CONSTRAINED GENETIC PROGRAMMING

with

CGP lil-gp 2.1;1.02

Cezary Z. Janikow
University of Missouri
janikow@umsl.edu
http://www.cs.usml.edu/~janikow
1. Evolutionary Computation

2. Genetic Programming

3. Constrained Genetic Programming
   - motivation and genesis
   - application environment and implementation
   - formalization and analysis
   - newer features
     - CGP2.1

4. Comparison with STGP

5. Example Applications

6. Summary
1 Evolutionary Computation

- Principles
- Visualization
- Models and Instances
- Search space
1.1 Principles

- Population-based collective learning
  - individuals (chromosomes) represent points in the representation space (genotype) with mapped corresponding points in the solution space (phenotype)
  - chromosomes may also include domain-independent information
- population faces Darwinian stochastic fitness-based selective pressure \((selection, competition)\)
  - chromosomes evaluated as solutions

- chromosomes generate offspring
  - \textit{mutation}
  - \textit{reproduction} \((crossover, co-operation)\)
1.2 Visualization

- Generational Model

Generate initial population \( P(0) \)
Evaluate \( P(0) \)
While not (resources exhausted and done)
  Select \( P(t=t+1) \)
  Reproduce \( P(t) \)
  Mutate \( P(t) \)
  Evaluate \( P(t) \)
  \( t = t+1 \)
1.3 Models and Instances

1.3.1 Models

- Generational
- Steady-state
- Tournament
- Others
1.3.2 Instances

• GA Genetic algorithm

• GP Genetic programming

• ES Evolution strategies

• EP Evolutionary Programming

• Hybrids and derivatives
1.4 Search Space

- State-space search
  - database of current solutions
    - population in EC
  - transition operators
    - selection, mutation, reproduction
  - solving a problem
    - search the representation space (genotype)
    - mapping to solution space (phenotype)
• Mapping (representation vs. solution space)

- *one-to-one*

- how to deal with “extraneous” representation space?
  
  - redundant (*many-to-one*)

  - invalid (how to avoid?)
2 Genetic Programming

• Representation

• Search space

• Assumptions
2.1 GP Representation

- Individuals are trees
  - flexibility
  - suitable for computer programs
    - nodes are labeled with functions
    - terminals are labeled with variables/constants
    - subtrees are arguments

\[
\begin{align*}
f1(\text{arg1}, \text{arg2}); \\
f2(\text{arg1}, \text{arg2}, \text{arg3}); \\
\text{program()} \\
\quad \text{var } x, y, z; \\
\quad \text{const } c1, c2; \\
\quad \text{read}(z); \\
\quad x = f2(c1, c2, z); \\
\quad y = 7 \times z; \\
\quad x = f1(x, y); \\
\quad \text{return}(x); \\
\end{align*}
\]
• Types of functions/terminals
  ■ Type I - internal functions
    ◦ with arguments
  ■ Type II - terminal functions
    ◦ no arguments
    ◦ variables
  ■ Type III - ephemeral constants
    ◦ for terminals
    ◦ instantiated
2.2 Search Space

- Tree structures
  - constrained by size limits and function arity
- Tree instances of specific structures
  - constrained by domain sizes
2.3 Assumptions

• **Sufficiency**
  
  - all necessary functions/terminals are given
  
  - increases representation and solution spaces
• Closure

- avoids invalid mappings
  - redundant
  - arbitrary
  - increases representation space
- can slow down “evolution”
- remedies
  - structure-preserving crossover and various typing methods
  - domain-specific crossovers
  - CGP
  - STGP
3 Constrained Genetic Programming (CGP)

• Motivation and genesis

• Early ideas

• Application environment and implementation

• Formalization and analysis

• Recent extensions

• CGP2.1 features and constraints examples
3.1 Motivation and Genesis

- NASA’s robotic projects

- Need to control evolvable structures
  - general
  - generic
  - powerful
  - efficient

- Modify lil-gp
  - utilize built-in features
3.2 Early Ideas

• Domain sets

• Domain compatibility notion
  - $D_1$ is compatible with $D_2$ iff $D_1 \subseteq D_2$
  - a node can be labeled with function/terminal having domain compatible with its parent’s expectations

```
  fm
 /   \
 v    v
 Dm expected on the second argument

f?
```
problems?

• lil-gp does not use explicit domain specifications
• difficulties with continuous domains

approach?

