GENETIC PROGRAMMING
THEORY AND PRACTICE

GENETIC PROGRAMMING
THEORY AND PRACTICE

Edited by
RICK RIOLO
Center for the Study of Complex Systems
University of Michigan

BILL WORZEL
Genetics Squared, Inc.

Kluwer Academic Publishers
Boston/Dordrecht/London

Contents

Chapter 1

FLEXGP.PY: PROTOTYPING FLEXIBLY-SCALED, FLEXIBLY-FACTORED GENETIC PROGRAMMING FOR THE CLOUD

James McDermott[1], Kalyan Veeramachaneni[1] and Una-May O'Reilly[1]
[1]*Evolutionary Design and Optimization Group, CSAIL, MIT*

Abstract

Running genetic programming on the cloud presents researchers with great opportunities and challenges. We argue that standard island algorithms do not have the properties of elasticity and robustness required to run well on the cloud. We present a prototyped design for a decentralized, heterogeneous, robust, self-scaling, self-factoring, self-aggregating genetic programming algorithm. We investigate its properties using a software "sandbox".

Keywords: cloud, island model, FlexGP, distributed

1. Introduction

Computing on the cloud offers elasticity and massive concurrent compute resources. For genetic programming (GP) researchers this represents both a challenge and an opportunity. Parallel and distributed evolutionary algorithms (PDEAs) can take advantage of cloud resources through massive, scalable computation, with built-in protection against premature convergence. With this opportunity comes a requirement for well-designed, decentralized, robust and flexible algorithms. To avoid single points of failure and unbalanced compute or network loads, decentralized algorithms are needed. Given many nodes, failures are certain, so robust algorithms are needed. Given the opportunity and requirement to scale resource usage up or down during computation, self-scaling algorithms are needed. To encourage exploration of different areas of the search space, one possible model has heterogeneous islands, e.g. differing in their objectives, their training data, and their GP representation. Self-factoring algorithms are needed in this case. In order to re-integrate good results and ensure that the computation converges in the long term, self-aggregating algorithms are also needed.

The aim of the FlexGP.py project presented in this paper is to provide a test-bed for such algorithms in which ideas, logic and protocols can be implemented and tested without requiring the rather arduous build-deploy cycle typical of cloud computing. It is hoped that problems can be identified and necessary design decisions made.

The remainder of this paper is laid out as follows. Section 2 presents related work and leads to a description of the properties we require in our system. Section 3 describes our proposed new system, and Section 4 presents a theoretical and empirical investigation into the system's properties. Section 5 briefly concludes.

2. Related Work

The two most common models for PDEAs are the *fixed-topology coarse-grained island model* and the *fixed-topology fine-grained cellular grid model* (Cantú-Paz, 1998; Tomassini, 2005; Crainic and Toulouse, 2010). The *fixed-topology hybrid model*, also common, features coarse-grained islands with fine-grained internal structure (Cantú-Paz, 1998; Tomassini, 2005; Crainic and Toulouse, 2010). All of these are often visualized using networks of nodes (representing individuals or populations) and edges (representing migration). They can deliver performance benefits chiefly because a structured population can avoid premature convergence on just one area of the search space, in contrast to a *panmictic* population.

These models typically depend on some sort of centralized algorithm to impose the desired, static neighbourhood structure between nodes, and possibly to deal with node or communication failures. This means that the algorithms may suffer from a single point of failure and from possible high compute or communication loads on the "master" machine.

Dynamic decentralized models offer an interesting alternative. A single point of failure, and imbalanced compute or communication loads, are avoided because the algorithm is decentralized. This necessitates a random graph model: when nodes are expected to leave or join the computation often, it is not possible to maintain a neat, fixed-topology toroidal graph in a decentralized way, for example. However, desired properties such as connectivity and takeover time can emerge in a natural way, as will be described later. Random graph models can be implemented using peer-to-peer algorithms in which nodes exchange not only migrants' genomes but also meta-data describing (for example) known nodes available as migration destinations.

Such ideas motivated the design of the *Distributed Resource Machine* (DRM) system. Here, a large number of compute nodes form a volunteer network, the DRM. Applications of many different types can be hosted on DRM: *DREAM* is the combination of DRM with an evolutionary algorithm framework (Arenas et al., 2002). Although DRM is capable of a dynamic, peer-to-peer network model, DREAM uses a *fixed-topology coarse-grained island model*.

Later authors questioned DREAM's scalability and robustness. According to Laredo et al. (Laredo et al., 2010), "the island-based parallelization of DREAM was shown [...] to be insufficient for tackling large-scale decentralized scenarios." That is, Laredo et al. (Laredo et al., 2007) showed that DREAM's speedup is super-linear from 1 to 4 nodes, but sub-linear or even negative after that. Analysis showed that this failure to scale is the result of communication costs: migration between islands required a number of network hops per individual which scaled linearly with the number of nodes. The authors concluded that DREAM's network functions more as a broadcast network (approximating a fully-connected network) than a small-world peer-to-peer one.

