# Genetic Programming & Bloat

Jiří Kubalík
Department of Cybernetics, CTU Prague

Substantial part of this material is based on

Sean Luke and Liviu Panait: A Comparison of Bloat Control Methods for Genetic Programming,
See http://portal.acm.org/citation.cfm?id=1182892.1182897

Sara Silva: Handling Bloat in GP,
See http://doi.acm.org/10.1145/2001858.2002146

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

# Contents

- Bloat and several theories on why bloating occurs
- Classification of bloat control methods
- Description of selected bloat control methods
- Comparison of bloat control methods

# 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 cannot 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. Examples:
   - (`not` (`not` (`not` (`not` *foo*))))
   - (`and d1 d1`)

# Theories of Code Bloat Based on Introns

**Hitchhiking** – based on genetic algorithms, where unfit building blocks propagate in the population because they adjoin highly fit building blocks.

- There is no real need to get rid of hitchhikers that do not damage fitness of the program.

  Introns are hitchhikers in GP.

- The theory only suggests a propagation method.

  It does not explain why it is more likely that the introns become attached in the first place than to be removed eventually.

# Theories of Code Bloat Based on Introns

**Hitchhiking** – based on genetic algorithms, where unfit building blocks propagate in the population because they adjoin highly fit building blocks.

- There is no real need to get rid of hitchhikers that do not damage fitness of the program.
  Introns are hitchhikers in GP.
- The theory only suggests a propagation method.
  It does not explain why it is more likely that the introns become attached in the first place than to be removed eventually.

**Defense Against Crossover**

- Genetic operators seldom create better individuals than their parents.
- Offspring who have the same fitness as their parents have a selective advantage.
  Introns provide code where changes will not affect fitness.
- Inviable code was selected because it made it more difficult to damage the fitness of an individual through a crossover event (more inviable code results in a higher likelihood that crossover would occur in an inviable code region).

## Theories of Code Bloat Based on Introns

**Removal Bias** – branches (subtrees) added to parents are deeper on average than branched removed from parents.

The presence of inviable code provides regions where removal or addition of genetic material does not modify the fitness of the individual.

- 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** – when better solutions become hard to find there is a selection bias towards programs that have the same fitness as their parents.

- There are many more longer ways than shorter ways to represent the same program, so a natural drift occurs to longer programs. Beyond a certain program length, the distribution of fitness does not vary with size.
- Since there are more longer programs, the number of long programs of a given fitness is greater than the number of short programs of the same 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** – when better solutions become hard to find there is a selection bias towards programs that have the same fitness as their parents.

- There are many more longer ways than shorter ways to represent the same program, so a natural drift occurs to longer programs. Beyond a certain program length, the distribution of fitness does not vary with size.
- Since there are more longer programs, the number of long programs of a given fitness is greater than the number of short programs of the same fitness.
- Over time, GP samples longer and longer programs simply because there are more of them.

**Modification Point Depth** – there is a correlation between the depth of the modified node and its effect on the fitness of the offspring when compared to the parent.

- When a genetic operator modifies an individual, the deeper the modification point the smaller the change in fitness.
- Small changes are less likely to be disruptive, so there is a preference for deeper modification points, and consequently a preference for larger trees (removal bias).
- 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

### Crossover Bias

- Subtree crossover operators do not add to or remove from the population any amount of genetic code, they simply swap it between individuals.
- So the average program length in the population is not changed by the crossover.
- 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.

# Classification Bloat Control Methods

Bloat control is possible at different levels of the evolutionary process

- **Evaluation** – Parametric Parsimony Pressure, The Tarpeian
- **Selection** – Multi-objective Optimization, Special Tournaments
- **Breeding** – Special Genetic Operators
- **Survival** – Size/Depth Limits, Operator Equalization
- **Others** – Code Editing, Dynamic Fitness

# Comparison of Methods: Experimental Setup

Nine bloat control methods:

- The Tarpeian method
- Linear Parametric Parsimony Pressure
- Lexicographic Parsimony Pressure (Direct Bucketing)
- Lexicographic Parsimony Pressure (Ratio Bucketing)
- Pareto-based Multi-objective Parsimony Pressure
- Double Tournament
- Proportional Tournament
- The Waiting Room
- Death by Size

The methods are used in combination with a simple **Depth Limiting** method – it rejects children whose depth exceeds some maximal tree depth (originally, 17 was used), placing copies of their parents in the population in their stead.

