Solving Rubik's Cube

Final Project in Introduction to Artificial Intelligence

Bar Iluz
Nadav Yunik
Ben Cohen
Noa Portugal-Gadot

Abstract—Solving a rubik's cube is a familiar problem that already has a fixed algorithm apprehendable for humans, however, such algorithm is not optimal in the solution length. The goal of our project is to use AI methods learned in class, such as informed search, to solve a random state of the cube optimally while reducing the amount of work done in the search (decreasing search time and the number of expanded node). We will explore various heuristic functions and compare their effectiveness on the search.

I. INTRODUCTION

Rubik's Cube is a 3D combination puzzle invented in 1974 by Hungarian sculptor and professor of architecture Ernő Rubik. The motivation for developing the cube was that he wanted to create a model that explains the three dimensional geometry to his students and use it as a teaching tool. Also known as the “Magic Cube”, the Rubik's cube has become a hit worldwide for nearly a generation. Individuals of all ages have spent hours, weeks or months trying to solve the cube. The cube became a world’s most famous, best selling and well-known puzzle to have a rich underlying logical, mathematical structure. The standard version is a 3D combination puzzle with six faces consisting of nine (3X3) tiles where each of the faces have one of six colors; white, red, blue, orange, green and yellow. The puzzle is scrambled by making a number of random moves, where any cube can be twisted 90, 180 or 270 degrees. The task is to restore the cube to its goal state, where all the squares on each side of the cube are of the same color. To solve a scrambled Rubik's cube, one needs an algorithm, which is a sequence of moves in order to come closer to a solution of the puzzle. Such sequences usually require more than fifty to hundred moves for a regular human being. However, Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge prove that God's Number for the Cube is exactly 26 [1] in quarter turns metric in which (+/-)90 degree face turns are allowed.

The problem is quite difficult. The most basic way a computer can represent a Rubik's cube is to use graph theoretical concepts and perform graph search with BFS which is guaranteed to return an optimal solution in terms of the number of moves needed. However, There are $4.3252 \times 10^{19}$ different states that can be reached from any given configuration, and such brute search is impractical on large problems. To overcome performance IDA* algorithm is used to reduce the search space significantly using heuristics.

II. APPROACH AND METHODS

First, we defined our problem as a search problem

< states, initial state, goal state, successor function >

where the nodes of the graph are the states of the cube, the root is an initial scrambled cube, the goal state is the cube in which all the tiles on every face have the same color, and the successors of each state are the cubes that obtained from rotating one face (+/-)90 degrees.

A. Terminology

Cubie - the cube is composed of 27 atomic cubes (in the sense that they can’t be further decomposed) each is called a cubie.

Tile - the outer facet of a cubie which is exposed to the user.

Faces - consists of 9 cubies and are referred by right(R), left(L), up(U), down(D), front(F) and back(B).

Corner - a cubie with 3 tiles.

Edge - a cubie with 2 tiles.
Center cubie - a cubie with 1 tile in the middle of each face.
out of the 27 cubies only the edges and the corners move while the center cubies are attached to the hidden central cubie.

B. State representation
We used a basic representation of a state as an array of 6 matrices each representing a face by the order [U:0, D:1, F:2, B:3, R:4, L:5]. Each face is a 3 by 3 matrix where each entry represents a specific tile and the number in that entry represents the color of the tile. Moreover, we kept another array similar to the above mentioned that represents the orientation of each cubie which is essential for some of the dB representations we use further in the project. The orientation\(^2\) of a corner cubie is determined by examining the unique facet of the cubie which faces up or down in the goal state (the corner tile with the "+" mark). Its orientation is the number of 120 degrees clockwise (with respect to the start and the end of the axis) rotations of the cubie about an axis from the center of the cube through the corner of the cubie which would map the up or down facet of its current position to the top or bottom side of the cube (the value in the tile under the "+" mark). For the edge cubies their orientation is zero if they can go to their goal position correctly oriented without using any of the left or right operators and one otherwise(using the predefined "+" mark).

C. Search Algorithm
As mentioned, the state space is too large to compute with brute BFS, thus, we resort to A* search method with different heuristics. That said, the regular version of this method still can’t handle efficiency with the memory issue inflicted by the complexity of the problem hence we use it’s more memory efficient version, the IDA*.

