1. Code design

Code Outline:

1. Class EightPuzzleSolver
   1. Fields
   2. Main method
   3. Constructor
   4. Methods to read commands from a file or the command line
      1. Including a gigantic switch block in a submethod executing the commands
   5. Methods handling the non-searching commands
   6. A\* Search method
   7. Beam Search method
2. Class Node<int[]>
   1. Fields
   2. Constructor
   3. Get methods
   4. Set methods
   5. equals method
   6. compareTo method

When EightPuzzleSolver is run, the constructor (called by main) will call the method that accesses and execute commands from a given file, then unless the file includes an “Exit” command it will switch over to the method that handles manual input of further commands. If no file is provided the program goes directly to manual input. All functional code is called from the commandLoop submethod.

commandLoop was written as a switch block to reduce visual clutter in the source code.

While the other commands manipulate an integer array directly, the search methods require the board state to be encapsulated in a node to pass additional data (heuristic and path). I wrote this node so it would possess exactly the required fields and so I could explicitly define a natural comparator (which, as it works with the heuristic, is not consistent with equals). It was not similarly necessary to write the priority queue used by the search methods.

The program makes liberal use of try-catch blocks. When I was writing the first version of randomizeState, I used if-else blocks based on the current position of the blank to explicitly prevent array index errors on movement commands. When I couldn’t figure out why it wasn’t working, I changed to try-catch blocks where the catch blocks did nothing, and it worked. They’re also to my knowledge the normal way of dealing with scanners and file accesses.

Of note is the difference in the return structure of the search methods. A\* will find and return the goal state node, after which further code in commandLoop extracts the sequence of moves the search method made to get to it. The null return type is used to flag the failure of the search. In contrast, the beam search method uses a field to store the moves of its goal node and returns a Boolean describing whether the search was successful. They are different because I wrote them that way in case one inexplicably failed, and I never changed them to match since they’re functionally equivalent in this context (if I did most likely I would make beam more like A\*).

Error codes:

1. The program doesn’t have a clue what you tried to say
2. The program thinks you tried to randomizeState but provided an illegitimate number
3. The program thinks you tried to solve A-star but provided an illegitimate number
4. The program thinks you tried to invoke maxNodes but provided an illegitimate number
5. The program thinks you tried to solve beam but provided an illegitimate number

2. Code Correctness

A\* is working excellently now. Beam still tends to fill its entire memory with clones of local minima but for sufficiently high ratios of k to solution length (I don’t think it’s linear) it agrees exactly with A\* in all tested cases.

Section 1: A\*:

The commandLoop calls the A\* method with the current board state. This method initializes a priority queue and the root of the search tree before sending them off to the main looping method.

starLoop initializes a hash set to store visited states (in this application, the first time any state is visited is always the lowest-cost path to that state) and other variables, then begins a search loop. The loop breaks on the failure condition of currentNodes >= maxNodes with a return null statement, which tells commandLoop to print the failure notification. currentNodes is incremented each time a node is popped off of the queue, while maxNodes is a constant passed into the search as a parameter. starLoop will also return failure if it is passed an empty pq or if the pq is empty at any point, but this should be impossible during normal operation (I will note that it was triggered during a standard h2 search when I mistyped a board state to have two sixes and no eight). The other return condition is success, which is determined by comparing the heuristic field of each node popped off of the queue with its depth. Since for A\* h(n) = hn(board) + depth(node), this signifies that hn(board) = 0. In this case the node containing this goal state is returned, which allows commandLoop to extract the sequence of moves from it and its parents.

If neither success or failure is triggered, the search functionality is engaged, acting like an enormous loop iterator on the priority queue. Each of the possible children of the expanded node are generated from a preserved clone of its board state. For each new board, if its equivalent is not contained in a node within the hash set of visited states its fields are set and it is added to both the hash set and the queue (fields: move for record-keeping, parent for tree traversal for move list extraction, depth and heuristic for goal checking and queue ordering).

Node’s equals method is explicitly overridden to compare each index of the board states between the calling node and the parameter node, supporting desired functionality from the hash table’s contain method.

Heuristic 1 is calculated by a simple loop that runs through the board state and increments whenever the element in a slot doesn’t match the index of that slot. Heuristic 2 similarly detects when a particular element is not in the correct spot, but further calculates how far it is from the correct spot (counting vertical moves to the correct row, then horizontal moves to the right spot). The total A\* heuristic sums that value and the depth of the node in the tree, which corresponds to the length of the path from the root to that node.

Node’s compareTo is explicitly overridden to directly compare values in heuristic fields so that the head of the priority queue always has the lowest heuristic (no explicit tie breakers).

