EPiC Series in Computing

Volume 50, 2017, Pages 202–212

GCAI 2017. 3rd Global Conference on Artificial Intelligence

Enhanced Simplified Memory-bounded A Star (SMA*+)

Justin Lovinger and Xiaoqin Zhang

Computer and Information Science Dept,
University of Mass Dartmouth, MA

Abstract

In 1992, Stuart Russell briefly introduced a series of memory efficient optimal search algorithms. Among which is the Simplified Memory-bounded A Star (SMA*) algorithm, unique for its explicit memory bound. Despite progress in memory efficient A Star variants, search algorithms with explicit memory bounds are absent from progress. SMA* remains the premier memory bounded optimal search algorithm. In this paper, we present an enhanced version of SMA* (SMA*+), providing a new open list, simplified implementation, and a culling heuristic function, which improves search performance through a priori knowledge of the search space. We present benchmark and comparison results with state-of-the-art optimal search algorithms, and examine the performance characteristics of SMA*+.

1 Introduction

Despite the effectiveness of A Star (A*) [1, 2, 3, 4, 5, 6] as an optimal heuristic search method, large branching factor and deep solutions lead to an explosion of memory usage that cripples any modern computer. To this end, many A* variants aim to reduce memory complexity by addressing the need to maintain all generated nodes in memory.

Iterative Deepening A Star (IDA*) [7, 5, 6, 8, 9] solves the memory problem by combining depth first search with incrementally increasing f-cost limits. Allowing for an optimal search that need not maintain every node in memory. IDA* pays for its memory efficiency with increased time complexity, by constantly re-evaluating nodes as the algorithm resets with a higher cutoff. The addition of a transposition table [10] can enable IDA* to better utilize available memory, for increased performance.

Simplified Memory-bounded A Star (SMA*) [11, 12] takes a different approach. Instead of sacrificing time efficiency for minimal memory usage, SMA* recognizes that an algorithm only needs to use as much memory as the machine has available. Available memory should be utilized to improve the effectiveness of an algorithm. The result is a memory bound that allows for fast search while memory is available, and falls back on memory efficient methods when the bound is reached.

In this paper, we supply details of our enhanced SMA* (SMA*+), necessary for implementation. We describe the back-up procedure, using a forgotten successor table. We then present the usage of two priority queues, one for selecting nodes to expand, and the other for selecting

nodes to cull. We provide a detailed SMA*+ algorithm (Algorithm 1) that can be easily translated into a program. A culling heuristic function $c(n)$ is introduced, to help select nodes to delete when memory is full. This culling heuristic can be developed based on the characteristics of the problem search space. A good choice of $c(n)$ will reduce the search effort and improve the performance of SMA*+. Last but not the least, we provide analysis and evaluation of SMA*+. We analyze the time complexity and space complexity of SMA*+ theoretically, we experimentally compare SMA*+ with A*, SMA*, and IDA* on benchmark problems.

In the next two sections, we introduce the predecessor of SMA*, and SMA* itself. In Section 4 we explore the SMA*+ algorithm, providing details of implementation and performance analysis. In Section 5 we compare SMA*+ to popular optimal search algorithms. Finally, we conclude in Section 6.

2 MA*

As the state-of-the-art in optimal heuristic search was explored, memory efficient methods were in high demand. Although most algorithms focused on reducing the worst-case memory complexity, some took the approach of instead bounding memory usage. The idea is simple: an effective algorithm should utilize memory to reduce time, so long as memory does not exceed a given limit.

Memory-bounded A Star (MA*) [13] embodies this concept. MA* matches the behavior of A*, as long as sufficient memory remains. Once a bound is reached, MA* begins dropping high f-cost leaf nodes, freeing memory for more promising nodes. To prevent infinite looping, f-cost values are backed up through the tree. The result is a search method that explores the state space very quickly while memory is available, and relies on re-expanding some nodes when memory is limited. In practice, with many problems and a realistic memory limit, few nodes are re-expanded. The result is an efficient search with the guarantee that complicated state spaces will not exceed system memory.

Unfortunately, the original MA* was unnecessarily complicated, leading to the development of SMA*.

