The Tree-String Problem: An Artificial Domain for Structure and Content Search

Steven Gustafson, Edmund K. Burke, and Natalio Krasnogor

School of Computer Science & IT, University of Nottingham
Jubilee Campus, Wollaton Rd. Nottingham, NG81BB, United Kingdom
{ smg | ekb | nxk }@cs.nott.ac.uk

Abstract. This paper introduces the Tree-String problem for genetic programming and related search and optimisation methods. To improve the understanding of optimisation and search methods, we aim to capture the complex dynamic created by the interdependencies of solution structure and content. Thus, we created an artificial domain that is amenable for analysis, yet representative of a wide-range of real-world applications. The Tree-String problem provides several benefits, including: the direct control of both structure and content objectives, the production of a rich and representative search space, the ability to create tunably difficult and random instances and the flexibility for specialisation.

1 Introduction

The behaviour of heuristic search algorithms in artificial intelligence domains (and other complex scenarios like operations research) is difficult to pin-down by conventional analytical methods. More specifically, as heuristic search algorithms are often stochastic in nature, they frequently result in incomplete searches, re-sample previously-visited states, oscillate between states and become trapped in local optimum. The fixed points of heuristics are usually hard to determine, making their run time average and worst case complexity difficult to assess [1]. Consequently, the design and application of heuristic methods for real-world problems typically proceeds by trial-and-error. However, artificial domains can provide insight into the search abilities of various algorithms, allowing future research to better apply these methods. Improving understanding of these methods is a step toward more general search and optimisation methods. This paper introduces a new artificial domain to improve the understanding of solution structure and content in heuristic methods.

Many real-world problems contain two key overlapping and often conflicting objectives: solutions must have a structure (e.g. topology), and the structure must be "filled" with the appropriate content. Examples of these objectives can be seen in planning, classification using decision trees and symbolic regression. Planning typically requires hierarchical solutions that encapsulate key low-level behaviours. An example in mobile robot planning is the issue of localisation [2]: robots have a difficult time maintaining a good approximation of their location

during moving and sensing. Localisation is a key low-level behaviour that needs to be carried out during a high-level strategy to allow for effective planning. The induction of decision trees for classification constructs solution structure simultaneously with data set features at each tree location. Higher level nodes typically encode more important features, while lower level nodes are used to make finer class distinctions. A classic example of such a method is Quinlan's ID3 algorithm [3]. Finally, structure and content issues can be found in the induction of mathematical expressions from data. In this case we look for both a functional form and its ideal operators and coefficients.

The above examples emphasise the interdependencies between solution structure and content. As these types of problems rarely contain features that can be optimised independent of the whole, artificial domains that allow direct manipulation of structure and content also need to ensure that the richness of the interdependencies is maintained. Genetic programming is the prototypical method that must deal with solution structure and content issues during its search for algorithmic solutions. Genetic programming handles structure and content issues implicitly in its search process. While genetic programming is shown to be competent in overcoming these conflicts in several real-world domains, it is not known whether the way it deals with structure and content is optimal or particularly good.

Genetic programming is an evolutionary algorithm that represents solutions as computer programs[4]. Artificial domains are frequently used as testbed problems: the most popular being the Artificial Ant problem, the family of Boolean problems (e.g. even-parity), and symbolic regression problems. These problems provide testbeds that represent problems such as planning, digital design, classification and mathematical regression. However, these domains typically lack random instance generation, the ability to easily create tunably difficult and large instances for studying asymptotic behaviour, and a clear distinction between the issues of solution structure and content conflicts.

Previous work has highlighted the desire of the community to address these issues. To improve solution generalisation, a random trail generator was created for the Artificial Ant problem to complement the existing use of the the Santa Fe trail [5]. While investigating *hardness* in genetic programming, tunably difficult instances of the Binomial-3 regression problem were found [6]. In this case, genetic programming was shown to have a harder time dealing with ill-suited constants. Also in the regression domain, tunably difficult random polynomials were created by considering the increased precision required by an approximation using the same search space (i.e. primitive constant ranges) [7]. This allowed the study of code growth under varied levels of difficulty for genetic programming. The aforementioned problems place emphasis on solution content, which is not independent from solution structure. The following problems direct attention back toward solution structure.

