COSC 4P77

Final Project

Improvements to lilgp

Genetic Programming System

Adam Hewgill

2400083

Introduction

The goal of this project is to improve a freely available genetic programming (GP) system so that it becomes a more useful tool for future researchers. *lilgp* is a widely used GP system that is free and quite robust. It has been around since 1995 and has been slowly but continually improved by both the original authors and external users of the system. There now exists multithreading, a windows compatible version, strongly typed GP, and constrained GP which link off of the main *lilgp* site[1]. The most recent version of *lilgp* is 1.1 beta which contains the multithreading update for multiple populations on multiple CPU's, this release is dated September 1998.

Strong typing was added to the kernel by Sean Luke in 1997 at which time some other multithreading bugs were fixed and rolled into a version of *lilgp* called "Sean's Patched *lilgp* Kernel"[2] which is recognized as an independent kernel or a branch off of the original *lilgp* kernel which now stands alone. The two versions are both well tested since they have been used for several years now by many researchers.

The improvements the have been made are applied to the strongly typed version of the kernel since one of the improvements is directly related to strong typing. Also the strongly typed version essentially contains the non-typed version within it as a special case when using only one type. So it seems logical to make updates to the highest order version so that everyone can use them, not just those who only want to deal with problems needed only one type.

The first and most major improvement to the kernel is the update to the tree growth methods. Initially the grow and full tree generation methods were limited due to their simplicity which meant that every type had to contain at least one function and terminal. This would cause a great deal of bloat in the tree since the author needed to create useless functions and terminal that the tree generators could use to fill the tree up to its maximum depth. Now these useless functions and terminals are no longer necessary so the avoidable useless bloat is reduced leaving only the bloat created by evolution for protection.

The second improvement was to add a method by which functions and terminals could be turned on and off before a run simply by setting their state in the input file. This removed the need to recompile to change the function set.

Thirdly, the random number generator which is built into the kernel is exposed to the user via calls to `random_seed( int )`, `random_int( int )`, and `random_double( )`. These where very useful functions since `random_int ( int )` returns a number between zero and the passed integer including zero and `random_double( )` returns a double between zero and one including zero. The problem was that if you were to use these number generators you would change the number sequence so while the same seed will give you the same run changing the app side in anyway relating to the random calls would totally change the resulting kernel random numbers. The other problem is if the user wants a set of objects to always be the same then the changing of the random seed would be best circumvented. This problem is fixed by componentizing the random number generator so that everything necessary is contained in a structure which all the calls for number pass as their first parameter. The final changes are much smaller bug fixes and beautification changed which make the kernel better overall.

Strongly Typed Tree Generation

Introduction

By default *lilgp* contains two methods for tree generation: grow trees and full trees. Both algorithms are recursive tree building functions that take the functions and terminals provided by the user and form them into a random tree that becomes one individual in the population. The pseudo code for each is given in listing 1. The pseudo code for each is very similar and is very simple to implement in a real system. However, the simplicity of the algorithm is mainly generated by two necessary conditions that reduce tree generation complexity immensely. For these algorithms to work it is necessary to have at least one terminal and function for every type.

Old Algorithm Problems

The terminal necessity is evident from the first if statement in both algorithms. When the algorithm reaches the maximum depth of the tree it must select a terminal for whatever type it is currently in so since it is possible to be in any type at this point every type must have a terminal. If the type set is augmented with an extraneous terminal expressly for the purpose of satisfying the tree generation constraints then the type set is changed to a different type set. This not only adds bloat if the form of branches which are terminated with placeholder terminals but changes everything to do with the search space being covered. When the GP system now does a run and evolves individuals there is a huge change to the search paths through the search space that are taken. In addition the search space is broadened to include the new terminal(s). Overall the effect weakens evolution and the quality of the trees produced by the system.

```
Old GROW tree algorithm

grow_tree ( int depth, int return_type )
{
  // Reached maximum depth of tree
  if depth is zero
    randomly select terminal of return_type

  // Select either function or terminal
  randomly select something of return_type

  // Continue growing tree unless terminal
  if selected a function
    for each child
      grow_tree ( depth - 1, my_type )
}
```

```
Old FULL tree algorithm

full_tree ( int depth, int return_type )
{
  // Reached maximum depth of tree
  if depth is zero
    randomly select terminal of return_type

  // Select only a function
  randomly select function of return_type

  // Continue growing tree
  for each child
    grow_tree ( depth - 1, my_type )
}
```

