# Semantycznie gładkie programowanie genetyczne

Krzysztof Krawiec and Tomasz Pawlak

Institute of Computing Science, Poznan University of Technology, Poznań, Poland

Oct 25, 2011

# Outline

# Sliding puzzle world

Consider $n \times n$ sliding puzzle, e.g., for $n = 3$:

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

Manipulating such a puzzle can be considered as a simple programming task.

- A *program* is any sequence composed of four instructions {L,R,U,D}, which shift the empty space of the puzzle
  - Note that some instructions can be ineffective.
- The state of the puzzle is the state of memory of the virtual machine that executes the program,
  - 9!=362880 possible memory states.

# Sliding puzzle task

Find the program that transforms the given starting configuration into the target configuration:

| | | |
|---|---|---|
| 5 | | 4 |
| 1 | 3 | 6 |
| 8 | 2 | 7 |

→ DLU...RR →

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

Evolutionary approach:

- Encode programs as individual's genotypes (vectors),
- Run evolution using (minimized) fitness function based on city-block distance, e.g.:

$$f\left(\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 6 & \\ \hline 7 & 8 & 5 \\ \hline\end{array}\right) = 4, \text{ because } \left\|\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 6 & \\ \hline 7 & 8 & 5 \\ \hline\end{array} - \begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & \\ \hline\end{array}\right\| = 4$$

# Sliding puzzle: Features

The domain of sliding puzzle is simple, but captures main important features of programming:

- *compositionality*: new programs can be created by composing (concatenating) other programs,
- *contextuality*: an effect of a program fragment depends on the input memory state:
  - some instructions (and instruction sequences) can be ineffective,

... and genetic programming:

- the programs are evaluated by *running* them on input data,
- the performance of the program is a function of a *distance* between its output and desired output,

Two-point homologous crossover:

| | |
|---|---|
| Parent A: | UDLLRURRLD |
| Parent B: | LLRUDDDLRR |
| Offspring 1: | UDRUDURRLD |
| Offspring 2: | LLLLRDDLRR |

Questions:

- How does this crossover impact the *behavior* of the program?
- Can we design better crossovers?

What is the *behavior* (*semantics, meaning, effect*) of a program *p*?

- Essentially a philosophical questions.
- Here: the set (vector) of results that *p* produces for all possible input configurations.
- However, it is sufficient to consider only 9 states that are unique w.r.t. location of space.

Example: consider program $p = \{\text{left,up}\}$:

9 input states:

[

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | | 8 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| | 7 | 8 |

...

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

]

Semantics of *p*:

[

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | | 6 |
| 7 | 5 | 8 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 5 | 6 |
| 4 | 7 | 8 |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 5 | 6 |
| 4 | 7 | 8 |

...

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

]

The semantic distance between two [sub]programs (sequences of instructions) $p_1$ and $p_2$ is the total distance $d_m$ between the final (resulting) memory states obtained by applying $p_1$ and $p_2$ to all possible [input/starting] memory states.

Example:
Semantics of $p_1$:

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | | 1 | 2 | 3 | | 1 | 2 | 3 | | | 1 | 2 | |
| [ | 4 | | 6 | | | 5 | 6 | | | 5 | 6 | ... | 3 | 4 | 5 | ] |
| | 7 | 5 | 8 | | 4 | 7 | 8 | | 4 | 7 | 8 | | 6 | 7 | 8 | |

Semantics of $p_2$:

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | | 1 | | 3 | | 1 | 2 | 3 | | 3 | 1 | 2 | |
| [ | 4 | 6 | | | 2 | 5 | 6 | | | 5 | 6 | ... | 6 | 4 | 5 | ] |
| | 7 | 5 | 8 | | 4 | 7 | 8 | | 4 | 7 | 8 | | | 7 | 8 | |