The Lid problem [8] focused only on the search for structure by using fixed arity primitives with no meaning themselves, other than for creating tree shapes. Instances in this problem, using a canonical representation and operator, were

tunably difficult and allowed a more direct examination of structure mechanisms and representation issues during search. The Max problem [9, 10] and the Royal Tree problem [11] were created to contain a singular goal state to allow analysis of how structure acquires appropriate content. These problems define an ideal solution that requires specific primitives at specific structure locations. Although these problems do have intermediate reward states, they can appear to be like needle-in-the-haystack problems that may not accurately reflect real-world problems and are somewhat limited in their flexibility for producing random instances that are tunable. A more complex Royal-Tree-like problem was defined in [12] that consisted of finding the correct proportions of subprograms using multi-arity nodes. This problem, along with the ORDER and MAJORITY problems [13, 14], investigated the relationship between content and structure, where the latter two were mainly concerned with the occurrence and location of primitives in solutions. Again, while these problems address particular issues in understanding difficulty with the canonical representation and operators, it is less clear as to how they are representative of real-world problems.

The Tree-String problem attempts to bridge the gap between simple and highly-specific problems to real-world problems by providing instance tunability, random instance generation, and a rich and complex search space, while still being amenable to analysis. This last point, amenability to analysis, is gained from the use of simple and clear methods and the ability to use small population sizes while maintaining complex behaviour. The paper proceeds by first defining the Tree-String problem. We then provide an empirical study to further demonstrate the tunability of instances and the complex search space attained using the Tree-String problem.

2 The Tree-String Problem Definition

The Tree-String problem was originally intended to be an artificial domain for genetic programming, but the domain also has possible applications in other areas of artificial intelligence. The goal of the Tree-String problem is to derive specific structure and content elements simultaneously. Instances are defined using a target solution consisting of a tree shape and content. Candidate solutions are then measured for their similarity to the target solution with respect to both tree shape and content objectives.

The Tree-String problem is defined as a tuple Π:

$$\Pi = (\Psi, \Xi, t, \alpha, \gamma, \delta),$$

where an instance is represented by a target solution t, composed of content elements from the set Ψ and has a tree shape defined by elements from the set Ξ. For example, binary tree structures which have internal nodes n and leaf nodes l would have $\Xi = \{n, l\}$. The functions α and γ map the instance t to two linear string representations, such that:

$$\text{For structure: } \alpha(t) \mapsto \Xi^*, \quad \text{and for content: } \gamma(t) \mapsto \Psi^*.$$

Finally, the function δ provides a measure of similarity that will represent fitness objectives, i.e. similarity, between two strings representing tree shape and the similarity between two strings representing tree content. That is, given a candidate solution t_c and target solution t_t:

$$\delta(\alpha(t_t), \alpha(t_c)) \mapsto i \in \aleph, \quad \text{and} \quad \delta(\gamma(t_t), \gamma(t_c)) \mapsto j \in \aleph,$$

where i and j represent the heuristic solution quality of t_c compared to instance t_t. The fitness function of the genetic programming system, or other heuristic search method, can then use these quality measures in a multiobjective selection method or linear combination, where the former is used in this paper. While the implementations of α, γ and δ can vary, to represent more closely the particulars of a given problem domain, in this paper we propose to fix them as follows:

- α : depth-first, in-order, tree traversal for solution content,
- γ : breadth-first tree traversal for solution structure,
- δ : longest common substring (LCS).