In the *Evolvable Agent* line of research, a main aim is to address some of the weaknesses of the DRM system. It uses a *fine-grained random-graph cellular model* in which each node in the topology is associated with just one individual: crossover happens only between nodes which are connected by edges. Evolvable Agent was demonstrated to be scalable and to improve performance in a series of simulation experiments (Jiménez Laredo et al., 2011). One caveat must be mentioned: in these simulations the effects of the network were neglected, that is

traversal of an edge was assumed to be of negligible cost. This assumption is not realistic if the pair of nodes are on different physical machines, as they often will if the system is to take advantage of distributed processing. In such circumstances the fine-grained model, with hundreds or thousands of nodes per physical machine, will cause a very high communication cost.

A natural alternative is a system which (like Evolvable Agent) uses a *peer-to-peer dynamic random graph model*, but also (like DREAM) uses a *coarse-grained island model*. This is the model FlexGP.py will investigate. It avoids high communication costs because each physical machine runs a single node composed of hundreds or thousands of individuals, with infrequent migration between nodes. However it retains the decentralized, robust aspects of the random graph topology.

Using a coarse-grained dynamic random graph model addresses some of the key issues for cloud-scale computing mentioned above, in particular **robustness to failure**, using a **decentralized** algorithm, and having the ability to **flexibly scale resource usage** up or down. The implementation and behaviour of the algorithm in these respects will be described in the next section.

Several other properties are also of interest. To ensure that the parallel computation does not split into multiple independent runs cut off from each other, another necessary property is **connectivity**. A related emergent graph property is *takeover time* (Tomassini, 2005). It is desirable to keep the network well-connected so that good genetic material can propagate to all nodes, but avoid overloading the network through heavy communication, and avoid reducing the model (by over-use of communication) to the equivalent of a standard pan-mictic algorithm. Connectivity properties have previously been investigated in the context of DRM (Jelasity et al., 2002).

Decentralization in particular introduces a need for the algorithm to be in some sense **self-scaling**, or **elastic**: it must be capable of changing its behaviour in response to changes in the network structure or the compute resources. The most important aspect of this, for our purposes, is that the system must be capable of adding or removing compute nodes, while retaining sane network behaviour.

Previous research including that mentioned above rarely deals with **heterogeneous** GP algorithms. This is an opportunity for new research with cloud-scale PDEAs. In the current contribution the focus is on GP, and heterogeneity is taken to mean that islands may differ from each other in their *objectives*, their *training data*, their GP *language* (i.e. internal nodes), and their input or *explanatory variables*. This "factoring" has been somewhat explored in the field of GAs (Crainic and

Toulouse, 2010), but relatively under-explored by the GP community. Some exceptions include the coevolutionary work of Heywood and colleagues, e.g. (Lichodzijewski and Heywood, 2008) and the speciating island model of Gustafson and Burke (Gustafson and Burke, 2006).

We hypothesize that any advantages of heterogeneous algorithms are not evident in algorithms with small populations and few generations, because such algorithms are dominated by "edge effects" and do not allow strong meta-evolutionary effects to occur. Heterogeneous algorithms are therefore a good fit for PDEAs in which very large populations are split up in a natural way. Again, however, there is a requirement for careful algorithm design. Heterogeneous algorithms in a decentralized context must be capable of deciding just how each island should differ from the next, and how these differences should vary over time. The current contribution uses a very simple type of heterogeneity between islands (to be explained in the next section), but aims to provide a testbed for future investigations.

Decentralized algorithms have to include methods of aggregating final results. In some contexts a separate aggregation phase is appropriate. In the context of PDEAs it is also possible to make the algorithm **self-aggregating**. The multiple heterogeneous islands can be programmed to gradually become more and more homogeneous over time through communication both of genetic material and of the differences in their heterogeneous settings (i.e. the objectives, training data, etc. mentioned above). This avoids the need for a centralized aggregation phase at the end of the algorithm. Again, in the current contribution we present only a simple aggregation system, leaving full investigation and refinement for future work.

Our previous work on this project has investigated necessary software components and cloud infrastructure, such as the MapReduce algorithm and Amazon EC2 cloud computing resources, see (Fazenda et al., 2012; Sherry et al., 2011). In the next section we present the FlexGP.py prototype.

3. The FlexGP.py System

We begin with a re-statement of the system's goals.

- Decentralized: no master, no single point of failure.
- Robust: node or communication failure should not bring down the system.
- Connectivity and desirable values for diameter and/or takeover time.

- Flexible scaling: can increase or decrease number of nodes during computation.
- Lightweight: avoid network overload and keep tasks CPU-bound to ensure maximum use of compute resources.
- Heterogeneous: islands differ by objectives, by training data, by language, and by explanatory variables.
- Aggregation: islands should become more homogeneous over time.

FlexGP.py attempts to achieve a subset of the above goals **in a single-machine simulation** in a minimal and elegant way. It simulates each cloud node using one compute process and one listener thread. Communication between islands is via sockets. FlexGP.py is easy to read and understand, being composed of less than 1000 lines of Python. Fitness evaluation, by far the most CPU-heavy aspect of most evolutionary algorithms, is farmed out from Python to Numpy, which allows it to run as a compiled C loop. FlexGP.py performance is respectable, but still considerably slower than an implementation in a language like C or Java.

