Evolving a Rubik's Cube Solver

2009 GECCO Rubik's Cube Competition

Ignacio Bona and Luis Martí

Group of Applied Artificial Intelligence
Universidad Carlos III de Madrid
Av. de la Universidad Carlos III, 22. Colmenarejo 28270 Madrid Spain
igbopie@gmail.com, lmarti@inf.uc3m.es

1 Introduction

Genetic programming (GP) (Koza, 1992) has been a latent matter in the evolutionary area. One of the reasons that have hindered the study of many interesting problems is the large computational requirement of GP. The use of parallel, grid or cloud computing architectures have been proposed as viable platforms for dealing with such large problems.

Parabon Computation Inc. has developed Origin, which is a port of the ECJ framework to their Frontier Grid Platform. In this context, Parabon proposed a contest to take place at GECCO 2009: Evolving a problem capable of solving a scrambled Rubik's cube. This competition was intended to test and demonstrate the capabilities of Origin when performing large-scale evolutionary computation.

The present document is a brief report of the work of the carried out by the authors when taking part of the aforementioned competition. Our main guideline when dealing with this problem was to provide a purely genetic solution, without any a priori knowledge bias and useful as a research tool for assessing different approaches. The results here described are the outcome of addressing the problem from different angles and of the progressive comprehension of the peculiarities of the issue being addressed.

In the following section we describe the problem being dealt with. Subsequently, we detail the language developed to represent the programs to be evolved. After that, we proceed with another important part of the work, the computation of the fitness of different programs. In section 4 we describe some of the software issues that were tackled in the experiments. Finally, some concluding remarks and lines of future work are presented.

2 Solving the Rubik's Cube Using Genetic Programming

The goal of this work is to create a genetic programming algorithm that evolves a program that can solve an arbitrarily scrambled cube in the fewest number of moves, as measured by the face-turn metric.

Everyone, at least once in theirs life, have tried to solve a Rubik's cube and they can tell us It is not an easy task. The complexity of the cubes comes with the large number of states it has. For each state you have up to twelve possibilities of moves (just counting face moves). The number of states grows exponentially in every move. Even math find difficult to find an optimal solution (fewest steps) for any state of the cube. That is why we are going to use genetic programming to create a Rubik's cube solver.

Genetic programming, as some others AI techniques, has the ability to find unusual solutions to a given problem. These techniques are applied when a problem domain is not entirely known. Taking advantage of the power of randomness and variety, you will only have to tell the program how well or bad is he doing, and GP will find an answer (or get close to it).

With GP we expect to find an unusual solution to a Rubik's cube, we expect to find an optimal solver.

3 Language

The language used to represent the programs to be evolved as devised with the intention to make the individuals to be capable of steps taken by a human when solving the problem. A human solver detects patterns of cube configurations that are abstract with regard to particular colors and cube faces. Then a certain action is carried out to "improve" the state of the cube and again tries to recognize a known pattern.

In order to be able to provide an abstract representation of these patterns we have unified every point of view to a single representation for a single cube state. Also, we do not have anymore concrete colors, we have six abstract colors tags where can be assigned to during the matching process to any concrete color in order to make true a certain condition. Cube faces also have a similar treatment. Summarizing, our language is capable of expressing abstract cube patterns and, when one of those patterns is detected a certain action is triggered. (Kunkle & Cooperman, 2007)

