ABSTRACT

Title of dissertation: ARTIFICIAL EVOLUTION OF ARBITRARY SELF-REPLICATING CELLULAR AUTOMATA

Zhijian Pan
Doctor of Philosophy, 2007

Dissertation directed by: Professor James Reggia
Department of Computer Science

Since John von Neumann's seminal work on developing cellular automata models of self-replication, there have been numerous computational studies that have sought to create self-replicating structures or "machines". Cellular automata (CA) has been the most widely used method in these studies, with manual designs yielding a number of specific self-replicating structures. However, it has been found to be very difficult, in general, to design local state-transition rules that, when they operate concurrently in each cell of the cellular space, produce a desired global behavior such as self-replication. This has greatly limited the number of different self-replicating structures designed and studied to date.

In this dissertation, I explore the feasibility of overcoming this difficulty by using genetic programming (GP) to evolve novel CA self-replication models. I first formulate an approach to representing structures and rules in cellular automata spaces that is amenable to manipulation by the genetic operations used in GP. Then, using this representation, I demonstrate that it is possible to create a "replicator factory" that provides an unprecedented ability to automatically generate whole families of self-replicating structures and that allows one to systematically investigate the properties of replicating structures as one varies the initial configuration, its size, shape, symmetry, and allowable states. This approach is then extended to incorporate multi-objective fitness criteria, resulting in production of diversified replicators. For example, this allows generation of target structures whose complexity greatly exceeds that of the seed structure itself. Finally, the extended multi-objective replicator factory is further generalized into a structure/rule co-evolution model, such that replicators with unspecified seed structures can also be concurrently evolved, resulting in different structure/rule combinations and having the capability of not only replicating but also carrying out a secondary pre-specified task with different strategies. I conclude that GP provides a powerful method for creating CA models of self-replication.

ARTIFICIAL EVOLUTION OF ARBITRARY SELF-REPLICATING CELLULAR AUTOMATA

by

Zhijian Pan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Dedication

To my beloved parents,

Wenyou Pan

and

Zhenyu Cai

who fill in my entire life with their everlasting care and love.

Acknowledgments

I am most grateful to my adviser and role model professor James Reggia, who has helped me to find this wonderful research topic, has always trusted me, and has provided me guidance in every aspect of my life. If it was not for his countless encouragement and help, I would not be able to complete this dissertation. I also want to extend my thanks to his wonderful wife, Carol Reggia, who was always kind enough to allow me to often visit Jim at their home and spend his family time with me discussing research issues.

I am also very grateful to Professor Clyde Kruskal, Atif Memon, Ramani Duraiswami, and Avis Cohen, who agree to serve on my defense committee and dedicate their time to review my dissertation.

I want to thank my employer, IBM, especially my director Randy Freeman and other colleagues at Annapolis Lab, who offered me tuition funds, emotional encouragement, as well as computational resources, without which it would be impossible for me to complete this doctoral research outside of my full time work.

Most importantly, I am whole-heartedly grateful to my lovely daughters, Alvina Pan and Kianna Pan, for being happy and healthy, for their unique power brightening me from the tremendous stress I endured in the pursuit of this doctoral study despite of a full time job and family responsibilities, and particularly for their forgiveness despite of my lack of time playing with them as often as they wished. And finally, I sincerely thank Donghong Gao, their mother, my wife, for everything she did for the family.

Table of Contents

List of Tables vii

List of Figures vii

List of Abbreviations xviii

1 Introduction 1
- 1.1 Motivations . . . 1
- 1.2 Contributions . . . 4
- 1.3 Dissertation Outline . . . 7

2 Background and Previous Work 8
- 2.1 Cellular Automata (CA) . . . 9
- 2.2 Embedding Self-Replicating Systems in CA . . . 13
  - 2.2.1 Universal Constructors in CA Space . . . 14
  - 2.2.2 Self-Replicating Loops . . . 16
  - 2.2.3 Self-Replicating Loops Expanding Problem Solutions . . . 18
  - 2.2.4 Other Self-Replicating Structures in CA Space . . . 22
- 2.3 Evolutionary Computation, Cellular Automata, and Self-Replication . 23
  - 2.3.1 Evolutionary Computation . . . 23
  - 2.3.2 Evolutionary Multi-Objective Optimization . . . 26
  - 2.3.3 Evolution of CA Rules . . . 27
  - 2.3.4 Evolution of CA Rules for Self-Replication . . . 28

3 Evolution of Self-Replicating Structures Using Genetic Programming 32
- 3.1 Overview . . . 32
- 3.2 S-tree Encoding and General Structure Representation . . . 32
  - 3.2.1 Moore Graph . . . 32
  - 3.2.2 MST and S-tree Generation . . . 34
  - 3.2.3 S-Tree Encoding for Arbitrary Structures . . . 38
- 3.3 R-tree Encoding and Rule Set Representation . . . 39
  - 3.3.1 R-tree Encoding . . . 40
  - 3.3.2 R-tree Genetic Operators . . . 43
    - 3.3.2.1 R-tree Crossover . . . 43
    - 3.3.2.2 R-tree Mutation . . . 45
- 3.4 Genetic Programming with S-tree and R-tree Encoding . . . 46
  - 3.4.1 Problem Formulation . . . 46
  - 3.4.2 R-tree Initialization . . . 47
  - 3.4.3 CA Simulation with R-tree . . . 48
  - 3.4.4 R-tree Fitness Assignment Based on S-tree . . . 52
    - 3.4.4.1 S-tree Probing . . . 52
    - 3.4.4.2 Determining the Number of Acceptable Probes . . . 61
    - 3.4.4.3 Overall Fitness Function . . . 63

3.4.5 R-tree Fitness Sharing 64
3.4.6 R-tree Tournament Selection 64
3.4.7 R-tree Evolution 65
3.4.8 R-tree Elitism 67
3.4.9 The Resulting Overall GP Model 67
3.5 Initial Experimental Results 69
3.6 Conclusions 82

4 The Replicator Factory and Properties of Discovered Self-Replicating Cellular Automata Systems 86
4.1 Overview 86
4.2 The Replicator Factory and Debris Cleaning with Multi-Staged GP . 86
4.3 Results: Properties of Evolved Self-Replicating Structures 88
4.3.1 Seed Size 89
4.3.2 Seed Shape 94
4.3.3 Cell States 99
4.3.4 Repeated Components 103
4.3.5 Repeated Sub-Structures 106
4.3.6 Type of Symmetry 109
4.4 Conclusions 113

5 Multi-Objective Artificial Evolution toward Task-Performing Self-Replication with Specified Seed Structures 115
5.1 Overview 115
5.2 Problem Formulation 119
5.3 R-tree Fitness Assignment with Multiple S-trees 121
5.3.1 Determining Acceptable Seed Probes and Target Probes . . . 122
5.3.2 Interlaced Acceptance of Seed Probes and Target Probes . . . 124
5.3.3 Fitness Function for Replicatability 128
5.3.4 Fitness Function for Taskability 129
5.3.5 Fitness Function for Sustainability 129
5.4 R-tree Tournament Selection Based on Multiple Criteria 130
5.4.1 Domination Pressure 133
5.4.2 Distribution Pressure 134
5.4.3 Parsimony Pressure 135
5.5 Multi-Objective R-tree Elitism 136
5.6 The Resulting Extended Multi-Objective Replicator Factory Model . 137
5.7 Task-Performing Self-Replication: Experimental Results 140
5.8 Conclusions 165

6 Multi-Objective Structure/Rule Co-evolution with Unspecified Seed Structures 168
6.1 Overview 168
6.2 Problem Formulation 169
6.3 Replicator Initialization 171

6.3.1 S-tree Initialization with Random Structures 171
6.3.2 R-tree Initialization for Co-evolution 172
6.4 Replicator Simulation 172
6.5 Replicator Fitness Evaluation and Tournament Selection 172
6.6 Replicator Evolution 173
6.6.1 Evolution in Structural Space with S-tree Genetic Operators . 174
6.6.2 Evolution in Rule Space with R-tree Genetic Operators 177
6.7 The Resulting Generalized Replicator Factory Supporting Structure/Rule Co-evolution 178
6.8 Structure/Rule Co-evolution: Experimental Results 181
6.9 Conclusions 197

7 Conclusions and Future Work 203

Bibliography 210

List of Tables

4.1 Performance measures for replicators in Figure 4.2. 90

4.2 Performance measures of replicators in Figure 4.6. 96

4.3 Performance measures of 2 sets of replicators shown in Figure 4.9. . . 99

4.4 Performance measures of the replicators shown in Figure 4.11. 103

4.5 Performance measures of replicators in Figure 4.13. 106

4.6 Performance measures of the replicators in Figure 4.16. 113

List of Figures

2.1 Neighborhood in 1-D Cellular Automata space. 10

2.2 Neighborhood in 2-D Cellular Automata space. 10

2.3 von Neumann's theoretical model: a universal constructor 15

2.4 Langton's structure is a sheathed square loop. First self replication takes 151 time steps. 17

2.5 A self-replicating loop in a 2D cellular automata space. Read clockwise starting at the lower right, there are a series of signals (+-) embedded in the loop structure, each indicating that the arm on the bottom right should grow out one step. These are followed by two signals indicating a left turn (L- L-). These signals circulate counterclockwise around the loop, advancing one cell per time step. As they do, copies of the signals pass out the arm at the lower right, causing it to extend and turn so that a second "child" loop is constructed. . 17

2.6 Successive states of a self-replicating loop that started at time t=0 as illustrated in Figure 2.5 [57]. The instruction sequence repeatedly circulates counterclockwise around the loop with a copy periodically passing onto the construction arm. At t=3 (**a**) the sequence of instructions has circulated three positions counterclockwise with a copy also entering the construction arm. At t=6 (**b**) the arrival of the first + state at the end of the construction arm produces a growth cap of X's. This growth cap, which is carried forward as the arm subsequently extends to produce the replica, is what makes a sheath unnecessary by enabling directional growth and right-left discrimination. Successive arrival at the growth tip of +'s extends the emerging structure and arrival of paired L's causes left turns, resulting in eventual formation of a new loop. Intermediate states are shown at t=80 (**c**) and t=115 (**d**). By t=150 (**e**) a duplicate of the initial loop has formed and separated (on the right); the original loop (on the left, construction arm having moved to the top) is beginning another cycle of self-directed replication. 19

2.7 A self replicating loop using only five unique components [57]. Shown here are eleven immediately successive configurations. Starting at $t = 0$, the initial state (shown at the upper left) passes through a sequence of steps until at $t = 10$ (last structure shown) an identical but rotated replica has been created. 20

2.8 Self-replicating loop with secondary capability: it can writes out LSL after embedding a pre-written computer program in the seed structure [68, 65]. 21

2.9 A canonical evolutionary computation algorithm. 24

2.10 GA (left) vs. GP (right). 25

2.11 The behavior of the best rule evolved by genetic programming on one set of initial states for the majority classification task. 29

2.12 An example self-replicating structure discovered by genetic algorithm. 30

2.13 A rule table is a linear listing of every possible combination of every state required for a given structure. 31

3.1 An example structure, its Moore graph, and its S-tree graph. 33

3.2 S-trees at different phases vs. rotated structures. 36

3.3 An example state-transition using a rule in an R-tree: a cell with state A at T=t becomes state B at T=t+1, after firing a rule in the R-tree based on the states of its von Neumann neighbors as shown. . 40

3.4 A list of 16 rules encoded by the sample R-tree shown in Figure 3.3. Note each path from the root node to a leaf represents a rule in the table, while each state shown in the table is its state index number rather than its symbolic representation shown in the tree. 41

3.5 One-point homologous crossover between parent R-trees. A crossover point is selected at the *same* location in both parent trees, ensuring that the child trees are valid R-trees. The children R-trees are formed by swapping the shaded sub-trees. 44

3.6 The R-tree point mutation simply deletes a sub-tree (here, the one indicated by shading), allowing the CA simulation to fill it in with a new randomly-generated subtree when needed. 45

3.7 The same cell can be probed by the same S-tree in 4 phases, to evaluate a potential match in 4 different orientations. Note such probing can be done with any cell in the configuration. 54

3.8 An actual S-tree (any phase) contains the immediate quiescent neighbors surrounding the structure. These quiescent cells are also traversed and inspected during the probing (in every orientations), and thus help to detect how much the structure is isolated in the current configuration. 56

3.9 A schematic view of the S-tree/R-tree based GP model toward self-replication 68

3.10 Self-replication, example 3.1: the seed, a 7-oriented-component structure, is programed to self replicate in only 2 time steps. In subsequent time steps, each replica attempts to repeat the same action, when collisions result. However, eventually, when enough space is reached, more replicas can be isolated from each other. More details of such a process is illustrated in the next example, shown in Figure 3.12. . . . 70

3.11 Self-replication, example 3.1: the R-tree automatically evolved from the given seed structure with the method established in this chapter. This example R-tree uses the von Neumann neighborhood. 71

3.12 Self-replication, example 3.2-1: executing the automatically programmed R-tree against a 6-oriented-component seed structure, from t=0 to t=6. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step. 73

3.13 Continuation of Figure 3.12 from t=7 to t=10. 74

3.14 Continuation of Figure 3.13 from t=11 to t=14. 75

3.15 Self-replication, example 3.2-2: executing a separately generated R-tree against the same 6-oriented-component seed structure, from t=0 to t=5. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step. 76

3.16 Self-replication, example 3.3: executing an automatically generated R-tree against a very large, irregular and arbitrary seed structure, from t=0 to t=4. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step. 77

3.17 Continuation of Figure 3.16 from t=4 to t=5. 78

3.18 Continuation of Figure 3.17 at t=12. 79

3.19 Continuation of Figure 3.18 at t=13. 80

3.20 Continuation of Figure 3.19 at t=14. 81

4.1 An example of the self-replication process, without (a) and with (b) debris removal. The seeds, number of time steps, and resulting replicators in (a) and (b) are identical, but the R-tree used in (a) is produced without encouraging debris removal while the R-tree used in (b) is produced with debris removal encouraged in the fitness function. 87

4.2 Examples of sixteen structures which vary from 4 to 56 components. With the rules evolved by the replicator factory, each of these structures replicates itself within $\leq$4 steps. See text for details. 90

4.3 The replicator factory evolved an R-tree causing this 7x8 structure (t = 0) to replicate in just 4 time steps. The seed structure splits into two isolated replicators as both of the replicas translate diagonally in opposite directions at light speed (one cell each time step). 91

4.4 The linear regression chart between the seed size and timeDup produced from the sample data shown in Table 4.1. This chart shows the observed values from the table, the regression line and both types of confidence interval. The determination coefficient value R^2 is shown on the chart title [113]. 92

4.5 When a sample shape is varied from (a) to (d) above, the correspondingly change in Average Number of Active Moore Neighbors and Moore Graph Coverage is illustrated in (b) and (e), and (c) and (f) respectively. The numbers in (b) and (e) reflect the number of active Moore neighbors of that cell. The shaded cells in (c) and (f) are the cells covered by the Moore Graph of the shown structures. This example shows that a shape of higher complexity tends to reflect lower average number of active Moore neighbors and higher number of cells in the Moore Graph. 94

4.6 Fourteen structures having identical size and allowable states, but varying shapes. Rules to make each self-replicate were produced by the replicator factory. 95

4.7 Sample replication process for one of the 14 structures of Figure 4.6. Note that the splitting replicators move at light speed toward opposite directions and get isolated in only 2 time steps. Further, the replicators have aligned themselves adaptively according to its own shape in a way that they take the minimum cellular space but nevertheless fully isolate from each other. 97

4.8 Linear regression chart between evoClean and the S-tree size (one of the shape complexity measures) derived from Table 4.2. This chart shows the observed values from the table, the regression line and both types of confidence interval around the predictions. The R^2 value shown on the chart title is interpreted as the proportion of the variability of the dependent variable (evoClean) linearly explained by the model (the S-tree size). 98

4.9 Samples of 2 sets of replicators: the first set (a) has size of 25, square shape, and gradually increasing number of allowable cell states (from 6 to 15). The second set (b) has size of 10, loop shape, and gradually increasing number of allowable cell states (from 7 to 13). 100

