CS486 A1

Jason Sun (#20387090)

May 28, 2015

1 Informed Search

1. Misplaced tile vs Manhattan distance heuristics

   (a) The Manhattan distance more accurately reflects the cost of each tile's displacement from its goal configuration. Consider two cases: case A where the 1 and 8 are interchanged versus case B where 1 and 2 are interchanged (from the goal configuration, labeled "c" in the assignment):

   - The number of misplaced tile, for both cases, is equal to 2. But obviously case B is easier to solve than case A, which involves shuffling around more tiles and thus performing more steps.
   - The Manhattan distance for both cases are different: the cost estimate for case A is higher than case B. Indeed it does cost more to solve case A because of more shuffling of tiles involved.

   (b) By definition[1], a heuristic is *consistent* (also called *monotone*) if, for ever node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

   $$h(n) \leq c(n, a, n') + h(n')$$

   In the case of an 8-puzzle, each node n represents a configuration state of where the tiles currently occupy on the tile. Each action a is an action upon the blank position: up/down/left/right. The cost of any action is 1, because it takes one move to get to the next configuration n'. Now let's see if the two heuristics are consistent.

   i. Number of misplaced tile is *not consistent*. I claim this heurstic violates monotonicity. To be consistent is the same as predicting the cost in a monotonic fashion. That is, the cost estimate to arrive at the goal state must be strictly increasing for each step. But this is not the case. Take for example figure (a) in the given

[1] AIMA (3rd ed.) p. 95

assignment. Because there is only one tile is out of place (the 6 tile), we know the previous state n_- would've had at 2 misplaced tiles, because it is impossible to have less than 2 misplaced tiles due to where the blank space is located. So $h(n_-) \geq 2$ and the current state n has exactly 1 misplaced tiles, $h(n) = 1$. But any next action we make, the 6 tile cannot be put in the right location. Thus the next state n_+ would have at least 2 misplaced tiles. Therefore $h(n_+) \geq 2$. And we have it: the heurstics is not monotonic.

ii. Manhattan distance is *consistent*. This heurstics does not have the downfall of the previous one; with the previous one it was possible for $h(n)$ to decrease and then increase (the herustic failed to account for tiles that would have to be displaced again) – whereas the Manhattan does account for this. It accounts for this by factoring in the distance the tile is from the desired goal state. Thus potentially displaceing all those tiles. For all next moves that we need to displace a tile, the displaced tile's distance to its desired block is accounted for.

2. IDA*

   (a) IDA* use at most $O(bd)$ memory, where d is depth, since A* picks a single node to expand at each level but still need to keep the other nodes queued up in case of backtracking. In terms of cost, the nodes on the bottom level (depth d) are explored once, those on the next-to-bottom are explored twice.. up until to the root. Hence $O(b^d)$ time cost in the worst.

   (b) Complete because in the worst case IDA* performs exactly the same as depth first search A*, and we know A* is complete, assuming good heuristic.

   (c) Optimal because if a goal state is found it must be the least deep, since the depth is only expanded after exploring all the nodes at that depth.

2 Constraint Satisfaction

2.1 Formulating Sudoku as CSP

Variables 81 variables, 1 representing each square.

Domains Empty squares have domain of $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Constaints 27 *Alldiff* constraints: one for each row, column, and box of 9 squares.

2.2 Sudoku Solver Runtime Summary

Source code is attached. Their md5 hashes should be:

```
MD5 (BacktrackSolver.java) = 5e20bf43beef413d04cacc2f94d8fdd1
MD5 (Cell.java) = 50480e3aad6a2496d6ee32802e6787d8
MD5 (ForwardCheckingSuccessor.java) = d9be554d5157b134fb414ea7be73a925

MD5 (Grid.java) = f6941858f85bd31baa0b91897f514e35
MD5 (HeuristicsSuccessor.java) = 94dcd29b549f0481dc6d9a7c5c8e7fa3

MD5 (Main.java) = a548d87bd0ce7de51b2f7ee69edf4f36
MD5 (Puzzles.java) = 9a124be28d8ea4218cc402ffea612029
MD5 (RandomSuccessor.java) = 0b0c0fe02e479c2da89b28c73a922749
MD5 (RunStat.java) = 43888d6f91291329603a0020d8710f17
MD5 (Statistics.java) = d011ea8efbd8d40cc4e68afe606c0b39
MD5 (SuccessorFunction.java) = d2aaa2b2f3d60eace6726877c44fdaf9
```

All timing performed on 2.3 GHz Intel Core i7, 16 GB 1600 MHz DDR3, Java 1.8.0_40 64-bit.

Format is in avTime ± stdTime in milliseconds. They are run 50 iterations.

