PRA*: A memory-limited heuristic search procedure for the Connection Machine

Matthew Evett James Hendler Ambujashka Mahanti
Dana Nau
Institute of Advanced Computer Studies
University of Maryland

Abstract

In this paper we describe a variant of A* search designed to run on the massively parallel, SIMD Connection Machine. 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 if more promising paths fail. Our algorithm, called PRA* (for Parallel Retraction A*), is designed to take maximum advantage of the SIMD design of the Connection Machine. In addition, the algorithm is guaranteed to return an optimal path when an admissable heuristic is used. Results comparing PRA* to Korf's IDA* [3] for the fifteen-puzzle show significantly less node expansions for PRA*.

1 Introduction

A major problem in producing a SIMD parallel version of a heuristic search program is that in most applications of search, the storage requirements of most of the well known heuristic search algorithms grow exponentially with some measure of the size of the problem. Thus, it becomes infeasible to solve many interesting and useful problems even using the large memories available across the many processors on a massively parallel machine. In the algorithm A* [5], for example, which forms the basis for many recent search algorithms, and several of its variants [1, 4, 5, 6], the entire explicitly generated search space needs to be stored prior to termination. To circumvent the large storage requirement of A*-like algorithms, attempts have been made to design algorithms which can run with limited available memory and yet which run satisfactorily producing optimal or near optimal solutions.

The algorithm IDA* [3] is the first known algorithm which solved the fifteen puzzle problem efficiently with limited memory. IDA* converts a search graph to a search tree. In every iteration, IDA* starts the search from the root node with an iteratively increasing threshold, making a depth first search until a goal node is found or the current threshold is exceeded. Since IDA* simulates the best first search strategy of A* by using an underestimating threshold value, it guarantees finding an optimal solution under admissible heuristics. The space complexity of IDA* is linear in the depth of the search tree. However, it is known in practice that IDA* takes more execution time than A* for many applications, in particular where node expansions take significant time. This is because IDA* will often expand more nodes than A*.

The algorithm MREC [7] is a variant of IDA*. MREC is a recursive marking algorithm, which 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*.

The algorithm MA* [2] is another algorithm which runs with limited memory. During node expansions, MA* generates immediate successors selectively one at a time. It also 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 it runs out of memory, it prunes some nodes and arcs from the explicit graph. It allows reexpansion of partially expanded nodes. It can be seen that through bottom-up computation and selective successor generations it maintains the best first node selection strategy of algorithm A*. However, because it generates a single node at a time, MA* reselects and reexpands a node again and again to get all immediate successors of a node—and this can occur even when there is enough available memory.

Each of the above approaches achieves the goal of limiting memory—but in general each of these approaches can expand the same nodes many times. How to design a limited-memory search algorithm which produces an optimal solution (with an admissable heuristic) without expanding the same nodes too many times is a significant problem.

In this paper we present an algorithm, called RA* and its parallel version, PRA*. RA* can be viewed as similar to algorithm A* except that it does pruning (retraction) of nodes because of memory limitation. Thus RA* retains the best-first strategy of A*, but runs with limited memory like IDA*. Unlike IDA*, in each iteration RA* does not expand nodes starting from the root node all over again. Instead, it maintains a partial explicit graph of potential (generated)

nodes. When RA* runs out of memory, it prunes some unpromising nodes of the explicit graph and backs up the best f-value of the pruned children at their immediate parents. Thus an already expanded node may be fully or partially retracted. In subsequent iterations, a partially expanded node becomes eligible for reexpansion, and other children of that node may be regenerated. It may be observed that RA* uses the f value or back up value at each node to forward the search path. Unlike MA* it uses a simple data structure at each node, and it does not do the search *ab initio* at each iteration. Thus, this algorithm makes effective usage of the available memory. For simplicity, it is assumed that the size of the available memory is constant. However, without loss of any generality our algorithms can be easily modified to run with variable size memory.

