EVCO ASSESSMENT - 2013/2014

Candidate Y1447424

Table of Contents

1. Introduction

Our goal was to design an evolutionary algorithm to create a player for 5T Morpion Solitaire. We chose to use a Koza-style Genetic Programming algorithm using a Java implementation of the game and the ECJ[1] system for the evolutionary algorithmic part. The Genetic Programming approach for this game has been used successfully by Hugues Juille[2] in his thesis, where a special technique called SAGE(Self-Adaptive Greedy Estimate) is presented. We did not use this technique but experimented with another GP implementation trying different terminals and parameter settings. Our program was based at existing implementations of other GP problems, especially the Koza Lawn Mower problem[3] as it is implemented by the ECJ group. Our implementation uses mainly Ephemeral Random Constants as leafs of the GP tree, which represent locations of the nodes of the game grid. We defined a proximity measure that chooses the closest to these locations moves at each configuration of the game. This way our GP algorithm works with numerical ERCs corresponding to available moves in the grid. This correspondence, however, depends on the configuration of the grid when the particular node is evaluated. So, an ERC at the leftmost child of the tree, for example, corresponds to the move closest to that point at the start of the game, whereas the same ERC found at another position of the tree corresponds to a completely different move depending on the configuration of the grid at the time of the node evaluation. This is a highly random search. In the case of sole use of ERCs the final phenotype is simply a sequence of vectors whose interpretation depends on their position in the sequence. Our GP algorithm actually tries random permutations of the vectors, or, more probably groups of vectors in the sequence and keeps the most successful ones. Even this naïve implementation does quite well compared to a human. As we will see later, adding an extra terminal, not one with a specific rationale, improves the efficiency of the search.

We designed an experiment to compare the performance of two different function sets, one including only ERCs and the other another terminal as well. We followed an approach similar to that of Simon and Poulding[4] to perform a fair comparison, tuning independently the two algorithms, but we investigated the effect of a small set of parameters. We used the R language[9] for our statistical analyses, due to its availability. The results are quite interesting as we show later.

2. The solution space and the representation

As Hugues[2] states, the game is “a combinatorial optimization problem”. Each solution consists of a sequence of moves reaching at a configuration where no other moves are available. Each move is described uniquely by 4 integers, the x and y coordinates of its start and end points. In the Morpion Solitaire website[5] there are many resources providing information about the history of the game and its progress. According to this site, research has shown that the number of possible moves is finite, although the upper limit is still under search. The minimum upper limit proved is 665 moves, by Flammenkamp[6]. The maximum published score is 178, by Christopher D. Rosin in August 2011, although unverified scores of even more than 200 moves have been reported.

The problem can be viewed as a CSP (constrained satisfaction problem) optimization problem, where the variables are the ordered moves, their domain is the set of all possible 5-length vectors in the 4 directions of the grid within the limits given and the constraints are the rules of the game. We can compare it with the well known n-queens problem, described extensively in the bibliography,

where each partial assignment (sequence of moves in our case, even the starting empty sequence) restricts the domain of the following moves. However, the n-queens problem is a satisfaction problem with a fixed number of variables, while, our problem is mainly an optimization one and the solutions have not a fixed length. Since we know that there is an upper limit to the total number of moves, we also know that the solution space is finite too. However, it is so big that an exhaustive search seems impossible. It includes all the possible sequences of moves which end up at every feasible score of the game.

Our initial thoughts to deal with the problem were concentrated at the use of other kinds of evolutionary algorithms using a vector representation, like GA, where the solutions would be represented by vectors of 4-digit vectors (the moves). An executable of the game was provided that accepted a maximum of 500 moves and ignored any non valid moves. The idea was that we could make random vectors of length 5 in any of the 4 possible directions, with start and end points in an area around the Greek Cross, without caring if they corresponded to valid moves. We would make individuals of the maximum number of vectors allowed as input(500) and leave the GA to find a combination of valid moves that would give a good result. This primitive approach looked tempting at first, since we could have a solution without implementing the game. The problem, however, was that there is a huge number of possible combinations of moves (5-length vectors in 4 directions) even if we are restricted in the central area of the grid and most of the possible moves would not be valid. Furthermore, the search space in this game is dynamic, which means that the possible movements in every step depend on the previous ones and we would waste a lot of the searching potential of the evolutionary algorithm only to discover valid moves. Moreover, an implementation of this kind would not even exhaust all the possible moves, leading to partial solutions. So it looked impossible for the program to discover a good pattern this way and the idea of using totally “blind” vectors was abandoned.

We chose to implement the game rules and have a function that would return all the valid moves at each configuration of the grid. Since we had this information we could leave behind the former idea of using totally random vectors and deal only with the valid moves. We decided to use a GP tree representation, since it allowed us more flexibility than any other EA. Indeed, we could work easily both with numerical data of different kinds, using Koza[7] Ephemeral Random Constants (ERC) and other functions as well, like choosing the move in the more dense neighbourhood or the move which creates the most available moves at the next step. Of course we could do the same using a GA with a vector representation and coding some vectors to perform specific tasks, but that would require from us to set specific rules for the operations used (mutation and crossover) depending on the kind of each gene. GP on the other hand, is especially designed to evolve programs and it gave us more freedom to experiment with different kind of nodes.

3. The implementation

First. we implemented a 5T Morpion Solitaire in java as an independent program that was used initially for testing purposes and to provide a visual representation of our solutions later. We used the core code of this program to create a GPProblem in ECJ. In our implementation, the whole map of the game is stored in a 3 dimensional array, the *map***.** The first two dimensions determine the position of a node (x and y). The third dimension is an integer array of length 6, holding information about the following features of each node, in the order they are given: if the node has a cross, a horizontal line, a positive-slope diagonal line, a vertical line, a negative-slope diagonal line. Each line direction is associated with an integer value from 1 to 4 as shown in Fig.1 below. Finally, in the last position there is a number indicating in which move this node acquired its cross, if it has one.

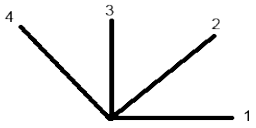

Figure 1

Each move is represented as an integer array containing 7 elements. The first four determine the start and end positions of the line added. The next two is the position of the cross point added and the last one is the direction of the move as described before. Each move has a single cross point, although the same point may correspond to different moves. We will consider this cross point to be the location of our move, that is, each move is considered to have a single node location, the location of its cross. When we refer henceforth to the position of a move, we mean the position of its cross. We will use functions and terminals using this kind of proximity between moves as our GP nodes.

We will describe shortly the main variables and functions of the program. A more detailed description is given in the comments of the source code given at the appendix B. All the code used is provided in the accompanying folder. The variable *validMoves* holds each time all the possible valid moves, whereas the variable *allMoves* holds all the moves added from the beginning of the game. Our program also has two fields, *xCurrent* and *yCurrent* holding the position of the last move added. The function *updateMoves* examines the current grid configuration to find the valid moves and stores them in the *validMoves* variable. The function *addMove* accepts an integer array, checks first if it corresponds to a valid move included in *validMoves* and, if it does, it adds it to the *allMoves* variable, updates the remaining moves and the two fields *xCurrent* and *yCurrent.* The function *getClosestAt* accepts two integers, the coordinates of a point and returns the closest valid move to this point. As we said before, the distance of a move from a point is the distance of the move cross from the point. The distance metric we used to determine node proximity is the maximum of the absolute horizontal and vertical distances of two nodes. This way diagonal steps are equivalent to the horizontal and vertical ones. In the case where more than one nodes have the same distance from a given point, the function determines a priority based on the direction of the move. This priority is given by the constant integer value corresponding to the direction, shown in Fig.1. That is, horizontal first, then the positive slope diagonal and so on. There is not any reason for setting this priority. We could choose any priority rule. What is important is that for a given configuration of the grid and a given point the function must return deterministically the same move. Otherwise, if we used for example a random choice between the equally distant moves, we would end up with a genotype that generated randomly different phenotypes.