• make the user responsible for analyzing domains and providing function compatibilities instead (CGP1.1)
• use redundant
  • syntactic constraints ($T_{specs}$)
  • semantic constraints ($F_{specs}$)
3.3 Application Environment and Implementation

- Constrained Genetic Programming with \texttt{CGP}

- Implemented within \texttt{lilgp 1.02}
lil-gp’s architecture

- Read functions & terminals creating \( fset \)
- Generate population of programs \( P \)
- Generate \( P' \) by selection, mutation, crossover
- Evaluate \( P' \)
- \( P = P' \)
- Termination?

\( P, P', fset \)
Constrained GP: Constrained Genetic Programming (CGP) ©Cezary Z. Janikow, presented as GP98 Tutorial

- **CGP 1.1’s architecture**

Legend:
- modified module

---

**CGP 1.1’s architecture**

1. **Read functions & terminals**
2. **Generate popul. of programs P**
3. **Generate P’ by selection, mutation, crossover**
4. **Evaluate P’**
5. **Termination**
6. **P=P’**

---

**create_MS_czj**

**Read/transform constraints into MS_czj**

---

**fset**

---

**Generate popul. of programs P**

---

**P=P’**

---

**MS_czj**

---

**P=P’**

---

**Terminate**

---

**modified module**
3.4 Formalization and Analysis (CGP1.1)

3.4.1 Definitions of constraints

- Define the following $T_{specs}$:
  
  - $T_{Root}$
    - the set of functions which return data type compatible with the problem specification
    - functions that can label the $Root$ according to data type
  
  - $T_{i}^{j}$
    - the set of functions compatible with the $j$th argument of $f_{i}$
    - functions that can label the $j^{th}$ child node of a node labeled with $f_{i}$ according to data type
Example 1
Assume

\[ F_I = \{ f_1, f_2, f_3 \} \] with arities 3, 2, and 1.

- \( f_1 \) takes boolean and two integers, respectively, and returns a real
- \( f_2 \) takes two reals and returns a real
- \( f_3 \) takes a real and returns an integer

\[ F_{II} = \{ f_4 \} \] - reads an integer.

\[ F_{III} = \{ f_5, f_6, f_7 \} \] - generate random boolean, integer, and real, respectively.

The problem states that a solution program should compute a real number.

Integers are compatible with reals
Booleans are not compatible with either.

Then, these are Tspecs:

\[ T^{Root} = \{ f_1, f_2, f_3, f_4, f_6, f_7 \} \]

\[ T_1^1 = \{ f_5 \}, T_1^2 = \{ f_3, f_4, f_6 \}, T_1^3 = \{ f_3, f_4, f_6 \} \]

\[ T_2^1 = \{ f_1, f_2, f_3, f_4, f_6, f_7 \}, T_2^2 = \{ f_1, f_2, f_3, f_4, f_6, f_7 \} \]

\[ T_3^1 = \{ f_1, f_2, f_3, f_4, f_6, f_7 \} \]
• Define the following $F_{specs}$:

  - $F^{Root}$
    - the set of functions disallowed at the Root
  - $F_i$
    - the set of functions disallowed as callers to $f_i$
  - $F^j_i$
    - the set of functions disallowed as $arg_j$ to $f_i$
Example 2
Continue Example 1. Assume
We know that the sensor reading function \( f_4 \) does not provide the solution to our problem.
We also know that boolean (generated by \( f_5 \)) cannot be the answer

(this information can be inferred from \( Tspecs \);
however, it will be easier for the user if no specific requirements are made as to how to specify non-redundant constraints)

For some semantic reasons we wish to exclude \( f_3 \) from calling itself.

\( Fspecs \) (the other sets are empty):

\[
F_{Root} = \{ f_4, f_5 \} \\
F_3 = \{ f_3 \} 
\]
Example 3  \( F_{\text{add}} = \sin \cos \)
This prevents \( \sin() \) and \( \cos() \) from calling \( \text{add}() \)
(\text{e.g. } \sin(x+y) \text{ is not allowed}).

Example 4  \( F_{\text{add}_0} = 0 \ PI \ log \)
This prevents 0, \( \Pi \), \( \log() \) from being argument 0 of \( \text{add}() \)
(\text{e.g. } (\Pi + x) \text{ is not allowed, but } (x + \Pi) \text{ may be}).