We used two main methods – heuristics and pattern DBs in each we explored different functions.

D. Heuristics
Heuristics which were used in IDA*

- 3D Manhattan distance:
  - Max Manhattan distance (not admissible): computes the maximum between the sum of the Manhattan distances of all corners from their right position and orientation, and the sum of the Manhattan distances of all edges from their right position and orientation.
  - sum Manhattan distance : Computes the Manhattan distance of every tile to to its original place in the face it belongs to (the face with the same color in the middle tile) and sums the results.
  - in order to make it admissible, we divided the result by 8 since every twist moves 8 cubies.
  - Colors union: this heuristic creates an abstraction of the original state. we tried four different abstractions: for \( n \in \{2,3,4,5\} \) the abstraction merges every combination of \( n \) colors into one color. For example, 2 colors abstraction creates \( \left( \frac{5}{2} \right) = 15 \) cubies with 5 different colors (paints one face with one of the other faces’ color) Then we run BFS on every abstracted state and return the maximum of the solution lengths found by every BFS run.
  - Number of misplaced tiles: sums the number of tiles that are not located in their place, for every face, and normalize the result by 6 (the number of faces). Because the minimal changes that can occur in the number of misplaced tiles is 3, dividing by 6 keeps the
  - Ignoring corners: this heuristic creates an abstraction of the original state. for each tile in each corner, replace it with the color of the tile in the center of the face (the ‘target’ color)
  - Null heuristic: returns 0 for every state

D.1 Heuristics Results
In order to test our simple heuristics, we created sets of rotations, each set of different length of rotation steps. We ran IDA* for each heuristic, and compared between the time taken to find a solution, and number of nodes expanded. We collected the most interesting results in tables, you can see the results in appendix C where the best heuristics in every depth (in running time and nodes expanded means) is marked in yellow.

Colors union heuristics were the slowest heuristics. union of 2, 3 and 4 colors took over 4 minutes and 5 colors union over a minute, only on 2 actions and for more actions they didn’t stop.

max manhattan heuristic, number of misplaced tiles heuristic, and sum manhattan heuristic were the best heuristics. max manhattan heuristic could solve cube with no more than 9 steps shuffling, number of misplaced tiles heuristic could solve cube with no more than 10 steps shuffling and sum manhattan heuristic could solve cube with no more than 12 steps shuffling.

figure 1 compares the running time of the best heuristics for different number of steps shuffling. as we can see, 3D manhattan sum and number of misplaced tiles are the fastest

\(^2\) http://www.sfu.ca/~jtmulhol/math302/puzzles-rc-cubology.html
heuristics. The reason for this result is that max Manhattan is not admissible in comparison to the two others. In addition, sum Manhattan heuristic dominates num of misplaced tiles heuristic since num of misplaced tiles is like sum Manhattan distance where all the distances are 1. This explains why sum Manhattan performs better than num of misplaced tiles.

![Figure 1](image.png)

**E. Pattern DBs**

Since heuristics alone were not fast enough and could take years to finish because of the high branching factor of the graph, we decided to keep in memory pattern DBs of the heuristic value of every abstract state. This will speed up the search by avoiding running BFS on the abstract state each time we want to calculate the heuristic. When we needed to calculate the heuristic value for a given state in IDA*, we generally did the following steps:

- Convert the state to the abstracted state according to the chosen abstraction (see section C.1.3)
- Search for the abstracted state in the DB
- Return the value saved in this location in the DB

The database was created by running BFS, when the root is a solved cube. At every step we retrieved all the successors of the current state. For every successor we checked if the abstracted state exists in the DB, if not, we pushed it to the queue and set the heuristic value to be the depth of the search. (Since it’s a graph and not a tree, this value is the distance of the node from the root). Meaning, the DBs contain the distance to a solution in the abstract space and as a result will always return a number on moves to a goal state which is lower than the real number of states needed assuring that the heuristic is admissible. Furthermore, since we run the BFS with step cost of 1 on only legal moves in the real state space we actually assure that the heuristic is also consistent, hence IDA* will return an optimal solution in the sense of the number of moves to a solved state.