As well as running with a fixed size memory, the RA* algorithm has been designed to run efficiently on a SIMD parallel processor, in particular the massively Connection Machine (CM). The CM contains up to 64,000 simple processors which can run in parallel. Each of these processors contains a separate memory. In the parallel implementation (called PRA* for Parallel RA*) one node to be expanded is passed to each processor. These nodes are expanded in parallel, and the nodes which are generated are sent to appropriate processors (based on a hash function, see section 4) for storage. This allows duplicates to be removed (which cannot occur in IDA*) and for certain types of pruning to be used. After each broadcast, those processors which have full memories have nodes retracted (in parallel) to free up space for the next phase.

In Section 2 of this paper we present the RA* algorithm, and in Section 3 we derive some bounds on the number of times it expands each node. Section 4 describes PRA*, the SIMD parallel version of RA*, and Section 5 contains concluding remarks.

2 RA* algorithm

Figure 1 describes a procedure called RA* (Retracting A*). In a subsequent section we will define PRA* (Parallel Retracting A*) as a parallel SIMD implementation of RA*.

G is the search graph, U is an upper bound used for pruning, and K is an arbitrary constant. t, u, and v are nodes, and the cost of an arc from t to u is $c(t,u)$. $g(t)$ is the cost of the best path yet found from the root node, r to t, and $p(t)$ is the immediate parent of t along this path. As with the A* algorithm, h is the heuristic function and $f(t) = g(t) + h(t)$. We assume that h satisfies the monotone restriction used in A*.

Retracting a node consists of removing a generated node from G to save space, with the possibility of generating it again later. 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 have been retracted. In the latter case, expanding t consists of re-generating the retracted children.

For each generated node t we maintain a "backed-up" value $e(t)$ as follows. When t is generated, $e(t)$ is set to $\max(f(t), e(p(t)))$. Upon retraction of any of t's children, $e(t)$ is set to $\min\{e(u)|u$ is a retracted child of $t\}$ (it can be proved by induction that this value is always greater than or equal to the initial value of $e(t)$).

In general, the goal of a node selection strategy is to explore first those nodes that satisfy some criterion (e.g., deepest nodes or cheapest nodes). In RA*, this involves choosing not only which nodes to expand, but which nodes to retract. Usually, the nodes most desirable for expansion are the least desirable for retraction, and vice versa. As examples, below we describe two node expansion/retraction strategies for RA*: depth-first and best-first.

The depth-first expansion rule for RA* selects for expansion the most recently generated node, breaking ties (i.e., sibling nodes that were generated at the same time) in favor of the node v that minimizes $e(v)$. The depth-first retraction rule for RA* selects for retraction the least recently generated node, breaking ties in favor of the node v that maximizes $e(v)$.

The best-first expansion rule for RA* selects for expansion the node v that minimizes $e(v)$, breaking ties in favor of the most recently generated node. The best-first retraction rule selects for retraction the node v that maximizes $e(v)$, breaking ties in favor of the least recently generated node.

3 Analysis

Suppose that in addition to the usual admissibility criteria for the A* algorithm, we also have $K \geq d + b$, where d is the length of the optimum solution path, and b is the number of children of the second-to-last node on this path. Then it is straightforward to prove that both the depth-first and best-first implementations of RA* are admissible; i.e., they are guaranteed eventually to return the optimum solution if a solution exists.

Below we analyze the performance of the best-first implementation of RA* under the above admissibility conditions. To simplify the analysis, we assume that $f(u)$ is always an integer.

For each integer $i > 0$, let

$$G_i = \{u | f(p(u)) \leq i\}.$$

For each i, both A* and the best-first version of RA* will generate every node in G_i before generating any node in $G_{i+1} - G_i$; the difference is that RA* may retract some of the nodes in G_i in order to generate others. We define the i'th phase of the algorithm to be the steps executed between the time all nodes in G_{i-1} have been generated and the time all nodes in G_i have been generated.