Example 5  \( T_{\text{add}_0} = \sin \cos 0 \ PI \)
This allows \( \sin() \), \( \cos() \), 0, and \( \Pi \) be argument 0 of \( \text{add}() \).
However, the above \( F_{\text{add}_0} \) overrides this \( Tspec \), and so only \( \sin() \) and \( \cos() \) are actually allowed.
3.4.2 Normal form

- $Tspecs$ and $Fspecs$ are redundant
- they can be inconsistent
- need to generate a normal form
  - extend $Fspecs$ using $Tspecs$
  - remove inconsistencies from $Fspecs$
3.4.2.1 Extend $Fs$ using $Ts$

- The following are valid inferences for extending $Fs$ from $Ts$:

\[ \forall (f_k \in F)(f_k \notin T^j_i \rightarrow f_k \in F^j_i) \]
\[ \forall (f_k \in F)(f_k \notin T^{Root} \rightarrow f_k \in F^{Root}) \]

- $Ts$ can be expressed with $Fs$

- $Fs$ are stronger
• If $F$specs do not satisfy the above for any function
  ■ call them $T$-intensive $F$specs

• $T$-intensive $F$specs list only semantics-based constraints which cannot be inferred from data types

• If $F$specs explicitly satisfy the above
  ■ call them $T$-extensive $F$specs

• $T$-extensive $F$specs are sufficient to express all $T$specs
3.4.2.2 Remove redundancies among *Fspects*

- Suppose \( f_k \in F \) and *Fspects* are \( T \)-extensive. Then

\[
\forall (f_i \in F)(f_k \in F_i \leftrightarrow \forall (j \in [1, a_k])(f_i \in F^j_k))
\]

- However, \( F_* \) and \( F'_* \) are not equivalent
  - a function may be allowed on some arguments but not on others.
  - \( F'_* \) *Fspects* are stronger
• If $F_{specs}$ include both forms of the equivalency
  ■ call them **F-extensive $F_{specs}$**

• Dropping all $F_*$ from the **F-extensive $F_{specs}$**
  ■ gives **F-intensive $F_{specs}$**

• **T-extensive F-intensive $F_{specs}$** are sufficient to express all $F_{spec}$ constraints

• Call the **T-extensive F-intensive $F_{specs}$** the **normal form**
  ■ contains only the $F^{Root}$ and $F_* F_{specs}$

• The normal form is sufficient to express all constraints of the $T_{spec}/F_{spec}$ language
Example 6  Constraints of Example 1 and Example 2 have the following normal form:

\[
F^{Root} = \{f_4, f_5, f_6\}
\]
\[
F_1^1 = \{f_1, f_2, f_3, f_4, f_6, f_7\}
\]
\[
F_1^2 = \{f_1, f_2, f_5, f_7\}
\]
\[
F_1^3 = \{f_1, f_2, f_5, f_7\}
\]
\[
F_2^1 = \{f_5\}, F_2^2 = \{f_5\}
\]
\[
F_3^1 = \{f_3, f_5\}
\]
3.4.3 Mutation Sets

• Constraints in the normal form cannot be effectively utilized

3.4.3.1 Useless functions

• If a function from $F$ cannot label any nodes in a valid tree, call it a *useless* function
  
  ▪ A function $f_i \in F$ is *useless* iff
    
    ▪ it is a member of all sets of the normal form, or
    
    ▪ it is a member of all sets of the normal form except for only sets associated with useless functions
  
• Useless functions can be removed from $F$
3.4.3.2 Mutation sets

- Define $\mathcal{F}_N$

  - the set of functions of type I that can label (thus, excluding useless functions) node $N$ without invalidating an otherwise valid tree containing the node

- Define $\mathcal{T}_N$

  - the set of terminals that can label node $N$ the same way
• Assume the normal form for constraints, and node $N$, not being the Root and being the $j$th child of a node labeled $f_i$.
Then

$$\mathcal{T}_N = \left\{ f_k \mid f_k \not\in F_i^j \land f_k \in F_{II} \cup F_{III} \right\}$$

$$\mathcal{F}_N = \left\{ f_k \mid f_k \not\in F_i^j \land f_k \in F_I \right\}$$
• Assume the normal form for constraints and node $N$ being $Root$.