### E.1) Pattern DBs - first attempt

At the beginning our BFS implementation was not efficient enough (see section E.1.a), and building the DB to depth 8 took over a week. Thus, we used partial DBs which we were able to collect up to depth 8. Now, when we needed the heuristic value, we did the following steps:

- Convert the state to the abstracted state according to the chosen abstraction (see section E.1.c)
- Run BFS on the abstracted state until a point where the state is found in the DB
- Return the value saved in the DB + the depth of the BFS.

#### E.1.a) synchronization

Running BFS in the normal way is not feasible to run in a reasonable time. Therefore, we tried to implement the BFS using multiprocessing. We created a process pool with as many processes as the number of CPU cores. The synchronization is made in the first level, that is, every child of the root gets a new process from the pool and a new queue, and runs BFS in parallel to the other processes. The branching factor is 12, therefore the tasks list contained 12 jobs. This method reduced the processing time but was still significantly long.

#### E.1.b) Saving the data

We tried a few methods to save the data in the most compact way:

1. Using sqlite3: we saved SQL tables that have two entries. The first entry contains a compact representation of the abstract state (see section E.1.c) as a key, and the second entry contains the solution length from the abstract state to the abstracted solved state, which is the heuristic value. This method wasn’t good enough because writing to the disk increased the overhead and the running time.

2. Using dictionary and saving to pickle: the keys of the dictionary were a compact representation of the abstract state (see section E.1.c), and the values were the heuristic value as explained above. This method wasn’t good enough because the representations of the keys took more than 3 bytes per key which for some abstractions congested the RAM. This is why it is not presented in the results.

In both methods, we converted the value of the heuristic to bytes. Since the maximum depth is in range of 0 to 26, we needed 5 Bits to save the depth. It is quite cumbersome to save Bits in Python, and therefore we saved it as one byte.
E.1.c) **Abstractions**

- One color: this abstraction considers only two colors 0 and 1. It sets the color of one of the faces to be 1, and the rest of the colors are set to be 0, resulting in 6 abstracted cubes and DBs. In order to represent a state (key) in each cube we converted the array representation to a 54 binary string indicating the values of each tile. We then treated this string as binary representation of an int and saved it as a 7 bytes integer number. When used, a maximum is taken over the returned values of each abstraction.
- Left edges: given a state, we chose 6 edges out of 12, concatenate the value of every edge in the current state and return the result as an int. Since we use 6 edges, and every edge has two values, the length of this number is 12, and every number is in the range of 0 to 6 (since there are 6 possible colors). This value was converted to bytes, and we needed 5 bytes to save this state key.
- Right edges: the same as Left edges is done here on the remaining 6 edges.
- Max over edges: uses the last two and takes max on their returned values.
- Corners: given a state concatenate the value of every tile of every corner in the current state and return the result as an int. Since there are 8 corners, and every corner has three values, the length of this number is 24, and every number is in the range of 0 to 6. This value was also converted to bytes, and we needed 10 bytes to save this number.
- One face: this abstraction considers only front face (F) and ignores the other faces. Given a state it concatenates the values of all the tiles in the front face except the middle tile which is never changed and returns the result as an int. This value was also converted to bytes, and we needed 4 bytes to save this number.

E.2) **Pattern DBs - second attempt**

Since the last attempt to collect the DBs took a long time, we improve the performance with two main changes. The first improvement was using a more efficient synchronization of the BFS when collecting the data (see section E.1.a). The second improvement was representing the state in the DB in a more compact way (see section E.1.b). These two improvements are described in article [3].

E.2.a) **synchronization**

After some research, we found out that the most efficient way to run BFS in parallel is using level synchronous BFS algorithm [2]. This time, comparing to our first attempt, the synchronization is made level-wise, that is, the processes are created for each level separately and we don’t proceed to the next level until the current level processes terminated. This approach has many advantages. First, we can control the number of processes at each level. By assigning less processes to the first levels, which are significantly smaller, we decrease the overhead created by too much locking. Second, we don’t have to wait until the end of the run to save the data, but we can save the data after the complete of each depth. Third, this synchronization appears to be faster than the last attempt. It finished collecting data up to level 7 in one hour, comparing to the last attempt which took about a week.