Next, we had to choose the GP fitness, function and terminal sets and other algorithmic features to implement the problem in ECJ. The fitness is the score but for the implementation we wanted a standardized fitness that would decrease whenever the score increased, with the value of zero corresponding to an ideal individual. We do not know what an ideal individual is in this case since we have no knowledge of the upper limit of moves, but, since the maximum published score is 178, we chose to use as fitness the difference between the number 200 and the score. The adjusted fitness arises from the standardized.

Initially we considered the following terminal nodes and functions:

Terminals:

random vector(ERC): a vector of two integers.

getClosestToLast: add the movement that is closer to the last one.

getBest : add the movement that creates the most moves at the next step

Functions:

progn2 : execute the two child branches in order from left-to-right
progn3 : execute the three child branches in order from left-to-right

All the files' mentioned below source code is given at appenix B. Our implementation in ECJ was based at an existing implementation of another GP problem, the Koza lawnmower problem[3]. All of our functions and terminals return a vector of two integers to their parents. The MSolData java file contains the GPData object we used. The random vector terminal is implemented as a Koza[7] ERC contained at the MsolERC java file. These constants get a random value during instantiation and they remain fixed since then. These vectors represent coordinates in the grid. When created, they get two integers in a 13 x 13 central area of the grid around the Greek cross. Each vector returns these coordinates then that are used by some function as a location in the grid. The getClosestToLast terminal calls the *getClosestAt* function with *xCurrent* and *yCurrent* as arguments, which hold the position of the last move added. This returns the closest valid move which is added by the function *addMove.* Finally, the getBest terminal calculates for each of the available moves how many moves they create for the next move and returns the one that produces the most. Another possible terminal we could have included is *getMaxNeighb* which would return the move with the most neighbors in a specified range.

The functions progn2 and progn3 are already implemented in the ECJ lawnmower example and are generally used widely in GP programming for tasks of this kind. We did a modification though. Progn2 calls the *getClosestAt* function for the data passed to it from the left branch and returns the right branch to its parent. This way we avoid evaluating twice the last child. Similarly, progn3 calls the same function for the first and second children and returns the third child to its parent. Terminals other than random vectors (ERCs) return a value [-1,-1], in which case *getClosestAt* returns -1 and *addMove* simply ignores it. Random vectors return their values and the corresponding move is added. Normally progn2 and progn3 are used only to evaluate their branches and do not implement any function on their own. Initially the random vectors were exploited by our program as inputs to another function, called RandomJump, as is used in the case of the lawnmower problem. But finally, instead of using this function, we decided to do this step into progn2 and progn3.

Since we had already implemented the game, we did not use the executable provided for the evaluation of our individuals. It was used only for testing purposes. MsolStatistics contains our statistics class, a subclass of the ECJ SimpleStatistics which describes the best individual at the end of the run. This description prints in a statistics file all the moves in two formats. The first can be used as input to the provided executable and the second to our independent program. MsolStatistics also prints the best individual genotype in a dot file that can be opened by a Graphviz renderer.

4. Design Of Experiment

White and Poulding in [4],compared the effects of mutation and crossover in GP, and the conclusion was that using only crossover and reproduction improves the results of GP only in 2 out of 6 problems examined, in comparison with the use of mutation and reproduction only. The standard Koza choices for the lawnmower problem, also, do not include any mutation at all. For these reasons we decided not to use any mutation in our experiments but keep the default ECJ implementation which uses crossover for 90% of the time and simple reproduction for the rest 10%.

Table 2 of the same paper presents the most important parameters of Genetic Programming. Since we had a lot of trial runs of the algorithm until the finished edition, we had an insight of the

effects of different function sets and the influence of many parameters. Due to time limitations we decided to use fixed values for many of them, usually the default ECJ settings which follow the standard Koza choices. More specifically, for our experiment we kept the population at 1024 and the number of the generations at 51. Trial and error runs showed that for this population size our algorithm has usually exhausted its search potential in 51 generations, so any increase in the number of generations would be redundant. We doubled the population also with the same number of generations hoping that we could hit a new high score but our trials showed that, although there was an improvement in the average results (which is not surprising) we did not manage to get a new highest score.

We used the "ramped half-and-half" technique to create the initial population. This technique provides a mixed initial population with various tree depths. It is the preferred method of Koza[7] and its implementation in ECJ works as follows: First, it chooses a random integer, d, between two given values, the minimum and maximum initial depths. Then, with a given probability, it uses either the "full method" which generates a fully developed tree of exact depth d, or the "grow method" which generates a tree with any depth between 1 and d. The "grow probability" which determines the frequency of the "grow method" has a default value of 0.5 which we did not change.

Koza by default does not use elitism but due to the highly random results of our algorithm we decided to use it. So, we always keep the two best individuals of our generations because we don't want to lose any good solution.

What is most interesting to be examined in our experiment is how different function sets affect out algorithm's solutions. Regarding this, the results from trial and error runs where different from our expectations. We thought the use of getBest would improve considerably the solutions, but what our runs showed was that this function, although might improve the average score in some cases, it did not increase our expectations to get a new high score. Instead, it increased considerably the execution time, more than ten times in many cases. This is reasonable since the functions used to find the next "best" move require testing a lot of possible configurations and are time consuming. So it seemed that ten runs with another terminal set had more chances to hit a better score than one including the GetBest terminal. Of course we would need a more detailed investigation than these trial and error runs to support our argument, but we could not include the getBest in our systematic experimentation. However, we include at the accompanying folder a file containing the results of 10 runs using ERC and GetBest. So, we conducted our experiment for the following two function sets:

Function Set 0: ERC, progn2, progn3
Function Set 1: ERC, getClosestToLast, progn2, progn3

To compare the performance of our algorithm for these two function sets we followed a simplified version of the approach used in Simon and Poulding[4] to evaluate the effects of mutation and crossover. First we designed a full factorial experiment for a limited set of parameters to optimize the performance of each function set independently. Specifically, we included in our experiment the following parameters:

-*the tournament size* : determines the number of the individuals that will be chosen at random from the population and compete for selection.

-*the probability of selecting terminals and non-terminals for crossover* : these are the probabilities that a node will be selected for crossover or other operations. ECJ allows setting the probabilities for selecting a terminal node, a non-terminal node, the root and any random node. The sum of these probabilities must be one. The last two probabilities are zero in our case, so we use as independent parameter the probability of selecting a terminal node and the

probability for selecting a non-terminal is dependent on this.

-*the maximum and minimum depth of the initial trees* : We could use both of these parameters independently, but to reduce the design points of our experiments we set the minimum initial depth to be equal to one third of the maximum initial depth.

We used a three level full factorial design with the parameter values shown at the table of Fig.2. We ran twice the program for each combination of parameter levels. This counts for 3*3*3*2 = 54 runs for each function sets. We record the fitness of the best individual of each run and the execution time as well.

Independent Parameters	Values		
Tournament Size	2	6	10
Prob. of selecting terminal	0.1	0.3	0.5
Max. depth of initial tree	4	7	10
Dependent Parameters	Values		
Prob. of selecting non-terminal	0.9	0.7	0.5
Min. depth of initial tree	1	2	3

Figure 2

For each design point the results of both runs are used. We omitted the comparison with the default ECJ values performed at [4]. Instead, we just used analysis of variance(AOV) to examine the effects of the parameters mentioned above to each function set. The underlying hypotheses of the AOV for the dependence of the response on each factor x_i is([8]):

Hypothesis H_{i0}: different levels of factor x_i have no effect on distribution of response
Hypothesis H_{i1}: different levels of factor x_i do have an effect

So, for each function set we tested these hypotheses for each parameter. We used the value 5% as our significance level.