The number of nodes generated by A* during its i'th phase is $|G_i - G_{i-1}|$; and at the end of the i'th phase, every expandable node u will have $f(u) > i$. As long as $|G_i| \leq K$, the same is true for RA*. Once i becomes large enough that $|G_i| > K$, RA* will start retracting nodes, to maintain exactly K nodes in G. In this case, at the end of the i'th phase, one or more expandable nodes u may have $f(u) \leq i$,

```
procedure RA*(r)
    install the root node in G
    U := ∞
    loop
        select an expandable node in G
        for every selected node t do
            if t is not a goal node then expand(t, U)
            else if g(t) < U then begin
                U := g(t)
                if f(v) > U for every expandable node v ∈ G then
                    return the path from r to t
            end
    repeat
end

procedure expand(t, U)
    for each child u of t not in G do
        if u has already been installed as a child of some other node then
            if making u a child of t would decrease f(u) then
                remove u and its descendants, and install u as a child of t
        else if f(u) < U then begin
            install u in G
            retract some nodes if necessary, so that G contains no more than K nodes
        end
end
```

Figure 1: RA* Procedure.

but the way $e(u)$ is computed guarantees that all will have $e(u) > i$.

At the beginning of the i'th phase, every expandable node t has $e(t) \geq i$. During the i'th phase, a node t becomes expandable either by being generated, in which case the monotonicity of h guarantees that $e(t) \geq i$; or by having one or more children retracted, in which case $e(t) = \min\{e(u) | u \text{ is a retracted child of } t\} \geq i$. Thus throughout the i'th phase, every expandable node t has $e(t) \geq i$. The i'th phase is finished when every expandable node t has $e(t) > i$.

Let u be any node that is expandable during the i'th interation. If u is chosen for expansion, its descendants will then be the preferred candidates for expansion unless they have e-values greater than i. Thus, after u has been expanded, all nodes chosen for expansion will be descendants of u, until such time as every expandable descendant v of u has $e(v) > i$. Any time a descendant v of u is retracted, the retraction will set $e(p(v)) > i$, so that $p(v)$ will not again be chosen for expansion during the i'th phase. In particular, u will not again be chosen for expansion during the i'th phase.

The above means that each node u is expanded at most once during the i'th phase. Thus, the total number of nodes generated during the i'th phase is no greater than $|G_i - K|$. In many search problems, $|G_i|$ grows exponentially with i; and in such cases,

$$|G_i - K| = O(|G_i - G_{i-1}|),$$

so that the asymptotic complexity of RA* is the same as that of A*.

As an example, consider the 15-puzzle with $h \equiv 0$, so that for every node u $f(u) = g(u) =$ the depth of u. Every node has at most four children, of which one is its parent, so $|G_i| \leq= O(3^i)$. Thus, for both RA* and A*, the number of nodes generated in the i'th phase is $O(3^i)$.

4 SIMD Implementation

As well as running with a fixed memory size, the RA* algorithm has been implemented to run efficiently on a SIMD parallel processor, in particular the massively Connection Machine (CM). The CM contains up to 64,000 simple processors which can run in parallel. Each of these processors contains a separate memory. In the parallel implementation, the memory of each processor in the CM is divided into two parts: One part, the *active node* which is used to represent one node to be expanded, and another, much larger, part which is used to contain a *bucket* in a large hash table. For each node t in the explicit search graph, the bucket (processor) that will contain t is determined using a hash function.

At each phase of the outer loop in the RA* algorithm, instead of expanding a single node, each processor chooses its best candidate, and these n nodes are chosen for expansion. Each node to be expanded is placed in the active node portion of its processor. These nodes are then expanded, and the nodes which are generated are sent to buckets in the appropriate processors based on the hash function (this allows a local computation at each node to be performed so as to eliminate duplicates). At each phase, all buckets which are filled have nodes retracted to free up space for the next phase. As each bucket chooses its own best node, and as any globally best node (i.e. lowest f-value over all currently open nodes) must be in some bucket, the globally optimal node will be chosen for expansion. (Where the same bucket contains several nodes with the same value as the globally best node, these nodes may be chosen for expansion first, however, as the heuristic is monotone, these nodes will eventually increasein f-value, and the global node will be chosen later for expansion.) Thus, in the presence of a monotonic heuristic PRA* will find the optimal path. As every processor in the machine is used both for a node expansion at each phase, and to represent a bucket in the hash table, maximum processor utilization results. Once the search space is large enough to start filling buckets, CM utilization remains close to 100 percent for expansion, broadasting, and duplicate removal. Retraction is also performed in all full buckets simultaneously, so in the presence of a good hashing function, once the search space is large, a large proportion of the processors all retract simultaneously.