Distance: $||s(p_1) - s(p_2)|| = 10$ :

$$(1+1) \quad + \quad (2+2) \quad + \quad 0 \quad + \quad (2+1+1)$$

# Semantic-based crossover operators

The general idea: use *some* semantic information derived from the parents to produce the offspring. For instance, *geometricity* (*non-colinearity*):

$$||s(p) - s(p_1)|| + ||s(p) - s(p_2)|| \tag{1}$$

Example 1: Semantically geometric crossover:

$$offspring(p_1, p_2) = \arg\min_p ||s(p) - s(p_1)|| + ||s(p) - s(p_2)||$$

Example 2: Stochastic semantically geometric crossover:

$$offspring(p_1, p_2) = o[\lfloor \lambda \exp(-\lambda x) \rfloor]$$

where $o$ is a table containing all programs $p \neq p_1, p_2$ sorted ascending with $||s(p) - s(p_1)|| + ||s(p) - s(p_2)||$

Pros:

- The fitness landscape that spans the semantic space is a cone $\Longrightarrow$ unimodal. With enough variation, the above operators *guarantee* progress and convergence to the global optimum.

Cons:

- Computationally infeasible. Require running *all* programs in advance.

# Semantic-based crossover operators

Idea: Use semantic-based crossover operators that work on program *fragments* (*subprograms*)

Illustration:

| | |
|---|---|
| Parent A: | `UDLLRURRLD` |
| Parent B: | `LLRUDDDLRR` |
| Offspring 1: | `UDUULURRLD` |
| Offspring 2: | `LLLRUDDLRR` |

The subprograms `UUL` and `LRU` have been pasted into Offspring 1 and Offspring 2 based on their semantic 'geometricity' w.r.t. to `LLR` and `RUD`.

For instance: stochastic semantically geometric crossover *applied to subprograms*:

1. Assume certain subprogram length $l$
2. Prior to run, for each pair of subprograms of length $l$, generate all the possible resulting subprograms, store them in a table, and sort them according to (1).
3. During crossover, draw the program fragments from the table.

# Experiment: Comparing crossovers

Assumption: all considered crossovers are homologous and affect a randomly selected continuous genome fragment (subprogram) of length $l$

- SX: Stochastic semantically geometric crossover *applied to subprograms*
- Control methods:
  - SXU: SX with uniform distribution
  - TWO: two-point crossover
  - MM: Macromutation: randomize the selected fragment

# Settings

- Generational EA
  - Fitness function: minimal city-block distance from the target calculated over program trace (i.e., all intermediate memory states)
  - Tournament selection (size: 7)
  - Population size: 100
  - Mutation probability (point mutation): 0.03
  - Max number of generations: 1000
- Genome length: 20, 40, 60, 80 instructions
- Subprogram length: 3
- $\lambda = 10$
- 60 runs per setting
  - Each run uses different starting puzzle state
  - Run outcome: success/failure

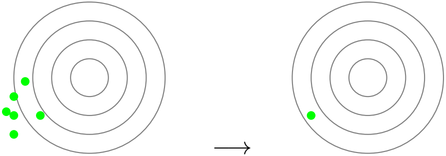
Success rate (the percentage of runs ended with success):

| Program length | $l$ | MM | TWO | SXU | SX |
|---|---|---|---|---|---|
| 20 | 3 | **0.22** | 0.12 | 0.17 | 0.15 |
| 20 | 4 | 0.20 | 0.10 | 0.20 | 0.15 |
| 20 | 5 | **0.23** | 0.10 | 0.20 | 0.15 |
| 40 | 3 | **0.53** | 0.28 | 0.37 | 0.25 |
| 40 | 4 | **0.47** | 0.27 | 0.47 | 0.25 |
| 40 | 5 | **0.50** | 0.23 | 0.47 | 0.37 |
| 60 | 3 | 0.57 | 0.38 | **0.60** | 0.40 |
| 60 | 4 | **0.65** | 0.35 | **0.65** | 0.47 |
| 60 | 5 | 0.63 | 0.32 | **0.78** | 0.45 |
| 80 | 3 | **0.83** | 0.53 | 0.80 | 0.52 |
| 80 | 4 | 0.72 | 0.62 | **0.82** | 0.62 |
| 80 | 5 | 0.82 | 0.58 | **0.83** | 0.70 |

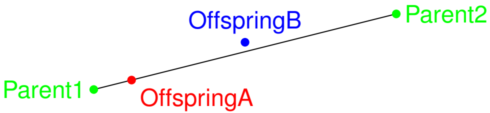
# What went wrong?

