Dept. Computer Science & Engineering

CSCE-625

Blocks World

Submitted **To:**
Dr.Thomas R.loerger
Computer Science &
Engineering Department

Submitted **By :**
Chen Chen
625008676

Read Me

There are three java file in my project, within the package com.blockworld.

blockWorld.java	Including the main() function. As well as A* search algorithm.
Node.java	Describing the structure of each state. Including F-value, parent node, state matrix and depth.
Utils.java	Including other methods such like output the path, generate child state and so on.

All of the project is created based on Intellij IDEA, so we can use IDE to run the project.
Besides, we can also run my code in cmd or terminal.

Compile and run:

1.cd to the folder of source code(ProjectName\src\com\blockworld),using command **javac *.java.**

2.Back to the src folder, then **java com.blockworld.blockWorld.**

3.The cmd will notify you to type in the number of blocks and stacks. Here, I generated the original state by arranging blocks in different stacks randomly.

4. if there is any trouble running my code, please contact with me.

```
PS G:\625BlocksWorld\src> java com.blockworld.blockWorld
Please input the number of blocks
5
Please input the number of stacks
3
Initial State is:
1 C, A
2
3 D, E, B
```

Example:

```
PS G:\625BlocksWorld\src> java com.blockworld.blockWorld
Please input the number of blocks
5
Please input the number of stacks
3
Initial State is:
1 E, D
2 B, A
3 C
```

```
iter=0, queue=0, f=g+h=21, depth=0
iter=1, queue=5, f=g+h=15, depth=1
iter=2, queue=9, f=g+h=10, depth=2
iter=3, queue=11, f=g+h=10, depth=3
iter=4, queue=15, f=g+h=10, depth=4
iter=5, queue=17, f=g+h=9, depth=5
iter=6, queue=21, f=g+h=10, depth=4
iter=7, queue=24, f=g+h=10, depth=6
iter=8, queue=28, f=g+h=10, depth=6
iter=9, queue=31, f=g+h=10, depth=5
iter=10, queue=35, f=g+h=11, depth=7
iter=11, queue=36, f=g+h=11, depth=7
iter=12, queue=39, f=g+h=11, depth=7
iter=13, queue=40, f=g+h=10, depth=8
iter=14, queue=42, f=g+h=9, depth=9
Success!Depth=9,total_goal_tests=15,max_queue_size=42
1|E,D
2|B,A
3|C

1|E
2|B,A
3|C,D

1|
2|B,A
3|C,D,E

1|A
2|B
3|C,D,E
```

```
1|A,B
2|
3|C,D,E

1|A,B
2|E
3|C,D

1|A,B
2|E,D
3|C

1|A,B,C
2|E,D
3|

1|A,B,C,D
2|E
3|

1|A,B,C,D,E
2|
3|
```

1. Algorithm and Data Structure

In the project, I designed different classes to solve the block world.

Node Class: Representing states in BlockWorld.

```
class Node{
    public List<List<Integer>> state=new ArrayList<>();//Denote current structure of blocks and stacks
    Node parent;//father state of present node, we can use this element to traceback
    int depth;//g value
    int f;//f=g+h

    //Constructor
    public Node(List<List<Integer>> state,Node parent,int depth)

    @Override
    public boolean equals(Object obj)

    @Override
    public String toString()

    @Override
    public int compareTo(Node o)
}
```

Utils Class: Other supportive Methods.

```
class utils{
    //Generate Original State,type in # blocks and # stacks, construct state randomly
    static List<List<Integer>> generateOriginal(int blocks,int stacks);

    //Generate Final State, all of the stacks should keep order in the first stack
    static List<List<Integer>> generateFinal(int blocks,int stacks);

    //output the state in String
    static String toStr(List<List<Integer>> state);

    //Both copy() and getNeighbor() help to generate child states
    static List<List<Integer>> copy(List<List<Integer>> root);
    static List<Node> getNeighbor(Node root);

    //Heuristic function
    static int computeH(List<List<Integer>> state);

    //Record and output the max-depth,iterations,queue size as well as f-value
    static void record(int iter,int size,int f,int depth);

}
```

blockWorld Class: Main function and A* search().