The level synchronous BFS algorithm is described in Algorithms 1, 2 (see appendix B). The algorithm manages two queues: current-level queue C and next-level queue N. Iteratively, the algorithm visits all the nodes in the current level (C) in parallel and collects the next-level set N. The number of processes are determined by a predefined threshold. If the size of the current queue C exceeds the threshold, we split C to two queues A, B, allocate a new process to run on A and keep running on the second half of the queue (B). Both of the processes have N as shared queue and they update it simultaneously using locking. That is, lines 9-11 are protected with a lock.

At the beginning of the next level iteration, C is populated with the values from N, and N is cleared for the new iteration. The iteration continues until there is no node in the next level. In short, this method visits all the nodes in each BFS level in parallel, with the parallel executions being synchronized at the end of each level iteration.

Note, the function find_idx maps states to location in the db (see section E.2.b)

E.2.b) **Saving the data**

We found that there is a way to effectively convert some of the abstractions to a continuous array using ranking function, that way we only need to keep the value of the heuristic without a key.

We used Myrvold & Ruskey’s ranking algorithm [4] to index the following abstraction:

- corners - calculating the amount of steps needed to get the corners to a solved state. There are 8 corners each of which has 3 orientations, hence the number of possible configuration is the number of arrangements times their possible orientations where the last cube’s orientation is determined by others \((8! \times 3^3)\). We represented each cube as a pair \((p, o)\) where p is the cube’s permutation and o is its orientation. Given a state we calculated the index to the DB array the following way: we use the ranking algorithm on the permutations \(p_0,p_7\) to calculate a permutation_index then we calculate the orientation_index by treating the orientations of the first 7 cubes in base 3 (the orientation is always a value between 0 and 2) using the
formula \( a_0 \times 3^6 + a_1 \times 3^5 + \ldots + a_6 \times 3^0 \). Eventually, 
\[
DB\ Array\ Index = 3^7 \times permutation\_index + orientation\_index.
\]

- 7 Edges - calculating the amount of steps needed to get 7 of the edges to a solved state and doing so to 2 sets of edges with a shared edge (taking the max on these values).

There are 7 corners each of which has 2 orientations, hence the number of possible configuration is the number of arrangements times their possible orientations \((12!/7! \times 2^7)\). We planned on doing the same as with the corners with an exception that the ranking algorithm has been modified to do \(k\) permutations of an \(n\) set (where \(k = 7\) and \(n = 12\)). Orientation is treated likewise but is in now in base 2. Unfortunately we didn’t get to finish this part.

E.3) **Pattern DB Results**

Important note before reading the results: every database was created on a different computer, some of the computers were stronger and could reach a higher depth on the same time. Therefore the comparison on higher depths might not reflect the real picture. On appendix A you can see the number of states found in every level of each DB. Note that in every DB, the last depth might not contain all the possible nodes exist in this depth since we might have stopped the run in the middle of the step.

Corners with sql and corners with pickle performed approximately the same on 2-6 actions (see fig.2)

Corners with sql couldn't find solution for more than 6 actions, while corners with pickle improved our search and found solution for 12 actions in 17 minutes, which made it the best heuristic we used.

E.3.1) **Pattern DB first attempt Results**

Edges left and edges right didn't improve much the search, and couldn't find solution for more than 6 actions shuffling. Corners heuristic performed better and could solve 12 actions shuffled cube and one face turned improved our search, and could find a solution for 12 actions shuffled cube in 53 minutes which is better than the manhattan sum heuristic which found solution in 10 hours. The timing results can be seen in figure 2. A possible reason that one face performed better than corners heuristic is that every step changes the corners position, therefore reaching the real goal state from the goal state in the corners abstraction must ruin the corners right position. But this is not the case in one face heuristic goal state.

E.3.2) **Pattern DB second attempt Results**

The creation of this db was significantly faster, but when we reached depth 8, the manger Queue data structure couldn't handle the amount of states needed for this level. Due to deadline constraints we didn’t have time to fix this, therefore we used the data saved until level 7 only. Nevertheless, this db performed better than the others since it is faster searching in the pkl file that is located in the RAM, rather than searching in the disk (as we have done in previous attempt). As you can see in the table (appendix C.4), on 12 steps shuffled cube, corners db using pickle file performs significantly the best. It finds a solution in 17 minutes and expands 958 nodes during the search, when the next best approach expands 4832 nodes.

