Abhilash Vallamkonda
126004453

CSCE 625 : Programming Assignment #1

The "Blocksworld" puzzle is very intuitive and not very difficult for humans. This problem is representative of a large set of such problems and solving this seems like a good step towards building AI programs which solve problems which are too difficult for humans to solve. The goal of this project is to design a A* search algorithm to solve the “Blocksworld” puzzle. The main component of the A* search algorithm is the heuristic function which estimates the cost of the cheapest path from any given node to the goal node. My heuristic function captures my own intuition in the computation of the heuristic value.

My “Human” Approach to solving the Blocksworld puzzle

Look at the first stack and check what is needed to get the first block, “a” to its final position. This may involve moving other blocks out of the first stack and also removing the blocks on top of “a” if there are any. While doing these, I try to avoid placing these blocks onto immediately needed blocks like “b” or “c”. Once “a” is in its final position, I use the same approach for “b” and then “c” and so on till I reach the goal state.

Pseudocode for Heuristic function

1) if state is the goal state
   return 0

2) If the state is not the goal state, start checking if the blocks are in their final position starting from “a”. The value of the heuristic only depends on the blocks which are out of place. At the end of this step, the first block which is out of place is found.

3) The first component of the heuristic depends on the first block which is out of place. So, let us refer to this block as heurBlk. The value of the heuristic is the number of steps needed to get heurBlk into its final position. If heurBlk is not on the first stack, then all the blocks on top of it must be moved. All the out of place blocks on the first stack must be moved irrespective of whether or not heurBlk is on the first stack. The heuristic value is at least this value and then +1 to actually move the heurBlk into its final position.

4) The second component of the heuristic estimates the cost of getting the other out of place blocks. This is to avoid placing the blocks moved to in order to get to heurBlk onto other immediately needed blocks. This component is computed as follows:
   In every stack other than the first one, all blocks > than the smallest block will have to be moved to other stacks to get the smallest one and this reverses their order, so considering only these blocks is sufficient. Note that is a good estimate since for every stack, we are always interested only in the smallest block.

5) The sum of the two components is the heuristic value for the current state.

Admissibility

The first component of the heuristic(Step 3) involves moving the blocks on top of heurBlk and the out of place blocks on the first stack. These are necessary steps and the computed value is not an over-estimation.

The second component(Step 4) counts the number of blocks to be moved in order to retrieve the minimum block needed from each of the stacks. This is definitely not an over-estimation because only the cost of retrieving the minimum block is being considered and the actual cost would involve getting all the blocks into place which would certainly be higher.

So, the first component of the heuristic approximates the actual value while the second component is an under-estimation. So, the heuristic value which is the sum of the two components can be safely assumed to never over-estimate the cost to reach the goal. Hence, the heuristic function is admissible.

Pseudocode for A* search

Every iteration of the algorithm involves finding the node with the least estimated cost to the goal node (heuristic value + pathcost) and adding all its children to the frontier. There is a possibility that a path of lower cost to a node already in the frontier is uncovered. In this case, the priority of the node must be updated. However, the priority queue does not provide a change_priority method since this involves an O(n) search to find the node!

Instead, simply add the node to the heap. This results in multiple copies of the same node in the priority queue and we need to figure out a way to avoid processing the node again. If we look closely, there is a neat trick to solve this! First, we observe that the better value for the node will have higher priority and is guaranteed to be explored earlier if it exists. So, all that is needed is a simple check to see if the node popped out of the heap is already in the explored set and this will suffice to discard the older copies.

Incremental Design of the Algorithm

My approach was to mimic my own way of solving the puzzle into the algorithm. So, my initial design for the heuristic function counted the number of steps needed to get lowest out of place block(heurBlk) into its final position. But this approach has a very basic limitation :

If ‘a’ is the heurBlk, then at the instant ‘a’ is placed onto stack 1, the heuristic value is actually 0 which is reserved for the goal state. My function returned a value of 1 which resulted in the following problem: Both the states below return a value of 1 even though the first one is definitely better.

a	
c	c
b e d	b e d a

My solution to the above problem was to initialize h(heuristic value) to #Blocks and decrement it whenever the heurBlk is placed in its final position. This way the first state has h = 4(out of place blocks) while the second has h = 5(out of place blocks) + 1(to get ‘a’ into position) = 6.

