World of Blocks States Planner

Ryan Nichols

Wright State University

December 18, 2017

Abstract

This report discusses a predicate calculus based World of Blocks solving program, which given an initial state and a goal state description, produces a list of intermediate steps to be performed. This solver was implemented using a modified version A*. It was initially implemented using A* without any modifications, which is far more computationally expensive (usually infeasible for more than 6 blocks), but was capable of finding the optimal (shortest) path from the initial state to the goal state. The modifications made to the solving algorithm removed the capability of finding the shortest path, but reduced the time complexity greatly, thus allowing complex problems involving 10 blocks to be solved within a reasonable amount of time (typically less than 7 seconds).

Table of Contents

Solving Methodology

This World of Blocks solver consists of four primary components: Predicates, States, Actions, and a modified implementation of the A* search algorithm. Before describing the solving algorithm, it is important to understand the structure of the Predicates, States, and Actions.

Predicates objects consist of a name (clear, clearloc, on, ontable, and holding) and associated block/location objects depending on the predicate type.

State objects contain lists of predicates, along with some additional information, including the parent State from which it was derived (if applicable), the action performed on the parent state to obtain the current state (if applicable), the estimated cheapest cost to reach the goal state from the current state, and the cost to reach the current state from the initial state.

Action objects define an action that is to be performed upon a list of Predicate objects. They consist of a name (pickup, putdown, unstack, and stack) and associated block/location objects depending on the action type. Applying an action object to a world (list of predicates) entails adding and removing predicates to and from the list. For example, applying the action Pickup(A) to the world {OnTable(A, L1), Clear(A)} would yield a new world {ClearLoc(L1), Holding(A)}.

When executed, the program asks the user to define the initial and goal states by inputting the name of blocks from bottom to top, one stack at a time. This input is converted into two lists of predicate objects, one for the initial world description and one for the goal world description. These two lists are then passed into a solver (ResolutionEngine) object, on which the solve() method is called to obtain a lists of State objects that describe a path from the initial to the goal.

The solve() method maintains a PriorityQueue of State objects, which defines the order in which the states will be evaluated, and a list of State objects that has already been evaluated (to prevent redundant evaluations). The ordering of the former is where the modifications to the A* algorithm comes in.

In a traditional A* implementation, the states would be evaluated in order of their value F(n), which is the sum of the cost to reach that state from the initial state (G(n)) and the estimated cost to reach the goal state from the current state (H(n)). Such an implementation will always return a path from the initial state to the goal state with the minimum number of intermediate steps. However, for many problems, evaluating states in this order results in an exponentially large search space, thus rendering the algorithm infeasible in terms of runtime. Given the requirement that this solver must be capable of finding solutions for worlds containing ten blocks, this proved to be an insufficient solution. The modification to resolve this issue came in the form of changing the order in which the PriorityQueue was sorted. Rather than sorting states by their F(n) value, they were instead sorted by their H(n) value. This change means that the algorithm no

longer attempts to find the overall shortest path (accounting for the cost incurred from reaching a given state from the initial state), but is instead always evaluating the states that are closest to the goal state first.

The solve() method begins by creating a State object with the list of initial Predicates. The parent and Action of this state is set to null, as is appropriate for an initial state. The H(n) value is calculated by calling another method contained in the solver object called calcDistanceToGoal(), the heuristic of the algorithm. It calculates the minimum costs of reaching the goal state from the current state by simply counting the number of predicates in the current state that are not contained in the goal state. While this heuristic cannot be used to predict which of a list of states is the fewest number of steps away from the goal state with perfect accuracy, it does a reasonably good job and results a greatly reduced search space compared to brute forcing.

This initial State is then added to the PriorityQueue and searching begins. The search takes place inside of a while loop that exits either when the PriorityQueue is empty (i.e. all possible solutions have been searched and none were able to generate the goal state, which should only happen if an incompatible initial and goal state are entered – e.g. a different number of blocks) or when a Boolean value indicating that the solution was found has been set to true. The first step of the search loop is to poll the PriorityQueue for the highest priority state to evaluate, which for the first pass can only be the initial state as no other states have been added yet. In all other cases, as mentioned above, it will be the state with the lowest H(n) value, as determined by the heuristic method. Next a list of possible actions that can be legally performed on the state being evaluated is obtained by calling the getPossibleActions() method on it. This method generates a list of Action objects by analyzing the list of predicates to determine what blocks/locations are clear to place a block on, if a block is being held, or what blocks are clear to be picked up, if nothing is being held. Finally, a for each loop, which iterates through each of the possible actions is nested within the while loop. The first step within the nested for each loop there is to apply the current action to current state to obtain a modified list of predicates. Next the modified list is checked against all entries contained in the method's list of evaluated states. If the predicate list is determined to have already been evaluated, a continue statement is used to skip to the next action in the for each loop. Otherwise, a new State object is created with the modified list, with the action set to the current action in the loop, the parent set to the original state before modification, and H(n) set to the value returned by calling calcDistanceToGoal() on the modified predicate list. This state is then checked against the goal state. If it is found to match the goal, the Boolean flag to exit the while loop is set to true, the state is saved to a variable outside of the loop, and a break statement is used to exit the for each loop. If it does not match the goal, the state is added to the PriorityQueue so its potential children can be checked at a later point. Once the while loop is exited, the solution state is passed to the buildStateList() method to generate a list of states from the initial state to the final state by recursively checking the parent of the states until null is reached. The states list is reversed to obtain the proper order and is finally returned to the main method, where it is printed in its entirety.

Results

After modifying the algorithm to sort the PriorityQueue by H(n) rather than F(n) the solver was able to consistently solves complex problems with 10 blocks within less than 7 seconds. Following is an example of inputs and outputs along with recorded runtime to demonstrate the algorithms efficiency. Note: outputted tables have been reformatted for readability's sake in this document and only the first and last two states are shown for conciseness.

