International Journal of Robotics and Automation (IJRA)
Vol. 3, No. 3, September 2014, pp. 201~211
ISSN: 2089-4856

Controlling Bloat in Genetic Programming for Sloving Wall Following Problem

Navid Bazrkar, Mostafa Nemati, Reza Salimi
Computer Science Department, Tabari University, Babol, Iran

Article Info

Article history:

Received Sep 5, 2013
Revised Jun 20, 2014
Accepted Jul 15, 2014

Keyword:

Automatic programming
Controlling bloat
Genetic programming
Wall following problem

ABSTRACT

The goal in automatic programming is to get a computer to perform a task by telling it what needs to be done, rather than by explicitly programming it.With considers the task of automatically generating a computer program to enable an autonomous mobile robot to perform the task of following the wall of an irregular shaped room. During the evolution of solutions using genetic programming (GP) there is generally an increase in average tree size without a corresponding increase in fitness-a phenomenon commonly referred to as bloat. Many different bloat control methods have been proposed. This paper review, evaluate, implementation and comparison of these methods in wall following problem and the most appropriate method for solving bloat problem is proposed.

Corresponding Author:

Mostafa Nemati,
Computer Science Department, Tabari University
Manjil, Bahar Ave, P:67, Tabari Babol, Iran
Email: Mostafa.manjil@gmail.com

1. INTRODUCTION

Genetic Programming (GP) is the automated learning of computer programs [2, 3]. Theoretically, it can solve any problem whose candidate solutions can be measured and compared, making it a widely applicable technique. Furthermore, the solutions found by GP are usually provided in a format that users can understand and modify to their needs. But its high versatility is also the cause of some difficulties. Users must set a number of parameters related to several aspects of the evolutionary process, some of which may influence the search process so strongly as to actually prevent an optimal solution to be found, if set incorrectly. And even when a reasonable match between problem and parameters is achieved, a major problem remains, one that has been studied for more than a decade: code growth [1].

The search space of GP is virtually unlimited and programs tend to grow in size during the evolutionary process. Code growth is a healthy result of genetic operators in search of better solutions, but it also permits the appearance of pieces of redundant code that increase the size of programs without improving their fitness. many different bloat control methods have been proposed.

This paper review, evaluate, implementation and comparison of these methods in wall following problem. The next section deals with bloat, describing the main theories regarding why it occurs. Section 3 describes the Dynamic Limits in detail, while Sect. 4 describes Variations on size and depth problems and Section 6 reports the results of the comparisons among the different techniques, while Sect. 6 discusses them and presents some considerations on the usage of limit restrictions in GP. Finally, Sect. 7 concludes and Sect. 8 provides ideas for future work.

2. BLOAT

When Koza published the first book on GP [3], most of the evolved programs therein contained pieces of code that did not contribute to the solution and could be removed without altering the results produced. Besides imposing a depth limit to the trees created by crossover to prevent spending computer resources on extremely large programs, Koza also routinely edited the solutions provided at the end of each run to simplify some expressions while removing the redundant code.

Two years later, Angeline remarked on the ubiquity of these redundant code segments and, based on a slight biological similarity, called them introns [7]. In spite of classifying them as extraneous, unnecessary and superfluous, Angeline noted that they provided crossover with syntactically redundant constructions where splitting could be performed without altering the semantics of the swapped sub trees. Referring to some studies where the introduction of artificial introns was helpful or even essential to the success of genetic algorithms, Angeline revels in the fact that introns emerge naturally from the dynamics of GP. He even goes as far as to state that ‘‘it is important then to not impede this emergent property as it may be crucial to the successful development of genetic programs’’ [7].

It is possible that introns may provide some benefits. A non-intuitive effect that introns may have in GP is code compression and parsimony. It is not the bloated code full of redundant a segment that is parsimonious, but the effective code that remains after removing the introns. Under specific conditions, particularly in the presence of destructive crossover, there is evidence that the existence of introns in the population results in shorter and less complex effective solutions .Compact solutions are thought to be more robust and generalize better [8]. Introns also do seem to provide some protection against the destructive effects of crossover and other genetic operators [9, 8] although this may not always be helpful. The usage of explicitly defined artificial introns has yielded generally good results in linear GP [10, 11], but in tree based GP it usually degraded the performance of the search process [13].