FlexGP.py is fundamentally a peer-to-peer algorithm in which each node performs computation and periodically communicates messages of various types to its neighbours. The messages are of three types, as illustrated in Fig. 1-1: individuals (i.e. migration between islands with semantics similar to a typical PDEA), node tables (maintaining the dynamic neighbourhood graph), and island descriptors (determining the characteristics of the heterogeneous islands). Each message type may be sent at a different frequency. The aim in sending messages is to spread information on new nodes and failures; to maintain a desired level of heterogeneity between islands, perhaps with a schedule for gradual homogenisation; and to avoid network overload. Node tables and island descriptors are described next.

Each node maintains its own node table (see example in Table 1-2). Each entry in the node table consists of an ID number, a binary status value, and a time-stamp associated with the status. The order of the node table is randomized. A node regards as its neighbours the first d nodes whose status is "good". When communication with a node succeeds, that node is marked "good" and its time-stamp is updated. When communication fails, the destination node is marked "bad" and its time-stamp is updated. Each node periodically sends out its node table to a neighbour. The receiving node merges this information into its own node table: unknown nodes are added to the list (outside the top d) and status values are updated, if their time-stamps are newer than those of the existing status values. The effect is that information on new nodes

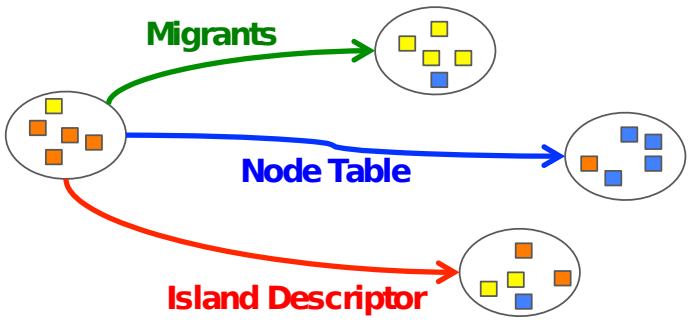
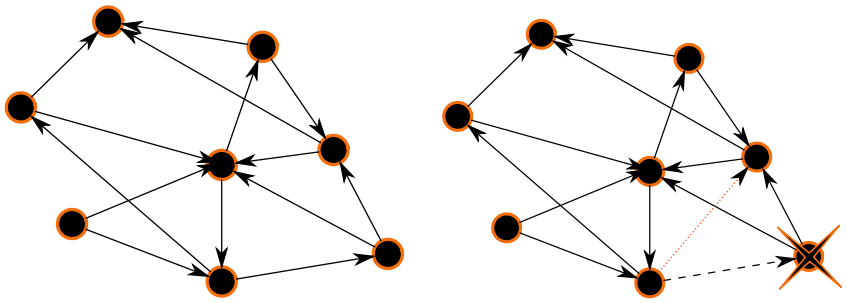


Figure 1-1. Message types: a node sends messages of three types as shown, at different rates.

ID	Status	Time-stamp
5	good	7.0
10	bad	9.0
3	good	7.0
7	good	10.0
0	bad	11.0
1	good	7.0
4	good	8.0
6	good	9.0
15	good	10.0

Figure 1-2. Sample node table: the top $d = 4$ "good" nodes are deemed neighbours and highlighted.



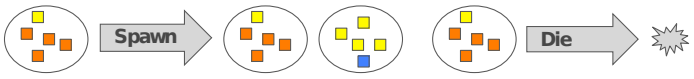
(a) Before Failure (b) After Failure

Figure 1-3. The digraph$_{n,d}$ model consists of a directed graph on n nodes each of out-degree $d = f(n)$. The in-degree varies. When one node fails to communicate with another, it marks the destination node bad (shown as a dashed arrow to the bad node), and now regards the next highest good node in its node table as a neighbour (shown as a dotted, highlighted arrow to the new neighbour), maintaining d.

and on node or communication failures propagates through the network in a decentralized way. When node or communication failures lead to neighbours being marked bad, they are implicitly replaced since the first d "good" nodes will now reach further down into the table. The table includes, in a sense, many "back-up" destinations, avoiding problems where a node is ready to send but has no viable destinations. Therefore the system is inherently robust to node failures. See Fig. 1-3. Because each node manages its own node table, the system is decentralized. Because the node order is randomized, there is no systematic overload of particular nodes. With an appropriate choice of d, connectivity of the entire graph can be achieved (see Sect. 4).

The node table also contains a *target node count*, i.e. the number of nodes which the computation *should* contain. This target count can passed-in at startup or set by an external process, mimicking the possibility of sending commands to a node in a true cloud system. The target is sent as part of the node table. In this way, the user or a sub-system with knowledge of changes to resource availability or pricing can send a message to any node and expect it to be propagated throughout the nodes over time.

Each node also maintains an island descriptor which parameterises the nodes' heterogeneous computation. A descriptor consists of four pieces of information. Two *island-level* pieces identify the island's characteristic subset of objectives and subset of training data. Two *genotype-level* pieces identify the island's characteristic subset of non-terminals and subset of explanatory variables. In all cases, a universal superset exists



(a) Spawn (b) Die

Figure 1-4. Nodes spawning and dying. At cloud scale, some existing nodes are expected to fail unintentialy during computation. Nodes can also *choose* to *spawn* or *die* as a method of scaling compute resource usage up or down. New nodes get randomly-created populations and island descriptors but inherit the parent's node table. Dying nodes send out many migrants—half their population.

and each island uses only a subset. This achieves heterogeneous computing. The island descriptor characterises the island's representation, taken to include its fitness function and encoding.

