Distributed Multi-Heuristic A* (HAMSTAR)

Noam Brown, Aram Ebtekar, Yuzuko Nakamura

15-712 Fall 2014 Final Project Report

1 Abstract

A* is a popular graph search algorithm used in many artificial intelligence applications. The original A* algorithm makes use of a single heuristic to prioritize exploration through a graph, but a variation on the algorithm known as multi-heuristic A* (MHA*) makes use of multiple heuristics to more quickly arrive at approximate solutions. For this project we implemented a distributed version of MHA*, with one process per heuristic. We show... [performance]

2 Introduction

Weighted A* is a simple, heuristic-based search algorithm used in artificial intelligence applications such as robotics and games. A heuristic function estimates the remaining distance from any state to the goal. Whereas a breadth-first method such as Dijkstra's algorithm will expand exponentially many states relative to the goal distance, wA* will focus toward the goal when guided by a good heuristic. If the heuristic is **admissible**, meaning it never overestimates the true distance, then the solution is guaranteed to be optimal to within the weight factor. If it meets a stronger condition called **monotonicity**, then no state needs to be expanded more than once.

Finding good efficiently computable heuristic functions is difficult and, moreover, finding an admissible heuristic function that is reasonably accurate over the entirety of the search space is often not practical. Multi-heuristic A* (MHA*) [1] is an alteration of the A* algorithm that allows combines multiple arbitrary heuristics to guide the search; to retain the optimality bound, it is only necessary to use one monotone heuristic, called an "anchor". Where the shortcomings of each individual heuristic may result in local optima which trap the search, the ability to change heuristics can provide an escape.

In their work, Aine et al. propose MHA* in two flavors: Independent Multi-Heuristic A* (IMHA*), where path cost and shortest path information are tracked separately for each heuristic, and Shared Multi-Heuristic A* (SMHA*), where shared knowledge of path cost and shortest path information are shared and updated by all heuristics. IMHA* would be almost trivial to parallelize, as it requires minimal communication between the searches run by different heuristics. SMHA* uses a very large amount of frequently updated shared memory, presenting major challenges for parallel settings, let alone distributed systems settings.

Nonetheless, as SMHA* is the more effective of the two (enabling more co-operation and less redundancy between heuristics), we decided for our project to implement a distributed version of SMHA*.

3 Relevant Work

Our work is directly based on Aine et al.'s original multi-heuristic A* search [1] mentioned above. There exists past work on the paralellization of A*; in particular, Phillips et al. [2] were able to achieve parallelization while retaining wA*'s guarantee of suboptimality without reexpansions. This method of parallelization makes inferences on when it's safe to expand multiple nodes in parallel, by resolving dependencies between potential optimal paths. Their speed gains are dependent on the expansion operation being expensive enough to dominate the cost of computing these dependency checks, which may not be the case for all problems. By contrast, our method parallelizes the search process by simultaneously expanding nodes chosen by different

