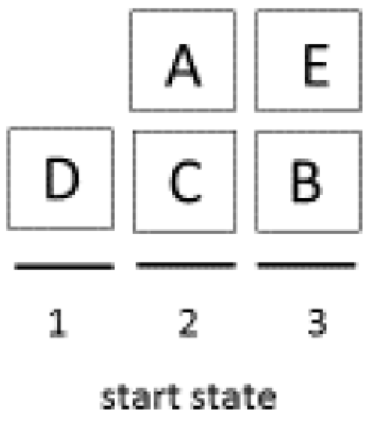
Heuristic Computation

A heuristic is any approach to problem solving, learning, or discovery that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals.

For the Blocksworld Project –

We need a heuristic that estimates the distance to the goal (number of moves to solve the problem) and improves the efficiency of the search - something more sophisticated than just "number of blocks out of place".

Initial Heuristic: The most obvious Heuristic is the "Number of Blocks Out of place". Although, this is the most obvious or fundamental heuristic, it gives the minimum number of moves that definitely needs to be performed to get to the goal state. So, this heuristic only provides the minimum number of moves or the shortest possible distance to the goal node. This means that the goal node cannot be reached in less that the number of moves provided by this heuristic.



For the above initial state, Heuristic 1 would provide a Heuristic value of 5 – because all 5 blocks are out of place.

Improved Heuristic: Stack resemblance -

The next step that can be taken is to look at the goal state to be reached. This provides a way in order to estimate how "far" we really are from the goal state. The final goal here is to get all blocks ordered on Stack 0. This means that we have the information of the number of blocks out of place + the 'closeness' to goal state with respect to stack 0.

However, we see that this does not give any extra information about the initial state provided above with 3 stacks and 5 blocks.

This heuristic also provides a value of 5 for the above initial state.

Final Heuristic: Compute complexity to reach goal state: Overall block movement 'friendliness' heuristic

Next, we can combine the order resemblence of the stack 0 (goal state) information along with the number of blocks 'obstructing the next move'. This can be obtained by finding the next required block and find out how far we are from moving this block. And this can be calculated iteratively for each block.

We can find out the Stack 0 sorted order – resemblance to goal. We can also infer the complement of this – that is – we can get the blocks that need to be removed from Stack 0.

For example, in the above initial state, the number of blocks obstructing the first required block - 'A' is 1 block. So, thus, we are 2 moves away from moving the required block because, it takes 1 move to move the 'D' block out of the way and 1 move to move the 'A' block to the correct position. Thus, this heuristic method, provides a good estimate for each block to reach its goal state which can be thought of as a partial goal state.

So, the parameters that can be computed that may help compute the heuristic are 'number of sorted blocks' in Stack 0, which provides a resemblance to the goal state with respect to stack 0; The next parameter is a complement to the first one, in the sense that blocks which are out of order in Stack 0 should be removed. So, this parameter provides additional information about the resemblance to goal state with respect to Stack 0;

Another parameter that can be computed is the number of empty stacks. The number of empty stacks actually provides a buffer space for transferring blocks in order properly. In other words, this number increases in cases where there are relatively more number of stacks compared to blocks. However, if we have an extreme case where the initial state is the exact opposite order compared to the goal or when the initial state has a configuration of the highest complexity or farthest possible initial state from the goal (in terms of search tree – any other initial state could be a possible child from this root).

Now, we know the stack 0 order and the number of blocks out of order in stack 0. Thus, we can compute the next-in-sequence block that is required. However, to move that block we need to determine if there is any other blocks above this block in any other stack which need to be cleared so that this next required block can be moved into its goal position. This parameter can be computed by finding which block is the next required block; in which stack this block is situated and how many other blocks are above this block. Thus, num_blocks_above_next captures this parameter value.

Now, using these 4 parameters, we can see that the best heuristic can be computed by combining these parameters as a weighted linear combination fashion. Clearly, the more sorted stack 0 is, the closer to the goal we are. However, the unsorted part of stack 0 hinders us from the goal two times as much as the number of unsorted blocks, as we need to clear these blocks and bring them back again at a later state. Similarly, the blocks above the next required block also hinders us twice as much. So, num_blocks_above_next and stack0_unsorted_length parameters can be weighted two times. The number of empty stacks provides a buffer space to switch around blocks. However, any block that is moved into the empty stack has to be moved back to stack 0 in order to reach the goal state. Thus, one move per 'block moved to empty stack' is clearly implied. This means that we can weight num_empty_stacks one single time to imply that atleat one move is required. Now, we know that if the stack 0 is fully sorted and perfectly matches the goal then the heuristic has to be 0, because we have already reached the goal. Thus, if the heuristic has to give a 0, then we need to weight the stack0_sorted_length = -(2+2+1) = -5.