Each node gets a randomly-generated island descriptor at startup. Nodes send out their descriptors periodically in order to allow the gradual homogenisation of the computation. There is also the possibility of *meta-evolution*, i.e. evolution of island descriptors. This concept could work by making successful islands more likely to send out their descriptors. However this concept is not pursued in the current work.

In order to achieve flexible scaling of resources, the ability to start and stop nodes is needed. As illustrated in Figure 1-4, new nodes can be *spawned* when an existing node chooses to spawn a new node. Existing nodes can also fail for various reasons, and can also *choose* to *die*. In both the spawning and dying cases, the choice is random but biased according to the number of existing known nodes and the target number of nodes. That is, if the number of known nodes is less than the target node count, a node *may* spawn (with a certain probability) If the number of known nodes is greater than the target node count, a node *may* die (with a certain probability). Spawned nodes are created *de novo*, with randomly-created populations and island descriptors, though they inherit the node table of their parent. Dying nodes send out many migrants—half their population—in order to preserve whatever genetic learning they have achieved.

In the one-machine FlexGP.py simulation, nodes are allowed to spawn if they choose to do so, subject to there being space for new processes in the system. In a real-world scenario, a node wishing to spawn would need access to the cloud API in order to gain access to newly-available compute resources. This is the only area in which the one-machine simulation abstracts away a significant detail which would need to be addressed if implementing a true distributed version of the system.

These features confer several advantages, to be demonstrated in the next section:

- Startup can happen in a decentralized way, because one can start a single node, setting its target node count, and it and its descendant nodes will gradually create new nodes until the target is reached. decentralized startup is useful because in real-world scenarios, for example on Amazon EC2, starting up hundreds of nodes at once is a time-consuming and fragile task.
- The entire computation can be scaled up or down (i.e. decentralized expansion or shrinking). This is again useful in real-world scenarios, where for example compute prices vary over time. The ability to use the "target node count" to gradually and gracefully expand or shrink the number of nodes in the computation is suited for this scenario.
- Because there is no single master node, no node or communication link is overloaded with systematically high computation or communication overhead.

Migration is the key mechanism in any island system. In FlexGP.py, migration follows the "neighbour" model described above. Each node has d neighbours. At each generation, with probability p_m, it sends a message containing its best n_m individuals to either one neighbour, randomly-drawn from the d, or to all d. In relation to previous island-model work (Tomassini, 2005) this is most similar to a random topology model.

In the context of heterogeneous islands, one extra migration feature is required. It is possible that newly-arrived migrants will have come from islands on which the population is evolving under different standards. They may not have access to the same variables or the same fitness cases. In such scenarios, the newly-arrived individuals may well have useful genetic material, but may not have fitness values sufficient to compete with the existing population. Therefore we implement a form of migrant protection. With this parameter switched on, every migrant is assured of one crossover event in the generation when it arrives. Afterward, the migrants are added to the population for normal selection.

In addition to implementing inter-island communication, migration serves as a method of *aggregation*. Over time, heterogeneous islands become more and more homogeneous by mixing of the populations. The best individuals are aggregated by migration.

Islands also become more homogeneous through the exchange of island descriptors. The receiving island merges the island descriptor into its own, potentially gaining the ability to use new variables and test cases.

These policies allow the researcher to be confident, during a long-running evolution, that sampling good individuals from any island will

give reasonably representative results. These policies have the benefits of being decentralized, with no single point of failure, and requiring no separate finalisation phase of the algorithm.

4. Results

In this section we investigate the system's behaviour through the following studies: performance of the dynamic island topology compared with a more standard static island model; performance of differing numbers of islands; performance of heterogeneous versus homogeneous islands; performance with differing numbers of migrants, and with and without migrant protection; connectivity of the island topology; robustness to failures; and the dynamics of self-scaling elastic compute.

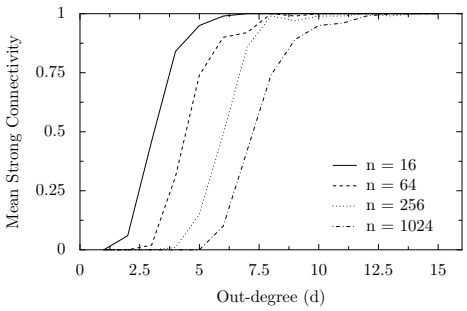
When performance is reported, it is generally the system's best fitness across islands, averaged over 30 runs. The problem is a symbolic regression of the function

$$f(x) = \sum_{i}^{4} 1/(1 + x_i^{-4})$$

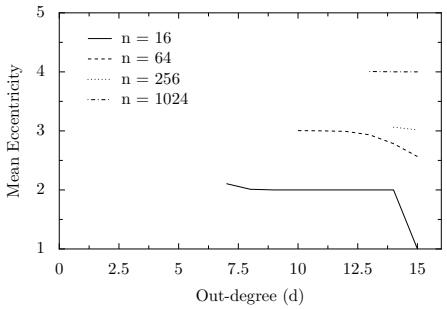
a four-dimensional version of the test function used by Pagie and Hogeweg (Pagie and Hogeweg, 1997). It has been reported to be a difficult problem in previous work (Harper, 2010). The GP system itself is not a focus of the current work: it is a linear-GP reverse polish notation system, with two-point crossover (crossover rate 1.0) and per-individual mutation (mutation rate 0.1). Individuals have an initial size of 8 genes. The non-terminal genes are {+, -, *, /}. The divide operator is protected: a zero-division exception results in the individual getting a poor fitness. The terminal genes are the variables x_i and the constants 0.1 and 0.5.