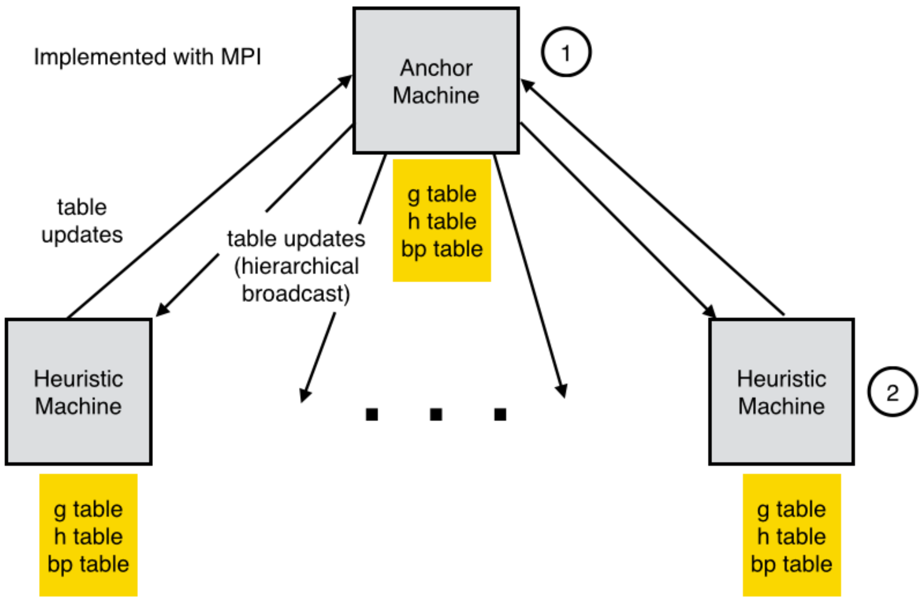


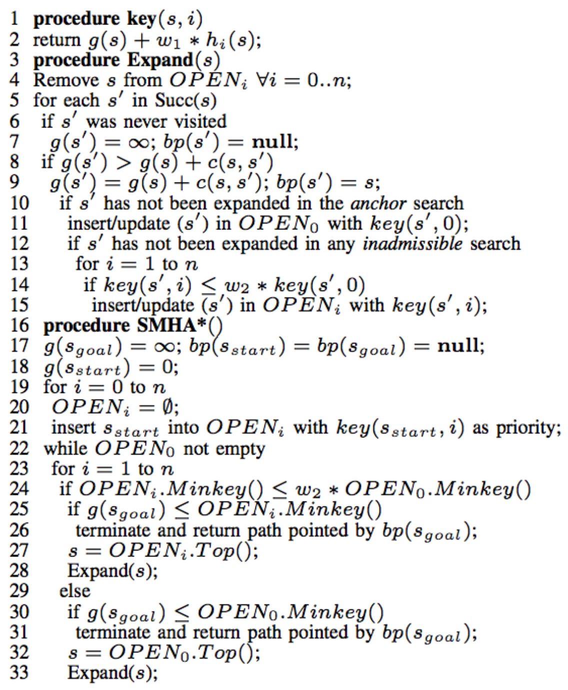
Figure 1: A diagram of our system. One machine starts up the others and runs the admissible anchor search. The other machines run their individual heuristics. Communication is done using MPI. ①**Anchor:** Starts heuristic machines; performs state expansions using anchor heuristic; receives, resolves, and broadcasts table updates; checks for termination conditions. ②**Heuristic:** Expands a likely state (chosen using heuristic); updates path cost to its successors; communicates updates to anchor; checks for termination conditions

heuristics, instead of multiple nodes chosen by a single heuristic. We allow each state to be expanded by each heuristic at most once and, furthermore, no state will be expanded by a non-anchor heuristic which has received a message from another heuristic after the latter has expanded the same state. In other words, aside from communication latency, no state will be expanded more than twice.

4 System Description

An overview of our system is in Figure 1. The main (anchor) machine starts up other heuristic machines. Each machine runs its own heuristic, with the anchor machine's being the admissible heuristic. Each of the non-anchor heuristic machines periodically sends updates of the data structures (tables containing shortest path information found so far) to the anchor machine, which compares the data it receives and sends out table updates of the best paths known so far to all other nodes.