3 SMA*

SMA* takes the concept of MA* and simplifies the implementation. SMA* uses a binary tree to store the open list, while MA* uses a less efficient data structure. SMA* simplifies the storage of f-cost by eliminating the need for four f-cost quantities (as in MA*), and SMA* performs the backup procedure when a node is fully expanded, instead of once for each generated node. SMA* also simplifies pruning by removing only enough nodes to remain below the memory limit, instead of pruning many unpromising nodes at a time.

4 SMA*+ Algorithm

Since SMA* and SMA*+ must cull nodes to maintain the memory limit, an alternative means of maintaining progress is required. The original SMA* uses a backup procedure that updates the f-cost of all nodes in a branch of the search tree, every time a node is expanded. SMA*+ uses a more efficient mechanism, that only updates the f-cost of a single node, when a successor of that node is culled. Furthermore, forgotten successors that are re-added to the open list must receive an updated f-cost. To this end, forgotten nodes maintain their f-cost on their

parent node, in a table mapping state to f-cost. Note that action can be used instead of state, when implementing this table. The original SMA* makes similar progress by keeping removed nodes in memory, until their parent is removed. Our forgotten f-cost table is a more efficient mechanism, saving more memory space, sooner.

Algorithm 1 presents comprehensive implementation details for SMA*+. The user provides a memory limit M, and an initial node to expand. Every node contains a table mapping the state of each forgotten successor to the f-cost of this forgotten successor.

Unlike SMA*, SMA*+ fully expands a node each iteration, instead of adding only one successor to the open list every iteration. While adding one successor at a time may seem more efficient, the overhead required to determine which successor to add, adds unnecessary complexity and decreases performance with minimal memory advantage.

4.1 Culling Heuristic

When the number of expanded nodes u exceeds the memory limit M, SMA*+ must remove a leaf node from the open list O to maintain the memory limit. The original SMA* specifies: remove the max f-cost leaf in O, ties favor lesser depth. However, Algorithm 1 specifies removing the *worst* leaf node. This wording is intentionally vague. As long as the removed node is never the best node, SMA*+ is complete and optimal. The choice of node to cull only affects performance. As such, we present the culling heuristic $c(n)$. When removing the worst node, we must find the leaf node n that *maximizes* $c(n)$.

In many problems, the highest f-cost node may not be least likely to lead to the best goal, because an f-cost heuristic, $h(n)$, must be admissible. A separate culling heuristic, that is not constrained by the same criteria, can vastly improve performance, by minimizing re-expanded nodes. For example, a culling heuristic based on the ratio between f-cost and depth, given as $c(n) = f(n)/\ln(d(n) + e)$, where e is Euler's number, can improve performance on problems with many similar f-costs, if we can reasonably assume that the goal has a high depth.

We note that, to avoid looping, a culling heuristic must never cull the best node during any culling step. Formally, $c(n)$ must always meet the criteria that $\arg\max_{n \in L}(c(n)) \neq \arg\min_{n \in O}(f(n))$, where L is the set of leaf nodes and O is the open list. However, this criteria is very easy to verify during runtime, and a simple procedure can ensure any culling heuristic is, for all intents and purposes, admissible. Simply compare the worst leaf to the best node, and if they match, cull the second worst leaf, as shown in Algorithm 3.

4.2 SMA*+ Example

To clarify the SMA*+ algorithm, Figure 1 provides a step by step example with memory limit $M = 3$. Until iteration 3, when the memory limit is exceeded, SMA*+ is identical to A*.

At iteration 3 the worst node, B, is culled. $f(B)$ is then stored in the forgotten f-cost table of A and $f(A)$ is updated to the minimum of its forgotten children, $\min(f(B)) = 3$. Also in iteration 3, we see $f(D) = \infty$. We note that the traditional $f(D) = h(D) + g(D) \neq \infty$. However, since depth $d(D) \geq M - 1$, where $d(D) = 2$ and $M - 1 = 3 - 1 = 2$, and D is not a goal node, it will never reach a goal. Note that in iteration 5, $f(F) = 4$ instead of ∞ because F is a goal node.

Each new iteration expands a node, and removes the worst leaf nodes, storing the (state, f-cost) pair on the parent of the removed node. Finally, we re-expand A, and after expanding B, we find goal node F with the lowest f-cost at iteration 5, and are finished.