Migration is probabilistic, happening every generation with probability 0.2. The number of migrants is 1 except where stated. The spawning and dying probabilities are 0 for most experiments: the exception is the set of experiments on elastic computation (Sect. 4.0). The node table is emitted every generation with probability 0.333.

In some of the experiments to be reported, the result of interest is network connectivity: best-fitness performance is not of interest and is not reported. In these cases we reduce the algorithm's CPU usage by setting fitness to a random-number generator, setting crossover to have null behaviour, and setting a trivial population size. These settings do not affect migration behaviour insofar as it affects connectivity.



(a) Probability of strong connectivity



(b) Mean eccentricity

Figure 1-5. Graph-theoretic properties of the random digraph$_{n,d}$ model.

Connectivity

A key requirement is connectivity of the graph. A well-known result in graph theory states that a broad class of a graph-theoretic properties which depend on the mean degree of nodes will be almost never fulfilled for random graphs of low mean node degree, and almost always fulfilled for high degree, with a sharp threshold in between (Bollobás, 2001). The best-known example is connectivity of an undirected random graph: in order to "almost" guarantee connectivity, it is sufficient to choose the degree d to exceed a threshold which depends only on n. The authors are unaware of a corresponding theorem for strong connectivity (i.e. connectivity taking edge direction into account) of random directed graphs of fixed out-degree d, but a simple numerical simulation suggests the same result: for $n = 1024$, for example, a fixed out-degree of $d = 12$ is enough to almost guarantee connectivity. See Fig. 1-5.

However, the results in Fig. 1-5 are concerned with the theoretical connectivity of the network topology: a path of any length between two nodes is sufficient to regard them as connected. In practice, when a computation runs for a limited number of generations, network behaviour

may be quite different. Therefore we investigate the effect of the parameter d for several types of topology, several different network sizes (i.e. numbers of islands), when running simulations over 100 or 400 generations. Fig. 1-6 again shows that the d parameter is not too important, so long as the computation is relatively long-running: any value of $d > 3$ is sufficient for high connectivity. However for shorter runs, no plausible value of d will achieve high connectivity (the results suggest an upward trend: extrapolating suggests that high connectivity will not be achieved until $d > 50$, requiring an unrealistic amount of migration and network usage). Meanwhile sending to all neighbours, versus to just one chosen randomly from neighbours, has little effect. Note that experiments are not required to compare the connectivity of static topologies in this regard. An intact topology of 4x4 or 8x8 nodes will achieve a perfect "infection rate"; the broken static 4x4 example topology shown later in Fig. 1-10 will achieve an "infection rate" of close to 0.39 (calculation not shown).

Numbers of Islands

Our first test investigates the effect of varying the number of islands. We assume a fixed budget for the total population size of 4000 individuals. This is allocated in three different ways: to a single island, to 4 islands of 1000 individuals, or to 16 islands of 250 individuals. The performance of these three setups is shown in Fig. 1-7. There are no significant differences among these setups. However the 1x4000 setup's performance is beginning to level off after 100 generations, while the 16x250 setup appears to be still increasing.

Aspects of Migration

Next we study two aspects of migration: the number of migrants (1, 5, or 25 migrants per batch) and whether migrant protection is on or off. When using 1 migrant with and without migrant protection, as in Fig. 1-8 (a) and (b), there is no significant difference; 1 migrant is slightly, but significantly, better than 5 (Fig. 1-8 (c)), but not significantly better than 25 (Fig. 1-8 (d)). There is no clear overall trend: the system's behaviour is rather robust to a wide variation in the numbers of migrants and their treatment at the destination island.

Static versus Dynamic Topology

Next we compare the performance of static and dynamic topologies. In each case we use 16 islands: a 4x4 one-way toroidal grid for the static topology, and 16 islands with out-degree 4, for the dynamic topol-