4.10 The loop structures in set 2, as one example shown here, quickly (within only 2 time steps) reaches enough cellular space to form replicators by moving, sometimes plus rotating (as shown here), both the parent and child structures at the same time. A manually designed loop, however, typically replicates itself by extending a single construction arm while the parent loop itself remains still, and hence usually takes a lot more time steps to reach enough to form a child replicator. 101

4.11 Three pairs of replicators used to study the impact of repeated components. Structures in each pair have the same size, shape, number of allowable states, but only the first one (A in each pair) has repeated components. 103

4.12 Rescaled parallel-coordinates-plot produced from Table 4.4 visualizes the impact of the presence of repeated components. The structures are grouped by the ifRepCom column, and the mean values of each performance measurement of each group is rescaled on a scale of 0∼1 such that all variables are represented on the same scale (for each measurement, 0 corresponds to the minimum and 1 to the maximum). The "Yes" line in the chart represents those observations of which the value of ifRepCom is Yes, and the "No" line in the chart represents those observations of which the value of ifRepCom is No. 105

4.13 Four pairs of replicators: structures in each pair are identical except only the first structure contains repeated sub-structures. 106

4.14 Self-replication of the first pair of replicators in Figure 4.13. The upper-left seed has repeated sub-structures. The bottom left seed does not. 107

4.15 Rescaled parallel-coordinates-plot and mean chart visualizing the impact of the presence of repeated sub-structures. The presence of repeated sub-structures consistently lowers each measured evolutionary and replication cost. The "Yes" line in the chart represents those observations of which the value of ifRepStr is Yes, and the "No" line in the chart represents those observations of which the value of ifRepStr is No. 108

4.16 Sample seeds with reflection (top row), rotation (2^{nd} row), translation (3^{rd} row), and glide-reflection (last row) symmetry. These structures otherwise have exactly the same sub-structures, size, and number of allowable states. 110

4.17 Self-replication of two of the structures in Figure 4.16. The structure on the left has reflection symmetry while the one on right has glide-reflection symmetry. 111

4.18 Rescaled parallel-coordinates-plot and mean charts visualizing the impact of the presence of four types of symmetry. It seems that glide-reflection often leads to much higher evolutionary cost than other types of symmetry. 112

5.1 With respect to the current R-tree shown in yellow, R-trees in the brown area dominate the current R-tree, the ones in the gray area are dominated by the current R-tree, and the one in white areas are indifferent. 132

5.2 A schematic view of the multi-objective replicator factory model. . . 138

5.3 Secondary task performance, example 5.1: (a)the pre-specified target structure, which is a 6-component 29-state weak-rotational-symmetric structure, (b)the pre-specified seed structure, which is a 4-component 29-state weak-rotational-symmetric structure. For illustrative purpose, the seed structure is chosen to spell the English word "seed", and the target structure is likewise chosen to spell the English word "target". An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., "manufacture" the target structure as it replicates. 141

5.4 Secondary task performance, example 5.1-1: executing first automatically programmed R-tree against the given seed structure from t=0 to t=6. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 142

5.5 Continuation of Figure 5.4 from t=11 to t=12. 143

5.6 Continuation of Figure 5.5 from t=13 to t=14. 144

5.7 Secondary task performance, example 5.1-2: executing second automatically programmed R-tree against the given seed structure from t=0 to t=6. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 146

5.8 Continuation of Figure 5.7 from t=7 to t=8. 147

5.9 Continuation of Figure 5.8 at t=12. 148

5.10 Continuation of Figure 5.9 at t=13. 149

5.11 Continuation of Figure 5.10 at t=14. 150

5.12 Secondary task performance, example 5.2: (a)the target structure, pre-specified as a 8-component 7-state weak-rotational-symmetric structure which writes out an English letter “U”; (b)the seed structure, pre-specified as a 3-component 7-state weak-rotational-symmetric structure. An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., “manufacture” the target structure as it replicates. 153

5.13 Secondary task performance, example 5.2: executing an automatically programmed R-tree against the given seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 154

5.14 Continuation of Figure 5.13 at t=6 and t=8. 155

5.15 Continuation of Figure 5.14 at t=10. 156

5.16 Continuation of Figure 5.15 at=12. 157

5.17 Continuation of Figure 5.16 at t=14. 158

5.18 Secondary task performance, example 5.3: (a)the target structure, pre-specified as a 18-component 7-state weak-rotational-symmetric structure which writes out "UM", the acronym for University of Maryland; (b)the seed structure, pre-specified as a 3-component 7-state weak-rotational-symmetric structure, same seed used in example 5.2. An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., "manufacture" this bigger, more complicated target structure. 159

5.19 Secondary task performance, example 5.3: executing an automatically programmed R-tree against the given seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 160

5.20 Continuation of Figure 5.19 at t=6 and t=7. 161

5.21 Continuation of Figure 5.20 at t=8 and t=9. 161

5.22 Continuation of Figure 5.21 at t=10. 162

5.23 Continuation of Figure 5.22 at t=12. 163

5.24 Continuation of Figure 5.23 at t=14. 164

6.1 An example of structure evolution with one-point S-tree crossover. A crossover takes place between S-tree$_1$ (a, left) and S-tree$_2$ (a, right) with the shaded sub trees exchanged. Correspondingly, Structure$_1$ (b, left) and structure$_2$ (c, left) are now replaced by structure$_{1new}$ (b, right) and structure$_{2new}$ (c, right), respectively. The resulting new S-trees after the crossover operation now represent the new structures respectively. 175

6.2 A schematic view of the generalized multi-objective co-evolving replicator factory model. 179

6.3 Structure/Rule co-evolution, example 6.1-1: (a)the target structure, pre-specified as a 6-component 29-state weak-rotational-symmetric structure, same target structure used in secondary task performance example 5.1 (Figure 5.3); (b)an automatically discovered seed structure, which is a 3-component 29-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree. 182

6.4 Structure/Rule co-evolution, example 6.1-1: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 183

6.5 Continuation of Figure 6.4 from t=6 to t=7. 184

6.6 Continuation of Figure 6.5 at t=14. 185

6.7 Structure/Rule co-evolution, example 6.1-2: (a)the target structure, same target structure as in example 6.1-1 above (Figure 6.3); (b)a different automatically discovered seed structure, which is also a 3-component 29-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree. 186

6.8 Structure/Rule co-evolution, example 6.1-2: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 187

6.9 Continuation of Figure 6.8 at t=6 and t=7. 188

6.10 Continuation of Figure 6.9 at t=8 and t=9. 188

6.11 Continuation of Figure 6.10 at t=17. 189

6.12 Structure/Rule co-evolution, example 6.2: (a)the pre-specified 8-component 7-state target structure, same target structure used in task performance example 5.2 (Figure 5.12); (b)an automatically discovered seed structure, which is also a 4-component 7-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree. 190

6.13 Structure/Rule co-evolution, example 6.2: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 191

6.14 Continuation of Figure 6.13 at t=6 and t=7. 192

6.15 Continuation of Figure 6.14 at t=12. 193

6.16 Continuation of Figure 6.15 at t=14. 194

6.17 Structure/Rule co-evolution, example 6.3: (a)the pre-specified target structure, a very big 28-component 7-state weak-rotational-symmetrical structure that writes out "UMD", the domain name of University of Maryland; (b)an automatically discovered seed structure, which is a 4-component 7-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree. 195

6.18 Structure/Rule co-evolution, example 6.3: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation. 196

6.19 Continuation of Figure 6.18 at t=6 and t=7. 197

6.20 Continuation of Figure 6.19 at t=8 and t=9. 198

6.21 Continuation of Figure 6.20 at t=10. 199

6.22 Continuation of Figure 6.21 at t=11. 200

List of Abbreviations

p	CA space
T	CA time
c	CA cell
g	GP generation
L	GP population
M	GP generation size
ξ	GP hesitation
ζ	tounament size
x	replicator
r	R-tree
s	S-tree
h	S-tree phase
μ	R-tree size
λ	S-tree size
σ	R-tree distance
$\mathcal{R}$	R-tree search space
α	R-tree density
γ	structure size
ν	random structure size
ρ	structure density
T^s	simulation time steps
T^v	evaluation time steps
β_δ	replicatability
β_η	productativity
δ	replica count
η	target count
κ	probe count
f_κ	probe measure
π_δ	number of seed tokens
π_η	number of target tokens
f_δ	fitness for replicatability
f_η	fitness for productativity
f_θ	fitness for sustainability
b^r_c	R-tree crossover probability
b^r_m	R-tree mutate probability
b^s_c	S-tree crossover probability
b^s_m	S-tree mutate probability
ϕ	domination pressure
τ	domination count
ψ	distribution pressure
ω	parsimony pressure

Chapter 1

Introduction

1.1 Motivations

Self-replication is a process by which an entity may make a copy of itself. *Self-replicating systems* are systems that are capable of producing copies of themselves. The mathematician John von Neumann is credited with being the first to conduct a formal investigation of artificial self-replicating machines. He believed that self-replicating biological organisms could be viewed as very sophisticated machines, but machines nevertheless. He argued that the important thing about a replicating organism was not the matter from which it is made, but rather the information and the complexity of the interactions between parts of the organism. In particular he asked whether we can use purely mathematical-logical considerations to discover the specific features of biological automata that make them self-replicate. Much subsequent work on artificial self-replicating machines has continued in this spirit, being motivated by the desire to understand the fundamental information processing principles and algorithms involved in self-replication, independent of how they might be physically realized.

It has also been argued that a better understanding of these principles could be useful in atomic-scale manufacturing (nanotechnology), in creating robust electronic systems, in facilitating future planetary exploration, and in gaining a better understanding of the origins of life. For example, nanotechnology is concerned in part with making nano meter scale assemblers. Without self-replication, capital and assembly costs of molecular machines may be impossibly large. On the other hand, the goal of self-replication in space exploration systems is to exploit large amounts of matter with a low launch mass. Once in place, machinery that replicates itself could achieve this goal and could also produce other manufactured objects. Since self-replication is generally seen as a fundamental property of life, the study of self-replicating systems is generally viewed as having a critical role in advancing artificial life research.

Following von Neumann's seminal work [47, 48], much research over the past several decades has focused on the difficult issue of creating models of self-replicating structures in cellular automata (CA) [65]. Past work on this topic has involved at least two major different approaches. The earliest work examined large, complex universal constructors like von Neumann's that are marginally realizable. This work established the feasibility of artificial self-replication, examined many important theoretical issues, and gradually created progressively simpler self-replicating universal systems [11, 63, 71, 72]. A second and more recent approach has involved the manual design of much simpler self-replicating loops [34]. Subsequent work produced simpler and smaller loops [57], and embedded instruction sequences in them so that they performed other tasks as they replicate [10, 53, 68].

However, it has been found to be very difficult and time consuming, in general, to design the local state-transition rules that, when they operate concurrently in each cell of the cellular space, produce a desired global behavior such as self-replication.

This has greatly limited the number of different self-replicating structures designed and studied to date, and has contributed to the lack of systematic study of their properties. Further, implemented past self-replication CA models do not allow arbitrary structures; instead, they mostly assume the restriction that replicas must have a specific loop-like structure. Further, those models that have the capability of performing a secondary function all rely on embedding pre-written manual instructions in the specific seed structures, i.e., at the cost of altering the complexity of the seed structure itself.

In this dissertation, I focus on a third approach that merits investigation: the automated evolution of self-replicators from arbitrary, initially non-replicating systems. Initial work using genetic algorithms showed the potential value of this approach [36, 37, 38], but the effectiveness of this approach to discovering state-change rules for self-replication proved to be quite limited, mainly due to the inherently large linearly-encoded chromosome required to accommodate the use of a genetic algorithm. This initial work led to some pessimism about the viability of evolutionary discovery of novel self-replicating structures.

This study re-visits the issue of using evolutionary methods to discover new self-replicating structures, and suggests that this earlier pessimism is not warranted. The central goal of this study is to formulate an approach to representing structures and rules in cellular automata spaces that are amenable to manipulation by the genetic operations used in genetic programming (rather than genetic algorithms). This representation is used to examine if it is possible to automatically generate a broad range of arbitrary self-replicating structures, and to systematically investigate the properties of such novel discovered self-replicators.

1.2 Contributions

The main contributions of this dissertation are as follows:

1. This is the first work to adopt genetic programming into self-replicating cellular automata models, and to demonstrate that genetic programming provides a powerful method for creating CA models of self-replication. More specifically, this work is the first study to introduce an unambiguous and universal tree encoding for arbitrary replicating structures in CA spaces, as well as a more efficient tree encoding to replace the linear rule table. The structure tree can be viewed as a representation of a desired global configuration, and the rule tree can be viewed as the representation of local state-transition rules. Local rules can now efficiently evolve in the form of trees, and receive fitness measures simply evaluated in terms of how well, when they are simulated in CA spaces, express the desired global configuration. The structure tree makes it possible to convert an arbitrary structure into a common tree-like representation which can be fully exploited by an existing evolutionary system built without requiring any knowledge of the structure a priori. This creates a new paradigm which can be applied to arbitrary future structures. As a result,

   (a) not only the application of genetic programming becomes feasible, but also this paradigm removes the restriction which is often assumed in past CA models that replicas must have a specific loop-like structure;

(b) the paradigm represents a significant step forward in creating self-replicating structures, because it qualitatively reduces the time cost of replicator construction compared to past manual methods;

(c) the replicators created in this fashion are qualitatively different from those generated in past studies (universal constructors and loops); here automatically discovered replicators can construct themselves very quickly, in a fission-like, rotational, and/or spiral process; some of these discovered replicators can self-replicate with only one time step, representing a speed which was not known to even be possible;

(d) the results demonstrate an unprecedented ability to automatically generate whole families of self-replicating structures, so that now it even becomes possible for one to systematically investigate the properties of replicating structures as one varies the initial configuration, its size, shape, symmetry, and allowable states;

(e) this led to statistical results suggesting that the number of GP generations, computation time, and number of resulted rules required by an arbitrary structure to self-replicate are positively and jointly correlated with the number of components, configuration shape, and allowable states in the initial configuration, but inversely correlated with the presence of repeated components or sub-structures, seed symmetry, and self-replicating sub-structures.

2. This is also the first work to introduce multi-objective evolution into self-

replicating cellular automata models, and shows that the new paradigm can incorporate multiple optimization criteria and produce replicators capable of carrying out secondary tasks. In contrast to previous models having similar secondary functionality, the new approach:

(a) automatically programs the CA to support both self-replication and secondary task performing at the same time;

(b) requires no manual design of the seed structure and rules;

(c) requires no pre-writing of manual programs;

(d) requires no instruction-embedding (altering the seed complexity itself);

(e) can add secondary capability to arbitrary given structures, assuming no specific knowledge about seed structures a priori;

(f) can yield multiple diversified self-replicators in a single run, with each providing alternative strategies carrying out the secondary computation;

(g) demonstrates the generation of target structures whose complexity greatly exceeds that of the seed structures.

3. Finally, this is also the first work to create a structure/rule *co-evolution* system, on top of the multi-objective genetic programming paradigm, to discover replicators without being given seed structures, toward the performing of a pre-specified secondary task. Now structures and rules can evolve concurrently and cooperatively, and multiple diversified seed/rule combinations can

be created in a single run, with each providing alternative strategies for carrying out the given task.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. An overview of past work and recent developments involving artificial self-replication implemented as cellular automata (CA) is first given in Chapter 2. Subsequent chapters present the results of my research efforts. Chapter 3 describes how a CA can be automatically programmed using GP to support self-replication of arbitrary structures. Chapter 4 demonstrates the unprecedented ability of this approach to create a whole family of self-replicators, and undertakes the systematic study of their properties. Chapter 5 further shows how this new paradigm can incorporate multi-objective optimization criteria and thereby produce diversified replicators capable of carrying out secondary pre-specified tasks. Chapter 6 shows how a seed structure itself can concurrently and cooperatively evolve along with the rules, yielding a structure/rule co-evolution system that can search for self-replicators without pre-specified seed structures that carry out a given task. Finally, Chapter 7 concludes with a summary, anticipated impact of the results presented in this dissertation, and some ideas about future work.

