Genetic Programming & Bloat

Jiří Kubalík

Czech Institute of Informatics, Robotics and Cybernetics
CTU Prague

http://cw.felk.cvut.cz/doku.php/courses/a0m33eoa/start

Bloat

Bloat – an uncontrolled program growth without (significant) return in terms of fitness [Poli08]. The bloat has significant practical effects:

- slows the evolutionary search process as large programs can be computationally expensive to evolve,
- large programs can be hard to interpret,
- large programs can exhibit poor generalization,
- consumes memory,
- can hamper effective breeding.

High-level general explanation of bloat [Luke06]: Adding material to a tree is more strongly correlated (or less negatively correlated) with fitness improvement than removing material from the tree is.

- However, the question is how or why this correlation arises?

Dynamics of GP selection, breeding and evaluation are complex $\Longrightarrow$ though, there have been many theories proposed to explain various aspects of bloat, there is still **no single unifying theory of code bloat**.

Theories of Code Bloat Based on Introns

Introns – regions of code that do not contribute to an individual's function (do not contribute to the fitness).

1. **Inviable code** – a particular form of intron that can be replaced with any code which can possibly contribute to the individual's function. This is due to the presence of so-called **invalidator**, a structure in the tree that nullifies the entire intron's effect.

   Inviable code examples

   - (`and` *false inviable*), (`if` *false inviable executed*), (`if` *inviable* `a0 a0`) where
     - the invalidator *true* can be created as (`not` (`and a0` (`not a0`)))
     - the invalidator *false* can be created as (`and d1` (`not d1`))
   - (`* 0` *inviable*), (`% 0` *inviable*) where the invalidator '0' can be created as (- x x)

2. **Unoptimized code** – code regions that do not contribute to an individual's function, but can be replaced with code which does contribute or an optimized form of the code. Examples:

   - (`not` (`not` (`not` (`not` *foo*))))
   - (`and d1 d1`)

Theories of Code Bloat Based on Introns

Hitchhiking

- Introns are hitchhikers that adjoin highly fit building blocks.
- There is no real need to get rid of hitchhikers that do not damage fitness of the program.

Theories of Code Bloat Based on Introns

Hitchhiking

- Introns are hitchhikers that adjoin highly fit building blocks.
- There is no real need to get rid of hitchhikers that do not damage fitness of the program.

Defense Against Crossover

- Genetic operators seldom create better individuals than their parents.
- Introns provide code where changes will not affect fitness.

Theories of Code Bloat Based on Introns

Hitchhiking

- Introns are hitchhikers that adjoin highly fit building blocks.
- There is no real need to get rid of hitchhikers that do not damage fitness of the program.

Defense Against Crossover

- Genetic operators seldom create better individuals than their parents.
- Introns provide code where changes will not affect fitness.

Removal Bias

- To maintain fitness, the **removed subtree** must be contained within the inviable region – they cannot be deeper than the inviable subtree.
- On the other hand, the **inserted subtree** can have any size.

Non-Intron Theories of Code Bloat

Fitness Causes Bloat

- There are many more longer ways than shorter ways to represent programs of a given fitness.
- Over time, GP samples longer and longer programs simply because there are more of them.

Non-Intron Theories of Code Bloat

Fitness Causes Bloat

- There are many more longer ways than shorter ways to represent programs of a given fitness.
- Over time, GP samples longer and longer programs simply because there are more of them.

Modification Point Depth

- Small changes are less likely to be disruptive, so there is a preference for deeper modification points, and consequently a preference for larger trees.
- The larger the individual, the deeper its modification nodes can be, so large parents have and advantage over small parents.

Non-Intron Theories of Code Bloat

Fitness Causes Bloat

- There are many more longer ways than shorter ways to represent programs of a given fitness.
- Over time, GP samples longer and longer programs simply because there are more of them.

Modification Point Depth

- Small changes are less likely to be disruptive, so there is a preference for deeper modification points, and consequently a preference for larger trees.
- The larger the individual, the deeper its modification nodes can be, so large parents have and advantage over small parents.

Crossover Bias

- The average program length in the population is not changed by the subtree crossovers.
- So, there is a bias of the crossover operators to create many small, and unfit, individuals.
- When these small unfit individuals compete for breeding, they are always discarded by selection in favor of the larger ones.

Lexicographic Parsimony Pressure Method

Idea: Two objectives with fixed priorities assigned are used in the selection procedure

- fitness – a primary objective,
- tree size – a secondary objective.

Realization: Uses a modified tournament selection rule of the form

A) An individual is considered superior to another if it is better in fitness.

