ACV
Course Work - Implementation of an AI Component for the Number Sequence Problem

Vicente Bosch Campos †
viboscam@posgrado.upv.es

May 3, 2011

Contents

List of Figures

1 Introduction

1.1 Context

As part of the ACV course project we will implement a genetic algorithm in order to resolve the Numbers problem of the Numbers and Letters game. The goal is to obtain a combination of arithmetic operations over a set of 6 integer numbers in order to obtain as a result of those operations another integer number.

The rules are as follows:

- The six selected numbers must be chosen randomly from: 1,2,3,4,5,6,7,8,9,10,25,50,75 & 100 taking into account that the probability to choose one of the numbers between 1 and 10 must be double than choosing the rest of the numbers.
- The target number must be chosen randomly from the range 101..999.
- Operations allowed to combine numbers are: sum, subtraction, multiplication and division. (division can only be applied if it is exact)
- Operations must be performed with the selected numbers or with the numbers obtained by operating on them.
- Results of operations must always be integer and positive.
- If the exact number can not be obtained, the best result will be the one closest to it.

1.2 Scope

In order to implement this AI component to resolve the number problem we will develop components in order to cover the following user cases:

- Evaluate a solution provided by a user against a target value.
- Obtain a best solution through running the genetic algorithm resolver.
- Programmer will be allowed to indicate the minimum and maximum number of individuals in the population
- The genetic algorithm will be time constrained by the time given for the test to be performed by the human players.

Experimental evaluation will be performed by testing the specific problems indicated in the project description.

2 Development

For our implementation of the sequence number problem we have implemented the algorithm in Ruby (v1.9.2) and the following libraries where used:

RMagick (2.13.1): RMagick is a wrapper library for the Image Magick software. RMagick has been used in our implementation to load the generated operation graphs.

Trollop (1.16.2): Trollop was used in order to parse and generate validators of the command line options for the num_generator command created.

Awesome Print (0.3.1): Used to perform pre-formatted human readable print outs of the objects.

rgl (0.4.0): Was used in order to generate the operation graph drawing.

Benchmark: Standard library object was used to measure the time taken to perform operations.

Timeout: Standard library object was used in order to enforce the time constraint on the execution of the genetic algorithm.

2.1 Program Structure

Next we will detail the main software components and characteristics for the sequence number resolver. The implementation can be divided in the following main software components:

Application logger: Coded in *./lib/application_logger.rb* is a global logger object implemented through the singleton and logger classes to allow the creation of logs easily from any object to a centralized file.

Number Problem Generator: Present in *./lib/number_problem_generator.rb* is the component in charge of loading a problem from a file or generating one randomly.

Operation Graph: Class coded in *./lib/operation_graph.rb*. It represents a population individual for our sequence number problem. The operation graph is stored internally as an actual graph of nodes that allow breed and mutation operations to be evaluated in constant time. Each node contains:

- A list of the numbers consumed beneath it and the current node type.
- Links to the root, its father and its left and right children.
- List of all of the numbers consumed and unconsumed in the whole of the operation graph.

The operation graph object allows the following methods:

- from_string - Allows an operation graph to be constructed from a string containing the operations to perform in polish notation.
- initialize - Performs a random initialization of the operation graph by selecting a random set of the possible values and generating a tree by subdividing the selected numbers into different groups at each node and combining them with a randomly selected operator.
- level= - Sets the level of the node inside the hierarchy (root node has level 0) and sets recursively the level on its child nodes.
- unconsumedNumbers - Returns the list of unused numbers in the whole of the operation graph.
- operatorNodes - Returns the full list of nodes that act as operation nodes inside the graph.
- valueNodes - Returns the full list of nodes that act as value nodes inside the graph.
- value - Returns the value of a node.
- value= - Sets the value of a node and updates the consumedNumbers list recursively in all of its parent nodes.
- result - Calculates the result of the operation graph
- score - Returns the distance ,as a positive integer value, between the current result of the operation graph and the expected target value.
- clone - Performs a deep copy of the operation graph.
- transform2Child - Transforms an specific node into value type node.
- transform2Op - Transforms an specific node into operator type node.
- changeConsumedNumbers - Modifies the consumedNumbers list of the node and recursively updates the parent nodes.
- addConsumedNumber - Adds a new value to the consumedNumbers list of the node and recursively updates the parent nodes.
- deleteConsumedNumbers - Deletes the consumedNumbers list of the node and recursively updates the parent nodes.
- draw - Using the rgl library it draws a visual representation of the operation graph and shows it to the user.
- mutate! - Selects at random a method terminated in *Mutation, by means of the reflection methods in ruby, and executes it over the current operation graph.
- cloneMutant - Returns a cloned an operation graph on which mutate! has been executed on.