Then

\[
T_N = \left\{ f_k | f_k \notin F_{Root} \land f_k \in F_{II} \cup F_{III} \right\}
\]

\[
F_N = \left\{ f_k | f_k \notin F_{Root} \land f_k \in F_{I} \right\}
\]

• Let us denote $T_{Root}$ and $F_{Root}$
  
  • the pair of mutation sets associated with $Root$

• Let us denote $T_i^j$ and $F_i^j$
  
  • the pair of mutation sets for the $j^{th}$ child of a node labeled with $f_i$
For an application problem

- there are exactly $1 + \sum_{i=1}^{\left| F_1 \right|} (a_i)$ mutation set pairs

- the normal form can be expressed with $2 \cdot (1 + \sum_{i=1}^{\left| F_1 \right|} a_i)$ different sets

- only two sets (one pair) are needed in \textit{lil-gp} itself

Example 7
Here are selected examples of mutation sets generated for Example 6:

$$T_{\text{Root}} = \{ f_7 \}, \ F_{\text{Root}} = \{ f_1, f_2, f_3 \}$$

$$T^1_3 = \{ f_4, f_6, f_7 \}, \ F^1_3 = \{ f_1, f_2 \}$$
3.4.4 Closed Search

• How are these sets sufficient to close lil-gp’s search in the space of constraints?

  - initialize with constraints-valid trees only
  - ensure constraints-valid offspring from constraints-valid parents

• Will we always have mutation sets?

  - for any node $N$ of a valid program at least one of the two mutation sets is guaranteed not to be empty
• Can we ensure that non-empty finite trees exist for arbitrary constraints?

  - non-empty trees do exist because one of the pairs for the \textit{Root} is guaranteed not to be empty
  - infinite trees are possible if $\exists (j \in a_i) F_i^j = F \setminus \{f_i\}$
    - \textit{e.g.}, when $\mathcal{T}_N = \emptyset$, $\mathcal{F}_N = \{f_i\}$
- a subtree whose root is labeled $f_i$ can be finitely instantiated without $F^+ \subseteq F_I$ iff finite valid trees without labels from $F^+$ do exist

- for each subtree we may check if for mutation sets of its root finite trees do exist
  - can be pre-done
  - (details skipped)

- better approach is that of STGP
  - take depth into account
  - to be added into CGP3.0
3.4.4.1 **CGP’s mutation**

- To mutate a node $N$
  - determine the kind of the node
    - *Root*
    - otherwise what the label of the parent is and which child of that parent $N$ is
  - if the growth is to continue
    - label the node with a random element of $\mathcal{F}_N$ and continue growing the proper number of subtrees, each grown recursively
    - if $\mathcal{F}_N = \emptyset$ then select a member of $\mathcal{T}_N$
      (guaranteed not to be empty now)
if the growth is to stop

- select a random element of $\mathcal{T}_N$ and instantiate it if from $F_{III}$ (stop expanding $N$)
- if $\mathcal{T}_N = \emptyset$ then select a member of $\mathcal{F}_N$
  (this will unfortunately extend the tree, but it is guaranteed to eventually stop)

- If a valid tree is selected for mutation, mutation will always produce a valid tree
  - how complex this closed mutation is?
    - negligible constant overhead
Example 8 Assume

The mutation sets of Example 7.

Mutating *parent1* as in Figure 1.

Node \( N \) is selected for mutation.

It is the 1st child of a node labeled with \( f_3 \)

Thus

\[
T_N = T_3^1 = \{f_4, f_6, f_7\} \quad F_N = F_3^1 = \{f_1, f_2\}.
\]

- if to grow the tree, then the mutated node will be randomly labeled with either \( f_1 \) or \( f_2 \)

- if the current node is to generate a leaf, then label \( N \) with either of \( f_4, f_6, f_7 \)
3.4.4.2  *CGP lil-gp* initialization

- Assume that \( T_{\text{Root}} \neq \emptyset \lor F_{\text{Root}} \neq \emptyset \) and that functions which can only label trees which cannot be constructed- \( \emptyset \) are removed from the mutation sets.