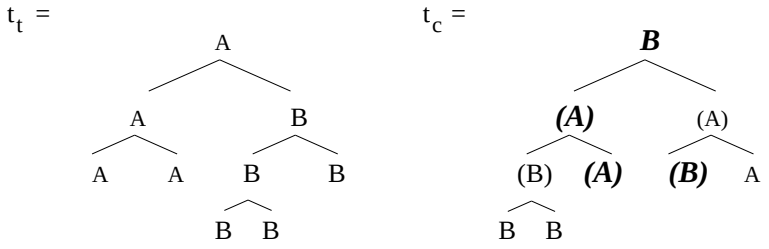
To further illustrate the Tree-String problem, let us consider an example using binary trees $\Xi = \{n, l\}$ with content using two symbols, $\Psi = \{A, B\}$. Note that the symbol A can either be a node or a leaf. Next, consider an instance t_t which has the following properties:

- the γ function makes a breadth-first tree traversal over the shape elements in t_t to produce $\gamma(t_t) = nnnllnlll$, and
- the α function makes a depth-first tree traversal over the content of t_t to produce $\alpha(t_t) = AAAABBBBB$.

Now, let us imagine that a search method generated a candidate solution t_c such that $\gamma(t_c) = nnnnllllll$ and $\alpha(t_c) = BBBAABBAA$. We then compute the measure of solution quality using δ (i.e. the longest common substring between the components of t_t and t_c), where the common substrings are underlined:

- $\delta(\alpha(t_t), \alpha(t_c))$ =LCS($\underline{nnnll}nlll, n\underline{nnnllll}$) = 5 and
- $\delta(\gamma(t_t), \gamma(t_c))$ =LCS($AA\underline{AABB}BBB, BBB\underline{AABB}AA$) = 4.

The elements of the candidate solution t_c that contributed to solution quality are shown below. The tree on the left shows the target tree instance t_t. The tree on the right shows the candidate solution t_c. The **structure** components in t_c that contribute toward fitness are denoted in parentheses (e.g. (A)), and the **content** components that contribute toward fitness are emphasized in bold italics (e.g. *A*):



The above example demonstrates the conflicting nature of structure and content objectives, where the portion of the solution that contributes to the structure objective is different from the part that contributes toward the content objective. This property is likely to make it difficult for transformation operators to effect either content or structure objectives alone, making the two features interdependent.

The choices of breadth-first and depth-first traversals for γ and α was purposefully done to exploit the hierarchical nature of solution structure and element juxtaposition of solution content, respectively. These functions also allow the search to focus on key *features* of target solutions. By features, we refer to more general properties (e.g. for structure: balanced, sparse or bushy trees). While an instance of the Tree-String problem would use a pre-selected structure and content, these do not necessarily define one unique goal state that would achieve maximal fitness. This is different from other domains like the Royal Tree or Max problems. However, the use of the longest common substring measure guarantees that strings are compared with their order preserved. Other measures like edit distance would provide the same value if two strings match every-other symbol or the same number of consecutive symbols. The longest common substring function complements the flexibility in the depth/breadth-first traversals with the more strict requirement of contiguous matching elements. It is our goal that these definitions allow for suitably complex behaviour representing real-world domains, but that is well-defined and amenable to analysis.

To further illustrate the Tree-String problem, we report preliminary work toward furthering the understanding of problem difficulty in genetic programming.

3 A Preliminary Study of Difficulty

In [15], the Tree-String problem was used to represent key properties of other common testbed domains (Artificial Ant, Parity, Regression) to study dissimilarity. A single instance of the problem using a binary tree shape and four content symbols was randomly produced. The tree shape was selected from those found to be more easily encountered by genetic programming [8]. The use of four symbols was an approximation to the typical size of function and primitive sets used in other testbeds. Random trials were carried out on this instance. A subsequent analysis over the three common testbed problems suggested that their specific behaviours were captured by the Tree-String problem. That is, the instance of the Tree-String problem represented the general behaviour of the other problems, see Chapter 7 of [15]. However, the full potential of the Tree-String problem was not used in that study, which is now being extended using a range of tunable instances. We report on that progress next.