Example 1)

Define the initial state.
Enter a block to stack on L1 (or clear): a
Enter a block to stack on A (or clear): b
Enter a block to stack on B (or clear): c
Enter a block to stack on C (or clear): clear
Enter a block to stack on L2 (or clear): j
Enter a block to stack on J (or clear): i
Enter a block to stack on I (or clear): h
Enter a block to stack on H (or clear): clear
Enter a block to stack on L3 (or clear): f
Enter a block to stack on F (or clear): e
Enter a block to stack on E (or clear): d
Enter a block to stack on D (or clear): clear
Enter a block to stack on L4 (or clear): g
Enter a block to stack on G (or clear): clear
Define the goal state.
Enter a block to stack on L1 (or clear): g
Enter a block to stack on G (or clear): b
Enter a block to stack on B (or clear): c
Enter a block to stack on C (or clear): clear
Enter a block to stack on L2 (or clear): d
Enter a block to stack on D (or clear): a
Enter a block to stack on A (or clear): e
Enter a block to stack on E (or clear): j
Enter a block to stack on J (or clear): i
Enter a block to stack on I (or clear): f
Enter a block to stack on F (or clear): clear
Enter a block to stack on L3 (or clear): h
Enter a block to stack on H (or clear): clear
Enter a block to stack on L4 (or clear): clear

Init State:
ONTABLE(A, L1)
ON(B, A)
ON(C, B)

CLEAR(C)
ONTABLE(J, L2)
ON(I, J)
ON(H, I)
CLEAR(H)
ONTABLE(F, L3)
ON(E, F)
ON(D, E)
CLEAR(D)
ONTABLE(G, L4)
CLEAR(G)

Goal State:
ONTABLE(G, L1)
ON(B, G)
ON(C, B)
CLEAR(C)
ONTABLE(D, L2)
ON(A, D)
ON(E, A)
ON(J, E)
ON(I, J)
ON(F, I)
CLEAR(F)
ONTABLE(H, L3)
CLEAR(H)
CLEARLOC(L4)

Solution found!

State 0:

C	H	D	
B	I	E	
A	J	F	G
L1	L2	L3	L4

State 1:

		G	
C	H	D	
B	I	E	
A	J	F	
L1	L2	L3	L4

.
.
.
.

State 69:

	I		
	J		
C	E		
B	A	F	
C	D	H	
L1	L2	L3	L4

State 70:

	F		
	I		
	J		
C	E		
B	A		
C	D	H	
L1	L2	L3	L4

Solution found in 6.753 seconds.

Conclusion

By initially implementing this project with an unmodified version of the A* algorithm it quickly became apparent that the search space of a 10 block World of Blocks problem is far too large to successfully traverse with A*. The simple modification of sorting the PriorityQueue by costs to goal (H(n)) rather than total costs (F(n)) quickly cut down the search space to a feasible, solvable level, without being forced to design a complex heuristic method for the algorithm.

Though it can be considered a mistake to have decided to attempt using A* for a problem with such a large search space, I can say that doing so was a helpful learning experience that enlightened me to the tradeoffs of finding optimal solutions versus finding solutions quickly. I also realized the importance of having a strong heuristic function to ensure the feasibility of A* for larger problems, which would have been a difficult feat to achieve in a problem like this. Ultimately, it was a good mistake to have made, as these are both learning outcomes that I would have missed out on had I taken a speed based approach from the start.

References

[1] En.wikipedia.org. (2017). A* search algorithm. [online] Available at: https://en.wikipedia.org/wiki/A*_search_algorithm [Accessed 2 Dec. 2017].

[2] "Java Platform SE 7", Docs.oracle.com, 2017. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/. [Accessed: 2- Dec- 2017].

[3] M. Genesereth, Logical Foundations of Artificial Intelligence. Palo Alto, CA: Morgan Kaufmann, 1987.