- To generate a valid random tree:
  - create the *Root* node
  - mutate it using the mutation operator

- The above will create a tree:
  - with at least one node
  - finite and valid with respect to the constraints
3.4.4.3 **CGP lil-gp crossover**

- To move genetic material from parent2 to parent1-node \( N \)
  - take \( F_N \) and \( T_N \)
  - assume that \( F_2 \) is the set of labels appearing in parent2
  - then \( (F_N \cup T_N) \cap F_2 \) is the set of labels determining which subtrees from parent2 can \( N \)
• If two constraints-valid trees are selected for crossover, the operator will always produce a constraints-valid tree

  • this is done with only the same (order) computational complexity (one more tree traversal)

**Figure 1** Illustration of mutation and crossover.
Example 9  Assume
The mutation sets of Example 7. Then

- \( \mathcal{T}_N = \mathcal{T}_3^1 = \{f_4, f_6, f_7\} \), \( \mathcal{F}_N = \mathcal{F}_3^1 = \{f_1, f_2\} \)

- only the subtrees with the shaded roots can be used to replace \( N \)

- crossover would select a random element from a so marked set of nodes, and copy the corresponding subtree.
3.5 Recent Extensions (in CGP2.1)

- Types for explicit type-based constraints
- Weights for heuristic constraints
- Overloaded functions
- Simplified notation for CSL with file-input
3.5.1 Types

- Following STGP, constraints can be inferred from *types*
  - returning types of all functions are specified
  - types allowed as function arguments are specified
  - constraints are automatically generated
• **CGP2.1** uses type constraints on the top of *Fspecs*/
  *Tspecs*

  - types are alternatives to *Tspecs* (type-based syntactic constraints)
  - types can augment constraints with something the user misses
  - with/without *Tspecs*, types extend *Fspecs*
  - *Fspecs* remain as semantic-based constraints
3.5.2 Heuristic Constraints with Weights

- Weights allow for more flexible and liberal constraints
  - allow some functions/types/terminals to be used more/less frequently than others
  - CGP3.0 will evolve weights as a means of evolving representation
Example 10 Assume
4 functions \((f_1, f_2, f_3, f_4)\) and 4 terminals \((t_1, t_2, t_3, t_4)\).
Through \(Fspecs\), \(t_2\) was excluded from being an argument to function \(f_n\).
The weights are given as
\[
\begin{align*}
  f_1 = 1.0, & \quad f_2 = 0.0, \quad f_3 = 0.1, \\
  t_1 = 1.0, & \quad t_3 = 1.0, \quad t_4 = 1.0.
\end{align*}
\]
- The probability of selecting individual elements is
  \[
  \begin{align*}
    p(f_1, t_1, t_3, t_4) &= 1.0/4.1 = 0.244 \\
    p(f_2) &= \text{MINWGT}/4.1 \sim 0 \\
    p(f_3) &= 0.1/4.1 = 0.0244 \\
    p(t_2) &= 0.0.
  \end{align*}
  \]
3.5.3 Overloaded Functions

- Extra flexibility in constraining the space
  
  - a function can be defined to generate different types based on the types of its arguments
  
  - only proper function “instances” can now participate in mutation and crossover
Example 11  Assume

In a given domain the following data types exist

\textit{angle, length, force, force-length, and number.}

\texttt{multiply()} takes 2 arguments

\begin{itemize}
  \item it can be overloaded as follows:
    \begin{tabular}{lll}
      \texttt{<arg1>} & \texttt{<arg2>} & \texttt{<return>} \\
      number & length & length \\
      length & number & length \\
      number & angle & angle \\
      angle & number & angle \\
      number & number & number \\
      length & force & force-length \\
      force & length & force-length \\
    \end{tabular}
\end{itemize}
3.5.4 Simplified CSL

- **CGP1.1** used interactive-input only for listing ALL constraints

- **CGP2.1** allows listing multiple constraints with a single specification
  - similar notation is used for weights and types
3.6 CGP2.1 and Constraints Examples

- http://www.cs.umsl.edu/~janikow/cgp-lilgp

3.6.1 Syntax of Simplified CSL

FTSPEC  #Section Header
F_(funclist | *) = funclist | * | null
F_(funclist | *)[arglist | *]
  = functermlist | * | null
T_(funclist | *)[arglist | *]
  = functermlist | * | null
F_ROOT = functermlist | * | null
T_ROOT = functermlist | * | null
ENDSECTION #Section Footer
WEIGHT #Section Header

\[(funclist \mid *)[arglist \mid *)(functermlist \mid *) = weightlist\]

ROOT\(\text{functermlist} \mid *) = weightlist\)

ENDSECTION #Section Footer

TYPE #Section Header

\text{TYPELIST} = typelist #defines valid types

\[(funclist)(argtypelist) = type\]

\[(termlist \mid *) = type\]

ROOT = type

ENDSECTION #Section Footer
Example 12
Simplified CSL for the inverse kinematics problem