3.1 Experimental Methodology

The genetic programming algorithm is generational with a population size of 50. Two-parent subtree crossover is used to transform existing solutions into new

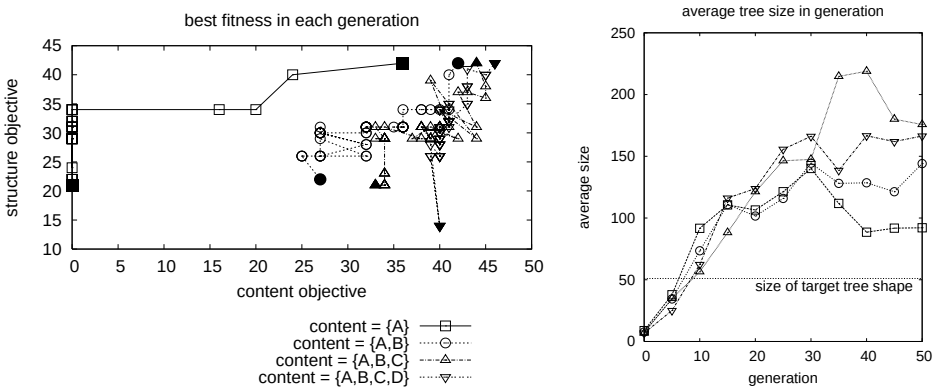


Fig. 1. The best fitness of four runs, using the same target tree shape and four different content strings, is plotted in the left graph. The first and last generation best fitness values are shown in solid symbols (■, ●, ▲, ▼), where the first generation of each run is located in the upper right of the plot. The average tree size of each population is shown in the right graph.

ones. Two parent crossover selects a subtree (where non-leaf nodes are selected 90% of the time) from each parent and swaps them. All children are valid provided they are within a predefined depth limit. To select parents for crossover, tournament selection with tournament size of 3 is used. The initial population is created by producing random trees using the Full and Grow methods equally between depths 2 and 4. A maximum depth for new trees is 17, and a stopping criterion of 50 generations is used. Ignoring the small population size, the system used here is a canonical system. A multiobjective pareto criterion is used for fitness evaluation with the objectives of structure and content, using the functions described in Section 2. A pareto optimal, or best fit solution is one which is better in at least one objective and no worse in the other compared to the rest of the population.

3.2 Single Structure, Multiple Content Behaviour

Initially, we look at the behaviour of four runs with the above system using one pre-selected tree shape (tree shape #2 in Figure 2 with depth 9 and 51 nodes) with four randomly created content strings (each with an increasing number of symbols, from 1 to 4). Figure 1 shows the evolution of the best fitness in each generation for each of the four runs. Here the fitness objectives report the size of the target strings (51 symbols) minus the longest common substring: a minimisation problem.

In Figure 1, the left graph shows that the more symbols in the content set Ψ, the more the search process optimises for tree shape. With one symbol in the content set, the search process can easily find a solution with the correct content (the size of the target tree in this case). However, as the number of

content symbols is increased to four, the search process makes very little progress improving the content objective, but focuses instead on the tree shape. The right graph of Figure 1 shows the evolution of the average solution size in each of the four runs. We can see that at generation 10, the easiest instance ($\Psi = A$), had the largest average tree size. However, the average size in this instance also reduced the most toward the end of the run when the structure objective is being improved. However, in the harder instances (3 and 4 symbols in Ψ) a larger average tree size is produced at the end of the run. The behaviour of more difficult instances producing larger solutions is similar to previous results for tunably difficult instance in genetic programming [6, 7].

This typical instance demonstrates that the search for both structure and content are conflicting in the Tree-String problem. While a population of size of only 50 individuals was used, the problem induces a complex search space. The remainder of the paper describes a much larger study of *hardness* in genetic programming, which is the subject of our future research. We show the generation of tunable instances and how genetic programming has a more difficult time improving both objectives when either one becomes more difficult.

