Evolutionary Discovery of Arbitrary Self-replicating Structures

Zhijian Pan and James Reggia

University of Maryland, Computer Science Dept. & UMIACS,
A. V. Williams Building, College Park, MD 20742, USA
{zpan, reggia}@cs.umd.edu

Abstract. In this paper we describe our recent use of genetic programming methods to automatically discover CA rule sets that produce self-replication of arbitrary given structures. Our initial results have produced larger, more rapidly replicating structures than past evolutionary models while requiring only a small fraction of the computational time needed in past similar studies. We conclude that genetic programming provides a very powerful tool for discovering novel CA models of self-replicating systems and possibly other complex systems.

1 Introduction

In the past studies of self-replicating CA structures, the rule sets governing cell state changes have generally been hand-crafted[1,2,3,4]. An alternate approach, inspired by the successful use of evolutionary computation methods to discover novel rule sets for other types of CA problems [5,6], used a genetic algorithm to evolve rules that would support self-replication [7]. This latter study showed that, given small but arbitrary initial configurations of non-quiescent cells ("seed structures") in a two-dimensional CA space, it is possible to automatically discover a set of rules that make the given structure replicate. However, some clear barriers clearly limited the effectiveness of this approach to discovering state-change rules for self-replication. First, to accommodate the use of a genetic algorithm, the rules governing state changes were linearly encoded, forming a large chromosome that led to enormous computational costs during the evolutionary process. In addition, as the size of the initial configuration increased, the yield (fraction of evolutionary runs that successfully discovered self-replication), decreased dramatically. As a result, it only proved possible to evolve rule sets for self-replicating structures having no more than 4 components, even with the use of a supercomputer, leading to some pessimism about the viability of evolutionary discovery of novel self-replicating structures.

In this paper, we revisit the issue of using evolutionary methods to discover new self-replicating structures and show that this earlier pessimism may be misplaced. We describe an innovative structure-encoding mechanism (S-tree) and a tree-like rule encoding mechanism (R-tree). As a result, genetic programming (rather than genetic algorithm) operators can be used. The resulting evolutionary system is qualitatively

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 404 – 411, 2005.
© Springer-Verlag Berlin Heidelberg 2005

more efficient and powerful than earlier methods, allowing the discovery of larger self-replicating structures with a standard computer rather than a supercomputer.

2 S-trees: General Structure-Encoding Representations

In the context of our work, an arbitrary structure can be viewed as a configuration of active cells in a CA space that satisfies two conditions. First, the active cells must be contiguous. Second, the configuration must be isolated from its environment. It follows that an arbitrary structure can be modeled as a connected, undirected graph, as we show in the following. The problem of structure encoding can then be converted to searching for a minimum spanning tree (MST) in order to most efficiently traverse the graph and encode its vertices (components).

Fig. 1 shows a simple structure in a 2D CA space, composed of 4 oriented components and satisfying the above two conditions. We convert the structure into a graph simply by adding an edge between each component and its 8 Moore neighbors, as shown in Fig. 2. The quiescent cells, shown empty in Fig. 1, are visualized with symbol "*" in Fig. 2. From this example we can see such a graph has the following properties: 1) it connects every component in the structure; 2) it also includes every quiescent cell immediately adjacent to the structure (which isolates the structure from its environment); and 3) no other cells are included in the graph. We name such a graph the *Moore graph*.

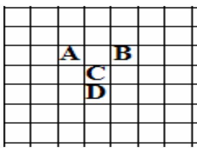

Fig. 1. The structure

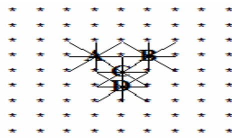

Fig. 2. The Moore graph

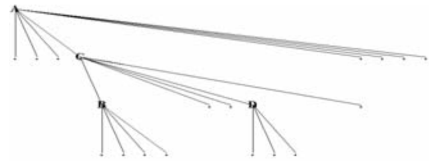

Fig. 3. The S-tree

Having the Moore graph for an arbitrary structure, we can then further convert the graph into a MST that we call the S-tree. Assigning a distance of 1 to every edge, we arbitrarily pick one component of the structure as the root, and perform a breadth-first-search of the Moore graph. The resultant tree is shown in Fig. 3. The essential idea is as follows. Starting from the root (A, in this example), explore all vertices of distance 1 (immediate Moore neighbors of the root itself); mark every vertex visited; then explore all vertices of distance 2; and so on, until all vertices are marked.