Our DoE uses a first-order linear model with interactions to approximate the performance y of the algorithm in relation to the parameters x_i:

$$Y = \beta_0 + \sum_{i=1}^{n} \beta_i * x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \beta_{ij} * x_i * x_j + \varepsilon \quad (1)$$

Of course, approximating the relation of the actual performance to the parameters using this model is a simplification. But we believe that even a first-order linear model (with interactions) can provide adequate evidence of the effect each parameter has. Our goal is to estimate the β coefficients of the model and then use an optimization method to identify these values that maximize the performance of each algorithm.

The second part of the experiment compares the tuned algorithms. The hypotheses formed in this case are:

Hypothesis C_0: There is no difference in the performance of the algorithm that uses only the

ERC and the algorithm that uses both the ERC and the GetClosestToLast terminals.

Hypothesis C_1: There is a significant difference in the performance between these algorithms

We ran each tuned algorithm 50 times. We randomized the order of execution in both stages to reduce the effect of the nuisance factor[8]. The statistical analysis of the results of our experiment is given at the next section, along with the highest scores achieved by our implementation.

5. Results

The best score we achieved during our systematic investigation is 118. Unfortunately and ironically, there was a higher score of 124 which we had achieved during an early development stage of our program with many bugs and random behaviours and it was impossible to achieve again such a high score. The files containing these two solutions are given at the accompanying folder with names 118.txt and 124.txt, although the 124.txt file is given only as a proof of authenticity and its contents do not follow what we said here, despite their similarity. It should be noted that till the final version of our program, used for the experiment described above, some changes in the MSolStatistics class took place and the output format and files changed a lot. So the files containing the output of earlier runs may differ from the latest ones. However, even if some helpful additions are missing (like printing in an arraylist all the actions along with the corresponding moves) at those early runs, the solution and the steps leading to this is still clear. The highest score achieved during our experiment is 110. We include all the files related to this run at the accompanying folder. The first 18 steps of the 110-solution are shown at Fig. 3., both in the genotype and the phenotype. Figure 4 provides the same information for the first 16 steps of the 118-solution. The two solutions had different settings.

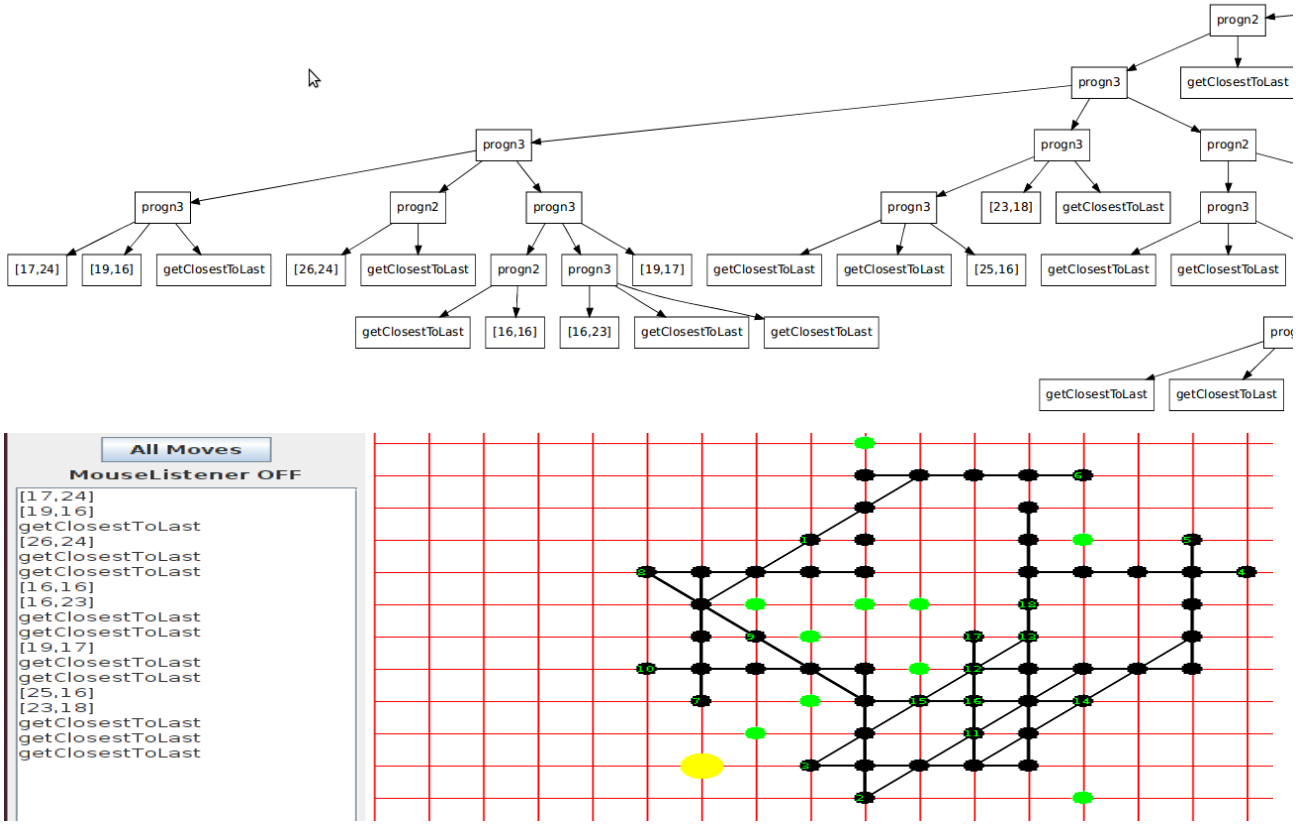

Figure 3. The 110-solution genotype (top) and the corresponding phenotype (bottom)

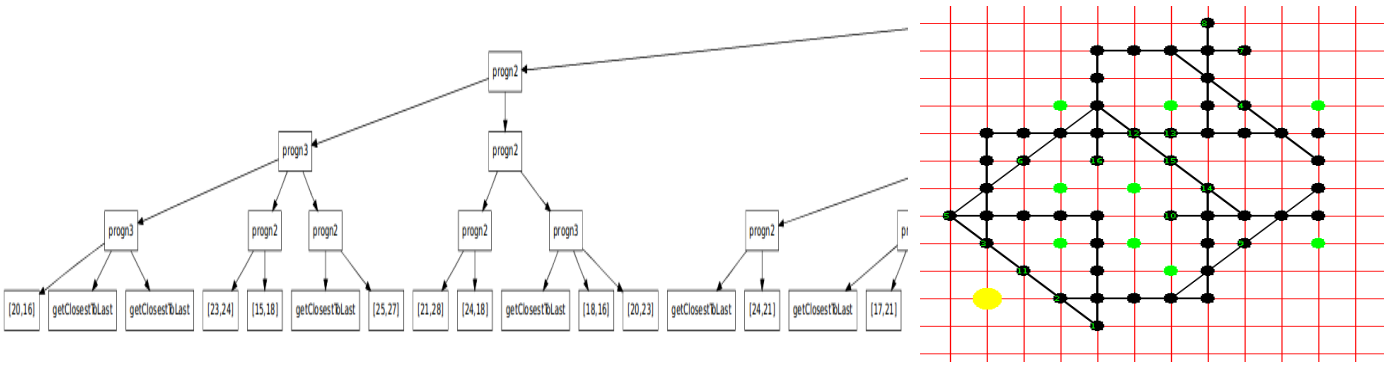

Figure 4. The 118-solution genotype (left) and phenotype(right)

Next we present the analysis of our experiment with the R language[9] . Files f0.txt and f1.txt in the “ResultsAndOther” folder contain the results of our runs in a format that can be read by R. In appendix A there is the R code used for our analysis. The coefficients of the linear model (1) for FunctionSet0 and FunctionSet1 along with their symbols interpretations are given at Fig. 5. AOV results are given at Fig. 6 and 7.