3.3 Tree-String Instances

We create instances in the Tree-String problem with an increasing number of nodes and increasing content alphabet size. These two features, tree size and content size, are likely to lead to increased difficulty for the genetic programming algorithm. To avoid the pitfall of selecting tree shapes which are in themselves difficult for genetic programming, and duplicating aspects of [8], we will use the method of creating tree shapes from [8] but select shapes that are the most commonly visited (also seen in other empirical studies in [10]). We are then ignoring two other ways of tuning instances: fixing content and tree size and choosing more difficult shapes – or – for a particular tree shape and content, using different generation of target content (e.g. non-random ways).

To create the set of tree shapes on which to place random content, forming an instance, we generate a tree shape using the iterative tree growth method from [8], similar to the hill-climbing method in [16]. The method iteratively adds two child nodes to a probabilistically chosen leaf node, starting with the root. The probability of selecting leaf nodes can be altered to restrict trees to be less than a particular depth. We produce 500 random trees with depths between 5 and 15, and with 15 and 272 nodes. We first randomly pick a tree size from the latter range. A tree shape is then grown with a limit of depth 15. Figure 2 shows the distribution of the depth and size of the 500 random trees. We select a tree shape from depths 7, 9, 11, and 13 that are close to the mean size for that depth, ensuring that tree shape alone will not effect difficulty. These trees are shown in Figure 2 using a circular lattice visualisation[1] [8]. The root node lies at the very center, and each two child nodes lie at the intersection of subsequent lines.

[1] Code to produce this visualisation is available at http://www.cs.nott.ac.uk/∼ smg/

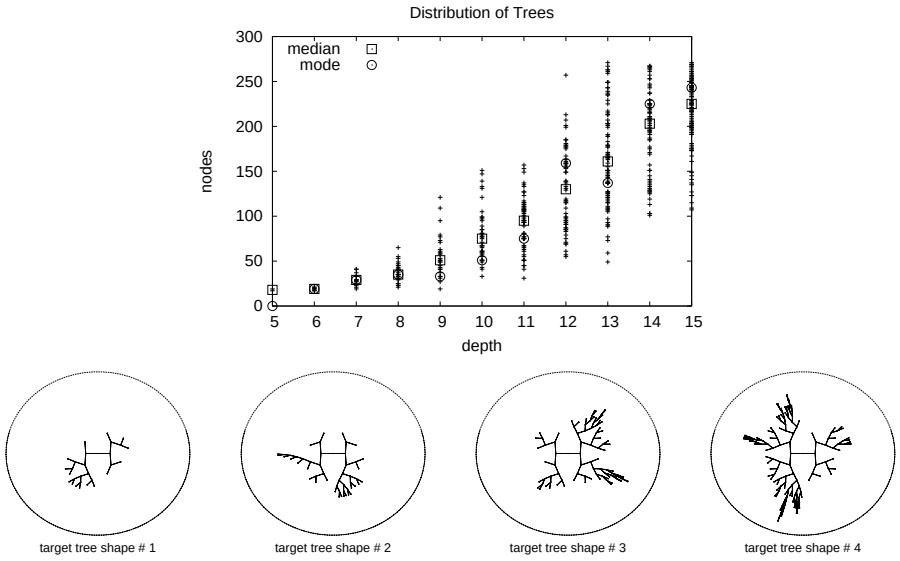


Fig. 2. Trees constructed by an iterative growth method, with bounds on maximum depth and randomly chosen size. The top graph shows the median and mode size for tree at each depth. A tree near to the median size was selected as a target tree from depths 7, 9, 11 and 13 in the empirical study. The bottom row of this Figure shows those four tree shapes.

The second step to define our instances is selecting which symbols from Ψ to use. We will create four random strings. The first using one symbol from Ψ, the second two, and so on. That is, one random string has $|\Psi| = 1$, another $|\Psi| = 2$, another $|\Psi| = 3$, and the last $|\Psi| = 4$. Each random string will be the same size as the tree shape under consideration - producing 4×4 instances. The genetic programming system will then use the same content set as used to create the current instance under consideration. That is, genetic programming will not need to address the additional potential problem of filtering out unnecessary elements from the content set.

