Theory of Computer Games Homework 1

1. How to Build the Pattern Database?

Building the pattern database is required before running the Pattern-database solver.

1. **Config**

   The pattern database generator takes a config in the following format:

   - The config starts with a “base board” indicating the position of the non-movable tiles. For a $M \times N$ board, it takes M lines. For each line, there are N numbers separated by whitespaces. The number of a tile should be -1 if the tile is non-movable, otherwise 0.
   - After the “base board”, list the patterns. Each pattern takes 2 lines. The first line is the path where the generated pattern database should be written to. The second line is the numbers in the pattern, separated by whitespaces.

   Note that to ensure correctness, the patterns should be disjoint, i.e. a number should not appear in multiple patterns.

2. **Compile**: `make database`
3. **Execute**: `./pattern_database_generator.out < <config-path>`

For example, to meet the homework requirement (< 30 minutes of pre-processing time and < 1GB data generated), use the config at `pattern_database/2_2_2_2_2_2_1_1_1.in`. Run the following command.

```
make database
./pattern_database_generator.out < pattern_database/2_2_2_2_2_2_1_1_1.in
```

2. How to Compile and Run the Solver?

1. **Compile**: `make`
2. **Execute**
   - *Brute-force solver*: `./main.out --brute-force < <testcase>`

- *A-star solver*: `./main.out --a-star < <testcase>`
- *Pattern-database solver*: `./main.out --pattern-database <config-path> < <testcase>`

For example, if you want to compile and run the Pattern-database solver with config `pattern_database/2_2_2_2_2_2_1_1_1.in`, and pass in the test case `test_case/case1_1.in`, run the following command.

```
make
./main.out --pattern-database pattern_database/2_2_2_2_2_2_1_1_1.in < test_case/case1_1.in
```

3. Algorithms

The *Brute-force solver* uses the brute force algorithm. The *A-star solver* and the *Pattern-database solver* both use the A* algorithm, but applies different heuristics.

3-1 Brute-Force Algorithm

Here we apply the breadth-first-search (BFS) algorithm.

Start by pushing the initial board configuration to the queue. Then, for every iteration, the solver pops the queue's top element, and check if it has reached the terminal state. If the board has not reach the terminal state, push all possible next configurations into the queue. If the terminal state is reached, output its number of moves.

This algorithm is used by the *Brute-force solver*.

3-2 A* Algorithm with Heuristics

The A* algorithm is quite similar to the brute-force algorithm. The main difference is that it uses a priority queue instead of a simple queue. The priority queue is sorted based on the sum of the following two costs.

1. The current cost (i.e., the number of moves so far)
2. The estimation of remaining cost, or the heuristic cost

In this project, we've implemented two types of heuristic:

- **Manhattan distance heuristic**

  For each tile, compute the Manhattan distance between its current position and its goal position. The heuristic cost is the sum of Manhattan distance of all tiles.

- **Pattern database heuristic**

  Here we pre-compute the minimum number of moves to move a set of tiles to its goal position while viewing others as don’t cares. Given a set of disjoint patterns, the pattern database heuristic is the sum of these minimum number of moves.

The *A-star solver* implements the A* algorithm with the Manhattan distance heuristic.

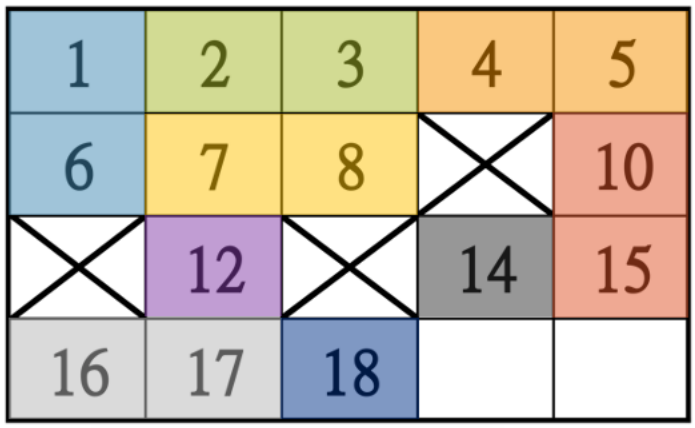
As for the *Pattern-database solver*, it implements the A* algorithm with a combination of the Manhattan distance heuristic and the pattern database heuristic. The heuristic cost is taken as the larger between the Manhattan distance heuristic and the pattern database heuristic.

