
White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/129963/

Version: Accepted Version

Proceedings Paper:
Atkinson, Timothy, Plump, Detlef and Stepney, Susan (2018) Probabilistic Graph Programs for Randomised and Evolutionary Algorithms. In: Weber, Jens and Lambers, Leen, (eds.) Graph Transformation - 11th International Conference, ICGT 2018, Held as Part of STAF 2018, Proceedings. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, pp. 63-78.

https://doi.org/10.1007/978-3-319-92991-0_5

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.



Probabilistic Graph Programs for Randomised and Evolutionary Algorithms

Timothy Atkinson*, Detlef Plump, and Susan Stepney

Department of Computer Science, University of York, UK
{tja511,detlef.plump,susan.stepney}@york.ac.uk

Abstract. We extend the graph programming language GP 2 with probabilistic constructs: (1) choosing rules according to user-defined probabilities and (2) choosing rule matches uniformly at random. We demonstrate these features with graph programs for randomised and evolutionary algorithms. First, we implement Karger's minimum cut algorithm, which contracts randomly selected edges; the program finds a minimum cut with high probability. Second, we generate random graphs according to the $G(n, p)$ model. Third, we apply probabilistic graph programming to evolutionary algorithms working on graphs; we benchmark odd-parity digital circuit problems and show that our approach significantly outperforms the established approach of Cartesian Genetic Programming.

1 Introduction

GP 2 is a rule-based graph programming language which frees programmers from handling low-level data structures for graphs. The language comes with a concise formal semantics and aims to support formal reasoning on programs; see, for example, [22, 19, 11]. The semantics of GP 2 is nondeterministic in two respects: to execute a rule set $\{r_1, \ldots, r_n\}$ on a host graph G, any of the rules applicable to G can be picked and applied; and to apply a rule r, any of the valid matches of r's left-hand side in the host graph can be chosen. GP 2's compiler [4] has been designed by prioritising speed over completeness, thus it simply chooses the first applicable rule in textual order and the first match that is found.

For some algorithms, compiled GP 2 programs reach the performance of hand-crafted C programs. For example, [1] contains a 2-colouring program whose runtime on input graphs of bounded degree matches the runtime of Sedgewick's program in *Graph Algorithms in C*. Clearly, this implementation of GP 2 is not meant to produce different results for the same input or make random choices with pre-defined probabilities.

However, probabilistic choice is a powerful algorithmic concept which is essential to both *randomised* and *evolutionary* algorithms. Randomised algorithms take a source of random numbers in addition to input and make random choices during execution. There are many problems for which a randomised algorithm

* Supported by a Doctoral Training Grant from the Engineering and Physical Sciences Research Council (EPSRC) in the UK.

is simpler or faster than a conventional deterministic algorithm [18]. Evolutionary algorithms, on the other hand, can be seen as randomised heuristic search methods employing the generate-and-test principle. They drive the search process by variation and selection operators which involve random choices [6]. The existence and practicality of these probabilistic algorithms motivates the extension of graph programming languages to the probabilistic domain. Note that our motivation is different from existing simulation-driven extensions of graph transformation [10, 14]: we propose high-level *programming* with probabilistic constructs rather than *specifying* probabilistic models.

To cover algorithms on graphs that make random choices, we define *Probabilistic GP 2* (P-GP 2) by extending GP 2 with two constructs: (1) choosing rules according to user-defined probabilities and (2) choosing rule matches uniformly at random. We build on our preliminary GP 2 extension [1], where all rule sets are executed by selecting rules and matches uniformly at random. In contrast, we propose here to extend GP 2 conservatively and allow programmers to use both probabilistic and conventional execution of rule sets. In addition, weighted rules can be used to define more complex probability distributions.

We present three case studies in which we apply P-GP 2 to randomised and evolutionary algorithms. The first example is Karger's randomised algorithm for finding a minimum cut in a graph [12]. Our implementation of the algorithm comes with a probabilistic analysis, which guarantees a high probability that the cut computed by the program is minimal. The second example is sampling from Gilbert's $G(n, p)$ random graph model [9]. The program generates random graphs with n vertices such that each possible edge occurs with probability p.

To our knowledge, these graph programs are the first implementations of the randomised algorithms using graph transformation. Our final case study is a novel approach to evolving graphs by graph programming [2]. We use graphs to represent individuals and graph programs as probabilistic mutation operators. Whereas our examples of randomised algorithms allow to analyse the probabilities of their results, performance guarantees for evolutionary algorithms are difficult to derive and we therefore turn to empirical evaluation. We use the well established approach of Cartesian Genetic Programming (CGP) as a benchmark for a set of digital circuit synthesis problems and show that our approach outperforms CGP significantly.

