utilization.

Please address correspondence to:

James Hendler
Computer Science Dept.
University of Maryland
College Park, MD 20742

*Institute for Advanced Computer Studies and Computer Science Department. E-mail: evett@cs.umd.edu.

†Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies. E-mail: hendler@cs.umd.edu

‡Systems Research Center, Computer Science Department, and Institute for Advanced Computer Studies. E-mail: am@cs.umd.edu.

§Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies. E-mail: nau@cs.umd.edu.

# 1 Introduction

The use of search is ubiquitous in the field of computer science. Problems such as computing a traveling salesman tour, playing chess, solving the Tower of Hanoi problem, etc., can, in principle, be solved by examining the nodes in a large search space. A major contribution of research in the field of artificial intelligence has been the design of *heuristic* search programs – programs that take advantage of informed guesses to guide the search. One of the best known examples of the use of this technique is the solution of the "15-puzzle," a sliding blocks problem consisting of fifteen numbered, movable tiles set in a $4 \times 4$ frame. One cell of the frame is always empty, thus making it possible to move an adjacent numbered tile into the empty cell. The problem is to find a sequence (preferably optimal) of tile moves that will transpose a given initial board position into a given goal position.

Heuristic search focuses on using information about the problem to guide the search. In the fifteen puzzle, for example, a well known heuristic can be used to estimate the maximum number of moves needed to get to a known goal from any board. Using this estimator, search techniques can be designed that will produce an optimal solution, while searching only a small portion of the search space. Among the best known of these techniques are A* [18] and several of its variants [1, 17, 18, 19]. These algorithms use a combination of the number of moves used in known partial solutions, added to the estimated distance (the *heuristic*) from each partial solution to the target, to estimate how "promising" each partial solution is. Paths are searched in order of this value; the most promising paths searched first. In addition, the A* algorithm is guaranteed to return the optimal path when the heuristic is "admissable" (that is, when the heuristic consistently underestimates the cost of reaching the goal). Even using heuristics, the size of the search space can be prohibitively large for many problems. For example, the number of search states in the 15 puzzle is $15!/2 > 2^{39}$. Even with heuristic guidance, A*'s search may require examining many millions, even billions, of board positions.

Given the large number of board positions that need to be searched, this problem (and others like it) would seem to be an excellent candidate for massive parallelism – the problem would appear to be "embarrassingly" parallel. However, a major problem in producing a parallel version of a heuristic search program is that in most applications of search, the storage requirements of most of the well known heuristic algorithms grow exponentially with some measure of the size of the problem. Thus, even using the large memories available across the many processors of massively parallel machines, it is still not feasible to solve many interesting and useful problems. In A* and its variants, for example, the entire explicitly generated search space must be stored prior to termination. To circumvent the large storage requirement of A*-like algorithms, attempts have been made to design algorithms that can run within limited available memory and yet produce optimal or near optimal solutions. Unfortunately, most of these efforts have produced algorithms that are not amenable to massively parallel implementation.

In this paper, we discuss an algorithm called PRA* (for *Parallel Retracting A**), which was designed to run in a fixed-size memory and to exploit (SIMD) massive parallelism. We describe[1] the algorithm and its implementation on the massively parallel Connection

[1] A preliminary description of PRA*, without most of the results discussed in the current paper, appears in [5].

Machine (CM-2). We present empirical results showing that PRA* examines fewer nodes than both the best known (serial) limited memory algorithm (IDA* [9]) and a parallel implementation (IDPS) of that algorithm. We also demonstrate that PRA* exhibits a near-linear speedup with the addition of extra processors.

The rest of this paper is organized as follows:

1. Section 2 presents the details of RA* (a serial algorithm of which PRA* is a parallelization) and PRA*.[2]

2. Section 3 discusses how the algorithm was programmed on the CM-2.

3. Section 4 discusses other algorithms related to RA* and PRA*, including IDA*, MREC, MA*, IDPS, and P-IDA*.[3]

4. Section 5 presents experimental results for PRA*. It includes a comparison of PRA* to a parallel version of IDA*, and parallel speedup results for PRA*. Section 6 discusses these results and how they might apply to other search problems.

5. Section 7 contains concluding remarks.

# 2 The Algorithm

In this section we present the details of a memory limited algorithm, PRA*, which was designed to run on a SIMD architecture, and in particular, on the Connection Machine (CM-2)[4]. PRA* is able to examine a large number of nodes in parallel, while using a "retraction" scheme to free memory when needed. As we will show below, PRA* returns an optimal path if the selection heuristic is admissable.

## 2.1 Description of RA*

PRA* is a parallel implementation of a serial algorithm called RA* (Retracting A*). Although this paper is about the parallel algorithm, in order to understand PRA* it is best to first to look at RA*. Pseudocode for RA* is shown in Figure 1. Appendix A contains the outline of a correctness proof for RA*. The easiest way to explain RA* is in a comparison to A*. Below we summarize the similarities and differences between RA* and A*:

1. Like A*, RA* searches a state space that can be represented as a graph $G$ in which nodes represent states in the search, and arcs represent legal transitions between them. $G$ has a start node $s$ and one or more goal nodes. A *solution path* is a path starting at $s$ and ending at a goal node. Each arc has a cost, and the cost of a path is the sum of the costs of its arcs.

[2] In addition, Appendix A contains an outline of a proof of correctness for the RA* algorithm.

[3] In addition, Appendix B contains a detailed comparison of RA* with MA*.