The causes of the problem:

1. SX makes premature convergence more likely.



2. SX promotes perfectly geometric offspring, even if that means little progress.



# Countermeasures

- Ad 1: Combine SX with MM (SX+MM).
  - If Parent1 and Parent2 are semantically distinct, do SX
  - Otherwise, do MM
- Ad 2: Use other measures, e.g., such that promote the offspring that are equidistant from parents.

Success rate (the percentage of runs ended with success):

| Program length | $l$ | MM | TWO | SXU | SX | SX+MM | TWO+MM |
|---|---|---|---|---|---|---|---|
| 20 | 3 | **0.22** | 0.12 | 0.17 | 0.15 | 0.18 | 0.15 |
| 20 | 4 | 0.20 | 0.10 | 0.20 | 0.15 | **0.27** | 0.15 |
| 20 | 5 | 0.23 | 0.10 | 0.20 | 0.15 | **0.27** | 0.12 |
| 40 | 3 | **0.53** | 0.28 | 0.37 | 0.25 | 0.47 | 0.27 |
| 40 | 4 | 0.47 | 0.27 | 0.47 | 0.25 | **0.57** | 0.28 |
| 40 | 5 | 0.50 | 0.23 | 0.47 | 0.37 | **0.53** | 0.28 |
| 60 | 3 | 0.57 | 0.38 | 0.60 | 0.40 | **0.65** | 0.43 |
| 60 | 4 | 0.65 | 0.35 | 0.65 | 0.47 | **0.77** | 0.40 |
| 60 | 5 | 0.63 | 0.32 | **0.78** | 0.45 | 0.75 | 0.43 |
| 80 | 3 | **0.83** | 0.53 | 0.80 | 0.52 | 0.70 | 0.57 |
| 80 | 4 | 0.72 | 0.62 | 0.82 | 0.62 | **0.95** | 0.60 |
| 80 | 5 | 0.82 | 0.58 | 0.83 | 0.70 | **0.93** | 0.73 |

(TWO+MM – yet another control experiment)

# Discussion

- Challenges:
  - Granularity of semantic distance.
  - Testing subprograms on all possible inputs often unfeasible.
- Does it work for more sophisticated programming languages?
- What is the true structure of semantic space?
- Further possibilities:
  - Make it more effective
  - Encapsulate the subprograms

# Discussion (2)

This is not guaranteed to work:

- A program that, *at some execution point*, produces a result that is semantically 'intermediate' w.r.t. the results produced by parent programs at the corresponding execution points, is not necessarily semantically intermediate w.r.t. the parent programs.
- Formally: Let $p^k$ denote program $p$ trimmed to its first $k$ instructions. Then:

$$\begin{aligned} ||s(p^k) - s(p_1^k)|| + ||s(p^k) - s(p_2^k)|| &= ||s(p_1^k) - s(p_2^k)|| \\ \not\Longrightarrow \quad ||s(p) - s(p_1)|| + ||s(p) - s(p_2)|| &= ||s(p_1) - s(p_2)|| \end{aligned}$$

[If the above held, every act of crossover applied to semantically different parents, producing an offspring that is semantically different from them, would improve the result]

## The problem

Definition of SX:

$$offspring(p_1, p_2) = \arg\min_{p} ||s(p) - s(p_1)|| + ||s(p) - s(p_2)||$$

requires us to check all possible children programs $\rightarrow O(|P|)$.

## The solution

Build the search space, such that it is semantically geometric, so the crossover can be done in $O(1)$ time.

# Search space = set + structure

**Structure**

The search space has a *structure*, which is critical for the performance of a search algorithm that uses some *operators* to traverse it.