Chapter 2

Background and Previous Work

Self replicating systems are systems that are capable of producing copies of themselves. The terms of *replication* and *reproduction* are often considered synonymous. However, replication is an ontogenetic, developmental process, potentially involving no genetic operators, resulting in an exact duplicate of the parent organism. Reproduction, on the other hand, is an evolutionary process, involving genetic operators such as crossover and mutation, thereby giving rise to variety and ultimately to evolution [21].

The mathematician von Neumann is the first who asked whether we can use purely mathematical-logical considerations to discover the specific features of biological automata that make them self-replicating [65]. Much subsequent work was also motivated by the desire to understand the fundamental information processing principles and algorithms involved in self-replication, independent of how they might be physically realized. As noted in the preceding chapter, it has also been argued that a better understanding of these principles could be useful in atomic-scale manufacturing (nanotechnology) [93, 94, 95], in creating robust electronic systems [99, 98], in facilitating future planetary exploration [97], and in gaining a better understanding of the origins of life [96].

To establish the context of this study, in the following, I give an overview of past work and recent developments involving artificial self-replication implemented as cellular automata (CA).

2.1 Cellular Automata (CA)

Von Neumann's discussions with Polish mathematician Stanislaw Ulam led to the idea of cellular automata, which they invented as a simplified and mathematically tractable model of the physics of our universe [47, 48]. With cellular automata, an artificial self replicating machine could be simply represented as a configuration of contiguous active cells, each of which represents a component of the machine, and each cell only interacts exclusively with its adjacent neighbors. The "physics" of the universe is encoded in the interaction rules which govern the state transition of each cell. In this way, an artificial self replicating machine could be studied purely on the fundamental, informational, and logical level, without considering its physical implementation details.

CA models are dynamical systems in which space and time are discrete. A cellular automaton consists of a regular grid of cells, each of which can be in one of a finite number of k possible states, updated synchronously in discrete time steps according to a local, identical interaction rule. The state of a cell is determined by the previous states of surrounding neighborhood of cells. Hence, cellular automata have three characteristics: 1) parallelism — individual cells are updated simultaneously and synchronously; 2) locality — the new state of a cell is exclusively based on its old state and the old states of its neighboring cells; and 3) homogeneity — all cells in the space use the same set of rules for updating their states [11, 47, 48, 75, 76].

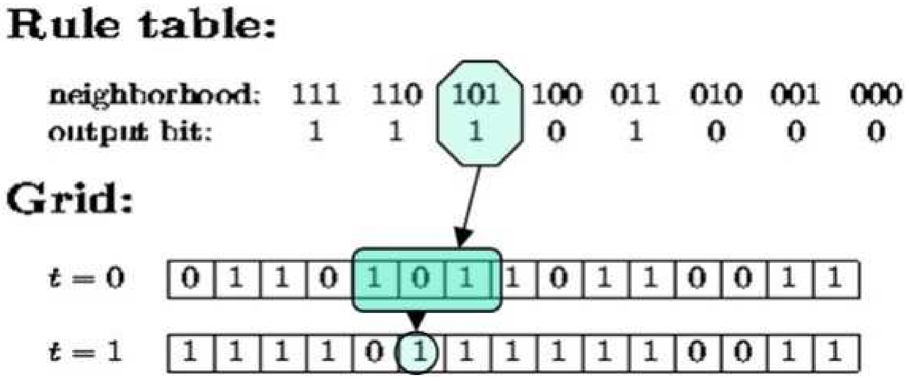

Rule table:
neighborhood: 111 110 101 100 011 010 001 000
output bit: 1 1 1 0 1 0 0 0
Grid:
t = 0 0 1 1 0 1 0 1 1 0 1 1 0 0 1 1
t = 1 1 1 1 1 0 1 1 1 1 1 1 0 0 1 1

Figure 2.1: Neighborhood in 1-D Cellular Automata space.

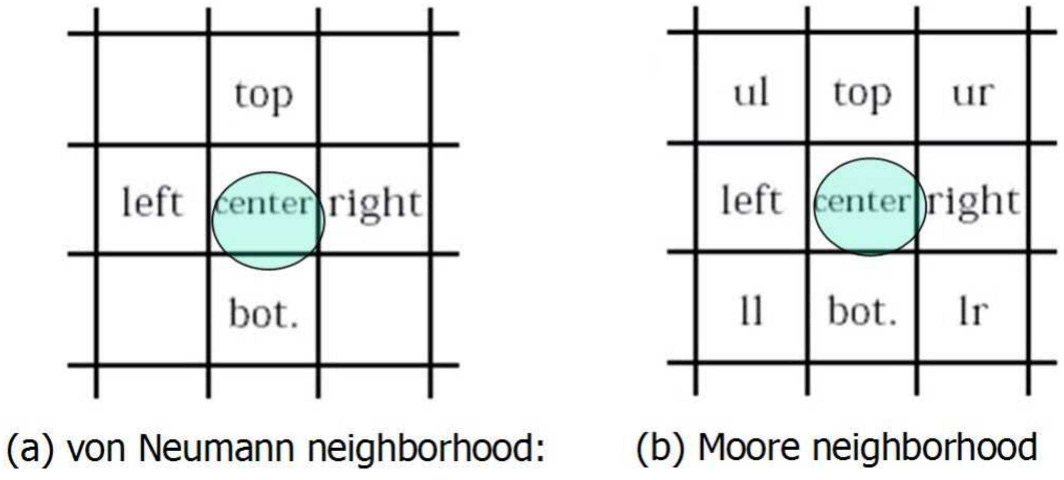


Figure 2.2: Neighborhood in 2-D Cellular Automata space.

In practice, the infinite or finite cellular array (grid) is n-dimensional, where n=1,2,3 is generally used. The identical rule contained in each cell is essentially a finite state machine, usually specified in the form of a rule table (also known as the transition function), with an entry for every possible neighborhood configuration of states. A one-dimensional example rule table and its meaning is illustrated in Figure 2.1. For two-dimensional CAs, von Neumann defined the neighborhood of a cell as the 5 cells consisting of itself along with its four immediate non-diagonal neighbors (Figure 2.2(a)). Moore defined neighborhoods in a different way. The Moore or 9 neighborhood includes both non-diagonal and diagonal immediately adjacent neighbors (Figure 2.2(b)).

The CA rule table is a complete list of transition rules that specify the next state for every possible neighborhood combination. With the von Neumann neighborhood, each rule can be encoded as a string CTRBL$\rightarrow C^{'}$, where each letter specifies respectively the current states of the Center, Top, Right, Bottom, and Left cells, and the next state $\mathrm{C}^{'}$ of the Center cell.

The underlying cellular space can be isotropic or non-isotropic. In a non-isotropic space, one direction may be specially distinguished and this is known to all automata. In an isotropic space, the absolute directions are indistinguishable. In addition, each of the k possible states of a cell is either directionally oriented or directionally non-oriented. If a state is designated as oriented, then a cell which takes the state would designate specific neighbors as being its top, right, bottom, and left neighbors. If a state is designated as non-oriented, then each neighbor to a cell which takes the state has no distinguishable orientation. For example, the quiescent state (0) is always non-oriented. The cell state denoted as $\uparrow$ in von Neumann's work, on the other hand, is directionally oriented and thus permutes to different cell states $\leftarrow$, $\downarrow$, and $\rightarrow$ under successive 90° rotations. It represents one oriented component that can exists in four orientations, with each orientation specifying different neighbor cells as its top, right, bottom, and left neighbors. For a specific CA model, if each of the k possible cell states is non-oriented, then the CA model is said to have strong rotational symmetry. If at least one of the k possible states is directionally oriented, then the CA model has weak rotational symmetry.

Finally, when considering a finite-sized grid, spatially periodic boundary conditions are frequently applied, resulting in a circular grid for the one-dimensional case, and a toroidal one for the two-dimensional case. For instance, in the one-dimensional case, the leftmost cell is viewed as the right neighbor of the rightmost cell, and vice versa. In two-dimensional case, the leftmost cell on each row is viewed as the right neighbor of the rightmost cell, and the topmost cell on each column is viewed as the bottom neighbor of the bottommost cell, etc. One of the states is always designated the quiescent or inactive state (0 or a blank cell). When a quiescent cell has an entirely quiescent neighborhood, a widely accepted convention is that it will remain quiescent at the next time step. This can be represented by the following entry in the rule table: 00000→0.

Despite the simple construction of cellular automata, they are capable of highly complex behavior. The general method to determine the qualitative (average) dynamics of cellular automata models is to run simulations on a computer for various initial global configurations [28, 77]. The analysis of CA dynamics studies the emergent behavior and computational capacity of the system [13, 22]. Borrowing concepts from the field of continuous dynamical systems, Wolfram first classified CA into four broad categories - (i) Class 1: CA which evolve to a homogeneous state; (ii) Class 2: displaying simple separated periodic structures; (iii) Class 3: which exhibit chaotic or pseudo-random behavior, and (iv) Class 4: which yield complex patterns of localized structures and are capable of universal computation [77, 78].

There has been a widespread application of cellular automata in a large number of application domains [79]. Researchers from diverse fields have exploited cellular automata dynamics with problems in their own fields. Cellular automata have been used to model highly complex systems [2, 4, 23, 74, 79], games [6, 14, 18, 19], parallel computing machines [10, 39, 49, 5, 69], physical and biological systems [50, 15, 41], behavioral and social problems [20, 25, 17, 73], VLSI design and testing [40, 64], and pattern recognition [45, 70], etc.

2.2 Embedding Self-Replicating Systems in CA

While a variety of approaches have been taken in the past to studying self-replicating systems, including mechanical [51], biochemical [56], artificial chemistry [90, 89], and replicators with which users can interact [91, 92], a central and enduring approach has focused on embedding abstract self-replicating structures in cellular spaces. In CA, a *structure* can simply be viewed as a configuration of contiguous active cells. Note that such a structure can also enclose empty (quiescent) cells, as long as all the active cells in the structure remain contiguous. The number of active cells in the structure is called its size. An active cell in a structure is also called a *component* of the structure. A structure is called an isolated structure if no active cells which are not in it are adjacent, i.e., no active cells are in the immediate neighborhood of any active cell in the structure.

An isolated structure at time t = 0 is called a *seed.* A structure at time t $\geq$ 1 is called a seed replica, or just *replica*, if the structure inherits all properties of the seed, that is, 1) it has the same configuration as the seed; 2) all of its active cells are contiguous with one another; and 3) it is isolated from other active cells. Note that the replica can be displaced and perhaps rotated relative to the original. Defining a self replicating system embedded in a CA consists of specifying all of the following: 1) a seed and its environment; 2) a rule table; and 3) a time t ≥ 1, such that after the rule table has been applied to the seed and recursively to subsequent configurations, n replicas are constructed in the infinite cellular space, for some positive integer n.

Past work on self-replicating systems in CA is best viewed as having involved two main approaches: universal constructors, and much simpler non-universal structures such as replicating loops, described as follows.

2.2.1 Universal Constructors in CA Space

To embed a hypothetical self-replicating system machine in CA space, von Neumann envisioned that the following characteristics should be present in a self-replicating system model: 1) constructional universal, that is the ability to construct any kind of configuration in the CA space from a given description; self-replication is then only a particular case of universal construction; and 2) computational university, that is the ability to operate as a universal Turing machine, and thus to execute any computational task [27].

To implement his idea, von Neumann developed his theoretical model in CA space with tens of thousands of components in 29-state cells and using a 5-cell neighborhood [48]. His model consists of a configuration of states which can be

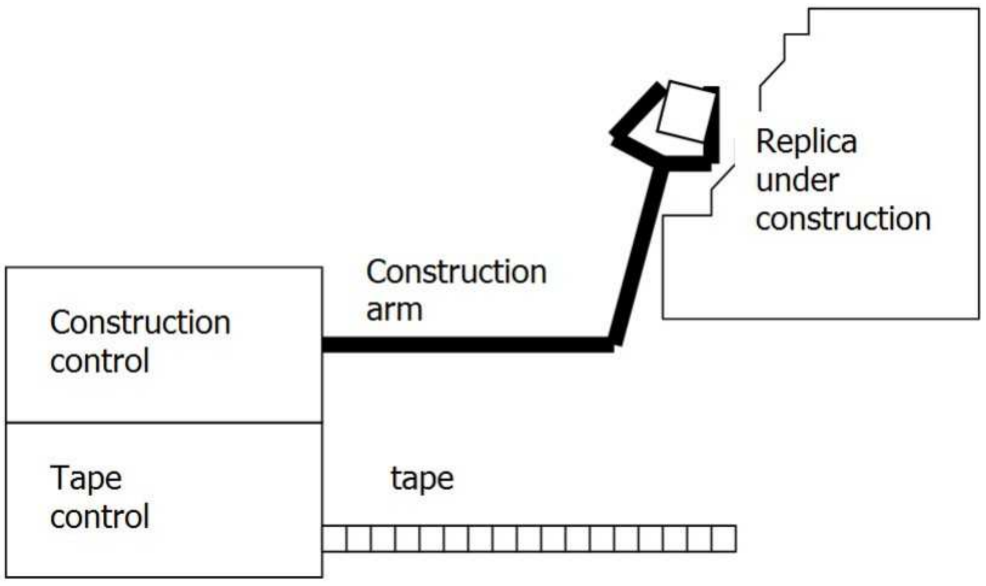


Figure 2.3: von Neumann's theoretical model: a universal constructor

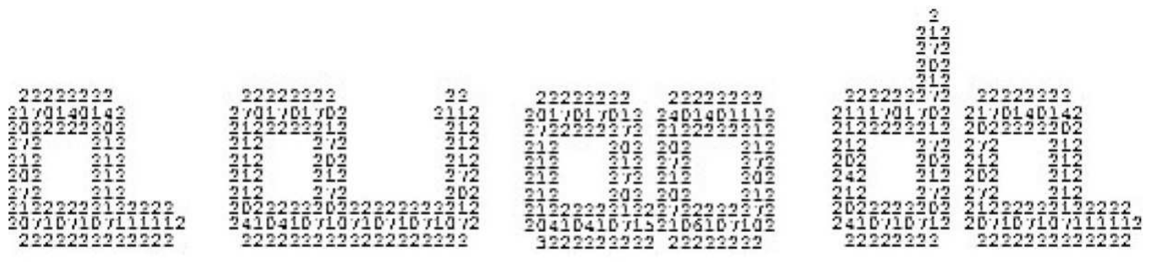
grouped into two functional units: a constructing unit, which constructs the new automaton, and a tape unit, which stores and reads the information needed to construct the automaton, as illustrated in Figure 2.3. The tape unit consists of a "tape" and a tape control. The tape is a linear array of cells that contains the information about the automaton to be constructed. The construction of the automaton is carried out by sending of signals (in the form of propagating cell states) between the tape unit and the construction unit. The construction unit consists of a construction arm and construction control. The construction arm is an array of cells through which cell states to be constructed can be sent from the construction control to the designated area places in the construction area.

Von Neumann's universal constructor model employs a complex transition rule set, with the total number of cells composing the universal constructor estimated to be ranging from 50,000 to 200,000 [65]. In late 1960s Codd demonstrated that if the component or cell states meet certain symmetry requirements, the von Neumann's model could be reduced to a sheathed loop structure embedded in an 8-state, 5-neighbor CA, over 4000 cells in the 2-D CA [11]. Vitanyi described a sexually reproducing cellular automata model and showed that the recombination of the parents' characteristics in the offspring closely conforms to recombination in nature. Similarities and differences with biological systems are discussed [71, 72].

A number of researchers also have considered the implementation of universal constructor. Signorini discussed the implementation of the 29-state transition rule and three organs (pulser, decoder, and periodic pulser) on a SIMD (single-instruction multiple-data) computer [63]. Pesavento provides a closer simulation of von Neumann's model, but self replication is not demonstrated since the tape required to describe the universal constructor is too large to simulate [54]. Beuchat and Haenni implemented a hardware module of a 25-cell pulser using field-programmable gate arrays (FPGAs) [7]. Buckley and Mukherjee described the constructibility of a signal-crossing solution in von Neumann's 29-state CA [8].