Regardless of its possible benefits to GP, the side effects of intron proliferation are very serious. Computational resources may be totally exhausted in the storage, evaluation and swapping of code that contributes nothing to the final solution, preventing GP from performing the effective search needed to find better solutions. Bloat is now widely recognized as a pernicious phenomenon that plagues most progressive search techniques based on discrete variable-length representations and using fixed evaluation functions [14, 15]. Bloat control has become a very active research area in GP, already subject to different theoretic and analytic studies [10]. Several theories concerning why bloat occurs have been advanced, and many different bloat control methods have been proposed.

Luke observed that underlying causes of code bloating contains Hitchhiking, Defense against Crossover, Removal Bias, Fitness Causes Bloat and Modification Point Depth. Then it coined that the code bloating in evolutionary computation field is identified with the C-value Paradox in the biology [12]. Therefore, if no any restrictions adding to genetic operations, code bloating is inevitable in GP. Just like Luke said in [6], “In a very real sense, bloating makes genetic programming a race against time, to find the best solution possible before bloat puts an effective stop to the search.” For further information on bloat see [16].

3. DYNAMIC DEPTH LIMIT

Dynamic Maximum Tree Depth [4] is a bloat control technique inspired by the traditional static limit. It also imposes a depth limit on the individuals accepted into the population, but this one is dynamic, meaning that it can be changed during the run. The dynamic limit is initially set with a low value, but at least as high as the maximum depth of the initial random trees. Any new individual who breaks this limit is rejected and replaced by one of its parents instead (as with the traditional static limit), unless it is the best individual found so far. In this case, the dynamic limit is raised to match the depth of the new best-of-run and allow it into the population. Figure 1 shows the pseudo code of this procedure. The result is a succession of limit risings, as the best solution becomes more accurate and more complex.

Dynamic Maximum Tree Depth does not necessarily replace the traditional depth limit: both dynamic and fixed limits can be used at the same time (not shown in Figure 1). When this happens, the dynamic limit always lies somewhere between the initial tree depth and the fixed depth limit. The simplicity of Dynamic Maximum Tree Depth makes it easy to use with any set of parameters and/or coupled with other techniques for controlling bloat.

The dynamic limit may also be used for another purpose besides controlling bloat. In real world applications, one may not be interested or able to invest a large amount of time in achieving the best possible solution, particularly in approximation problems. Instead, one may consider a solution to be acceptable only if it is sufficiently simple to be understood, even if its accuracy is known to be worse than the accuracy of other more complex solutions. Plus, shorter solutions tend to generalize better (Sect. 2). One way to avoid the

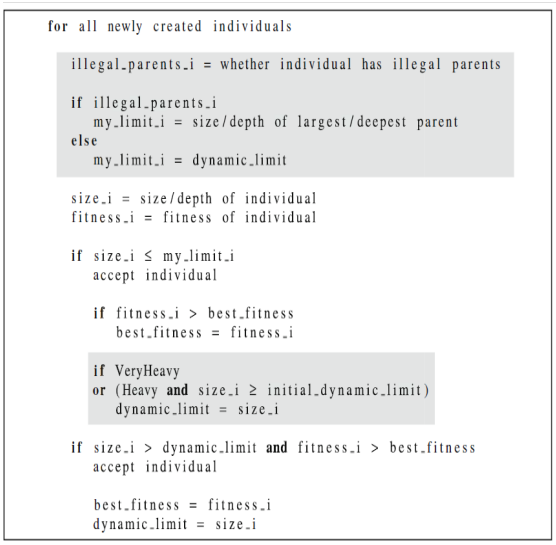
over-specialization of the solutions found by GP is to choose termination criteria that will not force the evolutionary process to go on indefinitely in search of a perfect solution. A less stringent stop condition yields a somewhat inaccurate solution, but one that is also simpler and hopefully generalizes better. However, setting the right stop condition may be a major challenge in itself, as one cannot predict the complexity needed to achieve a certain level of accuracy. By starting the search with a low dynamic limit for tree depth, the search is forced to concentrate on simple solutions first. The limit is then raised when a new solution is found that is more complex, but also more accurate, than the previous one. As the evolution proceeds, the limit is repeatedly raised as more and more complex solutions achieve increasingly higher levels of accuracy. Regardless of the stop condition, the Dynamic Maximum Tree Depth technique can in fact provide a series of solutions of increasing complexity and accuracy, from which the users may choose the one most adequate to their needs.

```
for all newly created individuals

   depth_i = depth of individual
   fitness_i = fitness of individual

   if depth_i ≤ dynamic_limit
      accept individual

      if fitness_i > best_fitness
         best_fitness = fitness_i

   if depth_i > dynamic_limit and fitness_i > best_fitness
      accept individual

      best_fitness = fitness_i
      dynamic_limit = depth_i
```