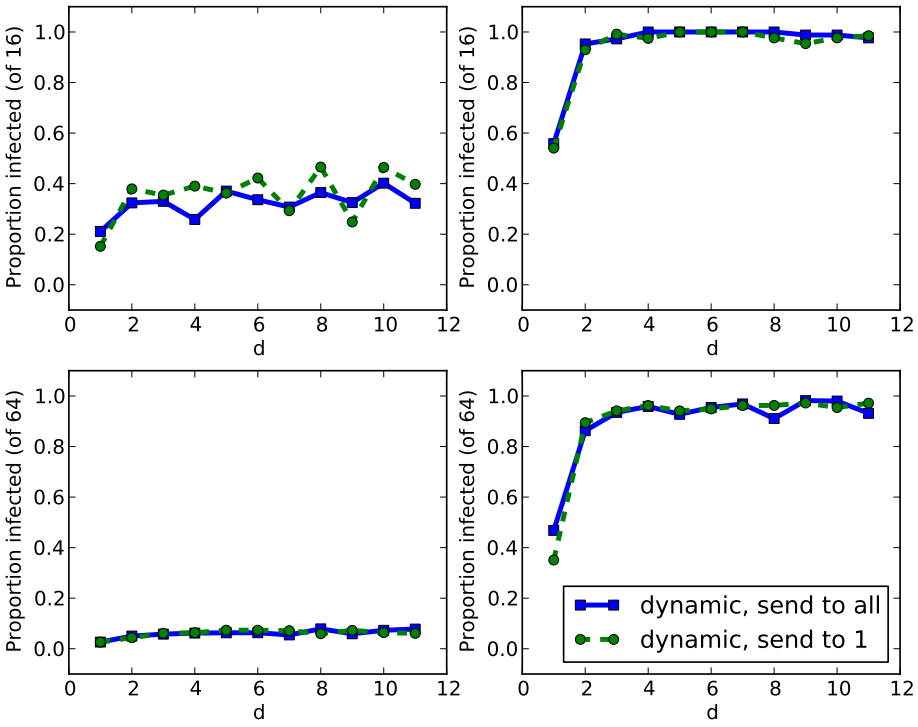


Figure 1-6. Inter-island connectivity: the effect of the d parameter and the length of the run. Above, 16-island runs of 100 and 400 generations; below, 64-island runs of 100 and 400 generations. The vertical axis shows the "proportion infected", i.e. the number of island-to-island pairs between which migration has taken place (directly or indirectly). 100 generations are insufficient to ensure high connectivity in this sense, but 400 are sufficient. From first principles, an intact static topology will achieve connectivity of 1.0, while a broken static topology will achieve poor connectivity. Sending to all neighbours, versus to just one chosen randomly from neighbours, has little effect.

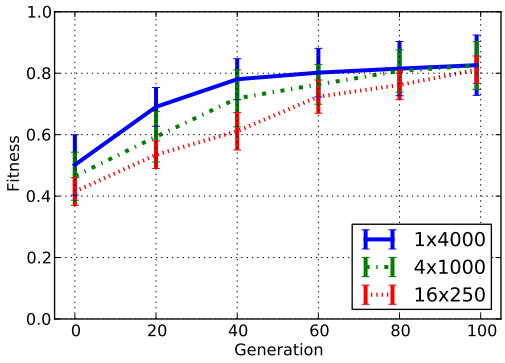


Figure 1-7. Best-fitness results with dynamic topology.

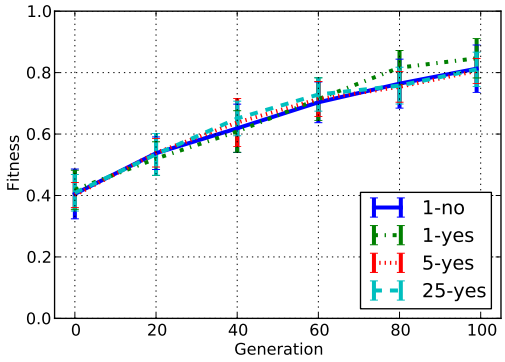


Figure 1-8. Best-fitness results with a 16-island dynamic topology, comparing different numbers of migrants. "No" and "Yes" indicate whether migration protection is turned on.

ogy. Results are shown in Fig. 1-9. The static topology marginally outperforms the dynamic after 100 generations (two-sided t-test, $p < 0.05$). However the effect is small.

Robustness to Link Failure

A key motivation for the dynamic island model is that node and link failures are expected to be inevitable at large scale. When a node or link fails in a static topology, an explicit repair is required. However our design specifications have ruled out the use of a master node, which would represent a single point of failure. In most previous work, the impact of node and link failures in static topologies has not been considered. Here, we present a rather extreme case of link failure in a 4x4 toroidal

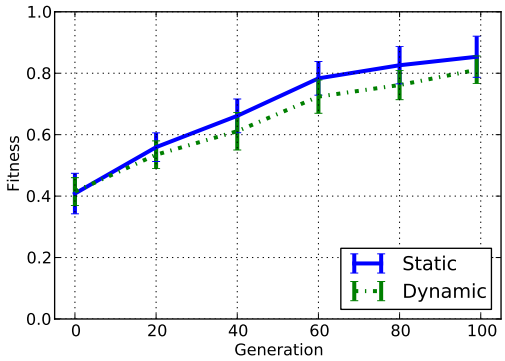


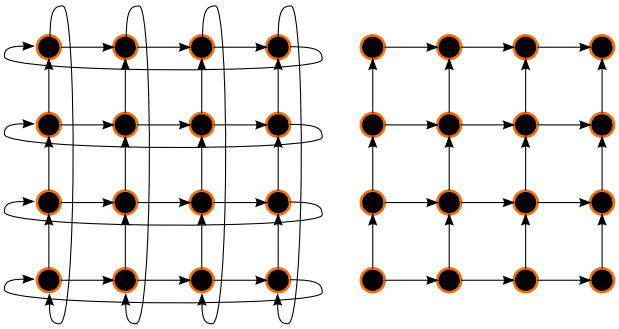
Figure 1-9. Best-fitness results with static and dynamic topologies. 16 islands each of population 250.