Algorithm 1 SMA*+ Algorithm

```
f(n): f-cost of n
g(n): path cost to n
h(n): heuristic value of n
s(n): state of n
d(n): depth of n

Let M be a given memory limit
O ← an empty open list                          ▷ See Section 4.3 for suggestions on open list
add the initial node to O
u ← 1                                           ▷ u is a counter for nodes in memory
while O contains nodes do
    b ← min f-cost node in O
    remove b from O
    if b is goal then
        return b
    else if f(b) = ∞ then
        return goal not found
    end if

    if b has been expanded then                 ▷ True when forgotten f-costs table is not empty
        N ← forgotten successors of b           ▷ Can be obtained from forgotten f-costs table of b
    else                                        ▷ b has not been expanded
        expand b
        N ← successors of b
    end if
    for node n ∈ N do
        if s(n) is in forgotten f-cost table of b then
            f(n) ← f-value of s(n) in forgotten f-cost table of node b
            remove s(n) from forgotten f-cost table of node b
        else if n is not the goal and (n has no successors or d(n) ≥ M − 1) then
            f(n) ← ∞
        else
            f(n) ← max(f(b), g(n) + h(n))
        end if

        add n to O
        u ← u + 1
    end for

    while u > M do
        Cull worst leaf in O                    ▷ See Algorithm 2
    end while
end while

if O is empty then
    return goal not found
end if
```

Algorithm 2 Cull Worst Leaf

$w \leftarrow$ worst leaf in O ▷ See Algorithm 3 and Section 4.1 for definition of worst leaf
remove w from O

$p \leftarrow$ the parent node of w
remove w from the successor list of p
add $s(w)$ to forgotten f-cost table of p, with value of $f(w)$
$f(p) \leftarrow$ min of forgotten f-costs of p ▷ Can be calculated incrementally, in $\mathcal{O}(1)$ time
if p is not in O **then** ▷ If necessary, nodes can remember if they are in O for $\mathcal{O}(1)$ time check
    add p to O
end if

$u \leftarrow u - 1$

Algorithm 3 Safe Culling Heuristic

$w \leftarrow$ worst leaf acc. $c(n)$ in O
if $w =$ best node acc. $f(n)$ in O **then**
    $w \leftarrow$ second worst leaf acc. $c(n)$ in O
end if

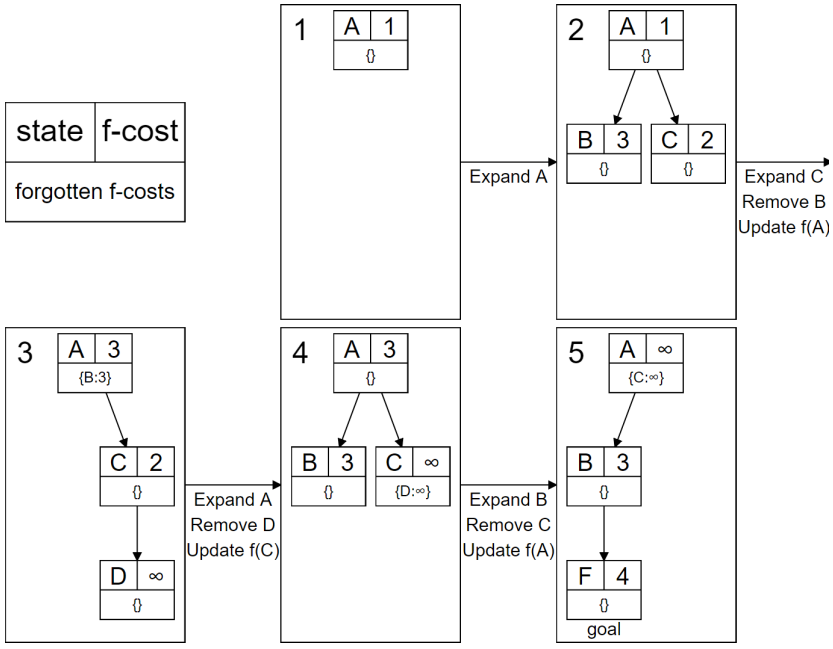

Figure 1: SMA*+ Example

4.3 Implementing the Open List

Special consideration must be given when implementing the open list. Most optimal search methods are concerned with only the minimization of the f-cost, or an analogous parameter. SMA*+, by contrast, must minimize f-cost when selecting the next node to expand, and maximize the culling heuristic $c(n)$ when selecting a node n to cull. This eliminates the simple use of popular and efficient heap structures like a priority queue, because a single priority queue cannot simultaneously select the minimum f-cost and the maximum $c(n)$ node.