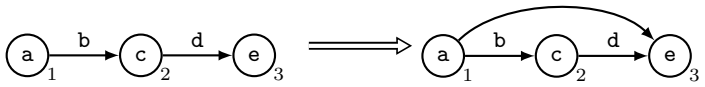
The rest of this paper is arranged as follows. Section 2 introduces the graph programming language GP 2, and Section 3 explains our probabilistic extension to GP 2. Sections 4 and 5 detail our applications of this extension to randomised and evolutionary algorithms, respectively. Section 6 summarises this work and proposes future topics of work.

2 Graph Programming with GP 2

This section briefly introduces the graph programming language GP 2; see [20] for a detailed account of the syntax and semantics of the language, and [4] for its implementation. A graph program consists of declarations of graph transfor-

```
Main := link!

link(a,b,c,d,e:list)
```




```
where not edge(1,3)
```

Fig. 1. A GP 2 program computing the transitive closure of a graph.

mation rules and a main command sequence controlling the application of the rules. The rules operate on host graphs whose nodes and edges are labelled with integers, character strings or lists of integers and strings. Variables in rules are of type `int`, `char`, `string`, `atom` or `list`, where `atom` is the union of `int` and `string`. Atoms are considered as lists of length one, hence integers and strings are also lists. For example, in Figure 1, the list variables `a`, `c` and `e` are used as edge labels while `b` and `d` serve as node labels. The small numbers attached to nodes are identifiers that specify the correspondence between the nodes in the left and the right graph of the rule.

Besides carrying list expressions, nodes and edges can be *marked*. For example, in the program of Figure 4, the end points of a randomly selected edge are marked blue and red to redirect all edges incident to the blue node to the red node.

The principal programming constructs in GP 2 are conditional graph-transformation rules labelled with expressions. The program in Figure 1 applies the single rule `link` *as long as possible* to a host graph. In general, any subprogram can be iterated with the postfix operator "`!`". Applying `link` amounts to nondeterministically selecting a subgraph of the host graph that matches `link`'s left graph, and adding to it an edge from node 1 to node 3 provided there is no such edge (with any label). The application condition `where not edge(1,3)` ensures that the program terminates and extends the host graph with a minimal number of edges. Rule matching is injective and involves instantiating variables with concrete values. Also, in general, any unevaluated expressions in the right-hand side of the rule are evaluated before the host graph is altered (this has no effect on the `link` rule because it does not contain operators).

Besides applying individual rules, a program may apply a rule set $\{r_1, \ldots, r_n\}$ to the host graph by nondeterministically selecting a rule r_i among the applicable rules and applying it. Further control constructs include the sequential composition $P; Q$ of programs P and Q, and the branching constructs `if` T `then` P `else` Q and `try` T `then` P `else` Q. To execute the `if`-statement, test T is executed on the host graph G and if this results in some graph, program P is executed on G. If T fails (because a rule or set of rules cannot be matched), program Q is executed on G. The `try`-statement behaves in the same way if T fails, but if T produces a graph H, then P is executed on H rather than on G.

Given any graph G, the program in Figure 1 produces the smallest transitive graph that results from adding unlabelled edges to G. (A graph is *transitive* if for each directed path from a node v_1 to another node v_2, there is an edge from v_1 to v_2.) In general, the execution of a program on a host graph may result in different graphs, fail, or diverge. The *semantics* of a program P maps each host graph to the set of all possible outcomes. GP 2 is computationally complete in that every computable function on graphs can be programmed [20].

3 P-GP 2: A Probabilistic Extension of GP 2

We present a conservative extension to GP 2, called Probabilistic GP 2 (P-GP 2), where a rule set may be executed probabilistically by using additional syntax. Rules in the set will be picked according to probabilities specified by the programmer, while the match of a selected rule will be chosen uniformly at random. When the new syntax is not used, a rule set is treated as nondeterministic and executed as in GP 2's implementation [4]. This is preferable when executing confluent rule sets where the discovery of all possible matches is expensive and unnecessary.

3.1 Probabilistic Rule Sets

To formally describe probabilistic decisions in P-GP 2, we consider the application of a rule set $\mathcal{R} = \{r_1, \ldots, r_n\}$ to some host graph G. The set of all possible rule-match pairs from $\mathcal{R}$ in G is denoted by $G^{\mathcal{R}}$:

$$G^{\mathcal{R}} = \{(r_i, g) \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some graph } H\} \tag{1}$$

We make separate decisions for choosing a rule and a match. The first decision is to choose a rule, which is made over the subset of rules in $\mathcal{R}$ that have matches in G, denoted by $\mathcal{R}^G$:

$$\mathcal{R}^G = \{r_i \mid r_i \in \mathcal{R} \text{ and } G \Rightarrow_{r_i,g} H \text{ for some match } g \text{ and graph } H\} \tag{2}$$

Once a rule $r_i \in \mathcal{R}^G$ is chosen, the second decision is to choose a match with which to apply r_i. The set of possible matches of r_i is denoted by G^{r_i}:

$$G^{r_i} = \{g \mid G \Rightarrow_{r_i,g} H \text{ for some graph } H\} \tag{3}$$

We assign a probability distribution (defined below) to $G^{\mathcal{R}}$ which is used to decide particular rule executions. This distribution, denoted by $P_{G^{\mathcal{R}}}$, has to satisfy:

$$P_{G^{\mathcal{R}}} : G^{\mathcal{R}} \to [0,1] \quad \text{such that} \quad \sum_{(r_i,g) \in G^{\mathcal{R}}} P_{G^{\mathcal{R}}}(r_i, g) = 1 \tag{4}$$

where $[0,1]$ denotes the real-valued (inclusive) interval between 0 and 1.