Figure 1. Pseudo code of the basic Dynamic Maximum Tree Depth procedure (with no static limit)

4. VARIATIONS ON SIZE AND DEPTH

The original Dynamic Maximum Tree Depth was soon extended to include additional variants: a heavy dynamic limit, called heavy because it falls back to lower values whenever allowed, and a dynamic limit on size instead of depth. Figure 2 shows the general acceptance procedure (including all the variants, using no static limit) that all newly created individuals must pass before being accepted into the new generation. This is an extension of the procedure in Figure 1. Only the shaded parts are completely new. Both the new code and the small differences in the common code will be explained in the next sections. Any individual that does not meet the size/depth/fitness requirements of the Dynamic Limits method will not be accepted by this procedure, but instead replaced by one of its parents.



```
for all newly created individuals

  illegal_parents_i = whether individual has illegal parents

  if illegal_parents_i
     my_limit_i = size/depth of largest/deepest parent
  else
     my_limit_i = dynamic_limit

  size_i = size/depth of individual
  fitness_i = fitness of individual

  if size_i ≤ my_limit_i
     accept individual

     if fitness_i > best_fitness
        best_fitness = fitness_i

     if VeryHeavy
     or (Heavy and size_i ≥ initial_dynamic_limit)
        dynamic_limit = size_i

  if size_i > dynamic_limit and fitness_i > best_fitness
     accept individual

     best_fitness = fitness_i
     dynamic_limit = size_i
```

Figure 2. Pseudo code of the general Dynamic Limits acceptance procedure (all variants, no static limit).

This is an extension of the procedure in Figure 1. Only the shaded code is completely new

5. HEAVY DYNAMIC LIMIT

Dynamic Maximum Tree Depth is capable of withstanding a considerable amount of parsimony pressure, as proven by the results obtained by initializing the dynamic limit with the lowest possible value, the maximum depth of the initial random trees [4] (Sect. 3.1.2). So there seems to be no reason why the limit should not be allowed to fall back to lower values in case the depth of the new best individual becomes lower than the current limit, an occurrence which is actually very common. So the first variation introduced to the original Dynamic Maximum Tree Depth is the Heavy dynamic limit, one that accompanies the depth of the best individual, up or down, with the sole constraint of not going lower than its initialization value [5]. An additional variation is the VeryHeavy limit, similar to the heavy variant but allowed to fall back even below its initialization value. Both these variants are covered in the second shaded block of Figure 2.

As expected, whenever the limit falls back to a lower value, some individuals already in the population immediately break the new limit, becoming ‘illegals’. There was a vast range of options to deal with them, the more drastic being their immediate removal from the population, possibly replacing them by new random individuals. However, since these new ‘illegals’ could be the ones who managed to produce the new best individual, eliminating them could be harmful for the search process. A much softer option was adopted: the ‘illegals’ are allowed to remain in the population as if they were not breaking the limit, but when breeding, their children cannot be deeper than the deepest parent. This naturally and gradually places the population within limits again. The first shaded block of Figure 2 deals with choosing the right limit (my limit i) to use for the new individual, depending on whether it has illegal parents. The first comparison involving the variable dynamic limit in Figure 1 (if size_i <= dynamic_limit) is now performed using the variable my_limit_i in Figure 2 (if size_i <= my_limit_i) .

6. DYNAMIC SIZE LIMIT

Even though bloat is known to affect many other search processes using variable length representations (Sect. 2), depth limits cannot be used on non tree-based GP systems. Extending the idea of a dynamic limit to other domains must begin with the removal of the concept of depth, replacing it with the concept of size. The second variation on the original Dynamic Maximum Tree Depth is the usage of a dynamic size limit, where size is the number of nodes [5]. If a static limit is to be used along with this dynamic limit, it should also be on size, not depth. The variable **depth_i** of the pseudo code in Figure 1 is now called **size_i** in Figure 2, although it can refer to either size or depth.

