CSE539: Final Project Report

Patrick Naughton[1], Yao Yiheng[1], and Aidan Kelley[1]

[1]Washington University in St. Louis, May 4, 2020

I. Background

A* is a popular best-first search algorithm. It utilizes information from a heuristic $h(s)$ to guide a graph search from a start state to a goal state [3]. Intuitively, this heuristic is a guess at how far away a given node is from the goal. The algorithm explores the graph by iteratively expanding a state, which means to explore that state's immediate successors. For each state we compute a cost-to-come, notated $g(s)$, which is the cost of the current shortest known path to that state. To decide which state to expand next, we look at the state which has the lowest $h(s) + g(s)$ value, which is an estimate of the length of the shortest path to the goal through this state. This is a "best-first" algorithm because the nodes that are "best," meaning that we estimate they result in the shortest path to the goal, are explored first.

There are two broad classes of heuristics: consistent and inconsistent. A consistent heuristic satisfies $h(s) - c(s, s') \leq h(s') \quad \forall s \in S, s' \in \textsc{Succ}(s)$ where $c(s, s')$ is the cost of an edge from s to s'. An example of such a heuristic would be distance as the crow flies in the case of travel directions. Consistent heuristics are desirable because they guarantee that every $g(s)$ value we compute is optimal, meaning that our path to the goal will be optimal [3].

A heuristic that does not satisfy this property is inconsistent. While the property of optimality is obviously desirable, informative consistent heuristics are generally difficult to design [6]. In practice, it is often desirable to relax the consistency constraint in favor of a more informative heuristic. This decision forfeits optimality guarantees but can result in dramatically reduced search times by more greedily steering towards the goal. One important class of inconsistent heuristics are consistent heuristics scaled by some factor w. These heuristics guarantee a w-optimal solution and often considerably reduce planning time [2]. Algorithms that use these weighted heuristics are often called wA* algorithms.

Multiple Heuristic A* (MHA*) improves on the traditional A* (and wA*) algorithm by leveraging several arbitrarily inconsistent heuristics to guide the search to a quick solution while still providing a w-optimality guarantee [1]. In high-dimensional, heterogeneous search spaces it is generally very difficult to design a single consistent heuristic that provides accurate cost estimates in all parts of the space. To address this issue, MHA* uses n inconsistent heuristics in addition to a consistent one. It performs a normal wA* search using the consistent heuristic but will expand a state from an inconsistent heuristic if its cost estimate is low enough. The use of the consistent heuristic allows MHA* to make the same w-optimality guarantee as wA*. In practice, the addition of the inconsistent heuristics tends to speed up the search by driving the algorithm towards the goal and reducing the required number of state expansions [1]. **Note: In this report, we use the terms "inadmissable" and "inconsistent" interchangeably**

Improved Multi-Heuristic A* (IMHA*) [6] makes MHA* more robust by allowing it to make better use of *uncalibrated* heuristics: heuristics that may convey useful information to the search but do not lie on the same scale as the costs of edges in the search graph. For example, a heuristic may rank states in which a robot has its arms folded in to fit through a door as better, but this ranking has no direct connection to path costs [6]. IMHA* does this by leveraging the separation of search priority-queues from MHA* so that heuristics serve as ranking functions without being combined with cost-to-come measures.

II. Approach

In IMHA*, heuristics independently decide which states to expand next. However, when a state is expanded, it's neighbors are put into an OPEN set that is shared among the states. We actually have an OPEN set for each heuristic, ordered by the heuristic value, so that lookup of the minimum element can be quick, but these ordered sets must all be coordinated so that they contain the same values. The inconsistent heuristics might also update the g-value of a state which is shared across the sets. For these two reasons, we had to make significant changes to the original algorithm to elicit significant parallelism, but we attempt to maintain several important invariants provided by [6]. Below, we address how we maintained the invariants that IMHA* uses to provide its theoretical guarantees and introduce additional ones necessary to make these guarantees during parallel execution.

- All of the ordered OPEN sets contain the same elements. This property is maintained with the use of two list reducers, discussed in Section III-F
- No state is expanded more than twice. This is true since a state is expanded at most once inadmissibly and at most once admissibly. We discuss this in Section III-E.