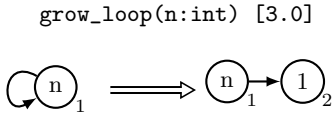


Fig. 2. A P-GP 2 declaration of a rule with associated weight 3.0. The weight is indicated in square brackets after the variable declaration.

P-GP 2 allows the programmer to specify $P_{G^{\mathcal{R}}}$ by rule declarations in which the rule can be associated with a real-valued positive weight. This weight is listed in square brackets after the rule's variable declarations, as shown in Figure 2. This syntax is optional and if a rule's weight is omitted, the weight is 1.0 by default. In the following we use the notation $w(r)$ for the positive real value associated with any rule r in the program.

To indicate that the call of a rule set $\{r_1, \ldots, r_n\}$ should be executed probabilistically, the call is written with square brackets:

$$[r_1, \ldots, r_n] \tag{5}$$

This includes the case of a probabilistic call of a single rule r, written $[r]$, which ignores any weight associated with r and simply chooses a match for r uniformly at random. Given a probabilistic rule set call $\mathcal{R} = [r_1, \ldots, r_n]$, the probability distribution $P_{G^{\mathcal{R}}}$ is defined as follows. The summed weight of all rules with matches in G is $\sum_{r_x \in \mathcal{R}^G} w(r_x)$, and the weighted distribution over rules in $\mathcal{R}^G$ assigns to each rule $r_i \in \mathcal{R}^G$ the following probability:

$$\frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)} \tag{6}$$

The uniform distribution over the matches of each rule $r_i \in \mathcal{R}^G$ assigns the probability $1/\left|G^{r_i}\right|$ to each match $g \in G^{r_i}$. This yields the definition of $P_{G^{\mathcal{R}}}$ for all pairs $(r_i, g) \in G^{\mathcal{R}}$:

$$P_{G^{\mathcal{R}}}(r_i, g) = \frac{w(r_i)}{\sum\limits_{r_x \in \mathcal{R}^G} w(r_x)} \times \frac{1}{\left|G^{r_i}\right|} \tag{7}$$

In the implementation of P-GP 2, the probability distribution $P_{G^{\mathcal{R}}}$ decides the choice of rule and match for $\mathcal{R} = [r_1, \ldots, r_n]$ (based on a random-number generator). Note that this is correctly implemented by first choosing an applicable rule r_i according to the weights and then choosing a match for r_i uniformly at random. The set of all matches is computed at run-time using the existing search-plan method described in [3]. Note that this is an implementation decision that is not intrinsic to the design of P-GP 2.

If a rule set $\mathcal{R}$ is called using GP 2 curly-brackets syntax, execution follows the GP 2 implementation [4]. Hence our language extension is conservative; existing

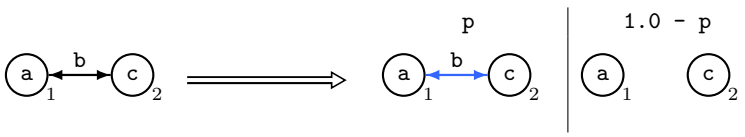


Fig. 3. A PGTS rule with multiple right-hand sides. The probability of each right-hand side is the value given above it.

GP 2 programs will execute exactly as before because probabilistic behaviour is invoked only by the new syntax. The implementation of P-GP 2 is available online[1].

3.2 Related Approaches

In this section we address three other approaches to graph transformation which incorporate probabilities. All three aim at modelling and analysing systems rather than implementing algorithms by graph programs, which is our intention. The port graph rewriting framework PORGY [8] allows to model complex systems by transforming port graphs according to strategies formulated in a dedicated language. Probability distributions similar to those in this paper can be expressed in PORGY using the `ppick` command which allows probabilistic program branching, possibly through external function calls.

Stochastic Graph Transformation Systems [10] (SGTS) are an approach to continuous-time graph transformation. Rule-match pairs are associated with continuous probability functions describing their probability of executing within a given time window. While the continuous time model is clearly distinct to our approach, the application rates associated with rules in SGTS describe similar biases in probabilistic rule choice as our approach.