2. Heuristic Function

I've tried two kinds of heuristic functions, first one is the number of blocks out of place, which is a simple method to estimate the distance from current state to destination. However, the efficiency of the method is going to decrease when it is applied on more complicate cases.
Besides the method above, I try to calculate the the distance from current position to the final state for each block, and add the distances together. This summation would be the H value. Plus with the G value (depth for each node, # movement from original state to current one), we can obtain the F value for each node.
Basically, there are three kinds of blocks we need to consider separately.

In stack, in position:
First of all, for the blocks in the first stack, we can skip the blocks which has already in the final state. But for those in the first stack with wrong position, they may stand above or below the target position.

In stack, not in position:
For instance, let's suppose the i^{th} block should be in the ***first stack and position i, state[0][i-1].*** The current position of i^{th} block is ***'index' (state[0][index])***. If index is larger than i ***(index>i)***, which means the ***block is above the destination***, we have to clear all of the blocks(inclusive) above i and push the i^{th} block back to the its final position. Here, **h=index-i+1+1**.(the last '+1' means we have to push the block back to the target).
On the other hand, if the ***block stands below the final position***, which means ***index<i,***we have to remove all of the blocks above the index(inclusive), represented by FirstStack.size()-index. Then, push (i-index) blocks back except the i^{th} one. Finally, put the i^{th} block in the destination.
h=FirstStack.size()+i-2*index+1.

Others:
Moreover, for blocks stays in wrong position and wrong stack, we have to remove the blocks both above this block and its final position.
h=stack.size()-index+FirstStack.size()-i+1.

Admissible?
For an admissible heuristic function, the F value should keep increasing during the iteration. However, we can observe the output in the experiment and notice that the F value is floating with an increasing trend. Also, from the conclusion in the programming result, it shows that there are some redundant moves to the final state. Therefore, this heuristic function is not admissible.

3. Experiment (Each case tests 10 times)

#Blocks	#Stacks	Max depth	Path length	#iteration	Max Queue Size	Successful rate
5	3	11.1	12.1	22.8	43.5	10/10
6	3	12.6	13.6	42.8	61.8	10/10
8	3	22.6	23.2	277.6	321	10/10
10	3	27	28	519.1	658.7	10/10
5	4	6.9	7.9	16.3	74.1	10/10
8	4	15.1	16.1	44.2	243.3	10/10
10	4	20.9	21.9	70.1	400.3	10/10
6	5	8.1	9.1	45.9	377	10/10
8	5	14.4	15.4	98.7	825.7	10/10
10	5	19.1	20.1	193	1580.2	10/10
8	6	13	14	111	1318.4	10/10
10	6	20.7	21.7	77.6	433.3	10/10
12	6	25.2	26.2	333.1	3922.2	10/10
8	7	12	13	130.8	1878.8	10/10
10	7	16.9	17.9	187.1	2517.4	10/10
12	7	23.3	24	185.6	2605.4	9/10
14	7	32.5	33.5	622	9797.9	8/10
16	8	42	43	827.5	12174.8	4/10

4. Discussion

From the experiment data above, we can observe that if the number of blocks is 5 and stacks is 3, the result would be generated within a few iterations. As we increase the number of stacks, the #iterations would also become greater. Our heuristic function works well under these simple cases.

Moreover, I increase the number of blocks and adjust the number of stacks gradually. As we can see, if the #stacks is greater than the #blocks, we could obtain the result quickly. Besides, not only the efficiency, but the successful rate is decreasing when the problem become more difficult. Here, I set the limited running time within 5 minutes, which means upon the program runs more than 5 minutes, the case would be classified as failure. We can conclude that when the #stacks is close to the #blocks, the random distribution of these blocks tends to be more 'average', which means we may be able to solve it easier. However, when the #stacks grows up, the #iteration and Queue size would become much larger than previous one, as well as lower successful rate and longer solving path.

So far, the most complicated case my program could solve is 16 blocks in 8 stacks. And if we keep enlarge the #blocks and #stacks, there is still possibility that we can handle these more hard cases with a simple and regular original state.

In my heuristic function, I just focus on the distance from current state to destination for each blocks, and neglect the relationships between each one and the others. Since as we just calculate the #moves to its target, without considering the impact of other blocks' movement, which may be the main reason results for the overestimating. Furthermore, in a specific case, the complexity of each block is different, we should arrange different weight to them depends on their status.