2.2.2 Self-Replicating Loops

In 1984, Langton observed that biological self-replicating systems are not capable of universal construction, and concluded that while universal construction may be a sufficient condition for self replication, it is not a necessity. He successfully took a loop structure from Codd's self-replicating model involving only 86 cells, in a 2-dimensional, 8-state, von Neumann neighborhood CA space, and showed that



(a) t=0 (b) t=70 (c) t=127 (d) t=151

Figure 2.4: Langton's structure is a sheathed square loop. First self replication takes 151 time steps.

```
0+-0L-0L
-      -
+      0
0      0
-      0
+      0
0      0
-+0-+0-+0000
```

Figure 2.5: A self-replicating loop in a 2D cellular automata space. Read clockwise starting at the lower right, there are a series of signals (+-) embedded in the loop structure, each indicating that the arm on the bottom right should grow out one step. These are followed by two signals indicating a left turn (L- L-). These signals circulate counterclockwise around the loop, advancing one cell per time step. As they do, copies of the signals pass out the arm at the lower right, causing it to extend and turn so that a second "child" loop is constructed.

it could be modified to replicate [34, 35]. As shown in Figure 2.4, the resulting self-replicating structure is essentially a square loop, with internal and external sheaths, where the data encoding the instructions to construct a duplicated loop circulate counterclockwise. A duplicated loop is formed after 151 time steps.

Langton's self replicating loop is strikingly simple, and can be easily simulated on computers. Further, Byl eliminated the internal sheath of Langton's loop and discovered a smaller loop, which is composed of only 12 cells embedded in a six state cellular space [9]. Reggia et al. removed the external sheath, and constructed a family of yet smaller self-replicating loops, with the smallest comprising only 5 cells, embedded in a 6 state cellular space [57]. An unsheathed loop that is capable of replicating, given an appropriate set of rules, is shown in Figure 2.5. The self-replication process is illustrated in Figure 2.6. Loops without sheaths can be made very small, replicating in less than a dozen steps, as illustrated in Figure 2.7.

2.2.3 Self-Replicating Loops Expanding Problem Solutions

The self-replicating models described up to this point all involved a manual design process and a trend toward producing smaller and simpler structures: from von Neumann's model which has the power of universal computation and universal construction to the simplest self-replicating loops which can do nothing but self replicate [65]. However, very soon it was realized that, a system capable of self replication but not much of anything else would not be very useful. In 1995, Tempesti asked whether it is possible to add additional computation capabilities to the simple

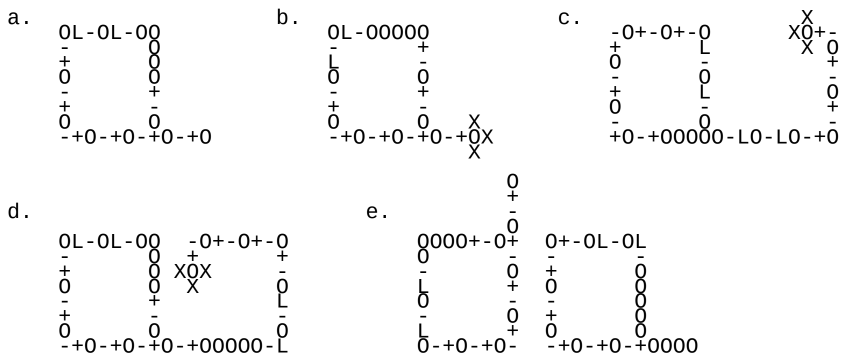


Figure 2.6: Successive states of a self-replicating loop that started at time t=0 as illustrated in Figure 2.5 [57]. The instruction sequence repeatedly circulates counterclockwise around the loop with a copy periodically passing onto the construction arm. At t=3 (**a**) the sequence of instructions has circulated three positions counterclockwise with a copy also entering the construction arm. At t=6 (**b**) the arrival of the first + state at the end of the construction arm produces a growth cap of X's. This growth cap, which is carried forward as the arm subsequently extends to produce the replica, is what makes a sheath unnecessary by enabling directional growth and right-left discrimination. Successive arrival at the growth tip of +'s extends the emerging structure and arrival of paired L's causes left turns, resulting in eventual formation of a new loop. Intermediate states are shown at t=80 (**c**) and t=115 (**d**). By t=150 (**e**) a duplicate of the initial loop has formed and separated (on the right); the original loop (on the left, construction arm having moved to the top) is beginning another cycle of self-directed replication.

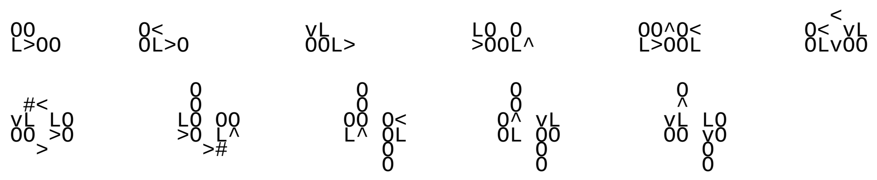


Figure 2.7: A self replicating loop using only five unique components [57]. Shown here are eleven immediately successive configurations. Starting at $t = 0$, the initial state (shown at the upper left) passes through a sequence of steps until at $t = 10$ (last structure shown) an identical but rotated replica has been created.

self replicating loops, and hence attain complex machines that are nevertheless completely realizable. He devised a self-replicating loop which resembles Langton's, but with the added capability of attaching an executable program, that writes out LSL, the acronym of the Logic Systems Laboratory, while it is duplicated and executed in each of the loops [68]. See Figure 2.8. Perrier et al. further demonstrated the capability of constructing a self-replicating loop which could implement any program, written in a simple yet universal language. Their self replicating machine includes three parts-loop, program, and data-all of which are self replicating, followed by the execution of the program on the given data [53]. In the models of Tempesti and Perrier et al., the program embedded in each loop is copied from parent to child unchanged, so that all replicating loops carry out the same program. Chou and Reggia reported a different approach in which each replica receives a distinct partial solution that is modified during replication [10]. Replicas with failed solutions are not allowed to continue replicating while the replicas with promising solutions will further replicate and explore finer solutions. This work demonstrated

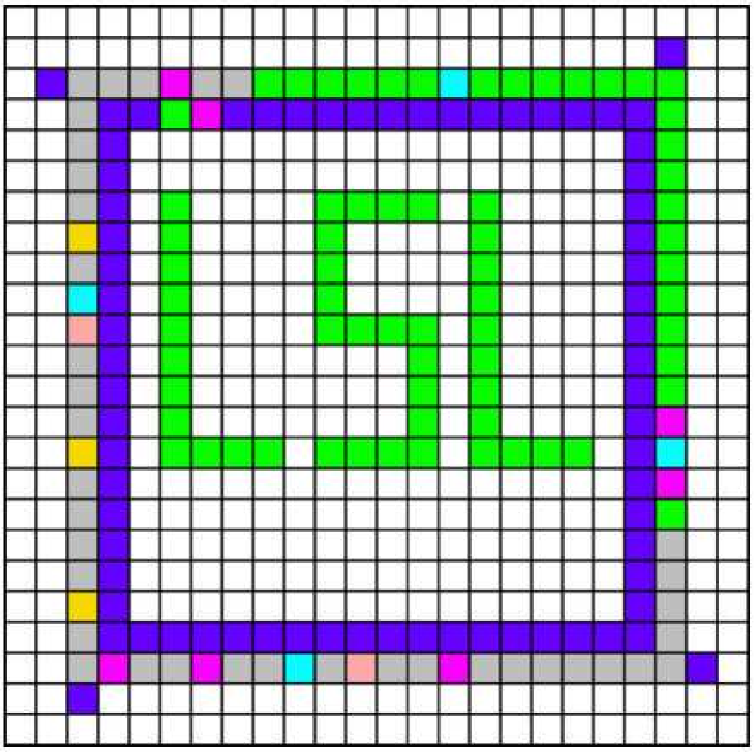


Figure 2.8: Self-replicating loop with secondary capability: it can writes out LSL after embedding a pre-written computer program in the seed structure [68, 65].

how a self-replicating machine could be used as a truly massively parallel machine to solve the NP-complete problem known as SAT. These works demonstrated that the simple manually designed self-replicating loops are capable of providing some limited “secondary” function beyond simple self replication. However, such secondary constructional or computational capability is all implemented as a pre-written executable program, which is attached to the loop itself, that is, at the cost of changing and increasing the complexity of the loop structure itself.

2.2.4 Other Self-Replicating Structures in CA Space

The structures outlined above all share the same restriction inherited from Langton's self-replicating loop: requiring the structure to be a simple, square (or rectangular) shape to enable their replication [61]. The structures differ in size more than complexity. Morita and Imai showed that the replication of simple non-loop structures could be realized with a "reversible" cellular space [43, 44, 88]. A reversible cellular automaton is a special, backward-deterministic type of CA in which every grid configuration of states has at most one predecessor. As a result, they created self replicating structures like worms as well as loops. Sayama further created self-replicating worms that are capable of increasing structure complexity in terms of the length and branching frequency of the worm [61]. Chou and Reggia took a new direction and, demonstrated that self-replicating loop can come about spontaneously and emerge from an initial random configuration of components. Replication occurs in a milieu of free-floating components, and replicas grow or change their sizes over the time, and the transition function is based on a functional division of data fields [57]. The cellular state is divided into four distinct bit fields, thus facilitating the emergence of self-replication. Salzberg et al. further studied the evolutionary dynamics and diversity in a model called Evo-loops [59], a modified version of structurally dissolvable self-replicating loops [60]. Nehaniv implemented the Evo-loop model asynchronously, and studied the evolution and self-replication in asynchronous cellular automata, where each cell can be updated randomly and asynchronously [46].

2.3 Evolutionary Computation, Cellular Automata, and Self-Replication

2.3.1 Evolutionary Computation

Evolutionary computation is a computational search method inspired by Darwinian evolution in biological organisms. A canonical evolutionary computation algorithm is shown in Figure 2.9. It maintains a population of structures, each of which represents a solution to a search problem and evolves according to rules of selection and other operators, that are referred to as genetic operators, such as crossover and mutation. Each individual in the population receives a measure of its fitness in the environment. Reproduction focuses attention on high fitness individuals, thus exploiting the available fitness information. Crossover and mutation perturb those individuals, providing general heuristics for exploration. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful adaptive search mechanisms.

A variety of evolutionary computation algorithms has been proposed, two prominent ones being genetic algorithms (GA) [26, 21, 42, 24] and genetic programming (GP) [3, 12, 29, 30, 31, 32, 33]. In GA, one creates a population of individuals, each represented by a linear chromosome (a collection of genes) appropriate for encoding a solution to the problem one is trying to investigate. A GA chromosome is usually implemented as binary strings, enough to cover all available alternative values. The value of each representation is initially assigned randomly, within the available parameter space. The population is then evaluated to determine how well each individual resolves the problem.

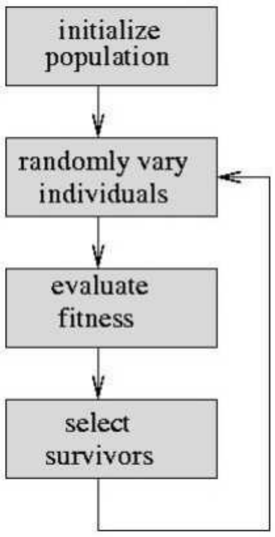
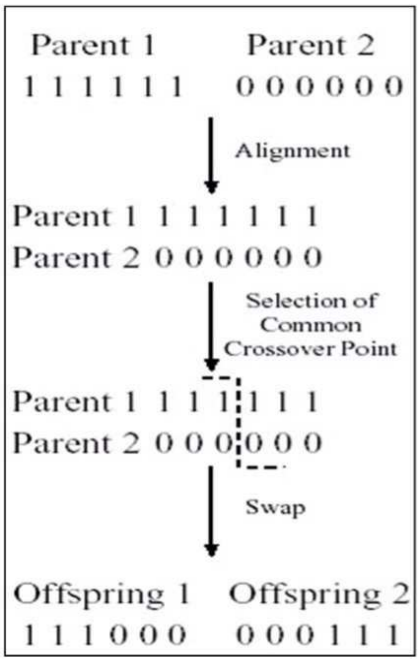


Figure 2.9: A canonical evolutionary computation algorithm.

The best chromosomes in the population then are favored to become the parents for the next generation. The genes of the offspring chromosomes are created by selecting two parents and recombining part of the chromosome of each, so parent gene combinations 1111 mated with 0000 may become two offspring 1110 and 0001, for example. This mimics the crossover or recombination of sexual reproduction in nature and is repeated for further pairs until a desired percentage of the population members is replaced. Random changes to individual bits are then made mimicking the natural role of mutation, at some desired rate. The resultant population is then evaluated as before, and the process is repeated as many times as desired until the required performance level has been achieved (or no further improvement seems possible).

Genetic programming is an extension of GA for evolving computer programs. In GP, each chromosome is an executable program encoded typically as a tree struc-



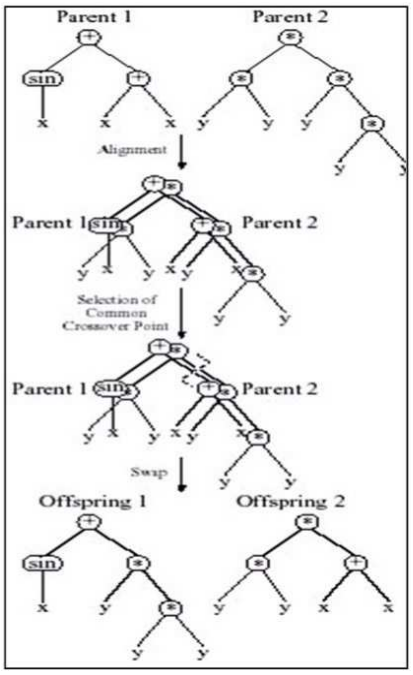


Figure 2.10: GA (left) vs. GP (right).

ture of dynamic shape and size, rather than, as in GA, a fixed-length linear binary string. The GP chromosomes evolve by swapping sub-trees between the parent trees, or random change to an individual tree. The fitness of a chromosome is measured by executing the represented program against a set of training data.

Since John Holland's seminal work in the mid seventies and his well known schema theory [26], schemata are often used to explain why and how GAs work. Schemata are similarity templates representing entire groups of chromosomes. The schema theorem describes how schemata are expected to propagate generation after generation under the effects of selection, crossover, and mutation. Poli et al. derived an improved schema theorem for GP which is a natural counterpart for GP of the schema theorem for GAs, with the introduction of one-point crossover and point mutation operator in GP [55]. The differences between GA and GP are illustrated in Figure 2.10.

2.3.2 Evolutionary Multi-Objective Optimization

Evolutionary computation (both GA and GP) has also been used to solve multi-objective optimization problems, defined by a function f which maps a vector of decision variables, the so-called decision vector, $\bar{x} = [x_1, x_2, ..., x_n]^T$, to a vector of objectives, $\bar{f}(\bar{x}) = [f_1(\bar{x}), f_2(\bar{x}), ..., f_m(\bar{x})]^T$, the so-called objective vector. In other words, the particular set, $x_1^*, x_2^*, ..., x_n^*$, is sought which yields the optimum values of all the objectives, according to function f. In this kind of problem, there is usually a set of solutions better than all other solutions in the search space [81, 85]. This solution set is called the Pareto optimum or non-dominated solutions. Multi-objective evolutionary methods try to find this solution set by using each objective separately, without aggregating them as an unique objective [105]. A number of evolutionary algorithms have been proposed to solve this type of problems, such as the Non-dominated Sorting Genetic Algorithm (NSGA) and NSGA2 by Srinivas and Deb et al. [81, 82], the Strength Pareto Evolutionary Algorithm (SPEA) and SPEA2 by Zitler et al. [83, 84], the Pareto Archived Evolution Strategy (PAES) and the memetic PAES (M-PAES) by Knowles and Corne [85, 86], etc. Although all of these algorithms share a common purpose — searching for a near-optimal, well diversified solution set to a given multi-objective optimization, they differ mainly in the strategies handling fitness assignment, diversity preservation, and elitism. Fitness assignment strategies in multi-objective evolution can be classified as aggregation-based, criterion-based, and Pareto-dominance based. Aggregation-based strategy typically uses weighted-sum aggregation where the weights can be systematically varied during the optimization process. Criterion-based methods switch between the objectives during the selection phase, which decides which member of the population will be copied into the mating pool. The last strategy calculates an individual's fitness based on the basis of Pareto-dominance, taking into account of dominance rank, dominance depth, and dominance count, or any combination, etc. More recent studies focus on incorporating in addition a niching concept in order to address the diversity issue, and recognize the importance of elitism with various implementations and experiments. These variations have received good surveys with reported strengths and weaknesses as well as performance metrics [107, 108, 109, 110, 106, 104]. Especially, SPEA2 has been used in genetic programming to solve a parity problem and it was reported SPEA2 outperforms four other strategies to reduce bloat with regard to both convergence speed and size of the produced programs [111].