Coefficients:	(Intercept)	V1	V2	V3	V1*V2	V1*V3	V2*V3
FunctionSet0	93.403935	-0.108796	-8.159722	-0.036458	1.250000	0.003472	-0.260417
FunctionSet1	105.50000	-0.92130	-3.61111	-1.35417	-0.69444	0.09722	0.62500

Description	Symbol	Level1	Level2	Level3
Max. Depth init. Tree	V1	4	7	10
Prob. Selecting Terminal	V2	0.1	0.3	0.5
Tournament Size	V3	2	6	10

Figure 5. Linear Model Fitting for FunctionSet0 and FunctionSet1(top) and the symbols used (bottom)

FunctionSet0	Df	Sum Sq	Mean Sq	F value	Pr(>F)
V1	1	26,7	26,694	1,749	0,192
V2	1	1,4	1,361	0,089	0,767
V3	1	4,7	4,694	0,308	0,582
V1:V2	1	13,5	13,5	0,884	0,352
V1:V3	1	0	0,042	0,003	0,959
V2:V3	1	1,0	1,042	0,068	0,795
Residuals	47	717,5	15,266		

Figure 6. FunctionSet0 AOV

FunctionSet1	Df	Sum Sq	Mean Sq	F value	Pr(>F)
V1	1	96.7	96.69	6,098	0.01722 *
V2	1	32.1	32.11	2,025	0.16133
V3	1	136.1	136.11	8,584	0.00522 **
V1:V2	1	4.2	4.17	0,263	0.61063
V1:V3	1	32.7	32.67	2,06	0.15783
V2:V3	1	6.0	6.00	0,378	0.54144
Residuals	47	745.3	15.86		

Figure 7. FunctionSet1 AOV

Any method used to fit experimental data to a linear model assumes that the noise term is normally distributed. The results showed that there is a deviation of normality for the Function Set 0, while the Function Set 1 is better adjusted. Fig.8 shows the plots of the residuals against the fitted data and the QQ plots for our models. Ideally the QQ plot should approximate the $y=x$ line. As we see at the plots, FuncSet1's QQ plot, while not perfect, approximates the $y=x$ line, while FuncSet0's QQ plot presents a larger deviation. We also present at Fig.9 the histograms of the residuals for our function sets. We would like a more normalized distribution of errors(especially for FuncSet0) to accept with confidence the AOV results and the model fitting. But since these are the only models we have, we continue our planned experiment, keeping in mind these facts.

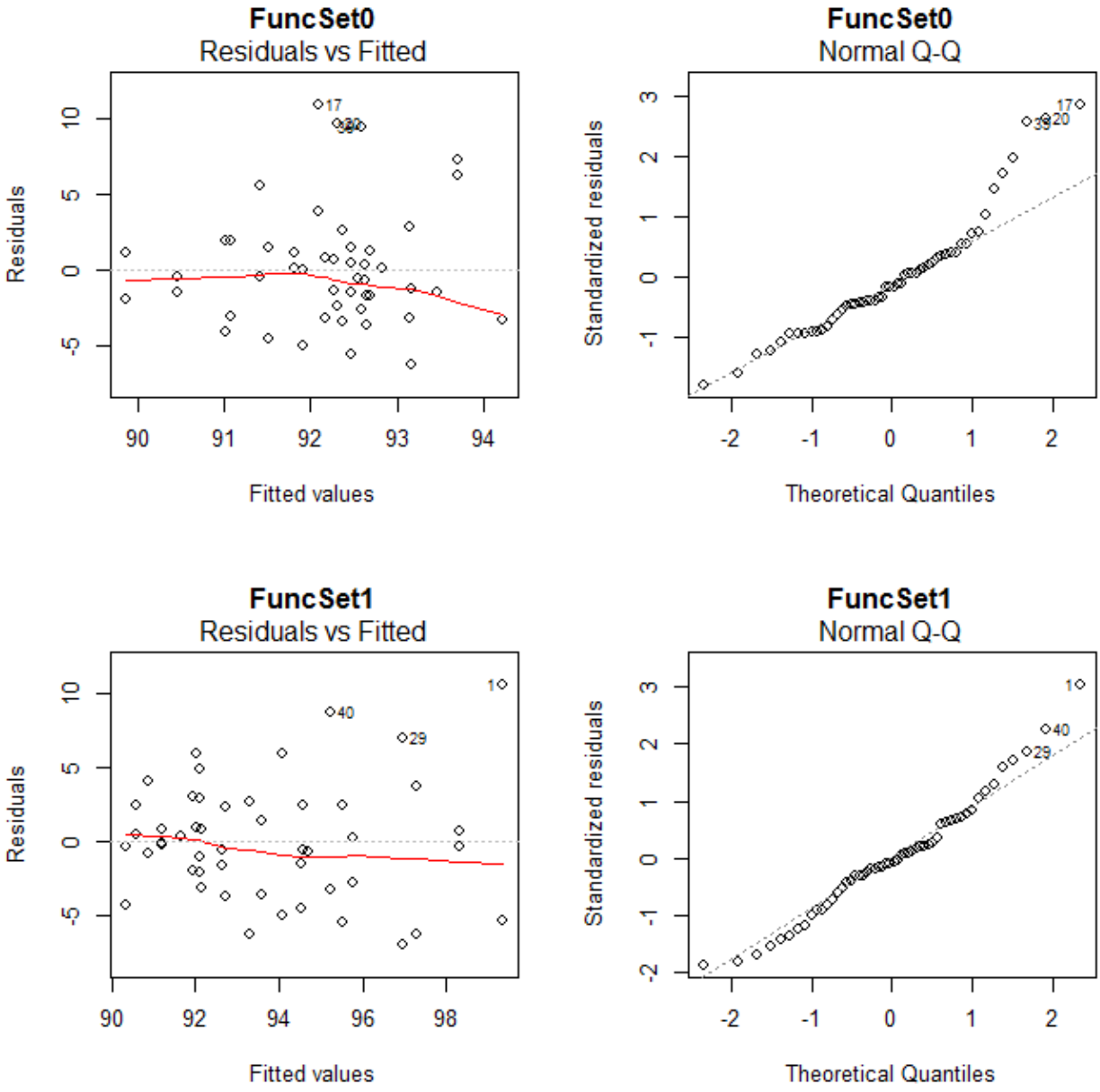

Figure 8

AOV analysis for FuncSet0 shows no dependence of the response on any of the parameters, since the p values for every term are much bigger than the 0.05 value. So, in this case the null hypotheses about the parameter effects cannot be rejected. FunctionSet1, on the other hand, seems to be slightly

affected by the tournament size(V3) and the maximum depth of the initial tree(V3). However these are only indications since p-values for theses parameters are slightly over the 5% value we set as our significance level, so neither in this case we can reject the null hypotheses and state that these parameters affect significantly the performance of FunctionSet1.