- We impose a structure on the program space $P$ by defining a *neighbourhood* $N$
- $N(p)$ = all programs that can be built form $p \in P$ by introducing small changes in it (e.g., substituting a single instruction).
- $N(p)$ = set of all mutants of $p$.

# Example

Univariate symbolic regression:

- Program space of full trees of depth 3: 3 instructions $\{+, \times\}$ and 4 terminals $x$.
- Total number of programs: $2^3 = 8$,
  - but only 6 semantically unique, due to symmetry of operators.

A hand-designed structure for this space that minimizes the total Hamming distance between the *syntax* of the neighboring programs:

| | | | | |
|---|---|---|---|---|
| $(x+x)+(x+x)$ | $\overset{1}{\leftrightarrow}$ | $(x+x)+(x\times x)$ | $\overset{1}{\leftrightarrow}$ | $(x\times x)+(x\times x)$ |
| $\updownarrow 1$ | | $\updownarrow 1$ | | $\updownarrow 1$ |
| $(x+x)\times(x+x)$ | $\overset{1}{\leftrightarrow}$ | $(x+x)\times(x\times x)$ | $\overset{1}{\leftrightarrow}$ | $(x\times x)\times(x\times x)$ |

Does syntactic structure corresponds with the behavior of program?

Univariate symbolic regression:

- Program space of full trees of depth 3: 3 instructions $\{+, \times\}$ and 4 terminals $x$.
- Total number of programs: $2^3 = 8$,
  - but only 6 semantically unique, due to symmetry of operators.

A hand-designed structure for this space that minimizes the total Hamming distance between the *syntax* of the neighboring programs:

$$
\begin{array}{ccccc}
\boxed{(x+x)+(x+x)} & \overset{1}{\leftrightarrow} & \boxed{(x+x)+(x\times x)} & \overset{1}{\leftrightarrow} & \boxed{(x\times x)+(x\times x)} \\
\updownarrow 1 & & \updownarrow 1 & & \updownarrow 1 \\
\boxed{(x+x)\times(x+x)} & \overset{1}{\leftrightarrow} & \boxed{(x+x)\times(x\times x)} & \overset{1}{\leftrightarrow} & \boxed{(x\times x)\times(x\times x)}
\end{array}
$$

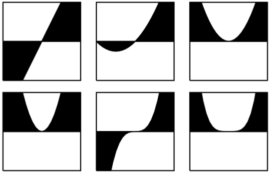
Does syntactic structure corresponds with the behavior of program?
**No!**

# Example: Structure 1

Program space $P$:

| | | |
|---|---|---|
| $4x$ | $x^2 + 2x$ | $2x^2$ |
| $4x^2$ | $2x^3$ | $x^4$ |

The corresponding program semantics:

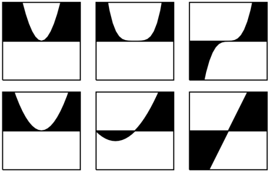


# Example: Structure 2

A different arrangement of programs in $P$:

| | | |
|---|---|---|
| $4x^2$ | $x^4$ | $2x^3$ |
| $2x^2$ | $x^2 + 2x$ | $4x$ |

The corresponding program semantics:



# Embedding

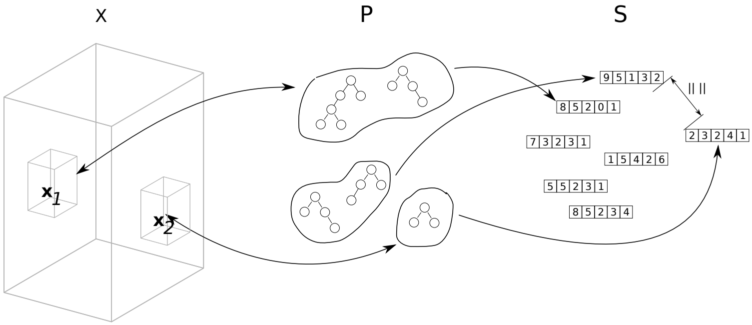
**The idea**

To explicitly rearrange the programs in the program space so that semantically similar programs occupy neighboring locations.