2.3.3 Evolution of CA Rules

Given the local concurrent computations in CA, it is generally difficult to program their transition function when the desired computation requires global communication, global integration, and global behavior [1]. Evolutionary computation algorithms, both GA and GP, have been used to automatically evolve cellular automata rules for computation problems in cellular automata space, other than self-replication. For example, various human-written algorithms have appeared for the difficult majority classification task in one-dimensional two-state cellular automata, prior to the introduction of genetic programming to evolve a rule for this task [1]. It was demonstrated that the rules evolved by genetic programming achieved an accuracy resolving the majority classification task exceeding all known human written rules, and that the GP produced rules are qualitatively different from all previous rules in that they employ a larger and more intricate repertoire of domains and particles to represent and communicate information across the cellular space (Figure 2.11). On the other hand, Richards et al outline a method for extracting two-dimensional cellular automaton rules directly from experimental data by employing genetic algorithms (GA), to search efficiently through a space of probabilistic CA rules for a local rule that best reproduces the observed behavior of the data [58].

2.3.4 Evolution of CA Rules for Self-Replication

Inspired by the successful use of evolutionary computation methods to discover novel rule sets for other types of CA problems, as outlined in the previous section, Lohn et al. used a genetic algorithm to evolve rules that would support self-replication [38, 36, 37]. This 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, and in ways quite different from self-replicating structures manually designed by investigators in the past. An example is shown in Figure 2.12. The seed is a structure of 4 oriented components (Figure 2.12, left), and it is not a

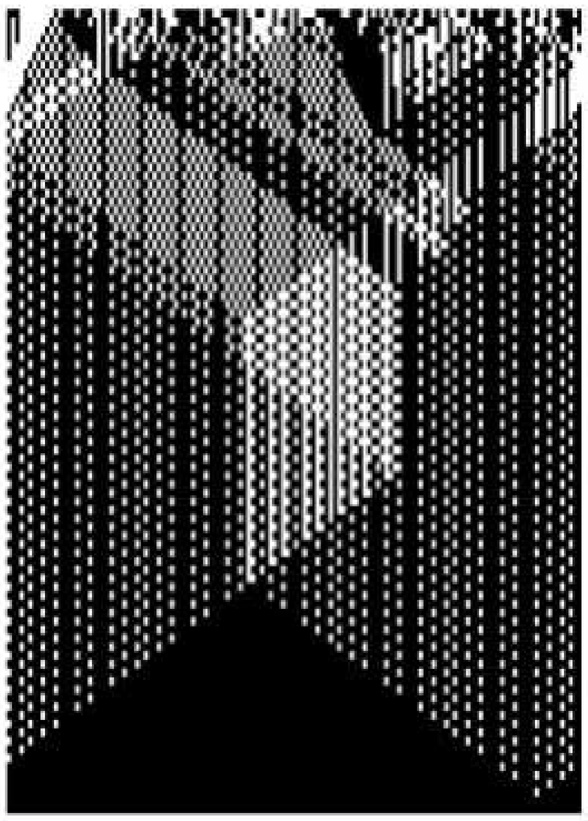


Figure 2.11: The behavior of the best rule evolved by genetic programming on one set of initial states for the majority classification task.

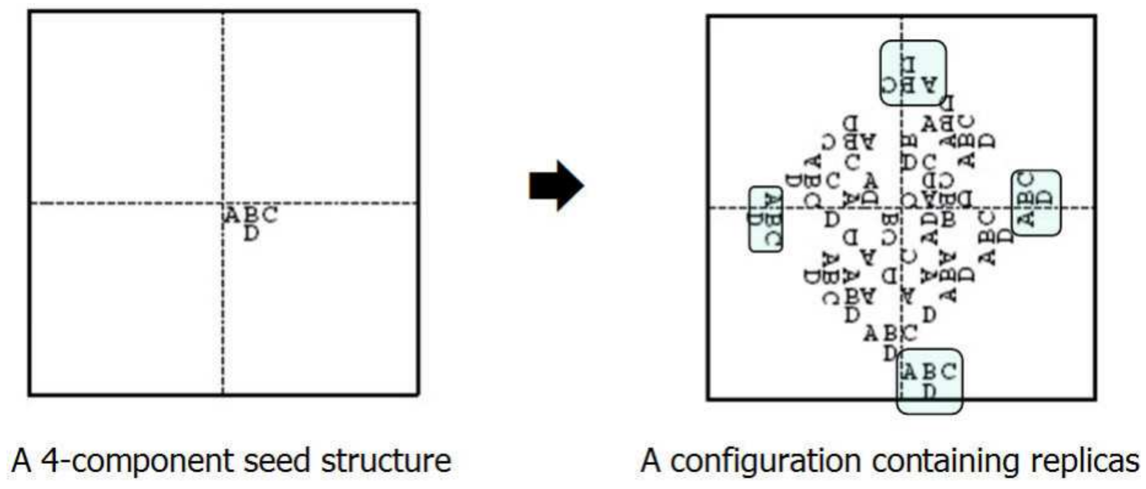

A 4-component seed structure
A configuration containing replicas

Figure 2.12: An example self-replicating structure discovered by genetic algorithm.

loop, worm, or any other previously studied structures. A rule table was generated using a GA. When it is used to guide each cell in the cellular automata to transit its state for certain number of time steps, we can see in Figure 2.12 (right) that multiple instances of the seed structures have formed.

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. As shown in Figure 2.13, the rule table to be evolved include a linear listing of every possible combination of every state required for a given structure. For example, a 4-component structure shown in Figure 2.12(a) would require a rule table containing more than 1.4 million rules (17^5 = 1,419,857). In addition, it creates the problem that when the size of the seed structure moderately increased, the computation cost becomes prohibitive for the rule table to be

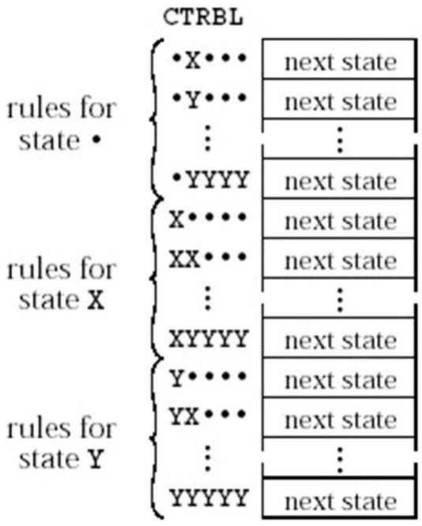


Figure 2.13: A rule table is a linear listing of every possible combination of every state required for a given structure.

effectively evolved, and the yield (fraction of evolutionary runs that successfully discovered self-replication) decreased dramatically.

As a result, it only proved possible to evolve rule sets using GA 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.

Chapter 3

Evolution of Self-Replicating Structures Using Genetic Programming

3.1 Overview

This chapter describes how cellular automata (CA) can be automatically programmed, through the evolution of local rules, to support the self-replication of arbitrary pre-specified structures.

3.2 S-tree Encoding and General Structure Representation

I establish in the following that a tree encoding provides an effective and efficient mechanism for representing arbitrary structures in a CA space that can be used by genetic programming (GP). While this approach is quite general (arbitrary neighborhoods and space dimensions), for concreteness it is developed for two-dimensional CA's and uses the 8-cell Moore neighborhood, which provides sufficient information to tell if a structure is fully isolated from its surroundings. The tree used to represent a seed structure is referred to as its *structure tree* or *S-tree.*

3.2.1 Moore Graph

An arbitrary structure can be viewed as a configuration of active cells in a CA space, with the conditions that the active cells inside the configuration are

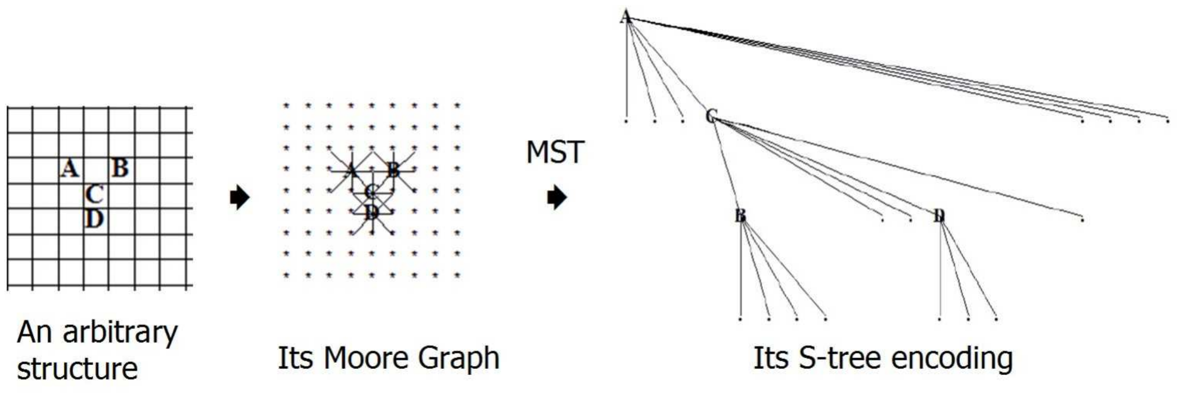


Figure 3.1: An example structure, its Moore graph, and its S-tree graph.

contiguous but isolated from all active cells outside of the configuration. However, a structure can enclose inactive cells in the quiescent state as long as active cells in the structure remain contiguous to each other. It follows that an arbitrary structure can be modeled as a connected, undirected graph, as shown 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). Figure 3.1(left) shows a simple structure in a 2-D CA space, composed of 4 oriented components. The structure is converted into a graph simply by adding an edge between each component and its 8 Moore neighbors, as shown in Figure 3.1(middle). The quiescent cells, shown empty in Figure 3.1(left), are visualized with symbol $*$ in Figure 3.1(middle).

From this example one 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 surroundings); and 3) no other cells are included in the graph. Such a graph is named as the *Moore graph* of the CA structure.

3.2.2 MST and S-tree Generation

Having the Moore graph for an arbitrary structure, one can then further convert the graph into a MST (Minimum Spanning Tree) that is called as the *S-tree*. The essential idea is as follows. After assigning a distance of 1 to every edge on the Moore graph, one can pick an arbitrary component of the structure as the root, and perform a breadth-first-search of the graph. The resultant example tree for the structure shown in Figure 3.1(left) is depicted in Figure 3.1(right). 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 (note 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; and 7) its size (the total number of nodes in the tree), denoted as λ), has an upper limit, which can be calculated from the size of the encoded structure (the total number of components), denoted as γ):

$$\lambda_{max}(\text{S-tree}) = \gamma(\text{encoded structure}) \times 8 + 1 \tag{3.1}$$

With a specific component selected as the root, is the S-tree unique for a given structure? The MST algorithm only guarantees the vertices 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 Figure 3.1(middle)), 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, it is guaranteed that a specific structure always yields the same S-tree. The resulting S-tree is said to be in phase I, II, III, or IV, respectively, if the selected position is top, right, bottom, or left. The S-tree shown in Figure 3.1(right) is in phase I. Figure 3.2(a)-(c) shows the other phases. As clarified later, the concept of S-tree *phase* is important in encoding or detecting structures in rotated orientations.

Note that the Moore graph is only to illustrate the concept and properties of the S-tree encoding. In the actual implementation, however, one can directly generate the S-tree encoding from the seed structure by recursively exploring its components in the order controlled by a first-in-first-out (FIFO) queue. In the beginning, a component in the seed is selected as the root, which is first deposited

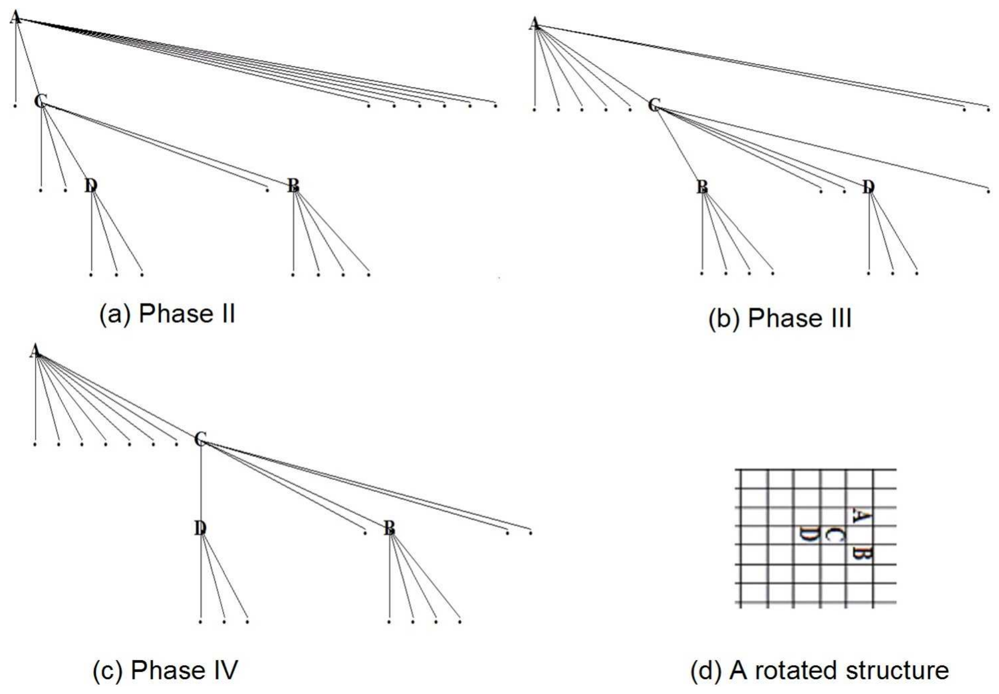


Figure 3.2: S-trees at different phases vs. rotated structures.

to the queue. When each cell is retrieved from the queue and explored, its un-visited active Moore neighbors are deposited to the queue, in the order of the specified phase as defined above. A pseudo C++ code of S-tree generation from the seed structure described above is as follows:

```
void deriveStree (STree stree, unsigned phase) {
    // first, find the first component in the seed structure
    Cell root = CA.findSeedRootComponent();
    // now, insert this root component into the S-tree
    stree.insert(root);
    // next define a FIFO queue to track those components for which
    // we need to explore its Moore neighbors
```

```
  Queue cellQ;
  // the root cell needs first to be explored for its Moore neighbors,
  // so deposit into the FIFO queue
  cellQ.push(root);
  // all what left to be done is to recursively explore Moore
  // neighbors for each component retrieved from the queue
  traverseMooreNeighbors(cellQ, stree, phase);
}

void traverseMooreNeighbors(cellQ, stree, phase) {
  // if there is no more cells in the queue, we are done,
  // exit the recursion by returning
  if(cellQ.size() == 0) return;
  // otherwise, pick next one in the queue, and attempt
  // to traverse its Moore neighbors
  Cell nextComponent = cellQ.front(); cellQ.pop();
  // get an active cell that has 8 Moore neighbors to be traversed
  unsigned nbrIndex;
  Cell nbr;
  // for each of its Moore neighbors
  for(nbrIndex = 1; nbrIndex <= 8; nbrIndex++) {
     // get next Moore neighbor, note the phase value determines
     // the first neighbor to be traversed
```

```
        nbr = nextComponent.getMooreNB(nbrIndex, phase);
        // first, append the traversed cells to the S-tree
        stree.appendChild(nbr);
        if(nbrState != QUIESCENT) {
           // if this recovered cell is not quiescent,
           // also explore its own Moore neighbors,
           // so also put it in the queue
           cellQ.push(nbr);
        }
    }
    // we are done traversing this cell, go to
    // next one in the current FIFO queue
    traverseMooreNeighbors(cellQ, stree, phase);
  }
```

3.2.3 S-Tree Encoding for Arbitrary Structures

Section 3.2.2 effectively yields an unambiguous and efficient way to represent an arbitrary CA structure with a tree structure. This makes it possible to represent arbitrary CA structures with a uniform data structure, and more importantly, enable an evolutionary model or rule learning system to be built and function without having knowledge of any details of the involved structures a priori. When it is desired, an S-tree can be used to fully reconstruct the structure it represents by recursively reconstructing each Moore neighbor from the root component as guided by the S-tree. Further, in a CA space, a structure may be translated, rotated, and/or permuted during processing. The S-tree encoding can handle each of these conditions. First, independent of absolute position, it can be used to detect a structure arbitrarily translated independent of absolute position. Second, the S-trees at 4 different phases are equivalent to the same structure rotated to 4 different orientations. Therefore, by detecting the way the S-tree phase has been shifted, one 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 Figure 3.2(d) is identical to the S-tree at phase I of the structure shown in Figure 3.1(left).

