Runtime Analysis of Mutation-Based Geometric Semantic Genetic Programming on Boolean Functions

Alberto Moraglio[1], **Andrea Mambrini**[1], Luca Manzoni[2]

University of Birmingham, Birmingham UK
Universitá degli Studi di Milano-Bicocca, Milan, Italy

January 26, 2013

Outline

- Traditional Genetic Programming, why is it hard to analyse?
- Previous work on runtime analysis of GP
- Geometric Semantic Genetic Programming (GSGP)
- Analysis of Geometric Semantic Genetic Programming
- Experiments

Black-box boolean function learning

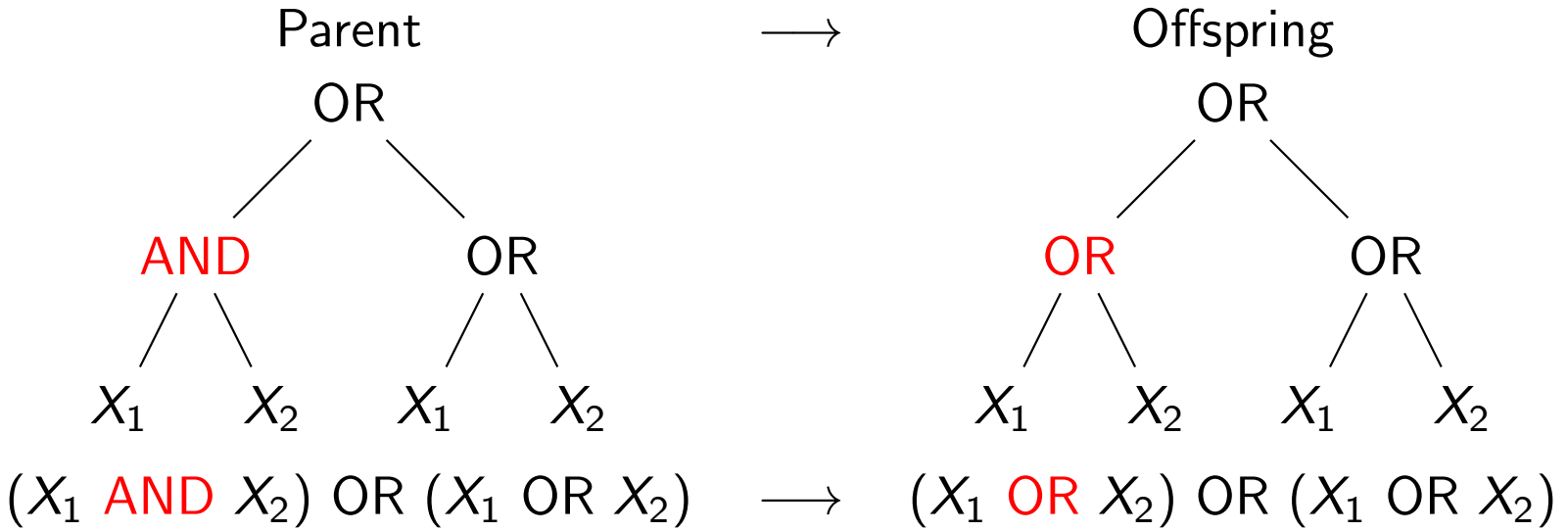
Which boolean expression produces the required output behaviour?

X_1	X_2	Output
1	1	1
1	0	0
0	1	1
0	0	0

Black-box setting:

- We don't have access to the truth table
- We can give our candidate function $f(X_1, X_2)$ to an oracle that will answer us the number of mismatched output rows.