FTSPEC

F_(*) =  #not required since it’s empty
F_(*)[*] =  #not required since it’s empty
F_(sin)[0]=add  #prevent sin(_+_)
F_ROOT=asin  #prevent asin() from being Root

#must specify some TSpecs
T_(*)[*]=*  #allow all TSpecs
T_ROOT=*  #allow all funs/term for Root

ENDSECTION
WEIGHT

# All unspecified weights default to 1.0

# Set the weights for the functions: add
#  asin sin 1 PI x y, as the arguments for
# the add & asin functions.

(add asin)[*](*)=.25 .25 .5 .2 .2 .3 .4

# similarly for the sin function

(sin)[0](*)=.5 .4 .3 .6 .4 .3 .1

ROOT(*)=1  # not needed as default is 1.0

ROOT(PI)=.2

ENDSECTION
TYPE

TYPELIST = float integer angle

(add)(float float)=float

(add)(integer float)=float

(add)(float integer)=float

(add)(integer integer)=integer

(add)(angle angle)=angle

(asin)(float)=angle

(asin)(integer)=angle

(sin)(angle)=float

(1)=integer

(PI)=angle

(x y)=float

ROOT=angle   #Root return type

ENDSECTION
4 Comparison with STGP

4.1 Similarities

- Developed at the same time
- Same motivations
  - constrain the evolution space (relax closure)
  - use syntax-related constraints based on types
- Same methods
  - evolve constraint-valid population
  - use constraint-preserving mutation/crossover to close the search space
4.2 Differences

- STPG provides level-specific mutation/crossover to ensure depth-limited trees

- CGP initially used explicit syntax (type) and semantic-based compatibility rather than explicit types
  - types added in CGP2.1
  - STGP uses types exclusively
• CGP allows semantic constraints, which may not be inferred from types
  
  ■ allows more detailed manipulation of the evolved structures

  ■ STGP can only reduce the search space to that based on syntax (types) and cannot reduce the space any more
• CGP’s crossover is capable of generating more constraint-valid trees

![Diagram]

- CGP uses parent’s information
- STGP uses a given node’s information
• CGP provides formal analysis of
  
  ▪ the transformation from constraints to the definition of mutation sets
  
  ▪ formal proof of the closed space
  
  ▪ complexity analysis
    
    - negligible computational overhead in performing constrained mutation/crossover
• CGP provides overloaded functions

• CGP provides means of processing heuristic constraints
  • weights
  • will allow to evolve representation

• CGP is plugged-in into \texttt{lil-gp} 1.02 to take advantage of its capabilities (minus ADFs)
5 Example Applications

• Multiplexer
  - varying redundancy
  - removing redundancy
  - evolving representation

• Machine learning
5.1 11-Multiplexer

- Boolean function

\[
\begin{align*}
& a_0..a_2 \\
& d_0..d_7
\end{align*}
\]

- Analytical DNF formula:

\[
\begin{align*}
& a_2a_1a_0d_7 \vee a_2a_1\overline{a_0}d_6 \vee a_2\overline{a_1}a_0d_5 \vee a_2\overline{a_1}\overline{a_0}d_4 \vee \\
& \overline{a_2a_1}\overline{a_0}d_3 \vee \overline{a_2a_1}a_0d_2 \vee a_2a_1a_0d_1 \vee a_2a_1\overline{a_0}d_0
\end{align*}
\]
• Other forms do exist (in the same DNF and in other representations)

• Functions

\[ F_I = \{if, or, and, not\} \]
\[ F_{II} = \{a_0…a_2, d_0…d_7\} \]
• Experiments

  - base: Unconstrained 11-multiplexer with *lil-gp*

    \[ T_{Root}^{*} = T_{*} = \{ \text{if, or, and, not, } a_0 \ldots a_2, d_0 \ldots d_7 \} \]

  - \( E_0 \): Unconstrained 11-multiplexer with *CGP lil-gp*

    \[ T_{Root}^{*} = T_{*} = \{ \text{if, or, and, not, } a_0 \ldots a_2, d_0 \ldots d_7 \} \]

  - \( E_1 \): using sufficient set \{*and*,*not*\}

    \[ F_{Root}^{*} = F_{*} = \{ \text{if, or} \} \quad F_{*} = \emptyset \]
- **$E_2$: DNF**

  \[
  F^{\text{Root}} = \{ \text{if} \} \quad F_* = \emptyset \\
  F^*_\text{if} = \emptyset \quad F^*_\text{not} = \{ \text{if, or, and, not} \} \\
  F^*_\text{and} = \{ \text{if, or} \} \quad F^*_\text{or} = \{ \text{if} \}
  \]

