Ellis Saupe ems236

Project 1 Write Up

Code Design

To keep the problem abstract, my code is organized into interfaces and abstract classes. Game is an abstract class that represents some type of puzzle, Heuristic is an interface representing a heuristic function h(s), Move is an interface representing a move, and State is an abstract class that stores the game state. Because State, Heuristic, and Move are all abstract, Game is generalizable for any game. To create a new game that can be solved with the search algorithms in Game, someone has to write a class that contains game specific logic for storing the state and testing for the goal state, a class for each heuristic that can interpret that representation of a state, a class for each move, and a class that organizes these objects into collections and has a main method to read text commands.

States:
State contains a static randomizeState method that takes a list of moves and returns a new state with k random moves taken. This method also populates a list<String> with the names of every move it makes. Every other method related to the State of a puzzle is abstract because it is dependent on the type of puzzle. These methods include setGoalState(), isGoalState(), setWithString(), isValidState(), print(), copy() and getHash(). EightState is a class that extends state and represents the state of the eight puzzle. The state is stored as an ArrayList<Integer>, where the integer values of the tiles are stored in order of top left square to bottom right square. In this representation, 0 is the blank tile. The implementation of each of these methods is relatively straight-forward. isValidState() is only used with setWithString(); it does not check if a state is solvable, only that is contains each tile once and only once. It is relatively easy to determine if an 8 puzzle is solvable, so EightState also includes an isSolvable() method and a getAllSolvable() method that returns a list of every solvable state. These will be discussed further in the Experiments section.

Moves:
The Move interface has 3 methods: State move(State), Boolean isValid(State), and String getName(). In the case of the eight puzzle, isValid checks if this move tries to move the blank out of the bounds of the board. If the blank is in space 6, 7, or 8, down is not a valid move. Move() takes a state, makes a move if its valid, and returns the successor state after making that move. getName() returns the name of a move, such as “up”. Each move on a type of puzzle must be its own class that implements this interface. This allows moves to be stored in arrays and be accessed by their name, etc. EightMoveUp, EightMoveDown, EightMoveRight, and EightMoveLeft are the move classes for the eight puzzle.

Heuristics:
The Heuristic interface only has the method int calculate(State) and String getName(). Calculate evaluates the heuristic function with the supplied state. EightHeuristicDistance represents h2 and EightHeuristicMisplaced represents h1. Again, this organization allows heuristic functions to be stored in collection classes and accessed by their name.

Games:
The Game abstract class implements most aspects of a puzzle. It stores the current state of the game in the property State state. Game contains the executeInput(String) method that takes the specified

string, a command from a file or command line input, and executes the corresponding method within Game. SetState(String) and SetState(State) each set the game state to state generated by state.setWithString() or to the provided state. randomizeState(int n, List<moves>) sets the game state to a random state that is n moves from the current state using the State.randomizeState() method. MakeMove(Move) changes the game state to the result of the specified move using the move.move() method if it is a legal move. PrintState calls the state.print() method. MaxNodes(int n) sets the maximum number of nodes to be considered with a* search. The abstract methods of Game involve generating the moves and heuristics associated with a specific game. getMoveByName(), getHeuristicByName(), getAllMoves(), and getDefaultHeuristic() are all abstract methods.

Eight Puzzle:
The class EightPuzzle extends Game. EightPuzzle has a static array of all moves that is returned by getAllMoves(). Getting a move or heuristic by name is implemented by storing all the moves and heuristics of an eight puzzle in a HashMap of Moves or Heuristics respectively with Strings as keys. EightPuzzle also contains the main method, which initializes the puzzle, moves, and heuristics then reads a file and command line input for text commands to execute. The initialization of the collections of Heuristics and Moves is shown below.

```
private static List<Move> moves;
private static Map<String, Move> moveMap;
private static Map<String, Heuristic> heuristicMap;

//Initialization
private static void initializeStaticVars()
{
    initialized = true;

    moves = new ArrayList<Move>();
    //Add moves for 8 puzzle
    moves.add(new EightMoveDown());
    moves.add(new EightMoveRight());
    moves.add(new EightMoveUp());
    moves.add(new EightMoveLeft());

    //Allows moves to be found by name.
    moveMap = new HashMap<String, Move>();
    for(Move m : moves)
    {
        moveMap.put(m.getName(), m);
    }

    heuristicMap = new HashMap<String, Heuristic>();
    heuristicMap.put("h1", new EightHeuristicMisplaced());
    heuristicMap.put("h2", new EightHeuristicDistance());
}
```

This is the majority of the organization required to create a new game that can be solved with the generic search algorithms in Game.