[4] Several of the techniques in the PRA* implementation are specific to the SIMD aspects of the CM-2. Nonetheless, the results presented in this paper should extend to a SPMD implementation on the CM-5. Future work is needed to demonstrate how PRA*'s implementation might be modified to produce superior results on the CM-5.

```
procedure RA*(s,M)
    put s into T
    g(s) := 0
    q(s) := ∞
    h̄(s) := h(s)
    f(s) := g(s) + h̄(s)
    loop
        u := the expandable node for which f(u) is smallest,
            resolving ties in favor of the most recently generated node
        if u is a goal node then
            return the path (s,...,parent(parent(u)),parent(u),u)
        else for each child v of u that is not already a child of u in T do
            if v ∉ T then Create-and-Install-Child(u,v)
            else if g(u) + c(u,v) < g(v) then begin
                f(parent(v)) := g(parent(v)) + q(parent(v))
                Create-and-Install-Child(u,v)
            end
        if u is a tip node then f(u) := g(u) + q(u)
        while T contains more than M − b nodes do begin
            let v be the tip node of T for which f(v) is largest,
                resolving ties in favor of the least recently generated node
            Retract(v)
        end
    repeat
end RA*

procedure Create-and-Install-Child(u,v)
    put v into T
    parent(v) := u
    h̄(v) := max(h(v),h̄(u) − c(u,v))
    f(v) := g(v) + h̄(v)
    q(v) := ∞
end Create-and-Install-Child

procedure Retract(v)
    u := parent(v)
    q(u) := min q(u),h̄(v) + c(u,v)
    f(u) := g(u) + q(u)
    remove v from T
    if u is now a tip node then begin
        h̄(u) := max(q(u),h̄(u))
        f(u) := g(u) + h̄(u)
    end
end Retract
```

Figure 1: RA*

For example, in the 15-puzzle each configuration of the tiles is a node in the state space, and a node $p$ is connected to a node $q$ if and only if by sliding a single tile, the board represented by $p$ can be transformed into the board represented by $q$. The start node is the starting configuration of the tiles, and the goal node is the desired configuration of the tiles. In the 15-puzzle, each arc has a cost of 1, so the cost of a path is the same as its length.

2. Like A*, RA* *expands* each node $t$ by generating its children. But unlike A*, it may subsequently retract some of the generated nodes to free space. In RA*, a node $t$ is *expandable* if $t$ has been generated but has not been expanded, *or* if $t$ has been expanded but some of its children subsequently have been retracted. In the latter case, expanding $t$ consists of re-generating the retracted children.

3. Like A*, RA* requires a heuristic function $h(t)$ that returns a lower bound on the cost of getting from $t$ to the goal node. The definition of "cost" is domain-dependent. For the 15-puzzle, the cost is the number of tile moves from the starting position, $s$.

4. Like A*, if RA* finds more than one path to $t$, it retains only a least costly one, and maintains $g(t)$ as the cost of this path. The paths themselves are maintained by having each node point to its immediate predecessor in the path.

5. Unlike A*, RA* does not require that the entire examined search space be stored in memory. To cope with search spaces that are too large to be stored in memory, RA* uses a *retraction* mechanism. When memory is needed, frontier nodes of the search space are retracted, storing their $f$-values in their parents. If any of a node's children are retracted, that node becomes available for reexpansion, its $f$-value being set to the minimum of those of its retracted children.

6. Like A*, RA* maintains an estimated cost, $f(t)$, of the cost of getting from $s$ to $g$ through $t$. However, in RA*, the initial value for $f(t)$ is not $g(t) + h(t)$ but instead $\max(g(t) + h(t), f(\text{parent}(t)))$. Furthermore, if RA* ever retracts all of the children of $t$, it updates $f(t)$ to be the minimum of the $f$-values of the retracted children.

### 2.2 Description of PRA*

Pseudocode for the PRA* algorithm is shown in Figures 2 and 3. Below, we describe the primary differences between RA* and PRA*.

Rather than keeping track of the generated nodes globally as A* and RA* do, PRA* distributes these nodes over the local memory stores available to its processors. Each processor has a bucket capable of holding some fixed number of nodes (280 nodes per bucket in the current implementation). As each node, $t$, is generated at its parent node's processor, it is installed in the search graph by assigning it to a processor (and thus to that processor's bucket). This is done using a hashing function $H(t)$ that returns the address of the processor to which $t$ is assigned. The parent processor broadcasts $t$ to processor $H(t)$, which then installs the node in its bucket. Provided the hashing function is a good one, the nodes will be well distributed across all processors, and the corresponding processing load also will be well distributed. (The use of hashing functions for load balancing was

```
procedure PRA*(s)
    P := NIL
    U := ∞
    tell all processors to empty their buckets
    Initialize(s)
    loop
        for all buckets (i.e. processors) containing expandable nodes begin
            select an expandable node t with lowest f-value,
            resolving ties in favor of most recently installed
            if t is not the goal node then
                for each child v of t such that c_v(t) ∈ {RETRACTED,NOT-GENERATED} begin
                    Create-Child(t, v)
                    c_v(t) := GENERATED
                    Install(v)
                end
            else if g(t) < U then begin                          t is a goal node
                U := g(t)
                P := the path from s to t
            end
        end
        for every tip node t such that g(t) ≥ U, Prune(t)
        until the bucket is empty enough or it contains no tip nodes do begin
            let u be the tip node with the largest f-value in the bucket,
            resolving ties in favor of the least recently installed
            Retract(u)
        end
    repeat
end PRA*

procedure Initialize(s)
    parent(s) := NIL
    h̄(s) := h(s)
    g(s) := 0
    f(s) := h̄(s)
    q(s) := ∞
    c_i(s) := NOT-GENERATED, ∀i
    Install(s)
end Initialize
```