- **$E_3$: Structure-restricted DNF**

  \[
  F^{\text{Root}} = \{ \text{if} \} \quad F_* = \emptyset \\
  F^*_\text{if} = \emptyset \quad F^*_\text{not} = \{ \text{if, or, and, not} \} \\
  F^1_\text{and} = \{ \text{if, or} \} \quad F^2_\text{and} = \{ \text{if, or, and} \} \\
  F^1_\text{or} = \{ \text{if} \} \quad F^2_\text{or} = \{ \text{if, or} \}
  \]
- $E_4$: using \{if\} only

$$F_{\text{Root}}^* = F_{\text{if}}^* = \{\text{or, and, not}\} \quad F_* = \emptyset$$

$$F_{\text{or}}^* = F_{\text{and}}^* = F_{\text{not}}^* = \text{irrelevant}$$

- $E_5$: $E_4$ with problem-specific knowledge
  - know that first 3 bits are addresses, others are data

$$F_{\text{Root}}^* = \{\text{or, and, not, } a_0, a_1, a_2\} \quad F_* = \emptyset$$

$$F_{\text{if}}^1 = \{\text{or, and, not, } d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$$

$$F_{\text{if}}^2 = F_{\text{if}}^3 = \{\text{or, and, not, } a_0, a_1, a_2\}$$

$$F_{\text{or}}^* = F_{\text{and}}^* = F_{\text{not}}^* = \text{irrelevant}$$
- $E_6$: $E_5$ with further heuristic knowledge
  - same as above, except do not recurse on \{if\}

$$F_{\text{Root}} = \{ or, and, not, a_0, a_1, a_2 \} \quad F^* = \emptyset$$

$$F_{if}^1 = \{ if, or, and, not, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7 \}$$

$$F_{if}^2 = F_{if}^3 = \{ or, and, not, a_0, a_1, a_2 \}$$

$$F_{or} = F_{and} = F_{not} = \text{irrelevant}$$

![Diagram showing a decision tree with an `if` node, an `a? or not(a?)` node, and two `compute a data bit` nodes.]
- $E_7$: $E_6$ relaxed

\[ F^{\text{Root}} = \{\text{or, and, not, } a_0, a_1, a_2\} \]

\[ F_* = \emptyset \quad F^*_{\text{or}} = F^*_{\text{and}} \text{ is irrelevant} \]

\[ F^1_{\text{if}} = \{\text{if, or, and, } d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\} \]

\[ F^2_{\text{if}} = F^3_{\text{if}} = \{\text{or, and, not, } a_0, a_1, a_2\} \]

\[ F^1_{\text{not}} = F - \{a_0, a_1, a_2\} \]
Figure 2 Comparison of the quality of the best-of-population tree.
**Figure 3** Comparison of complexity needed for evolving solutions in 100 generations (complexity 0 used on finished runs).
• Other runs

  • tried different ways to penalize
    □ constant
    □ generation-dependent
    □ did not help

  • tried different initialization methods
    □ somehow helped if constraint-invalid trees were thrown away
• Conclusions from this experiment

  ■ invalid subspaces should be removed

  ■ redundant subspaces are “better” than invalid

  ■ removal of redundant subspaces “may” improve evolution

  ■ some redundant subspaces are better than others

  ■ pays off to know which representation subspaces are better for the evolution

  ▪ need to evolve if unknown
5.2 Machine Learning

- Given attribute-based space

<table>
<thead>
<tr>
<th>Attribute (abbr.)</th>
<th>Domain values (abbreviations.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head (HE)</td>
<td>Round, Square, Octagon (R,S,O)</td>
</tr>
<tr>
<td>Body (BO)</td>
<td>Round, Square, Octagon (R,S,O)</td>
</tr>
<tr>
<td>Smiling (SM)</td>
<td>Yes, No (Y,N)</td>
</tr>
<tr>
<td>Holding (HO)</td>
<td>Sword, Balloon, Flag (S,B,F)</td>
</tr>
<tr>
<td>Jacket (JA)</td>
<td>Red, Yellow, Green, Blue (R,Y,G,B)</td>
</tr>
<tr>
<td>Tie (TI)</td>
<td>Yes, No (Y,N)</td>
</tr>
</tbody>
</table>