Our algorithm is based off of the single-threaded, round-robin SMHA* code presented in [1] (Figure 2). Our parallelized code was written in C++ with inter-process communication accomplished using MPI. The work was distributed over one or more of the 16-core machines in the Apt cluster.



```
procedure key($s, i$)
return $g(s) + w_1 * h_i(s)$;
procedure Expand($s$)
Remove $s$ from $OPEN_i\ \forall i = 0..n$;
for each $s'$ in Succ($s$)
  if $s'$ was never visited
    $g(s') = \infty$; $bp(s') =$ null;
  if $g(s') > g(s) + c(s, s')$
    $g(s') = g(s) + c(s, s')$; $bp(s') = s$;
      if $s'$ has not been expanded in the anchor search
        insert/update ($s'$) in $OPEN_0$ with $key(s', 0)$;
        if $s'$ has not been expanded in any inadmissible search
          for $i = 1$ to $n$
            if $key(s', i) \leq w_2 * key(s', 0)$
              insert/update ($s'$) in $OPEN_i$ with $key(s', i)$;
procedure SMHA*()
$g(s_{goal}) = \infty$; $bp(s_{start}) = bp(s_{goal}) =$ null;
$g(s_{start}) = 0$;
for $i = 0$ to $n$
  $OPEN_i = \emptyset$;
  insert $s_{start}$ into $OPEN_i$ with $key(s_{start}, i)$ as priority;
while $OPEN_0$ not empty
  for $i = 1$ to $n$
    if $OPEN_i.Minkey() \leq w_2 * OPEN_0.Minkey()$
      if $g(s_{goal}) \leq OPEN_i.Minkey()$
        terminate and return path pointed by $bp(s_{goal})$;
      $s = OPEN_i.Top()$;
      Expand($s$);
    else
      if $g(s_{goal}) \leq OPEN_0.Minkey()$
        terminate and return path pointed by $bp(s_{goal})$;
      $s = OPEN_0.Top()$;
      Expand($s$);
```

Figure 2: Pseudocode from Aine et al., detailing the original single-threaded version of the algorithm we parallelized.

4.1 Design Decisions

Interprocess communication with MPI

We investigated other frameworks designed for iterative algorithms on a stable dataset such as Spark and GraphLab. However, most of these frameworks were designed for chunking up a large dataset upon which a single operation is performed. Our application's needs differed from this model in that the dataset being operated on (the graph being explored) was shared by all workers, and the workers themselves were performing differing operations on subsets of the same data.

In the case where the working set of data is small enough to be contained within each process, the system is much less complex, and the services provided by Spark and GraphLab are not so crucial. For these reasons we instead decided to distribute our code using MPI.

Synchronous broadcast vs. asynchronous sync-ups

Update collation performed at anchor node

SMHA* needs to share the values of g (distance estimates) and bp (back-pointers) for every state. As sharing memory is infeasible in our distributed implementation, processes periodically broadcast their g and bp values to update one another. This can result in conflicting data for a particular state, which is always resolved by taking the data which gives the lowest g value for that state as it contains the best estimate found so far. During conflict resolution, we also take the inclusive OR of the flags specifying whether the state has ever been expanded by the anchor, or by any machine at all.

Each broadcast by a heuristic machine only needs to contain the portion of the graph which has been updated by the machine since its most recent network communication. Nonetheless, if every machine broadcasts its updates directly to all other machines, the network would flood with messages and much conflict-resolution work would be duplicated. Therefore, we have every machine send its table updates exclusively to the anchor machine. The anchor alone takes the burden of resolving all conflicts between duplicate states, and then periodically broadcasts the merged collection of updates to all machines.

The disadvantage is that we rely heavily on the anchor machine functioning reliably. This is a small price to pay, given that the anchor is already necessary for updating the optimality bound. To avoid unduly burdening the anchor, every machine is able to compute the anchor heuristic and includes its value in the state updates it sends to the anchor machine.

Back-pointer optimization

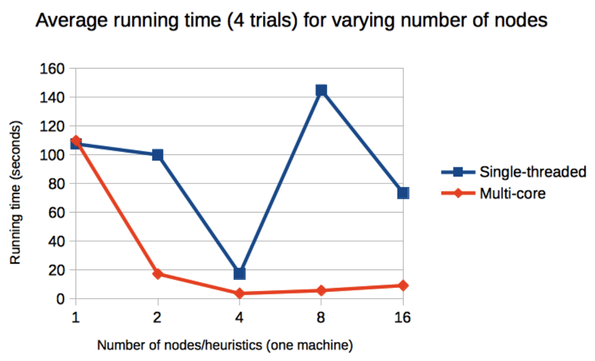
$bp(s)$ represents the parent of s along the optimal path found so far from the start to s. As a single pointer cannot be shared across a distributed replicated data structure, our first implementation used a full state representation (5x5 sliding tile puzzle) in place of $bp(s)$, nearly doubling the length of network messages. However, notice that $bp(s)$ is not needed to perform the search and compute g-values; it is only needed to reconstruct the solution path once the search terminates.