Closest to our approach are Probabilistic Graph Transformation Systems (PGTS) [14]. This model assumes nondeterministic choice of rule and match as in conventional graph transformation, but executes rules probabilistically. In PGTS, rules have single left-hand-sides but possibly several right-hand sides equipped with probabilities. This mixture of nondeterminism and probabilistic execution gives rise to Markov decision processes. There are clear similarities between our approach and PGTS: both operate in discrete steps and both can express nondeterminism and probabilistic behaviour. However, PGTS are strict in their allocation of behaviour; rule and match choice is nondeterministic and rule execution is probabilistic. In our approach, a programmer may specify that a rule set is executed in either manner. It seems possible to simulate (unnested) PGTS in our approach by applying a nondeterministic rule set that chooses a rule and its match followed by a probabilistic rule set which executes one of the right-hand sides of this rule. For example, the first loop in the $G(n, p)$ program in

[1] https://github.com/UoYCS-plasma/P-GP2

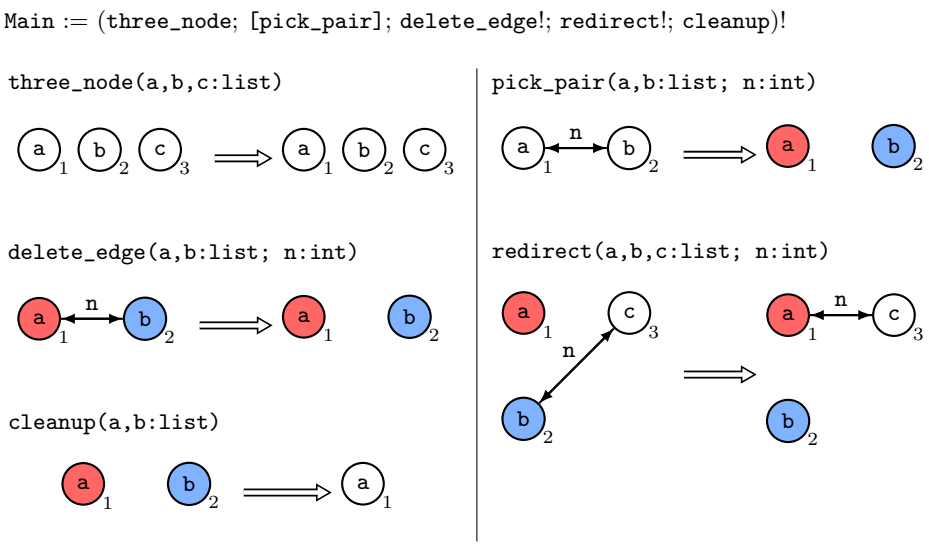


Fig. 4. The contraction procedure of Karger's algorithm implemented in P-GP 2

Figure 6 simulates a single PGTS rule; `pick_edge` nondeterministically chooses a match, and `[keep_edge, delete_edge]` probabilistically executes some right-hand side on the chosen match. Figure 3 visualises this single PGTS rule.

4 Application to Randomised Algorithms

4.1 Karger's Minimum Cut Algorithm

Karger's contraction algorithm [12] is a randomised algorithm that attempts to find a minimum cut in a graph G, that is, the minimal set of edges to delete to produce two disconnected subgraphs of G. The contraction procedure repeatedly merges adjacent nodes at random until only two remain. As this algorithm is designed for multi-graphs (without loops or edge direction), we model an edge between two nodes as two directed edges, one in each direction. For visual simplicity, we draw this as a single edge with an arrow head on each end. We assume that input graphs are unmarked, contain only simulated directed edges, and are connected. We also assume that edges are labelled with unique integers, as this allows us to recover the cut from the returned solution.

Figure 4 shows a P-GP 2 implementation of this contraction procedure. This program repeatedly chooses an edge to contract at random using the `pick_pair` rule, which marks the surviving node `red` and the node that will be deleted `blue`. The nodes' common edges are deleted by `delete_edge` and all other edges connected to the `blue` node that will be deleted are redirected to connect to the `red` surviving node by `redirect`. In the final part of the loop, `cleanup` deletes the `blue` node and unmarks the `red` node. This sequence is applied as

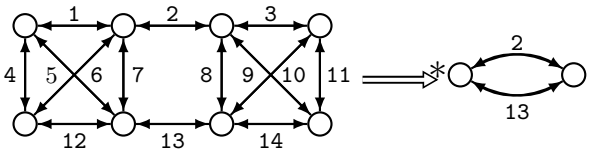


Fig. 5. Karger's contraction algorithm applied to a simple 8-node graph to produce a minimal 2-edge cut. The probability of producing this cut is at least $\frac{1}{28}$; our implementation generated this result after seven runs.

long as possible until the rule `three_node` is no longer applicable; this rule is an identity rule ensuring that a contraction will not be attempted when only 2 nodes remain. The final graph produced by this algorithm represents a cut, where the edges between the 2 surviving nodes are labelled with integers. The edges with corresponding integer labels in the input graph are removed to produce a cut.