Updates to the OPEN sets are delayed (see Section III-F), which changes the operation of the algorithm in that later searches in a given iteration do not get to see states discovered by earlier inadmissible expansions. Thus, these searches may choose less optimal states for expansion when compared to the serial version of the algorithm.

Additionally, since all $n+1$ expansions occur in parallel, there is a possibility that several of the expansions will discover the same state. Since we only keep the state with the lower g value, the work done to discover this state was wasted. Thus, this algorithm achieves the best parallelism when the heuristics are diverse, meaning they generate substantially different orderings of the states in OPEN. This means it is more likely that the different rounds of expansion will select states far from each other and so the new states discovered will all be unique.

Our algorithm is presented below:

```
procedure ExpandState($s$, OPEN$_i$)
    Add $s$ to REMOVE
    for all $s' \in$ SUCC($s$) do
        if $s'$ not seen before then
            $g_{\text{candidate}} \leftarrow g(s) + c(s, s')$
            Insert $s'$ into OPENUPDATE
        else if $s \notin$ CLOSED$_a$ then
            $g_{\text{candidate}} \leftarrow g(s) + c(s, s')$
            $g_{\text{current}} \leftarrow g(s')$
            if $g_{\text{candidate}} < g_{\text{current}}$ then
                Lock( )
                if $g(s) + c(s, s') < g(s')$ then
                    $g(s') \leftarrow g_{\text{candidate}}$
                    if $s' \in$ OPEN$_i$ then
                        Update $s'$ in OPEN$_0$
                    else
                        Insert $s'$ into OPENUPDATE
                    end if
                end if
                Unlock( )
            end if
        end if
    end for
end procedure
procedure Main
    OPEN$_0, \ldots$ OPEN$_n \leftarrow \emptyset$
    CLOSED$_a \leftarrow \emptyset$, CLOSED$_u \leftarrow \emptyset$
    $g(s_{\text{start}}) \leftarrow 0, g(s_{\text{goal}}) \leftarrow \infty$
    Compute heuristic values for $s_{\text{start}}$
    Insert $s_{\text{start}}$ in each OPEN$_i$
    while not Term-Criterion($s_{\text{goal}}$) do
        if OPEN$_0$.Empty( ) then return null
        $s_a \leftarrow$ OPEN$_i$[0]
        set $s_a$(CLOSEDFLAG$_u$)
        CLOSED$_a \leftarrow$ CLOSED$_a \cup \{s_a\}$
        CLOSED$_u \leftarrow$ CLOSED$_u \cup \{s_a\}$
        spawn ExpandState($s_a$, OPEN$_1$)
        for parallel $i$ in $1, 2, \ldots n$ do
            $j = 0$
            repeat
                $s_i \leftarrow$ OPEN$_i$[$j$++]
            until P-Criterion($s_i$) and atomic TAS $s_i$(CLOSEDFLAG$_u$)
            ExpandState($s_i$, OPEN$_i$)
        end for
        sync
        for parallel $i$ in $0, 1, \ldots n$ do
            for $s$ in OPENUPDATE do
                $h \leftarrow$ Heuristic$_i(s)$
                $p \leftarrow$ Priority(s) if $i == 0$ else $h$
                add $s$ to OPEN$_i$ with priority $p$
                break ties by lower $g$
            end for
            for $s$ in REMOVE do
                remove $s$ from OPEN$_i$
                if i == 0 then
                    CLOSED$_u \leftarrow$ CLOSED$_u \cup \{s\}$
                end if
            end for
        end for
    end while
end procedure
```

This algorithm derives all of its parallelism from the number of heuristics used to expand states. As a result, the amount of parallelism is constant throughout the running of the algorithm and is actually quite low for a reasonable number of heuristics. For n heuristics (including the consistent one) our parallelism is upper bounded by $n/lg(n)$ where the lg term comes from the binary spawning performed by the `cilk_for` loop. Because of this, we may actually violate the key assumption used to justify the work-first principle in that we do not have ample parallel slack. This could result in our algorithm achieving worse scalability and speedup than expected.