4.1 Empirical Results

The fifteen puzzle has been used 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. This puzzle is known to have a large search space; In [3] a table is presented which shows the results of running IDA* for one hundred randomly starting states. The number of nodes expanded by IDA* varies from several hundred thousand, to over six billion. The number of nodes generated by A* itself for these puzzles is currently unknown, as keeping such a large open or closed list is prohibitively expensive on serial machines.

To test this algorithm, we have compared the performance of IDA* (as reported in [3] and PRA* using 8000 processors on a CM-2 (see Table 1). To show that PRA* can function under different memory limitations it is run for each problem using first limiting each processor to contain only 100 elements in its "bucket" (thus forcing retraction on smaller problems) – column 2, and then using representing 400 boards per processor (the maximum available in the current implementation) – column 3[1]. As can be seen, the number of nodes expanded by PRA* is significantly less than that used by IDA* even where a significant amount of retraction was required.

For the first three puzzles run, when these results were obtained by using the full memory of the 8,000 processors (400 nodes per bucket) the results are quite similar to expected A* values given the nature of the 15-puzzle and the slow growth of the Manhattan Distance heuristic used (see [3] for more details). Thus, the empirical results show that PRA* can be expected to compare quite favorably with A* for these puzzles[2].

The run-time for these puzzles ranged from approximately 20 minutes for the smallest to about four hours for the largest. While these numbers obviously compare quite favorably with typical serial LISP implementations of A*-like algorithms, they are unacceptably slow for the Connection Machine. An analysis of our performance has shown that the implementation of PRA* used currently, suffers from a number of design flaws which greatly slow its performance. The most important of these flaws is the manner in which the nodes are stored in the buckets. To select a node from a bucket, or to determine if any node already existed in a bucket it is necessary to scan the entire bucket. As this paper is being written, we are currently completing the implementation of a new version of PRA* which structures the bucket contents in such a way that only a very small portion (not more than 6 or 7 nodes, and usually only 1 or 2) needs to be scanned to retrieve an element. Current experimentation leads us to predict a performance speed-up of approximately 100x.

[1] The 4th board shown was not run in the 100 processor version due to execution time limitations on the available Connection Machine.

[2] The third puzzle shown is the largest of the puzzles presented in [3] which the 8000 processor CM-2 could handle with no retraction.

IDA*	PRA* 100/proc.	PRA* 400/proc
expansions	expansions (retractions)	expansions(retractions)
546,344	293,891 (264)	293,888
3,222,276	351,403 (6459)	351,414
17,984,051	1,447,434 (1,795,563)	1,077,685
183,526,883	—	11,857,876 (14,646,556)

Table 1: Results for PRA* vs. IDA*

5 Conclusions

We have shown in this paper that an algorithm for heuristic search can be designed which takes advantage of SIMD architectures on the CM. The algorithm has the advantages of a breadth-first search, such as A*, while still functioning with a limited memory size (as in IDA*), by using a process of retraction (and possible later reexpansion) of nodes with poor f-values. The algorithm is proven to be comparable in complexity with that of A* and has been shown empirically to expand significantly less nodes than IDA*, in fact, comparing favorably with A*. In addition, the algorithm is designed to make maximum processor utilization in SIMD, and thus 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. Dr. Nau and Dr. Hendler are also associated with the UM Systems Research Center. Dr. Mahanti is currently on leave from the Indian Institute of Management, Calcutta, India.

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 Sreach in restricted memory *AI Journal*, 41 (1989)

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

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

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

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

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