3.3 R-tree Encoding and Rule Set Representation

Just as the seed structure can be represented by an S-tree, the rules that govern state transitions of individual cells can be represented as a *rule tree* or *R-tree*. This section introduces R-tree encoding, which is much more efficient and largely resolves the limitations of an exhaustive (all rule) linear encoding previously used with genetic algorithms evolving self-replicating [38, 36, 37]. Even though both von Neumann or Moore neighborhood can be used, the R-tree formulation considered here is based on the 5-neighborhood (or von Neumann neighborhood). In other

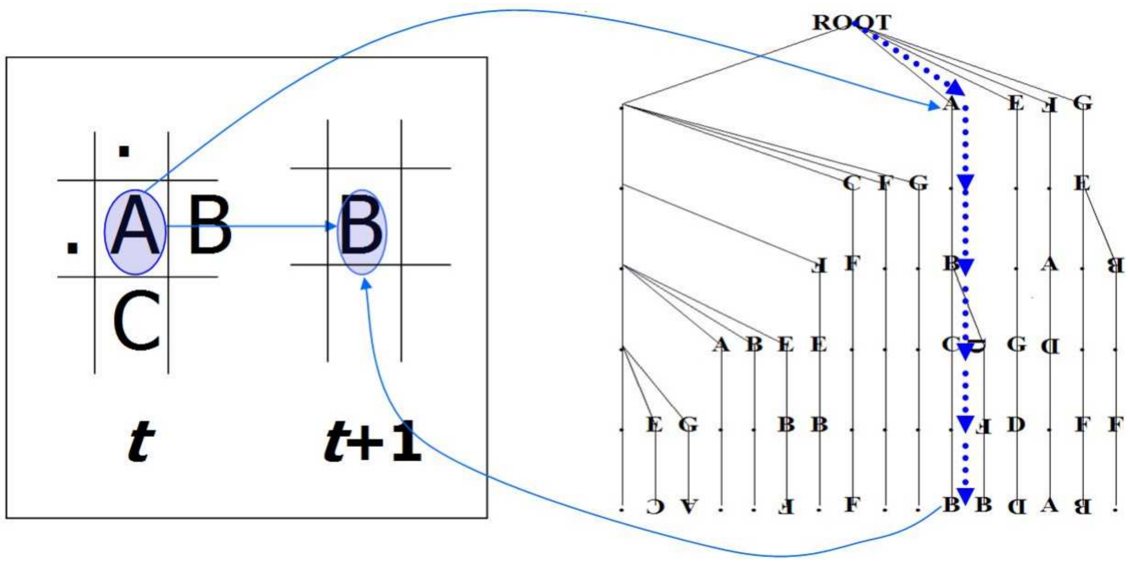
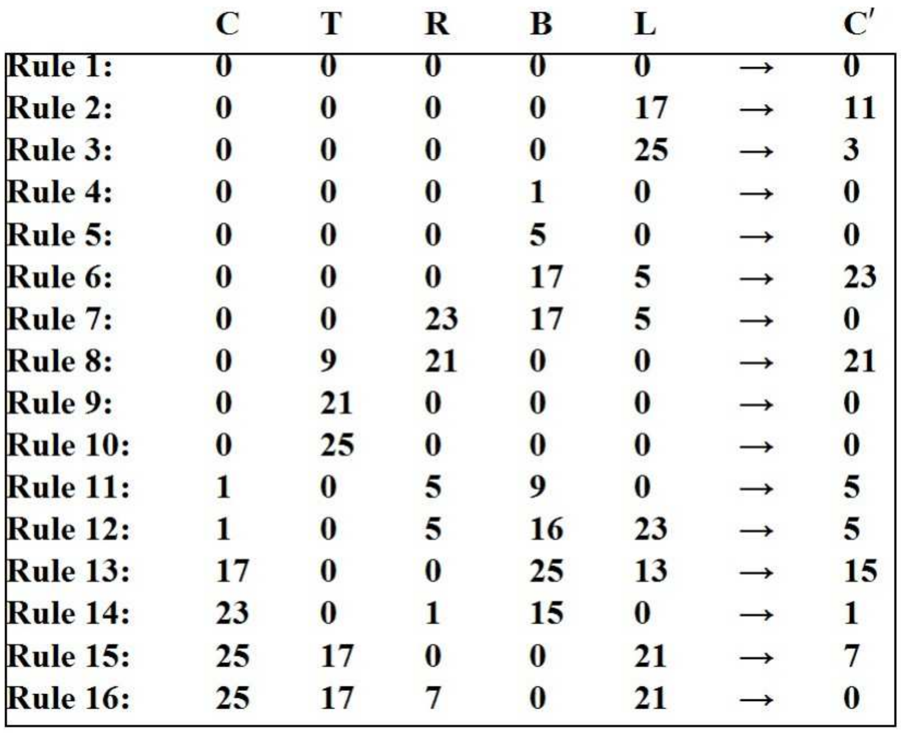


Figure 3.3: An example state-transition using a rule in an R-tree: a cell with state A at T=t becomes state B at T=t+1, after firing a rule in the R-tree based on the states of its von Neumann neighbors as shown.

words, just as with the case of evolving self-replication with a genetic algorithm, the rules evolved are of the form CTRBL → C'.

3.3.1 R-tree 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 (i.e., C in CTRBL → C'). 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 nodes) represents the state of the cells at time t+1 (i.e., state *C'*). Therefore, the R-tree may also be



	C	T	R	B	L		C′
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

Figure 3.4: A list of 16 rules encoded by the sample R-tree shown in Figure 3.3. Note each path from the root node to a leaf represents a rule in the table, while each state shown in the table is its state index number rather than its symbolic representation shown in the tree.

viewed as similar to a decision tree, where each cell can find a unique path to a leaf by selecting each sub-branch based on the states of itself and its von Neumann neighbors. Figure 3.3 (left) shows a sample cell with a state A at T=t transits to state B at T=t+1, as a result of firing a specific rule in the R-tree shown in Figure 3.3 (right), after matching the states of its von Neumann neighbors to the nodes in the rule path highlighted in blue dot line. Note each path from the root to a leaf node corresponds to one rule. In the R-tree, the actual symbol in correct orientation is displayed for each state. Figure 3.4 shows the corresponding table containing 16 rules encoded by the sample R-tree shown in Figure 3.3(right). Note each state in the table is shown with its index number rather than its symbol representation used in the R-tree.

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) taking N_s = number of possible cell states, the root and each node at level 1, 2, 3, and 4 may have a maximum of 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 always exists at most one path that applies to any cell at any time. 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).

3.3.2 R-tree Genetic Operators

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 [55], even though R-trees do not represent conventional sequential programs.

3.3.2.1 R-tree Crossover

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 one simply picks 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 may no longer be height balanced.

This problem can be resolved by restricting R-tree crossover to be a version of homologous one-point crossover, an alternative to the "standard" crossover operator in GP [55]. 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 that node as a potential crossover point. As soon as a mismatch is seen, stop the traversal. Next, pick an edge from the ones marked as potential crossover points, with uniform probability, and swap the sub-trees under that edge between

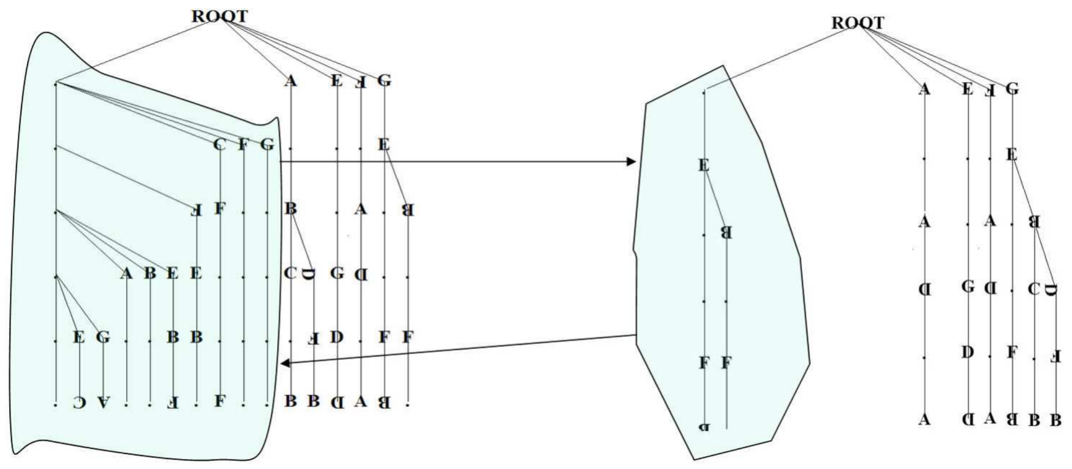


Figure 3.5: One-point homologous crossover between parent R-trees. A crossover point is selected at the *same* location in both parent trees, ensuring that the child trees are valid R-trees. The children R-trees are formed by swapping the shaded sub-trees.

both parent R-trees. An example is shown in Figure 3.5.

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.

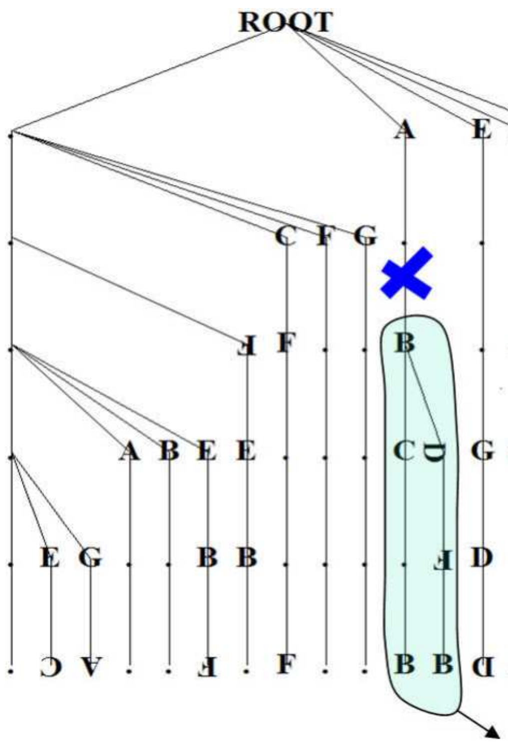


Figure 3.6: The R-tree point mutation simply deletes a sub-tree (here, the one indicated by shading), allowing the CA simulation to fill it in with a new randomly-generated subtree when needed.

3.3.2.2 R-tree Mutation

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. An example is shown in Figure 3.6. The R-tree *pruning operator* is an explicit mutation operator that is applied when the R-tree is used to run the CA to assess the R-trees fitness. The CA monitors the R-tree and marks inactive edges (through which no rules has been activated by any CA cell), and then the entire sub-trees below the inactive edges will be eliminated. This helps to always keep each R-tree as parsimonious as possible. New paths in the R-tree are generated as needed, as explained in the next section.

3.4 Genetic Programming with S-tree and R-tree Encoding

The introduction of S-tree/R-tree encodings and the obtained capability for representing arbitrary structures and cellular rules with a universal and uniform data structure makes it possible to build a genetic programming system that can be used to program the CA to support self-replication. To do that, the seed structure is first encoded with an S-tree. Then, an R-tree population is randomly initialized and starts to evolve as guided by a fitness function. The fitness function evaluates how well intermediate structures produced at evaluation time steps by each R-tree match the S-tree. The R-tree reproduction focuses on high fitness individuals, thus exploiting the available fitness information. This process repeats, from generation to generation, until an R-tree forms which produces a desired number of isolated structures that perfectly match the S-tree encoding, in other words, copies of the seed structure itself.

3.4.1 Problem Formulation

Let r represent an evolving R-tree, s represent an S-tree for a given structure, p represent an infinite cellular space, t represent a time step applying r on s in p. Define function $\delta(r,s,p,t)$ to be, after applying r on s in p recursively for t time steps, the number of instances of isolated structures found in p that match s.

With the definitions above, we can now formally state the problem as follows:

Given an infinite cellular space p and a seed structure s, the goal is to find an R-tree r, which will satisfy the following goal:

there exists a time step T_j, for any positive integer j, such that

$$\delta(r, s, p, t_j) \geq j \tag{3.2}$$

Note this equation requires that, given enough time steps, one shall be able to obtain any desirable number of replicated seed structures.

3.4.2 R-tree Initialization

In the beginning, a population of R-trees is initialized, with each having only one default branch, corresponding to the well accepted default rule that a quiescent cell surrounded by only quiescent cells will remain quiescent in the following time step. Therefore, at GP generation g=0, every R-tree in the population is identical, each only containing one rule. However, as described in next sub-section, as soon as an R-tree is used in the CA simulation, it is expanded and acquires new, randomly generated rules. Naturally, each R-rtree will generally grow into different sizes containing different rules.

Denote the initialized population as L. Note that the population size, denoted as $M = |L|$, is a configurable model parameter, such as $M = 100$.

3.4.3 CA Simulation with R-tree

To evaluate the fitness of each R-tree in the evolving population, first one needs to simulate the R-tree in the CA space and measure how well it expresses the S-tree. This means that one needs to execute the R-tree against the S-tree inside the given cellular space for a certain number of time steps, and measure how well the configurations it generates during the considered time steps match the structure encoded by the S-tree. The range of the time steps during which the simulation is performed is referred the Simulation Time Steps (T^s), such as T^s = (1,2,...,12), and the collection of time steps during which the configurations are considered for fitness evaluation is referred as the Evaluation Time Steps (T^v), such as T^v = (6,7,...,12). Note that T^v can be equal, or a subset of T^s, i.e., $T^v \subseteq T^s$. Obviously in real simulations neither the cellular space p nor the simulation time steps T^s can be infinite.

Before a simulation starts (t=0), every cell in the entire CA space is quiescent, except those cells containing the active components of a single seed structure. At each subsequent time step, $t \in T^s$, each cell $c \in p$ attempts to transit its state c_t to next time step c_{t+1} by identifying and firing a specific rule in the R-tree based on the states of its von Neumann neighbors. If such a rule is not found from the current R-tree, a new rule is inserted into the R-tree with its target state (the leaf node) randomly generated. This operation is referred as *R-tree expansion.* On the other hand, at the end of simulation, those branches in the R-tree which represent a rule or rules that were never fired by any cells at any time step are explicitly removed with the pruner operator (Section 3.3.2.2), in order to prevent R-trees from becoming bloated. This operation is referred as *R-tree pruning.*

On a conventional computation platform, the cellular space is re-initialized to the seed state after an R-tree completes its simulation and fitness evaluation, before another R-tree starts the same process. In a parallel computation platform, each R-tree could be simulated and evaluated concurrently.