Listing 1

The function necessity results in the same problems mentioned above for the extraneous terminals. When the full tree algorithm is not at the maximum depth for a tree it must continue to select functions until it does reach the bottom. However, if your type set has a dead-end type such that if entered there exists only terminals and no functions then the tree generator becomes stuck and can not extend the branch to the maximum depth. This leads to the necessity of having a function for every type so the full tree can use it to extend the branches to the maximum depth. This also changes the search space if unfavorable ways.

The two issues outlined above have been dubbed the "maximum depth cutoff" problem and the "full tree fill" problem respectively for use in this paper. "Maximum depth cutoff" and "full tree fill" are essentially the same problem but occur at different points in the tree generation process as was discussed previously. The remedy for these problems is to increase the complexity of the algorithms by adding the ability to do backtracking when one or the other of the two problems crops up in tree generation.

New Algorithm

The new algorithm is based on the original with only the problem outlined above fixed. Nothing has been added to improve the quality, shape or size of the resulting population of trees. See listing 2 and 3 for the new algorithm pseudo code.

The first necessary change was to the function definitions themselves. Before, the two tree generators were void functions (procedures) that never failed due to the constraints placed on the type set. Now we wish to tell the parent level of the recursion that a child sub-tree can not be created to fit this branch or that the maximum possible sub-tree is too short for this branch. This is implemented by making the function return an integer that is one on error and zero otherwise. This will let the parent know if a different function should be selected leading to the backtracking behavior desired.

In the grow tree algorithm there are two places where terminals can be selected for use in the tree. The first is when the depth reaches 0 (at bottom of tree) and the second is if the tree is not at the bottom but a terminal is chosen randomly so terminal selection is spread out. To alleviate this, the very first thing done in the new grow tree algorithm is to decide whether a terminal or function is selected probabilistically. This has nothing to do with the maximum depth cutoff and simply provides a way of breaking the terminal selection part off and having it by itself at the beginning of the algorithm. This is desirable for continuity between the two tree generation algorithms and to reduce code duplication now that the algorithm is more complex. The decision is made by calculating the percent of nodes which are functions and then testing if a random number [0..1) is greater or equal, which signifies selection of a terminal or less than, which signifies selection of a function. The first if statement then changes from "if depth is zero" to "if depth is zero or terminal selected" which covers all terminal selection possibilities.

The solution to the "maximum depth cutoff" problem is simple to solve now that the algorithm includes a framework for backtracking. When the first if statement is entered a check is done immediately as to whether a terminal exists for the current return type. This will always be true in the case of the grow algorithm where a terminal is selected probabilistically (as discussed in the previous paragraph). If the check fails and we have no terminals to place in the tree then we have reached a point where it is impossible to generate the sub-tree to attach to the parent so a one is returned forcing the parent to try again. Adding a terminal to a branch is the base case for recursion in both algorithms and since the new algorithm maintains the base case behavior that was present before, the recursive algorithm is still valid. If there is a terminal to add to the tree then one is selected randomly and the recursion ends normally.

The solution to the “full tree fill” problem is also simple now that we have backtracking. After we pass through the section where a terminal would be selected we must definitely select a function to add to the tree. So a check is done to see if there are any functions that are of the current return type. If none exist then a one is returned and the parent is forced to try again. If there is a function for the type then one is selected randomly and attached to the tree.

```
New GROW tree algorithm

int grow_tree ( int depth, int return_type)
{
  // Handle the terminal selection process
  decide whether terminal or function selected probabilistically
  if depth is zero or terminal selected
    // Handle the “maximum depth cutoff” problem
    if terminal doesn’t exist
      return 1
    randomly select terminal
    return 0

  // Handle the “full tree fill” problem
  if function doesn’t exist
    return 1

  // Handle backtracking and sub-tree creation
  do
    // Handle function exhaustion case
    if no more functions to try
      return 1

    randomly select function

    // Try to generate the children
    for each child
      grow_tree ( depth - 1, my_type )

  while any child not valid

  // This sub-tree is correctly generated
  return 0
}
```