B) If they have the same fitness, then an individual is considered superior if it is smaller.

C) If they have the same fitness and they are of the same size, the superior individual is determined at random.

Lexicographic Parsimony Pressure Method

Idea: Two objectives with fixed priorities assigned are used in the selection procedure

- fitness – a primary objective,
- tree size – a secondary objective.

Realization: Uses a modified tournament selection rule of the form

A) An individual is considered superior to another if it is better in fitness.

B) If they have the same fitness, then an individual is considered superior if it is smaller.

C) If they have the same fitness and they are of the same size, the superior individual is determined at random.

Characteristics:

- Non-parametric method – nothing to tune.
- Works well only in environments which have a large number of individuals with identical fitness.

  Otherwise, the branch B) of the tournament operator would not be activated with a sufficient frequency.

  To overcome this inefficiency, two modifications based on grouping individuals of similar fitness into buckets with the same quality were proposed.

Lexicographic Parsimony Pressure Method: Direct Bucketing

Realization: The number of buckets, b, is specified beforehand, and each is assigned a quality rank from 1 to b (the bucket with rank 1 contains the worst-fit individuals).

1. The population of size p is sorted by fitness.
2. The bottom $\lceil p/b \rceil$ individuals are placed in the worst bucket.

   All individuals remaining in the population with the same fitness as the best individual in the bucket are placed in the bucket as well.

   This is to guarantee that all individuals of the same fitness fall into the same bucket (they have the same rank).
3. The same procedure is used to fill in the second worst bucket, the third one etc.

   This continues until there are no individuals in the population.
4. The fitness of each individual is set to the rank assigned to the bucket holding it.

Lexicographic Parsimony Pressure Method: Direct Bucketing

Realization: The number of buckets, b, is specified beforehand, and each is assigned a quality rank from 1 to b (the bucket with rank 1 contains the worst-fit individuals).

1. The population of size p is sorted by fitness.
2. The bottom $\lceil p/b \rceil$ individuals are placed in the worst bucket.

   All individuals remaining in the population with the same fitness as the best individual in the bucket are placed in the bucket as well.

   This is to guarantee that all individuals of the same fitness fall into the same bucket (they have the same rank).
3. The same procedure is used to fill in the second worst bucket, the third one etc.

   This continues until there are no individuals in the population.
4. The fitness of each individual is set to the rank assigned to the bucket holding it.

Characteristics:

- It has the effect of trading off fitness differences for size.
- The larger the bucket, the stronger the emphasis on size as a secondary objective.
- The topmost bucket with the best-fit individuals can hold fewer than $\lceil p/b \rceil$ individuals.

Lexicographic Parsimony Pressure Method: Ratio Bucketing

Realization: The buckets are proportioned, so that low-fitness individuals are placed into larger buckets than high-fitness individuals. A parameter of the method is the bucket ratio $1/r$.

1. The population of size p is sorted by fitness.
2. The bottom $\lceil 1/r \rceil$ fraction of individuals are placed into the worst bucket.

   All individuals remaining in the population with the same fitness as the best individual in the bucket are placed in the bucket as well.
3. The same procedure is used to fill in the second worst bucket with the bottom $\lceil 1/r \rceil$ fraction of the remaining population, etc.

   This continues until every individual of the population has been placed in a bucket.
4. The fitness of each individual is set to the rank assigned to the bucket holding it.

Lexicographic Parsimony Pressure Method: Ratio Bucketing

Realization: The buckets are proportioned, so that low-fitness individuals are placed into larger buckets than high-fitness individuals. A parameter of the method is the bucket ratio $1/r$.

1. The population of size p is sorted by fitness.
2. The bottom $\lceil 1/r \rceil$ fraction of individuals are placed into the worst bucket.

   All individuals remaining in the population with the same fitness as the best individual in the bucket are placed in the bucket as well.
3. The same procedure is used to fill in the second worst bucket with the bottom $\lceil 1/r \rceil$ fraction of the remaining population, etc.

   This continues until every individual of the population has been placed in a bucket.
4. The fitness of each individual is set to the rank assigned to the bucket holding it.

Characteristics:

- As the remaining population decreases, the $\lceil 1/r \rceil$ fraction decreases as well.
- Higher-ranked buckets hold fewer individuals than lower-ranked buckets.

  Thus, the tree-size comparisons are more frequently applied to low-fitness individuals than high-fitness individuals.
- Both bucketing schemes require user-specified bucket parameters b or r that determines how strong an effect of parsimony can have on the selection procedure.

