A* Search Implementation for Blocks World

Matthew Grogan

October 2, 2017

1 Overview

We have been tasked with implementing an A* search solution to the Blocksworld problem.In this version of Blocksworld, we must arrange a finite number of blocks on a finite number of stacks such that the blocks are placed in ascending order on the first stack. To facilitate ordering, each block is given a label such as A or B and each stack is numbered. The last block on any given stack can be moved to the last position on any other stack. For simplicity's sake, I have replaced letters A, B, etc. with numbers 1, 2, etc.

2 Implementation

2.1 A*

My implementation of A* and Blocksworld is written in C++. I use three classes:

a. A state class containg stack and block information as well as member functions for moving and evaluating current block configurations.

b. A node class which contains a parent pointer, a state class variable, and a path cost estimate.

c. An A* search class containing a frontier priority queue, visited list, and the search algorithm.

2.2 Heuristic

2.2.1 Description

The heuristic I have constructed is presented below. We start our move counter with the number of blocks in the problem. We check the first stack and decrement the counter for each block in place. This gives us a simple number-of-blocks-out-of-place heuristic. But we know that each block out of place in the first stack must be moved again, so we increment count in this case. Next we check the other stacks. For each block in the stack, we look at each of the blocks

underneath it. If the current block is the larger of the comparison, we know that it must be moved to free up the burried block and we increment the move count.

```
Set count = # of blocks
For each block in stack one, do:
    if block is in position, decrement count
    else, increment count
For each stack past stack one, do:
    For each block in this stack, do:
        For each sub-block under this top-block,do:
            if sub-block < top-block, increment count
return count
```

2.2.2 Admissibility

This heuristic is not admissible. Consider a case such as
1 :
2 :
3 : A B C
In this case, the heuristic returns a value of 6 when the optimal solution is in only 5 moves. Even in this case, the heuristic guides us towards the correct solution by encouraging proper ordering and avoidance of the first row for incorrect blocks. The above case is not particularly likely and for smaller values (e.g. 3 stacks and 5 blocks). It becomes more prominant with larger numbers of blocks in proportion to stacks as this results in generally lasrger stacks.

3 Results

The solution performs quite well in comparison to the simple blocks-out-of-place baseline. This can be seen in Figure 1 and in Table 1. The solution is also reasonably scalable, generally finding solutions within 10,000 search steps for problems in the 3-stack-8-block to 5-stack-11-block range. Beyond this, performance is largly dependent on a favorable starting configuration. Some performance metrics can be found in Figure 2 and Table 2.

4 Discussion

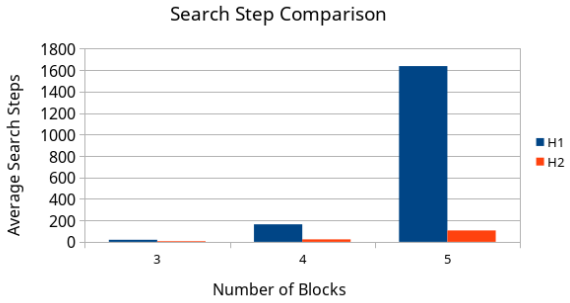
The heuristic worked as expected in that it drastically improved search time over an uninformed approach. A consequence of my implementation is that paths with smaller (B=2, C=3, ...) blocks on top are strongly favored. As a result of this and the fact that my heuristic is not admissible, performance breaks down with a small number of large stacks and non-optimal solutions can be produced. However, good results are still produced within the belt of 3 stacks and 8 blocks, 5 stacks and 11 blocks, and 10 stacks with 15 blocks. With these ranges, the queue can grow to a substantial size. The size of the queue is related tothe

Heuristic Comparison with 3 Stacks							
Number of Blocks	H	Depth	Steps	QL	Error	Samples	Failures
3	H1	3.91	18.995	57.334	0.263	1000	0
	H2	3.855	6.042	21.202	0.011	1000	0
4	H1	5.91	162.833	541.627	0.347	1000	0
	H2	5.934	22.787	84.101	0.022	1000	0
5	H1	7.811	1636.75	5903.22	0.38	289	4
	H2	8.143	105.792	410.92	0.029	1000	0

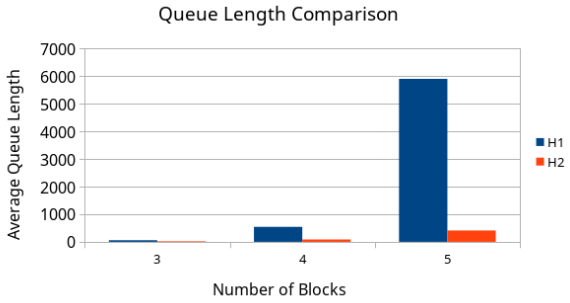
Table 1: Comparison of a range of parameters. H1 indicates the out-of-place heuristic and H2 is outlined above

efficiency of the search, obviously, but specifically the number of stacks and the number of search steps. The branching factor increases by a factor of around the number of stacks and search iterations multiplies this. When the number of stacks is increased, as in Figure 2 the heuristic performs better in terms of search time. This is due to the fact that more available stacks allow blocks to be rearranged without creating conflicts for another block.