Appendix

AI_Final.java

```
/*
* Program for Final Project of CS4850
* This program acts a states based solver for the World of Blocks problem.
* It uses a modified version of A* that finds a solution as quickly as possible
* at the expense of no longer being the optimal solution. This is need for most
* problems larger than 6 blocks. If desired, the algorithm can be changed back
* to traditional A* by commenting State.java: line 109 and uncommenting line 110
*/
package ai_final;

import java.util.ArrayList;
import java.util.Scanner;
import java.util.Stack;

/**
 *
 * @author Ryan Nichols
 */
public class AI_Final {
    //Global vars
    static Block A = new Block("A");
    static Block B = new Block("B");
    static Block C = new Block("C");
    static Block D = new Block("D");
    static Block E = new Block("E");
    static Block F = new Block("F");
    static Block G = new Block("G");
    static Block H = new Block("H");
    static Block I = new Block("I");
    static Block J = new Block("J");

    static Location L1 = new Location("L1");
    static Location L2 = new Location("L2");
    static Location L3 = new Location("L3");
    static Location L4 = new Location("L4");

    /**
     * Main Method
     * @param args
     */
    public static void main(String[] args) {
        ArrayList<Predicate> initialWorld;
        ArrayList<Predicate> goalWorld;
```

```
ArrayList<State> states = new ArrayList<>();

//Get initial state
System.out.println("Define the initial state.");
initialWorld = defineWorld();

//Get goal state
System.out.println("Define the goal state.");
goalWorld = defineWorld();

//Display init world desciption in console
System.out.println("Init State:");
for (Predicate p : initialWorld) {
   System.out.println(p.toString());
}
System.out.println("");

//Display goal world desciption in console
System.out.println("Goal State:");
for (Predicate p : goalWorld) {
   System.out.println(p.toString());
}
System.out.println("");

//Solve
ResolutionEngine res = new ResolutionEngine(initialWorld, goalWorld);
long start = System.currentTimeMillis();
states = res.solve();
long end = System.currentTimeMillis();
long total = end - start;   //Calculate runtime


//Display states in console
ArrayList<Stack<String>> stringStacks = new ArrayList<>();             //Description of a
world
ArrayList<ArrayList<Stack<String>>> allStateStrings = new ArrayList<>();//List of world
descriptions

for (int i = 0; i < states.size(); i++) {
   stringStacks = states.get(i).toStringStacks();      //Convert each state in the path to a
printable form
   allStateStrings.add(stringStacks);                //Add it to an iterable list
}

for (int i = 0; i < allStateStrings.size(); i++) {      //For each state
   System.out.println("State " + i + ": ");           //Display the state number
```

```
            System.out.println("_____");
            for (int k = 0; k < 10; k++) {                    //For each row
                System.out.print("|   ");
                for (int j = 0; j < allStateStrings.get(i).size(); j++) {   //For each stack
                    String temp = allStateStrings.get(i).get(j).pop();      //Print the block name or empty
space
                    System.out.print(temp + "   "); //Spacing
                }
                System.out.println("|");
            }
            System.out.println("    L1  L2  L3  L4\n\n");
        }
        System.out.println("Solution found in " + total/1000.0 + " seconds.");
    }//end main

    /**
     * Get user input to define the world
     * @return A list of predicates describing the world defined by the user
     */
    public static ArrayList<Predicate> defineWorld() {
        ArrayList<Predicate> predicates = new ArrayList<>();
        predicates.addAll(defineStack(L1));
        predicates.addAll(defineStack(L2));
        predicates.addAll(defineStack(L3));
        predicates.addAll(defineStack(L4));
        return predicates;
    }

    /**
     * Get user input to define one stack
     * @param loc Location/Block on which the stack is to be placed
     * @return A list of predicates describing the stack defined by the user
     */
    public static ArrayList<Predicate> defineStack(Object loc) {
        ArrayList<Predicate> predicates = new ArrayList<>();
        Scanner s = new Scanner(System.in);
        String ans = "";
        Object top = loc;

        while (!ans.equals("CLEAR")) {
            System.out.print("Enter a block to stack on " + top.toString() + " (or clear): ");
            ans = s.next().toUpperCase();
            switch(ans) {
                case "A":
                    if (top instanceof Location) {
                        predicates.add(new Predicate(Predicate.ONTABLE, A, (Location) top));
```

```
            top = A;
        }
        else if (top instanceof Block) {
            predicates.add(new Predicate(Predicate.ON, A, (Block) top));
            top = A;
        }
        break;
    case "B":
        if (top instanceof Location) {
            predicates.add(new Predicate(Predicate.ONTABLE, B, (Location) top));
            top = B;
        }
        else if (top instanceof Block) {
            predicates.add(new Predicate(Predicate.ON, B, (Block) top));
            top = B;
        }
        break;
    case "C":
        if (top instanceof Location) {
            predicates.add(new Predicate(Predicate.ONTABLE, C, (Location) top));
            top = C;
        }
        else if (top instanceof Block) {
            predicates.add(new Predicate(Predicate.ON, C, (Block) top));
            top = C;
        }
        break;
    case "D":
        if (top instanceof Location) {
            predicates.add(new Predicate(Predicate.ONTABLE, D, (Location) top));
            top = D;
        }
        else if (top instanceof Block) {
            predicates.add(new Predicate(Predicate.ON, D, (Block) top));
            top = D;
        }
        break;
    case "E":
        if (top instanceof Location) {
            predicates.add(new Predicate(Predicate.ONTABLE, E, (Location) top));
            top = E;
        }
        else if (top instanceof Block) {
            predicates.add(new Predicate(Predicate.ON, E, (Block) top));
            top = E;
        }
```

```
    break;
case "F":
    if (top instanceof Location) {
        predicates.add(new Predicate(Predicate.ONTABLE, F, (Location) top));
        top = F;
    }
    else if (top instanceof Block) {
        predicates.add(new Predicate(Predicate.ON, F, (Block) top));
        top = F;
    }
    break;
case "G":
    if (top instanceof Location) {
        predicates.add(new Predicate(Predicate.ONTABLE, G, (Location) top));
        top = G;
    }
    else if (top instanceof Block) {
        predicates.add(new Predicate(Predicate.ON, G, (Block) top));
        top = G;
    }
    break;
case "H":
    if (top instanceof Location) {
        predicates.add(new Predicate(Predicate.ONTABLE, H, (Location) top));
        top = H;
    }
    else if (top instanceof Block) {
        predicates.add(new Predicate(Predicate.ON, H, (Block) top));
        top = H;
    }
    break;
case "I":
    if (top instanceof Location) {
        predicates.add(new Predicate(Predicate.ONTABLE, I, (Location) top));
        top = I;
    }
    else if (top instanceof Block) {
        predicates.add(new Predicate(Predicate.ON, I, (Block) top));
        top = I;
    }
    break;
case "J":
    if (top instanceof Location) {
        predicates.add(new Predicate(Predicate.ONTABLE, J, (Location) top));
        top = J;
    }
```

```
                else if (top instanceof Block) {
                    predicates.add(new Predicate(Predicate.ON, J, (Block) top));
                    top = J;
                }
                break;
            default:
                ans = "CLEAR";   //Break loop
                if (top instanceof Location) {
                    predicates.add(new Predicate(Predicate.CLEARLOC, (Location) top));
                }
                else if (top instanceof Block) {
                    predicates.add(new Predicate(Predicate.CLEAR, (Block) top));
                }
                break;
        } //End Switch
    }//End while
    return predicates;
  }

}//end class
```