Comparisons of the combination of depth limiting with nine bloat control methods with the plain depth limiting method alone were performed.

## Comparison of Methods: Experimental Setup

Four problems and their characteristics:

- **Symbolic Regression** $(x^4 + x^3 + x^2 + x)$ – small improvements can always be made through additions to the existing tree.
- **Artificial Ant** (Santa Fe trail) – strong relationship between nodes throughout the tree, where changes in the left part of the tree have a dramatic effect on the operation in the right part of the tree due to execution order.
- **11-Multiplexer** and **5-bit Even Parity** – both have large-sized solutions.
  - Multiplexer is generally the most difficult of the four problems.
  - Parity is among the easier ones.

**Measure of bloat** – the mean tree size of all individuals generated during the course of an experimental run.

# The Tarpeian Method

**Idea**: Some individuals with above-average size are made uncompetitive by assigning some very poor fitness.

Called after the *Tarpeian Rock* in Rome, which in Roman times was the infamous execution place for traitors and criminals. They would be led to its top and then hurled down.

**Realization**:

1. Before the evaluation process, new individuals with above-average size are assigned a very bad fitness with probability $W$.
2. These individuals are not evaluated further.
3. Evaluate remaining individuals.
4. Use tournament selection to select individuals that will take part in breeding.

# The Tarpeian Method

**Characteristics**:

- The individuals that are marked uncompetitive are not evaluated – this reduces the number of fitness evaluations calculated.

  **More samples of the solution space can be tried!**
- The method is overly aggressive – the big individuals are rejected by size before considering their fitness even though the uncompetitive individuals still have a very small chance of being selected thanks to the tournament selection.
- The method is very sensitive to parameter $W$.

  If it is high, the method will tend to reject an individual no matter how fit it is.
- The setting $W = 0.3$ was consistently good across all four problems.

## 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.

# Lexicographic Parsimony Pressure Method: Performance

## Plain Lexicographic Parsimony Pressure

- Successful on all problems but the symbolic regression.

  The symbolic regression is unusual in that occurrence of individuals of exactly the same fitness in the population is rare since small changes in fitness can be achieved by adding code fragments to the bottom of trees.

## Direct bucketing

- Successful on all four problem, but no single setting of the parameter $b$ that would be consistently good across all problems was found.

  $b \in \{25, 50, 100\}$ is good for the symbolic regression.

  $b = 250$ is the common setting for other problems.

## Ratio bucketing

- Nearly uniformly superior in any setting, considering $r = \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8}, \frac{1}{9}, \frac{1}{10}$.

# 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**:

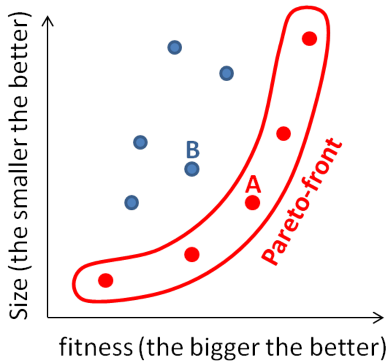
- A user must set up the *parsimony coefficient* so that it optimally defines $f$ as being worth so many units of $s$.
  - This can be difficult when the fitness assessment procedure is nonlinear.

    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.
- In the experiments, parameter $x$ was varied from $1/16$ to 65536, doubling $x$ each time.

  There are several settings ($x$=32, 64, 128, 512, 1024) for which the method was effective on all four problems.

# Pareto-Based Multi-Objective Parsimony Pressure Method

**Idea**: **Size and fitness are considered as separate objectives** in a multi-objective selection procedure. The goal is to find the **front** of solutions that are not dominated by any one else.
The **dominance** is defined so that an individual A is said to dominate individual B if A is as good as B in all objectives and is better than B in at least one objective.