The S-tree therefore is essentially a sub-graph of the initial Moore graph. It has the following desirable properties as a structural encoding mechanism: 1) it is acyclic and unambiguous, since each node has a unique path to the root; 2) it is efficient, since each node appears on the tree precisely once, and takes the shortest path from the root; 3) it is universal, since it works for arbitrary Moore graphs and arbitrary CA spaces; 4) quiescent cells can only be leaf nodes; 5) active cells may have a maximum

of 8 child nodes, which can be another active cell or a quiescent cell (the root always has 8 child nodes); 6) it is based on MST algorithms, which have been well studied and run in near-linear time. Is the S-tree unique for a given structure? The MST algorithm only guarantees the vertexes of distance d to the root will be explored earlier than those of distance d+1. However, each Moore neighbor of a visited component lies the same distance from the root (such as B and D in Fig. 2), which may potentially be explored by the MST algorithm in any order and therefore generate different trees. This problem may be resolved by regulating the way each active cell explores its Moore neighbors, without loss of generality. For instance, let the exploration be always in a clock-wise order starting at a specific position (for instance, the left). As a result, we are guaranteed that a specific structure always yields the same S-tree. We say the resulting S-tree is in phase I, II, III, or IV, respectively, if the selected position is top, right, bottom, or left. The S-tree shown in Fig. 3 is in phase I. Fig. 4 shows the other phases. As clarified later, the concept of phase is important in encoding or detecting structures in rotated orientations.

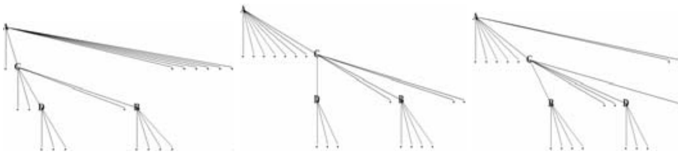

Fig. 4. S-tree at phase II, III, IV (from left to right)

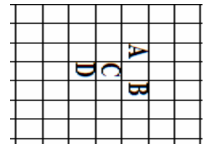

Fig. 5. Rotated Structure

We can easily convert an S-tree to a string encoding, simply by traversing the S-tree in a breadth-first order, and concatenating the state of each visited node to an initially empty string. The S-tree string encoding inherits the desirable properties of S-tree itself. It provides an unambiguous, efficient, and universal mechanism for representing an arbitrary structure, which enables an artificial evolution and rule learning system to be built and function without requiring the knowledge of any details of the involved structures a priori. Corresponding to the S-tree, there may be 4 different phases of S-tree encoding for a given structure. For each specific phase, the S-tree encoding is unique. Fig. 6 shows the S-tree encoding at each phase corresponding to the structure in Fig. 1.

S-tree encoding (Phase I) = " 1 0 0 0 9 0 0 0 0 5 0 013 0 0 0 0 0 0 0 0"
S-tree encoding (Phase II) = " 1 0 9 0 0 0 0 0 0 0 013 0 5 0 0 0 0 0 0 0"
S-tree encoding (Phase III)= " 1 0 0 0 0 0 0 0 913 0 5 0 0 0 0 0 0 0 0 0"
S-tree encoding (Phase IV)= " 1 0 0 0 0 0 9 0 0 5 0 013 0 0 0 0 0 0 0 0"

Fig. 6. The S-tree encoding at phases I, II, III, and IV

Note that the actual state index, rather than the symbol, of each component is used. This helps to distinguish the same component at different orientations. Also, to elimi-

nate any potential ambiguity, each state index takes two characters. Therefore, the spaces in the S-tree encoding are important.

In the CA space, a structure may be translated, rotated, and/or permuted during processing. The S-tree encoding can handle each of these conditions. First, since the S-tree encoding is independent of absolute position, it can be used to detect a structure arbitrarily translated. Second, the S-tree indicates that a string encoding at 4 different phases is equivalent to the structure rotated to 4 different orientations. Therefore, by detecting the way the S-tree phase has been shifted, the model can determine how the structure has been rotated. Further, if the structure's components have weak symmetry, the rotation of the structure will also cause the state of its individual components to be permuted. This can be handled by permuting each state by 90° every time the S-tree encoding shifts its phase. For instance, S-tree at phase II of the structure shown in Fig. 5 is identical to the S-tree at phase I of the structure shown in Fig. 1.