Traditional Genetic Programming

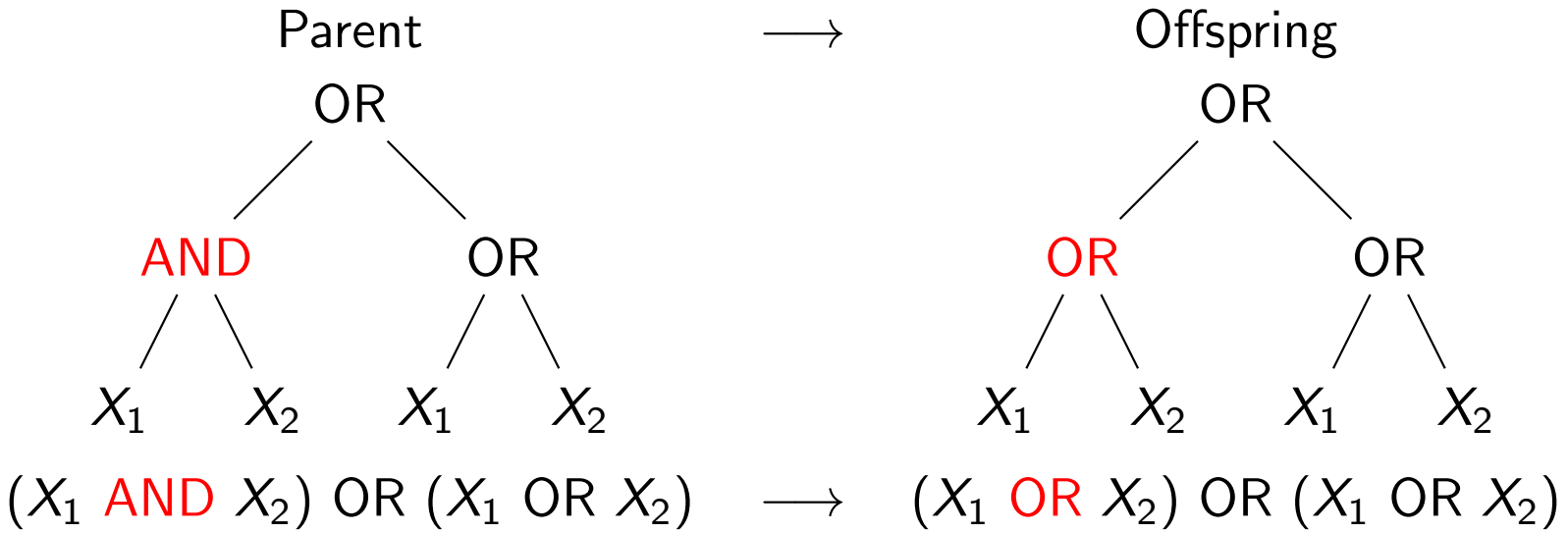


- Expressions are represented by trees.
- Variation operators produce offspring by syntactic manipulation of parent trees

Why runtime analysis of GP is difficult?

- The fitness depends on the behaviour (semantic) of the function to evolve, while variation operators act on the syntax.
- How a syntactic variation operator affects the semantic of the expression?