Therefore, it suffices to use pointers referring to the machine's local entry for that state. These pointers need not be communicated until the search terminates. At that point, the path can be obtained by walking backwards from the goal. At each step, we move from the current state s to the neighbor s' which minimizes $g(s') + c(s', s)$ among the possible parents $s' = bp(s)$ pooled from all machines. Although we still end up sending full state representations, these are only for the parents of states along the path.

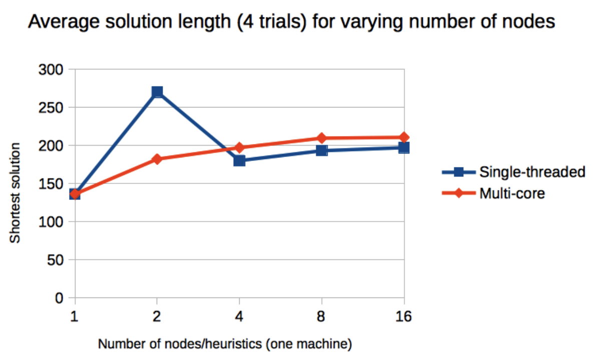
5 Experimental Setup

We tested our algorithm on a set of four random sliding tile puzzles of size 5x5, averaging the results of each trial into a single result. The data we collected were (1) running time, (2) number of states in the problem graph that were expanded during the course of searching for a solution, and (3) the length in steps of the best solution at termination.

In MHA*, two weights are specified. The first weight (w_1) determines how much priority to give heuristic expansions over the anchor search. A higher w_1 results in more depth-first-search-like behavior. The second weight (w_2) determines how close to the current known minimum path length a path to the goal must be before being selected as a solution. MHA* gives a guarantee that the solution found at termination is at most $w_1 * w_2$ times the optimum value. For all experiments, we used w_1, $w_2 = 2$. Note that when there is only one heuristic in a test (the admissible heuristic), the optimal shortest path to the goal will be found by only having to expand each state at most once (guaranteed by running A* with an admissible heuristic) and neither w_1 nor w_2 have any effect.

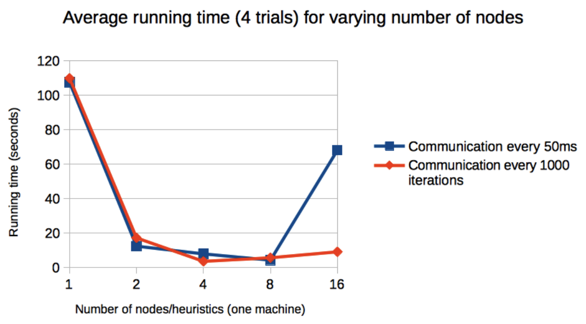


(a) Running time

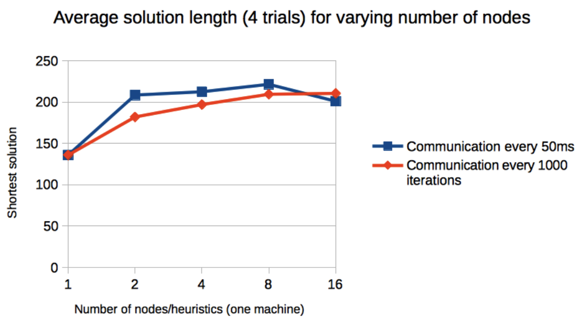


(b) Solution length

Figure 3: Comparison with Runtime performance and solution quality as the number of different heuristics (cores) on a single machine increases.