III. **GUI**

We have used an interactive cube interface to visualize the cube and modified it to run the search.
A. GUI structure

The GUI contains the following functions:
Instructions and explanations window - upper left red text box.
A reset view button - rotates the entire cube back to a state where the F,U,R faces are presented.
Solve Cube Button - interactively solves the cube.
Randomization section - allows the user to decide on amount of steps and interactively randomizes the cube.
Search section - in which the user can choose the heuristic used in the IDA* algorithm and run the search.
An information window - updates the user upon different event that occur during the usage.

B. Usage

1. Each of the “UDFBRL” keys causes a clockwise turn to the relevant face. Combining the shift key causes a counter clockwise turn to the relevant button. The keyboard should be set to english when using manual keys.
2. Using the left mouse button the user can turn the whole cube.
3. When using either randomize, solve or search functions the manual actions are blocked until the function is finished.
4. After using the search method, a computer that has both cygwin and GCC will also show in the results of an optimal solution an one of the possible solutions for state configuration that is around 18 steps from the solved state(more than that might take significant amount of time).

IV. Analysis and Conclusions

among all approaches, the best approaches were sum manhattan heuristic, one face heuristic (using sql DB) and corners heuristic (using pickle DB). figure 3 shows the timing results of the best three approaches. they all managed to solve a 12 shuffled cube. corners heuristic using pickle DB is the best one. it is faster than one face heuristic because searching in the RAM is better than searching in the Disk, and it is faster than 3D manhattan sum since it takes O(1) to search in the DB. the number of expanded nodes is the smallest in corners heuristic using pickle (see appendix C.4) in addition, both corners heuristic using SQL DB and corners heuristic using pickle were consistent in the number of nodes expanded. in depth 2 they both expanded 4 nodes, in depth 4 they both expanded 8 nodes and in depth 6 they both expanded 118 nodes. this was a sanity check for us to confirm our constructions were correct.

In conclusion, solving Rubik’s Cube found to be most efficient using IDA* combined with pattern databases of the corners abstraction. future work improvements will include finish building the database for all heuristics and combining multiple heuristics.
V. REFERENCES

https://www.cube20.org/quad/


VI. APPENDIX

A. DBs built - depth with number of nodes

oneFace.db
depth 0 has 1 nodes in db
depth 1 has 8 nodes in db
depth 2 has 54 nodes in db
depth 3 has 321 nodes in db
depth 4 has 2016 nodes in db
depth 5 has 11354 nodes in db
depth 6 has 58343 nodes in db
depth 7 has 240907 nodes in db
depth 8 has 137106 nodes in db

corners.db
depth 0 has 1 nodes in db
depth 1 has 12 nodes in db
depth 2 has 60 nodes in db
depth 3 has 509 nodes in db
depth 4 has 3779 nodes in db
depth 5 has 24346 nodes in db
depth 6 has 130246 nodes in db
depth 7 has 568770 nodes in db
depth 8 has 984016 nodes in db

edges_L.db
depth 0 has 1 nodes in db
depth 1 has 8 nodes in db
depth 2 has 27 nodes in db
depth 3 has 192 nodes in db
depth 4 has 1384 nodes in db
depth 5 has 9499 nodes in db
depth 6 has 40400 nodes in db
depth 7 has 16380 nodes in db

edges_R.db
depth 0 has 1 nodes in db
depth 1 has 10 nodes in db
depth 2 has 48 nodes in db
depth 3 has 338 nodes in db
depth 4 has 2402 nodes in db
depth 5 has 16226 nodes in db
depth 6 has 87213 nodes in db
depth 7 has 20851 nodes in db

B. Parallel level BFS pseudo code

<table>
<thead>
<tr>
<th>Parallel level BFS (state):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ( db[\text{find}_\text{idx}(v)] = -1 ) for every ( v )</td>
</tr>
<tr>
<td>2. ( db[\text{find}_\text{idx}(\text{state})] = 0 )</td>
</tr>
<tr>
<td>3. ( \text{level} = 0 )</td>
</tr>
<tr>
<td>4. ( C = {\text{state}} )</td>
</tr>
<tr>
<td>5. while not ( C.\text{isEmpty} )</td>
</tr>
<tr>
<td>6. ( N = { } )</td>
</tr>
<tr>
<td>7. ( \text{Process_level(len_orig, C, N, level)} )</td>
</tr>
<tr>
<td>8. ( C = N )</td>
</tr>
<tr>
<td>9. ( \text{level}++ = 1 )</td>
</tr>
</tbody>
</table>