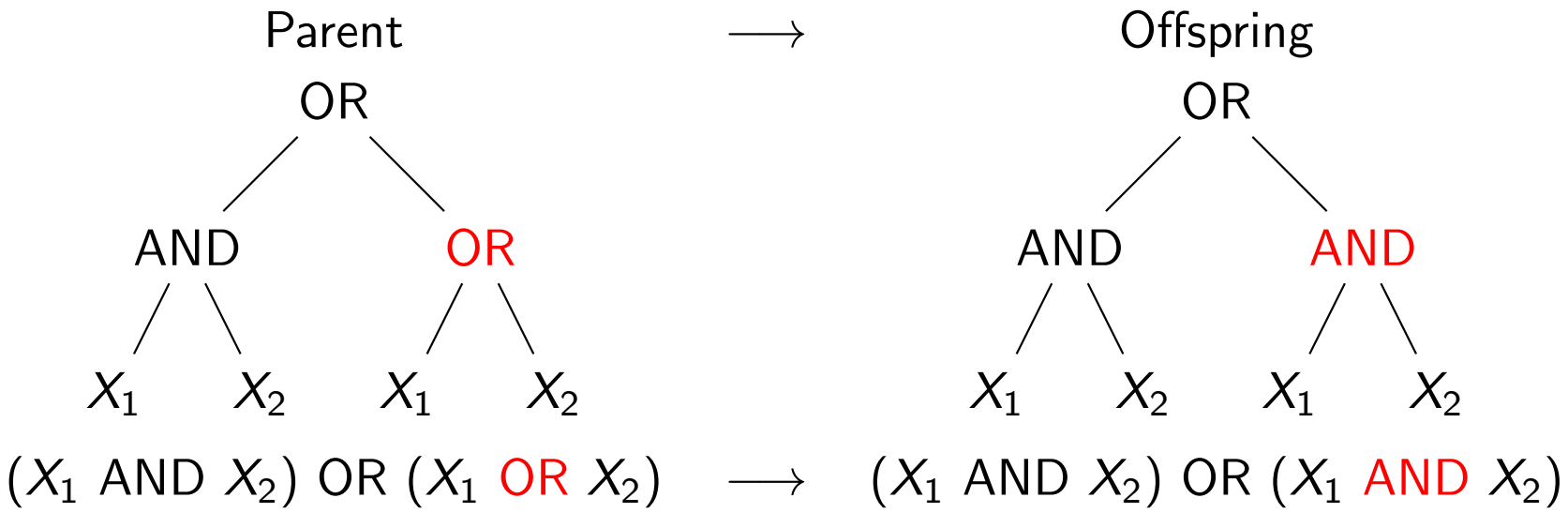
Traditional Genetic Programming



X_1	X_2	Output	Parent	Offspring
1	1	1	1	1
1	0	0	1	1
0	1	1	1	1
0	0	0	0	0

In this case the variation operator **haven't produced any change** in the semantic of the function.

Traditional Genetic Programming



X_1	X_2	Output	Parent	Offspring
1	1	1	1	1
1	0	0	1	0
0	1	1	1	0
0	0	0	0	0

In this case it has produced a **big changes** on the truth table.

Previous work on runtime analysis of GP

- Neumann, OReilly, Wagner – *Computation complexity analysis of Genetic Programming, Initial results and future directions*, 2011
- Kotzing, Sutton, Neumann, OReilly – *The max problem revisited: the importance of mutation in genetic programming*, 2012
- ...

The path followed to analyse GP has been the following:

1. simplifying the GP algorithm
2. find a specific problem that is simple enough to get runtime results (i.e. MAJORITY, ORDER, MAX)
3. get initial runtime results

Can we improve the situation?

Can we redesign GP such that:

- it is easy to deal with it from a runtime point of view
- we can easily get runtime results for classes of problems
- we can use those results to design better operators
- those better operators actually outperform traditional GP on real problems

?

Let's find it out!

Semantic Genetic programming

Semantic variation operator

- acts on the syntax of an expression (given a parent expression produce as offspring another expression)
- **guarantee** some semantic features for the offspring (e.g. a semantic mutation might produce an offspring whose truth table always differs just by one row from the parent)

How to implement a semantic operator?

By trial & error use standard operators and reject offspring that do not conform to the semantic requirement $\rightarrow$ wasteful

By construction : Operate on the syntax of the expression in a way that by construction the semantic criterion is satisfied

A semantic mutation operator

Definition

Bit-flip point mutation: Given a parent function $P : \{0,1\}^n \to \{0,1\}$ the mutation returns the offspring boolean function $P' = \left(P \wedge \overline{M}\right) \vee \left(M \wedge \overline{P}\right)$, where M is a random minterm of all input variables.

Example:

Random minterm: $\mathrm{M} = X_1 \wedge \overline{X_2}$

Parent: $\mathrm{P} = (X_1 \wedge X_2) \vee (X_1 \vee X_2)$

Offspring: $O = (P \wedge \overline{M}) \vee (\overline{P} \wedge M)$

X_1	X_2	Output	M	Parent	Offspring
1	1	1	0	1	1
1	0	0	1	1	0
0	1	1	0	1	1
0	0	0	0	0	0

Semantic Bit-flip point mutation

X_1	X_2	Output	M	Parent	Offspring
1	1	1	0	1	1
1	0	0	1	1	0
0	1	1	0	1	1
0	0	0	0	0	0

Effect on the output vector (semantic):

- Select a row at random (randomly generating M)
- Flip the output of the function on that row (the row where M is true)

Basically at each generation we flip one bit from a bitstring (the output vector) aiming to reach a desired bit configuration...

that reminds RLS on OneMax!