It is important to use some mechanism for maintaining a diversity along the Pareto front in order to prevent the system from clustering the solutions in two extreme corners of the front

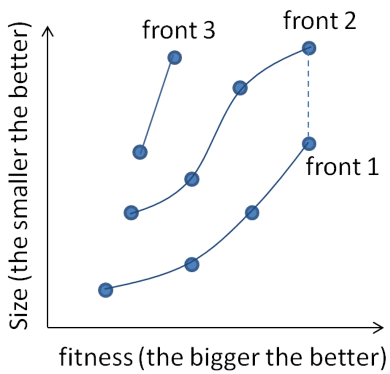
- highly-fit but too large programs,
- tiny but poorly-fit programs.

**Biased Multi-objective Parsimony Pressure** - tries to bias the search along the front towards the high-fitness end of the Pareto-front.

## Biased Multi-objective Parsimony Pressure

**Realization**: At each generation,

1. Fronts of non-dominated individuals are found within the population.
2. The following tournament-like selection is used to select individuals for breeding
   - with probability $F$ individuals are compared based solely on their fitness,
   - with probability $1 - F$ individuals are compared based on their respective front.

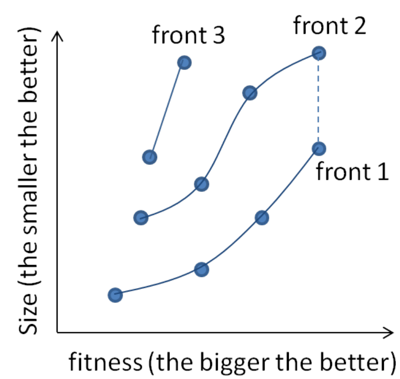



Ties are broken using the alternative objective. If both individuals are identical in both the fitness and front, one of them is chosen at random.

# Biased Multi-objective Parsimony Pressure

**Realization**: At each generation,

1. Fronts of non-dominated individuals are found within the population.
2. The following tournament-like selection is used to select individuals for breeding
   - with probability $F$ individuals are compared based solely on their fitness,
   - with probability $1 - F$ individuals are compared based on their respective front.




Ties are broken using the alternative objective. If both individuals are identical in both the fitness and front, one of them is chosen at random.

**Characteristics**:

- As $F$ decreases, the tree-size comparisons dominate in the selection, and vice versa.

  If $F = 1$, the methods approaches Lexicographic Parsimony Pressure.

  If $F = 0$, the method is entirely Pareto-based.
- The range of values for which the method is successful is 0.8-0.95, where 0.95 was the common value for which the method was successful on all four problems.
- For small values of $F$, populations clustered many small individuals near the extreme parsimony end of the Pareto-front, pulling resources from search for better fit solutions.

# Double Tournament Method

**Idea**: Selection procedure applies two layers of tournaments in series

- qualifier tournaments and
- final tournaments.

**Realization**:

1. Tournament contestants are chosen as winners of qualifier tournaments.
2. Winner chosen in the qualifier tournaments compete in the final tournament.

   Fitness objective can be used in qualifier tournaments and tree size objective in final tournaments or vice versa.

The selection is parameterized by

- fitness tournament size $F$,
- parsimony tournament size $D$,
- *do-fitness-first* – indicates whether fitness tournaments are used in the qualifiers or final tournaments.

# Double Tournament Method

**Characteristics**:

- $D$ should be smaller than 2, otherwise it puts too much pressure on parsimony.

# Double Tournament Method

**Characteristics**:

- $D$ should be smaller than 2, otherwise it puts too much pressure on parsimony.

  In order to permit $D$ to hold real values between 1.0 and 2.0 the following rule was implemented: Given two individuals, the smaller one wins the tournament with probability $D/2$, else the larger one wins.

  Thus, $D = 1$ is random selection, while $D = 2$ is the same as a plain parsimony-based tournament of size 2.

- An individual passes the double tournament if it is generally

  **low in size and high in fitness**.

- $D = 1.4$ was consistently superior on all four problems.