The original SMA* uses a binary tree. This approach provides $\mathcal{O}(\log x)$ time to insert, remove, and pop the best successor, where x is the number of nodes in the tree. However, since this tree must maintain all nodes with successors, instead of only leaf nodes, the worst-case time to cull a leaf node is $\mathcal{O}(x)$, because we must search nodes in descending order of f-cost until a leaf node is found. Furthermore, this structure does not support a culling heuristic other than $c(n) = f(n)$, since the worst leaf and best node are obtained from the same data structure.

We present an alternative approach involving two priority queues. One ordered by ascending f-cost and containing all nodes not fully expanded, is used to select the next node to expand. A second queue, ordered by descending $c(n)$ and containing only leaf nodes, is used to select the next node to cull. This approach provides $\mathcal{O}(\log x)$ time to select and remove a node from the open or cull list. By maintaining two lists, removal and addition must be performed on both lists, resulting in worst-case $\mathcal{O}(2 \log x)$ time, when all nodes in the open list are leafs. Additionally, the position of each node in each queue should be tracked for efficiency. A naive approach, that does not track position, requires $\mathcal{O}(x)$ time to find the correct node to remove from the open list, once it is removed from the cull list, and vice versa. This time may be reduced to $\mathcal{O}(1)$ using a hashtable. Empirical evidence shows this approach consistently outperforms a binary tree.

For additional performance, the worst leafs priority queue can be created and maintained only once the memory limit is reached. This requires a one time $\mathcal{O}(x)$ operation to build the worst leafs queue from the next node queue, but requires no time to maintain the worst leafs queue before the memory limit is exceeded. Furthermore, if the memory limit is never exceeded, this open list has the same performance as A*.

4.4 Theoretical Performance

The greatest strength of SMA*+ comes from the ability to set an upper bound on space. Given memory limit M, SMA*+ clearly has worst-case space complexity $\mathcal{O}(M)$.

Assuming $M \geq G$, where G is the number of nodes generated, SMA*+ functions as A*, with the same $\mathcal{O}(b^d)$ worst-case performance, where b is the branching factor of the problem and d is the depth of the solution. When $M < G$, SMA*+ becomes difficult to analyze theoretically, as performance depends on how many nodes must be re-expanded. In the best case, where no nodes are re-expanded, worst-case performance remains $\mathcal{O}(b^d)$, but space complexity is $\mathcal{O}(M)$, a clear advantage given $M < b^d$.

Every node re-expanded adds to the cost of SMA*+, giving a worst-case $G \geq b^d$. However, since the size of the open list is bound by M, the cost of adding and removing elements from the open list is less than A*, when $M < G$. Empirical evidence (Table 1) shows that a smaller memory limit does not significantly reduce performance, even when nodes are re-expanded.

The empirical analysis in Section 5 shows that nodes are rarely re-expanded, even when M is significantly less than G. As such, for many problems, SMA*+ performance is minimally affected by M.

4.5 Completeness and Optimality

The following properties, same as SMA*, also hold for SMA*+, which prove the completeness and optimality of SMA*+ with sufficient memory available. We cite the original SMA* [11] paper for borrowing these properties.

Lemma 1. *f-cost of removed nodes are maintained to give a correct lower bound on the cost of solution paths through unexamined successors of a node with unexamined successors.*

Lemma 2. *SMA*+ always expands the node that has the best lower bound on its unexamined successors.*

Theorem 1. *SMA*+ is guaranteed to return an optimal solution, given an admissible heuristic, and memory limit $M \geq d + 1$, where d is the depth of an optimal solution.*

Theorem 2. *SMA*+ behaves identically to A* when its memory limit is greater than the number of nodes it generates.*

5 Benchmark and Comparative Analysis

The state-of-the-art in informed search is primarily dominated by A Star (A*) [2, 3, 4, 5, 6] and Iterative Deepening A Star (IDA*) search [5, 6, 8, 9]. A* makes no attempt to optimize memory usage, in exchange for speed, while IDA* is memory efficient, at the risk of decreased speed. In this section, we compare the performance of SMA*+ to these popular methods, and the original SMA*. This analysis shows that SMA*+, with even a *moderate memory limit*, has the *speed of A** and the *memory efficiency of IDA**.