Karger's analysis of this algorithm finds a lower bound for the probability of producing a minimum cut. Consider a minimum cut of c edges in a graph of n nodes and e edges. The minimum degree of the graph must be at least c, so $e \geq \frac{n.c}{2}$. If any of the edges of the minimum cut are contracted, that cut will not be produced. Therefore the probability of the cut being produced is the probability of not contracting any of its edges throughout the algorithm's execution. The probability of picking such an edge for contraction is:

$$\frac{c}{e} \leq \frac{c}{\frac{n.c}{2}} = \frac{2}{n} \tag{8}$$

Thus the probability p_n of never contracting any edge in c is:

$$p_n \geq \prod_{i=3}^{n} 1 - \frac{2}{i} = \frac{2}{n(n-1)} \tag{9}$$

For example, applying Karger's algorithm to the host graph G shown in Figure 5 can produce one possible minimum cut (cutting 2 edges), which happens with probability greater or equal to than $\frac{1}{28}$. By using rooted nodes (see [4]) it is possible to design a P-GP 2 program that executes this algorithm on a graph with edges E in $O(|E|^2)$ time, with `pick_pair` being the limiting rule taking linear time to find all possible matches, applied $|E| - 2$ times.

4.2 $G(n, p)$ model for Random Graphs

The $G(n, p)$ model [9] is a probability distribution over graphs of n vertices where each possible edge between vertices occurs with probability p. Here we describe an algorithm for sampling from this distribution for given parameters n and p. This model is designed for simple graphs and so we model an edge between two nodes, in a similar manner to that used in Karger's algorithm, as two directed edges, one in each direction.

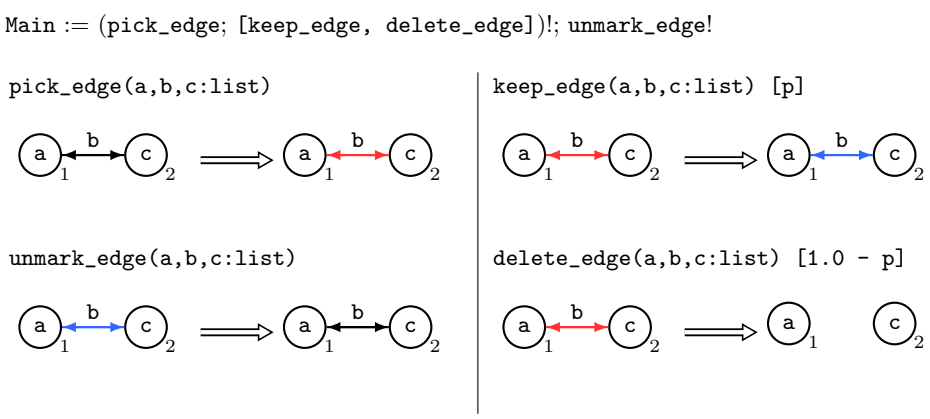


Fig. 6. P-GP 2 program for sampling from the $G(n,p)$ model for some probability `p`. The input is assumed to be a connected unmarked graph with n vertices.

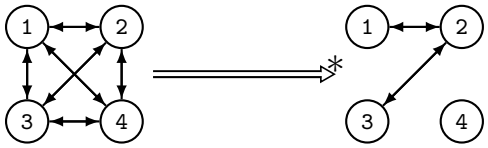


Fig. 7. The $G(n,p)$ program applied to a complete 4-node graph with $p = 0.4$. The probability of producing this result is 0.0207.

As we are concerned with a fixed number of vertices n, we assume an unmarked input graph with n vertices and for each pair of vertices v_1, v_2 exactly one edge with v_1 as its source and v_2 as its target – effectively a fully connected graph with two directed edges simulating a single undirected edge. Then $G(n,p)$ can be sampled by parameterising the GP 2 algorithm given in Figure 6 by p. In this algorithm, every undirected edge in the host graph is chosen nondeterministically by `pick_edge`, marking it `red`. Then this edge is either kept and marked `blue` by `keep_edge` with probability p or it is deleted by `delete_edge` with probability $1-p$. After all edges have either been deleted or marked `blue`, `unmark_edge` is used to remove the surviving edges' marks. By applying this algorithm, each possible edge is deleted with probability $1-p$ and hence occurs with probability p, sampling from the $G(n,p)$ model.

Sampling from the $G(n,p)$ model yields a uniform distribution over graphs of n nodes and M edges and each such graph occurs with probability:

$$p^M (1-p)^{\binom{n}{2}-M} \tag{10}$$

Figure 7 shows a possible result when applying this algorithm to a simple 4-node input with $p = 0.4$.

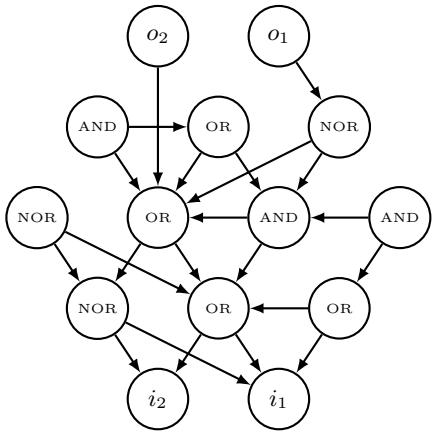


Fig. 8. An example EGGP Individual for a digital circuit problem. Outgoing edges of nodes represent the nodes that they use as inputs; for example $o_2 = (i_2 \downarrow i_1) \vee (i_2 \vee i_1)$.