Note that we make calls to TERM-CRITERION(s), P-CRITERION(s), and PRIORITY(s). The original algorithm defines three different variants by modifying the behavior of these functions. We originally implemented the "Unconstrained" variant but switched to "MHA++" [6]. We made this transition because Sokoban has states from which it is impossible to find a solution. In the Unconstrained algorithm, it's possible that we select these states for inadmissible expansion, even though we know that this would be wasted work. By limiting the candidates for expansion in MHA++, we avoid doing this extra work.

III. IMPLEMENTATION DETAILS

III-A Keeping track of states and associated values

In our implementation, states are generated by the SUCC($\cdot$) function, which is agnostic to the implementation of the algorithm. However, there is other information we want to store about states, so typically, when we use states in our algorithm, they are wrapped in the `AugmentedState` class, which includes the current best g-value of the state, a pointer to the parent state (used to build the solution), a vector of this states $n+1$ heuristic values, and additionally a flag indicating membership in CLOSED$_u$, used in the parallel version.

III-B OPEN

The original algorithm has just one OPEN set and at each iteration constructs P-SET, another set from which it selects states to expand [6]. We represent OPEN using $n+1$ different ordered sets, where n is the number of inadmissible heuristics. These are `std::sets`, implemented in the STL as Red-Black trees. Each set contains shared pointers to `AugmentedStates` and stores them in ascending order according to their heuristic function scores (except for OPEN$_0$ which orders states according to PRIORITY(s)). All of the sets contain exactly the same states. We chose this representation because it allows us to avoid explicitly constructing P-SET each iteration, which could require iterating through the entire, potentially very large, OPEN set.

III-C CLOSED$_a$ and CLOSED$_u$

Both CLOSED$_a$ and CLOSED$_u$ are represented as unordered sets of states, with CLOSED$_u$ actually being an unordered map of states to `AugmentedStates`. These are `std::unordered_set` and `std::unordered_map`, implemented as hash tables. We use these sets to lookup states when we discover them during expansion. In particular, we will never select a state for inadmissible expansion if it is already in CLOSED$_u$ to ensure that we inadmissibly expand a state at most once. Inside EXPANDSTATE($\cdot,\cdot$), we will only add a state into OPEN if it is not already inside CLOSED$_a$ to make sure that we only ever admissibly expand a state at most once. These guarantees ensure that we expand a given state at most twice: once inadmissibly, and once admissibly.

For the parallel version of our algorithm, we additionally maintain the flag CLOSEDFLAG$_u$. CLOSEDFLAG$_u$ is an atomic variable that is used to ensure that the same state is not selected by multiple heuristics for expansion. This is implemented by `std::atomic_flag`.

III-D OPENUPDATE and REMOVE

We use two list reducers to track updates in EXPANDSTATE($\cdot,\cdot$) which will be played back after threads have synced. OPENUPDATE tracks states that need to be inserted into the OPEN sets, and REMOVE keeps track of states that need to be removed. These are implemented as `cilk::reducer<cilk::op_list_append<·>>`.

III-E Picking States to Expand

The original algorithm constructs P-SET, a set of potential states to explore, each iteration. Instead of constructing this set then picking the best states for each heuristic, we instead pick the best value in each OPEN set that satisfies the criteria of P-SET, meaning that P-CRITERION(s) is true and $s \notin$ CLOSED$_u$. Because we select these in parallel, we must pay careful attention to ensuring that we do not inadmissibly expand the same state twice. To coordinate between threads, we use the atomic flag CLOSEDFLAG$_u$. When we find an element in some OPEN set we decide we'd like to expand inadmissibly, we do an atomic Test-and-Set operation on its flag, and expand it only if it succeeds. If it does not succeed, we look at the next best candidate. Atomicity guarantees CLOSEDFLAG$_u$ will appear cleared to at most one thread, ensuring it is expanded inadmissibly only once. Similarly, we make sure that a state is only expanded admissibly once using CLOSED$_a$, but since it is only modified serially operations do not need to be coordinated.

III-F Coordinating updates to OPEN

In EXPANDSTATE($\cdot,\cdot$), states need to be inserted and removed from OPEN. However, doing this within the loop would require coordination via locks and could be incredibly expensive, since operations that require changes to the OPEN sets are frequent. Instead, we record changes that need to be made in OPENUPDATE and REMOVE and then record those changes after all calls to EXPANDSTATE($\cdot,\cdot$) have finished. We play the changes back in parallel in a way that requires no locking, since each OPEN$_i$ will be assigned uniquely to a thread. If there are any repeated states in OPENUPDATE, we deterministically choose the state with the lowest g-value. Doing updates to the OPEN sets in this way means that all OPEN sets contain the same states.