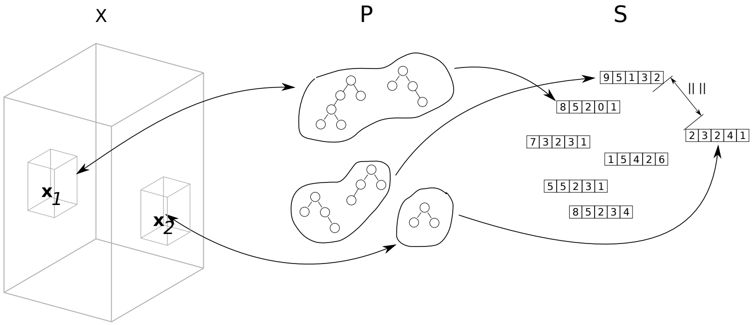
Formally:

- We *embed* the program space in a *prespace* $X$ of convenient topology.
- We use a search algorithm to assign the programs from $P$ to particular locations in $X$, so that the corresponding semantic space is more *smooth*.

The assumed topology: toroidal hypercube.



The assumed topology: toroidal hypercube.




- Coordinates correspond to semantic equivalence classes of programs.
- *P* becomes transparent.

Measures the amount of space deformation when program $p$ and its neighbors get mapped into $S$:

$$l(p, s) = \frac{1}{|N(p)|} \sum_{p' \in N(p)} \frac{1}{1 + \|s(p') - s(p)\|} \tag{2}$$

where:

- $s(p)$ is the semantics of $p$,
- $\| \|$ is a metric in the semantic space $S$.

Properties:

- $l = 1 \implies$ all neighbors of $p$ have the same semantics as $p$
- $l \approx 0 \implies$ all neighbors have very different semantics from $p$

Total locality:

$$L(s) = \frac{1}{|P|} \sum_{p \in P} l(p, s) \tag{3}$$

Notice: $L$ is independent on program instance. It depends only on $P$ and $N$.

# The embedding optimization algorithm

The problem is NP-hard.

Greedy local search heuristic:

- Start with the hyper-rectangle $X$ filled up with randomly ordered programs,
- Repeat until locality improvement drops below a given threshold $t$:
  - For each location $\mathbf{x} \in X$ (column by column, row by row):
    - Consider every $\mathbf{x}' \in X \setminus \{x\}$,
    - Temporarily swap the programs located in $\mathbf{x}$ and $\mathbf{x}'$,
    - If improvement of locality is greater than $t$, then start loop from the beginning,
    - Otherwise retract the move.

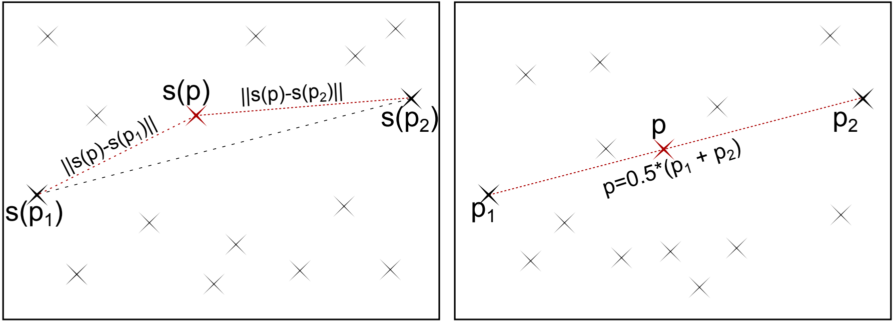
Typically, tens thousand iterations to convergence.