The ability of genetic programming to search for tree content as well as tree shape can now be tested. By using tree shapes near the median of the distribution, we can assume with some confidence that they represent those shapes which genetic programming should be able to find more easily [8, 10]. However, by increasing the size of the instances, we hope to increase the difficulty of finding the correct tree shape. Also, by generating four random strings for each shape with an increasing content set size, we expect to control difficulty for finding correct content.

3.4 Experimental Study

The genetic programming method is run for 30 runs on each instance, creating 480 runs. We report the *improvement* of solution quality as the total size of the tree minus the longest common substring for structure and the total size minus the longest common substring for content. These values are then normalised by dividing them by the target tree size. We report the best (pareto optimal) candidate solution quality found in each population during the run. A similar study using a linear combination of the structure and content objectives (instead of a pareto criterion) was not seen to be significantly different.

We first examine the fitness distributions for the runs with different sizes of the set Ψ. The left column of Figure 3 shows the best fitness in each generation of each of the runs. From top to bottom in the left column of Figure 3 the alphabet Ψ size is increased. We can see how it is initially very easy to find good content (top). However, as Ψ size increases to 4 (bottom), the search gradually shifts toward improving tree shape. Thus, over all the random instances consisting of different target tree sizes, shapes and depths, increasing alphabet size increases the difficulty in the algorithm, causing the search to shift from improving content toward improving structure.

We now examine the effect of target tree size and fitness improvement for the same experiments, but now the instances are grouped by target tree size from smallest to largest. Again, we normalise both fitness objectives by the size of the target tree. The right column of Figure 3 shows the best fitness improvement in both objectives as target tree size increases (from top to bottom). Note that the amount of computation given to all experiments is equal, i.e. the same population size and generations. With a content alphabet of size 1 (as seen in the left column of this figure), genetic programming is still capable of finding trees large enough to match the target content string, seen with all tree sizes. However, the algorithm is unable to find similar improvements with regard to structure. That is, overall improvement is not in proportion to size when content complexity is greater than one symbol.

The empirical results demonstrate the creation of random instances for the Tree-String problem that are tunably difficult. Instance difficulty was achieved by increasing either the content complexity or the size of the size complexity of the instance. When one aspect of the instance (content or size) is *easy enough* (i.e. a content alphabet of size one or small tree shape), genetic programming can improve solution quality. Adding complexity to one objective, however, greatly effects the ability to improve that objective, and sometimes both objectives. More content complexity (more symbols in the content set) makes it harder to improve the content objective (left column of Figure 3), and larger tree shapes make it harder to improve either objective in proportion to the size (right column of Figure 3). A similar behaviour was also seen in the context of the multiobjective optimisation of size and quality [17], where it was easier to reduce tree size than improve quality. Runs converged toward improving the easier objective of size rather than equally improving size and quality simultaneously.