grid topology. The topology is presented intact in Fig. 1-10 (a). Note that the arrows represent one-way links: each island sends to "north" and "east" neighbours only, and receives from "south" and "west". Although this one-way scenario is not the most common for toroidal grids in previous work, it is common in one-dimensional "ring" topologies. Here, we choose this one-way scenario in order to demonstrate more clearly the impact of extreme link failure, as in Fig. 1-10 (b). Here all the "wraparound" or toroidal links have failed. Nodes to the "west" and "south" of the grid are now receiving fewer inward migrants than before (and none at all in the case of the "south-westernmost" node). As in most previous work, islands do *not* block when they fail to receive inward migrants: they simply continue computing. However performance is affected, as shown in Fig. 1-11. The intact topology performs significantly better than the broken one (two-sided t-test of the best fitness per run (across islands) achieved after 100 generations: $p < 0.01$).

In contrast, a key feature of the dynamic island topology is that node failures and link failures, which are expected to be inevitable at large scale, do not greatly damage the topology. It is not vulnerable to link failures, as argued in Section 3, Fig. 1-3.

Heterogeneous Islands

The system's ability to use heterogeneous islands is intended to be useful in problems characterised by many input variables, large amounts of training data, or other complexities. In this prototype system, with its 4-dimensional test problem, it is difficult to test whether the heterogeneity feature is useful. Nevertheless we present preliminary re-



(a) Intact (b) Broken

Figure 1-10. A static grid topology, both intact and with broken links.

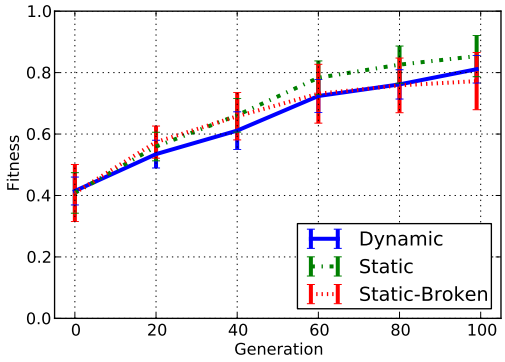


Figure 1-11. Best-fitness results with the static grid topology (dimensions 4x4). The intact version performs better.

sults. In this experiment, the homogeneous setup used all four input variables (x_0, x_1, x_2, x_3). In the heterogeneous setup, even-numbered islands *started* the run with just (x_0, x_1) while odd-numbered islands *started* with just (x_2, x_3). Migration led to the gradual homogenisation of the islands. Fig. 1-12 shows that there is no significant difference between these heterogeneous and homogenous setups.

Elastic Computation

The system is capable of automatically adding islands to a computation which appears to have stagnated. As a pilot experiment, we have tested this feature by starting a single island and allowing it to scale up. Each island spawns a new island whenever the best fitness on that island has not changed in 10 generations. After an island is spawned, this

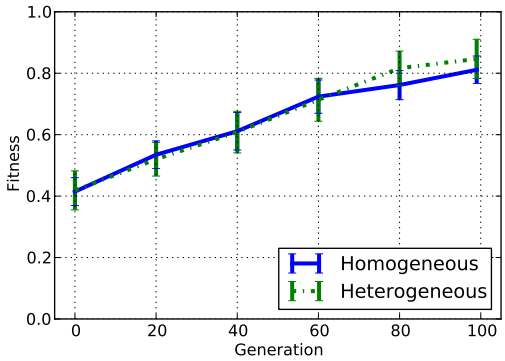


Figure 1-12. Best-fitness results with the 16-island dynamic topology, comparing homogeneous and heterogeneous islands.

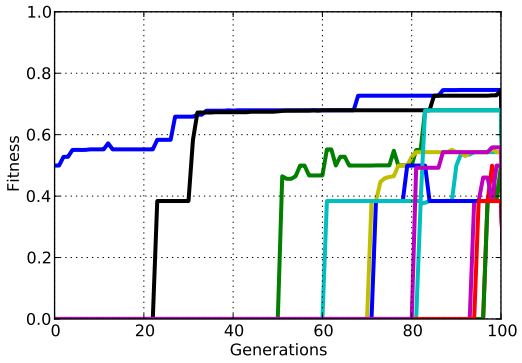
stagnation counter is reset. Also, each island is limited to spawn at most 2 islands during the run. With these parameters, different runs result in a variety of behaviours. Some runs never stagnate, and thus finish with just the original island. In some runs a few new islands are spawned. In others, up to about 15 or 20 islands can be spawned during the 100 generations. In the ideal scenario, some of the new islands introduce diversity and their outward migrants then contributes to bringing other islands out of stagnation. A typical run in which this effect appears successful is visualized in Fig. 1-13.

We also briefly demonstrate elasticity in the opposite direction, that is the ability of the computation to automatically scale-down the resources it uses. In this case, the run begins with 10 islands, and after 20 generations a signal is received setting a new target node count of 2 nodes. This causes the computation to begin scaling down. The process is visualized in Fig. 1-13 (b).