- operatorSwitchMutation - Selects one of the operator nodes at random and changes the operation performed in it.
- numberSwitchMutation - Selects two value nodes in the operation graph and swaps them.
- unusedNumberSwitchMutation - Changes a value used in a node of the operation graph by an unused number.
- unusedNumberMutation - Selects an unused number from the original set and combines it with a current value node to create a subtree composed of a new operation node, the existing value node and a new value node for the unconsumed number hanging in the operation graph in the same position as the existing value node prior to the mutation.
- deleteOperationMutation - Deletes all nodes below an operation node updating the value node list,operator node list and the unconsumed numbers list.
- deleteSubTreeOperators - Deletes all operator nodes in a sub tree beneath a selected node. Used by the deleteOperationMutation.
- breed - Combines two operation graphs into a new child. If the base graphs do not share any consumed number it just randomly selects two cross points (one at each graph) and combines them. If the base graphs do share consumed numbers it then compiles a list of pairs of base nodes that can be combined together and selects one of those pairs to combine them so that the child graph will be compliant with the base rules. If there exists no possible combination the operation returns a cloned mutant of the mother.
- setChildren - Sets the children nodes of an specific node.
- getTypeNodes - Returns a list with all of the nodes of an specific type in a subtree beneath a node.

Genetic Population: Main object of the genetic algorithm implementation present in *./lib/genetic_population.rb*. The genetic algorithm implementation has the following base methods:

- initialize - Creates an initial population of a random number of individuals between the indicated minimum and maximum population.
- reducePopulation - This method reduces the population until it reaches the upper bound indicated by the user. The reduction is performed by sorting the children by score value and eliminating the worst individuals until the limit is reached.
- updateAverage - Calculates the average score of the current population.
- evaluatePopulation - Reviews all current individuals, it sorts them by score value, stores the best individual and eliminates any individuals with a negative score.
- recoverPopulation - If the current population is above the minimal population people indicated by the user it recovers it by generating randomly new individuals until the limit is reached.
- draw - Draws the operation graph of all current individuals.
- scores - Returns a histogram of the number of operation graphs per score value.
- draw_best_history - Plots the score evolution of the best result in the population.
- draw_avg_history - Plots the score evolution of the average result of the population.
- draw_population_history - Plots the population count through history.
- draw_scores - Draws the score histogram of the current population.
- actOnIndividual - Performs a random action on an individual. Action can be:
  - Mutate
  - Clone
  - Clone a mutant
  - Breed
- selectIndividuals - Returns a list of the current individuals that have a score between the current best and the current best plus a random % from 20 to 50 of the target value.

- evolve - Is the main evolution loop. The evolve method has been implemented so it can be time constrained by the user. If the algorithm obtains an exact solution it will stop immediately and provide it in case it does not it will continue to iterate until the indicated computation time is over at which point it will return the best solution obtained.

2.2 Command Output

The generated code will present us with the following outputs:

Solution reached before time constraint is reached: In this case the program will output the following messages:

```
I, [2011-04-20T11:44:52.994423 #4635]  INFO -- : LOGGER CREATED
Cycles performed: 19
Target value: 718
Smallest from target: 0
```

Solution not reached before time constraint is reached: For the other case the program will present the messages:

```
I, [2011-04-20T12:10:29.158400 #5150]  INFO -- : LOGGER CREATED
I, [2011-04-20T12:10:34.168199 #5150]  INFO -- : Termination due to time constraint
Cycles performed: 131
Target value: 919
Smallest from target: 1
```

The component can present us with the operation graph of the best solution and a graph representing the best solution score's evolution through time.