5 Application to Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a class of meta-heuristic search and optimisation algorithms that are inspired by the principles of neo-Darwinian evolution. In its most general sense, an EA is an iterative process were a population of individual candidate solutions to a given problem are used to generate a new population using mutation and crossover operators. Individuals from the existing population are selected to reproduce according to a fitness function; a measure of how well they solve a given problem. Mutation operators make (typically small) changes to an individual solution, whereas crossover operators attempt to combine two individual solutions to generate a new solution that maintains some of the characteristics of both parents.

Graphs have been used extensively in EAs due to their inherent generalisation of various problems of interest; digital circuits, program syntax trees and neural networks are commonly studied examples. For example, there are extensions of Genetic Programming (a type of EA that evolves program syntax trees) that incorporate graph-like structures, such as Parallel Distributed GP [21] and MOIST [15]. Neural Evolution of Augmenting Topologies [23] evolves artificial neural networks treated as graph structures. Cartesian Genetic Programming (CGP) evolves strings of integers that encode acyclic graphs [17], and has been applied to various problems such as circuits and neural networks [24].

In [2], probabilistic graph programming (specifically, the P-GP 2 variant described in [1]) was proposed as a mechanism for specifying mutation operators. The approach, Evolving Graphs by Graph Programming (EGGP), was evaluated on a set of classic digital circuit benchmark problems and found to make statistically significant improvements in comparison to an existing implementation of CGP [25]. In the rest of this section, we explain the implementation of EGGP

using probabilistic graph programming and present new benchmark results for a set of odd parity digital circuit synthesis problems.

5.1 Evolving Graphs by Graph Programming

Individuals in EGGP represent computational networks with fixed sets of inputs and outputs. They have a fixed set of nodes, each either representing a function, an input or an output. Output and function nodes have input connections, which are given by their outgoing edges. These edges are labelled with integers to indicate the ordering of the inputs of that node; this is an necessary feature for asymmetric functions such as division. Figure 8 shows an example EGGP individual for digital circuit synthesis with 2 inputs, 2 outputs and function set {`AND`, `OR`, `NAND`, `NOR`}.

Definition 1. *[EGGP Individual] An EGGP Individual over function set F is a directed graph $I = \{V, E, s, t, l, a, V_i, V_o\}$ where V is a finite set of nodes and E is a finite set of edges. $s\colon E \to V$ is a function associating each edge with its source. $t\colon E \to V$ is a function associating each edge with its target. $V_i \subseteq V$ is a set of input nodes. Each node in V_i has no outgoing edges and is not associated with a function. $V_o \subseteq V$ is a set of output nodes. Each node in V_o has one outgoing edge, no incoming edges and is not associated with a function. $l\colon V \to F$ labels every "function node" that is not in $V_i \cup V_o$ with a function in F. $a\colon E \to \mathbb{Z}$ labels every edge with a positive integer.*

In our implementation of EGGP in P-GP 2, function associations are encoded by labelling a node with a string representation of its function. For example, in Figure 8, a node with function `AND` is labelled with the string "`AND`". Input and output nodes are encoded by labelling each of those nodes with a list of the form `a:b` where `b` is the string "`IN`" or "`OUT`" respectively, and `a` is a list uniquely identifying that output or input. The function a described in Definition 1 is used to order inputs, an important feature for avoiding ambiguity in asymmetric functions; as we deal with symmetric functions in this work these details are omitted from Figure 8.

In [2], two types of atomic mutations are used. The first, function mutation, relabels a function node with a different function, where the new function is chosen with uniform probability. The second mutation, edge mutation, redirects an edge so that a function node or output node uses a different input. In [2] and here, digital circuits are of interest, therefore we require mutation operators that preserve acyclicity.

The edge mutation used, given in Figure 9, selects an edge to mutate with uniform probability using `pick_edge`. The source of this edge is marked `blue` and its target is marked `red`. Then `mark_output` is applied as long as possible. This marks all nodes, for which there is a directed path to the source of the selected edge, `blue`. The remaining unmarked nodes have no such directed path to the source of the selected edge and so can be targeted by a new edge from that source without introducing a cycle. A new (unmarked) target is chosen with uniform