Listing 2

The final change in the algorithm is the implementation of the backtracking behavior when generating sub-trees to attach to the tree. After selecting a function to use for the current node a loop is entered, which has an exit condition stating that all the children of the current node have been filled correctly. If this loop exits then we know this sub-tree is valid and we can inform the parent of that fact by returning a zero.

Inside the loop the algorithm implements the backtracking behavior. First a check is done to see if we have run out of function to try. If this happens then we have to tell the parent that a sub-tree cannot be created using the functions available so a one is returned. The next step is to randomly select one of the remaining functions and then attempt to generate the children to attach to the function parameters. If any of the children return one then the loop with not exit and a different function will be selected until finally the loop exits or the function set is exhausted.

```
New FULL tree algorithm

int grow_tree ( int depth, int return_type)
{
  // Handle the terminal selection process
  if depth is zero
    // Handle the “maximum depth cutoff” problem
    if terminal doesn’t exist
      return 1
    randomly select terminal
    return 0

  // Handle the “full tree fill” problem
  if function doesn’t exist
    return 1

  // Handle backtracking and sub-tree creation
  do
    // Handle function exhaustion case
    if no more functions to try
      return 1

    randomly select function

    // Try to generate the children
    for each child
      grow_tree ( depth - 1, my_type )

  while any child not valid

  // This sub-tree is correctly generated
  return 0
}
```

Listing 3

Function Set Updater

When using *lilgp* all of the functions and terminals that are to be used in tree generation are placed in a table and passed into the kernel from the user’s code. This table allows the user to setup many properties for each node including: implementing function, number of parameters, string name, type of node, and the return and parameter typing information. This structure is very rigid and can only be changed by editing the code then recompiling the system with the new nodes.

During previous research work it became necessary to test whether certain nodes in the language were in fact useful or extraneous. So runs were performed using a certain function set providing results that were then compared with runs that didn’t contain a certain node. Over many runs it would be decided whether the node being checked was improving, hindering or doing nothing to the results of the runs. However, doing these tests required that between experiments the function set table was modified and the GP system recompiled.

This problem was circumvented by writing an addition to the kernel which runs a function set updater on all the user function sets. By default every node in the function set is assumed to be enabled. In order to disable a function its string name is used in a line in the input file which is set to zero. So if you have a function with string “AND” then in the input file you would add `app.use_AND = 0` to disable the function and = 1 to enable it again. Recompilation is no longer necessary for enabling and disabling functions.

The function set updater has been fully integrated into the kernel such that it runs automatically every time the GP system is run. So the user can treat it like another feature of the kernel and can subsequently ignore it when it's not needed. The updater prints out a message at the beginning of the run that describes which nodes are removed from which function set.

In order for the updater to work correctly the string name used for each function must be unique over every function set. This is because when a node is disabled using the input file, every other node with the same string name will also be disabled causing undesired results. This uniqueness is not a new constraint on the function sets because the kernel already required it for check pointing to work. However, this constraint was never mentioned by the authors in the supporting documentation (that I could see). This constraint was stumbled upon when trying to find a method of exporting and importing both individuals and whole populations so that separate programs could use the generated individuals. This is described further in the future work section below.

New Random Number Framework

The kernel has its own built-in random number generator which operated through three functions. `random_seed( int )` initializes all of the associated variables with the seed provided in the parameter. This prepares the random generator for queries for random values. `random_double( )` retrieves a random floating point value from [0..1). Finally, `random_int( int )` retrieves a random integer value from [0..int). These calls for random numbers change the state of the random number generator. The random number generator has a period where upon it starts at the beginning again and repeats exactly the set of numbers received after seeding. I believe the period is very long for this implementation so looping should never occur. The benefit of these pseudo random number generators is that for a given seed value the sequence of generated numbers is always the same. However order of calls to the generator matters since using the same seed but a different number of calls or a different order of calls will not result in the same sequence of number going to the same places as previously which changes the output of the GP system.