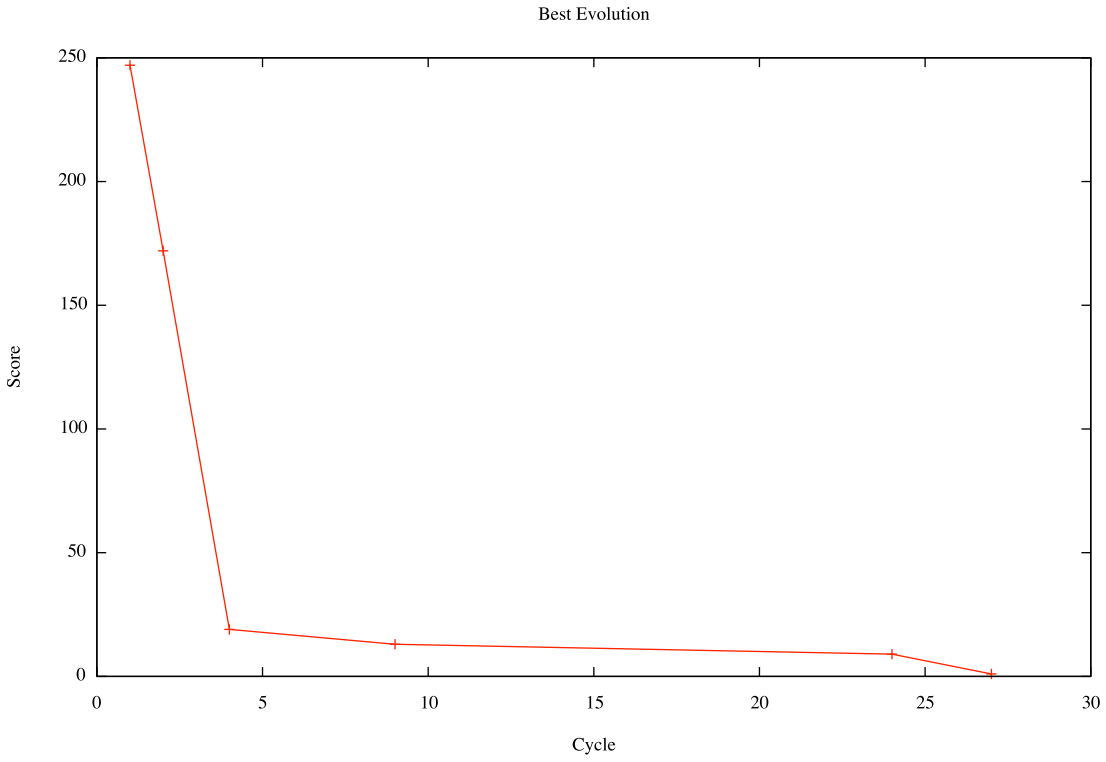

Figure 1: Plot shows the best score, as a distance between the best individuals result and the target distance, as a function of the evolution cycles performed.

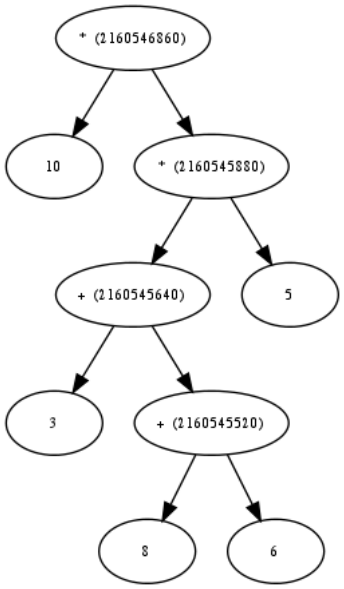

Figure 2: Diagram shows a visualization of an operation graph where besides the operators the object id has been included so that the nodes are unique.

3 Experimental Results

We have evaluated the performance of our implementation of the genetic algorithm resolver we will experiment over a base set of problems:

Table 1: Provided experiments set

Provided Experiments Set

Experiment Id	Initial Values						Target Value
F1	4	10	7	9	2	25	232
F2	10	2	9	5	7	100	298
F3	2	75	9	6	100	8	474
F4	3	10	7	6	75	10	381
F5	7	3	50	25	6	100	741
F6	2	4	75	4	50	2	502
F7	8	25	10	2	9	4	549
F8	7	10	3	2	1	25	223
F9	7	4	100	5	9	10	156
F10	2	10	7	50	100	10	259
D1	9	6	75	7	2	50	264
D2	5	3	9	50	4	5	458
D3	1	10	3	3	75	4	322
D4	100	2	6	4	25	6	305
D5	2	1	10	10	7	100	274
D6	1	100	4	50	3	4	661
D7	10	8	75	2	100	4	431
D8	100	75	1	8	3	75	511
D9	100	3	2	25	3	9	407
D10	5	50	1	8	75	8	713

We will perform two sets of experimentation:

- Unconstrained: We will allow the random tree builder to use as many of the selected numbers as

it requires.