Runtime

Search equivalence

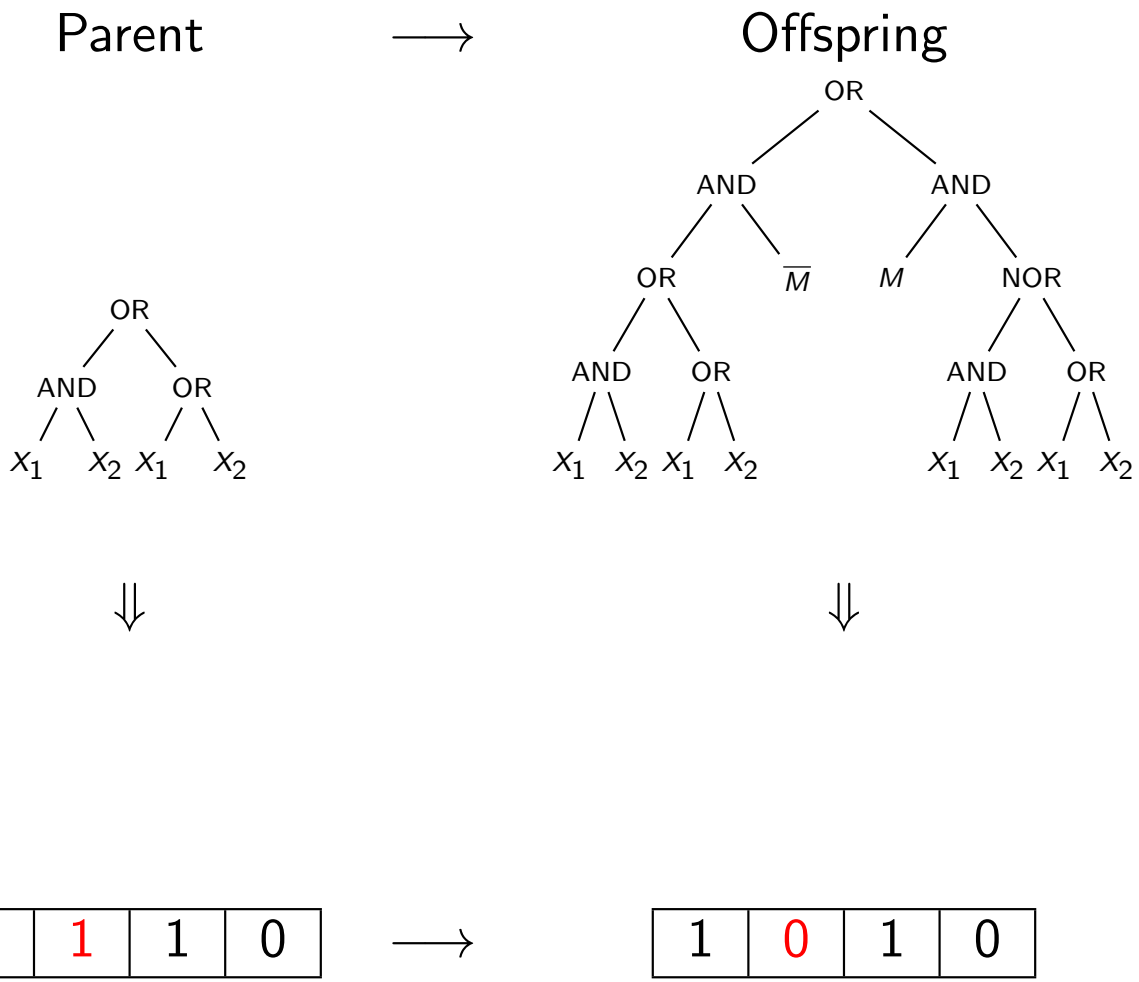
Genetic Programming using semantic operators solving **ANY** black-box boolean function learning problem

$\Updownarrow$ **is equivalent to** $\Updownarrow$

a Genetic Algorithm (evolving bitstrings) solving OneMax

The search outputs a tree (i.e., a boolean expression), but the runtime analysis can be done on the GA!

Equivalence



Runtime

Search equivalence

Genetic Programming using semantic operators solving **ANY** black-box boolean function learning problem

⇳ **is equivalent to** ⇳

a Genetic Algorithm (evolving bitstrings) solving OneMax

The search outputs a tree (i.e., a boolean expression), but the runtime analysis can be done on the GA!

Theorem

Geometric Semantic Genetic Programming solves the black box boolean function learning problem in time $T = O(N \log N)$, where N is the length of the output vector (i.e. the number of rows of the truth table).