Figure 2: Pseudocode for the PRA* algorithm (Part 1).

```
procedure Create-Child(t,v)
    parent(v) := t
    h̄(v) := max(h(v), h̄(t) − c(t,v))
    g(v) := g(t) + c(t,v)
    f(v) := g(v) + h̄(v)                          Same as max(f(t), g(v) + h(v)).
    q(v) := ∞
end Create-Child

procedure Install(v, status)
    send v to processor H(v)                      Collisions are possible here
    for every processor receiving a v             Note context shift
        if the bucket is full then status := OUT-OF-MEMORY
        else if v is already in the bucket as v' then
            if g(v') ≤ g(v) then status := CHEAPER-PATH-EXISTS
            else begin                            New node offers cheaper path
                Prune(v')
                replace v' with v in bucket
            end
        send status to processor H(parent(v))
    if status = OUT-OF-MEMORY then                Returned to original context
        Retract(v)                                Retract and Prune are intra-processor here.
    else if status = CHEAPER-PATH-EXISTS then Prune(v)
end Install

procedure Retract(v)
    send v to processor H(parent(v))
    for every processor receiving v               Context shift
        let u = parent(v)                         Retrieve parent from bucket.
        q(u) := min(q(u), h̄(v) + c(u,v))
        set c_v(u) = RETRACTED
        if c_j(u) ∈ {RETRACTED, PRUNED}, ∀j        u has no other children.
            h̄(u) := max(q(u), h̄(u))
            f(u) = g(u) + h̄(u)
    remove v from bucket                          Returned to original context
end Retract

procedure Prune(v)
    send v to processor H(parent(v))
    for every processor receiving v               Context shift
                parent(v) is stored in current processor's bucket, and so is easily retrieved.
        c_v(parent(v)) := PRUNED
        f(parent(v)) := g(parent(v)) + q(parent(v))
end Prune
```

Figure 3: Pseudocode for the PRA* algorithm (Part 2).

independently developed by Manzini and Somalvico in [15] and [16].) In fact, for all the test problems we ran on PRA*, every processor of the CM-2 was continually active after the initial set-up phase.

Like RA*, PRA* chooses nodes for expansion on the basis of their $f$-values. But unlike RA*, PRA* expands many nodes simultaneously. In particular, each processor selects from its bucket an expandable node having the smallest $f$-value among the nodes in that bucket, and expands that node. Because each processor expands the node that is *locally* the best, a locally chosen goal node is not necessarily globally the best. Consequently, the first solution path found by PRA* may not be optimal. To handle this situation, PRA* maintains two global values: $P$ is the best complete solution path found so far, and $U$ is the cost of this path. Any leaf node $t$ for which $f(t) \geq U$ is pruned. Once a solution is found, the algorithm does not terminate until all nodes have been pruned. (Solutions are recorded as they are found, so that the algorithm is free to prune the solution paths as well.)

Retraction does not occur in RA* until all memory is exhausted. In PRA*, however, retraction occurs whenever any one processor's memory is exhausted. Unfortunately, because the search space is distributed across all processors, it is possible for a bucket to become filled with non-frontier nodes, and thus have no candidates for retraction. In particular, in the last few lines of the INSTALL procedure (see Figure 3,) if the bucket is full then the node to be installed, $t$, will instead be retracted, even if it has a better $f$-value than all the nodes in the bucket. In almost all cases, this will only cause a temporary delay: as the algorithm proceeds, a node will eventually be retracted from the bucket (because the f-value of the frontier will eventually become larger than that of some leaf node in the bucket), so that the next time $t$'s parent is expanded, $t$ will be installed. Nonetheless, in the current implementation of PRA*, a "full-bucket" condition theoretically can lead to deadlock state from which PRA* will not terminate–$t$ might be permanently locked out of the bucket. (RA*, on the other hand, is guaranteed to terminate if a solution exists.)

The situations that could give rise to the kind of lockout described above are rather bizarre. For example, lockout could arise if there is a subtree $T$ such that (1) all non-leaf nodes in $T$ are hashed to some set of buckets $\{i_1, \ldots, i_k\}$, (2) these nodes completely fill the buckets $\{i_1, \ldots, i_k\}$, (3) the leaf nodes of $T$ are all hashed to buckets other than $\{i_1, \ldots, i_k\}$, and (4) upon expansion of any leaf node $t$ of $T$, each child of $t$ hashes to one of the buckets $\{i_1, \ldots, i_k\}$. If the hash function is a good one, the probability of occurrences such as this are so low as to make it impossible in any practical sense. PRA* can be modified to check for such lockouts at the cost of extra memory or node expansions. However, given the implausibility of such situations occurring (particularly in the 15-puzzle) the current implementation doesn't make such checks.

# 3 Implementation Details

The implementation of PRA* on the SIMD CM-2 was relatively straightforward. However, several aspects of the algorithm could be implemented in multiple ways. While we generally did not try to optimize board expansion or other 15-puzzle specific code, we did attempt to make best utilization of the processors, and to minimize the time spent in interprocessor communication, bucket manipulation, and the pruning of non-leaf nodes.

### 3.1 Interprocessor Communication

Because PRA* distributes the nodes of the search space across all processors, the algorithm requires frequent interprocessor communication. Some of these operations result in a high degree of communication congestion at certain processors. This congestion can significantly degrade run-time performance. [5] The overhead of dealing with such communication congestion was significant in PRA*.

For example, in the first line of the *Install* procedure (Figure 3), every processor that has created a new node (and that is usually every processor) attempts to send that node, $t$, to its assigned processor, $H(t)$. Because these broadcasts are simultaneous, many processors will receive broadcasts from several others simultaneously. In the current implementation, however, only one of these "colliding" broadcast messages gets through. The current implementation uses a "transfer, acknowledge, retry" technique to deal with collisions. The receiving processor acknowledges receipt to the successful sender. Processors receiving no such acknowledgement rebroadcast their message until it gets through. Thus, it can take many actual broadcasts (the average was six to nine for the 15-puzzle) to effect one abstract broadcast.

To ameliorate the performance degradation resulting from the need for these multiple broadcasts we split the process into two phases. In the first phase, the broadcast messages consisted of only the address of the sending processors (16-bits). As described above, these messages were subject to collision. As these messages arrived they were enqueued. In the second phase, after all messages had been received, the processors iterated through their queues, retrieving from the original senders the full node data structures (160 bits) of the newly received children[6]. The retrieval process does not suffer message collisions.

The enqueing method requires more communication operations (because of the additional operations constituting the retrieval phase), but fewer large ones. This is important because the run-time of *send* operations on the CM-2 is proportional to the size of the message. In most circumstances the run-time saved by using the smaller (16-bit) *sends* outweighed the time required for the few additional larger (160-bit) ones. PRA* minimizes its losses due to message collision by minimizing the size of colliding messages.