III-G Coordinating updates to g

In the case that we re-observe a state we have seen before, we might have to update it's g-value. However, updating a g-value requires an update to OPEN$_0$, meaning we must somehow coordinate between threads. We could again accomplish this with another reducer, but we hypothesised that g-value updates were probably rare, meaning these updates could be efficiently coordinated with locking.

On line 11 of our parallel algorithm we acquire a lock before updating the g value of s'. We do this because it is possible that another state that is being expanded this round also has s' as a successor and can decrease its g value. If we have already seen s', both of these threads could try to modify the same g value simultaneously, meaning that we could retain the wrong (larger) value. Furthermore, in some cases we will also need to update OPEN$_0$ which can only be done safely by one thread at a time.

There is a possibility that this synchronization will completely serialize threads' expansions and eliminate the parallelism. However, we reason that this lock is actually acquired relatively infrequently. For a thread to grab the lock, it needs to have found a better path to a previously explored state. While possible, in many domains, the most direct path to a state is also the cheapest one. The purpose of the heuristics is also to guide expansions towards the cheapest overall path. Thus, we expect that better paths to a given state will be discovered relatively rarely. The data support this conclusion as well. Across 5 random Sokoban puzzles, fewer than 15% of state discoveries result in updates that require grabbing the lock. Thus, in the common case, the threads can operate relatively independently without any synchronization.

IV. Evaluation

IV-A Sliding Tile Puzzles

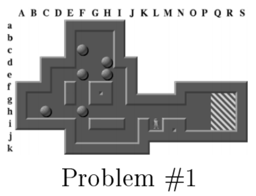
Sliding tile puzzles, also known as the 8-puzzle in the $d = 3$ case, are puzzles on a $d \times d$ grid. There are $d^2 - 1$ tiles and 1 empty square. The puzzle can be manipulated by sliding a tile that is adjacent to the empty square into the position that was empty. The puzzle is initially scrambled, and the goal of the puzzle is to perform slides to return it to its original state.

We collected data for the runtime of solving a 5×5 sliding tile puzzle using different numbers of processors with $w = 100$. To generate each problem instance we randomly scrambled the puzzle for $1,000$ moves. Each run of the algorithm attempted to solve the puzzle from the same starting configuration. We used the Manhattan distance plus linear conflicts as the consistent heuristic and a modified Nilsson sequence score [7] along with 5 pattern databases as inadmissible heuristics. Unfortunately, the data had such a high variance that they don't say anything about the scalability of our algorithm (on 8 processors, the standard deviation across 5 runs is about 95% of the mean). This appears to mostly result from large fluctuations in the number of states expanded. Indeed, on 8 processors, the number of states expanded varied from approximately $5,000$ to approximately $30,000$ across the five runs. Interestingly, we did not observe this wild variation when running on just one processor. Additionally, we tested the effect of serializing the selection of states to inadmissibly expand which substantially reduced the variation in the number of states expanded. Unfortunately, this modification also essentially eliminated the scalability of the algorithm, suggesting substantial work is done in the selection of states to expand.

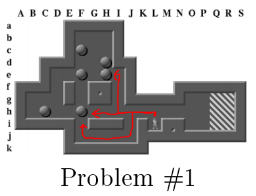
Based on these results, it seems that the ordering of selection of states to inadmissibly expand substantially impacts how many states are eventually expanded. When we select these states in parallel, we ensure that the same state is never selected twice in the same iteration. This means that other heuristics can "steal" the state that a given heuristic prefers to expand. This could cause a heuristic to choose a relatively bad state to expand so that its successors' g values are very unoptimal. When the anchor search does its expansions then, it is much more likely that it will reinsert these states into OPEN because it finds that it can decrease their g values. This in turn means that we have to perform more admissible expansions to find the goal. This dependence on ordering means that scrambling the state selection process can result in extremely variable execution times.