It is undesirable for T^s to be either too small or too large. If it is too small, it may be insufficient for the seed structures to reach enough cells to sample and capture the self-replication phenomenon. If it is too big, it may lead to a significant decrease of evolution efficiency due to over-sized R-trees with initial random rules. To address this problem, a strategy allowing T^s to adaptively and gradually increase is designed and introduced as follows. First, introduce a new concept called *hesitation*, which is defined as the current accumulated continuous number of generations during which the fitness of the best R-tree fails to further improve. In other words, at any GP generation, if the current elite R-tree is not better than the elite R-tree in the previous generation, the current hesitation value is increased by 1. Likewise, at any GP generation, if the current elite R-tree is found to be better than the elite R-tree in the previous generation, the hesitation is reset to zero. Typically, during the early phase of an evolutionary searching process, there is very little or low hesitation, as the algorithm finds it relatively easy to climb in the broader area of the fitness landscape. In the later phases, evolutionary searching is more focused on seeking of a peak in the fitness landscape, and it usually takes more and more hesitation before a new improvement can be made. Thus, hesitation can be viewed as the effort at any GP generation the evolutionary algorithm has taken to make a new ascent of the fitness landscape. As described above, hesitation, ξ, can be determined by the following pseudo code:

```
at each GP generation, $g$, {
    if
        currentEliteRTree.fitness $\leq$ previousEliteRTree.fitness
    then
        $\xi = \xi + 1$;
    else
        $\xi = 0$;
}
```

Now with the introduction of the hesitation concept, we can design a control mechanism as follows. First, as configurable items, define three input (model) parameters, namely minimum simulation time step T^s_{min} (such as $T^s_{min} = 1$), maximum simulation time step T^s_{max} (such as $T^s_{max} = 12$), and maximum hesitation ξ_{max} (such as $\xi_{max} = 200$). Then, launch the GP evolution with $T^s = (1,2,..., T^s_{min})$, and monitor the hesitation at each generation. When and only when hesitation, ξ, reaches ξ_{max}, increase T^s with increment of 1, until it reaches T^s_{max}. This is illustrated by the following pseudo code:

```
at each GP generation, {
        if
            $\xi \geq \xi_{max}$
        and
            $T^s < T^s_{max}$
        then
```

```
            $T^s = T^s + 1$;
            $\xi = 0$;
    }
```

This means the simulation will start with a small number of time steps, and the evolution and searching for an optimal R-tree will start from ones with small sizes. Only when GP evolution has made enough effort (by reaching the maximum hesitation) to climb to the currently possible fitness peak with current range of simulation time steps and R-tree sizes, it will adaptively add more simulation time steps and allow the currently optimized R-trees to introduce more rules and gradually expand itself. This keeps the R-trees parsimonious, avoiding GP bloating problem [111], and maintains effective evolutionary searching for optimal rules in a paced fashion.

Further, we can make ξ_{max} itself to be adaptive, allowing it to grow with the fitness. This means we can allow the evolutionary algorithm to take relatively more effort to climb when it gets closer to the global peak. On the other hand, if T^s_{max} is already reached, and ξ still keeps increasing, define a maximum value ξ_{exit}, so that when ξ reaches ξ_{exit}, the GP algorithm will terminate itself and optionally restart, as it has been determined that the GP search has been stuck in pre-mature convergence. Even ξ_{exit} can also be adaptive to the fitness, meaning the search is allowed to take more effort to get out of a local minimum when it gets closer to the peak, instead of dropping immaturely.

3.4.4 R-tree Fitness Assignment Based on S-tree

3.4.4.1 S-tree Probing

The purpose of R-tree simulation is to evaluate its fitness in terms of producing duplicated seed structures. However, since every R-tree in the initial population is randomly generated, it is extremely unlikely any of them will directly lead to self-replication. Thus we cannot just run the R-tree against the seed in CA for a certain number of time steps, and count how many duplicated seed structures it produces. In fact, fitness measuring for self-replication has been found to be a very difficult task [37], mainly because self-replication is a dynamic and complex process. When evolution begins, randomly generated R-trees produce a set of configurations during the given simulation time steps. These configurations very likely also appear as random components and lack any clear patterns (see a sample configuration in Figure 3.7). However, presumably configurations produced from one R-tree may be slightly more promising for leading to self-replication than those from another R-tree, and it is the purpose of fitness assignment to capture such slight differences in the very early stage, before any full or partial seed structure appears. An essential part of any evolutionary algorithm is to reproduce more promising candidates and discard less promising candidates, as indicated by the fitness measures. A past study applying genetic algorithms in self-replication developed a set of fitness functions [37], which provide distilled values based on the numbers of each type of active component, relative positions of the active components, and number of isolated replica, respectively. These early fitness assignment mechanisms were found useful for discovering self-replication of very simple and concrete structures. However, these fitness functions were not satisfactory for adoption into the genetic programming paradigm introduced here, mainly because of the following drawbacks:

1. These are not precise measures exploiting the complete structural information provided by the seed structure itself, but a collection of heuristic measures based on partial information, such as the number of active components in the configuration and their average number of neighbors, etc.

2. The values produced from these fitness function components may provide a good indication of evolutionary progress for very small and simple structures. However, since they only exploit partial information in the seed structure, when the size and complexity of the structure increases, especially when the structure contains repeated components with distinct neighborhoods, the meaning of the results becomes very unclear.

3. More importantly, these heuristic methods assume the specific knowledge of a concrete structure. These methods do not allow an evolutionary system to be established that can be applied to future arbitrary structures without any knowledge of the structure a priori.

Fortunately, the introduction of the S-tree as a universal encoding mechanism of arbitrary structures gives us an unprecedented ability to perform precise fitness assignment for full or partial matching structures at any GP stage. More importantly, such an S-tree based fitness measuring approach can be universally applied

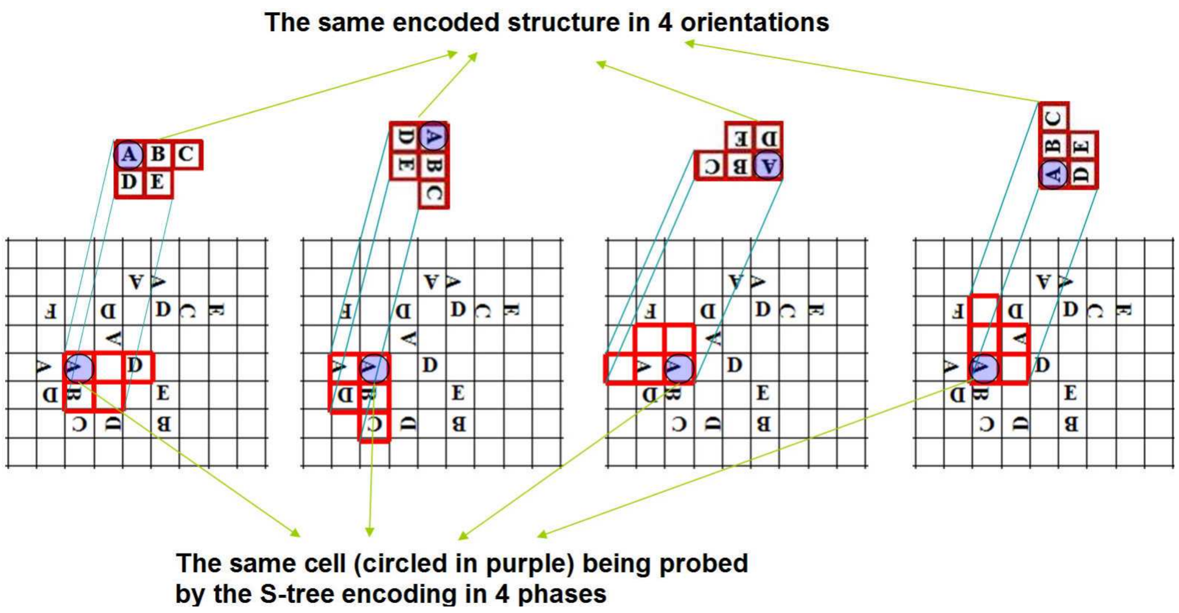

The same encoded structure in 4 orientations
The same cell (circled in purple) being probed by the S-tree encoding in 4 phases

Figure 3.7: The same cell can be probed by the same S-tree in 4 phases, to evaluate a potential match in 4 different orientations. Note such probing can be done with any cell in the configuration.

to arbitrary, future structures without requiring any knowledge about the structure a priori. This is because S-tree encoding has the capability of converting an arbitrary structure into a common tree-based structure, which can be fully exploited by an existing evolutionary system to retrieve the complete structural information provided by the structure itself, such as details about every Moore neighbor of every component in the structure. By retrieving such complete structural information from the S-tree, in an efficient way as allowed by the structure representation as a MST (minimum spanning tree), and comparing them to a given configuration, we can precisely tell how well they are matched. Figure 3.7 illustrates conceptually how this can be done. From a given configuration produced at any time step simulating a candidate R-tree, pick any active cell as the root cell (such as the cell circled in purple in Figure 3.7). Conceptually we can pretend using the S-tree and recovering the entire encoded structure from the root (by recursively recovering every Moore neighborhood of every component from the root, as guided by the S-tree), such as the structure shown on the top in Figure 3.7. Now we can pretend we have the recovered structure overlapping on top of the cells aligned with the root in the configuration and the root in the structure (both circled in purple in Figure 3.7). We can now compare the state of every component in the structure with the state of the overlapped cell in the current configuration, and count the total number of components that matches. If we then divide the result by the total number of components in the structure, we get a precise scalar measure in range of [0, 1], indicating a complete mis-match, partial match, or a perfect match. In a real implementation, we do not need actually to recover the encoded structure as conceptually illustrated above. We only need to, starting from the root cell, recursively traverse the needed number of neighboring cells as guided by the S-tree, and compare the state in a traversed neighbor cell to the state in a corresponding node in the S-tree. Since the S-tree by its nature is a minimum spanning tree (see Section 3.2.2), it allows traversing every component precisely once after traveling the shortest distance. This process is hereafter referred as *S-tree probing.* Thus, S-tree probing is a process that can be used to test every cell in a given configuration, and measure how much a structure can be matched if we align the structure with that cell.

As described in Section 3.2.2, an S-tree can have 4 different phases, corresponding to 4 different orientations of the encoded structure. Likewise, each cell in the current configuration can be probed in 4 different ways with the same S-tree,

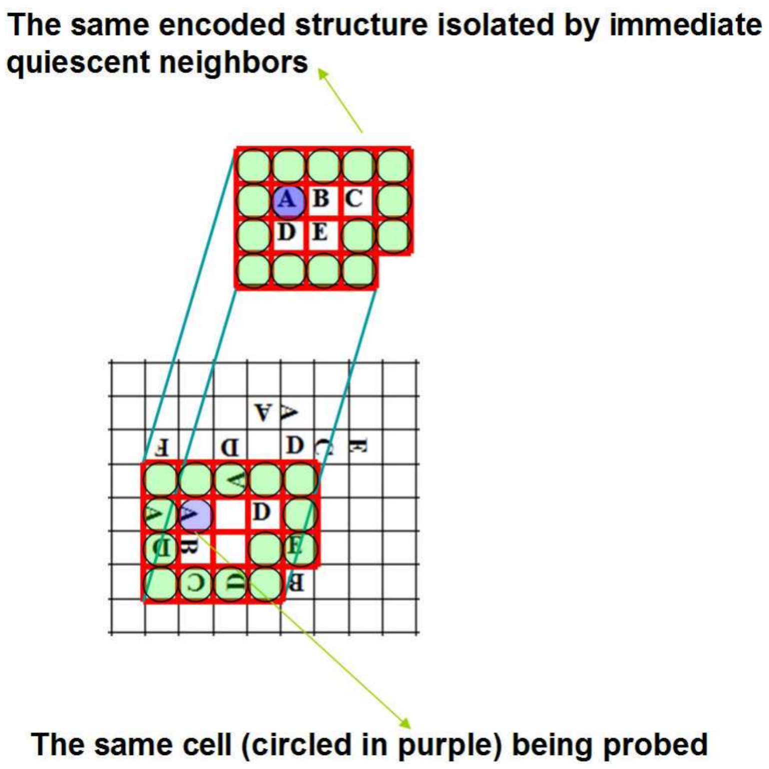

The same encoded structure isolated by immediate quiescent neighbors
A
B
C
D
E
The same cell (circled in purple) being probed

Figure 3.8: An actual S-tree (any phase) contains the immediate quiescent neighbors surrounding the structure. These quiescent cells are also traversed and inspected during the probing (in every orientations), and thus help to detect how much the structure is isolated in the current configuration.

and thus help to detect possible structures located from that cell in 4 orientations. Figure 3.7 shows each of these 4 probes from the same cell. On the other hand, also as described in Section 3.2.2, the actual S-tree contains not only the active components from a structure, but also the immediate quiescent cells surrounding the active cells. Therefore, an actual probing also traverses those surrounding cells in the current configuration, such as those circled in green in Figure 3.8. By inspecting how many of these surrounding cells also match the states of corresponding nodes in the S-tree (which, of course, are all quiescent), we can also precisely measure if the currently probed structure is completely non-isolated, partially isolated, or fully isolated from its surrounding. The same can be said for probes in other orientations, even though Figure 3.8 illustrates only one orientation.

Now let r represent a simulated (evaluated) R-tree, s represent an S-tree for a given structure, p represent an infinite cellular space, $\bar{c} \in p$ represent a root cell being probed, $h \in$ (1,2,3,4) represent the phase of the current probe, t represent a time step applying r on s in p. Define function *κ(r,s,p,t,$\bar{c}$,h)* to be, after applying r on s in p recursively for t time steps, then probing s from $\bar{c}$ in phase h, the number of traversed cells which matches the state of the corresponding node (active or quiescent) as guided by s. Then, we can define a probing function as follows:

$$f_\kappa(r, s, p, t, \bar{c}) = \max_{h \in (1,2,3,4)} \frac{\kappa(r, s, p, t, \bar{c}, h)}{\lambda(s)} \tag{3.3}$$

Recall $\lambda(s)$ represents the size of the S-tree, or the number of nodes in the S-tree, or the number the active components in the encoded structure plus the number of immediate quiescent neighbors. Therefore, when $\kappa(r, s, p, t, \bar{c}, h) = \lambda(s)$, it means every traversed cell perfectly match the encoded structure, and also means the detected structure is also completely isolated (because all of the immediate neighbors are obviously also quiescent if $\kappa(r, s, p, t, \bar{c}, h) = \lambda(s)$). Since $\kappa(r, s, p, t, \bar{c}, h)$ cannot exceed $\lambda(s)$, $f_\kappa(r, s, p, t, \bar{c})$ must get a value in range of [0, 1]. Note Equation 3.3 above indicates $f_\kappa(r, s, p, t, \bar{c})$ returns a best probing value after testing the encoded structure in 4 orientations at location of $\bar{c}$.

Note that any cell in the current configuration can be used as the root cell, $\bar{c}$ in Equation 3.3 above, for probing. When $\bar{c}$ is different, the result of $f_\kappa(r, s, p, t, \bar{c})$ will very likely be different too. Since our goal is to find the best matched structure, we can sequentially perform a probing on every cell, $c \in p$, and then pick the one that yields the highest value, i.e., we can revise Equation 3.3 as follows:

$$f_\kappa(r, s, p, t) = \max_{c \in p} f_\kappa(r, s, p, t, c), \tag{3.4}$$

or,

$$f_\kappa(r, s, p, t) = \max_{c \in p} \Big(\max_{h \in (1,2,3,4)} \frac{\kappa(r, s, p, t, c, h)}{\lambda(s)} \Big). \tag{3.5}$$

Thus, $f_\kappa(r, s, p, t)$ in Equation 3.5 can probe every possible root cell and every possible orientation with the given S-tree, identify the best probe among all these possibilities, and return the best (matching) result. However, our goal is not just to produce one instance of the seed structure at time t. For self-replication to take place at this time step, we need at least 2 instances of the duplicated seed structure.