### 3.2 Bucket Manipulation

Bucket nodes are sorted and indexed with two keys, $f$-value and board position, to facilitate the two most common reasons for accessing the buckets. Even with the double-indexing, though, operations involving the buckets are slow because the CM-2 does not provide efficient address indirection. There is no operation to, "move the data at the address stored at address $X$ to address $Y$".

Address indirection can be accomplished through use of the CM's *sideways array* data structures, but these are fairly cumbersome, and the operations using them are slow. Nonetheless, they are the best mechanism available, and PRA* uses sideways arrays to represent buckets, and several indexing schemes (requiring a significant amount of memory) to optimize sideways array operations. Even with optimization, these operations comprise

[5] We have discussed this elsewhere [6].

[6] The most recent release of *Lisp includes a *send-with-queue* operation, but we've not yet altered the implementation to use it.

a significant amount of the program's run-time. It can be argued that this is the price to pay for duplication detection. Nonetheless, if the CM-2 could be modified to provide quick address indirection (say, two to ten times slower than a standard *move* operation), we estimate that PRA* would run at least ten times faster.

### 3.3 Messages to Pruned Nodes

Another implementation complication arose from our treatment of pruned non-leaf nodes. Two situations can cause non-leaf nodes to be pruned: a cheaper path may be found to the non-leaf node, $t$, or a solution node with a better $f$-value may have been found, prompting PRA* to prune $t$. In either case, we decided *not* to prune existing descendants of $t$; to traverse $t$'s entire subtree, scattered across many processors, for the purpose of pruning the subtree, is too expensive an operation. Instead, these nodes remain in their respective buckets. These nodes are left to prune themselves upon detecting that their parent node has been pruned. Eventually, the "detached" descendants of $t$ may have reason to send a message to their no longer existing (pruned) parent (usually to indicate to their parent that they have been retracted or pruned.) For example, node $t'$ sends a message to its parent node, $t$. The receiving processor (which had formerly held $t$) detects that $t$ is not stored in its bucket, and returns a message to this effect. Only then, having detected that it is a component of a pruned subtree, does $t'$ prune itself.

## 4 Related Work

### 4.1 Serial Limited-Memory Heuristic Search Algorithms

As shown in [3], every admissible search algorithm must expand all *surely expandable nodes* before finding a solution. Since the number of such nodes often grows exponentially with some measure of the problem size, every admissible search algorithm must either have an exponential amount of memory in which to store these nodes, or else (if it operates in a limited amount of memory) it must delete nodes that have already been generated and possibly regenerate them later.

IDA* [9] was the first algorithm to solve 15-puzzles efficiently within limited memory. IDA* is a tree-search algorithm, and works with graph search spaces by unfolding them into search trees. IDA* is iterative, executing a depth-first search that is delimited by a *threshold* that is increased at each iteration. In each iteration, IDA* executes a depth-first search from the root node until a goal node is found or all nodes with $f$-values within the current threshold have been examined. If a goal node is not found, the threshold is incremented and the search restarted (a new iteration) from the root node. Because IDA* simulates the best-first search strategy of A* by using an underestimating threshold value, it it is guaranteed to find an optimal solution under admissible heuristics. The space complexity of IDA* is linear in the depth of the search tree. However, it is known that IDA* makes significantly more node expansions than A* in practice. Thus, for problems where node expansion time is significant, IDA* fares badly.

The algorithm MREC [24] is a variant of IDA*. MREC is a recursive marking algorithm that maintains an explicit graph during the search. When it runs out of memory, it uses

IDA* to increase the threshold value at its tip nodes and continue searching. Eventually MREC finds a solution path in the explicit graph or with the help of IDA*. Thus, MREC can use a larger initial memory than IDA*, but otherwise is not much different from IDA*.

MA* [2] is a significantly different algorithm for use with limited memory—it attempts to optimize the use of memory during search. During each expansion phase, MA* generates only one child of each node selected for expansion, only partially expanding that node. It maintains an explicit graph and runs like any standard marking algorithm for networks, such as MarkA [1]. As in MarkA, MA* performs bottom-up cost computations. When out of memory, it selectively prunes nodes and arcs from the explicit graph. These pruned nodes, and other partially expanded nodes can be reexpanded at a later time. It can be shown that through bottom-up computation and selective generations of children, MA* maintains the best-first node selection strategy of algorithm A*.

In a general sense, RA* and MA* are similar algorithms: both do a best-first search, deleting nodes from memory when necessary to save space. However, there are also a number of differences. MA*'s need to generate children selectively, and the bottom-up nature of its marking algorithm, make it difficult to envision a massively parallel form of MA* without a fully shared memory. In contrast, RA* is explicitly designed to use update algorithms that are efficiently parallelizable. These and other differences are described in more detail in Appendix B.

### 4.2 SIMD Heuristic Search Algorithms

While there has been great deal of research into serial heuristic search, and some into coarse-grained MIMD search[7] [7, 10, 23], research into SIMD search algorithms is just beginning. Other than PRA*, the only attempts we know of to develop SIMD search algorithms have been parallelizations of IDA* [20, 22, 11, 12].

The earliest attempt at a SIMD version IDA* was done by Powley and Korf [20]. They parallelized IDA* by allowing individual processors to execute separate depth-first searches to different thresholds (cost bounds). The first solution found is not guaranteed to be optimal, so search continues until a better solution is found or until a solution is proven to be optimal. This approach requires very little load-balancing, but the speedup result obtained by this algorithm was only 707 (4.9% of the 16K processors used).

Three more parallelizations of IDA* were developed later: two versions of the P-IDA* algorithm [22, 21], and the IDPS algorithm [11]. These more recent efforts have had more success. One of the versions of P-IDA* has achieved a 4600-fold speed-up on a 16K Connection Machine [22], and similar results have been obtained for IDPS [11].