ResolutionEngine.java

```
package ai_final;

import java.util.ArrayList;
import java.util.Collections;
import java.util.PriorityQueue;

/**
 * This class contains methods to find a path (list of State objects) from an
 * initial world description (list of predicates) to a goal world description.
 * To solve with this class. Call the two argument constructor and provide
 * an initial world and goal world description (two ArrayList of Predicate
 * objects). Then call the solve() method on this object to get the path.
 * @author Ryan Nichols
 */
public class ResolutionEngine {
   ArrayList<Predicate> initialWorld;
   ArrayList<Predicate> goalWorld;

   /*******************************************************
    *                Constructors                 *
    *******************************************************/

   public ResolutionEngine() {
   }

   public ResolutionEngine(ArrayList<Predicate> initialWorld, ArrayList<Predicate>
goalWorld) {
      this.initialWorld = initialWorld;
      this.goalWorld = goalWorld;
   }


   /*******************************************************
    *            Getters and Setters              *
    *******************************************************/
   public ArrayList<Predicate> getInitialWorld() {
      return initialWorld;
   }

   public void setInitialWorld(ArrayList<Predicate> initialWorld) {
      this.initialWorld = initialWorld;
   }

   public ArrayList<Predicate> getGoalWorld() {
      return goalWorld;
```

```
    }

    public void setGoalWorld(ArrayList<Predicate> goalWorld) {
        this.goalWorld = goalWorld;
    }


    /*****************************************************
     *                Core Methods                       *
     *****************************************************/
    /**
     * This method uses a modified version of the  A* algorithm to find list of
     * states that describe a path from an initial world description to a goal
     * world description.
     * Upon realizing that the traditional A* algorithm is infeasible for large
     * problems, such as a complex 10 block world of blocks problem, this
     * algorithm was modified slightly by adjusting the sorting method of the
     * priority queue.
     *
     * @return An ArrayList of States which define a path from the initial world
     * to the goal world
     */
    public ArrayList<State> solve() {
        PriorityQueue<State> unevaluatedStates = new PriorityQueue<>();    //List of intermediate
states, sorted by the State method calcFn()
        ArrayList<State> evaluatedStates = new ArrayList<>();
        ArrayList<State> stateList;
        boolean solutionFound = false;  //Loop breaker
        State finalState = null;


        //Create a state from the initial world description
        int distanceToGoal = calcDistanceToGoal(initialWorld, goalWorld);
        State initState = new State(initialWorld,   //State using the initial world conditions
                                    null,          //No action has been performed as this is the initial state
                                    null,          //No parent exists for the
                                    0,             //The distance from the initial state to itself is 0
                                    distanceToGoal);//The distance from the inital state to the goal as
calculated by the heuristic function, calcDistanceToGoal

        //Add the initial state to the list of
        unevaluatedStates.add(initState);

        //Iterate through the priority queue until it is empty or the solution has been found
        while(unevaluatedStates.size() > 0 && solutionFound == false) {
            //Obtain and remove the current best state in the queue
```

```
        State current = unevaluatedStates.poll();
        //System.out.println("Evaluating state, distance: " +
calcDistanceToGoal(current.getPredicate(), goalWorld));

        //Get a list of possible actions to be performed on this state
        ArrayList<Action> possibleActions = current.getPossibleActions();

        //Add to completed list
        evaluatedStates.add(current);

        //Iterate through possible actions
        for (Action a : possibleActions) {
           //Run an action on the current world to obtain a modified world
           ArrayList<Predicate> modifiedWorld = a.applyAction(current.getPredicate());

           boolean checked = false;

           //Check if new world is equivalent to that of an already evaluated state
           for(State s : evaluatedStates) {
              if (areWorldsEqual(s.getPredicate(), modifiedWorld)){   //If match is found
                 checked = true;                          //Mark world as already checked
                 break;                                   //Stop searching
              }
           }
           if (checked)
              continue;                                   //Continue to next possible world

           //Create a State from this modified world
           State childState = new State(modifiedWorld,                    //The new predicate
set
                                        a,                                //The action performed to obtain the
state
                                        current,                          //The parent of the new state
                                        current.getGn() + 1,              //Gn of the new state is 1 more
than that of its parent
                                        calcDistanceToGoal(modifiedWorld, goalWorld)); //The calculated
distance from the new state to the goal

           //Check if the modified world is the same as the goal world
           if(areWorldsEqual(modifiedWorld, goalWorld)) {  //If it matches the goal
              solutionFound = true;                        //Mark solution as found to break out of while
loop
              finalState = childState;                     //Save the state
              System.out.println("Solution found!\n");
              break;                                       //Stop searching
           }
```

```
            else { //Otherwise
                //Add the new state to the list of unevaluated states
                unevaluatedStates.add(childState);
                //System.out.println("Not goal and not evaluated, adding to queue: " +
calcDistanceToGoal(modifiedWorld, goalWorld)); //Debug
            }
        }
    }

    //Build the list of states from the initial state to the final state
    stateList = buildStateList(finalState);
    return stateList;
}

/**
 * Heuristic function for solving algorithm. Returns the number of
 * predicates in the given world description that is not in the target world
 * description.
 * Lower is better.
 * @param current
 * @param goal
 * @return
 */
public int calcDistanceToGoal(ArrayList<Predicate> current, ArrayList<Predicate> goal) {
    int predCount = 0; //Number of predicates in current that are not in goal
    for (Predicate currP : current) {
        if (!goal.contains(currP))
            predCount++;
    }
    return predCount;
}

/**
 * Compares two lists of Predicates to determine if they describe the same
 * world. This is necessary as the predicates may not be listed in the same
 * order.
 * @param world First world
 * @param world2 Second world
 * @return true if the worlds are equal; false otherwise
 */
public boolean areWorldsEqual(ArrayList<Predicate> world, ArrayList<Predicate> world2) {
    boolean equal = world2.containsAll(world); //All predicate in world are also in the goal
world
    equal = equal && world.containsAll(world2);//All predicate in the goal world are also in
the world
```

```
        return equal;
    }

    /**
     * Builds a list of State objects from the start State (i.e. the ancestor
     * state that has no parent) to the provided State
     * @param state The state to which the list should be built
     * @return an ArrayList of State objects from the start State to the
     * provided state.
     */
    public ArrayList<State> buildStateList(State state) {
        ArrayList<State> stateList = new ArrayList<>();
        State current = state;
        if(state == null) {
            return stateList;
        }
        while(current.getParent() != null) {    //While current has a parent
            stateList.add(current);             //Add the current state
            current = current.getParent();      //Set current to its parent
        }
        stateList.add(current);

        //Reverse the state list
        Collections.reverse(stateList);

        ArrayList<State> removals = new ArrayList<>();
        for (State s : stateList) {
            for (Predicate p : s.getPredicate()) {
                if(p.getName().equals(Predicate.HOLDING)) { //If it's a holding clause
                    removals.add(s);                        //Mark for removal
                }
            }
        }
        stateList.removeAll(removals);

        return stateList;
    }
}
```