- Learn this concept

  - Head is Round and Jacket is Red

    - or

  - Head is Square and Holding a Balloon
• Use $VL_1$

  ■ condition
  of one variable with internal disjunctions

  ■ rule
  conjunction of unique-variable rules

  ■ concept
  set of rules

  ■ $[HE=R][JA=R] \lor [HE=S][HO=B]$
• Complex constraints imposed by VL$_1$
  
  - [HE=Y] is an invalid condition
  
  - [HE=R][HE=S] is an invalid rule
  
  - GP, under closure, would assign some interpretations to such cases
    
    - arbitrary redundancy
    
    - tailored approach could also be taken
  
  - CGP can do it with constraints only
• Assume \textit{eval()} denotes the result of evaluating an argument

  - \textit{ifAnd}(a_1, a_2, a_3)
    - return \((\text{eval}(a_1) \in \text{eval}(a_2)) \land \text{eval}(a_3)\)

  - \textit{or}(a_1, a_2)
    - return \(\text{eval}(a_1) \lor \text{eval}(a_2)\)

  - \textit{read()}
    - return the current value of a variable

  - \textit{genSet()}
    - return a subset of the domain of a variable

  - \textit{bool()}
    - return a random boolean.
• *ifAnd, read, genSet* will have to have individual copies for each variable

- assume that *ifAnd*$_i$, $i \in [1, 6]$ is the function corresponding to the $i^{th}$ variable

- assuming some arbitrary consistent enumeration

- assume the same for *read*$_i$ and *genSet*$_i$

- when referring to all individual functions, the indexes will be dropped

- this extension may seem unnecessary, but the extended function set will enable us to manipulate the structures being evolved in a very precise way
• **Constraints**

- only functions returning boolean can be *Root:*

  \[ T^{Root} = \{ \text{ifAnd, or, bool} \} \]

- each condition refers to the same attribute (not necessarily unique yet in a rule):

  \[
  \forall (i \in [1, 6]) \begin{cases}
  T^1_{ifAnd_i} = \{ \text{read}_i \} \\
  T^2_{ifAnd_i} = \{ \text{genSet}_i \} \\
  T^3_{ifAnd_i} = \{ \text{ifAnd, bool} \}
  \end{cases}
  \]
- how different rules will be built?
  
  - no restrictions needed for VL₁
  
  - alternatively we can allow only left-recursive disjunctions (to reduce redundancy)

\[ T_{or}^1 = \{ \text{ifAnd, or, bool} \}, T_{or}^2 = \{ \text{ifAnd, bool} \} \]
the above does not yet guarantee valid nor efficient DFN expressions

- a single rule the same variable may appear with a number of possibly contradictory (or redundant) conditions

- add the following $F_{spec}$ constraint

  $$\forall (i \in [1, 6])(F_{ifAnd_i} = \{ifAnd_j | (j \leq i)\})$$

- to enumerate all the conditional functions and only allow descendants with strictly smaller indexes

- now only trees equivalent to $VL_1$ expressions can be represented
**Figure 4** VL₁ tree in evolved in CGP.

- **Evaluation**
  - raw-fitness based on all 432 examples
• Experiments

  • GP1 - non-indexed functions
    ▫ VL₁-invalid trees were penalized
  
  • GP2 - GP1 with forced syntactic closure
    ▫ extended terminal set
      - one terminal per variable-value
      - as HER for \([HE=R]\)
    
     ▫ many redundant solutions
      - as \((or (and HER JAR) (and HES HOB)))\)

• CGP
Figure 5 Learning curves for the robots.
6 Summary and More Work

• Constraints

  - allow processing a limited set of constraints

  - allow closing the search in a single *one-to-one* space, or in controlled redundant spaces

  - allow exploration of various search spaces
• Weights
  - allow adding heuristic constraints

• Overloaded functions
  - allow context-dependent mutation and crossover
  - reduce the number of functions needed
• More work

  • weighted overloaded instances

  • level-specific mutation sets

    ▪ to remedy problems with too-deeply growing trees

  • evolving constraints (strong and heuristics)

    ▪ will evolve representation while evolving solutions
- extensions for ADFs
  - needed for evolving modular solutions and computer programs
  - needs to pass constraints between module calls
• Do we always need to constrain?
  
  ■ not necessarily

  ■ one space may have better search characteristics than another

  ■ redundancy can improve search

  ■ constraints evolution should help
7 More Information


http://GARAGE.cps.msu.edu/software/software-index.html#lilgp