3 R-trees: General Rule Set Encoding

A CA rule determines the state of a cell at time t+1 based on the states of the cell and its adjacent neighbors at time t. The complete set of such rules, called the rule table, determines the state transition of each cell in the CA space. The previous study evolving rules for self-replicating CA structures adopted a linear encoding of the rules [7]. The essential idea is that the rule table took the form of a linear listing of the entire rule set. Each rule was encoded as a string CTRBLC', where each letter specifies respectively the current states of the Center, Top, Right, Bottom, and Left cells, and the next state C' of the Center cell. Let's denote the total number of states as N_s. The rule table will contain $(N_s)^5$ individual rules. The simple structure shown in Fig. 1 has 17 states, so a huge rule table of 17^5 = 1,419,857 rules is needed. This means that each cell has to make, in the worst case, 5 x 17^5 = 7,099,285 comparisons for a single state transition. Second, genetic operators have to manipulate individuals in such an enormous search space that computational barriers become prohibitive for the rule table to evolve effectively when the structure's complexity is moderately increased [7]. This section introduces R-tree encoding, which is much more efficient and effectively resolves the limitations of linear encoding.

An R-tree is essentially a rooted and ordered tree that encodes every rule needed to direct the state transition of a given structure, and only those rules. The root is a dummy node. Each node at level 1 represents the state of a cell at time t. Each node at level 2, 3, 4, and 5 respectively, represents the state of each von Neumann neighbor of the cell (without specifying which is top, left, bottom, and right). Each node at level 6 (the leaf node) represents the state of the cells at time t+1. An example R-tree is shown in Fig. 7, which has an equivalent rule table shown in Fig. 8. Rule 1 corresponds to the leftmost branch going to the 1st (leftmost) leaf, rule 2 corresponds to the 2nd leaf, etc.

The R-tree has the following properties: 1) it is a height balanced and parsimonious tree, since each branch has precisely a depth of 6; 2) the root and each node at level 1, 2, 3, and 4 may have maximum N_s child nodes, which are distinct and sorted by the state index; 3) each node at level 5 has precisely one child, which is a leaf; 4) it handles arbitrarily rotated cells with a single branch and therefore guarantees that there

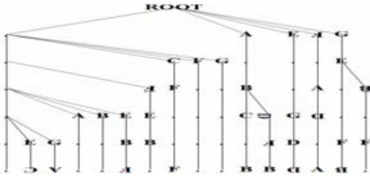

Fig. 7. An example R-tree

Rule 1:	0	0	0	0	0	->	0
Rule 2:	0	0	0	0	17	->	11
Rule 3:	0	0	0	0	25	->	3
Rule 4:	0	0	0	1	0	->	0
Rule 5:	0	0	0	5	0	->	0
Rule 6:	0	0	0	17	5	->	23
Rule 7:	0	0	23	17	5	->	0
Rule 8:	0	9	21	0	0	->	21
Rule 9:	0	21	0	0	0	->	0
Rule 10:	0	25	0	0	0	->	0
Rule 11:	1	0	5	9	0	->	5
Rule 12:	1	0	5	16	23	->	5
Rule 13:	17	0	0	25	13	->	15
Rule 14:	23	0	1	15	0	->	1
Rule 15:	25	17	0	0	21	->	7
Rule 16:	25	17	7	0	21	->	0

Fig. 8. The equivalent rule table

always exists at most one path that applies to any cell at any time, even after rotating and or permuting its orientation.

Due to the R-tree properties described above, the worst search cost for a single state transition is reduced to $5\ln(N_s)$ (5 nodes on each path to leaf, each has maximum N_s child nodes, ordered for quicksort search). Therefore, the ratio of the run cost between linear and R-tree encoding is: $5(N_s)^5/5\ln(N_s) = (N_s)^5/\ln(N_s)$. This means, for a simple structure shown in Fig. 1, R-tree encoding is $(17)^5/\ln(17) \approx 500{,}000$ times more efficient than linear encoding. The more complex a CA structure is, the better an R-tree encoding will outperform the linear encoding.

R-trees also allow efficient genetic operations that manipulate sub-trees. As with regular genetic programming, the R-tree crossover operator, for instance, swaps sub-trees between the parents to form two new R-trees. However, the challenge is to ensure that the crossover operator results in new trees that remain valid R-trees. If we simply pick an arbitrary edge E_1 from R-tree$_1$ and edge E_2 from R-tree$_2$, randomly, and then swap the sub-trees under E_1 and E_2, the resulting trees, for example, may no longer be height balanced. This problem can be resolved by restricting R-tree crossover to be homologous one-point crossover. The essential idea is as follows. After selecting the parent R-trees, traverse both trees (in a breadth first order) jointly in parallel. Compare the states of each visited node in the two different trees. If the states match, mark the edge above the node as a potential crossover point (PCP). As soon as a mismatch is seen, stop the traversal. Next, pick an edge from the ones marked as PCP's, with uniform probability, and swap the sub-trees under that edge between both parent R-trees.