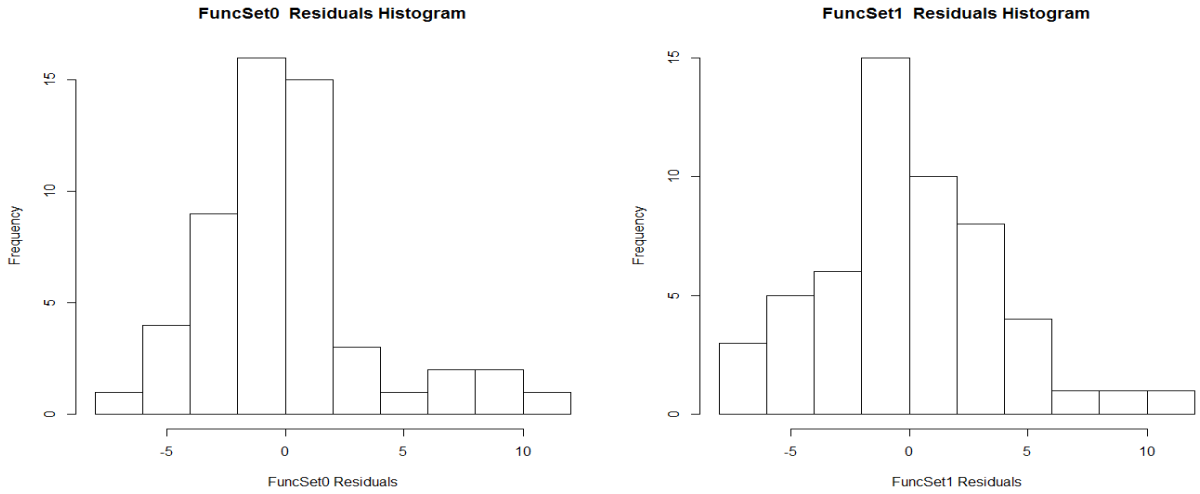

Figure 9

To obtain the tuned parameters for each algorithm we applied a generalized optimization function of R (optim) to the linear models with bounds the lowest and highest levels we used during our factorial design. We ended up with the following values for each set, shown at fig.10.

	FunctionSet0	FunctionSet1
Tournament Size	2	2
Prob. of selecting terminal	0.5	0.1
Max. depth of initial tree	10	4

Figure 10. Tuned parameters

We ran 50 times each algorithm. The files optf0.txt and optf1.txt contain the execution time and the score of each run. The boxplots for scores and run times (in seconds) are shown in Fig.11. As we can see there is a clear difference both in the score and the run time between these two function sets.

We apply non-parametric statistics, specifically the Mann-Whitney-Wilcoxon or rank sum test to analyse *statistical significance* of both measures . The null hypothesis for this test is that the responses follow the same distribution and have similar medians[8]. The alternative hypothesis is that they are different. We use again a 5% significance level. The p-value of this test for scores is 3.521e-09 and for run times 2.2e-16. In both cases the p-values are well below 5%, so we can reject the null hypothesis with confidence.

To analyse *scientific significance* we calculate the Vargha-Delaney A statistic, called the measure of stochastic superiority[10]. The value of this test, A_{12}, expresses the probability that a randomly sampled number from the first distribution will be greater than a randomly sampled number from the second distribution. A value of 0.5 of this test indicates no difference in performance, while values between 0.5 and 1.0 indicate that first algorithm is better than the second and reversely. Most statistical packages have not yet implemented the A test, but an estimation of this can be easily calculated, according to equation (12) of [10], which is repeated here for clarity.

$$A_{01} = N(X_i > Y_j)/nm + 0.5\,N(X_i = Y_j)/nm \quad (2)$$

where m and n are the sizes of the first and second samples , X_i and Y_j are observations of these samples and N here denotes the number of occurrences of each case. We named our test result A_{01} because we consider the first population to be the one of the FuncSet0 and the second of the FuncSet1. After the calculations we have for the scores, $A_{01} = 0.159$ and for the run times, $A_{01} = 1.0$. These results are consistent with the box plots of Fig.11. Especially at the case of times, we see that all the responses of the FuncSet0 are greater than those of FuncSet1. These results show a scientifically significant difference between the algorithms.

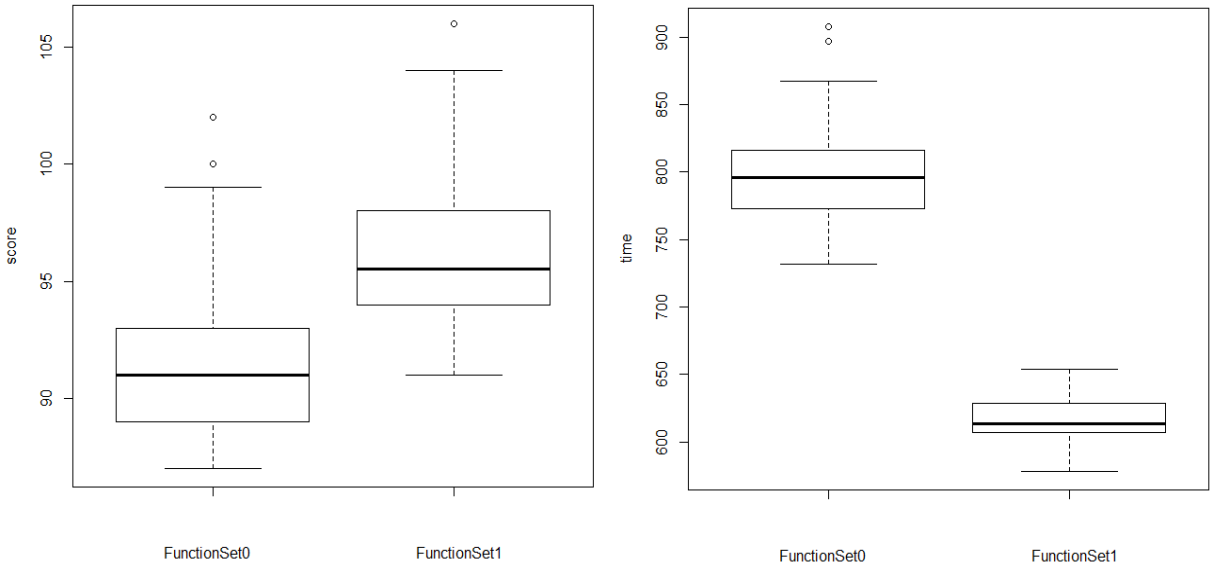

Figure 11. Scores (left) and run times in seconds (right) for FuncSet0 and FuncSet1

6. Conclusion

As we saw, the two algorithms compared present significant differences in scores and run times. The difference in run times can be explained mainly by the fact that tuned FuncSet0 uses a maximum depth of initial tree equal to 10, whereas tuned FuncSet1 uses only 4.

Regarding the difference in scores, we see that the use of an extra terminal, GetClosestToLast, improves considerably the performance. There is not any rationale behind the choice of this particular terminal. If we had used the GetBest terminal and had similar results we could justify it due to the selective function it performs. But GetClosestToLast does not make any reasonable choice at all. It simply gets the closest move, based on the proximity measure we defined. And, surprisingly, it is the function set that gave us the best scores throughout our investigation. So it is really interesting to try to understand why this is happening.

The most possible explanation is that GetClosestToLast perturbs the system from its current state and this fact by its own gives the search more flexibility. We said before that during evaluation each tree is finally equivalent to a sequence of nodes. ERC nodes represent locations on the grid, whereas other terminals perform specific functions. We believe that the improvement observed by the addition of GetClosestToLast nodes would be the same if we used any other similar function, like getting the move in the same diagonal that is furthest away. The introduction of a random

function expands the searching potential of our algorithm. Any search involving only ERCs defined at the start of each run seems to be restricted. Indeed, lets say that the finite set containing all ERCs created at the start of the run is called S. If we view each solution as a sequence of nodes, then the solutions that can be produced by our ERCs are at most equal to the number of permutations of elements of S and this is a very small subset of all the possible search paths. Inserting a function like GetClosestToLast increases significantly the number of available paths and we think that this explains the improvement in performance.

There are a some adjustments that we considered to try during the development of our program. One of them was to use evolving ERCs. Perhaps this could improve the performance of FuncSet0. Another interesting adjustment related to the use of ERCs would be to interpret them according to their position in the tree, or, equivalently, their order of evaluation. For example, like the program works now, ERCs are restricted to a central area of the grid. It does not mean necessarily that the produced moves are restricted to this area, due to the way our proximity is defined, but certainly it is quite restrictive compared with the use of ERCs which could take values from a wider area at a later stage of the game. So, instead of representing absolute positions, ERCs could represent positions relative to an evolving point, like a centre determined by the maximum and minimum dimensions of the crosses in each direction. It would be interesting to examine the behaviour of our algorithm with these adjustments.