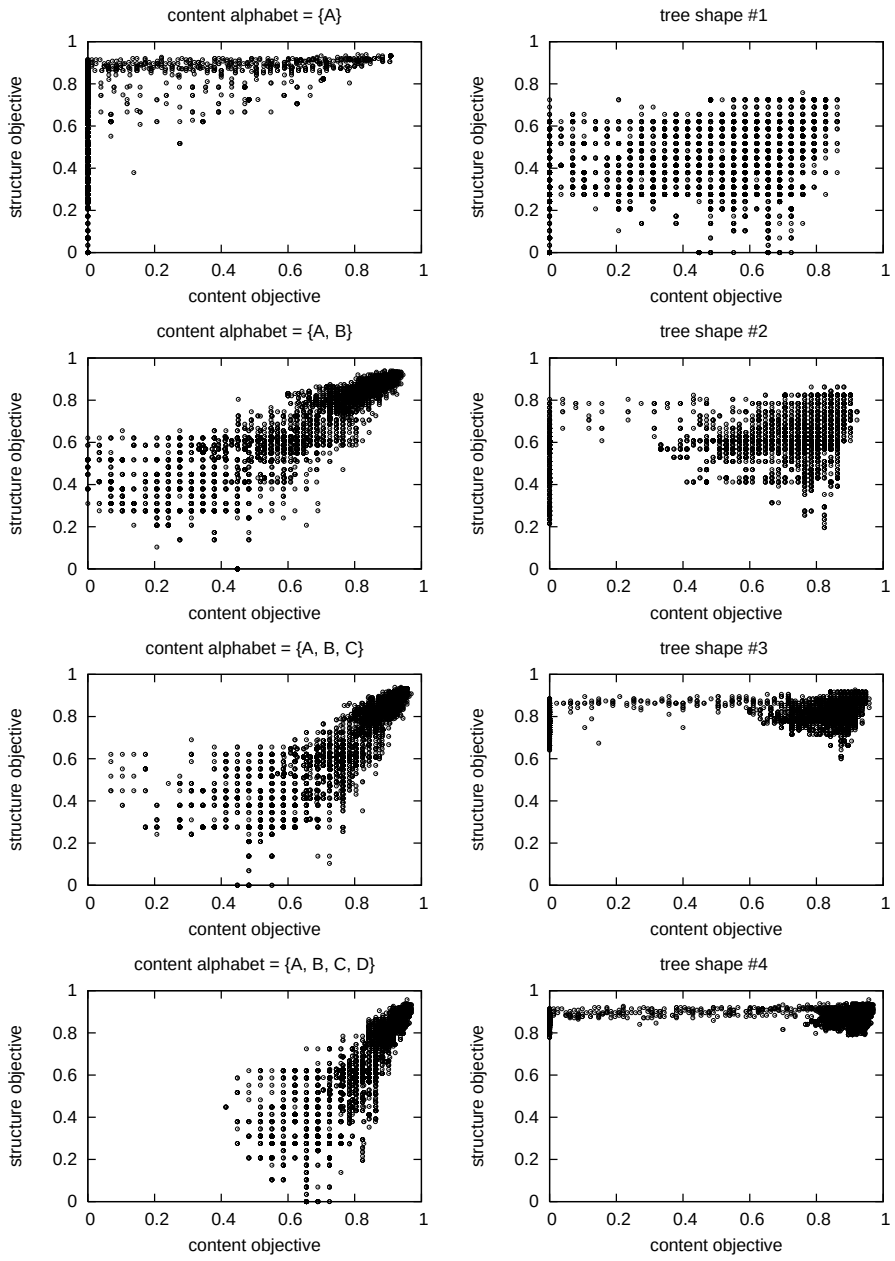


Fig. 3. The improvement of the content and structure objectives. Improvement is normalised by instance (target tree) size. The left column groups runs according to the size of the content set Ψ, increasing complexity from top to bottom. The right column groups runs according to the target tree shape, increasing size from top to bottom.

4 Conclusions

Analytical work for genetic programming has always encountered difficulty due to large population sizes, variable sized solutions and expensive fitness evaluation. The Tree-String problem offers the ability to simulate complex solution behaviour (content and structure dependencies) using variable length strings. That is, we do not need to compile new individuals or use precompiled elements for calculating fitness. All the functions used to convert Tree-String elements to strings representing structural and content features are generic (i.e. breadth-first, depth-first tree traversals and longest common substring). We are currently developing a very simple genetic programming system for the Tree-String problem that incorporates efficiency improvements described in [18] for the iTree data structure. It is our goal that the Tree-String problem allows for efficient research to take place on a complex problem, ultimately making significant contributions to the scientific community. We feel that for such a problem to be useful, it must be relevant to realistic genetic programming applications. It is for this reason that the Tree-String problem requires explicit focus on solution structure and content.