**public** Node<**int**[]> aStar(**int**[] board, **int** maxNodes) {

PriorityQueue<Node<**int**[]>> pq = **new** PriorityQueue<Node<**int**[]>>(100);

Node<**int**[]> root = **new** Node<**int**[]>(board);

root.setHeuristic(calculateHeuristic(root.getCargo(), heuristicType)); //root depth is zero

pq.add(root);

**return** starLoop(pq, maxNodes, 1);

}

**private** Node<**int**[]> starLoop(PriorityQueue<Node<**int**[]>> pq, **int** maxNodes, **int** currentNodes) {

HashSet<Node<**int**[]>> hash = **new** HashSet<Node<**int**[]>>();

Node<**int**[]> expandable;

Node<**int**[]> first;

Node<**int**[]> second;

Node<**int**[]> third;

Node<**int**[]> fourth;

**int**[] boardPrime = **new** **int**[9];

**int**[] board;

**while** (currentNodes < maxNodes) {

expandable = pq.poll();

**if** (expandable == **null**) {

System.***out***.println("Unexpected Behavior, see source line ~403");

**return** **null**; //Should be impossible to ever trigger this.

}

**if** (expandable.getHeuristic() == expandable.getDepth()) { //i.e., h(n) = 0

**return** expandable; //Search success

} //Otherwise the search must continue. The lines below are effectively one large loop variable iterator

**if** (! hash.contains(expandable)) {

hash.add(expandable);

currentNodes++;

first = **new** Node<**int**[]>(**null**);

second = **new** Node<**int**[]>(**null**);

third = **new** Node<**int**[]>(**null**);

fourth = **new** Node<**int**[]>(**null**);

**for** (**int** i = 0; i < 9; i++) {

boardPrime[i] = expandable.getCargo()[i];

}

**int** newDepth = expandable.getDepth() + 1;

**try** {

board = boardPrime.clone();

first.setCargo(moveUp(board));

**if** (! hash.contains(first)) {

first.setMove("Up");

first.setParent(expandable);

first.setDepth(newDepth);

first.setHeuristic(newDepth + calculateHeuristic(first.getCargo(), heuristicType));

pq.add(first);

}

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

}

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

}

} **catch** (ArrayIndexOutOfBoundsException e) {}

}

}

**return** **null**;

}

Section 2: Beam

commandLoop calls localBeam with the requisite information, that being the initial state, the k value, and maxNodes. The header method sets up a priority queue stateMemory (the primary queue) to hold the k states of interest and adds the initial state node to it, then calls the looping submethod, which is architecturally similar to the one for A\*. (minus the aforementioned return type difference).

The submethod first initializes a second priority queue stateCollector (the secondary queue) to hold all generated successors of the k states in stateMemory, as well as some node and board variables. It then begins looping. Termination conditions are that currentNodes has exceeded maxNodes or an internal return statement has been reached, i.e. a solution was found. The loop comes in two layers: the inner loop as long as stateMemory has nodes will poll one of them (and increment currentNodes), do some variable maintenance, then generate each child of the polled state. If it is a goal state, the method returns immediately, otherwise the new node (with all relevant field information set) is added to the stateCollector queue. Once this process has generated all of the children from the nodes formerly in stateMemory, the inner loop terminates (condition: stateMemory.peek() == null) and the outer loop moves the k best states from stateCollector to stateMemory and flushes the rest. Then the process repeats.

When a goal state is found, prior to returning true the loop calls the move extraction method on the node containing the goal state. This method simple initializes an ordered list and appends the move of the current node, followed by the move of its parent, etc. until it reaches root of the tree (parent == null). This list is stored in a field so it can be accessed by the printing code in commandsLoop.

It is worth noting that beam search is not protected from clogging up in local minima. Given sufficient iterations, local minima states will accumulate in stateMemory until they occupy it entirely and prevent further meaningful searching, unless a state with a better heuristic is found from the nonminima states before the redundant ones flush them out. Functionally, this means that the requirement for the size of k increases very fast and nonlinearly with the length of the shortest solution to the initial state. Beam was able to successfully solve a randomly generated 14-length solution board state but required k to be around 375.

**public** **boolean** localBeam(**int**[] state, **int** numStates, **int** maxNodes) {

**int**[] goal = {0,1,2,3,4,5,6,7,8};

**if** (Arrays.*equals*(state, goal)) {

**return** **true**; //Check on being fed the goal state

}

PriorityQueue<Node<**int**[]>> stateMemory = **new** PriorityQueue<Node<**int**[]>>(numStates);

stateMemory.add(**new** Node<**int**[]>(state));

**return** beamLoop(stateMemory, numStates, maxNodes, 0);

}

**private** **boolean** beamLoop(PriorityQueue<Node<**int**[]>> stateMemory, **int** numStates, **int** maxNodes, **int** currentNodes) {

PriorityQueue<Node<**int**[]>> stateCollector = **new** PriorityQueue<Node<**int**[]>>(4\*numStates);

Node<**int**[]> expandable;

Node<**int**[]> first;

Node<**int**[]> second;

Node<**int**[]> third;

Node<**int**[]> fourth;

**int**[] boardPrime = **new** **int**[9];

**int**[] board;

**while** (currentNodes < maxNodes) {

**while** (stateMemory.peek() != **null**) {

currentNodes++;

expandable = stateMemory.poll();

boardPrime = expandable.getCargo().clone();

first = **new** Node<**int**[]>(**null**);

second = **new** Node<**int**[]>(**null**);

third = **new** Node<**int**[]>(**null**);

fourth = **new** Node<**int**[]>(**null**);

**try** {

board = boardPrime.clone();

first.setCargo(moveUp(board));

first.setMove("Up");

first.setParent(expandable);

first.setHeuristic(calculateHeuristic(first.getCargo(), 2));

**if** (first.getHeuristic() == 0) {

beamMoves = extractMoves(first);

**return** **true**;

}

stateCollector.add(first);

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

}

stateCollector.add(third);

} **catch** (ArrayIndexOutOfBoundsException e) {}

**try** {

...