Search Algorithms:
Game has a method solveAStar(Heuristic h, List<Move>, Boolean shouldPrint), which takes a heuristic h, a set of moves available for the game, and finds a path to the goal state using a* search with the provided heuristic. Game also has solveBeam(int k, Boolean shouldPrint), which uses the game's default heuristic to implement a beam search with width k. These methods print the path to the goal if shouldPrint is true, otherwise they return a solution object. Solution is essentially a tuple with the

number of moves to a solution and the number of nodes considered. The number of moves or the depth is set to -1 if no solution is found.

A*:

A* search is implemented with a priority queue of SearchNodes. SearchNode is a class that contains properties for costToHere, the state, the heuristic of the state, and a list of the names of the moves taken to get to this state. SearchNode is comparable, using the sum of the heuristic and the costToHere. This method also uses a HashMap<Integer, Boolean> to store states that have already been explored. The integer key is generated with the State.getHash() method, which for the 8 puzzle is the hashCode of the list structure that stores the state. Every time a node that has been expanded is polled from the heap, it is ignored. This prevents the algorithm from wasting time on repeated nodes; on the second time it is polled from the queue, the path cost of a node must be greater than or equal to its path cost when it was first polled from the queue. The snippet to select the next node to expand is shown below.

```
//Get next unexplored node from heap
SearchNode node = pQueue.poll();
while(explored.containsKey(node.getState().getHash()))
{
    node = pQueue.poll();
    if(node == null)
    {
        System.out.println("Error, queue was empty.");
        return new Solution(-1, nodesConsidered);
    }
}

//add state to list of explored states
explored.put(node.getState().getHash(), true);
```

When expanding a node, the method checks if it is the goal state. If not, it applies every valid move to the node and checks the hash table for the new state. This check may seem redundant, but checking the hash map is less expensive than adding a new node to the heap and throwing it out later. If the new state has not been explored, a new SearchNode is created and added to the heap. The code for expanding a node is shown below.

```
//Make every move from this state
for(Move move : moves)
{
    //check for valid.
    if(move.isValid(node.getState()) )
    {
        State nextState = move.move(node.getState());

        //Check if move already explored
        if(!explored.containsKey(nextState.getHash()))
        {
            List<String> nextMoves = node.getMoves();
            nextMoves.add(move.getName());
            pQueue.add(
                new SearchNode(
                    nextMoves
                    , nextState
                    , heuristic.calculate(nextState)
                    , node.getCostToHere() + 1
                )
            );
        } //end if is unexplored
    } //end if is valid
} //end for
```

Every time a node is expanded, the nodeCount is incremented. The method returns no solution if nodeCount exceeds maxNodes.

Beam:

Beam search requires k random initial states so that there is diversity in the successor states at each iteration. One state can be the initial game state, the remaining k-1 states are generated with 10 random moves from the original state. As a consequence of this, the solutions from this method often have 10 random moves at the beginning of the solution. Beam search re-uses the SearchNode class. All information is the same, but the costToHere is always set to 0 because the path is irrelevant in beam search. These k nodes are added to a list. At every iteration of the search, every node is taken from this list, its state is added to a hash map, all valid successor states are generated, all successor states are checked for the goal state, and every successor node is added to the priority queue.

To select the next k states for the beam, they are polled from the priority queue. If a state polled from the priority queue is already in the hash table, it is placed in a temporary queue. If there are no more unexplored states in the priority queue, nodes are polled from the temporary queue. Once there are k states selected, the priority queue is cleared and the temporary queue is discarded. This behavior is shown below.

```
SearchNode nextNode;
if(!outofNodes)
{
    nextNode = pq.poll();
    if(nextNode == null)
    {
        outofNodes = true;
        nextNode = exploredTemp.poll();
    }
}
else
{
    nextNode = exploredTemp.poll();
    if(nextNode == null)
    {
        if(shouldPrint)
        {
            System.out.println("Out of moves. Failed after " + iterationCount + " iterations.");
        }
        return new Solution(-1, iterationCount * k);
    }
}

if(outofNodes || !explored.containsKey(nextNode.getState().getHash()))
{
    nodes.add(nextNode);
    i++;
}
//node is explored but should store it temporarily in case heap runs out of new nodes.
else
{
    exploredTemp.add(nextNode);
}
```