Tree initialization in tree-based GP also typically relies on the concept of depth. This is the case with the popular Grow, Full, and Ramped Half-and-Half initialization methods [3]. Both Grow and Full methods create trees by adding random nodes until a maximum specified depth. The Grow method adds

internal or terminal nodes, except at the maximum depth, where the choice is restricted to terminals. This creates trees with different shapes and sizes. The Full method creates balanced trees, with all terminal nodes at the maximum depth. It does this by adding random internal nodes except at the maximum depth, where it selects only terminals. The Ramped Half-and-Half method is a combination of the two, where half the population is initialized with Grow, and the other half with Full. In each half trees are created with depth limits ranging from 2 to a specified maximum value, ensuring a very diverse initial population.

When using the dynamic size limit, it makes no sense to keep using depth as a restriction on tree initialization. So a modified version of the Ramped Half-and-Half initialization method was created [5], where an equal number of individuals are initialized with sizes ranging between 2 and the initial value of the dynamic size limit. For each size, half or the individuals are initialized with the Grow method, and the other half with the Full method, that have also been modified to fit the size constraints only. In the modified Grow method, the individual grows by addition of random nodes (internal or terminal) without exceeding the maximum specified size; the modified Full method chooses only internal nodes until the size is close to the specified, and only then chooses terminals. Unlike the original Full method, it may not be able to create individuals with the exact size specified, but only close (and never exceeding). Figure 3 shows the pseudo code of both methods.

```
nodes_left = maximum specified size

while nodes_left > 0

   if nodes_left = 1
      pool = terminals

   else
      selected_functions = functions with arity < nodes_left

      if Grow
         pool = selected_functions + terminals

      if Full
         pool = selected_functions
         if pool is empty
            pool = terminals

   add a random node from the pool
   nodes_left = nodes_left – 1
```

Figure 3. Pseudo code of the modified Grow and Full initialization methods

7. WALL FOLLOWING PROBLEM

Mataric [17] described the problem of controlling an autonomous mobile robot to perform the task of following the walls of an irregular room. The robot is capable of executing the following five primitive motor functions: moving forward by a constant distance, moving backward by a constant distance, turning right by 30 degrees, turning left by 30 degrees, and stopping. The robot has 12 sonar sensors which report the distance to the nearest wall. Each sonar sensor covers a 30 degree sector around the robot. In addition, there was a sensor for the STOPPED condition of the robot. Figure 4 shows an irregularly shaped room and the distances reported by the 12 sonar sensors. The robot is shown at point (12, 16) near the center of the room. The north (top) wall and west (left) wall are each 27.6 feet long.

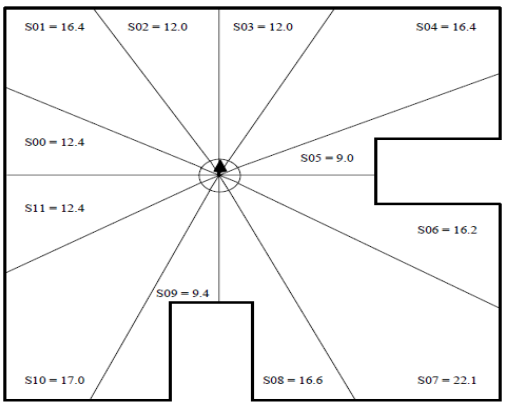


Figure 4. Irregular room with robot with 12 sonar sensors located near middle of the room

8. EXPERIMENTS

All the experiments were performed on tree-based GP using my java code [??] Wall following problem was chosen to test, and this is a first research of bloat problem in wall following.

8.1. Process of Work:

The problem is solved using a GP.
Gene: A Scheme program that is represented by a Tree (Like a compiler AST).
(a) Terminals: {SS0…SS11, EDG, SS, MSD}
EDG == 2.3, SS ==minimum of all sensors, MSD == 2.0.
SS0..SS11 are the sensors in a 30 degree interval, and each returns the distance to the first intersected wall.
(b) Functions: {PLUS, IFLTE, PROGN2, TL, TR, MB, MF}
All the functions are taken from the article, except PLUS that takes 2 arguments and return their sum. TL, TR, MB, MF – moves the robot and returns the minimum of SS2 and SS3. IFLTE - equals to "if less then equal then else". PROGN2 takes 2 arguments evaluates both of them and returns the second. All the functions that make the robot move take one time step.