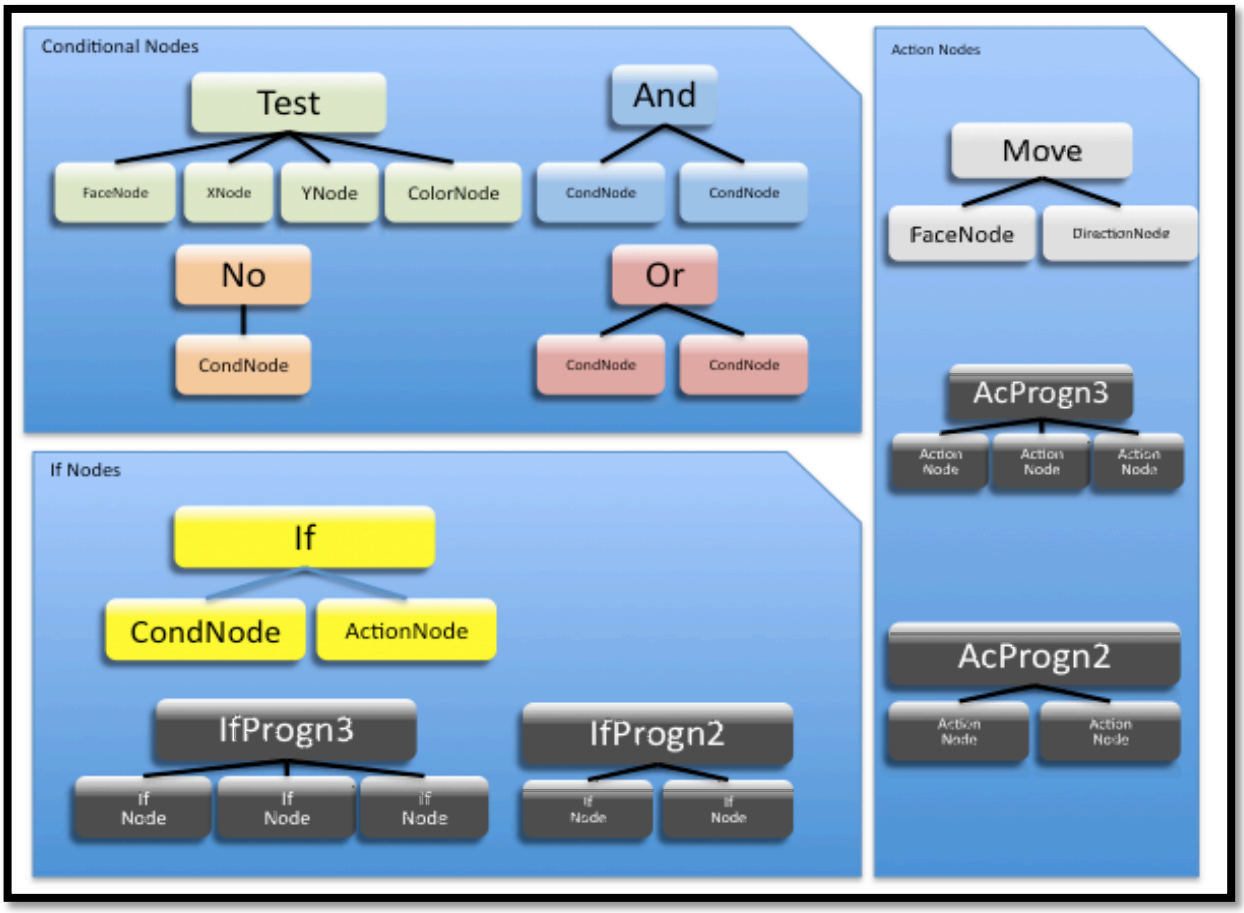
Table 3.1. Description of the terminal nodes.

Node(s) Name(s)	Type	Description
FaceUp	FaceNode	Upper face of cube (relative to the current one).
FaceLeft	FaceNode	Left face of cube (relative to the current one).
FaceFront	FaceNode	Front face of cube (relative to the current one).
FaceDown	FaceNode	Lower face of cube (relative to the current one).
FaceRight	FaceNode	Right face of cube (relative to the current one).
FaceBack	FaceNode	Back face of cube (relative to the current one).
X0, X1, X2	XNode	Position x0, x1 and x2 in a given face (see figure 1.1).
Y0, Y1, Y2	YNode	Position x0, x1 and x2 in a given face (see figure 1.1).
Color1, Color2,…, Color6	ColorNode	One of the six colors of a cube.
Clockwise	DirectionNode	Clockwise direction.
CounterClockwise	DirectionNode	Counterclockwise direction.

Table 3.2 Descriptio of non-terminal nodes.

Node Name	Node Type	Arity	Description
Test	CondNode	4	Test if the specified cell match with the color specified
And	CondNode	2	And Condition
Or	CondNode	2	Or Condition
Not	CondNode	1	Not Condition
Move	ActionNode	2	Move face in a direction
AcProgn3	ActionNode	3	Execute 3 actions.
AcProgn2	ActionNode	2	Execute 2 actions.
If	IfNode	2	Tries to make true the condition specified by rotating the cube.
IfProgn3	IfNode	3	Execute 3 If statements
IfProgn2	IfNode	2	Execute 2 If statements

As required for ECJ, the Java class that represents our individualsis ec.gp.GPIndividual, which has a tree to keep the code. Therefore, our language will be expressed in a tree form. The description of the terminal and non-terminal nodes is summarized on tables 3.1 and 3.2.



4 Fitness

One of the main theoretical issues regarding the Rubik's cube is how many steps are required to find a solution. It has been shown that twenty-six moves are enough for solving any given cube (Kunkle & Cooperman, 2007). However, the computation of how distant is a given cube from being solved is still very expensive and therefore not suitable to be used as-is for fitness assignment. As a result, we had to find alternatives ways to measure the proximity a cube is from its solution. One natural way is to count the number of cells with the same color on the same face. Consequently, if a cube is solved, it will have the highest score. This scoring method will be called Entropy Score. Although this score does not give exact numbers of steps, it is very easy to calculate and gives some general information about the cube state.