If the search seems to be stuck at a local maximum, it will return with no solution. When the first node is selected for to test for successor states, the method checks if this state is already in the hash table. Because all explored states are pushed to a temporary queue when selecting the next k nodes, if the first node in the list of k nodes has already been expanded, there were no unexplored states during the last iteration of the search. If this happens with 25 consecutive sets of k nodes, the method assumes the search is stuck in a loop and returns no solution. 25 is a reasonable tolerance because the number of moves required to solve a puzzle is rarely significantly above 20, so 25 consecutive iterations with no new states strongly suggests that the search is stuck in a loop.

Code Correctness

All of my tests are shown in CommandDemo.txt. The first set of commands are simply included to show that all of my commands function properly. Solving 731 5b2 468 with all three search algorithms results in the same 12 move solution. A*(h1) considered nearly 4 times as many nodes as A*(h2), suggesting that h2 is often significantly more efficient. Beam search took 12 iterations of k=10 states to find a solution, which is similar to considering 120 nodes. Setting k to 1, beam search is clearly much less efficient, taking 302 moves to find the goal. After randomizing the state 100 more moves to 731 5b8 426, beam search with k=10 states has 10 extra moves at the beginning of its solution from generating a random initial successor state. This solution is still valid, as making all of the moves shows.
The next examples show where these algorithms fail. A*(h2) can solve every puzzle while considering less than 50000 nodes. Depending on maxNodes, A*(h2) never fails. A*(h1) technically solves every puzzle if maxnodes is close to 9!/2 = 181440, the total number of possible states. Because explored nodes are stored in a hash table, h1 always explores a new node. By brute force, that node is eventually

the goal state. An example of each search exceeding maxNodes is given with extra information being printed. Each step shows that the sum of the heuristic and path cost is being minimized by the priority queue, as is specified by A*. Beam search can solve every puzzle with k >= 2. For lower beam widths, beam search can be caught in an infinite loop as is shown with the state b14 826 753. For k=1 this happens frequently because there is very little diversity in successor states. If the blank is on a corner, there are only two successor states, and one is guaranteed to have been considered already.

Experiments

The problem space is relatively small, so it's not incredibly time consuming to run exhaustive tests of every solvable state. I have a function permutation that generates every permutation of 0,1,2,3,4,5,6,7,8. Additionally, an 8 puzzle if solvable if the number of inversions is even where an inversion is an instance where a tile with a lower number is further to the right in the array than a tile with a higher number. The list of all permutations is processed to remove the configurations with odd parity and then every configuration can be tested.

A. Beam search is not limited by maxNodes, but rather by k and the limit of 25 repeated iterations. Setting k to 2, every puzzle is solvable. With k = 1 only 163353/181440 (90%) states are solvable. A table of maxNodes and # of puzzles solved for h1 and h2 is shown below.
Maxnodes: 50 | h1 solved: 715 | h2 solved: 4351
Maxnodes: 100 | h1 solved: 1617 | h2 solved: 11430
Maxnodes: 500 | h1 solved: 9672 | h2 solved: 70705
Maxnodes: 1000 | h1 solved: 18865 | h2 solved: 113340
Maxnodes: 5000 | h1 solved: 66563 | h2 solved: 177824
Maxnodes: 10000 | h1 solved: 97333 | h2 solved: 181251
Maxnodes: 50000 | h1 solved: 175517 | h2 solved: 181440
Maxnodes: 100000 | h1 solved: 181351 | h2 solved: 181440
Maxnodes: 150000 | h1 solved: 181440 | h2 solved: 181440

B. As the above table shows, h2 is significantly faster than h1 because it traverses much fewer nodes of the state space. H1 searches all solutions in 2571716ms, h2 in 181174ms.