IV-B Sokoban

Sokoban is a Japanese puzzle game translating to "warehouse keeper" in English. The board is a square grid with walls. On the board are b movable boxes and b goal squares. The goal of the player is to get the boxes to be on the goal squares. The player can move in four directions and can push the boxes by standing on one side of them and moving in that direction. The player can push only one box at once and a box cannot be pushed if the position it would land on is another box or a wall.



(a) Thinking Rabbit #1



(b) First move actions

Fig. 1: Sample Sokoban Puzzle

If a box ends up in a corner, it can no longer be moved. As certain sequences of actions lead to an unsolvable state, the search graph of Sokoban is directed [5].

We chose to implement Sokoban because it is more comparable to true robot motion planning due to it being P-SPACE complete [5]. Additionally, we thought it might be a better test of our algorithm to pick an application where the expand state requires more work, as this means the program will spend more time doing real work rather than syncing. In our implementation of states and successors (see below), the branching factor of Sokoban is up to $4b$, computing successors takes non-trivial time, and reasonable heuristics are both more plentiful and complicated. These factors contribute to a more costly $\textsc{Expand}(\cdot, \cdot)$ method. The fact that we use more heuristics and that a larger portion of our total computation time is spent computing heuristics contributes to the parallelism.

Sokoban solvers are typically written as A* agents with packing heuristics[1], pattern matching databases to identify deadlock situations and extensive pruning of the action space. A consistent heuristic that tightens the lower bound is difficult to compute and at times falls far below what is feasible [5, sec. 3.2.3]. We implement minimal pruning of viable actions to test if IMHA* with multiple simple inadmissible heuristics can yield better results than our baseline A* in this more complex environment.

We represent each Sokoban state using 3 sets: boxes, walls, and goals. We also store a hashmap of player-accessible locations and the associated minimum costs. By hashing the locations of boxes, walls, goals and player-accessible locations, we fully describe similar states. States that do not differ in keys in these 3 sets and hashmap cannot have different successors.

In the previous example, Figure 1a, we do not generate successors as movements of the player at cell Li in the cardinal directions, rather, we generate actions that push boxes the player can reach. This is illustrated in Figure 1b (3 successor, action pairs for the first move). Each action we generate is a player movement onto a location that a box is currently occupying and an associated push direction. Therefore, an upper bound on the branching factor of our implementation of Sokoban is $4b$ since each box can be pushed in 4 directions.

The consistent, admissible heuristic used in our implementation is the BFS-distance from each box to the nearest goal cell ignoring collisions with other boxes. This serves as a lower-bound estimate on our cost-to-goal. Furthermore, we prune off obvious unsolvable states by only propagating this BFS to locations we can actually push a box to. We know that if we want to push a box in a particular direction, the cell adjacent to the box in the opposite direction of our push must not be a wall or box. If a box cannot be pushed to any goal even ignoring collisions with other boxes, there is no way we can arrive at a solution and we mark this state as unsolvable by returning an infinite value. As a result, this state will never be expanded unless the puzzle does not have a solution (our queue contains only infinite-valued states).

The inadmissible heuristics used for Sokoban are similar to the pattern database heuristic used for the Sliding Tile Puzzle. First, we construct K sequences of L_k reverse-actions from the goal-state of a Sokoban puzzle (where all boxes are in all goals and the player is in any location) to create K stored patterns with known-cost solution paths (sum of the series of actions). L_k are integers sampled uniformly in the range $[1, 10)$[2]. Next, a distance estimate to each stored pattern is treated as a heuristic score. This distance estimate between two states is computed via the BFS-distance from each box to the nearest box in this stored pattern similar to our consistent heuristic. However, this heuristic is inadmissible since the random series of actions may overestimate the actual cost-to-goal when we arrive at a given stored pattern. Nevertheless, these heuristics are helpful in driving our search towards multiple possible states with known paths to a solution. Furthermore, we can vary the range we sample L_k from and the size of our database K to generate more diverse and a greater number of heuristics respectively.

V. Results - Sokoban

We test our algorithm's performance on Sokoban using a set of 107 puzzles of increasing least-moves (estimated measure of difficulty), the SokEvo puzzle col-

[1]Prefer actions that places boxes next to each other on goal squares

[2]An obvious improvement would be to run a low depth search to generate these patterns to be more spread out since there are many more possible patterns at depth-9 than at depth-1. A uniform sampling like we used here for simplicity will likely generate some repeated low-depth patterns.