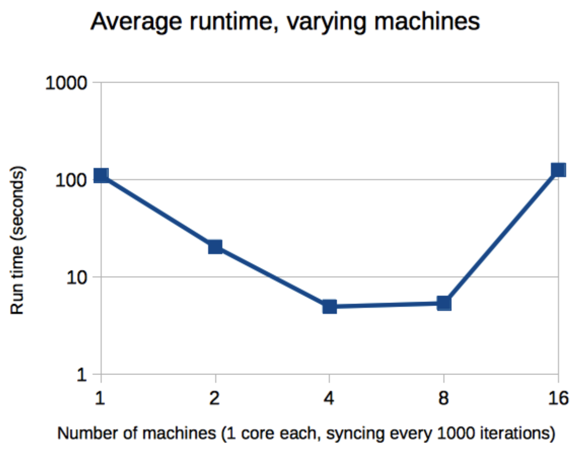


(a) Running time

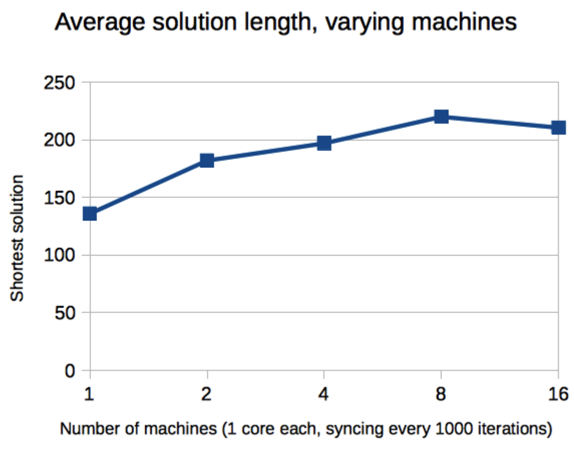


(b) Solution length

Figure 4: Runtime performance and solution quality as the number of different heuristics (cores) on a single machine increases.

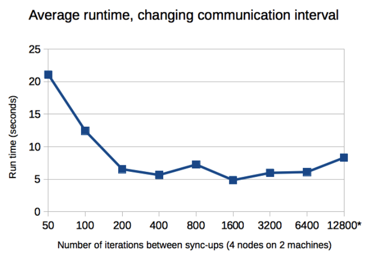


(a) Running time

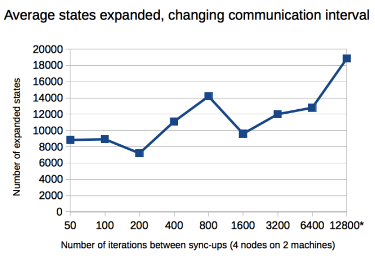


(b) Solution length

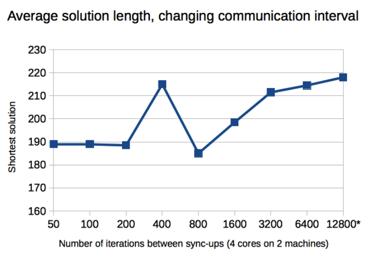
Figure 5: Runtime performance and solution quality as the number of different machines (with one process each) increases from 1 to 16.



(a) Running time



(b) Number of state expansions



(c) Solution length

Figure 6: Runtime performance, solution quality, and number of states expanded by the heuristic machines as the number of iterations between communication increases. *At the high end (12800 iterations between sync-ups), one of the four trials timed out (failed to find a solution within 5 minutes). The values in this graph are the average of the remaining three trials.

Single-threaded SMHA* implementation			
# cores	time	# expand	soln len
1	1	2	3
2	4	5	6
4	7	8	9
Parallelized SMHA* (single machine, syncing every X iterations)			
# cores	time	# expand	soln len
Distributed SMHA* (one process per machine, syncing every X iterations)			
# machines	time	# expand	soln len
Distributed SMHA* with varying frequency of sync-ups			
(4 processes across 2 machines)			
# iterations	time	# expand	soln len

Figure 7: Data from our trials

6 Results

7 Conclusions

References

[1] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev, "Multi-heuristic A*," in *Seventh Annual Symposium on Combinatorial Search*, 2014.

[2] M. Phillips, M. Likhachev, and S. Koenig, "PA*SE: Parallel A* for slow expansions," in *International Conference on Automated Planning and Scheduling*, 2014.