State.java

```
package ai_final;

import java.util.ArrayList;
import java.util.Stack;

/**
 * This class defines a state object, which consists of:
 * 1) a list of predicates that describing a world
 * 2) a parent state, from which this state was derived
 * 3) the action that was applied to the parent state to obtain this state
 * 4) the cost of reaching this state from the initial state (used for A*)
 * 5) the estimated cheapest cost of reaching the goal state from this state
 * @author Ryan Nichols
 */
public class State implements Comparable {
    ArrayList<Predicate> predicate;
    Action action;
    State parent;
    int gn; //Cost from start to here
    int hn; //Estimated cheapest cost from here to goal


    /**
     * Default constructor
     */
    public State() {
        this.predicate = null;
        this.action = null;
        this.parent = null;
    }

    /**
     * Complete constructor
     * @param predicate this list of predicates that define the world of this
     * state
     * @param action the action applied to obtain this state from its parent
     * @param parent this states parent
     * @param gn distance from initial state to this state
     * @param hn estimated distance from this state to the goal state
     */
    public State(ArrayList<Predicate> predicate, Action action, State parent, int gn, int hn) {
        this.predicate = predicate;
        this.action = action;
        this.parent = parent;
        this.gn = gn;
```

```
    this.hn = hn;
}


/*****************************************************
*              Getters and Setters                  *
*****************************************************/
public ArrayList<Predicate> getPredicate() {
    return predicate;
}

public void setPredicate(ArrayList<Predicate> predicate) {
    this.predicate = predicate;
}

public Action getAction() {
    return action;
}

public void setAction(Action action) {
    this.action = action;
}

public State getParent() {
    return parent;
}

public void setParent(State parent) {
    this.parent = parent;
}

public int getGn() {
    return gn;
}

public void setGn(int gn) {
    this.gn = gn;
}

public int getHn() {
    return hn;
}

public void setHn(int hn) {
    this.hn = hn;
}
```

```
    /*******************************************************
    *                Utility Methods                *
    *******************************************************/
    /**
     * This method was originally used in the A* implementation to find the
     * total estimated length of the path from the start state -> this state ->
     * goal state. By changing this method to return hn instead, it returns the
     * length of the path from just this state -> goal state. This means that
     * the solver is no longer technically an A* implementation, and does not
     * find the optimal path, however it does find an answer many orders of
     * magnitude faster, which is absolutely necessary for non trivial problems
     * with more than 7 blocks.
     */
    public int calcFn() {
        return hn;          //Find an answer as quickly as possible, technically no longer A*
        //return gn + hn;   //Find the answer with the shortest possible path (original A* method)
    }

    /**
     * This method generates and returns a list of Action objects that can be
     * validly applied to this state.
     * @return ArrayList of Action objects that can be validly applied to this
     * state.
     */
    public ArrayList<Action> getPossibleActions() {
        ArrayList<Action> actions = new ArrayList<>();
        Block holding = null;

        //Check if there is a block being held
        for (Predicate p: predicate) { //Scan all predicates
            if(p.getName().equals(Predicate.HOLDING)) {
                holding = p.getArgument1();
            }
        }

        //Pickup
        if (holding == null) {                      //Can only pick up if not already holding a block
            for (Predicate p : predicate) {             //Scan all predicates
                if(p.getName().equals(Predicate.ONTABLE)) { //If there's an ONTABLE predicate
                    Block blockOnTable = p.getArgument1();  //Get the block on the table
                    for(Predicate p2 : predicate) {         //Scan through predicates again
                        if(p2.getName().equals(Predicate.CLEAR) && p2.getArgument1() != null &&
p2.getArgument1().equals(blockOnTable)) { //If the block on the table is found to be clear
```

```
                    actions.add(new Action(Action.PICKUP, blockOnTable, p.getLocation()));
//Add pickup block as a possible action
                    break;
                }
            }
        }
    }
    //Putdown
    if (holding != null) {                    //Can only put down if already holding a block
        for (Predicate p : predicate) {            //Scan all predicates
            if(p.getName().equals(Predicate.CLEARLOC)) {//If there's a clear location
                if (p.getLocation() != null) {        //If the location is not null, then it is a clear
location
                    actions.add(new Action(Action.PUTDOWN, holding, p.getLocation()));   //Add
putdown block on location as a possible action
                }
            }
        }
    }
    //Unstack
    if (holding == null) {                    //Can only unstack if not already holding a block
        for (Predicate p : predicate) {            //Scan all predicates
            if(p.getName().equals(Predicate.ON)) {      //If there's an ON predicate
                Block topBlock = p.getArgument1();      //Get the block on top
                Block bottomBlock = p.getArgument2();   //Get the block on bottom
                for(Predicate p2 : predicate) {        //Scan through predicates again
                    if(p2.getName().equals(Predicate.CLEAR) && p2.getArgument1() != null &&
p2.getArgument1().equals(topBlock)) {    //If the top block is found to be clear
                        actions.add(new Action(Action.UNSTACK, topBlock, bottomBlock));
//Add unstack top block as a possible action
                        break;
                    }
                }
            }
        }
    }
    //Stack
    if (holding != null) {                    //Can only stack if already holding a block
        for (Predicate p : predicate) {            //Scan all predicates
            if(p.getName().equals(Predicate.CLEAR)) {   //If there's a clear location
                if (p.getArgument1() != null) {        //If the argument1 is not null, then it is a clear
block
                    actions.add(new Action(Action.STACK, holding, p.getArgument1()));   //Add
putdown held block on clear block as a possible action
                }
```

```
            }
        }
    }

    return actions;
}



/**
 * Simple print method that shows all predicates
 */
public void print() {
    for(Predicate p : predicate) {
        p.print();
        System.out.print(" ");
    }
    System.out.println("");
}

/**
 * Converts the predicate calculus statements of this state to a list of
 * four Stacks of Strings (one each for L1, L2, L3, and L4).
 * @return an ArrayList of four Stacks of Strings
 */
public ArrayList<Stack<String>> toStringStacks() {
    ArrayList<Stack<String>> stacks = new ArrayList<Stack<String>>();
    Stack<String> L1 = new Stack<>();
    Stack<String> L2 = new Stack<>();
    Stack<String> L3 = new Stack<>();
    Stack<String> L4 = new Stack<>();

    ArrayList<Predicate> copy = new ArrayList<>(predicate);
    ArrayList<Predicate> toRemove = new ArrayList<>();

    //L1 first pass
    for (Predicate p : copy) {
        if (p.getLocation() != null) {
            if (p.getLocation().getName().equals("L1")) {
                if(p.getName().equals("CLEARLOC")){           //L1 is clear
                    for(int i = 0; i < 10; i++){
                        L1.push("_");                    //Fill stack with 10 empty strings
                    }
                }
                else if(p.getName().equals("ONTABLE")) {    //A block is on L1
                    L1.push(p.getArgument1().getName());    //Push the name of the block
```

```
            }
            toRemove.add(p);
        }
    }
}
copy.removeAll(toRemove);
toRemove.clear();

//L2 first pass
for (Predicate p : copy) {
    if (p.getLocation() != null) {
        if (p.getLocation().getName().equals("L2")) {
            if(p.getName().equals("CLEARLOC")){           //L2 is clear
                for(int i = 0; i < 10; i++){
                    L2.push("_");                   //Fill stack with 10 empty strings
                }
            }
            else if(p.getName().equals("ONTABLE")) {    //A block is on L2
                L2.push(p.getArgument1().getName());    //Push the name of the block
            }
            toRemove.add(p);
        }
    }
}
copy.removeAll(toRemove);
toRemove.clear();

//L3 first pass
for (Predicate p : copy) {
    if (p.getLocation() != null) {
        if (p.getLocation().getName().equals("L3")) {
            if(p.getName().equals("CLEARLOC")){           //L3 is clear
                for(int i = 0; i < 10; i++){
                    L3.push("_");                   //Fill stack with 10 empty strings
                }
            }
            else if(p.getName().equals("ONTABLE")) {    //A block is on L3
                L3.push(p.getArgument1().getName());    //Push the name of the block
            }
            toRemove.add(p);
        }
    }
}
copy.removeAll(toRemove);
toRemove.clear();
```

```
        //L4 first pass
        for (Predicate p : copy) {
            if (p.getLocation() != null) {
                if (p.getLocation().getName().equals("L4")) {
                    if(p.getName().equals("CLEARLOC")){           //L4 is clear
                        for(int i = 0; i < 10; i++){
                            L4.push("_");                   //Fill stack with 10 empty strings
                        }
                    }
                    else if(p.getName().equals("ONTABLE")) {    //A block is on L4
                        L4.push(p.getArgument1().getName());    //Push the name of the block
                    }
                    toRemove.add(p);
                }
            }
        }
        copy.removeAll(toRemove);
        toRemove.clear();

        L1 = toStringStackHelper(L1,copy);
        L2 = toStringStackHelper(L2,copy);
        L3 = toStringStackHelper(L3,copy);
        L4 = toStringStackHelper(L4,copy);

        stacks.add(L1);
        stacks.add(L2);
        stacks.add(L3);
        stacks.add(L4);

        return stacks;
    }

    /**
     * Helper method for toStringStacks. Should not be called from anywhere else
     * @param strStack The stack that is to be completed
     * @param preds The list of predicates
     * @return The complete string stack representation
     */
    private Stack<String> toStringStackHelper(Stack<String> strStack, ArrayList<Predicate>
preds) {
        String top = strStack.peek();
        if(top.equals("")){
            return strStack;
        }
        else {
            for (Predicate p : preds) {
```

```
            if(p.getArgument1().getName().equals(top) &&
p.getName().equals(Predicate.CLEAR)){  //Top is clear
                for(int i = strStack.size(); i<10; i++) {
                    strStack.push("_");                                    //Fill the rest of the way with
emptry strings
                }
            }
            if(p.getArgument2() != null && p.getArgument2().getName().equals(top) &&
p.getName().equals(Predicate.ON)) {
                //Arg 1 is on top
                top = p.getArgument1().getName();
                strStack.push(top);
            }
        }
    }
    return strStack;
  }

  /**
   * Compare this State to another State object
   * @param o The State object to which this State will be compared
   * @return Standard 1, 0, -1 depending on comparison result.
   */
  @Override
  public int compareTo(Object o) {
    State other = (State)o;
    int result = Integer.compare(calcFn(), other.calcFn());
    return result;
  }
}
```