The dexterity of a program solving a particular cube is not enough for assign it. It is also needed to know how many cubes a program can solve, and, specifically, how many cubes for each level of difficulty a program can solve (we will discuss later on our classification cubes by their difficulties). As there are different difficulty levels, we need to reward solved higher-level cubes with regard to lower-level ones. Also, this reward must be exponential as solving an n+1-level cube is more difficult than solving a number of n-level ones. Taking this reflection into account, we have devised the following formula to score solved cubes:

$$Score = \sum_{i=0}^{MaxDif} Solved_i * Z^i,$$

here Z is parameter that can be set by the user. After some trial-and-error experiments we choose Z=2.

Finally, individual size and length of the solution is something we have to control in terms of efficiency and quality.

The final step is to decide whether or not an individual is better than another. An individual is better than another if it has a higher *solved score.* When *solved score* is the same then, the individual with the higher *entropy score* wins. If *solved score* and *entropy score* is the same in both individuals, then we will choose the one with the shorter solution. In the case that they are the same, we will choose one randomly if they didn't solved all the cubes. If they did, we will choose the shorter individual.

5 Evaluation of Programs

Evaluating an evolved program must encompass a diverse set of cubes with different levels of difficulty. Before describing the evaluation process it is required to explain how this set of cubes is organized. We will consider that a cube belong to a level *n* if there are *n* steps from a solved cube to reach its current state. For example, a level one cube is a cube that only needs a move to be solved. Also, we have a constraint implying that level *n* cubes must have lower entropy than a level *n*+1 cube and so.

Now, we have to supply a training sample set of arbitrarily scrambled cubes to the evaluation process. In order to speed-up the execution, we will pre-generate every cube that will be used for testing, and keep them in different difficulty bins. Each bin will have a maximum storage size and no repeated cubes are allowed. Because of this, some levels will not have enough diversity to fill all the space. In this case, all existing cubes from this difficulty will be stored.

The best way we have found to evaluate individuals is to try with the whole training sample set. This will have the following benefits:

- An individual does not have to be reevaluated: training set is the same for all the generations;
- ensure that the individual solves every cube in the set: even if an individual solve a cube at any point of the run, he can forget it in the next generation;
- choosing the best individual is easier: as every individual is tested with the same set, no one is favored, and;
- lower computational demands.

Last, we will test all the cubes for each difficulty. Each test will consist in a loop evaluating program's tree with the specified cube whilst the cube is not solved and the solver have performed any modification in the cube and have not reached the maximum iterations. Maximum iterations is parameter that can be set by the user. After some trial-and-error experiments we choose *40* iterations.

6 Execution and Test

There is a SVN repository with an Eclipse project, which can be downloaded with the most up-to-date source code. You are free to download and see the code. You have the access codes in the next page. Alternatively an Ant build script is also provided. The target to be executed is "performance-GPKoza".

To test the result of the evolution, there is a class named *ec.app.rubik.TestIndividual,* which implements *com.parabon.rubik.Solver* interface. The default constructor (*new TestIndividual()*) will provide the code of the best individual we could ever have till date. This class is also executable, which in that case will perform a test and print out the statistics of the best individual.

7 Final Remarks

In this work we have addressed the problem of evolving a genetic programming solver of the Rubik's cube. Mainly because of the tight schedule and the multiple software issues raised during the elaboration of the program the results here presented are of a preliminary nature.

An obvious conclusion is that longer executions are required. In our experiments we couldn't proceed beyond relatively few iterations and this is a clear drawback that can be extended to many GP applications.

There are some lines of improvement that deserve to be studied in the future. One important issue is the improvement of the fitness assignment. Probably the introduction of multi-objective optimization features will provide a great assistance, as they will make the algorithm to be able to take into account different relatively simple cube improvement indicators. These indicators would not be adequate by their own but combined together might provide a sizable improvement.

Similarly, a better language would yield smaller, more flexible and easier to analyze programs, but this issue is a rather controversial one as some modifications might render the algorithm unviable.

As this is a work in progress we have set up some Internet resource to publish the results as they appear, in particular:

- web page:
  - http://www.giaa.inf.uc3m.es/miembros/lmarti/rubik
- subversion repository with the most up-to-date source code:
  - https://svn.xp-dev.com/svn/lmarti_gp/
  - user: uc3m-guest; password: guest12.

Acknowledgement

This work was supported in part by projects CICYT TIN2008-06742-C02-02/TSI, CICYT TEC2008-06732-C02-02/TEC, SINPROB, CAM MADRINET S-0505/TIC/0255 and DPS2008-07029-C02-02.

Bibliography

Koza, J. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Boston: MIT Press.

Kunkle, D., & Cooperman, G. (2007). Twenty-six moves suffice for Rubik's cube. *ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation* (pp. 235-242). Waterloo, Ontario, Canada: ACM.