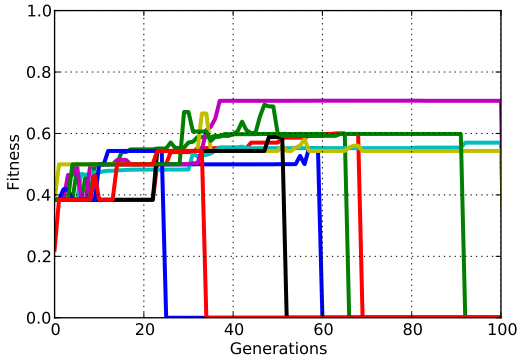
In both auto-scale-up and auto-scale-down cases, however, further tests would be needed to demonstrate that the elastic compute feature is reliable.

5. Conclusions

In this paper we have motivated, presented and investigated FlexGP.py, a novel prototype system for decentralized, heterogeneous, robust, self-scaling, self-factoring, self-aggregating GP on the cloud. Results show that performance with the novel features is comparable to standard island-based GP, with a noticeable advantage in terms of robustness. It is not sensitive to changing parameters such as numbers of migrants



(a) Automatic scale-up



(b) Automatic scale-down

Figure 1-13. Visualisations of elastic computation. In (a), auto scale-up. The run begins with a single island. New islands are spawned in response to stagnation and are indicated by fitness values ascending from 0. They seem to help move the computation out of its local optimum. In (b), auto scale-down. The run begins with 10 islands. After 20 generations a signal is received indicating that computation should scale down. Dying nodes are indicated by fitness values descending to 0.

or topology issues. Our next step is to implement a cloud-based version of the system for real deployment.

Acknowledgment

We would like to GE Global Research for the generous funding of this work. Dr. McDermott acknowledges the support of the Irish Research Council for Science, Engineering and Technology and a Marie Curie Fellowship.

References

Arenas, M., Collet, P., Eiben, A., Jelasity, M., Merelo, J., Paechter, B., Preuß, M., and Schoenauer, M. (2002). A framework for distributed evolutionary algorithms. In *Parallel Problem Solving from Nature VII*, pages 665–675. Springer.

Bollobás, Béla (2001). *Random Graphs*. Cambridge University Press, 2 edition.

Cantú-Paz, Erick (1998). A survey of parallel genetic algorithms. *Calculateurs Paralleles*, 10(2).

Crainic, Teodor Gabriel and Toulouse, Michel (2010). Parallel metaheuristics. In *Handbook of Metaheuristics*, pages 497–541. Springer.

Fazenda, Pedro, McDermott, James, and O'Reilly, Una-May (2012). A library to run evolutionary algorithms in the cloud using MapReduce. In Di Chio, Cecilia et al., editors, *Applications of Evolutionary Computing, EvoApplications2012: EvoCOMNET, EvoCOMPLEX, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoNUM, EvoPAR, EvoRISK, EvoSTIM, EvoSTOC*, volume 7248 of *LNCS*, pages 416–425, Malaga, Spain. Springer Verlag.

Gustafson, Steven and Burke, Edmund K. (2006). The speciating island model: An alternative parallel evolutionary algorithm. *Journal of Parallel and Distributed Computing*, 66(8):1025–1036. Parallel Bioinspired Algorithms.

Harper, R. (2010). Spatial co-evolution in age layered planes (SCALP). In *CEC*. IEEE.

Jelasity, M., Preuß, M., Van Steen, M., and Paechter, B. (2002). Maintaining connectivity in a scalable and robust distributed environment. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 389–394. IEEE.

Jiménez Laredo, J., Lombraña González, D., Fernández de Vega, F., García Arenas, M., and Merelo Guervós, J. (2011). A peer-to-peer approach to genetic programming. In *EuroGP*, pages 108–117. Springer.

Laredo, J., Castillo, P., Paechter, B., Mora, A., Alfaro-Cid, E., Esparcia-Alcázar, A., and Merelo, J. (2007). Empirical validation of a gossiping communication mechanism for parallel EAs. In *Applications of Evolutionary Computing*, pages 129–136. Springer.

Laredo, J.L.J., Eiben, A.E., van Steen, M., and Merelo, J.J. (2010). EvAg: a scalable peer-to-peer evolutionary algorithm. *Genetic Programming and Evolvable Machines*, 11(2):227–246.

Lichodzijewski, P. and Heywood, M.I. (2008). Coevolutionary bid-based genetic programming for problem decomposition in classification. *Genetic Programming and Evolvable Machines*, 9(4):331–365.

Pagie, L. and Hogeweg, P. (1997). Evolutionary Consequences of Coevolving Targets. *Evolutionary Computation*, 5:401–418.

Sherry, Dylan, Veeramachaneni, Kalyan, McDermott, James, and O'Reilly, Una-May (2011). Flex-GP: Genetic programming on the cloud. In Di Chio, Cecilia et al., editors, *Applications of Evolutionary Computing, EvoApplications 2012: EvoCOMNET, EvoCOMPLEX, EvoFIN, EvoGAMES, EvoHOT, EvoIASP, EvoNUM, EvoPAR, EvoRISK, EvoSTIM, EvoSTOC*, volume 7248 of *LNCS*, pages 477–486, Malaga, Spain. Springer Verlag.

Tomassini, Marco (2005). *Spatially structured evolutionary algorithms.* Springer.