Action.java

```
package ai_final;

import java.util.ArrayList;

/**
 * This class defines a single action that can be performed upon a world (list
 * of predicates). Possible actions include:
 * pickup a block from a location,
 * putdown a block onto a location,
 * unstack a block from a block
 * stack a block onto a block
 *
 * Call constructors for each of these actions as follows:
 * pickup: new Action(Action.PICKUP, BLOCK, LOCATION);
 * putdown: new Action(Action.PUTDOWN, BLOCK, LOCATION);
 * unstack: new Action(Action.UNSTACK, BLOCK, BLOCK);
 * stack: new Action(Action.STACK, BLOCK, BLOCK);
 *
 * @author Ryan Nichols
 */
public class Action {
    //Fields
    String name;
    Location location;  //For certain actions like pickup(A, L1) or putdown(A, L1)
    Block block1;
    Block block2;       //For certain actions like unstack(A, B) or stack(A, B)

    //String constants
    final public static String PICKUP = "PICKUP";
    final public static String PUTDOWN = "PUTDOWN";
    final public static String UNSTACK = "UNSTACK";
    final public static String STACK = "STACK";
    final public static String NOOP = "NOOP";

    /**
     * Constructor for pickup/putdown actions
     * @param name
     * @param location
     * @param block1
     */
    public Action(String name, Block block1, Location location) {
        this.name = name;
        this.location = location;
        this.block1 = block1;
        this.block2 = null;
```

```
    }

    /**
     * Constructor stack/unstack actions
     * @param name
     * @param block1 top block
     * @param block2 bottom block
     */
    public Action(String name, Block block1, Block block2) {
        this.name = name;
        this.location = null;
        this.block1 = block1;
        this.block2 = block2;
    }

    /*******************************************************
     *                  Core Methods                       *
     *******************************************************/
    /**
     * Generates a list of predicates that should be added and removed from a
     * world to which this action is applied.
     * @return a Changes object, which contains a list of Predicates to be
     * removed and a list of Predicates to be added
     */
    public Changes getChanges() {
        Changes changes;
        ArrayList<Predicate> removed = new ArrayList<>();
        ArrayList<Predicate> added = new ArrayList<>();

        switch (name) {
            case PICKUP:                                    //Pickup block1 from location
                removed.add(new Predicate(Predicate.ONTABLE, block1, location));    //Remove
ONTABLE(block1, location)
                removed.add(new Predicate(Predicate.CLEAR, block1));                //Remove
CLEAR(block1). This will be added back as soon as it is placed somewhere
                added.add(new Predicate(Predicate.CLEARLOC, location));             //Add
CLEARLOC(location)
                added.add(new Predicate(Predicate.HOLDING, block1));                //Add
HOLDING(block1)
                break;
            case PUTDOWN:                                   //Putdown block1 on location
                added.add(new Predicate(Predicate.ONTABLE, block1, location));      //Add
ONTABLE(block1, location)
                added.add(new Predicate(Predicate.CLEAR, block1));                  //Add
CLEAR(block1)
```

```
            removed.add(new Predicate(Predicate.CLEARLOC, location));          //Remove
CLEARLOC(location)
            removed.add(new Predicate(Predicate.HOLDING, block1));             //Remove
HOLDING(block1)
            break;
         case UNSTACK:                                    //Unstack block1 from block2
            removed.add(new Predicate(Predicate.ON, block1, block2));          //Remove
ON(block1, block2)
            removed.add(new Predicate(Predicate.CLEAR, block1));               //Remove
CLEAR(block1). This will be added back as soon as it is placed somewhere
            added.add(new Predicate(Predicate.CLEAR, block2));                 //Add
CLEAR(block2)
            added.add(new Predicate(Predicate.HOLDING, block1));               //Add
HOLDING(block1)
            break;
         case STACK:                                      //Stack block1 on block2
            added.add(new Predicate(Predicate.ON, block1, block2));            //Add ON(block1,
block2)
            added.add(new Predicate(Predicate.CLEAR, block1));                 //Add
CLEAR(block1). This will be added back as soon as it is placed somewhere
            removed.add(new Predicate(Predicate.CLEAR, block2));               //Remove
CLEAR(block2)
            removed.add(new Predicate(Predicate.HOLDING, block1));             //Remove
HOLDING(block1)
            break;
         case NOOP:
            break;
      }

      changes = new Changes(removed, added);
      return changes;
   }

   /**
    * Modifies and returns the provided world by applying the changes that
    * occur upon performing this action.
    * @param world
    * @return
    */
   public ArrayList<Predicate> applyAction(ArrayList<Predicate> world) {
      ArrayList<Predicate> modifiedWorld = new ArrayList<>(world);    //Copy the given world
      Changes predicateChanges = getChanges();                        //Calculate the changes caused by
this action
      ArrayList<Predicate> removals = new ArrayList<>();              //Predicates marked for
removal
      //Apply changes
```

```
        for (Predicate p : predicateChanges.getRemoved()) {     //For each predicate that is to be
removed
            for(Predicate p2 : modifiedWorld) {              //For each predicate in the world
                if (p.equals(p2))                          //If the predicates match
                    removals.add(p2);                      //Mark the predicate for removal
            }
        }
        modifiedWorld.removeAll(removals);                  //Delete all predicates that are removed
as a result of this action
        modifiedWorld.addAll(predicateChanges.getAdded());      //Add all predicates that are
added as a result of this action

        return modifiedWorld;
    }

    /*********************************************************
     *              Getters and Setters              *
     *********************************************************/
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Location getLocation() {
        return location;
    }

    public void setLocation(Location location) {
        this.location = location;
    }

    public Block getBlock1() {
        return block1;
    }

    public void setBlock1(Block block1) {
        this.block1 = block1;
    }

    public Block getBlock2() {
        return block2;
    }
```

```
    public void setBlock2(Block block2) {
        this.block2 = block2;
    }

    /*****************************************************
    *                Utility Methods                    *
    *****************************************************/
    public String toString() {
        String output = name + "(";
        if(block1 != null) {                    //If there is an block1
            output += block1.getName();
            if(block2 != null) {                //And block2
                output += ", " + block2.getName();  //e.g. ON(A, B)
            }
            if(location != null) {              //And location
                output += ", " + location.getName();//e.g. ONTABLE(A, L1)
            }
        }
        output += ")";
        return output;
    }

    public void print() {
        String output = toString();
        System.out.print(output);
    }

    public boolean equals(Action other) {
        boolean eq = false;
        if (name.equals(other.getName())) {                 //If the predicates are of the same type
            eq = true;                                      //True so far
            if(block1 != null) {
                eq = eq && block1.equals(other.getBlock1());        //Check if block1 matches
            }
            if(block2 != null) {
                eq = eq && block2.equals(other.getBlock2());        //Check if block2 matches
            }
            if(location != null) {
                eq = eq && location.equals(other.getLocation());    //Check if location matches
            }
        }
        return eq;
    }
}
```