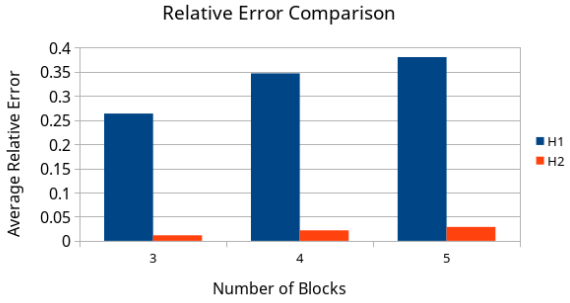
One possible heuristic that I considered was one which considered the relationships between blocks and when what at first appears to be a conflict actually lies along the optimal solution path. That direction seemed to be an endless expansion of corner cases and the relatively simple heuristic which I settled on generally outperformed a more complicated approaches. My heuristic could, however, be stregnthened by 'improving' admissibility. This would involve accounting for the case outlined above when the blocks are arranged in nearly reverse order on non-goal stacks. Since my heuristic punishes large stacks, immediately succesful moves may be overlooked in the interest of reducing a stack. This is highly troublesome on problems with high block to stack ratios. As can be seen in Table 2, the relative error becomes negative for large numbers of blocks indicating a consistent overestimate of cost. Interestingly, underestimating heuristics which I worked on were outperformed by the inadmissible heuristic. I believe that to effectively improve performance, I might need to restructure my formulation of the problem space, adding relational attributes for blocks rather than treating them as numbers in arrays.



(a)

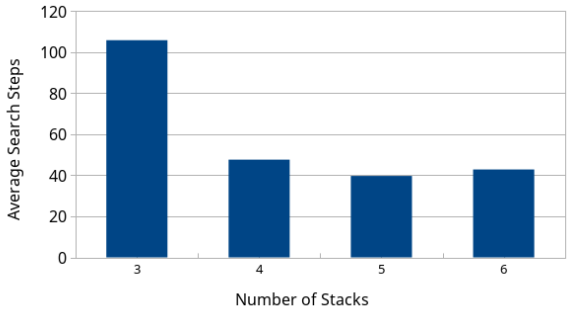


(b)

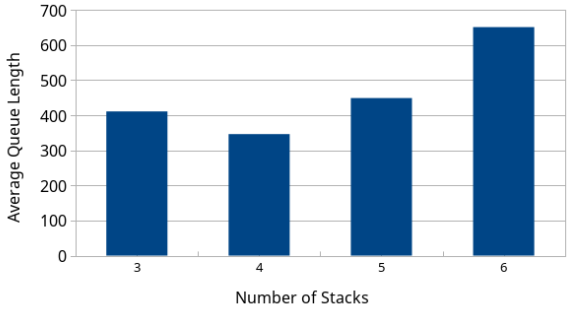


(c)

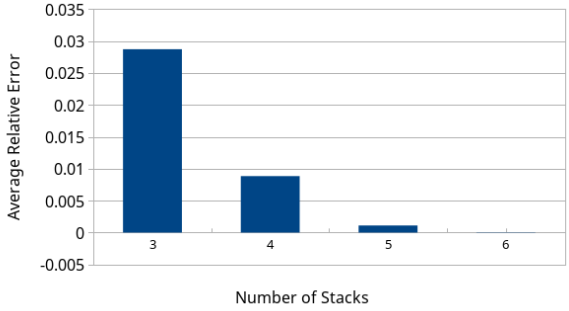
Figure 1: A comparison of the performance of two heuristics for three stacks: H1 is blocks-out-of-place, H2 is that outlined above



(a) Number of search steps



(b) Queue length



(c) Relative error

Figure 2: Performance of heuristic with five blocks and varying stacks

Performance Metrics							
# Stacks	# Blocks	Depth	Steps	QL	Error	Samples	Failures
3	3	3.855	6.042	21.202	0.011	1000	0
3	4	5.934	22.787	84.101	0.022	1000	0
3	5	8.143	105.792	410.92	0.029	1000	0
4	5	6.926	47.614	346.335	0.008	1000	0
5	5	6.405	39.675	449.017	0.001	1000	0
5	10	15.185	689.407	10863.44	-0.012	30	4
6	5	6.199	42.819	651.251	-4.76E-05	1000	0
10	10	12.552	132.862	7910.103	-0.034	60	2
10	15	9.965	158.827	11239.24	-0.023	30	1

Table 2: Comparison of a range of performance metrics for hueristic