Fitness function:

The fitness function is the number of tiles (out of 56) that the robot moves through them. Each robot has 400 time steps to simulate, and if there is no change from the last evaluation we will stop the evaluation loop. Selection method: Fitness proportion function, using roulette wheel sampling. Population size = 500. Num of generations = 51. Crossover: Pc = 0.9.
The cross over is done between sub trees, the sub tree is picked randomly. Mutation: Pm = 0.01. In our case the mutation objective is to reduce tree size. It will pick a sub tree and replace it with a new sub tree of depth 1.

Steps / Process:
1. Initial population calculated randomly, all the trees are complete in depth of 2.
2.1 Fitness calculation.
2.2 apply dynamic depth method [1]
2.3 apply heavy method (size/depth) [1]
2.4 apply falls back method (lower values) [1]
3. Selection.
4. Crossover and mutation.
5. Moving to next generation.

One of the problems in GP is that the Tree – Code have the tendency to grow throughout the generations. We tried to moderate the problem using tree method and mutation function (with probability of Pm=0.01), we replaced a random sub tree with a new tree of size 1. This should cause the tree – code to reduce its size. The Fitness function doesn't take into account the length of the path, and number of time steps needed to complete "collecting" all the tiles. So we tried to replace the Fitness function with a new Function. F1 = (Tiles stepped on) + factor /(Time steps) + factor /(Wall collisions). Where Tiles stepped on - the number of Tiles that the robot "picked" during the run. Time steps – the number of time steps that took the robot to complete the mission (max 400) .Wall collisions – the number of walls that the robot collided during the run. The new fitness function had no effect on the evolution progression, so we decided to drop the idea.
You can also see the despite the fact that the robot didn't take the above data it found a very short way to solve the problem. You can see clearly the increasing value of the average and best fitness during the generations. It confirms the correctness of using GP to solve this problem. In the simulated plot of generation 30, 40, 51 you can see an interesting behavior at the beginning of the robot path, it does some unexplained loops. We witnessed the same behavior in Koza's article.

9. RESULTS

This section presents the results of all the experiments. Several plots and their brief and Best Tree Depth for blot descriptions are presented. Figure 5 shows a boxplot of the best fitness of run and shows the evolution of the best fitness along the run. And Compare canonical GP with new method in standard room and new room are presented.
A.1 output in canonical GP in standard room
Best individual: 49
Best Tree Depth: 5

(IFLTE (PROGN2 (IFLTE S06 (PROGN2 (PROGN2 MF S11) (PROGN2 S11 MF)) (PROGN2 MB S10) (PROGN2 S04 S09)) MF) (PROGN2 (IFLTE EDG TL TR MB) MSD) MF TR)

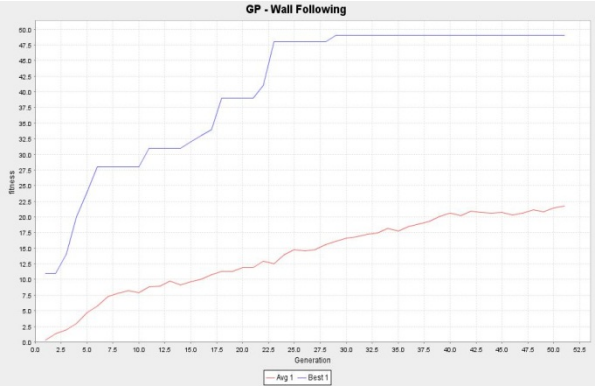


Figure 5.1. Output in canonical GP

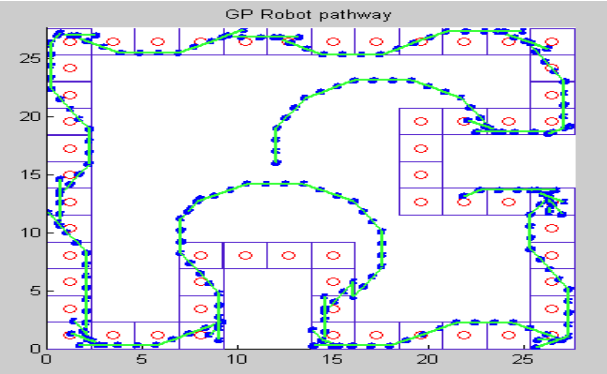


Figure 5.2. Output in canonical

A.2 output with new methods in standard room
Best individual: 50
Best Tree Depth: 3
(IFLTE (IFLTE MF MF MF S00) (IFLTE S08 MB S05 TR) (IFLTE TL MF MF MF) (PROGN2 (IFLTE TR S10 S02 MF) MF))