- Constrained: We will only allow the tree builder to select one of the selected numbers, hence it will a very basic solution.

For each problem configuration described in the table above we will perform 50 experiments with a time constraint of 10 seconds and record for each:

- Number of times exact solution is reached
- Average number of cycles for exact resolution
- Number of times time constrained solution is given
- Average number of cycles for time constrained resolution
- Average distance from target Average number of cycles for exact resolution
- Best distance from target Average number of cycles for exact resolution

3.1 Unconstrained Initialization

The results for the unconstrained generation are as follows:

Table 2: Unconstrained Tree Generation Experiments Statistics

Experiments Statistics

Experiment Id	Exact Resolution		Time Constrained Resolution			
	Count	Avg. Cycles	Count	Avg. Cycles	Avg. Score	Best Score
F1	50.0	13.8	0.0	0.0	0.0	-1
F2	28.0	59.678	22.0	221.818	1.545	1
F3	25.0	83.28	25.0	194.96	1.08	1
F4	23.0	50.043	27.0	206.629	1.296	1
F5	38.0	58.657	12.0	192.916	1.0	1
F6	39.0	41.487	11.0	191.636	2.0	2
F7	14.0	71.857	36.0	202.361	1.0	1
F8	48.0	49.875	2.0	224.5	1.0	1
F9	50.0	13.76	0.0	0.0	0.0	-1
F10	40.0	53.575	10.0	226.2	1.3	1
D1	43.0	43.674	7.0	231.142	1.142	1
D2	32.0	32.031	18.0	235.888	1.0	1
D3	43.0	49.441	7.0	206.285	4.571	1
D4	7.0	90.857	43.0	228.651	1.0	1
D5	2.0	83.0	48.0	231.979	1.375	1
D6	8.0	94.25	42.0	230.452	2.166	1
D7	24.0	99.875	26.0	227.192	1.0	1
D8	23.0	71.826	27.0	225.777	3.925	2
D9	21.0	75.095	29.0	232.379	1.0	1
D10	35.0	84.914	15.0	229.866	1.0	1

3.2 Constrained Initialization

Table 3: Constrained Tree Generation Experiments Statistics

Experiments Statistics

Experiment Id	Exact Resolution		Time Constrained Resolution			
	Count	Avg. Cycles	Count	Avg. Cycles	Avg. Score	Best Score
F1	38.0	79.578	12.0	289.916	1.0	1
F2	24.0	82.833	26.0	308.884	1.692	1
F3	38.0	79.263	12.0	283.25	2.5	1
F4	10.0	127.1	40.0	310.6	1.95	1
F5	9.0	74.888	41.0	286.609	1.975	1
F6	14.0	68.714	36.0	313.583	2.0	2
F7	23.0	140.130	27.0	306.148	1.0	1
F8	18.0	77.222	32.0	299.406	1.0	1
F9	50.0	27.18	0.0	0.0	0.0	-1
F10	36.0	57.0	14.0	314.0714	1.0	1
D1	43.0	70.861	7.0	281.285	1.571	1
D2	8.0	21.5	42.0	283.547	1.5	1
D3	18.0	82.055	32.0	306.0	2.468	1
D4	3.0	176.0	47.0	311.425	1.085	1
D5	2.0	170.5	48.0	317.666	2.166	1
D6	2.0	42.5	48.0	310.875	1.166	1
D7	9.0	187.777	41.0	288.439	1.731	1
D8	15.0	157.533	35.0	298.542	4.028	1
D9	6.0	91.0	44.0	315.091	1.023	1
D10	14.0	110.214	36.0	305.555	1.0	1

3.3 Conclusions

As we have seen in the experimentation above:

- The ratio of exact solutions is significantly worse for the problems tagged as hard.
- The unconstrained initialization of the operation graphs yield better results.
- In both constrained and unconstrained initializations the solutions provided in 10 seconds are very close from the target either an exact solution is reached or at an average distance of 2 from the target value.