When the user uses the kernels random number generator to get random values, the problem of changing order will occur. So what can happen is that the user wishes to perform the same run as in previous experiment but cannot due to changes in their code which completely change the output of the GP system solely based on number order (almost like using a different seed). This problem is undesirable in two ways: static randomly generated information changes with seed and changing app code changes run output.

The first problem occurs when the user wishes to generate random data that should stay constant over the various runs of the GP system. This information could be an item list with weights and values or food pellet locations for an ant simulation. The best way to do this is to seed the random number generator with a set value then generate the list. Previously this was done by using a separate platform specific random number generator or seeding the kernel's random number generator then reseeding it back to the value in the input file.

The second problem occurs if the app uses the kernel's random number generator. When the app uses it, it advances its state so after a run is completed and the output received it is possible to generate the same results again. This is because the app takes the same states as before and the kernel produces the same output. However, if the app were to change slightly so a different number of random numbers are taken then the kernel will receive and entirely different set of numbers and the output will be much different for the same seed. This is an undesirable result because as long as nothing fundamentally changes the GP system should produce the same results with a given seed.

To remedy these problems and give the kernel and app a better way to use build-in random numbers, the random number generator was componentized. Now all the variables associated with the current state of the random number generator are packaged into a structure that can be created by the user as a variable anywhere in their code. The kernel now has its own copy of this structure and it uses it exclusively throughout the code. Now the app and kernel can uses entirely separate built-in random number generators.

This is done by changing the three functions which access random numbers. A parameter was added which specifies the structure containing the random number generator which should be seeded or used to get a number from. In addition a new function is added which cleans up the structure when it is no longer needed it is called: `random_destroy ( randomgen* )`. The other three function now look like this: `random_seed( randomgen*, int )`, `random_double( randomgen* )`, and `random_int( randomgen*, int )`.

If for some reason the old method is preferred it can still be accessed via calls to the kernels random number generator instead of a local app one. This is done with this call: `random_double( globrand )`.

Bug-Fix to Random Algorithm

In the actual random number algorithm there is a very small bug which causes segmentation faults for certain seed values. For some reason the algorithm truncates values which causes some generated values to be out of bounds of the internal random number state arrays. This is remedied by simple rounding instead of truncating the value.

Now there seem to be no problems with any seed that was tested and in addition no change was made to the sequence of number generated by the algorithm for seeds which previously caused no problems.

Bug-Fix to input parameter NUM_TIMES

There was a small bug in the input file parser which caused the num_times parameter to not be accepted. num_times is used when generating trees via crossover and mutation. It is the number of times to retry generation of an individual if it is invalid.

If a crossover or mutation causes a tree to become invalid because of node number or depth limits then the tree by default is thrown away and the parent is used instead. This causes the replication rate to become very high over a generation and will cause degradation to the results of the GP system; this was how Koza initially outlined it. In order to force the system to continue trying to make a child you must add a section to the breeding parameter lines in the input file.

```
breed[1].operator = crossover, select=(tournament, size=5), keep_trying=on.
```

The keep trying addition on the end forces the retrying outlined above. In addition you can specify how many times to retry before giving up and using the parent. This is specified in the num_times section where N is the number of times to retry.

```
breed[1].operator = crossover, . . . , keep_trying=on, num_times=N
```

Minor Changes

Initially the old kernel used an obsolete version of two functions relating to multi-threading. These functions worked on the target system that it was initially developed on but not on operating systems with more recent version of the POSIX thread library. Now the user can specify if they are using an older version of the POSIX library by defining `PTHREAD_1.X` in the GNUmakefile in the CFLAGS section as `-DPTHREAD_1.X`. This incompatibility was initially discovered by Cale Fairchild while working with the multi-threading turned on.