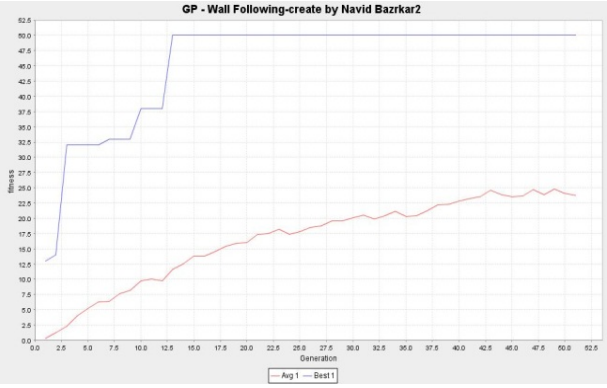


Figure 6.1. Output with new method

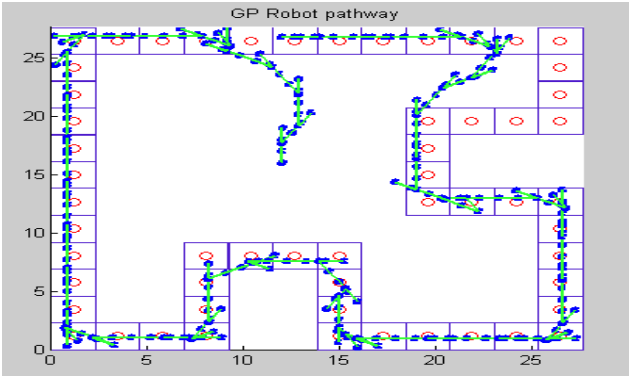


Figure 6.2. Output with new method

B1. output in canonical GP in new room1
Best individual: 54
Best Tree Depth: 6
(IFLTE (IFLTE S00 MF (PROGN2 (PLUS (PROGN2 S01 (IFLTE MF MF MF MF)) (PROGN2 (PROGN2 TR MB) MB)) S02) MF) S08 MF MF)

Figure 7.1. Output in canonical GP

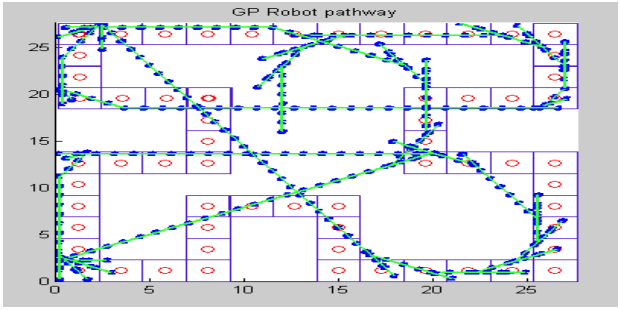


Figure 7.2. Output in canonical GP

B2. output with new method in new room1
Best individual: 54
Best Tree Depth: 4
(IFLTE (IFLTE MB (IFLTE S00 MB MB MB) (IFLTE MB EDG MB MB) MB) (IFLTE MB MB MB (IFLTE TL MB MB MB)) (IFLTE MF S10 TR (IFLTE EDG MB MB MB)) (IFLTE MF TR (IFLTE S00 MB (IFLTE MSD S09 MF TR) MB) TR