P-IDA* works by dividing up the search space into subspaces, each with its own "start" node (which may be some distance away from the real start node). Initially, a single processor is assigned to each subspace, to do an IDA*-style search of that subspace. But after each threshold, more or fewer processors may be allocated to each subspace in order to achieve load balancing.

IDPS differs from P-IDA* in three essential ways. First, IDPS uses a different static

[7] Coarse-grained MIMD parallelization of IDA*, with each processor searching to a different threshold, has shown near-linear speed-ups, but only for small numbers of processors [23]. As the number of processors increases, speed-up declines.

load balancing strategy to initially distribute the subspaces across the processors. Second, IDPS uses a different dynamic load balancing strategy which results in very high efficiency. Third, IDPS offers two variations of node generation strategy, allowing the algorithm to optimize its performance for different domains.

### 4.3 Comparing PRA* with the Parallizations of IDA*

IDA* stores only the current node and the path from that node back to the start node $s$; and similarly, the parallel versions of IDA* store only the current nodes and the paths from those nodes back to $s$. This technique requires that IDA* and its parallelizations regenerate many nodes many times over. These regenerations are necessitated in two ways. First, whenever the threshold is increased, the search tree of the previous threshold cycle must be regenerated. Second, any node $t$ in the search space may be accessed along many different paths. For the 15-puzzle, for example, most board positions (the exceptions are those in which the "empty" tile is in a corner or on an edge) are accessible by a single tile move from any of four adjacent board positions. Because IDA* and its parallelizations maintain no history of which nodes have been previously examined, they must reexamine the entire subtree of descendants of $t$ each time $t$ itself is examined.

A* avoids this redundant work by storing all nodes as they are examined. This makes it possible for A* to detect when a node is later reached via a path using a different adjacent node. (We call this *duplicate detection.*) A* doesn't reexamine the duplicate's descendant sub-tree unless it has discovered a cheaper path to the duplicate. The central problem with A* is that the search space for most interesting domains is far too large to be stored in memory. PRA*, rather than trying to solve a search problem with a minimal amount of memory, as does IDA*, searches by *optimizing* its use of memory. It maximizes its use of available memory so as store as many of the previously examined nodes as possible, and so minimizes the need for redundant node examinations.

Unfortunately, in terms of run-time, the cost of duplicate detection in PRA* is quite high. It necessitates PRA*'s use of general interprocessor communication (among the most expensive operations on the CM-2) and the code to search each bucket for a duplicate whenever a new node is added. These two functions account for about 92% of PRA*'s run-time on 15-puzzle problems (though some interprocessor communication would be required for load-balancing in any case.)

PRA*'s load-balancing scheme is predicated on the need for duplicate detection. The hashing function, because it is known to every processor, guarantees that a duplicate, regardless of at which processor it was generated, will be sent to the processor holding the original. Using a hashing function for load balancing has disadvantages. First, there is no guarantee that nodes will be distributed evenly across the processors. The distribution is a product of the hashing function and the search space. Second, the topology of the hashing distribution has nothing to do with the topology of the search space itself. This necessitates general interprocessor communication–as opposed to faster "nearest neighbor" communication–for communication between processors representing nodes adjacent in the search space.

Our main criticism of IDA* derivatives like P-IDA* and IDPS is that though they enjoy a run-time speed-up with increasing numbers of processors, they do not benefit from the

larger memory associated with larger parallel machines, and thus they do not do any less total work. For example, Powley *et al* report [21] that P-IDA* required roughly the same number (slightly more, actually) of node expansions to solve a problem as serial IDA*, regardless of the number of processors. For some problems [8], IDA*'s number of node expansions (and thus its runtime) can be as high as $\Omega(2^{2n})$, where $n$ is the number of node expansions that would be done by A*. Thus, simply adding more processors to an IDA*-like algorithm may not be enough to make a problem tractable. In view of this criticism, it is of obvious interest to design a parallel algorithm that will become closer to A* as more processors and their corresponding memory are added. PRA*, which is more like A*, exhibits this behavior.

Powley *et al* [21] argue that PRA* will enjoy less and less benefit from duplicate checking as problem size increases. They claim that for large enough problems, the benefit to be derived from the finite number of nodes that PRA* can store will become inconsequential. This is far from obvious. Indeed, the opposite may be true.

It is true that as problem size increases, PRA* is able to store a relatively smaller proportion of the search space, and thus must fail to detect an increasing number of duplicates. On the other hand, PRA* will tend to store a large proportion of the higher (closer to the root) nodes in the search graph. The value of such duplicate detection for such nodes becomes increasingly important with larger problems because the benefits of avoiding duplicates are exponential in the distance of the duplicate from a goal. Alternatively, it may work out that the topology of the search space is such that the duplicates tend to be those nodes that have not yet been retracted. The interaction between PRA*'s retraction mechanism, and the topology of the search space would seem to be the main factor in the effectiveness of duplicate checking in PRA*. We don't believe that the simple proportion of search space size to memory size is an adequate metric of duplicate-checking effectiveness.

# 5 Experimental Results

We have used the 15-puzzle to test the performance of PRA*. The algorithm is invoked with a starting board position, and outputs the moves necessary to reach a given goal state.

The 15-puzzle is known to have a large search space. In [9] Korf presents the results of running IDA* for one hundred random start states. The number of nodes expanded by IDA* varies from several hundred thousand, to over six billion. The P-IDA* parallelization of IDA* [22] achieves similar results. (P-IDA* requires about four percent more node expansions than IDA*, on average.)

To test our algorithm, we have compared the performance of the IDPS algorithm (which is similar to P-IDA*) against that of PRA* using a 16K processor CM-2 with a Sun-4 front-end. In addition, we have run PRA* using different numbers of processors to examine parallel speed-ups. For the heuristic function $h$ we used the sum over all tiles $t$ of the Manhattan distance from $t$'s current to its desired location.