References

[1] ECJ(2008), http://cs.gmu.edu/~eclab/projects/ecj/
[2] J.Hugues, "Methods for Statistical Inference: Extending the Evolutionary Computation Paradigm", Doctoral Dissertation, Brandeis University, Department of Computer Science, May 1999. [Online] Available: http://www.demo.cs.brandeis.edu/papers/hugues_thesis.pdf
[3] J.R.Koza, "Scalable Learning in Genetic Programming using Automatic Function Definition", [Online] Available: http://www.genetic-programming.com/jkpdf/aigp1994lawn.pdf
[4] D. R. White and S. Poulding, "A Rigorous Evaluation of Crossover and Mutation in Genetic Programming" in Proc. Of 12th EuroGP, 2009, pages 220–231
[5] ©Christian Boyer, www.morpionsolitaire.com
[6] A. Flammenkamp(November 2011), "Le Morpion Solitaire", [Online] Available: http://www.morpionsolitaire.com/Flammenkamp2011.pdf
[7] J.R.Koza, in "Symbolic Regression—Error-Driven Evolution**"** *On the Programming of Computers by Means of Natural Selection*, 6th printing, J. H. Holland USA: MIT Press, 1998. pp 243., pp 94
[8] Simon Poulding's Empirical Methods [Online] Available: www.cs.york.ac.uk/MSc/NC/empirical.php5
[9] The R language, http://www.r-project.org
[10] Vargha, A., Delaney, H.: A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J. Educational and Behavioral Statistics 25(2), 101–132 (2000)

Appendix A– The R Code

```
f0 <- read.table("f0.txt")
f1 <- read.table("f1.txt")

f0.lm <- lm(V5 ~ V1 + V2 + V3 + V1*V2 + V1*V3 + V2*V3, data = f0)
f1.lm <- lm(V5 ~ V1 + V2 + V3 + V1*V2 + V1*V3 + V2*V3, data = f1)

f0.resids <- resid(f0.lm)
f1.resids <- resid(f1.lm)

summary(aov(f0.lm))
summary(aov(f1.lm))

#optim is used for minimization, so we put a minus sign before the functions to maximize them

f0.function <- function(x) {
    -(93.404 - 0.109 * x[1] -8.16*x[2] - 0.036*x[3] + 1.25*x[1]*x[2] + 0.0035 *x[1]*x[3]- 0.26*x[2]*x[3])}
f1.function <- function(x) {
    -(105.5 - 0.921 * x[1] -3.611*x[2] - 1.354*x[3] - 0.694*x[1]*x[2] + 0.097 *x[1]*x[3] + 0.625*x[2]*x[3])}

optim(c(7, 0.3, 6),f0.function, lower = c(4, 0.1, 2), upper = c(10, 0.5, 10))
optim(c(7, 0.3, 6),f1.function, lower = c(4, 0.1, 2), upper = c(10, 0.5, 10))

optf0 <- read.table("optf0.txt")
optf1 <- read.table("optf1.txt")

wilcox.test(x, y, paired = FALSE)
```

Appendix B– The Java Code

Here we include the code of the most significant classes of our implementation. The full code can be found at the accompanying folder. Part of the used code, like some MSolERC functions performing common ECJ tasks are omitted to save some space, , as this code is mainly copy of the one used for the ECJ implementation of the lawn mower problem, with slight modifications. The same holds for progn3 class which is similar to progn2 class and GetBest class which uses the functions of Msol class with some modifications.