R-tree crossover as defined above has clear advantages over linear-representation crossover. First, R-tree crossover is potentially equivalent to a large set of linear crossovers. Second, linear crossover randomly selects the crossover point and hence is not context preserving. R-tree crossover selects a crossover point only in the common upper part of the trees. This means that until a common upper structure emerges, R-tree crossover is effectively searching a much smaller space and therefore the algorithm quickly converges toward a common (and good) upper part of the tree, which cannot be modified again without the mutation operator. Search incrementally concentrates on a slightly lower part of the tree, until level after level the entire set of trees converges. The R-tree mutation operator simply picks an edge from the entire tree with uniform probability, and then eliminates the sub-tree below the edge. The R-tree encoding and genetic operators used allow CA rules to be constructed and evolved under a non-standard schema theorem similar to one proposed for genetic programming [8], even though R-trees do not represent conventional sequential programs.

4 Genetic Programming with S-Trees and R-Trees

Given an arbitrary structure for which a R-tree is sought to make the structure self-replicating, the seed is first encoded by an S-tree string, and then the R-tree is evolutionarily synthesized as follows:

```
Evolve_RTree (S, T, p_c, p_m)
    S: the R-tree population size
    T: the tournament selection size
    p_c: the fraction of S to be replaced by crossover at each generation
    p_m: the fraction of S to be replaced by mutation at each generation
  Initialization
    - Encode seed configuration as an S-tree string
    - Initialize Current_Population, with R-trees each with one branch "ROOT . . . . . ."
    - Max_Time = 1, Terminate = false
  WHILE Terminate == false DO
    FOR each R-tree in Current_Population DO
      Each CA cell advances time from 0 to Max_Time, directed by current R-tree
      IF missing rule condition THEN
        allow the R-tree to self-expand, with the leaf state randomly selected
      ENDIF
      Compute the fitness of the R-tree based on the S-tree encoding
      Prune inactive branches in the R-tree
    ENDFOR
    IF terminate condition THEN
        - Terminate = true
    ELSE IF fitness no longer improves THEN
        - Max_Time = Max_Time + 1
    ENDIF
    FOR RTree_Pair from 1 to S/2 DO
        - Randomly pick two parent R-trees using tournament selection.
        - Generate a random number p in (0,1)
        IF p_c > p THEN
            -Perform crossover and store offspring R-trees in Temporary_Population
        ELSE
            -Directly store the parents in Temporary_Population
      ENDIF
    ENDFOR
    FOR each R-tree in Temporary_Population DO
        - Generate a random number p in (0,1)
        IF p_m > p THEN
            - Mutate the R-tree
        ENDIF
    ENDFOR
    SET Current_Population = Temporary_Population
  ENDWHILE
  RETURN the R-tree with highest fitness
```

In the algorithm depicted above, "missing rule condition" means no path/leaf in the R-tree applies to change that cell's state even after rotating and permuting its von Neumann neighbors, "terminate condition" means finding a set of rules capable of constructing the replicated structures or reaching a pre-specified maximum number of iterations, and "fitness no longer improves" means the best fitness at each generation is not further increased, after a configurable number, say 300, of continuous GP generations. Therefore, only gradually do the number of CA iterations increase over time as fitness im-

proves; this was an important factor in controlling the R-tree size and increasing algorithm efficiency. Typically we used S = 100, T = 3, $p_c = 0.85$, and $p_m = 0.15$.

The fitness of an R-tree is evaluated in terms of how well the states it produces "match" the structural information encoded in the S-tree. More specifically, the following fitness functions are defined: 1) the matched density measure f_d evaluates how many components appearing in the S-tree encoding are detected; 2) the matched neighbors measure f_n evaluates how many Moore neighbors of the components found above also match the neighborhood encoded by the S-tree encoding; 3) the matched component measure f_c evaluates how many components found above have their Moore neighbors perfectly matching the S-tree encoding; and 4) the matched structure measure f_s evaluates the number of root components which perfectly match the entire S-tree encoding. The overall R-tree fitness function is then defined as: f $= w_d * f_d, + w_n * f_n, + w_c * f_c, + w_s * f_s$. Typical weights that we used were: $w_d = 0.1$, $w_n = 0.1$, $w_c = 0.4$, $w_s = 0.4$. In the early generations of the evolutionary process described above, f_d encourages the same components to appear in the cellular space as in S-tree encoding (other measures will likely be near 0 at this phase). Early or late, f_n comes into play and rewards the R-trees which tend to organize components to form a neighborhood that is the same as in the Moore graph. Naturally, sometimes components will appear with Moore neighbors perfectly matching the S-tree, and so f_c will cause a significant jump in the overall fitness. Eventually, the perfectly matched components may form replicates of the original structures, which will be strongly rewarded by f_s.

