Brock University 50 1964-2014

Faculty of Mathematics and Science

Incorporating Expert Knowledge in Object-Oriented Genetic Programming

Michael Richard Medland Kyle Robert Harrison Beatrice Ombuki-Berman

Department of Computer Science, Brock University, Saint Catharines, Canada

Abstract

Genetic programming (GP) has proven to be successful at generating programs which solve a wide variety of problems. Object-oriented GP (OOGP) extends traditional GP by allowing the simultaneous evolution of multiple program trees, and thus multiple functions. OOGP has been shown to be capable of evolving more complex structures than traditional GP. However, OOGP does not facilitate the incorporation of expert knowledge within the resulting evolved type. This paper proposes an alternative OOGP methodology which does incorporate expert knowledge by the use of a user-supplied, partially-implemented type definition, i.e. an abstract class.

Background

Types of Genetic Programming

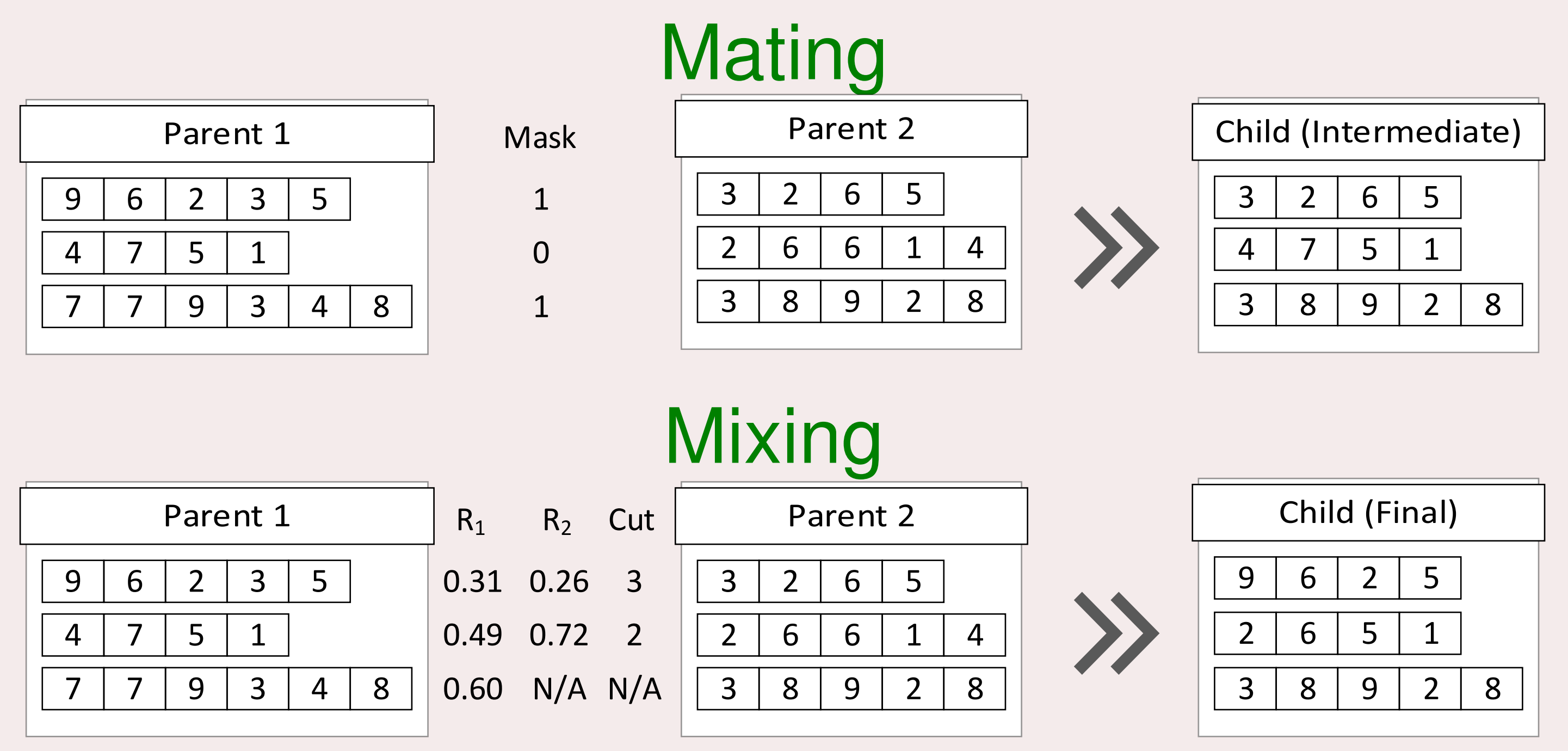
- Tree Based Genetic Programming (Koza-Style)
  - Produces Single S-Expression Trees
  - Facilitates a functional-style program
  - Difficult to represent a state-memory
- Linear Genetic Programming
  - Captures the imperative programming paradigm
  - Uses an array-based chromosome structure
  - Difficult to model multiple behaviours
- Object-Oriented Genetic Programming
  - Multi-tree representation
  - Each tree represents a behaviour