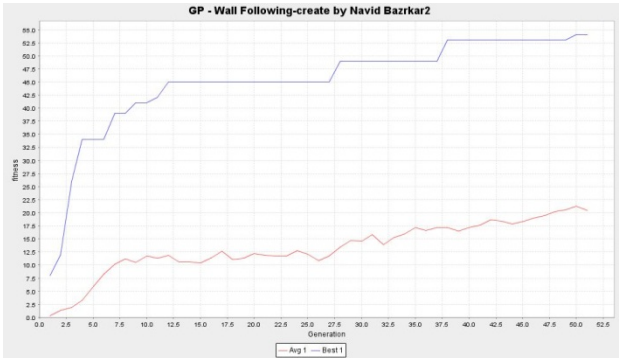


Figure 8.1. Output with new method

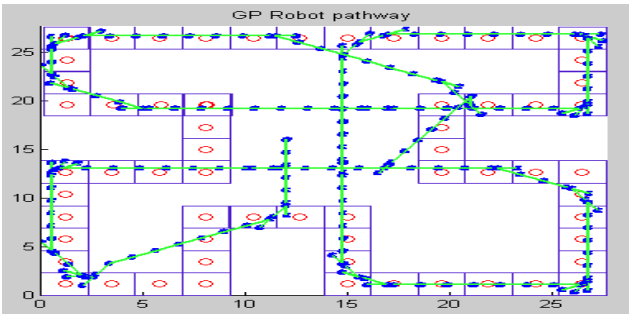


Figure 8.2. Output with new method

C1. output in canonical GP in new room2
Best individual: 33
Best Tree Depth: 6
(PROGN2 (IFLTE S09 (IFLTE (IFLTE MF (IFLTE S09 MF (IFLTE TR SS MB MB) MF) (IFLTE TR SS MB MB) (PROGN2 SS S070)) MF (IFLTE (IFLTE S09 MF (IFLTE TR SS MB MB) MF) MF (IFLTE TR SS MB MB) (IFLTE S09 MF (IFLTE TR SS MB MB) MF)) MF) (IFLTE (IFLTE (IFLTE S09 MF (IFLTE TR SS MB MB) (PROGN2 S05 S03)) MF (IFLTE TR SS MB MB) MF) (IFLTE S09 MF (IFLTE TR SS MB MB) MF) (IFLTE S09 (IFLTE S09 MF (IFLTE TR SS MB MB) S09) (IFLTE TR SS MB MB) MF) (IFLTE (IFLTE (IFLTE S09 MF (IFLTE TR SS MB MB) (PROGN2 S05 S03)) MF (IFLTE TR SS MB MB) MF) (IFLTE S09 MF (IFLTE TR SS MB MB) MF) (IFLTE S09 (IFLTE S09 MF (IFLTE TR SS MB MB) S09) (IFLTE TR SS MB MB) MF) MF)) (PROGN2 (IFLTE S09 MF (IFLTE TR SS MB MB) (PROGN2 SS S070)) MF)) (IFLTE (IFLTE (IFLTE S09 MF (IFLTE TR SS MB MB) (PROGN2 S05 S03)) MF (IFLTE TR SS MB MB) MF) (IFLTE S09 MF (IFLTE TR SS MB MB) (IFLTE S09 (IFLTE S09 MF (IFLTE TR SS MB MB) (PROGN2 SS S070)) (IFLTE TR SS MB MB) MF)) MF MF))

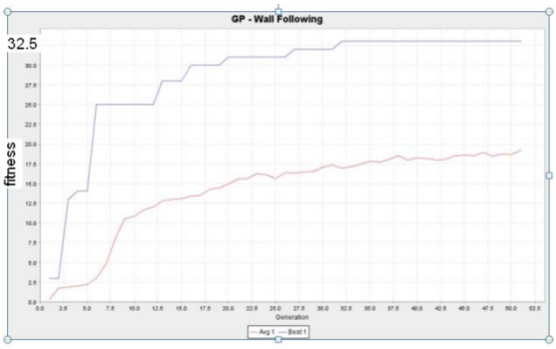


Figure 9.1. Output in canonical GP

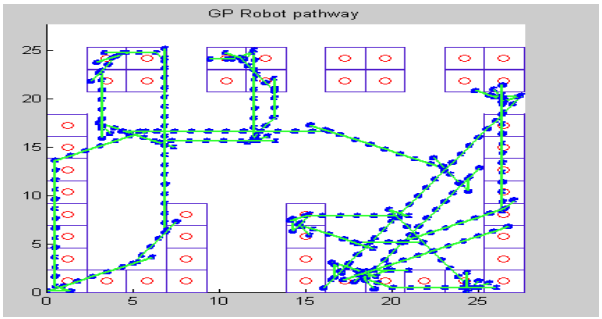


Figure 9.2. Output in canonical GP

C2. output with new method in new room2
Best individual: 30
Best Tree Depth: 2
(IFLTE (IFLTE MF S09 S06 MF) MF (PLUS MB TR) S02)