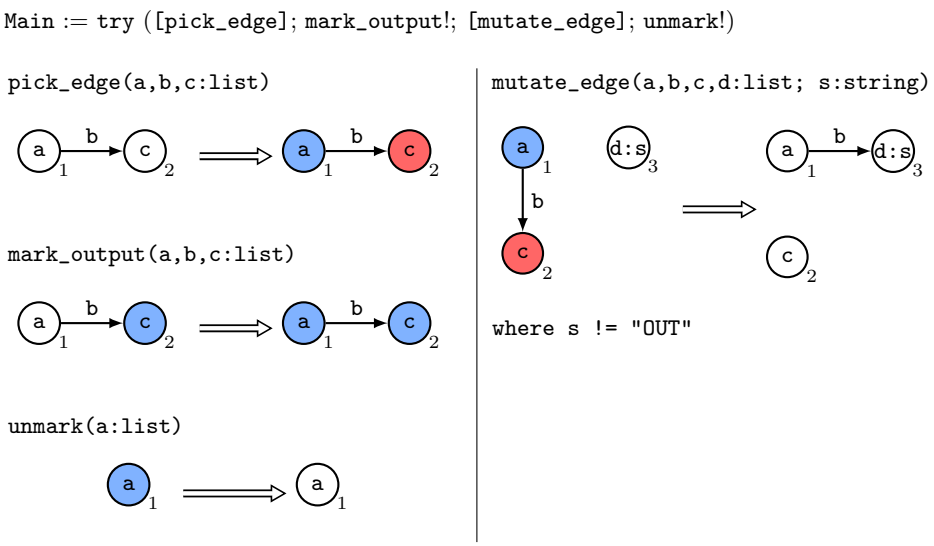


Fig. 9. A P-GP 2 program for mutating an individual's edge while preserving acyclicity.

probability using `mutate_edge`, executing the edge mutation. The condition of `mutate_edge` means that an output node cannot be targeted, thereby preserving the requirement that output nodes have no incoming edges. Finally, `unmark` unmarks all `blue` marked nodes, returning the now mutated EGGP individual to an unmarked state. The entire program is surrounded by a `try` statement to prevent errors in the case that an edge is chosen to mutate but there are no valid new targets.

In our implementation[2], edge and node mutations are written as P-GP 2 programs which are compiled to C code and integrated with raw C code performing the rest of the evolutionary algorithm. As a crossover operator has not yet been developed for EGGP, the $1 + \lambda$ evolutionary algorithm is used, where in each generation 1 individual survives and is used to generate λ new solutions. A more detailed explanation of EGGP and its parameters is available in [2].

5.2 Odd-parity Benchmark Problems

Here we compare EGGP against the commonly used graph-based evolutionary algorithm Cartesian Genetic Programming (CGP) for a new set of benchmark odd-parity circuit synthesis problems. CGP is a standard approach in the literature that uses a graph-based representation of solutions, but uses linear encodings that do not exploit graph transformations during mutation. The problems studied are given in Table 1 and complement the even-parity problems examined in [2]. We use the publicly available CGP library [25] to produce CGP results.

[2] https://github.com/UoYCS-plasma/EGGP

problem	inputs	outputs
5-bit odd parity (5-OP)	5	1
6-bit odd parity (6-OP)	6	1
7-bit odd parity (7-OP)	7	1
8-bit odd parity (8-OP)	8	1

Table 1. Digital circuit benchmark problems.

	EGGP			CGP				
Problem	ME	MAD	IQR	ME	MAD	IQR	p	A
5-OP	38,790	13728	29,490	96,372	41,555	91,647	10^{-18}	**0.86**
6-OP	68,032	22,672	52,868	502,335	274,132	600,291	10^{-31}	**0.97**
7-OP	158,852	69,477	142,267	1,722,377	934,945	2,058,077	10^{-33}	**0.99**
8-OP	315,810	128,922	280,527	7,617,310	4,221,075	9,830,470	10^{-34}	**0.99**

Table 2. Results from Digital Circuit benchmarks for CGP and EGGP. The p value is from the two-tailed Mann-Whitney U test. Where $p < 0.05$, the effect size from the Vargha-Delaney A test is shown; large effect sizes ($A > 0.71$) are shown in **bold**.

For both algorithms we use the following common parameters. We use the $1 + \lambda$ evolutionary algorithm with $\lambda = 4$. 100 fixed nodes are used for each individual. Fitness is defined as the number of incorrect bits in the entire truth table of a given individual. CGP is applied with a mutation rate of 0.04, considered to be appropriate in [17], whereas EGGP has been observed to perform better at lower rates and is applied with a mutation rate of 0.01.

For both algorithms, we execute 100 runs until a solution is found and measure the number of evaluations required to find a correct solution in each run. This value approximates the effort required by an algorithm to solve a given problem. We measure the following statistics; median evaluations (ME), median absolute deviation (MAD), and interquartile range in evaluations (IQR). The median absolute deviation is the median absolute difference in evaluations from the median evaluations statistic. We test for significant differences in the median of the two results using the non-parametric two-tailed Mann-Whitney U test [16] and measure the effect size of significant differences using the Vargha-Delaney A test [26].