5.1 Benchmark Problems

A number of common search problems are included to aid our algorithm comparison. The missionaries and cannibals (MC) problem is a simple problem with low branching factor and depth. Three missionaries and three cannibals stand on one side of a river, a boat can take up to two people, and at least 1, across the river at a time. The cannibals can never outnumber the missionaries, on either side, and the goal is to move all individuals to the right side. An example of state transitions for this problem is depicted in Figure 2a.

The next problem is a common game for recreation and search benchmarking: the sliding puzzle (SP) game. This problem presents an n by n grid of numbers, $0..(n^2 - 1)$. The numbers are initially arranged in random order, and the goal is to arrange them in ascending order for each row, by swapping adjacent numbers. Note that our goal state has the 0 number in the upper left corner. We present the more difficult 4 by 4 version of this problem, also known as the 15 puzzle. An example state is depicted in Figure 2b. A selection of 5 initial states from Korf's 100 SP instances [7] are benchmarked; instances 12, 42, 55, 79, and 97.

Finally, maze and random obstacles (Obst) pathfinding problems are included [14], depicted in Figure 3. Both maps are a 512 by 512 grid. White pixels can be passed though, while black pixels cannot. These maps are part of the pathfinding benchmark sets from Moving AI [15], titled maze512-1-0 (Maze) and random512-35-0 (Obst). Maze uses 100 problem instances, given by scenarios 4000 through 4100. Obst uses 5 problem instances, given by scenarios 65 through 70. Each scenario provides a different initial and goal position. See Moving AI for details of these problem instances. Step costs for diagonal movements are $\sqrt{2}$ and cardinal movements are 1. Agents cannot move diagonally through a wall.