# Proportional Tournament Method

**Idea**: A proportion, $P$, of tournaments is based on tree size; remaining $1 - P$ tournaments are based on fitness.

**Realization**:

- Each tournament selection flips a coin to determine which objective to use.

**Characteristics**:

- Higher values of $P$ imply less of an emphasis on fitness, and vice versa.

  $P = 0.0$: All tournaments will select based on fitness.

  $P = 0.5$: Tournaments will select based on fitness or size with equal probability.

- An individual passes the proportional tournament if it is generally

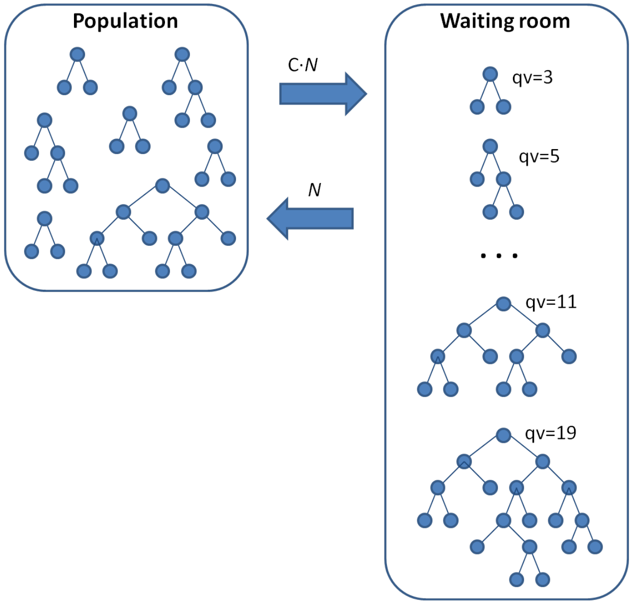
  **low in size or high in fitness**.

- $P = 0.2$ was consistently superior on all four problems.

# The Waiting Room Method

**Idea**: All newly created individuals are forced to wait in a *waiting room* for certain period of time – larger children wait longer than smaller ones – before they are permitted to enter the population.

**Realization**:




1. Create and evaluate $C \cdot N$ individuals, where $N$ is the population size and $C > 1$ is the ratio of child pool size to population size.
2. Add newly created individuals to the waiting room.
3. To each individual in the waiting room assign a queue value equal to the individual's size.
4. Remove $N$ children with the smallest queue values from the waiting room and place them to the new population.
5. All individuals in the waiting room have their queue values reduced by subtracting a value $A$, which is an *ageing parameter*.

   Effect of reducing the queue values is that even giant individuals may have a chance to be introduced to the population.

## The Waiting Room Method

**Characteristics**:

- Larger values of $C$ (smaller values of $A$) result in stronger parsimony.

  Success is almost entirely dependent on $C$.

- There are no setting for $C$ and $A$ which were consistently superior across all four problems.

- The method is strongly domain sensitive.

  $C$ have to be small for the 11-bit Multiplexer and the 5-bit Parity problems that are the problems with large tree solutions.

  $C$ should be larger for the Symbolic Regression and the Artificial Ant.

# Death by Size Method

**Idea**: At each step of an iterative process some individuals are selected to breed children, while other individuals are selected to die and be replaced by the new children.

**Realization**: Steady-state generational model, where at each generation selection of the parents and replacements is done as follows:

- Selection of the parents to breed is based on the fitness.

  Tournament selection with size $S$.
- Selection of the individuals to die is based on tree size (larger programs are more likely to die). Tournament selection with size $K$. This value should be very moderate ranging between 1.0 and 2.0.

  This is implemented so that given two competing individuals, the larger one wins with probability $K/2$, else the smaller one wins.

**Characteristics**:

- Under no combination of settings the method was superior on the 11-bit Multiplexer problem. Only one setting ($S = 7$, $K = 2.0$) was successful on the 5-bit Even Parity problem.
- As $K$ increases, larger individuals are selected to die, resulting in stronger bloat control.
- Steady-state evolution model tends to apply a strong selection pressure (large trees are produced very rapidly).