Issues

The proposed operator still suffers from the following issues:

1. In the way we designed the semantic mutation operator ($O = (P \wedge \overline{M}) \vee (\overline{P} \wedge M)$), the length of the offspring grows exponentially.
2. A fair choice of the problem size is the number of input variables n. Since $N = 2^n$, the runtime (which is $O(N \log N)$) is actually exponential in the problem size.

A shorter operator

Problem Since $O = (P \wedge \overline{M}) \vee (\overline{P} \wedge M)$, the length of the offspring grows exponentially.
Solution:

Definition

Forcing point mutation: Given a parent function $P : \{0,1\}^n \to \{0,1\}$:

- generate a random minterm M
- return as offspring $P' = P \vee M$ with probability 0.5 and $P' = P \wedge \overline{M}$ with probability 0.5

Effect on the output vector: forcing a random bit (corresponding to the row that makes M true) to a random value (1 with probability 0.5, and 0 with probability 0.5).

Polynomial-sized training set

Problem The runtime $O(N \log N)$ is exponential in the problem size n (since $N = 2^n$).
Actually, in practice, the training set is *small*. Particularly we can assume it to have polynomial size w.r.t. the number of input variables.
$\tau = \text{poly}(n)$.

X_1	X_2	X_3	O
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	0
0	0	1	1
0	0	0	1

This transforms the problem seen by the EA on output vectors into a "sparse" version of OneMax. Still, using the bit-flip operator, the time to reach the optimum is exponential.

What are we looking for?

What we want is a mutation operator that:

- gives rise to a polynomial runtime on the sparse OneMax (thus when the size of the training set τ is polynomial in the number of variables)
- can be implemented efficiently (sampling an offspring takes polynomial time)
- makes the offspring growing polynomially with the number of generations (we are looking for short boolean expressions).

The last operator proposed (Forcing Point Mutation) still fails to satisfy the first condition.

How to achieve that?

Why is it slow?

Most of the times the bit-flip operator flips bits that are not in the training set $\rightarrow$ fitness does not increase $\rightarrow$ slow runtime.

- How can we increase the chances to flip a bit of the training set at each generation? **flipping more bits!**

How many bits?

Sparse OneMax with τ out of N active bits can be solved in $O(\tau \log \tau)$ by a GA flipping at each generation each bit with probability $1/\tau$.

Proof idea: in expectation at each generation one active bit is flipped, thus the runtime is that of OneMax on a bitstring of length τ.

A bitwise mutation operator

How to implement a semantic operator inducing in the output space bitwise mutation with probability $p = 1/\tau$?

Definition

Bitwise mutation: Given a parent function $P : \{0,1\}^n \to \{0,1\}$ the mutation does the following:

- Sample an integer number x from $x \sim \text{Bin}(p, N)$
- Generate x minterms uniformly at random without repetitions $\{M_1, \cdots, M_x\}$
- The offspring is $P' = P \vee M$ with probability 0.5, and $P' = P \wedge \overline{M}$, where $M = M_1 \vee \cdots \vee M_x$.

Problem: Setting $p = 1/\tau$ (in order to solve the problem in $O(\tau \log \tau)$), the offspring takes exponential time to be generated (since M has expected length $pN = \frac{N}{\tau}$ which is exponential in the size of the problem n).

Block mutation

Why is it slow?

Most of the times the bit-flip operator flips bits that are not in the training set $\rightarrow$ fitness does not increase $\rightarrow$ slow runtime.

- How can we increase the chances to flip a bit of the training set at each generation? **flipping more bits!**
- How can we flip more bits keeping the size of the offspring small? **incomplete minterms!**

Incomplete minterms

Incomplete minterms force mutation on blocks of the output vector instead of single bits

X_1	X_2	X_3	$X_2 \wedge X_3$	$\overline{X_2} \wedge X_3$	$X_2 \wedge \overline{X_3}$	$\overline{X_2} \wedge \overline{X_3}$
1	1	1	**1**	0	0	0
1	1	0	0	0	**1**	0
1	0	1	0	**1**	0	0
1	0	0	0	0	0	**1**
0	1	1	**1**	0	0	0
0	1	0	0	0	**1**	0
0	0	1	0	**1**	0	0
0	0	0	0	0	0	**1**

We flip more bits at each generation keeping the length of the expression short.