In sum, the R-tree encoding is evolutionarily, adaptively, incrementally, and parsimoniously self constructed from the S-tree encoding, through genetic programming. As a result, replicates of an arbitrary seed structure can be synthesized.

```
S-tree encoding(I)   =" 1 0 0 513 9 0 0 0 0 0172521 0 0 0 0 0 0 0 0 0"
S-tree encoding(II)  =" 1 513 9 0 0 0 0 0 017 02521 0 0 0 0 0 0 0 0 0"
S-tree encoding(III)=" 1 9 0 0 0 0 0 513 0 021 0 01725 0 0 0 0 0 0 0"
S-tree encoding(IV)  =" 1 0 0 0 0 513 9 0 0 0172521 0 0 0 0 0 0 0 0 0"
```

Fig. 9. The S-tree encoding

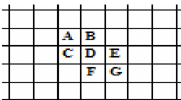

Fig. 10. The seed

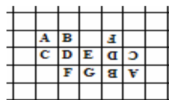

Fig. 11. At $t = 1$

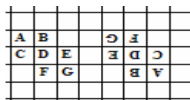

Fig. 12. At $t = 2$

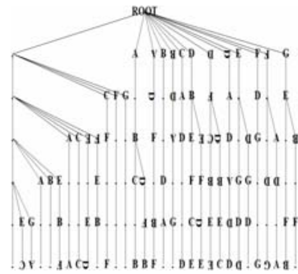

Fig. 13. The R-tree

5 Experimental Results

The model described above was tested in a number of experiments. We achieved success with structures of arbitrary shape and varying numbers of components. The largest seed structure for which it was previously possible to evolve rules with over a week's computation on a super-computer had 4 components [7]. Figure 10 shows one of the seed structures, consisting of 7 oriented components, for which our approach found a rule set that allowed the structure to self-replicate. The R-tree (Fig. 13) was evolved from the S-tree encoding (Fig. 9) after about 20 hours of computation on an

IBM ThinkPad T21 laptop. With the resultant R-tree, at time t=1 (Fig. 11), the structure starts splitting (the original translates to the left while a rotated replica is being born to the right). At time t=2 (Fig. 12), the splitting completes and the original and replica structures become isolated. Thus, the seed structure has replicated after only 2 time steps, a remarkably fast replication time that has never been reported before. As time continues, more replicas appear, with debris remaining between replicas (not illustrated). These experiments suggest that our model is much more efficient than previous genetic algorithm models [7].

6 Conclusions and Future Work

In this article, we introduced an S-tree—R-tree—structure synthesis model coupled with genetic programming methods. Our experimental results so far indicate that such a model is indeed capable of evolving rule sets that make arbitrary structures of limited size self replicate, as well as efficient computation. There is much room for further study and additional experiments. For instance, one motivation for the S-tree encoding is that it should eventually allow both structure and rules to evolve concurrently and cooperatively. The S-tree and R-tree encoding might also be used to evolve rule sets replicating extra structures in addition to the seed itself, or structures with higher complexity than the seed, etc. Acknowledgements: JR's work on this project is supported by NSF award IIS-0325098.

References

1. J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966. Edited and completed by A. W. Burks.
2. Sipper, M. (1998). Fifty years of research on Self-Reproduction: An overview. *Artificial Life*, 4, 237-257.
3. Langton C, Self-Reproduction in Cellular Automata, *Physica D*, 10, pp. 135-144, 1984.
4. Reggia J, Armentrout S, Chou H & Peng Y, Simple Systems That Exhibit Self-Directed Replication, *Science*, 259, 1282-1288, 1993.
5. Andre D, Bennett F, & Koza J. Discovery by Genetic Programming of a Cellular Automata Rule ..., Proc. First Ann. Conf. on Genetic Programming, MIT Press, 1996, 3-11.
6. Richards F, Meyer T & Packard N. Extracting cellular automaton rules directly from experimental data, Physica D, 45, 1990, 189-202.
7. Lohn J & Reggia J. Automated discovery of self-replicating structures in cellular automata.' IEEE Trans. Evol. Comp., 1,1997, 165-178.
8. Poli R and Langdon W, Schema theory for genetic programming with one-point crossover and point mutation, Evolutionary Computation, 6, 231-252, 1998.