$$offspring(p_1, p_2) = \arg\min_p ||s(p) - s(p_1)|| + ||s(p) - s(p_2)||$$

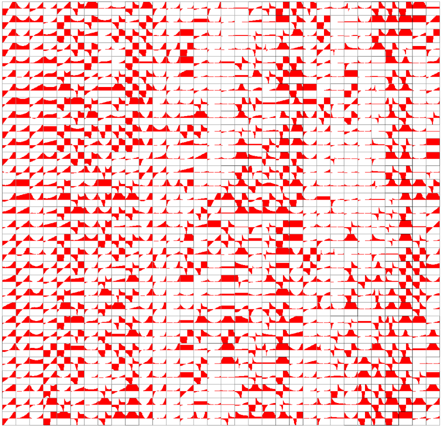
$$\Downarrow$$

$$offspring(p_1, p_2) = \frac{p_1 + p_2}{2}$$

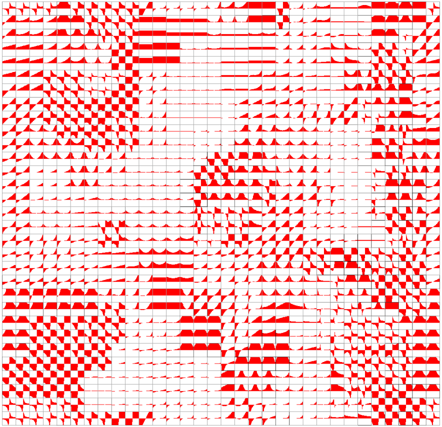
Semantics space | Optimized program space



- Symbolic regression:
  - Instructions: $\{+, -, \times, /, x\}$
  - Trees of depth at most 4
  - Semantics in interval $-1..1$ with step 0.1
  - Total number of programs: 27284, discarding symmetric: 21385, semantically unique: 962.

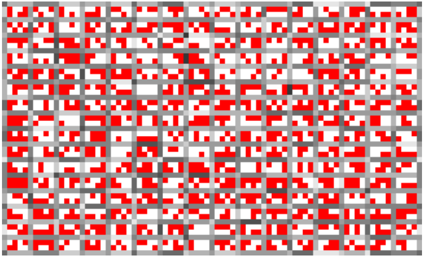


Random space

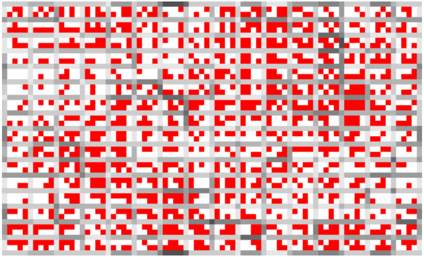


Optimized space

- Synthesis of logic functions:
  - Instructions: Ordered Binary Decision Diagrams (OBDD)
  - Diagrams of depth equal to 3
  - Total number of programs: 256 – each semantically unique!

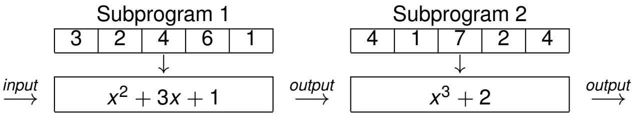


Random space



Optimized space

# Experiment: Embedding of subprograms (1)

- Consider all binary trees of depth 4 composed of $\{+, -, \times, /, 1, x\}$, 20 fitness cases $[-10, 10]$
  - $4,194,304$ programs, $14,673$ semantically unique
- Phase 1: Optimization of embedding
- Phase 2: Program evolution in the optimized space
  - We evolve compound programs by concatenating simple (sub)programs.
  - Individual's genome is a vector of $2d$ numbers.
  - The output produced by the first subprogram becomes the value of the independent variable for the second program.
  - 

  - Program space size: $|P| = 4,194,304^2$

# The results (1)

Success rate of EA evolving compound programs on a sample of $3,000$ problem instances.