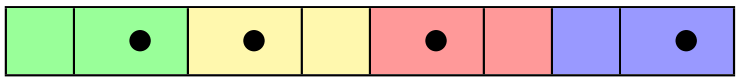
Probability of success

Definition

Fixed Block Mutation (FBM):

- Select randomly $v < n$ variables
- At each generation draw an incomplete minterm M comprising all the fixed v variables
- Use M for the forcing mutation

The output vector is partitioned into $b = 2^v$ blocks. At each generation a whole block is forced to the same value (0 or 1).



N = 8, b = 4

If each training point lays in a different block it is possible to reach the optimum in time $O(b \log b)$.

Polynomial runtime with high probability

We say that an operator solve a problem in polynomial time with high probability if:

- Choosing uniformly at random a **truth table** from all the possible truth tables using n boolean variables
- Choosing uniformly at random τ rows out of $N = 2^n$ to form the **training set**

$$\Downarrow$$

the probability that the problem chosen is one that can be solved in polynomial time using that operator is $\Omega(1 - \frac{1}{n})$.

Runtime of GSGP using FBM

If each training point lays in a different block it is possible to reach the optimum in time $O(b \log b)$.
This happens with probability

$$P(b, \tau) = \prod_{i=1}^{\tau} \frac{b - (i-1)}{b} \approx e^{-\frac{\tau^2}{2b}}$$

Choosing v $[b = 2^v]$

v must be:

- **Big enough** to guarantee that a solvable problem is selected with high probability
- **Small enough** to guarantee, when the problem is solvable, to solve it in polynomial time.

Solution (assuming $\tau = O(n^c)$)

$$v = 2(c+1)\log_2(n) \Rightarrow \text{runtime:} O(n^{2c} \log n), p = O(1 - 2/n)$$

Can we do better?

Definition

Multiple Size Block Mutation (MSBM): At each generation:

- Sample v uniformly at random between 0 and n
- Select randomly v variables (among n)
- Draw an incomplete minterm M comprising all the selected v variables
- Use M for the forcing mutation

On the output vector this is equivalent to flip 1 bit with probability $\frac{1}{n}$, 2 bits with probability $\frac{1}{n}$, 4 bits with probability $\frac{1}{n}$, $\cdots$, and N bits with probability $\frac{1}{n}$.

- Probability of success: **1**
- Time complexity: still **polynomial** on all the instances that can be solved by FBM but exponential in the worst case.

Experiments

- Generate a random truth table with $n = 8, 16, 24, 32, 40, 48$ variables
- Select $\tau = n$ training points
- Check if it is possible to reach the optimum (**success rate**) and the **average runtime**.

	Problem Size					
	8	16	24	32	40	48
FBM	0.95	0.76	0.93	0.74	0.87	0.88
VBM	0.95	0.99	1.00	1.00	1.00	1.00
FABM	1.00	1.00	1.00	1.00	1.00	1.00
MSBM	1.00	1.00	1.00	1.00	1.00	1.00

Table: The success rate registered for the different problem sizes and mutations tested.