## 5.1 Comparison with Parallel IDA*

It was our hypothesis that the design of PRA* would enable it to expand significantly fewer nodes than IDPS when solving 15-puzzle problems. To test this hypothesis, we ran

Figure 4: Comparison of PRA* and IDPS sorted by IDA* node expansions

Figure 5: Comparison of PRA* and IDPS sorted by solution path length

PRA* and IDPS on 45 15-puzzle initial and target pairs for which known IDA* performance has been published [9]. We then compared the number of node expansions performed by IDPS with the number of node expansions (we use the term *expansions* to represent both expansions and reexpansions) computed by PRA*.

Figure 4 shows the results sorted by the number of node expansions for IDPS (circles represent the IDPS results, squares the PRA* results). As can be seen, PRA* made fewer node expansions than IDPS for all but the smallest problems. For the 15-puzzle, as in many search problems, solution length is a reasonable indicator of the difficulty in finding an optimal solution, although there is a large variance. Figure 5 shows the same data as the previous figure, sorted by the length of the optimal solution paths. Exponential curves have been fit to both sets of data. As can be seen, the curve fit to the PRA* data appears to indicate that PRA* performs exponentially better than IDPS ($e^{0.20}$ for IDPS vs. $e^{0.17}$ for PRA*). In the absence of any contrary evidence, it seems reasonable to anticipate that PRA* will continue to require exponentially fewer expansions than IDPS for larger problems.

### 5.2 Parallel Speed-up

For the CM-2 (and other SIMD systems) computing speed-up is difficult as the "single processor" case is often uncomputable, as is the case for PRA*. To run the algorithm on a CM-2 restricted to use only one processor would require a prohibitive amount of CPU-time. For some algorithms, speed-up can be calculated indirectly by first calculating the algorithm's efficiency (as Powley *et al* did for P-IDA* in [22].) Unfortunately, this technique is possible only when the efficiency of the algorithm can be estimated directly. This appears impossible for PRA* because the interaction between the retraction and hashing mechanisms makes it difficult to determine what proportion of PRA*'s "work" (retractions and expansions) would be germane to RA*'s.

To observe parallel speed-up, we calculated an *adjusted speed-up* which takes the 1000 processor computation as the base case, and computes run-time speed-ups with respect to this number. Thus, although exact comparisons to a serial case cannot be computed (as they could for coarse-grained MIMD algorithms) an approximation of a speed-up effect can be computed.

Figure 6 shows graphs of adjusted speed-up vs. 1000's of processors for four different fifteen puzzles of increasing difficulty (the number of expansions for the 16K processor case is displayed for each graph). As can be seen, relative speed-up appears to be linear with the

Figure 6: Speed-up comparisons (multiplier is 1000)

number of processors. For small problems, though, this linearity can plateau at a saturation point. Such problems do not present enough work to adequately utilize all the available processors. The addition of more processors doesn't hasten the solution of the problem. As the number of nodes to be searched increases, the number of processors at which this saturation occurs also increases. As shown in Figure 6, by 800K expansions we are already seeing the saturation disappearing for the 16K processor case. For the 1.6M expansion problem, the saturation effect is completely absent. This leads us to believe that larger puzzles, which can require tens of millions of insertions, will not exhibit this saturation effect even for significantly larger SIMD machines. Consequently, we believe PRA* will scale-up well to larger problems and machines.

# 6 Discussion

Heuristic search algorithms may be divided into two groups: graph-search algorithms (which do duplicate detection) such as A*, C [1], Graphsearch [18], MIMD Parallel A* [10] and PRA*, and tree-search algorithms (which do not do duplicate detection) such as IDA*, MREC, MA*, P-IDA*, IDPS and MIMD Parallel IDA* [23]. The choice of whether to use a parallel graph-search algorithm such as PRA* or a parallel tree-search algorithm such as IDPS or P-IDA* depends on the problem domain. There are two major features of the domain which will affect the space-time tradeoffs: node expansion time, and how heavily the domain is "latticed".

## 6.1 Node Expansion Time

For domains in which expansion time dominates run-time performance, such as some representations of the Traveling Salesman Problem, a graph-search algorithm such as PRA* is to be preferred over an IDA* derivative. PRA*'s strength is that it tries to minimize the total number of expansions needed to find a solution. This strength offers the biggest returns in such a domain.

In domains where node expansion is relatively simple, an IDA* derivative, like IDPS, may be best. Of course, the topology of the search space is an important factor, too. If the search graph is heavily cross-linked or latticed, the amount of redundant reexaminations of search subtrees in IDPS may outweigh any savings in avoiding duplicate checks.

## 6.2 Search Space Topology

For domains which are not heavily latticed, and in which expansion time is relatively inexpensive, a tree-search algorithm such as IDPS is probably the best choice. There, the run-time savings due to duplicate checking are not as significant as they might be for more complex domains. The 15-puzzle is such a domain, and thus we expected IDPS to run faster

than PRA* on this problem. Empirical studies confirmed this, with IDPS running about ten times as fast as PRA* for the problem suite featured in 5.[8]

In heavily "latticed" domains the sheer amount of redundancy avoided by duplicate-checking may outweigh the time spent discovering the redundancies. In such cases, the number of nodes expanded by algorithms such as IDA* and IDPS (which do not do duplicate-checking) can sometimes be as high as $\Omega(2^{2n})$, where $n$ is the number of nodes expanded by algorithms such as A* which do duplicate-checking [8]. PRA*, which does as much duplicate-checking as possible within its memory restrictions, will tend toward A*'s performance characteristics. Thus in such cases, PRA* may still outperform IDPS even if node expansion is cheap.

# 7 Conclusions

In this paper we have presented PRA*, an algorithm for heuristic search that takes advantage of the SIMD architecture of the CM-2. The algorithm has some of the advantages of a best-first search such as A*, while still functioning within a limited memory. To enable it to operate in limited memory, PRA* uses a process of retraction (and possible later reexpansion) of nodes with poor $f$-values.