C. A table of moves to solution and number of instances solved with that number of moves is shown below for each algorithm. In this example, k = 10. The number of moves for beam search is arbitrarily much higher than the number of moves for either A* search. This is partly due to the fact that beam search makes 10 random moves at the beginning of every search, and partly due to the fact that beam search is a local search and can get stuck in loops for a few iterations before moving towards the actual solution. A* will never loop because every duplicate is filtered by the hash map before it is re-evaluated. The moves of

Max Moves: 2 | h1 solved: 3 | h2 solved: 3 | beam solved: 3
Max Moves: 4 | h1 solved: 15 | h2 solved: 15 | beam solved: 15
Max Moves: 6 | h1 solved: 51 | h2 solved: 51 | beam solved: 47
Max Moves: 8 | h1 solved: 152 | h2 solved: 152 | beam solved: 125
Max Moves: 10 | h1 solved: 420 | h2 solved: 420 | beam solved: 317
Max Moves: 14 | h1 solved: 2874 | h2 solved: 2874 | beam solved: 1535
Max Moves: 18 | h1 solved: 17402 | h2 solved: 17402 | beam solved: 4863
Max Moves: 24 | h1 solved: 116088 | h2 solved: 116088 | beam solved: 15286

Max Moves: 30 | h1 solved: 181217 | h2 solved: 181217 | beam solved: 32724
Max Moves: 40 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 77173
Max Moves: 50 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 121223
Max Moves: 75 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 171753
Max Moves: 100 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 180342
Max Moves: 150 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 181438
Max Moves: 200 | h1 solved: 181440 | h2 solved: 181440 | beam solved: 181440

D. As discussed, every puzzle is solvable with h2 within 50000 nodes, every puzzle is solvable within h1 within 15000 nodes, and every puzzle is solvable with beam for k > 2.

Discussion

Based on these experiments, A* search is better for a shortest path solution, but beam search is better for finding solutions quickly and is more memory efficient. H1 searches all solutions in 2571716ms, h2 in 181174ms, and beam in 63346ms. Beam search never has more than the hash table + k * 4 states in the priority queue + k in the array. A* search has the same hash table and a much higher bound on the number of nodes in the priority queue. A simple estimate is that there are no more than 4 * number of nodes explored in the priority queue. This is obviously much larger than k * 5. However, the path of the solution from beam search is generally much longer than the path returned by A*.

Beam search was significantly more difficult to implement than A* search. Beam search does not fit the 8 puzzle as nearly as A* search fits the 8 puzzle because the path is important rather than just finding the goal state. Generating k-1 random states and keeping track of the moves is somewhat strange. Beam search was also more difficult to implement because I tried to limit repeated moves to limit loops. This improved the efficiency and success rate of the algorithm, but is much less straight forward than just adding and removing from a priority queue like in A*. A* search is better suited to the problem in terms of simplicity and path efficiency, but beam search is a faster algorithm.

Extra Credit:

I implemented a 2x2x2 Rubik's cube reusing most of the code from the 8-puzzle solution. I had to define new moves for the cube, a new state, a new heuristic, and a new child of game, but nearly all of the code was re-usable. The cube stores its state in an array of blocks. The cube is composed of 8 blocks with a set of 3 faces, one parallel to each plane (x, y, z). The cube also has an id, the integer number of its position in the goal state. It is not possible to have all 8 corners of a 2x2x2 in the correct position and not be in the goal state. This id is used for the heuristic. The heuristic is defined as the total distance to the goal position of each block. The heuristic is this sum / 4 to keep it admissible. Every move changes the positions of 4 blocks, and the heuristic cannot overestimate the cost to the goal.
Because the cube is isomorphic, one block always remains fixed as a point of reference. Moves are defined relative to this block (the top left front corner). A move is a quarter turn either clockwise or counterclockwise in one of the 3 planes of blocks not including the reference block. My moves are called XY+, XY-, XZ+, XZ-, YZ+, YZ-, where XYZ represent axes of the coordinate plane. A move moves the block to a new position in the array and than calls block.rotateXY etc. This method swaps the colors on the two specified faces. If the YZ plane is rotating, the face perpendicular to the x direction does not

change direction when the cube is rotated. The y direction face and z direction face swap. This is true for every move on the cube. I'm sure that there are better heuristics for this cube, but mine seems to work. It expands much more nodes than were required for the 8 puzzle. A 10 move solution required expanding 133075 nodes. Similarly, solving a random state with beam search k = 10 resulted in a 3614 move solution. I did not have time to run extensive tests but I assume a better heuristic would greatly improve both of these search algorithms. A text file rubikDemo.txt is included with the project to demo the problem. The rubiks cube can be run as java RubiksCube rubikDemo.txt. In the demo, A* takes a reasonably long time to find an 8 move solution, then beam search very quickly finds an 8246 move solution. Every file related to the rubiks cube solution in my code if prefixed Rubik.