This is out GPproblem, Msol, the main class

```
public class MSol extends GPProblem implements SimpleProblemForm {

    //each node is determined by its x and y coordinates in the map array
    //and has 6 attributes whose indices in the third dimension of the map array are given
    //below along with their description
    public static final int CROSS = 0; //map[x][y][CROSS] is 1 if the node has a cross, 0 otherwise
    public static final int HORIZONTAL = 1; //map[x][y][HORIZONTAL]  is 1 if the node has a line to map[x+1][y], 0 otherwise
    public static final int DIAG_PLUS = 2; //map[x][y][DIAG_PLUS]  is 1 if the node has a line to map[x+1][y+1], 0 otherwise
    public static final int VERTICAL = 3; //map[x][y][VERTICAL] is 1 if the node has a line to map[x][y+1], 0 otherwise
    public static final int DIAG_MINUS = 4; //map[x][y][DIAG_MINUS]  is 1 if the node has a line to map[x-1][y+1], 0 otherwise
    public static final int MOVE = 5; //if the node acquired a cross when a move was added, this field stores the score
                              //at this moment, 0 default

    //constant values we use
     public static final int NOD = 41; //number of nodes per side 0 to 40
    public static final int X_REF = 16; // point (16,17) is our anchor point. It is the left bottom vertice
    public static final int Y_REF = 17; // of the minimum square that includes the Greek cross.
    public static final int RND_BAND = 13; //RND_BAND is used by MSolERC. Determines a central area of the grid
                                           //where the ERC vectors will get values from

    //the map contains all the information of the grid. Two dimensions are for the x and y coordinates
    //of each node and the third one contains the node's attributes described above
     public int[][][] map;

     //in every step this arraylist contains all valid moves
     //it is updated when a new move is added
     public ArrayList<int[]> validMoves;

     //keeps the history of all the moves from the beginning of a game
     public ArrayList<int[]> moveHistory;

     //keeps a record of the actions (GP nodes description) taken that produce the moveHistory
     public ArrayList<String> actionHistory;

    //the score is equal to the length of the moveHistory arraylist, but we also keep it in a separate variable
     public int score;

    //the position of the last movement cross
    public int xCurrent;
    public int yCurrent;

    public String movesForExecutable;

    //this returns the distance between two nodes as the maximum of the absolute difference of x and y coordinates
    //that is, the distance between two nodes is their vertical or horizontal distance, the bigger one
     public static int getDistance(int x1, int y1, int x2, int y2) {
         return Math.max(Math.abs(x1 - x2), Math.abs(y1 - y2));
     }

    //the evolutionary-code that follows was based in the implementation of the lawn mower problem from the ECJ app folder

    //this is to make a deep copy of the problem, copies the arrays and the arraylists
    public Object clone() {

     MSol myobj = (MSol) (super.clone());

     myobj.map = new int[NOD][NOD][6];
        for(int i=0;i<NOD;i++)
         for(int j=0;j<NOD;j++)
              for(int k=0;k<6;k++) {
                   myobj.map[i][j][k] = map[i][j][k];
              }

        myobj.moveHistory = new ArrayList<int []>();
        for (int[] move : moveHistory) {
         myobj.moveHistory.add(Arrays.copyOf(move, move.length));
        }

        myobj.actionHistory = new ArrayList<String>();
        for (String action : actionHistory) {
         myobj.actionHistory.add(new String(action));
        }

        myobj.validMoves = new ArrayList<int []>();
        for (int[] move : validMoves) {
         myobj.validMoves.add(Arrays.copyOf(move, move.length));
        }

        return myobj;
```

```
    }


    //the setup function of the problem
    public void setup(final EvolutionState state, final Parameter base) {

        super.setup(state,base);

        // verify our input is the right class (or subclasses from it)
        if (!(input instanceof MSolData))
            state.output.fatal("GPData class must subclass from " + MSolData.class, base.push(P_DATA), null);
        state.output.exitIfErrors();

        map = new int[NOD][NOD][6];
         validMoves = new ArrayList<int []>();
         moveHistory = new ArrayList<int []>();
         actionHistory = new ArrayList<String>();

        clear();
    }

    //clears the map and the arraylists and sets everything to its default value
    public void clear() {

         for(int i = 0; i < NOD; i++)
             for(int j=0; j < NOD; j++)
                 for(int k=0; k<6;k++)
                     map[i][j][k] = 0;

         moveHistory.clear();
         validMoves.clear();
         actionHistory.clear();

         score = xCurrent = yCurrent = 0;

         //draws the Greek cross at the position given in the figure1 of the assessment description
         drawCross(X_REF, Y_REF);

         update();
    }

    //draws the Greek cross relative to the given point. Particularly, sets its nodes' CROSS fields to 1
     public void drawCross(int xstart, int ystart) {
         for (int x = xstart; x < xstart + 4; x++) {
             map[x][ystart + 3][CROSS] = 1;
             map[x][ystart + 6][CROSS] = 1;
             map[x + 6][ystart + 3][CROSS] = 1;
             map[x + 6][ystart + 6][CROSS] = 1;
         }

         for (int y = ystart; y < ystart + 3; y++) {
             map[xstart + 3][y][CROSS] = 1;
             map[xstart + 6][y][CROSS] = 1;
             map[xstart + 3][y + 7][CROSS] = 1;
             map[xstart + 6][y + 7][CROSS] = 1;
         }

         for(int k=1; k<=2; k++) {
             map[xstart][ystart + 3 + k][CROSS] = 1;
             map[xstart + 9][ystart + 3 + k][CROSS] = 1;
             map[xstart + 3 + k][ystart][CROSS] = 1;
             map[xstart + 3 + k][ystart + 9][CROSS] = 1;
         }
     }

     //the update function searches the four directions of a grid and updates the valid moves arraylist
     //it calls the checkNextFive function that follows
     public void update() {

         int x, y;

         validMoves.clear();

         //horizontal - direction 1, EAST
         for (y=0; y<NOD; y++) {
             for(x=0; x<= NOD - 5; x++) {
                 int []cnf = checkNextFive(x, y, HORIZONTAL);
                 if (cnf != null) validMoves.add(cnf);
             }
         }

         //diagonal positive slope - direction 2, NORTH-EAST
         for (x=0; x < NOD - 5; x++) {
             for (y=0; y< NOD - 5; y++) {
                 if (x>=0 && x < NOD && y >= 0 && y < NOD) {
                     int []cnf = checkNextFive(x, y, DIAG_PLUS);
                     if (cnf != null) validMoves.add(cnf);
                 }
             }
         }

         //vertical direction 3 NORTH
         for (x=0; x<NOD; x++) {
             for (y=0; y<= NOD - 5; y++) {
                 int []cnf = checkNextFive(x, y, VERTICAL);
                 if (cnf != null) validMoves.add(cnf);
```

```
            }
        }

        //diagonal negative slope - direction 4, NORTH-WEST
        for (x=4; x < NOD ; x++) {
            for (y=0; y< NOD - 5; y++) {
                if (x>=0 && x < NOD && y >= 0 && y < NOD) {
                    int []cnf = checkNextFive(x, y, DIAG_MINUS);
                    if (cnf != null) validMoves.add(cnf);
                }
            }
        }

    }

    //to be a legal movement must have exactly four crosses and no lines at all
    //returns a move as an array integer
    public int[] checkNextFive(int x, int y, int d) {// d direction 1..4

        if (d < 1 || d > 4) return null;

        int ticks = 0;
        int lines = 0;
        int cind = 0;
        int []a = {};

        for(int k=0; k<5; k++) {

            switch (d) {
                case HORIZONTAL:  a = map[x + k][y];
                                  break;
                case DIAG_PLUS:   a = map[x + k][y + k];
                                  break;
                case VERTICAL:    a = map[x][y + k];
                                  break;
                case DIAG_MINUS:  a = map[x - k][y + k];
                                  break;
            }
            if (a[d] == 1 && k < 4) lines++;
            if (a[CROSS] == 1) ticks++; else cind = k;
        }

        if (lines == 0 && ticks == 4) {

            int cx, cy, ex, ey;
            cx = cy = ex = ey = -1;

            switch (d) {
                case HORIZONTAL:  cx = x + cind; cy = y; ex = x + 4; ey = y;
                                  break;
                case DIAG_PLUS:   cx = x + cind; cy = y + cind ; ex = x + 4; ey = y + 4;
                                  break;
                case VERTICAL:    cx = x; cy = y + cind; ex = x ; ey = y + 4;
                                  break;
                case DIAG_MINUS:  cx = x - cind; cy = y + cind ; ex = x - 4; ey = y + 4;
                                  break;
            }

            return new int[]{x, y, ex, ey, cx, cy, d};

        }

        return null;

    }

    // this function adds a move. A move is an integer array of size 7
    // individual fields are described inside the function
    // the function first searches the valid moves to confirm that the given array corresponds to a valid move
    // we may want purposely to give null value in which case the function does nothing
    // giving a not-null non-valid array indicates unwanted behaviour and prints a message
    public int addMove(int [] move) {

        if (move == null || validMoves.isEmpty()) return -1;

        //check to see if it is a valid move
        boolean isValid = false;

        for (int vm[] : validMoves)
            if (Arrays.equals(vm, move)) {isValid = true;break;}

        if (!isValid) {System.out.println("Wrong move!!");return -2;}

        int sx, sy, ex, ey, cx, cy, d;
        sx = move[0]; //the start point x coordinate
        sy = move[1]; //the start point y coordinate
        ex = move[2]; //the end point x coordinate
        ey = move[3]; //the end point y coordinate
        cx = move[4]; //the cross x coordinate
        cy = move[5]; //the cross y coordinate
        d = move[6]; // the direction of the line (1..4)

        score++;
        map[cx][cy][CROSS] = 1;          //updates the CROSS field of the node where the cross was added
        map[cx][cy][MOVE] = score;       //stores the number of the move which is the updated score
```

```
            xCurrent = cx;                  //updates the global variables xCurrent and yCurrent
            yCurrent = cy;

            int step_x = (ex - sx) /4;      //sets the step for the loop
            int step_y = (ey - sy) /4;

            for(int k=0; k<4; k++) {        //updates the proper direction field (d = 1..4) of all the nodes of the line added
                map[sx + k*step_x][sy + k * step_y][d] = 1;
            }

            moveHistory.add(move);  //adds the move to the move history

            update(); //updates the valid moves

            return 1;
        }

        //returns the closest move(its cross) to a point in the map
        //in case of equivalence returns deterministically the one with the "smallest" direction
        public int [] getClosestMoveAt(int x, int y) {

            if (x<=0 || y<=0) return null;

            int minDistance=100;
            ArrayList<int []> eqMoves = new ArrayList<int []>();

            for (int[] move : validMoves) {
                int cx = move[4];
                int cy = move[5];

                int dist = getDistance(x, y, cx, cy);

                if (dist<minDistance) {
                    minDistance = dist;
                    eqMoves.clear();
                    eqMoves.add(move);
                }
                else if(dist == minDistance) {
                    eqMoves.add(move);
                }
            }

            if (eqMoves.size() == 0) return null;

            if (eqMoves.size() == 1) return eqMoves.get(0);

            int [] minDirMove = null;
            int minDir = 10;
            for (int[] move : eqMoves) {
                if (move[6] < minDir) {
                    minDir = move[6];
                    minDirMove = move;
                }
            }

            return minDirMove;
        }

        //evaluate the individual
       public void evaluate(final EvolutionState state, final Individual ind, final int subpopulation, final int threadnum) {

           if (ind.evaluated) return; // if it is evaluated return

           MSolData input = (MSolData)(this.input);

           clear();

           // evaluate the individual
           ((GPIndividual)ind).trees[0].child.eval(state,threadnum,input,stack,((GPIndividual)ind),this);

           // we use KozaFitness as is the default of ECJ
           KozaFitness f = ((KozaFitness)ind.fitness);

           //our standardized fitness : 200 - score(moveHistory.size())
           f.setStandardizedFitness(state,(float)(200 - moveHistory.size()));
           f.hits = score;
           ind.evaluated = true;

       }

       //this is used by the statistics class, MSolStatistics, to print the best of the run
       //it prints all the moves in two formats:
       // 1) as a single string that can be given as input to the executable given to us
       // 2) as many strings in seperate lines that can be read as input by our program
       public void describe(final EvolutionState state, final Individual ind, final int subpopulation, final int threadnum, final
int log) {

           state.output.println("\nBest Individual's Map\n====================", log);

           clear();
           moveHistory.clear();

           // evaluate the individual
           ((GPIndividual)ind).trees[0].child.eval(state,threadnum,input,stack,((GPIndividual)ind),this);

           //this is for the executable given, prints all the moves in a single string
```

```
        movesForExecutable = "";
        for (int[] m : moveHistory) {
            state.output.print(m[0] + " " + m[1] + " " + m[2] + " " + m[3] + " ", log);
            movesForExecutable += m[0] + " " + m[1] + " " + m[2] + " " + m[3] + " ";
        }

        state.output.println("", log);
        state.output.println("\n\nScore=" + moveHistory.size(), log);
        state.output.println("", log);
        state.output.println("", log);

        state.output.println("\n\n====================\n", log);

        //the following are to be read by our program
        state.output.println("TO_BE_USED_AS_INPUT", log);

        int i=0;

        for (int[] m : moveHistory) {
            state.output.println(m[0] + " " + m[1] + " " + m[2] + " " + m[3] + " " + m[4] + " " + m[5] + " " + m[6] + "\taction\t" +
actionHistory.get(i++), log);
        }
    }
 }
```