Table 2 shows the results from running these experiments. These are consistent with the results in [2], in that EGGP and its mutation operators perform statistically significantly better (with large effect size) for digital circuit synthesis problems (on all of the problems studied here) than CGP under similar conditions. These results are not intended to represent a detailed study of the application of probabilistic graph programming to EAs. Instead they give a flavour of promising results published elsewhere, and represent a possible approach to empirical evaluation of P-GP 2 programs when formal approximations of behaviour are intractable.

6 Conclusion and Future Work

We have presented P-GP 2, a conservative extension to the graph programming language GP 2 which allows a programmer to specify probabilistic executions of rule sets, with weighted distributions over rules and uniform distributions over matches. This language has been used to implement Karger's randomised algorithm for finding a minimum cut of a graph with high probability. We have also implemented a P-GP 2 program for the $G(n, p)$ random graph model which generates n-node graphs in which edges between nodes exist with probability p. Finally, we have described the application of P-GP 2 to evolutionary algorithms in our approach EGGP. The program of this case study was evaluated empirically on common circuit benchmark problems and found to significantly outperform a publicly available implementation of Cartesian Genetic Programming.

There are a number of possible directions for future work. We would like to explore which algorithms from the areas of randomised graph algorithms and random graph generation can be described in P-GP 2. Obvious examples include randomised algorithms for checking graph connectedness [18], generating minimum spanning trees [13] and generating random graphs according to the model of [7]. Additionally, it would be interesting to investigate the efficiency of using incremental pattern matching [5] in the implementation as an alternative method for identifying all matches. Turning to evolutionary algorithms, there are various avenues to be explored, such as using graph programming to represent crossover operators, and combining domain knowledge and graph representations to possibly achieve even better performance.

References

1. T. Atkinson, D. Plump, and S. Stepney. Probabilistic graph programming. In *Pre-Proc. Graph Computation Models (GCM 2017)*, 2017.
2. T. Atkinson, D. Plump, and S. Stepney. Evolving graphs by graph programming. In *Proc. 21st European Conference on Genetic Programming (EuroGP 2018)*, Lecture Notes in Computer Science. Springer, 2018. To appear.
3. C. Bak. *GP 2: Efficient Implementation of a Graph Programming Language*. PhD thesis, Department of Computer Science, University of York, 2015.
4. C. Bak and D. Plump. Compiling graph programs to C. In *Proc. International Conference on Graph Transformation (ICGT 2016)*, volume 9761 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2016.
5. G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2008.
6. A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, second edition, 2015.
7. P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.
8. M. Fernández, H. Kirchner, and B. Pinaud. Strategic port graph rewriting: An interactive modelling and analysis framework. In *Proc. 3rd Workshop on Graph*

Inspection and Traversal Engineering (GRAPHITE 2014), volume 159 of *Electronic Proceedings in Theoretical Computer Science*, pages 15–29, 2014.
9. E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.
10. R. Heckel, G. Lajios, and S. Menge. Stochastic graph transformation systems. *Fundamenta Informaticae*, 74(1):63–84, 2006.
11. I. Hristakiev and D. Plump. Checking graph programs for confluence. In *Software Technologies: Applications and Foundations – STAF 2017 Collocated Workshops, Revised Selected Papers*, volume 10748 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2018.
12. D. R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1993)*, pages 21–30. Society for Industrial and Applied Mathematics, 1993.
13. D. R. Karger. Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. In *Proc. 34th Annual Symposium on Foundations of Computer Science (FOCS 1993)*, pages 84–93, 1993.
14. C. Krause and H. Giese. Probabilistic graph transformation systems. In *Proc. International Conference on Graph Transformation (ICGT 2012)*, volume 7562 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2012.
15. E. G. López and K. Rodríguez-Vázquez. Multiple interactive outputs in a single tree: An empirical investigation. In *Proc. EuroGP 2007*, volume 4445 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2007.
16. H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 1947.
17. J. F. Miller, editor. *Cartesian Genetic Programming*. Springer, 2011.
18. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
19. D. Plump. Reasoning about graph programs. In *Proc. Computing with Terms and Graphs (TERMGRAPH 2016)*, volume 225 of *Electronic Proceedings in Theoretical Computer Science*, pages 35–44, 2016.
20. D. Plump. From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, 88:154–173, 2017.
21. R. Poli. Parallel Distributed Genetic Programming. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 403–431. McGraw-Hill, 1999.
22. C. M. Poskitt and D. Plump. Verifying monadic second-order properties of graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2014)*, volume 8571 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2014.
23. K. O. Stanley and R. Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *Proc. Annual Conference on Genetic and Evolutionary Computation (GECCO 2002)*, pages 569–577. Morgan Kaufmann, 2002.
24. A. J. Turner and J. F. Miller. Cartesian Genetic Programming encoded artificial neural networks: A comparison using three benchmarks. In *Proc. GECCO 2013*, pages 1005–1012. ACM, 2013.
25. A. J. Turner and J. F. Miller. Introducing a cross platform open source Cartesian Genetic Programming library. *Genetic Programming and Evolvable Machines*, 16(1):83–91, 2015.
26. A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.