Success rate [%]

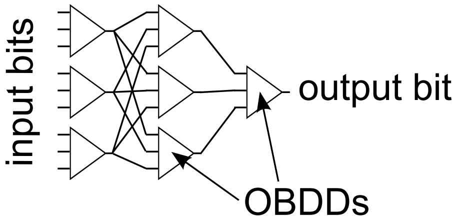
| | $\mathrm{Pr}_m$ | | | | | |
|---|---|---|---|---|---|---|
| $d$ | 0.0 | 0.2 | 0.4 | 0.6 | 0.8 | 1.0 |
| 2 | 0.20 | 0.90 | 1.50 | 0.80 | 1.07 | 0.73 |
| 3 | 0.23 | 0.63 | 0.60 | 0.87 | 1.20 | 0.90 |
| 4 | 0.67 | 0.73 | 0.60 | 1.00 | 1.50 | 2.60 |
| 5 | 0.53 | 0.67 | 0.77 | 0.77 | 1.70 | 3.50 |
| 6 | 0.30 | 0.50 | 1.37 | 1.13 | 1.77 | 5.43 |
| 7 | 0.40 | 0.50 | 1.17 | 1.07 | 1.93 | 3.83 |
| 9 | 0.63 | 0.80 | 1.27 | 1.67 | 3.10 | 5.80 |

Relative increase of success rate compared to random embedding

| $\mathrm{Pr}_m$ | | | | | |
|---|---|---|---|---|---|
| 0.0 | 0.2 | 0.4 | 0.6 | 0.8 | 1.0 |
| 0.47 | 1.70 | 2.24 | 1.04 | 0.97 | 0.84 |
| 0.58 | 1.58 | 1.40 | 1.05 | 1.56 | 1.00 |
| 2.03 | 0.95 | 1.40 | 1.49 | 1.72 | 3.13 |
| 1.33 | 1.26 | 0.93 | 1.00 | 1.59 | 2.63 |
| 0.91 | 0.94 | 2.58 | 1.26 | 1.72 | 4.94 |
| 1.74 | 0.94 | 1.75 | 1.39 | 1.61 | 3.27 |
| 1.58 | 1.33 | 2.02 | 1.92 | 3.33 | 7.25 |

$Pr_c = 1 - Pr_m$

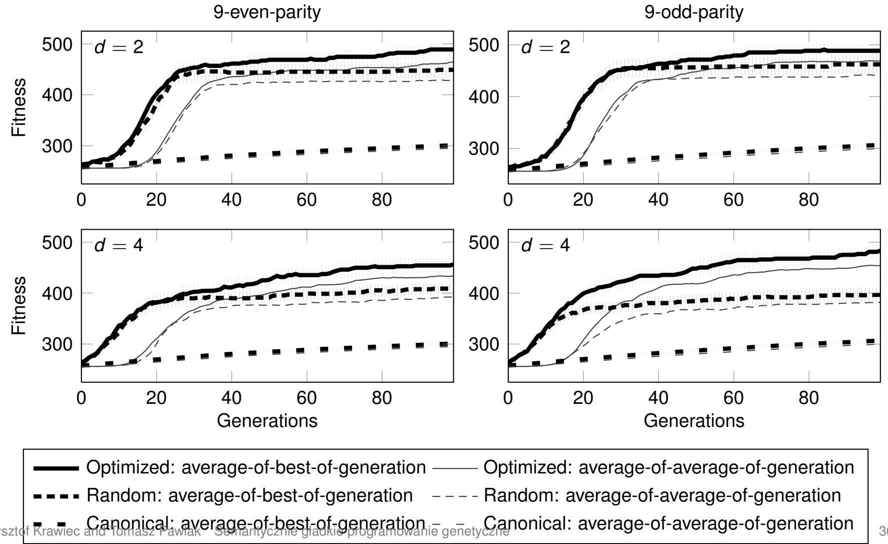
- Consider all OBDDs of depth 3
- 256 programs, 256 semantically unique
- Phase 1: Optimization of embedding
- Phase 2: Program evolution in the optimized space
  - We evolve compound programs by concatenating simple (sub)programs.
  - Individual's genome is a binary neural network of topology $3 \times 3 \times 1$.
  - The output produced by subprograms in the first layer becomes the input for subprograms in the second layer and so on.




- Program space size: $|P| = 256^7 = 2^{56}$

# The results (2): Synthesis of logic function

Search performance of GP working on optimized and random *d*-dimensional program space and canonical problem implementation. Charts show fitness, averaged over 30 runs, of both average-of-best-individual and average-of-average-individual in each generation. The vertical lines represent 0.05 confidence intervals.