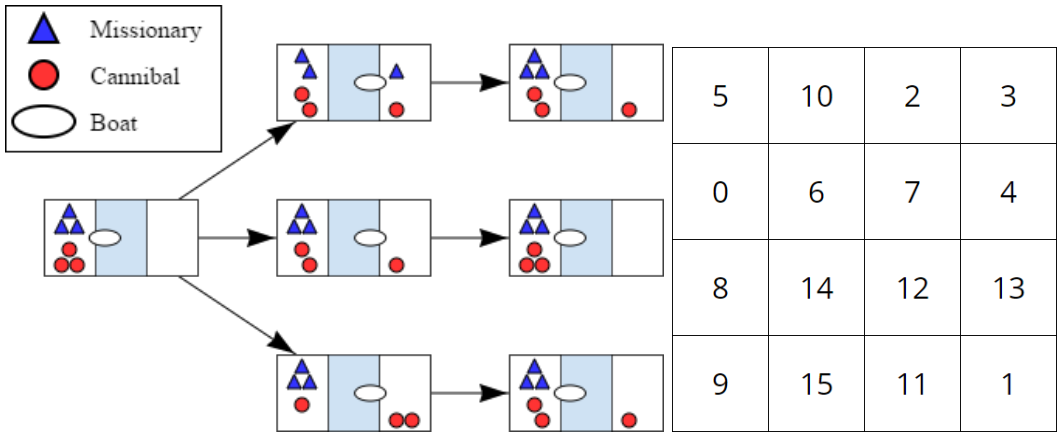

Figure 2: (a) MC Problem Example (b) SP Example

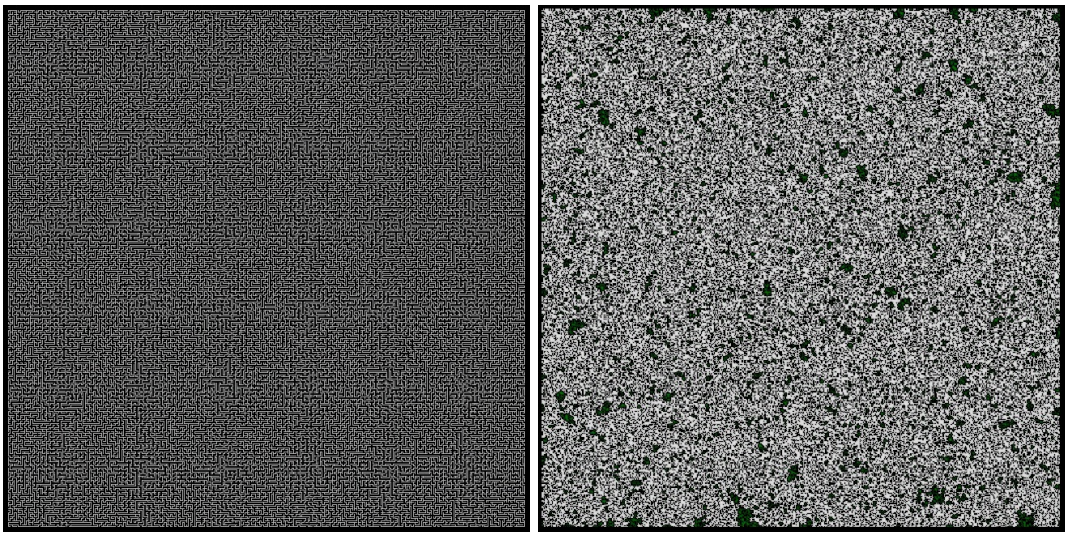

Figure 3: Maze and Obst Problems

Table 1 presents depth and breadth for the comparison problems. Problems with multiple instances present range of depth. Note that, to enable a comparison with the high memory usage A* algorithm, problems and problem instances are chosen to fit within 32gb of memory. As such, many exceptionally difficult problem instances are excluded. Despite this limitation, these problems represent a variety of search situations, and, as results show, many of these problems provide sufficient challenge for the state-of-the-art.

5.1.1 Benchmark Problem Heuristics

The MC heuristic $h(n) = (m_l^n + c_l^n)/2$, where m_l^n is the number of missionaries on the left for node n, and c_l^n is the number of cannibals on the left for node n.

The SP heuristic $h(n) = \sum_{t=1}^{m} d(p_t^n, g_t)$, where p_t^n is the position of tile t for node n, g_t is

Problem	Solution Depth	Problem Breadth
MC	11	2.36
SP	41 to 45	3
Maze	1603 to 1643	2
Obst	26 to 27	3.8

Table 1: Benchmark Problems

Problem	Algorithm	Node Limit	Nodes Expanded	Nodes Generated	Runtime	Memory
MC	A*	N/A	24	27	0.001	18448
	IDA*	N/A	92	110	0.002	14224
	SMA*	20	27	27	0.002	53408
	SMA*+	20	24	27	0.002	42704
SP	A*	N/A	186243	378218	40.281	345453680
	IDA*	N/A	454814	921290	23.099	59270
	SMA*	100000	267814	267814	3402.57	62618062
	SMA*+	∞	186153	378218	77.405	686098752
	SMA*+	100000	182560	369931	130.214	202568294
	SMA*+	5000	181982	367783	117.192	11023507
Maze	A*	N/A	40034	40098	5.540	23230625
	IDA*	N/A	459967092	460947921	22947.299	11410280
	SMA*	15000	89948	89948	534.837	112667149
	SMA*+	∞	40034	40098	8.172	351221604
	SMA*+	15000	40580	40649	10.876	148238555
Obst	A*	N/A	102931	221101	16.786	161488024
	IDA*	N/A	25324693	54139813	609.892	2182384
	SMA*	100000	1561313	1561313	3064.72	437034164
	SMA*+	∞	102931	221100	20.253	380394969
	SMA*+	100000	113193	243928	68.811	167010308
	SMA*+	50000	184538	389555	161.522	98382844

Table 2: Search Algorithm Comparison

the position of tile t in the goal state, d is the Manhattan distance function, and m is highest tile number. Note that we skip the 0 tile, which is necessary for an admissible heuristic.

The heuristic for all pathfinding problems $h(n) = \sqrt{(n_x - g_x)^2 + (n_y - g_y)^2}$, where n_x and n_y are the x and y coordinate of node n; and g_x and g_y are the x and y coordinate of the goal state. This heuristic is commonly known as euclidean, or straight line, distance.

For these problems, we do not take advantage of a custom culling heuristic $c(n)$, instead using the natural $c(n) = f(n)$. Not all problems can easily benefit from a culling heuristic other than $c(n) = f(n)$. However, it is important to note that the choice of which node to cull is purely a performance optimization, which does not affect the completeness or optimality of SMA*+. The option of a custom $c(n)$ is available should it prove beneficial.