The above problem is resolved but the algorithm is far from perfect. Consider the following state:

a		a		
c	children:	c d	c a	c
b e d		b e	b e d	b e d a

The first child has h = 2(to get ‘b’ into place) + 4(blocks which are out of place) = 6 while the second child has h = 1(to get ‘a’) + 5 = 6 and the third also has 1(to get ‘a’) + 5 = 6. All three nodes have the same heuristic value even though the first child is the optimal choice. Because all of them have the same priority, any one of the three can be popped!
The problem here is the discontinuity of computing the heuristic value. In this case, the heurBlk is changing from ‘a’ to ‘b’ after ‘a’ is in its final position.
My initial solution was to introduce some global state to know how far the search has progressed. The idea was to let the heuristic know that, for eg, ‘a’, ‘b’ and ‘c’ are already in place so that the child states which don’t have ‘c’ in place should be ignored. This turned out to be a really bad idea because of a subtle problem : The nodes already in priority queue already have their priority values which must be recomputed whenever heurBlk changes. Doing this really messed up the code and ended with me reverting my changes …
My next idea was to incentivize getting the heurBlk into position. So, I tried initializing h with 4*#Blocks instead of #Blocks as before. Then, when heurBlk is placed in its final position, I do h = h - 4 instead. This solves the problem but there is still scope for improvement!

Consider the following state:

a d	children:	a	a
c b		c b d	c b
e		e	e d

The second child is better since ‘b’ is immediately needed after placing ‘a’ into its final state. However, the heuristic function only considers the heurBlk and hence both the child nodes are assigned the same priority value.
My first solution was to choose the alternative in which the blocks are sorted “better”. For the first child, ‘d’ > ‘c’ and ‘d’ > ‘b’ while in the second child ‘b’ < ‘c’. So, the blocks in the second child are sorted “better” than the first in the sense that the ‘b’ block which is needed earlier is on top. To quantify this idea, I counted the number of blocks in the stack needed earlier than the one on top. The reason for considering the block at the top is that this is the only block which we have control over. This results in an improvement but there are some problems, for eg, consider the following state :

		d	
a d		a	b
b	children:	b	b
c e		c e	c e d

The first child is better but both states will have the same heuristic value since ‘c’ < ‘e’ in the first one and ‘c’ < ‘d’ in the second one. The problem is that only the block at the top of the stack is considered by my heuristic function.
My solution to this was to consider all the out of place blocks. However, this resulted in the second component of the heuristic dominating over the first component which is more important for making progress.
My final solution was to find the smallest element in the stack and count the number of blocks on top of it. The reasoning is that these blocks would have to be removed sooner or later in order to get to the smallest block which is needed earlier than any other blocks in the stack. In the example above, the number of blocks on top of ‘c’ is 1 for the first child and 2 for the second child solving the problem.

This final modification worked better than I expected and the algorithm does everything I can do and it does it much faster!

Performance
The performance of the program is measured for a number of configurations taking 10 random start states for each combination of #Blocks and #Stacks. The only choices for #Blocks are taken as 10 and 20 since for lesser values, the algorithm performs exceedingly well and the results are not very interesting. The summary of the results is presented below (please refer the Appendix for the full results). Note that the median of the results is presented rather than the average, this is because there are certain corner cases which result in my heuristic performing really well or abysmally. These manifest as extreme values of #Goal Tests or Max Queue Size disproportionately affecting the average value. Using the median ignores these extrema to provide a more holistic view of the performance. These corner cases are considered separately.
The random start state is generated by choosing a stack at random and then choosing a block at random to place onto it. I chose this approach because I believe this method produces a more “random” start state than the suggested approach of starting with the goal state and making a sequence of random moves.
The time limit is set as 4s and exceeding this results in the attempt being classified as a failure.
Using the heuristic function described, the algorithm is observed to produce “optimal” solutions in the sense that the steps taken in the solution path by the algorithm is a close approximation to the steps I would have chosen had I been solving the problem myself.

# Blocks	# Stacks	Median # Goal Tests	Median Max Queue Size	Median Length of Solution Path	# Failures
10	3	30	138	26	0
	5	18	294.5	17	0
	7	12.5	351	12.5	0
	10	13	596	13	0
	15	12	962.5	12	0
20	3	1405	6136	80	3
	5	67.5	1216	44.5	0
	7	42	1489	33	0
	10	28	1837	28	1
	15	26	3026.5	26	0

The Median length of the solution path represents the optimal solution size since the algorithm finds the “optimal” solution as described earlier. Hence this value is indicative of the difficulty of the problem for the given configuration.
The difficulty of the problem increases with increasing number of blocks as expected. However, from the table above it is seen that the difficulty actually reduces as the number of stacks increases. This seemingly counterintuitive observation is focus of the discussion which follows: Since we start at a random initial state, the blocks are likely to be spread across all the stacks. With a large number of stacks, there are fewer blocks in each stack and fewer blocks would have to be moved in order to retrieve a specific block. The larger number of stacks also means that the heuristic has more options to choose from while deciding where to place the moved blocks. This is where the second component in my heuristic function really shines!