# Does it pay off?

# Does it pay off?

No.

The overall computational cost is the sum of:
Phase 1: The cost of redesigning the program space, which requires:

- generating *all* programs (to provide completeness),
- calculating semantics of every program,
- running the optimization process.

Phase 2: The cost of running the search algorithm in the redesigned space.

# Does it pay off?

No.

The overall computational cost is the sum of:
Phase 1: The cost of redesigning the program space, which requires:

- generating *all* programs (to provide completeness),
- calculating semantics of every program,
- running the optimization process.

Phase 2: The cost of running the search algorithm in the redesigned space.

But:

- Phase 1 has to be run only once.
- This idea can be exploited in a compositional manner.
  Embeddings optimized for code *fragments* (subprograms) can be used for building larger programs.

# Conclusions

- Semantic embedding can make the search process more effective.
- An embedding works for an entire *class* of problems (an instruction set and program length limit).
- The optimized prespace can be re-used multiple times for different problem instances.
- Embedding of short programs can be used to speed up the search in the space of compound programs.
- Different view: an embedding defines a set of parameterized meta-instructions.

# What's next?

- The space optimized w.r.t. locality works great with certain problems only. We found multiple problem/instances, where this approach performs worse than canonical method.
- Current experiments:
  - Replacement of locality function with "geometricity" function:

$$G(X) = \frac{2\sum_{p_1,p_2}(\|s(p_1),s(p)\| + \|s(p_2),s(p)\|)}{|X| \times (|X|-1) \times \max\limits_{p',p''}\{\|s(p'),s(p'')\|\}}$$

where:

  - $p_1, p_2, p$ are coordinate vectors in the $X$ space,
  - $p = \frac{p_1+p_2}{2}$

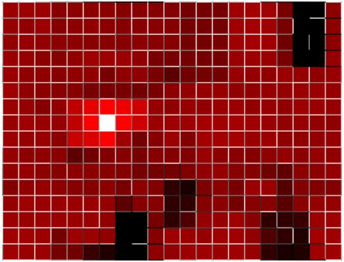
# What's next?

- The space optimized w.r.t. locality works great with certain problems only. We found multiple problem/instances, where this approach performs worse than canonical method.
- Current experiments:
  - Replacement of locality function with "geometricity" function:

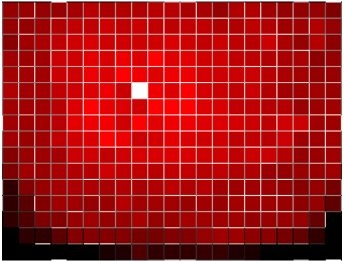
$$G(X) = \frac{2\sum_{p_1,p_2}\left(\|s(p_1), s(p)\| + \|s(p_2), s(p)\|\right)}{|X| \times (|X| - 1) \times \max_{p',p''}\{\|s(p'), s(p'')\|\}}$$

where:
  - $p_1, p_2, p$ are coordinate vectors in the $X$ space,
  - $p = \frac{p_1+p_2}{2}$

Why? Because use of locality unlikely emphasizes global convexity.

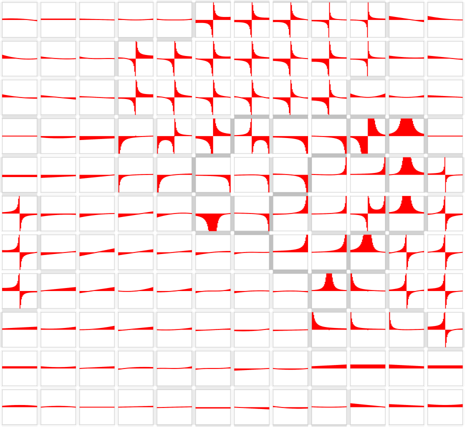


Space optimized w.r.t. locality



Space optimized w.r.t. geometricity

Thank you.



Optimized embedding of 132 semantics obtained from 1024 programs.