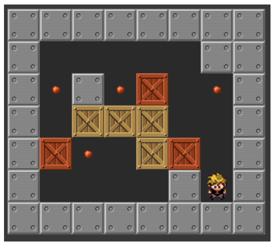
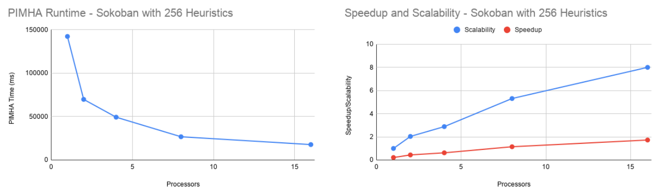


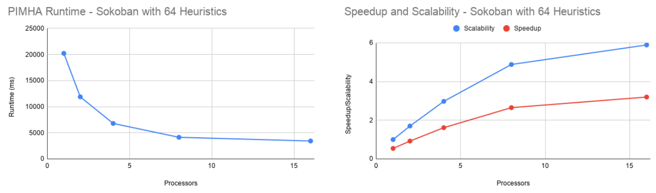
Fig. 2: Testing Sokoban Level (SokEvo #73) [8]



(a) (b)

Fig. 4: Results for Sokoban using 256 heuristics.

lection [4]. In particular, we chose puzzle 73 illustrated in Figure 2 which had a balanced level of difficulty (optimal moves found using A* takes 125 steps) against total runtime. For reference, with a constraint of two minutes execution time running on 16 cores, our implemented parallel IMHA* algorithm with 64 heuristics is able to solve 85/107 puzzles [3] compared to 69/107 [4] for weighted A* and 61/107 for serial IMHA* with 64 heuristics [5]. Note that we have not used any of the extensive pruning strategies involved in most Sokoban solvers and our pattern database heuristics are rather simple. Additional pruning and tightening the lower-bound estimate by our consistent heursitic would be helpful in solving more puzzles. However, we show here the feasibility of using simpler inadmissible heuristics to help guide a search for Sokoban.



(a) (b)

Fig. 3: Results for Sokoban using 64 heuristics.

Figure 3a shows the average runtime of our parallel implementation across 10 runs. The standard deviation for runs using up to 8 processors was at most 3% and was about 6% using 16 processors. We believe this results from previously mentioned variance in state expansions induced by the variability in the ordering of state expansions. This problem gets worse when more processors (or heuristics) are used. This variance is likely much less than was observed in the Sliding Tile Puzzle because Sokoban has many states from which it is impossible to find a solution. The search algorithm will never choose to expand these states so it is less likely that a long string of wasted states will be expanded without running into a dead-end. In contrast, any action in the Sliding Tile Puzzle preserves solvability, so it was possible for expansions to keep occurring that were very far from the solution. Figure 3b shows the scalability our algorithm achieved solving a Sokoban puzzle using different numbers of processors and 64 heuristics. Our scalability is quite sublinear. We believe this is primarily a function of the limited parallelism present in this algorithm. With 64 heuristics, our parallelism is only about 10.7, meaning that when using 16 processors we have extracted most of the opportunity for parallel execution. Our speedup is also substantially worse than our scalability. Examining the expansions performed by the serial compared to the parallel versions reveals that the parallel execution expands about 15% more expansions than the serial one. This is likely due to the delayed update causing threads to select relatively bad states for execution so that a significant amount of wasted expansions occur.

To test the effect of adding heuristics, we ran on the same puzzle with 256 heuristics which gives us an upper-bound for parallelism of $\frac{256}{log_2(256)} = 32$. Figure 4a shows the average runtime across 5 runs for these experiments. The larger number of heuristics actually makes the algorithm worse at solving the puzzle: the runtime for each processor count was higher than when only 64 heuristics were used. Additionally, the variability increases to about 20% on 16 processors. This result is expected since there are more heuristics meaning there are more possible interleavings of searches and it is more likely that some searches will be forced to expand highly unoptimal states. We see that the scalability is now closer to linear for up to 16 cores in Figure 4b.