Predicate.java

```
package ai_final;

/**
 * This class defines a predicate, which consists of a name and various
 * arguments, depending on the type.
 * @author Ryan Nichols
 */
public class Predicate {
    //Fields
    String name;
    Block argument1;
    Block argument2;
    Location location;

    //String constants
    final public static String ON = "ON";
    final public static String ONTABLE = "ONTABLE";
    final public static String CLEAR = "CLEAR";
    final public static String CLEARLOC = "CLEARLOC";
    final public static String HOLDING = "HOLDING";



    /*******************************************************
     *                Constructors                 *
     *******************************************************/
    public Predicate(String name) {
        this.name = name;
        this.argument1 = null;
        this.argument2 = null;
        this.location = null;
    }

    /**
     * Used for Holding(block), Clear(block)
     * @param name
     * @param argument1
     */
    public Predicate(String name, Block argument1) {
        this.name = name;
        this.argument1 = argument1;
        this.argument2 = null;
        this.location = null;
    }
```

```
/**
 * Only used for Clear(loc)
 * @param name
 * @param location
 */
public Predicate(String name, Location location) {
    this.name = name;
    this.argument1 = null;
    this.argument2 = null;
    this.location = location;
}

/**
 * Used for pickup(block, loc), putdown(block, loc)
 * @param name
 * @param argument1
 * @param location
 */
public Predicate(String name, Block argument1, Location location) {
    this.name = name;
    this.argument1 = argument1;
    this.argument2 = null;
    this.location = location;
}

/**
 * Used for stack(block, block), unstack(block, block)
 * @param name
 * @param argument1
 * @param argument2
 */
public Predicate(String name, Block argument1, Block argument2) {
    this.name = name;
    this.argument1 = argument1;
    this.argument2 = argument2;
    this.location = null;
}


/*******************************************************
 *              Getters and Setters                    *
 *******************************************************/
public String getName() {
    return name;
}
```

```
public void setName(String name) {
   this.name = name;
}

public Block getArgument1() {
   return argument1;
}

public void setArgument1(Block argument1) {
   this.argument1 = argument1;
}

public Block getArgument2() {
   return argument2;
}

public void setArgument2(Block argument2) {
   this.argument2 = argument2;
}

public Location getLocation() {
   return location;
}

public void setLocation(Location location) {
   this.location = location;
}

/*****************************************************
*                Utility Methods                    *
*****************************************************/
/**
 * Returns true if this predicate matches the provided Predicate
 * @param o Other Predicate to be compared
 * @return true if equal, false otherwise
 */
public boolean equals(Object o) {

   if (!(o instanceof Predicate)) {
      return super.equals(o);
   }
   Predicate other = (Predicate) o;

   boolean eq;
   eq = name.equals(other.getName());
   if (eq) {
```

```
            if(argument1 != null && other.getArgument1() != null) {
              eq = eq && argument1.equals(other.getArgument1());   //Check if argument1 matches
            }
            if(argument2 != null && other.getArgument2() != null) {
              eq = eq && argument2.equals(other.getArgument2());   //Check if argument2 matches
            }
            if(location != null && other.getLocation() != null) {
              eq = eq && location.equals(other.getLocation());     //Check if location matches
            }
        }

        return eq;
    }

    public String toString() {
        String output = name + "(";
        if(argument1 != null) {                    //If there is an argument1
            output += argument1.getName();         //e.g clear(A) or the following
            if(argument2 != null) {                //And argument 2
                output += ", " + argument2.getName();//e.g. on(A, B)
            }
            if(location != null) {                 //And location
                output += ", " + location.getName();//e.g. onTable(A, L1)
            }
        }
        else if (location != null) {               //If there is a location but not argument 1
            output += location.getName();          //e.g. clear(L1)
        }
        output += ")";
        return output;
    }

    public void print() {
        String output = toString();
        System.out.print(output);
    }
}
```