Time

	B	B+FC	B+FC+H
Easy	58737.020000 ± 138057.375015	0.840000 ± 1.222457	0.020000 ± 0.140000
Medium	Timed out	2.760000 ± 2.518412	0.560000 ± 0.571314
Difficult	Timed out	2.320000 ± 2.266627	0.340000 ± 0.473709
Evil	Timed out	2.660000 ± 3.541807	0.640000± 0.520000

# of Nodes

	B	B+FC	B+FC+H
Easy	7913745.800000 ± 19000004.059474	2.000000 ± 0.000000	2.000000 ± 0.000000
Medium	Timed out	16.480000 ± 7.884770	16.000000 ± 0.000000
Difficult	Timed out	30.820000 ± 20.308314	19.000000 ± 0.000000
Evil	Timed out	55.780000 ± 60.872421	15.000000 ± 0.000000

Note on time out

The timinig on the easy puzzle that the timing results varies a lot, and it's certain expected that medium, difficult, and evil puzzles would vary even more. I gave the same amount of and some of them took longer than 1 hour to solve.

Solution for Each Test Puzzle

Format: 81 numbers reading from the top to bottom column, left to right.

Easy

359768142218943675746152893583297461427631589961485237872314956194576328635829714

Medium

876129435953487216421536798719864523342975861568312974185793642694251387237648159

Difficult

612859473395147286478623951256481397831796524749235168563918742184572639927364815

Evil

152749368794638152863251497471396285925874631386125749639512874248967513517483926

Code Details & Explainations

About the runtime.

- The runtime behaviour for B+FC+H is certainly expected, the number of goal states explored is expected to be the same for every run because the input is the same. Having heurstics implies having a deterministic way of going about to solve it. The timing varied a bit, but not by much. The timing is too short, the stddev is as large as the timing itself. So timing differences is probably due to kernel's scheduler.
- The runtime behaviour for B+FC is expected to have some variance, because the next states are not picked in a consistent manner. It is sitll much much faster than simple B alone.
- The runtime behaviour for B is definitely expected to be extremely long. I've tried to implement efficiently the code by using bitset representations for the cells. But still, the sheer number of states to explore is definitely expected. Although I suspect my code does better than others because I randomize order in which the next cell is to be filled and do not try to fill the same cell again.

Below is about the code.

The code is written in java and distributed across several files, in the java style. I describe those files and explain what they do below.

Main

Mostly administrative stuff. Calls 3 timing functions, for each type of implementation. Collects the runtime stats and calls the mean and stddev calculations.

BacktrackSolver

Implements the backtracking algorithm found in textbook p. 215 of AIMA, 3rd ed. This function is designed to take a `SuccessorFunction` interface as input and we can abstract how the calculation part.

At each stage, recursive call, the grid is checked that it should be still possible to solve and that there does not exist internal conflicting numbers.

SuccessorFunction

This is an interface. Three implementations available: `RandomSuccessor`, `ForwardCheckingSuccesor`, `HeursticsSuccessor`, which respectively represents B, B+FC, B+FC+H solving methods.

RandomSuccessor

A random unfilled cell is chosen and new states (`Grids`) are generated for each possible value in its list. The sort is actually a consistent random shuffle, for the sole purpose of keeping track what cells we have already guessed, so to prune the search space and not repeatedly guess the same cells. It is consistent because the cell comparator is created once and the value is final. It is random because the value is randomized on creation. I think this is pretty neat and include the relevant snippet below.

```
public static class CellComparatorRandom implements Comparator<Cell> {
    private static Random random = new Random();
    private final int xMult = random.nextInt(1000);
    private final int yMult = random.nextInt(1000);
    private final int truncation = 200;
    @Override public int compare(Cell o1, Cell o2) {

        int order1 = (o1.x * xMult + o1.y * yMult) % truncation;
        int order2 = (o2.x * xMult + o2.y * yMult) % truncation;
        return order1 - order2;
    }
}
```

ForwardCheckingSuccessor

This builds upon the random successor, except it does FC, which does filtering on potential values on unfilled cells. For every unfilled cell, we can eliminate potential values that we've already seen in the row/column/region. A region is a 9x9 cell block.

HeuristicsSuccessor

This builds upon FC. The heuristics essentially sorts the collection of potential states by their 3 criterias. I implement the three criterias as comparison operators, for comparing state, into a priority queue.

Cell

A cell is just a list of values, and it is valid if and only if there is one value in the list. For efficiency, the list of possible values for the cell is implemented from `BitSet` for efficiency.

Grid

A grid represents a *state* in solving the puzzle. It is a collection of 9x9 cells, and it indexes into the cell via a statically allocated array.

Statistics

A class that contain methods to calculate stddev and mean.

RunStat

This is a container to hold the result of an execution call to the `BacktrackSolver`. It contains the timing and number of nodes explored.

Puzzles

This is a container to store the inputs. There are 4 strings, each of length 81 and represents a test puzzle. The puzzle is represented in a left to right, top to bottom fashion. Unknown cells are filled with dashes `'-'`.