The GPData

```
//our GPData object
public class MSolData extends GPData
{
    public int x;
    public int y;

    public void copyTo(final GPData gpd) {
         MSolData d = (MSolData)gpd;
        d.x = x;
        d.y = y;
    }
}
```

Our Statistics Class

```
//our statistics class extends SimpleStatistics and creates 3 extra files
public class MSolStatistics extends SimpleStatistics {

    long startTime;
    String statFileName;

    public int dotLog = 0;  //prints the genotype tree of the best if the run in a dot file
    public int bestLog = 0; //prints the best of the run in various forms
    public int globalLog = 0; //used to print only score and run time in a single line

     public void setup(final EvolutionState state, final Parameter base)
     {
          super.setup(state,base);

          //keep a record of the start time, we will print the total run time at the end
          startTime = System.currentTimeMillis();

          File statisticsFile = state.parameters.getFile(base.push(P_STATISTICS_FILE),null);

          try {
              DateFormat dateFormat = new SimpleDateFormat("_ddHHmm");
              String timeString = dateFormat.format(Calendar.getInstance().getTime());
              statFileName = statisticsFile.getCanonicalPath(); //the .params file name

              timeString = "";

              String dotFileName = statFileName.replaceAll(".stat", timeString + ".dot");
              String bestFileName = statFileName.replaceAll(".stat", timeString + "_best.txt");
              String globalFileName = statFileName.replaceAll(".stat", "global.txt");

              File dotFile = new File(dotFileName);
              File bestFile = new File(bestFileName );
              File globalFile = new File(globalFileName);

              dotLog = state.output.addLog(dotFile, true, false);
              bestLog = state.output.addLog(bestFile, true, false);
              globalLog = state.output.addLog(globalFile, true, false);

              //read parameter values from the params file and write them to bestLog
              int popSize = state.parameters.getInt(new Parameter("pop.subpop.0.size"), null);
              int generations = state.parameters.getInt(new Parameter("generations"), null);
              double selTerm = state.parameters.getDouble(new Parameter("gp.koza.ns.terminals"), null);
              double selNonTerm = state.parameters.getDouble(new Parameter("gp.koza.ns.nonterminals"), null);
              int initMinDepth = state.parameters.getInt(new Parameter("gp.koza.half.min-depth"), null);
              int initMaxDepth = state.parameters.getInt(new Parameter("gp.koza.half.max-depth"), null);
              int tournamentSize = state.parameters.getInt(new Parameter("select.tournament.size"), null);

              state.output.println("popSize= " + popSize, bestLog);
              state.output.println("generations= " + generations, bestLog);
              state.output.println("initMaxDepth= " + initMaxDepth , bestLog);
              state.output.println("initMinDepth= " + initMinDepth , bestLog);
              state.output.println("selTerm= " + selTerm, bestLog);
              state.output.println("selNonTerm=" + selNonTerm, bestLog);
```

```
                state.output.println("tournamentSize= " + tournamentSize, bestLog);
                state.output.println("\n\n", bestLog);
            }
            catch (IOException i) {
                 state.output.fatal("An IOException occurred while trying to create the dotLog file:\n" + i);
            }
        }


    /** Logs the best individual of the run. */
    public void finalStatistics(final EvolutionState state, final int result)
        {
        super.finalStatistics(state,result);

        GPIndividual bestInd = (GPIndividual)best_of_run[0].clone();
        GPTree tree = (GPTree)bestInd.trees[0].clone();

        //this is for the dot file
        tree.printStyle = GPTree.PRINT_STYLE_DOT;
        tree.printTreeForHumans(state,dotLog);

        long runTime = System.currentTimeMillis() - startTime;

        if (state.evaluator.p_problem instanceof SimpleProblemForm) {

         SimpleProblemForm p =  ((SimpleProblemForm)(state.evaluator.p_problem.clone()));
         p.describe(state, best_of_run[0], 0, 0, bestLog);

         String[] fields = statFileName.split("/");
         String entry = fields[fields.length -2] + "\t" + fields[fields.length -1].replace(".txt", "");

            state.output.println(entry + "\trunTime= " + runTime + "\tscore=" + ((MSol)p).score, globalLog);

        }


    }
}
```

Class progn2 – progn3 is similar with three children

```
public class Progn2 extends GPNode {

    public String toString() { return "progn2"; }

    public int expectedChildren() { return 2; }

    public void eval(final EvolutionState state, final int thread, final GPData input, final ADFStack stack, final GPIndividual
individual, final Problem problem) {

     MSol p = (MSol)problem;
        MSolData d = (MSolData)(input);

        children[0].eval(state,thread,input,stack,individual,problem);

        //only if it is an ERC it has d.x>0 and d.y>0
        if (d.x >0 && d.y >0) {
          int [] move = p.getClosestMoveAt(d.x, d.y);
          if (p.addMove(move) > 0) p.actionHistory.add(new String("[" + d.x + "," + d.y + "]"));
        }

        children[1].eval(state,thread,input,stack,individual,problem);
    }
}
```

This is our ERC class

```
//this is our ERC - it is a copy from ECJ lawnmower problem ERC with slight modifications
//comments in this file are from the original file
public class MSolERC extends ERC {

    public int x;
    public int y;

    //takes a value in the central region of the grid, defined by MSol.X_REF & MSol.Y_REF (16,17) and MSol.RND_BAND(13)
    public void resetNode(final EvolutionState state, final int thread) {
     x = MSol.X_REF - 1 + state.random[thread].nextInt(MSol.RND_BAND);
     y = MSol.Y_REF - 1 + state.random[thread].nextInt(MSol.RND_BAND);
    }
…
}
```

The terminal GetClosestToLast

```
//return the "closest" move to the last move added
//whenever a move is added, global variables xCurrent and yCurrent of the MSol problem
//hold its cross position
public class GetClosestToLast extends GPNode {

     public String toString() { return "getClosestToLast"; }

    public int expectedChildren() {return 0;}

    public void eval(final EvolutionState state, final int thread, final GPData input, final ADFStack stack, final GPIndividual
individual, final Problem problem) {

     MSol p = (MSol)problem;
```

Candidate Y1447424

```
     MSolData d = (MSolData)(input);

     int [] move = p.getClosestMoveAt(p.xCurrent, p.yCurrent);
     if (p.addMove(move) > 0) p.actionHistory.add(new String("getClosestToLast"));


     //returns -1
        d.x = -1;
        d.y = -1;
    }

}
```