Capturing elements of real-world problems in artificial domains can be difficult. Artificial domains are intended to allow precise and efficient analytical work but often focus on singular aspects of solutions (structure or content). Additionally, testbed domains typically handle properties of solution structure and content implicitly, making it difficult to glean their effects. The Tree-String problem is proposed to make a stronger bridge between testbed functions and real-world applications. Toward this goal, we have seen the following properties of the Tree-String problem:

1. Control over both structure and content issues,
2. Clear and simple methods defining fitness and representation,
3. A complex and behaviour-rich search space,
4. The ability to create tunably difficult and random instances,
5. Substantial room for specialisation toward specific research goals.

Our future work is examining problem hardness in genetic programming. We are also examining variants of the Tree-String problem to carry-out efficient algorithmic analysis with respect to other problems. While we have hypothesised that the Tree-String problem is representative of other genetic programming domains, our current work is attempting to create mappings between these domains or between important domain features. **Acknowledgements:** This work was supported by EPSRC grant GR/S70197/01. SG thanks Jano van Hemert and the reviewers for their comments.

References

1. D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37(1), August 1988.

2. D. Fox, W. Burgard, H. Kruppa, and S. Thrun. A probabilistic approach to collaborative multi-robot localization. *Autonomous Robots*, 8(3):325–344, 2000.
3. J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
4. J.R. Koza et al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.
5. I. Kushchu. An evaluation of evolutionary generalisation in genetic programming. *Artificial Intelligence Review*, 18(1):3–14, 2002.
6. J.M. Daida et al. What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2(2):165–191, June 2001.
7. S. Gustafson, A. Ekárt, E.K. Burke, and G. Kendall. Problem difficulty and code growth in genetic programming. *Genetic Programming and Evolvable Hardware*, 5(3):271–290, 2004.
8. J.M. Daida, H. Li, R. Tang, and A.M. Hilss. What makes a problem GP-hard? validating a hypothesis of structural causes. In E. Cantú-Paz et al., editors, *Proceedings of the Genetic and Evolutionary Computation*, volume 2724 of *LNCS*, pages 1665–1677, Chicago, IL, USA, 12-16 July 2003. Springer-Verlag.
9. C. Gathercole and P. Ross. An adverse interaction between crossover and restricted tree depth in genetic programming. In J.R. Koza et al., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 291–296, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
10. W.B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, Berlin, 2002.
11. W.F. Punch, D. Zongker, and E.D. Goodman. The royal tree problem, a benchmark for single and multi-population genetic programming. In P.J. Angeline and K.E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. The MIT Press, Cambridge, MA, USA, 1996.
12. U.-M. O'Reilly. The impact of external dependency in genetic programming primitives. In *Proceedings of the IEEE World Congress on Computational Intelligence*, pages 306–311, Anchorage, AL, USA, 5-9 May 1998. IEEE Press.
13. U.-M. O'Reilly and D.E. Goldberg. How fitness structure affects subsolution acquisition in genetic programming. In J.R. Koza et al., editors, *Proceedings of the Third Annual Genetic Programming Conference*, pages 269–277, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.
14. D.E. Goldberg and U.-M. O'Reilly. Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. In W. Banzhaf et al., editors, *Genetic Programming, Proceedings of the First European Workshop*, volume 1391 of *LNCS*, pages 16–36, Paris, 1998. Springer-Verlag.
15. S. Gustafson. *An Analysis of Diversity in Genetic Programming*. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, Nottingham, England, February 2004.
16. A. Juels and M. Wattenberg. Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. Technical Report Technical Report CSD-94-834. Computers Science Department, University of California at Berkeley, USA, 1995.
17. E.D. de Jong and J.B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233, September 2003.
18. A. Ekárt and S. Gustafson. A data structure for improved GP analysis via efficient computation and visualisation of population measures. In M. Keijzer et al., editors, *Genetic Programming, Proceedings of the 6th European Conference*, volume 3003 of *LNCS*, pages 35–46, Coimbra, Portugal, April 2004. Springer-Verlag.