4. Pattern Database Design

Given the time limit (30 min) for the database generation, we adopt the following pattern.

- Tile 1 & Tile 6
- Tile 2 & Tile 3
- Tile 4 & Tile 5
- Tile 7 & Tile 8
- Tile 10 & Tile 15
- Tile 16 & Tile 17
- Tile 12
- Tile 14
- Tile 8

A visualization of the split is shown below.



Take the first group for example.

First, we treat all movable tiles that is not numbered 1 or 6 as *don't cares*, e.g. treat all of them as 31.

Then, generate all possible starting board configuration. That is, all combination of the 1's, 6's and 0's. In this case, there will be $(17 \times 16 \times 15 \times 14)/2 = 28560$ possibilities.

Pass them to the A* solver. Compute the minimum number of moves that is required to put tile 1 and tile 6 in the target position.

Then, write the result to a file in binary. Every starting board configuration maps to 2 entries. The first is the 100-bit representation of the board. Each tile is represented with 5 bits. The second is an integer indicating the minimum number of moves to put the pattern into its target position.

The total size of the generated databases is about 4MB.

5. Experiment Results

The following shows the result of running the 3 solvers on the public test cases. The numbers show the number of explored states.

The execution time is not listed here as most of the test cases complete within 1 second on the NTU CSIE workstation. Its magnitude is too small that comparison results may be easily affected by the machine's workload variations. The number of explored states will be a better indicator of the solver performance.

	Brute-force solver	**A-star solver**	**Pattern-database solver**
case1_1	462	26	26

	Brute-force solver	A-star solver	Pattern-database solver
case1_2	130	9	9
case1_3	1129	19	18
case1_4	871	53	53
case1_5	1720	38	38
case2_1	5993	47	44
case2_2	169208	575	248
case2_3	9053	97	93
case2_4	10704	55	53
case2_5	12570	46	48
case3_1	1680590	1793	1825
case3_2	463650	2408	1152
case3_3	2580805	1657	1438
case3_4	477302	3278	1441
case3_5	275632	904	614

From the table above, we can see that the *A-star solver* explores a large smaller number of states than the *Brute-force solver.* Through the Manhattan distance heuristic, the solver successfully avoids visiting the board states that are far from completion.

Comparing the *A-star solver* and the *Pattern-database solver*, we can see that the *Pattern-database solver* explores less states in 60% of the test cases. The two numbers appear the same for 27% of the test cases. And in 13% of them, the *Pattern-database solver* explores slightly more than the *A-star solver*. Though for a given board configuration, the heuristic cost of the *Pattern-database solver* is at least as large as that of the *A-star solver*, there still exists some boards whose cost is estimated badly. As the order of the board states are not guaranteed to be improved, it is not ensured that the *Pattern-database solver* always performs better then the *A-star solver*.

6. Discussion

In our implementation, the pattern database does not improve the performance a lot. We think that this may be due to the size of our patterns. In our current design, each pattern contains at most 2 tiles. However, the number of tiles in a pattern may be increased if we improve the performance of our pattern database generator.

Currently, the generator enumerates all possible starting board configuration. Then it computes the corresponding minimum number of moves by calling the *A-star solver*. Given that tile slides are invertible, this process can be largely improved by recording the reached states and their current number of moves while performing a BFS from the terminal board configuration. Through this method, the pattern database can be generated much faster, and larger database can be generated under the time limit.