We have shown empirically that PRA* expands significantly fewer nodes than parallel implementations of IDA*. In addition, PRA* is designed to maximize processor utilization on SIMD architectures, and empirical results show that it promises to be scalable to larger and more complex search problems.

# Acknowledgements

This work was supported in part by an NSF Presidential Young Investigator award for Dr. Nau with matching funds from Texas Instruments and General Motors Research Laboratories, NSF Equipment grant CDA-8811952 for Dr. Nau, NSF Grant NSFD CDR-88003012 to the University of Maryland Systems Research Center, NSF grant IRI-8907890 for Dr. Nau and Dr. Hendler, and ONR grant N00014-88-K-0560 for Dr. Hendler. Matthew Evett is supported in part by a DARPA/NASA assistanceship administered through the UM Institute of Advanced Computer Studies.

# References

[1] Bagchi, A. and Mahanti, A. Three approaches to heuristic search in networks *JACM* 32, (1985)

[2] Chakrabarti P., Ghose S., Acharya A. and de Sarkar S., Heuristic Search in Restricted Memory, *Artificial Intelligence* **41**, 1989, pp. 157–196.

[8] This comparison is somewhat unfair to PRA*, because our implementation of IDPS used a heavily optimized node expansion routine and our implementation of PRA* did not. Our IDPS node expansion routine ran several times as fast as our PRA* node expansion routine. If we had recoded PRA* to use the same node expansion routine we used in IDPS, its runtime would have been much closer to what we observed for IDPS.

[3] Dechter, R. and Pearl, J. The Optimality of A* *in* Kanal, L. and Kumar, V. (eds) *Search in AI*, North-Holland.

[4] Dechter R. and Pearl J., Generalized Best-First Search Strategies and the Optimality of A*, *JACM*, Vol. 32 (3), July 1985, pp. 505-536.

[5] Evett, M., Hendler, J., Mahanti, A. and Nau, D. "PRA*: A Memory-Limited Heuristic Search Procedure for the Connection Machine," *Proceedings of The Third IEEE Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, October 1990, pp. 145–149.

[6] Evett, M. *Massively Parallel Knowledge Representation.* Ph. D. Dissertation, Computer Science Department, University of Maryland, December 1993.

[7] Ferguson, C. and Korf, R. "Distributed Tree Search and its Application to Alpha-Beta Pruning," *Proceedings of The Seventh AAAI National Conference on Artificial Intelligence*, Saint Paul, Minnesota, August 1988, pp. 128–132.

[8] Ghosh, S. Doctoral Thesis (in preparation), University of Maryland.

[9] Korf, R. Depth First Iterative Deepening *AI Journal,* 27 (1985)

[10] Kumar, V., Ramesh, K. and Rao, V.N. "Parallel Best-First Search of State-Space Graphs: A Summary of Results," *Proceedings of the Seventh AAAI National Conference on Artificial Intelligence*, Saint Paul, Minnesota, August 1988.

[11] Mahanti, A. and Daniels, C.J. "SIMD Parallel Heuristic Search," Technical Report CS-TR-2633, Computer Science Department, University of Maryland (March, 1991).

[12] Mahanti, A. and Daniels, C.J. "A SIMD Approach to Parallel Heuristic Search," Workshop on Parallel Computing of Discrete Optimization Problems, Univ. Minnesota, May 22-24, 1991.

[13] Mahanti, A. and Ray, K. "Network Search Algorithms with Modifiable Heuristics," *in* Kanal, L. and Kumar, V. (eds) *Search in AI*, North-Holland, 1988.

[14] A. Mahanti, D. S. Nau, S. Ghosh, A. K. Pal, and L. N. Kanal. Performance of limited-memory heuristic search algorithms for networks: A theoretical and empirical analysis. 1991. Submitted for journal publication.

[15] Manzini, G. and Somalvico, M. Probablisitic Performance Analysis of Heuristic Search Using Parallel Hash Tables, extended abstract, *Proceedings of the International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, FL, January, 1990.

[16] Manzini, G. and Somalvico, M. Probablisitic Performance Analysis of Heuristic Search Using Parallel Hash Tables, submitted to *Annals of Mathematics and Artificial Intelligence.*

[17] Nau, D., Kumar, V., and Kanal, L. General Branch and Bound and its relation to A* and AO*, *AI Journal*, 31 (1987).

[18] Nilsson, N. *Principles of Artificial Intelligence,* Tioga, CA (1980).

[19] Pearl, J. *Heuristics*, Addison-Wesley, MA (1984).

[20] Powley, C. and Korf, R. SIMD and MIMD parallel search, *Working notes of the 1989 AAAI Spring Symposium on Planning and Search*, Stanford, CA (1989)

[21] Powley, C., Korf, R. and Ferguson, C. IDA* on the Connection Machine. International Workshop on Parallel Processing for Artificial Intelligence (PPAI-91), Sydney, Australia, August 24-25, 1991.

[22] Powley, C., Korf, R. and Ferguson, C. IDA* on the Connection Machine. Workshop on Parallel Computing of Discrete Optimization Problems, Univ. Minnesota, May 22-24, 1991.

[23] Rao, V.N., Kumar, V., and Ramesh, K. A Parallel Implementation of Iterative-Deepening-A*, *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87).*

[24] Sen, A. and Bagchi, A. Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory *Proceedings IJCAI-89* (1989).

# A Correctness of RA*

Below is an outline of a proof that RA* is guaranteed to terminate with an optimal solution, provided that the following conditions are satisfied:

1. The heuristic function $h$ is admissible, i.e., $h(n) \leq h^*(n)$ for every node $n \in G$.
2. $G$ contains at least one solution path.
3. There is a number $\delta > 0$ such that the cost of each arc is no less than $\delta$.
4. $M \geq bL$, where $b$ is the maximum node branching-factor and $L$ is the maximum number of nodes on any path that can be generated by A*.[9]

The first three conditions above are identical to the conditions required for A* to terminate with an optimal solution. The fourth condition is necessary in order to guarantee that RA* has enough memory to run to completion, as discussed in Section 4.1.