The median number of goal tests closely follows the median solution length for most of the configurations. This is because of the heuristic function that allows the search algorithm to make good guesses regarding which state is more likely to lead to the goal state.
The maximum queue size approximately follows from the number of goal tests since the algorithm checks if the current state is the goal state and pushes its children into the heap if it isn’t. The number of possible child states is mostly decided by the number of stacks in the configuration. So, designing a good heuristic which reduces the number of goal tests eases the memory requirements in addition to making the algorithm go faster.

The number of goal tests mostly approximates the solution length except for the configuration with 20 blocks + 3 stacks where 1405 goal tests are needed for a solution path length of 80. This is also where 3 out of the 4 failures occur. This seemingly anomalous behaviour can be explained by tracing through what my algorithm actually does for this configuration.

Consider the following configuration :

| a +
| b
| c x x x x x x
where x and + represent some random blocks larger than ‘a’,’b’ and ’c’.

Possible child states are :

| a | a +
| b |
| c x x x x x x + | c x x x x x x b

The first one is the optimal choice since ‘b’ can be placed in its final position with the next move. But this also increases the second component of the heuristic by 1.
In case of the second child, placing ‘b’ onto the 3rd stack reduces the second component of the heuristic by 6 since the minimum block in the stack is now ‘b’ and not ‘c’ which is buried underneath. This results in a huge reduction in the heuristic while making no progress whatsoever!
Even though my heuristic incentivizes getting ‘b’ into place by prioritizing the first component, with 20 blocks and 3 stacks there is high likelihood of such a situation and also there may be a lot of blocks under which the needed block is buried resulting in a huge decrease in the heuristic value. This effect is more pronounced in this case because when the number of stacks is 3, the the second component of the heuristic doesn’t really matter. In the above example, placing + onto the 3rd stack is the only real choice even though it increases the heuristic value.
On the other hand, giving too much importance to the first component will result in the second component of the heuristic being ignored altogether. But from the table, it is also seen that the problem is fairly isolated for the case of 20 blocks and 3 stacks and the algorithm performs well for all other configurations.

Future Work:
The problem described above stems from the somewhat competing components of the heuristic. In my current algorithm, it is possible that one of the components dominates over the other depending on the configuration of the blocks and the number of stacks. One possible solution would be to prioritize the two components in the heuristic based on the given configuration. Right now, the weights of the 2 components are independent of the configuration and this works reasonably well in most scenarios.
Another possible solution would be to decide the second component of the heuristic value only based on the block which is changed from the parent state but this involves a risk of being a short-sighted approach.

Appendix :

- **Block configurations and performance metrics for 10 random start states**

10 blocks and 3 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
34	155	24
27	122	26
30	141	29
27	123	26
24	106	22
81	370	32
24	111	24
32	138	30
36	164	34
30	138	26

10 blocks and 5 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
30	457	18
18	295	17
15	222	15
18	295	18
16	221	15
29	471	19
16	261	16
19	294	17
13	200	13
20	309	17

10 blocks and 7 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
13	342	13
15	400	15
14	377	14
13	360	13
12	265	12
13	396	13
12	307	12
12	313	12
12	337	12
12	379	12

10 blocks and 10 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
15	715	15
12	637	12
14	698	14
13	537	13
13	609	13
12	583	12
12	565	12
13	564	13
14	626	14
12	484	12

10 blocks and 15 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
12	969	12
12	913	12
14	1051	14
12	815	12
11	970	11
10	845	10
13	996	13
11	858	11
14	1009	14
11	956	11

20 blocks and 3 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
3198	12867	98
1050	4555	83
3603	14142	76
Failed		
104	469	80
3574	14639	3574
1405	6136	74
117	512	72
Failed		
Failed		

20 blocks and 5 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
775	12841	50
54	943	44
52	897	47
66	1197	39
1867	28735	45
82	1434	41
65	1136	42
69	1235	43
52	912	47
127	2121	53

20 blocks and 7 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
31	1032	31
29	1028	28
53	1946	33
28	1005	28
131	4919	44
63	2353	38
27	940	27
60	2289	33
34	1095	34
50	1883	43

20 blocks and 10 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
25	1641	25
28	1890	28
29	1763	29
31	1923	31
28	1539	28
27	1837	27
34	2460	32
31	2067	31
Failed		
28	1602	28

20 blocks and 15 stacks

# Goal Tests	Max Queue Size	Length of Solution Path
28	3403	28
28	2997	28
24	2931	24
25	3252	25
26	3251	26
26	2985	26
28	3263	28
25	3056	25
24	2833	24
27	2816	27