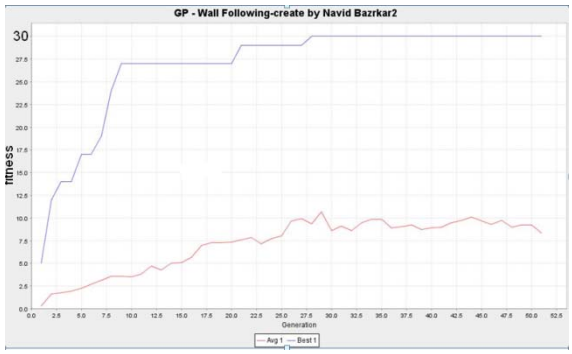


Figure 10.1. Output with new method

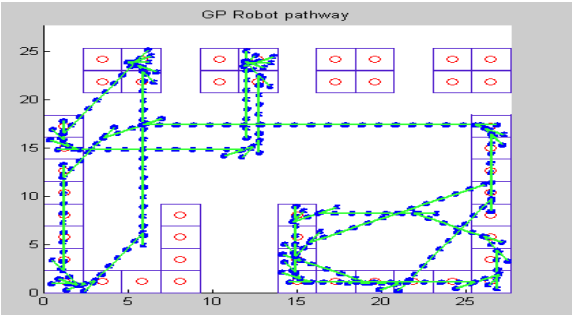


Figure 10.2. Output with new method

10. CONCLUSION

This paper was reviewed and evaluated implementation and comparison of these methods in wall following problem and the most appropriate method for solving bloat problem has been proposed.

REFERENCES

[1] Sara Silva, "Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories", Springer Science+Business Media, LLC 2009.
[2] W. Banzhaf, P. Nordin, R.E. Keller, F.D. "Francone, Genetic Programming—An Introduction" , (dpunkt.verlag and Morgan Kaufmann, Heidelberg and San Francisco, CA, 1998)
[3] J.R. Koza, "Genetic Programming – On the Programming of Computers by Means of Natural Selection" , (MIT Press, Cambridge, MA, 1992)
[4] S. Silva, J. Almeida, "*Dynamic maximum tree depth—a simple technique for avoiding bloat in treebased GP*", in Proceedings of GECCO-2003, ed. by E. Cantu´-Paz et al. (Springer, Berlin, 2003)
[5] S. Silva, E. Costa, "*Dynamic limits for bloat control—variations on size and depth*", in Proceedings of GECCO-2004, ed. by K. Deb et al. (Springer, Berlin, 2004
[6] Sean Luke, "Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat," doctoral dissertation, University of Maryland, 2000
[7] P.J. Angeline, "Genetic programming and emergent intelligence, in Advances in Genetic Programming", ed. by K.E. Kinnear Jr. (MIT Press, Cambridge, MA, 1994)
[8] P. Nordin, W. Banzhaf , "*Complexity compression and evolution*", in Proceedings of ICGA'95, ed. By L. Eshelman (Morgan Kaufmann, San Francisco, CA, 1995),
[9] T. Blickle, L. Thiele, "Genetic programming and redundancy, in Genetic Algorithms within the Framework of Evolutionary Computation" , ed. by J. Hopf (Max-Planck-Institut fu¨r Informatik, Saarbriicken, 1994),
[10] W.B. Langdon, W. Banzhaf, " *Genetic programming bloat without semantics*" , in Proceedings of PPSN- 2000, ed. by M. Schoenauer et al. Springer, Berlin, 2000
[11] P. Nordin, F. Francone, W. Banzhaf, "*Explicitly defined introns and destructive crossover in genetic programming*", in Proceedings of the Workshop on Genetic Programming: From Theory to Real- World Applications, ed. by J.P. Rosca (1995)
[12] Sean Luke, "*Evolutionary computation and the C-value paradox*", Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC, USA
[13] D. Andre, A. Teller, " *A study in program response and the negative effects of introns in genetic Programming*" , in Proceedings of GP'96, ed. by J.R. Koza et al. (MIT Press, Cambridge, MA, 1996)
[14] W. Banzhaf, F.D. Francone, P. Nordin, "Some emergent properties of variable size EAs". Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97 (1997)
[15] [W.B. Langdon, "*The evolution of size in variable length representations*", in Proceedings of the 1998 IEEE International Conference on Evolutionary Computation (IEEE Press, Piscataway, NJ, 1998)
[16] Sean Luke, "Modification point depth and genome growth in genetic programming", Evolutionary Computation, vol. 11, no. 1, pp. 67-106, 2003.
[17] Mataric, Maja J, "A Distributed Model for Mobile Robot Environment-Learning and Navigation" MIT Artificial Intelligence Laboratory technical report AI-TR-1228. May 1990R. Arulmozhiyal and K. Baskaran, "Implementation of a Fuzzy PI Controller for Speed Control of Induction Motors Using FPGA," *Journal of Power Electronics*, vol. 10, pp. 65-71, 2010.