Linear Parametric Parsimony Method

Idea: Parsimony pressure methods consider **size as part of the selection process** – a fitness of the program is a function of its quality and size. A fitness of a program is decreased by an amount that depends on its size. The intensity with which bloat is controlled is determined by a parameter called *parsimony coefficient*.

- If it is too small then the control of bloat is not effective.
- If it is too large then the minimization of tree size will become a primary target and fitness will be ignored.

Linear Parametric Parsimony Method

Idea: Parsimony pressure methods consider **size as part of the selection process** – a fitness of the program is a function of its quality and size. A fitness of a program is decreased by an amount that depends on its size. The intensity with which bloat is controlled is determined by a parameter called *parsimony coefficient*.

- If it is too small then the control of bloat is not effective.
- If it is too large then the minimization of tree size will become a primary target and fitness will be ignored.

Realization:

- Linear Parametric Parsimony Method treats the individual's size as a linear factor in fitness

  $$g = x \cdot f + y \cdot s$$

  where the parameters x and y weight contributions of raw fitness f and the size s to the final fitness g, that is to be minimized.
- Linear Parametric Parsimony Method with a limit applies the size component only if s is greater than some specified limit z. Then

  $$g = x \cdot f, \text{ if } s \leq z$$
  $$g = x \cdot f + y \cdot (s - z), \text{ otherwise.}$$

Linear Parametric Parsimony Method

Characteristics:

- A user must set up the *parsimony coefficient* so that it optimally defines f as being worth so many units of s.
  - This can be difficult when the fitness assessment procedure is nonlinear.

    Assume a situation where a difference between 0.9 and 0.91 in raw fitness is much more dramatic than a difference between 0.7 and 0.9. Then the size can be given an advantage over the raw fitness when the difference in raw fitness is only 0.01 as opposed to 0.2.
  - Proper setting of the *parsimony coefficient* can be hard when the raw fitness values are converging late in the evolution procedure.

Dynamic Size Limits

Idea: A dynamic limit on the maximum size can increase or decrease during the run

- is applied to the depth or size of evolved trees
- $dynamic\_limit$ is initially set to a small value
- **a new individual who breaks this limit is discarded** and replaced with one of its parents, **unless it is the best individual found so far**.

  In this case, the individual is inserted to the population and the $dynamic\_limit$ is raised to match the depth of the new best-of-run.

Dynamic Limits: Procedure

initialize $dynamic\_limit$
for all newly created individuals
$depth_i =$ depth of individual
$fitness_i =$ fitness of individual
if $depth_i \leq dynamic\_limit$
accept individual
if($fitness_i$ is better than $best\_fitness$)
$best\_fitness = fitness_i$
if(($depth_i > dynamic\_limit$) and ($fitness_i$ is better than $best\_fitness$))
accept individual
$best\_fitness = fitness_i$
$dynamic\_limit = depth_i$

Dynamic Limits: Heavy Variants

Heavy variant allows the $dynamic\_limit$ to decrease its value in case the depth of the new best individual becomes lower than the current limit. The value of $dynamic\_limit$ cannot drop below the initialization value.

Individuals, already present in the population that break the new limit become **illegals**:

- Illegals are allowed to remain in the population and breed children.
- If an individual has illegal parents then it cannot be deeper than its deepest parent.

Dynamic Limits: Final Remarks

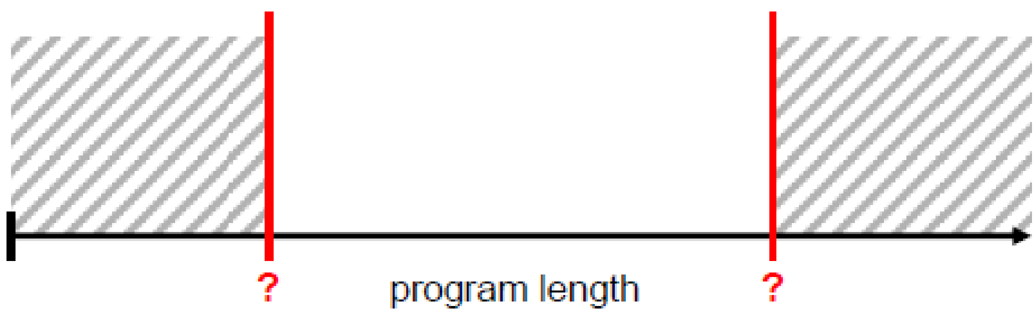
- Uses a $dynamic\_limit$ that can be raised or lowered during the search process, depending on the best solution found so far.
- The depth limit performs very well across various problems, achieving high quality solutions using significantly smaller trees.
- The size limits did not perform so well.
- Does not require specific genetic operators, modifications in fitness evaluation or different selection schemes, nor does it add any parameters to the search process.
- Works even better when coupled together with other techniques such as the Lexicographic Parsimony Pressure.