However, we observe a drop in speedup as compared to the version running with 64 heuristics. One major

[3]Parallel IMHA* Unsolved puzzles: 48, 57, 64, 65, 66, 67, 76, 81, 82, 84, 85, 86, 87, 88, 89, 90, 94, 96, 99, 103, 106, 107

[4]wA* Unsolved puzzles: 29, 36, 39, 40, 42, 44, 46, 47, 48, 49, 53, 54, 57, 58, 64, 65, 66, 67, 68, 72, 76, 81, 82, 84, 85, 86, 87, 88, 89, 90, 92, 94, 96, 99, 103, 105, 106, 107

[5]Serial IMHA* Unsolved puzzles: 28, 29, 34, 36, 40, 41, 42, 46, 48, 49, 53, 54, 57, 58, 64, 65, 66, 67, 68, 69, 75, 76, 77, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 94, 95, 96, 97, 99, 100, 103, 104, 105, 106, 107

difference in our serial implementation is that heuristics in the same iteration can expand successors from a previously evaluated heuristic which could advance a path closer towards the goal by multiple steps in one iteration (of evaluating our heuristics). This is not available in the parallel implementation: due to the delayed update to OPEN, we only ever take 1 additional step on a given path. Furthermore, with 256 heuristics, our pattern heuristics are likely very similar. Once we start to saturate the space of possible patterns, many of the heuristics we compute will be very close or even overlapping since we did not check for duplicate patterns. This results in a lot of extra non-advancing work done in expansions by our heuristics which leads to a slowdown compared to the serial version we implemented. In most runs, the parallel version would expand more than twice as many states as the serial version. These results further emphasize the importance of having numerous but diverse heuristics.

Overall, this search algorithm proved relatively difficult to parallelize and did not experience massive benefits from parallelization. The fundamentally sequential nature of the original algorithm meant that we had to make significant modifications to its operation to elicit parallelism. While these changes did enable some speedup, they also made the algorithm less efficient and less reliable. With all of this said, our parallelization does still offer some benefits in solving Sokoban puzzles. Even just the 3 times speedup achieved on 16 processors allowed our parallel implementation to solve 16 more puzzles than a wA* implementation and 24 more puzzles than a serial implementation of IMHA*. Our parallel implementation also solved every puzzle solved by either wA* or our serial implementation, meaning that the parallel implementation has an advantage over these algorithms across a wide range of puzzles.

VI. Discussion and Future Work

Our implementation of this algorithm was plagued by poor speedup and high variance, which makes the poor speedup hard to improve on. We found that if we chose which states to expand in serial but still expanded them in parallel, that the algorithm consistently did the same amount of work to find the solution. However, having the large serial section added a large amount to the span–effectively killing parallelism. To get a better sense of what the scalability would be, we paused the clock for the section where the states to be expanded are chosen. We chose not to include those numbers in this report as we do not feel they fairly represent the absolute performance of the algorithm, but we see the low variance version as a platform to performance engineer before pushing improvements back to the original version.

VII. Teamwork

While we all worked on everything, Yiheng led building the framework for search and implementing Sokoban and relevant Heuristics, Patrick led on implementation on the search algorithm, and Aidan led on designing the parallel version of the algorithm and writing the report.

References

[1] Sandip Aine, Siddharth Swaminathan, Venkatraman Narayanan, Victor Hwang, and Maxim Likhachev. Multi-heuristic a. *The International Journal of Robotics Research*, 35(1-3):224–243, 2016.

[2] R. Ebendt and R Drechsler. Weighted a* search–unifying view and application. *Artificial Intelligence*, 173(14):1310–1342, 2009.

[3] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[4] Lee Haywood. Sokoban evolution puzzle collection, 2003.

[5] Andreas Junghanns. *Pushing the Limits: New Developments in Single-Agent Searh*. PhD thesis, University of Alberta, 1999.

[6] Venkatraman Narayanan, Sandip Aine, and Maxim Likhachev. Improved multi-heuristic a* for searching with uncalibrated heuristics. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

[7] Nils J Nilsson. *Problem-solving methods in Artificial Intelligence*. McGraw-Hill, 1971.

[8] Thinking Rabbit. Online sokoban player. https://sokoban.info/?88_79.