Objective

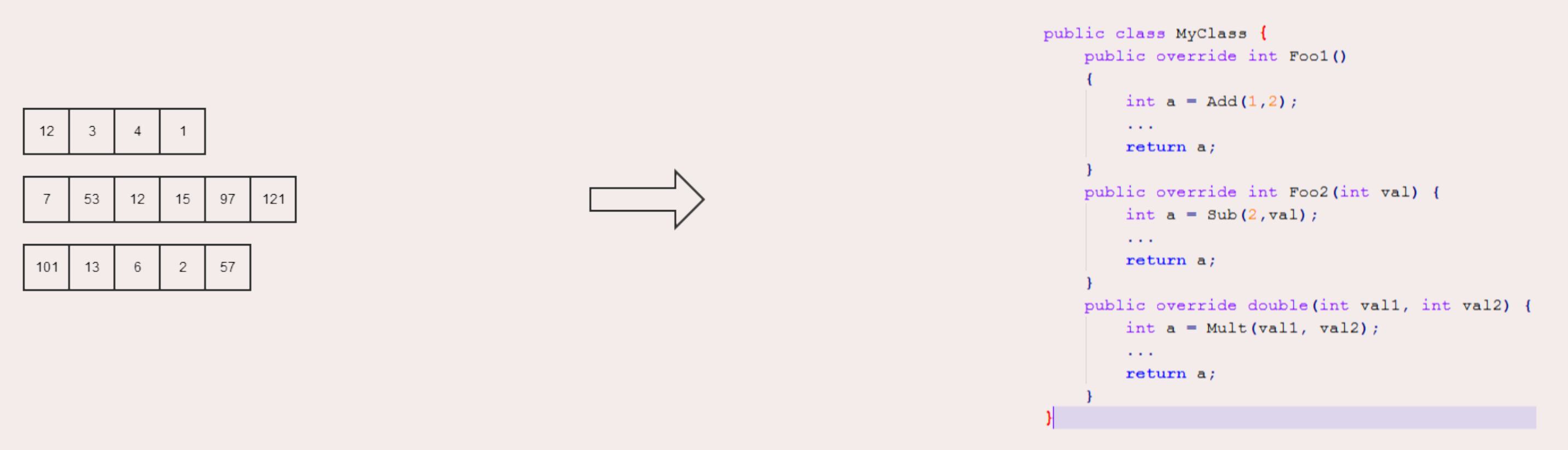
This work proposes a novel GP paradigm which combines the imperative style of linear GP with the object-oriented approach of OOGP while allowing the incorporation of expert knowledge in the resulting programs. The proposed GP system, LinkableGP, makes use of a partially-implemented type definition, i.e., an abstract class, to allow expert knowledge to be embedded within the evolved program.

Genotype

- The Genotype of an individual consists of a number of chromosomes determined by the number of abstract functions in the Phenotype Abstract Class.
- Each chromosome is an array of integers.
- Crossover consists of 2 phases
  - **Mating** where a bitmask determines which chromosomes are inherited by the child
  - **Mixing** – where one point crossover operations are performed on randomly selected chromosome pairs



Phenotype



```
public class MyClass {
    public override int Foo1()
    {
        int a = Add(1,2);
        ...
        return a;
    }
    public override int Foo2(int val) {
        int a = Sub(2,val);
        ...
        return a;
    }
    public override double(int val1, int val2) {
        int a = Mult(val1, val2);
        ...
        return a;
    }
}
```

Figure: Visualization of Genotype to Phenotype Mapping

- Each individual is converted from a genotype to its phenotype by mapping each chromosome to an abstract function.
- Each function is built using a set of functions and terminals, called the language.
- Terminals consist of constants, input parameters, and declared variables during the mapping of the abstract function.
- The abstract function is mapped by selecting functions from the language who's arguments can be satisfied with the terminals in the language.
- If a function has a non-void result, then a variable is either selected from the mutable terminals with the resulting type, or a new variable of the correct type is created.
  - If the new variable is selected, then it is added to the available terminals.

Example Phenotype Abstract Class

```
abstract function initializeGraph()
abstract function selectVertices(graph g, int n)
abstract function createEdge(graph g, vertex v, vertex s)
function GeneralizedGraphModel(int t)
  Result: A graph
  $g \leftarrow initializeGraph()$;
  for $i \in 1 : t$ do
    $v \leftarrow g.AddVertex()$, $n \leftarrow selectNumber()$;
    $S \leftarrow selectVertices(g, n)$;
    for $s \in S$ do
      $createEdge(g, v, s)$;
    end
  end
  return $g$;
```

Experiments

Evolving Stacks and Queues

- Evolved add, remove, and peek functions
- Evaluated based on ability to mimic behaviours (120pts)
- 30 runs performed

Table: Results for Evolving Stack and Queue Data Structures

Result	Stack	Queue
Number of Optimal Solutions Found	28	19
Average Iterations to Obtain an Optimal Solution	28.11	40.53
Average Fitness of Non-Optimal Solutions	101.00	97.82

Conclusion

This work proposed a novel genetic programming (GP) paradigm which facilitates the incorporation of expert knowledge within the evolved program structure. The proposed GP system, LinkableGP, was inspired by both linear GP and object-oriented GP (OOGP) methodologies. However, the LinkableGP system combined the benefits of each approach by allowing the simultaneous evolution of multiple, imperative-styled methods. LinkableGP uses a representation whereby an individual is comprised of multiple chromosomes, each of which directly correspond to a method which is to be evolved. Furthermore, LinkableGP facilitates expert knowledge through partially-implemented types, allowing the user to embed portions of the solution known *a priori* within the resulting individuals.

Addtional Information

LinkableGP is available at http://linkablegp.sourceforge.net

Michael R. Medland .... mm08sj@brocku.ca
Kyle R. Harrison .... kh08uh@brocku.ca
Beatrice Ombuki-Berman .... bombuki@brocku.ca