Checkpoint compression is a very little used function of the kernel which is configured in the GNUmakefile. It is initially set to use VFORK to spawn the compression algorithm but for wider compatibility the default was changed to SYSTEM which works on all system.

When using an older compiler with strong checking for variables not being used and other trivial code problems, the compiler will result in a bunch of warnings. All of these that came up with our compiler have been resolved.

Future Work

The current implementations of the tree generation functions are simple naive backtracking algorithms. With a more complicated type set it could be very costly with the worst case running time being for a tree that is impossible to grow given the function set. It would be more efficient to keep track of possible ending patterns that could be used once tree generation reaches a certain level. This could take the form of an automatically generated table that could be queried at every level to check if a forced function or terminal selection should be made so the tree is possible to grow. This would greatly reduce the necessity for backtracking and on the whole speed up tree generation drastically.

The documentation provided with the original un-typed version of *lilgp* is moderately well done but has not been expanded by the author of the strongly typed version. So now the documentation is scattered over different websites and not very easy for a new user to assimilate. The task of compiling and updating the documentation with all new features and updates is the next step before releasing this as a new version of the *lilgp* system for other researchers to use.

The programs that are evolved using the GP system should be available for use outside the system itself otherwise the results are purely academic in nature. *lilgp* already has the built-in capacity to export a population and then to import it again. This feature is called check pointing because if your application takes a very long time to run and you are forced to stop half way you can begin again at a check point and not at the beginning. This could save you a great deal of time and can be used to test various changes to the standard GP algorithms. For example if you have a check point that only has ten more generations to perform then you can test if a change to the algorithm improves the tail end of the evolution process where usually the evolution has flattened out considerably. This change can be a tail end only change such as increased mutation and decreased crossover for the last ten generations, etc. A library needs to be produced so that when imported into an application a check point file can be loaded and the individual therein can be used in that application as they are in *lilgp* itself.

Kernel Comparison using Mushroom Clairvoyant

The mushroom poisonous/edible detector for the standard problem (dubbed mushroom clairvoyant) was created originally using the old kernel and has been ported to the new kernel. When running tests to compare if the kernel updates are effective both version are run and the output statistics are compared. It was found quite quickly that the running time of the GP system was an unreliable measurement due to how the function set was structured. Some key functions in the GP language are function expressions which can possibly skip their other sub branches if a certain value is returned from a child. In the old system the function expressions were AND and OR and the extraneous Boolean terminal was FALSE so this reduced a great deal of the subtree computation. In the new system there are no longer and extraneous FALSE terminals so the saving of the function expressions is greatly reduced. This is evident by observing the output trees which contain for than 50% extraneous FALSE terminals in the old system. The only noticeable difference time wise is a slight (almost negligible) increase in the time taken to create the initial population.

Overall the results using the new system were better than the old but not by much in this example due to the very slim amount of room for improvement. The type set for this problem uses Boolean for most of the tree except the leaves and the node immediately above the leaves. There is a function which converts the terminals into Boolean so that every tree must have these at the leaf ends of the branch. This is a rather simple type set problem to overcome via backtracking so the algorithm is not necessarily stress tested.

For more specific data as to the improvement from old to new see the following table (Table 1). The training set consisted of threehundred positive and three hundred negative training examples. Each run lasted for 50 generations and then all the individuals are tested with the entire set to get the final scores that are listed in the table. The score posted is the best score for the run.

Seed	Old Score	New Score
1	97.34	**98.72**
73	98.13	**98.82**
12367	98.62	**99.06**
9634	98.72	**99.02**
486978	98.62	**99.70**
4	98.23	**99.80**
Avg	98.286	**99.187**

Table 1

Conclusions

The updates made to the kernel are effective for the problem they were tested on which is a general type of the problems it is designed to handle. It is safe to assume that they changes will be effective in all cases of a similar difficulty level and further research is needed to decide whether complex type sets are also improved.

References

[1] *lilgp* 1.1 Beta homepage
http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html

[2] Sean's Patched *lilgp* Kernel
http://www.cs.umd.edu/users/seanl/gp/patched-gp/