Let $P$ be any solution path in the state space $G$. We define

$$\text{pathmax}(P) = \max\{f(n) | n \text{ is a node of } P\},$$

and

$$Q = \min_{P \in G} \text{pathmax}(P).$$

A solution path $P'$ is called a $Q$-min path if $\text{pathmax}(P') = Q$ and $\text{cost}(P') = \min\{cost(P'') | \text{pathmax}(P'') = Q\}$. As an immediate consequence of this definition, $G$ is guaranteed to have at least one $Q$-min path $P_0$. Note that if $h$ is an admissible heuristic, then $Q = h^*(s)$.

Consider the point where RA* selects a node $u$ for expansion, and let $P_0'$ be the portion of $P_0$ that RA* has generated so far. From the definition of $P_0$, it follows that the tip node $p$ of $P_0'$ has $f(p) \leq Q$. Therefore, since RA* always selects the node $u$ for which $f(u)$ is smallest, it follows that $f(u) \leq Q$.[10] This guarantees that if RA* terminates, it returns an optimal solution.

To show that RA* terminates, we note that when a node $n$ is selected for expansion, a depth-first search below that node continues (by the "most recent node" selection strategy) until all generated descendants of $n$ have $f$-values greater than some tip node of $T$ not below $n$. If any descendant of $n$ is ever retracted, its $q$-value is propagated upward (i.e., to ancestors), and this value is reflected in the $\bar{h}$-values that are propagated downward when nodes are regenerated. This prevents the algorithm from oscillating (i.e., generating and retracting the same node again and again).

[9] Of course, neither $b$ nor $L$ actually needs to be known beforehand. As long as $M$ happens to be less than $bL$, RA* will work, regardless of whether or not $b$ and $L$ happen to be known to the user. Note also that most other search algorithms have analogous requirements. For example, any implementable search algorithm can be assumed to have a finite, bounded branchout $b$, for otherwise we could not store the rules for node expansion [18] [19, p. 33].

[10] For more details about this kind of argument, the reader is referred to [13].

## B Comparison of RA* with MA*

In a general sense, RA* and MA* are similar algorithms: both do a best-first search, deleting nodes from memory when necessary to save space. However, the details of the algorithms differ significantly. Here are some of the primary differences:

1. RA* *fully* expands nodes, installing all children of $n$ in the explicit graph. MA*, on the other hand, installs only one child during each expansion; the order in which MA* generates and installs the children of a node is determined by the relative heuristic values of the children. This difference has two main consequences, as outlined below.

   Because of the different expansion techniques, MA* can get by with a smaller amount of memory than RA*: the minimum amount of memory required for MA* is $L$, and is $bL$ for RA*, where $b$ is the maximum node branching-factor and $L$ is the maximum number of nodes on any path that can be generated by A*.[11] However, MA*'s smaller memory requirement comes at the expense of additional runtime overhead for MA*: it must come back and generate $n$'s other children later in additional expansion steps. With larger amounts of memory, RA* will run more and more like A*, behaving exactly like A* if there is enough memory that no retractions are needed. In contrast, because MA* generates only one of a set of sibling nodes at a time, it will need to reselect and reexpand a node again and again to get all the children of that node—and this will occur even if MA* has sufficient memory to hold the entire state space.

2. RA* computes $f$-values according to the formula $f(n) = g(n) + h(n)$, just as in algorithms such as A* and IDA*. In contrast, MA* computes $f(n)$ by looking ahead at $n$'s children and calculating their $h$ values.

   MA*'s lookahead step was intended to reduce the number of node generations done by MA*, but it incurs a high overhead, and has not been shown in practice to actually reduce the number of node generations.[12] Furthermore, MA*'s lookahead step sometimes causes it to calculate incorrect $f$-values for goal nodes—and as a result, MA* will sometimes return a suboptimal solution path and/or return a solution cost value greater than the actual cost of the solution path that it returns [14].

3. Since both MA* and RA* need to delete nodes from memory in order to save space, their correct performance depends on updating the heuristic values of the ancestor nodes based on the heuristic values of the deleted nodes. However, MA* and RA* use different updating techniques. MA* maintains an IDA*-style threshold value, and whenever it generates a path $P$ whose cost exceeds this threshold, it does an AO*-style update of the $h$ values of every node of $P$. In contrast, RA* updates heuristic values only when some node $n$ is actually retracted—and rather than updating the heuristic values of all nodes in the path to $n$, it updates only the value of $n$'s parent.

[11] Note that $b$ and $L$ are not parameters of RA* and thus do not need to be fixed in advance. Rather, if the amount of memory given to RA* happens to be at least $bL$, then RA* will perform correctly, as proved in Appendix A.

[12] For example, in experimental tests on a set of 15-puzzle problems, MA*(0) did slightly more node generations than IDA* (the claim by Chakrabarty *et al.* [2] that MA* only does 57% as many node generations as IDA* is incorrect). For further details, see [14].

MA*'s updating step involves a number of computations along the path that are not easily parallelizable. RA*, on the other hand, is explicitly designed to use update algorithms which are efficiently parallelizable. In addition, RA*'s approach eliminates unnecessary overhead: RA* only computes those values that are actually needed—and rather then computing repeatedly whenever some threshold is exceeded, it only computes them once, at the time that they are needed.

4. It is well known that when a search algorithm is run on a graph (rather than a tree) with a non-monotone heuristic function, it can sometimes find lesser-cost paths to nodes that it has already expanded. In such cases, RA* will move those nodes from its CLOSED list back to the OPEN list, to expand them again. In contrast, MA* does a top-down propagation of $h$-values which the authors claim makes $h$ "effectively monotone," so that there is no need to bring a node from CLOSED list to its OPEN list. However, there are several problems with the technique used in MA*. First, it involves a number of computations along the path that are not easily parallelizable. Second, studies show that it leads to incorrect results, and cannot be made to work correctly without incorporating some major modifications so that it brings nodes from CLOSED to OPEN as done by RA* [14].