Changes.java

```
package ai_final;

import java.util.ArrayList;

/**
 * This class defines a set of changes which can be applied to a world (list of
 * predicates). It consists of a list of predicates to be removed from the world
 * and a list of predicates to be added to the world.
 * @author Ryan Nichols
 */
public class Changes {
   ArrayList<Predicate> removed;
   ArrayList<Predicate> added;

   public Changes() {
   }

   public Changes(ArrayList<Predicate> removed, ArrayList<Predicate> added) {
      this.removed = removed;
      this.added = added;
   }

   public ArrayList<Predicate> getRemoved() {
      return removed;
   }

   public void setRemoved(ArrayList<Predicate> removed) {
      this.removed = removed;
   }

   public ArrayList<Predicate> getAdded() {
      return added;
   }

   public void setAdded(ArrayList<Predicate> added) {
      this.added = added;
   }
}
```

Block.java

```
package ai_final;

/**
 * This class defines a block, which for this problem only has a name
 * @author Ryan Nichols
 */
public class Block {
    String name = "";

    public Block() {
    }

    public Block(String n) {
        this.name = n;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean equals(Block other) {
        return name.equals(other.getName());
    }

    public String toString() {
        return name;
    }
}
```

Location.java

```
package ai_final;

/**
 * This class defines a location, which for this problem only has a name
 * @author Ryan Nichols
 */
public class Location {
    String name = "";

    public Location() {
    }


    public Location(String n) {
        this.name = n;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean equals(Location other) {
        return name.equals(other.getName());
    }

    public String toString() {
        return name;
    }
}
```