Algorithm 1

<table>
<thead>
<tr>
<th>Process_level (len_orig, C, N, level):</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. if (</td>
</tr>
<tr>
<td>2. ( A, B = \text{split}(C) )</td>
</tr>
<tr>
<td>3. ( \text{spawn} \text{Process_level(len_orig, A, N, level)} )</td>
</tr>
<tr>
<td>4. ( \text{Process_level(len_orig, B, N, level)} )</td>
</tr>
<tr>
<td>5. ( \text{sync} )</td>
</tr>
<tr>
<td>6. else:</td>
</tr>
<tr>
<td>1. while not ( C.\text{isEmpty} )</td>
</tr>
<tr>
<td>2. ( \text{cur} = C.\text{pop} )</td>
</tr>
<tr>
<td>3. for ( s ) in ( \text{cur._successors} ):</td>
</tr>
<tr>
<td>4. if ( db[\text{find}_\text{idx}(s)] &lt; 0 ):</td>
</tr>
<tr>
<td>5. ( db[\text{find}_\text{idx}(s)]++ = 1 )</td>
</tr>
<tr>
<td>6. ( N.\text{push}(s) )</td>
</tr>
<tr>
<td>7. ( \text{Algorithm_2} )</td>
</tr>
</tbody>
</table>
C. Heuristics Comparison table

C.1) 6 actions:

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Time taken (minutes)</th>
<th>Nodes expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D manhattan sum</td>
<td>0.003</td>
<td>24</td>
</tr>
<tr>
<td>3D manhattan max</td>
<td>0.027</td>
<td>163</td>
</tr>
<tr>
<td>Number of misplaced tiles</td>
<td>0.012</td>
<td>96</td>
</tr>
<tr>
<td>DB: One face</td>
<td>0.114</td>
<td>18</td>
</tr>
<tr>
<td>DB: Corners - SOL</td>
<td>2.2</td>
<td>118</td>
</tr>
<tr>
<td>DB: left edges</td>
<td>13.4</td>
<td>188</td>
</tr>
<tr>
<td>DB: Corners – pickle</td>
<td>2.13</td>
<td>118</td>
</tr>
</tbody>
</table>

C.2) 9 actions:

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Time taken (minutes)</th>
<th>Nodes expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D manhattan sum</td>
<td>0.02</td>
<td>192</td>
</tr>
<tr>
<td>3D manhattan max</td>
<td>2.63</td>
<td>15443</td>
</tr>
<tr>
<td>Number of misplaced tiles</td>
<td>0.7</td>
<td>5375</td>
</tr>
<tr>
<td>DB: One face</td>
<td>0.1</td>
<td>18</td>
</tr>
<tr>
<td>DB: Corners – pickle</td>
<td>1.6</td>
<td>94</td>
</tr>
</tbody>
</table>

C.3) 10 moves:

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Time taken (minutes)</th>
<th>Nodes expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D manhattan sum</td>
<td>0.7</td>
<td>6484</td>
</tr>
<tr>
<td>Number of misplaced tiles</td>
<td>15.5</td>
<td>118948</td>
</tr>
<tr>
<td>DB: One face</td>
<td>0.1</td>
<td>20</td>
</tr>
<tr>
<td>DB: Corners – pickle</td>
<td>1.5</td>
<td>96</td>
</tr>
</tbody>
</table>

C.4) 12 moves:

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Time taken (minutes)</th>
<th>Nodes expanded</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D manhattan sum</td>
<td>639</td>
<td>3679126</td>
</tr>
<tr>
<td>DB: One face</td>
<td>53</td>
<td>4832</td>
</tr>
<tr>
<td>DB: Corners – pickle</td>
<td>17.1</td>
<td>958</td>
</tr>
</tbody>
</table>