} **catch** (ArrayIndexOutOfBoundsException e) {}

}

**for** (**int** k = 0; stateCollector.peek() != **null** && k < numStates; k++) {

stateMemory.add(stateCollector.poll());

}

stateCollector.clear();

}

**return** **false**;//Failure case of exceeding the node limit

}

**private** LinkedList<String> extractMoves(Node<**int**[]> node) {

LinkedList<String> moves = **new** LinkedList<String>();

**while** (node != **null** && node.getParent() != **null**) {

moves.addFirst(node.getMove());

node = node.getParent();

}

**return** moves;

}

3. Experiments

The method experiments(), called by commandsLoop on reading “exeriments”, produced the data used to answer the below. In a compromise between interest of generating good statistics and the ability to run the code in a reasonable time, for each of 21 quantities of maxNodes 3000 searches are run on boards randomized with 200 moves, 1000 searches per method. Spacing is not even, as A\*-h2 behavior is not interesting after 6000, A\*-h1 is not interesting after 60000, and beam changes slowly if at all.

Data: % solvable directly below headers, with average lengths of solutions in the next-right column.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| maxNodes | A\*-h1 |  | A\*-h2 |  | Beam (k=400) |  |
| 1000 | 14.1 | 14.92 | 38.8 | 17.81 | 0.4 | 7 |
| 2000 | 25.2 | 16.11 | 59.6 | 19.04 | 1.7 | 8.59 |
| 3000 | 27.7 | 16.77 | 72.4 | 19.9 | 4.2 | 12.19 |
| 4000 | 37.3 | 17.44 | 80.2 | 20.26 | 6.3 | 12.7 |
| 5000 | 39.5 | 17.85 | 85.8 | 20.71 | 9.5 | 14.93 |
| 6000 | 43.2 | 17.94 | 89.5 | 20.67 | 11.1 | 15.84 |
| 11000 | 64 | 19.41 | 97.3 | 21.31 | 13.7 | 17.08 |
| 16000 | 67 | 19.96 | 99.2 | 21.63 | 14.1 | 18.14 |
| 21000 | 83.5 | 20.3 | 99.7 | 21.31 | 17.5 | 17.36 |
| 26000 | 83.6 | 20.55 | 100 | 21.56 | 14.5 | 17.64 |
| 31000 | 86 | 20.37 | 100 | 21.25 | 13.7 | 16.54 |
| 36000 | 90.1 | 20.65 | 100 | 21.27 | 15 | 17.64 |
| 41000 | 95.9 | 21.12 | 100 | 21.43 | 13.7 | 17.39 |
| 46000 | 97.3 | 21.17 | 100 | 21.39 | 14 | 17.24 |
| 51000 | 96.8 | 21.2 | 100 | 21.44 | 14 | 17.74 |
| 56000 | 97.1 | 21.19 | 100 | 21.43 | 14.5 | 17.5 |
| 61000 | 97.5 | 21.27 | 100 | 21.48 | 12.6 | 17.3 |
| 91000 | 100 | 21.42 | 100 | 21.47 | 15.5 | 17.99 |
| 121000 | 100 | 21.55 | 100 | 21.59 | 12.2 | 18.11 |
| 151000 | 100 | 21.42 | 100 | 21.46 | 12.9 | 17.91 |
| 181000 | 100 | 21.43 | 100 | 21.46 | 13.1 | 17.36 |

a. See relevant columns above.

b. From a theoretical standpoint, h2 is superior because it can vary more greatly over the same set of states. For instance, h1 ( 483 015 627 ) = h1 ( 483 105 627 ), but h2 on the second state will be 2 more than h2 on the first state – meaning that h2 is capable of supplying more information than h1.

Data in the table bears this out. H2 is capable of more solutions with fewer nodes searched. Notably, beam’s performance does not depend strongly on maxNodes after about 10000, and more likely is most affected by the choice of k.

c. See table. Beam is always less capable of finding longer solutions than the other two, which correspond fairly closely except at lower maxNodes, where h2 is generally higher.

d. See table.

4. Discussion

a. Certainly A\* with h1 is inferior (there’s a table in the lecture slides comparing it to A\* with h2). Since beam is prone to clogging in local minima but A-star h2 is not, the latter is clearly superior and the data supports this conclusion. Between A\*-h1 and beam, from experiments it is indicated that A\*-h1 outperforms beam.

b. The vast majority of implantation difficulty stemmed from PEBKAC issues (mainly pointer mishandling) more than inherent difficulty in the algorithms.