Thus, we have the heuristic to be equal to 2*num_blocks_above_next + 2*stack0_unsorted_length + num_empty_stacks – 5*stack0_sorted_length. Thus, this gives a much better estimate of the goal distance compared to the previous heuristics.

Also, we can see that this heuristic contains a linear combination of 4 parameters each of which is weighted according to the minimum number of moves required when these parameters are valid as explained above. So, clearly, the heuristic is admissible.

Performance Metrics

Given below is computed performance metrics taken from execution statistics. The execution time is computed during execution; However, the statistic is ignored as it is highly dependent on the hardware and software used.

Blocks	Stacks	No of Iterations	Max Queue Size	Path Length
5	3	9	20	8
5	3	9	20	7
5	3	8	19	7
5	3	17	33	12
5	3	7	14	6
5	3	11	24	10
5	3	11	26	10
5	3	8	19	7
5	3	9	18	8
5	3	7	16	6
Mean Path Length = 8.1				
6	3	10	23	9
6	3	12	29	11
6	3	11	26	10
6	3	16	37	13
6	3	14	31	12
6	3	9	22	8

6	3	10	25	9
6	3	13	26	11
6	3	12	27	11
6	3	13	28	12
Mean Path Length = 10.6				
7	3	16	39	15
7	3	17	44	12
7	3	17	40	15
7	3	21	56	14
7	3	39	78	22
7	3	19	46	15
7	3	17	42	13
7	3	19	50	14
7	3	14	33	13
7	3	64	152	19
Mean Path Length = 15.2				
7	4	11	68	10
7	4	13	74	11
7	4	18	105	15
7	4	12	72	11
7	4	13	75	11
7	4	15	94	12
7	4	12	63	11
7	4	14	85	13
7	4	17	111	14
7	4	11	68	9
Mean Path Length = 11.7				
10	5	18	197	17
10	5	30	329	20
10	5	20	218	17
10	5	28	350	22
10	5	17	186	15
10	5	22	251	16
10	5	19	198	16
10	5	16	179	15
10	5	17	190	16
10	5	18	197	15
Mean Path Length = 16.9				
7	5	13	122	10
7	5	12	116	19
7	5	13	146	10
7	5	8	69	7
7	5	14	159	11

7	5	9	92	8
7	5	11	112	10
7	5	12	111	11
7	5	13	146	12
7	5	11	114	9
Mean Path Length = 10.7				
10	7	15	328	14
10	7	15	386	14
10	7	22	567	16
10	7	15	357	13
10	7	13	282	12
10	7	19	515	14
10	7	12	288	11
10	7	17	419	15
10	7	18	438	13
10	7	20	507	13
Mean Path Length = 13.5				

Discussion

The Heuristic seems to be quite good as expected. The results as tabulated above is quite intriguing. It is interesting in that there are multiple possible children states that need to be explored from every given state. The sheer number of children states that need to be computed during certain execution scenarios was beyond my expectation.

The queue length seems to increase exponentially with the average number of iterations given a particular (#stack, #blocks) tuple.

The heuristic is admissible. However, I feel it is not as close as the best possible heuristic. This particular heuristic seems to be better than average perhaps. Clearly, the mean path length information states that it increases when there is a low number of stacks with a relatively high number of blocks. This clearly increases the path length & makes cases more complicated to reach the goal state regardless of the heuristic.

The queue grows exponentially in the number of iterations especially in case of higher number of blocks. Clearly, the number of possible children nodes increase with both stacks and blocks. However, the worst case scenarios are when the number of stacks are low and the number of blocks are high and when stack 0 has absolutely no sorted blocks.

Increasing the number of stacks made the problem easier as proved by the tabulated data above.

The heuristic generally found optimal solutions. Heuristic can be improved to some extent in terms of finding the next two or more levels of blocks in sequence and how many moves need to be performed for them to be moved into the correct locations. This implies to define a more accurate heuristic where multiple levels of blocks in sequence can be determined and see how “far” they really are from their goal states.

The heuristic a* algorithm also generates children nodes and does backtrack if there is a better solution, which I find to be very good.

The largest problem that could be solved was with 24 blocks and 7 stacks. The execution statistics were:

Iteration no.= 65

Queue = 2034

Depth = 46