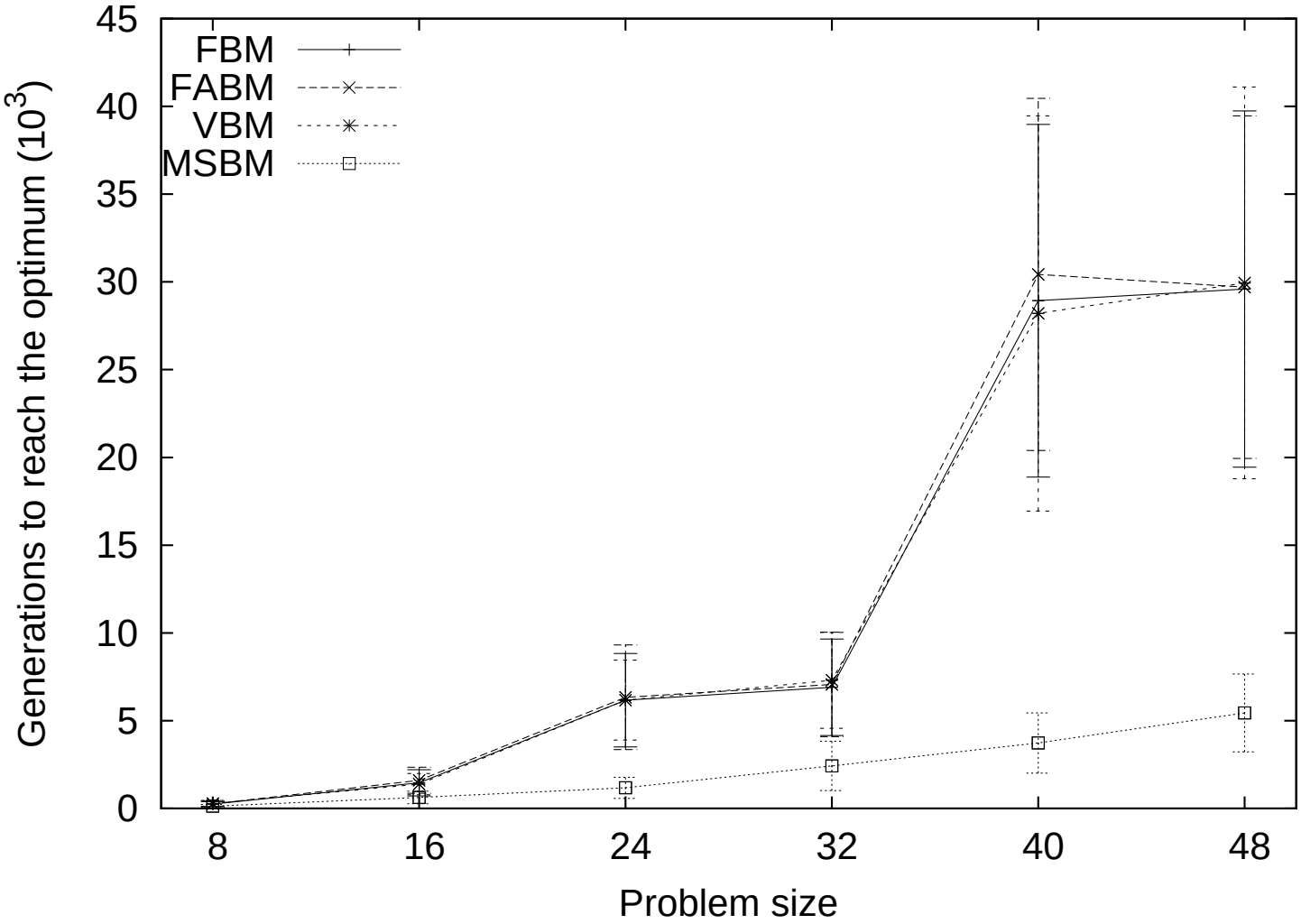


Figure: Average number of generations to obtain an optimal solution. The error bars represent one standard deviation above and below the average.

Summary

- Unlike traditional GP, GSGP searches directly the semantic space of functions.
- GP with semantic operators on boolean expressions is equivalent to a GA with traditional operators on output vectors (bitstrings).
- For any boolean function to learn, the landscape seen by GSGP is OneMax. This allows us to analyse the runtime of GSGP and set some parameters using results from theory.
- We designed a block mutation (FBM) which reach the optimum in polynomial time with high probability on all black box boolean function learning problems.
- Experiments on the average-case complexity showed that MSBM could be much faster than the others even if it has an exponential worst case complexity.

Conclusion

Take home messages:

- Geometric Semantic Genetic Programming makes the search performed by GP on the space of functions (trees) equivalent to that performed by a GA on bitstrings.
- It is thus possible to easily get runtime results for GP reusing runtime results for GA
- We can use those theoretical results to design good semantic GP operators

Future work:

- Test GSGP on common GP benchmarks
- Study the generalization ability of GSGP as a machine learning tool
- Design semantic operators to evolve continuous functions.

Thank you!

More operators

We propose two more operators to try to improve the results of FBM:

Variable Block Mutation (VBM) We choose the set of variables to build the incomplete minterms at each generation (instead of fixing it at the begining) $\rightarrow$ better success probability of FBM but exponential runtime on some instances solvable by FBM in polynomial time.

Fixed alternative Block Mutation (FABM) We choose at each generation the set of variables from a fixed subsets of all the possible sets of variables $\rightarrow$ success probability higher than FBM but lower than VBM but polynomial runtime in the worst case.