## Comparison of Methods: Experiment Setup

**General conclusion**: The combination of a method with depth limiting was nearly universally superior to either the method alone or depth limiting alone.

Double Tournament and Biased Multi-objective appeared the best across all problem domains when tuned to their optimal per-problem values.

**Question**: Which method with its single problem-independent setting is the best across all four problems?

## Comparison of Methods: Experiment Setup

Highlighted in blue are methods that have one or more settings with which they are superior to plain depth limiting across all problems.

- shown are the best settings with the lowest mean tree size of run averaged over all four problem domains.

good

| Method | Settings |
|---|---|
| Double Tournament | $D$=1.4, *do-fitness-first*=false |
| Proportional Tournament | $P$=0.2 |
| Lexicographic with Direct Bucketing | (not contending) |
| Lexicographic with Ratio Bucketing | $R$=1/2 |
| Plain Lexicographic | (not contending) |
| Linear Parametric | $X$=32 |
| Biased Multi-objective | $F$=0.95 |
| The Waiting Room | (not contending) |
| Death by Size | (not contending) |
| Tarpeian | $W$=0.3 |

Statistical significance was determined by pairwise t-test:

- at 95% confidence when comparing fitness values,
- at 99.995% confidence when comparing tree sizes.

  This setting was chosen in order to make sure that the mean tree size is much smaller to be considered significantly better (while the fitness did not deteriorate).

## Comparison of Methods: Experiment Setup

**The Linear Parametric Parsimony Pressure seems the best.**

- However, this results might be affected by the selection of test problems.

| Method | Settings |
|---|---|
| Double Tournament | $D$=1.4, *do-fitness-first*=false |
| Proportional Tournament | $P$=0.2 |
| Lexicographic with Direct Bucketing | (not contending) |
| Lexicographic with Ratio Bucketing | $R$=1/2 |
| Plain Lexicographic | (not contending) |
| Linear Parametric | $X$=32 |
| Biased Multi-objective | $F$=0.95 |
| The Waiting Room | (not contending) |
| Death by Size | (not contending) |
| Tarpeian | $W$=0.3 |

The best → Linear Parametric

# Operator Equalisation

Should have rather been called "Program Length Equalisation".

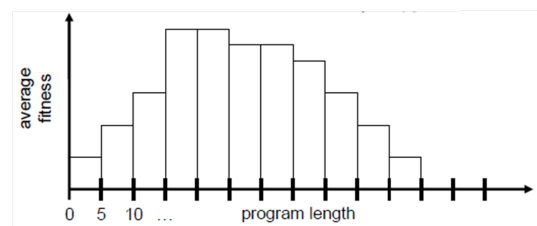
**Controls the distribution of program lengths in the population**, biasing the search towards the desired lengths. Too small and excessively large programs are eliminated from the population.

- 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 – 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.

**Concept of histograms**

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

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




- 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.

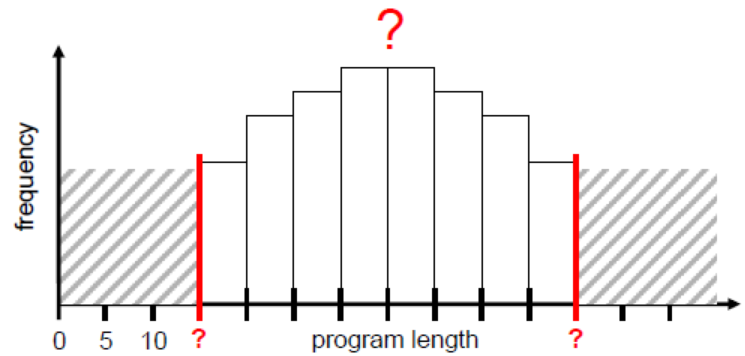
# Operator Equalisation: Two Variants

## 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

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

where

- $\overline{f}_i$ is the average fitness in the bin $i$,
- $\overline{f}_b$ is the average fitness of the individuals in bin $b$, 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

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.

# 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`