5.2 Benchmark Results

Table 2 presents our comparison of A*, IDA*, SMA*, and SMA*+ search. Various SMA*+ memory (node) limits are tested, to examine the effect on performance. For each algorithm and problem, the number of nodes generated and expanded, runtime in seconds, and max memory usage in bytes are shown. These values are averaged over all instances of a problem, with the exception of IDA* on Maze, which, due to each instance runtime exceeding 1 hour, is only the value of scenario 4060. Note that SMA* and SMA*+ can expand the same node multiple times, and SMA* adds only 1 node to the open list for every node expansion. All presented algorithms avoid looping states. That is to say, node n is not added to the open list if $state(n) \in path(p)$, where p is the parent of n. This optimization is necessary for A* to complete with reasonable memory usage.

While IDA* excels on the simpler MC and SP problems, we see its performance severely lacks on pathfinding problems, in both nodes generated and runtime, especially the high depth maze problem. While the original SMA* has a low number of nodes generated on most problems, the complex implementation and inefficient open list proves extremely detrimental to runtime performance.

SMA*+ is significantly more efficient, featuring runtime performance comparable to A* on even the high depth and breadth pathfinding problems. Note that SMA*+ uses more memory per node than SMA*, due to the two priority queue open list. However, on each problem, an SMA*+ node limit with memory usage comparable, or less, than SMA* is presented. Even with a smaller node limit, SMA*+ shows significantly better runtime performance, compared to SMA*. SMA*+ similarity outperforms IDA* on all but the SP problem, and presents runtime comparable to the high memory usage A* on all problems. While not as memory efficient as IDA*, modern computers can easily provide enough memory for extremely fast SMA*+ search.

6 Conclusion

In this paper, we present SMA*+, an enhanced version of the SMA* algorithm originally presented by Stuart Russell [11]. A new open list data structure is provided, implementation is simplified, and a culling heuristic function is introduced. Our generalized culling mechanism allows a priori problem knowledge to significantly improve performance, often providing speed equivalent to A* with significantly reduced memory usage. Performance is analyzed, exploring the computational, memory, and search performance of SMA*+ through both empirical benchmarks and theoretical analysis.

Our algorithm comparison in Section 5 shows that memory limits easily achievable on modern hardware allow SMA*+ to achieve A* like performance with low memory usage, significantly outperforming the state-of-the-art in low-memory heuristic search. Implementation details in Section 4 and Algorithm 1 enable easy use of SMA*+, furthering the state-of-the-art in efficient optimal search.

References

[1] 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.

[2] Hiromasa Arai, Crystal Maung, and Haim Schweitzer. Optimal column subset selection by a-star search. In *AAAI*, pages 1079–1085, 2015.

[3] Imad S AlShawi, Lianshan Yan, Wei Pan, and Bin Luo. Lifetime enhancement in wireless sensor networks using fuzzy approach and a-star algorithm. *IEEE Sensors journal*, 12(10):3010–3018, 2012.

[4] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96:59–69, 2014.

[5] Tristan Cazenave. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *2006 IEEE Symposium on Computational Intelligence and Games*, pages 27–33. IEEE, 2006.

[6] Khammapun Khantanapoka and Krisana Chinnasarn. Pathfinding of 2d & 3d game real-time strategy with depth direction a algorithm for multi-layer. In *Natural Language Processing, 2009. SNLP'09. Eighth International Symposium on*, pages 184–188. IEEE, 2009.

[7] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[8] Stefan Schroedl. An improved search algorithm for optimal multiple-sequence alignment. *Journal of Artificial Intelligence Research*, 23:587–623, 2005.

[9] Stefan Edelkamp and Zhihao Tang. Monte-carlo tree search for the multiple sequence alignment problem. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

[10] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[11] Stuart Russell. Efficient memory-bounded search methods. *ECAI-1992, Vienna, Austria*, 1992.

[12] Stuart Jonathan Russell and Peter Norvig. Artificial intelligence: a modern approach (3rd edition), 2009.

[13] Partha Pratim Chakrabarti, Sujoy Ghose, Arup Acharya, and SC De Sarkar. Heuristic search in restricted memory. *Artificial intelligence*, 41(2):197–221, 1989.

[14] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[15] N. Sturtevant. Moving ai, 2012. http://www.movingai.com/benchmarks/.