Operator Equalisation

Should have rather been called "Program Length Equalisation".

Motivation: To control a distribution of program lengths in the population by biasing the search towards the desired lengths. Too small and excessively large programs are eliminated from the population.

- Too small programs – very likely to be of poor quality, so would be discarded by selection in favor of the large ones.
- Too large programs – beyond a certain program length, the distribution of fitness converges to a limit. So, there is no need for large programs if they do not bring significantly better fitness.



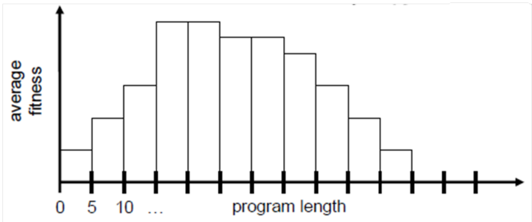
Operator Equalisation: Histograms

Concept of histograms

- bin width – the range of lengths that fall into the bin.

$$b = \left\lfloor \frac{l-1}{bin\_width} \right\rfloor + 1,$$

where b is the number of the bin to which the program of length l is assigned.



- bin capacity – the number of programs allowed within.

The population is biased towards a desired target distribution by **accepting or rejecting each newly created individual** into the corresponding bin in the population.

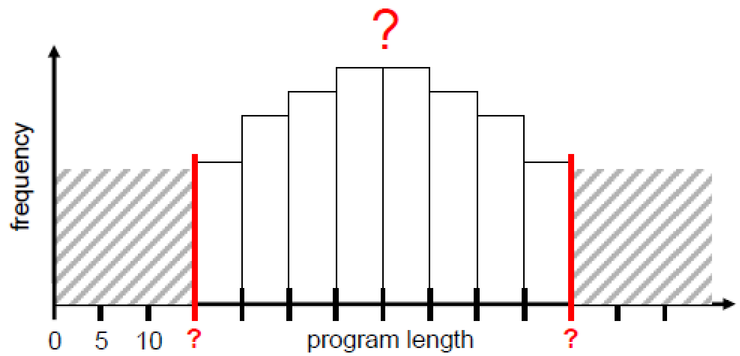
Operator Equalisation: Two Concepts

Static

- fixed number of bins,
- fixed predetermined target distribution.

Dynamic

- variable number of bins,
- target distribution self adapted every generation.



Dynamic Operator Equalisation: Calculating the Target Distribution

Target number of individuals in bin b is proportional to the average fitness of individuals within the bin, calculated as

$$binCapacity(b) = round(n \times (\overline{f}_b / \sum_i \overline{f}_i))$$

where

- $\overline{f}_i$ is the average fitness of the individuals in bin i, and
- n is the number of individuals in the population.

The target is updated every generation.

Bins with better average fitness will have higher capacity – allowing the population to sample regions where the search proved to be more successful.

Dynamic Operator Equalisation: Following the Target Distribution

First, a length of the offspring and its corresponding bin is identified.

Rule for accepting/rejecting newly created offspring in the bin

1. If the bin already exists and is not full

    then the offspring is accepted.

2. If the bin does not exist yet and the fitness of the offspring is the new best-of-run value

    then the bin is created to accept the new individual.

    Any other non-existing bins between the new bin and the target boundaries also become available with capacity for only one individual each.

3. If the bin exists but is already at its full capacity and the offspring is the new best-of-bin one

    then the bin is forced to increase its capacity and accept the individual.

4. Otherwise the new individual is rejected.

The **dynamic creation of new bins** and allowing the **addition of individuals beyond the bin capacity** allows overriding of the target distribution by biasing the population towards the lengths where the search is having high degree of success.

Dynamic Operator Equalisation: Final Remarks

- Uses concept of a histogram as a target distribution of programs length in the population
- Dynamic self-adaptive target distribution
- Variable number of bins

Reading

[Poli08] Poli, R., Langdon, W., McPhee, N.F.: *A Field Guide to Genetic Programming*, 2008.

[Luke06] Luke, S. and Panait, L.: A Comparison of Bloat Control Methods for Genetic Programming. *Evolutionary Computation*, Volume 14 Issue 3, 2006.
`http://portal.acm.org/citation.cfm?id=1182892.1182897`