The value $f_\kappa(r, s, p, t)$ in Equation 3.5 finds the first best match in the current configuration (among all possible locations and orientations), so let's now denote it as $f^1_\kappa(r, s, p, t)$, and see how we can find the second best match in the current configuration $f^2_\kappa(r, s, p, t)$. Before further discussion, let's first recognize the fact that two distinct seed structures cannot occupy a common active cell, while they can perfectly share a common immediate quiescent neighbor which helps to isolate them from each other. Note that whether a cell is active or quiescent is defined with respect to its state in the S-tree (or the aligned seed structure in Figure 3.7 and 3.8), not the state in the probed configuration. It means before we start probing for a second instance of a matching structure, we need to first "accept" the previously identified best probe and mark all of the active cells traversed by the accepted probe as "UNAVAILABLE", so that these active cells, which are already counted by a previously accepted probe, will not be counted again and contribute incorrectly to fitness measures of subsequent probes.

Hence, to clarify the concepts, a *probe* means a test on a certain location and orientation to see how well a matched structure can be found in that way; *probing* means a process in which many probes are tested at various locations and orientations; *accepting a probe* means an action in which one of the probes are identified as producing the best result, with its result recorded and the active cells it traversed marked "UNAVAILABLE"; and *an accepted probe* means one of probes with which the acceptance action has taken place.

In summary, let $\breve{p}_1$ represent the set of cells marked as "UNAVAILABLE" by a previously accepted probe $f^1_\kappa(r, s, p, t)$, we can now find and accept two best probes from the configuration at time t, as follows:

$$f_{\kappa}^{1}(r,s,p,t) = \max_{c\in p}\big(\max_{h\in(1,2,3,4)}\big(\frac{\kappa(r,s,p,t,c,h)}{\lambda(s)}\big)\big) \tag{3.6}$$

$$f_{\kappa}^{2}(r,s,p,t) = \max_{c\in p,\ c\neg\in\check{p}_1}\big(\max_{h\in(1,2,3,4)}\big(\frac{\kappa(r,s,p,t,c,h)}{\lambda(s)}\big)\big) \tag{3.7}$$

If we use these probing functions at one time step for each R-tree, and compare the candidate R-trees in terms of their accepted probes, we can effectively identify which R-trees are more likely to generate a pair of perfectly matching structures, i.e., self-replication. However, our goal is to allow self-replication to carry on sustainably. Given more time steps, the initial seed structure can reach more and more active cells, eventually making enough rooms for more than two duplicated structures. Thus, ultimately we will want to accept more probes so that we can reward those R-trees which are more likely to generate a maximum number of replicas. But the question is, how do we know how many probes we shall accept at each time step? One might ask, why don't we accept as many probes as possible? If we accept too many probes in a given time, it may have the effect of promoting the trend of forming many partially matching structures, but few would have enough room and potential to grow into full replicas, with the constraint of limited time and space, and ultimately degrade the performance of the evolution. This problem is hereafter referred as *over-probing*. Over-probing can adversely impact the evolution performance due to the following reasons: 1) The seed structure can reach more cells without exceeding light speed. This means that within certain number of time steps, there will not be sufficient active cells for the excessive number of replicas to appear. Normally, a bigger seed would need more time steps to reach enough space for the first pair of replicas. 2) During the early phase of GP evolution, the fitness level of most R-trees in the population is still very low, so accepting many probes pre-maturely means distributing the focus on a much larger searching space, causing R-trees to become bloated, and hindering the population from converging.

Thus, we need to develop an appropriate approach to determine the number of acceptable probes adaptively, in order to avoid the over-probing problem. The following section describes such an approach.

3.4.4.2 Determining the Number of Acceptable Probes

As described in Section 3.4.3, R-tree simulation starts with a small number of time steps, which adaptively and gradually increase. When evolution starts, each R-tree is automatically allowed to accept two probes at each time step. After some R-trees become more and more successful at generating 2 perfect probes (i.e., perfectly matched replicas), and also consequently simulations reach a higher number of time steps, we can allow these R-trees to incrementally increase the number of acceptable probes (and so become more aggressive in working on additional replicas). Denote the number of acceptable probes at evaluation time $t \in \mathcal{T}^v$ for R-tree r as $\pi_\delta^t(\mathrm{r})$, then summarize a strategy as follows:

Probing Strategy 3.1: An evolving R-tree starts with an initial goal that probes and finds the starting evaluation time step when enough cells are reached to form the first pair of duplicated structures, and that only when an R-tree succeeds in forming the first pair can it start to become more aggressive in attempting to form more seed structures. The better it performs in previous evaluation time steps, the faster it can be allowed to accept more probes.

This strategy says an evolving R-tree can start with a basic goal, programing itself to find a minimum CA space just to produce 2 isolated seed structures. Not until this is achieved can it accept more probes. On the other hand, once the current goal is achieved, it can gradually raise its goal by adaptively adjusting the number of acceptable probes (π_δ^t) at a controlled and adaptive pace. This strategy can be implemented as in the following pseudo-code:

```
for R-tree $r$, seed S-tree $s$, and cellular space $p$ {
  $\pi_\delta^0 = 2$;
  at each time step $t > 0$ {
      if ($\delta(r, s, p, t-1) = \pi_\delta^{t-1}$ ) {
          $\pi_\delta^t = \pi_\delta^{t-1} + 1$;
      }
  }
```

This pseudo-code says that, at t=0, the number of acceptable probes is 2. At any subsequent time step, if each of these accepted probes finds a perfectly matching structure (e.g., initially, finding 2 isolated seed replicas), the number of acceptable probes can be increased by 1. Otherwise, it remain the same. Whenever the number is adjusted, the new number has to be realized, before it can be adjusted again.

3.4.4.3 Overall Fitness Function

Based on Sections 3.4.4.1 and 3.4.4.2, an R-tree r at evaluation time t is allowed to accept $\pi_{\delta}^{t}(r)$ probes. Each accepted probe identifies a best probe from the cells not yet marked as "UNAVAILABLE" by previously accepted probes. Hence, we can write a fitness function at time t as follows:

$$f(r,t) = \sum_{n=1}^{n=\pi_{\delta}^{t}(r)} f_{\kappa}^{n}(r,s,p,t), \tag{3.8}$$

or, according to Equation 3.5 and 3.7,

$$f(r,t) = \sum_{n=1}^{n=\pi_{\delta}^{t}(r)} (\max_{c \in p,\ c \neg\in \bigcup_{m=1}^{m=n-1} \breve{p}_m} (\max_{h \in (1,2,3,4)} (\frac{\kappa(r,s,p,t,c,h)}{\lambda(s)}))). \tag{3.9}$$

Then, considering all evaluation time steps, $t \in \mathcal{T}^v$, we get the overall fitness function for R-tree r:

$$f(r) = \sum_{t \in \mathcal{T}^v} (\sum_{n=1}^{n=\pi_{\delta}^{t}(r)} (\max_{c \in p,\ c \neg\in \bigcup_{m=1}^{m=n-1} \breve{p}_m} (\max_{h \in (1,2,3,4)} (\frac{\kappa(r,s,p,t,c,h)}{\lambda(s)})))). \tag{3.10}$$

Here, Equation 3.10 indicates the overall fitness measure for a candidate R-tree at a given GP generation is its accumulated result of every accepted probe at every evaluation time step. Every accepted probe finds a best probe among tested probes at every location and every orientation, given the remaining cells that are not marked as "UNAVAILABLE". Note the number of evaluated time steps at a given generation, $\mathcal{T}^v$, is common among all R-trees in the population, however, at any specific evaluation time step, the allowable number of accepted probes varies from R-tree to R-tree. Obviously an R-tree can gain higher fitness value by either earning higher number of accepted probes, or better results produced from individual accepted probes, or both.

3.4.5 R-tree Fitness Sharing

It is possible that at the end of R-tree simulation and fitness evaluation, certain R-trees are found to be identical to each other. This is un-desirable because it may cause duplicated R-trees to be picked in the tournament selection, increase the chance of pre-mature convergence, and reduce the probability of other diversified R-trees to be selected. *Fitness Sharing* is adopted to address this problem [21]. The idea is to punish an R-tree by adjusting its fitness according to how many existing R-trees are identical to it. For example, if an R-tree is found identical to two of the previously evaluated R-trees, its fitness value will be adjusted by 4÷(4+n)= 4÷(4+2) = 0.66, or 66%, of its original fitness value.

3.4.6 R-tree Tournament Selection

At each GP generation, each R-tree $\bar{r}$ in the population is simulated in T^s (as described in Section 3.4.3) and evaluated in T^v (as described in Section 3.4.4), resulting in a scalar fitness measure in range of [0, 1]. After fitness sharing adjustment (as described in Section 3.4.5), the final fitness values $f_\delta(\bar{r})$ of each R-tree essentially makes the entire population fully ordered. Accordingly, a tournament selection algorithm can be designed [112]. As a configurable model parameter, a tournament size, ζ, is given. To select an R-tree from the current population and copy it into the mating pool, ζ R-trees are randomly picked, but only the one of highest f_δ wins the tournament and get selected. Hence, the tournament selection algorithm gives higher probability to those better R-trees to enter the mating pool, and hopefully produce better offspring in the evolution. Note that, if ζ is too big, the population will be under too much selection pressure, which may lead to pre-mature convergence. If ζ is too small, such as 1, evolution will act like random searching. So, ζ needs to be appropriately selected. For example, for a population size of 100, typically $\zeta = 2$ is selected.

3.4.7 R-tree Evolution

After a pair of parent R-trees are selected from the current R-tree population using the tournament selection algorithm (as described in Section 3.4.6), this pair of parents can be directly copied into the mating pool, or first execute an R-tree crossover operation (detailed in Section 3.3.2.1), controlled by the R-tree crossover probability b_c^r, another configurable model parameter. This is done as shown in the following pseudo code:

```
for each pair of R-tree parents {
    //get a random value in range of [0,1]
    newRandomValue = RANDOM.generate(0,1);
    if
        newRandomValue $\leq b_c^r$
    then
        pair.performCrossover();
    else
        pair.performCopyToMatingPool();
}
```

Even though it is perfectly fine for a common R-tree to be selected in multiple crossover operations, it does not make sense for the same parent R-tree to be directly copied into the mating pool multiple times. Therefore, if such a situation happens, normally tournament selection will be repeated until a different parent R-tree is yielded.

After the crossover operation is completed as described above, every R-tree in the mating pool is subject to a mutation operation (detailed in Section 3.3.2.2), controlled by the R-tree mutation probability b_m^r, another configurable model parameter. This is done as shown in the following pseudo code:

```
for each *rtree* in the mating pool {
    //get a random value in range of [0,1]
    newRandomValue = RANDOM.generate(0,1);
    if
        newRandomValue $\leq b_m^r$
    then
        *rtree*.performMutation(); //execute the mutation operator
    else
        // do nothing;
}
```

After the evolution operations are completed, the R-trees in the mating pool are copied and become the population of a new generation.

3.4.8 R-tree Elitism

Elitism means that the very best or "elite" R-tree cannot be expelled from the population in favor of worse individuals. This also means that in an elitistic evolutionary system the fitness of the elite R-tree at the current generation has to be at least as good as, if not better, than the elite R-tree in the previous generation. However, the tournament selection and R-tree evolution operations described above do not prevent an elite R-tree from being lost. The problem is addressed as follows. After fitness evaluation and fitness sharing after performed, the current elite R-tree is compared to the stored elite R-tree from the previous generation. If the current elite R-tree is not worse than the previous one, store the new elite and nothing else is to be done. Otherwise, the previous elite R-tree will be re-inserted back into the new population and replace a randomly chosen member.

3.4.9 The Resulting Overall GP Model

The schematic view of the resulting S-tree/R-tree based GP model toward self-replication is illustrated in Figure 3.9. First, S-tree s is derived from the pre-specified seed structure (Section 3.2.2). Then, an R-tree population of size M is initialized (Section 3.4.2). Each R-tree is simulated in the given cellular space within the current T^s, while each R-tree may potentially expand or prune itself as needed (Section 3.4.3). Next, based on the simulation results, fitness is measured for each R-tree within the current T^v (Section 3.4.4). If the desired fitness level is reached, the algorithm has produced the best R-tree and stops. Otherwise, the fitness values

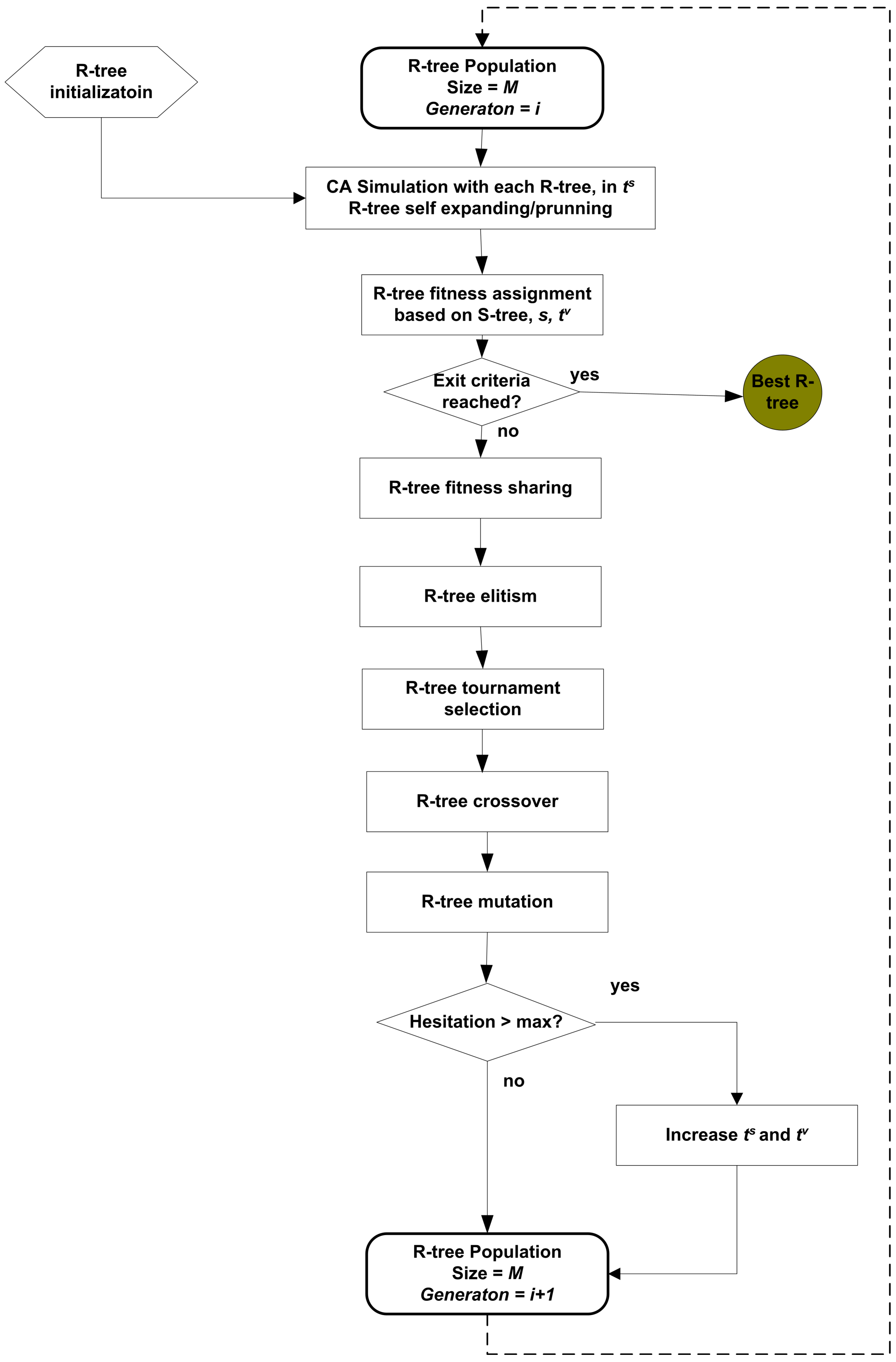

R-tree initializatoin
R-tree Population Size = *M* *Generaton = i*
CA Simulation with each R-tree, in t^s R-tree self expanding/prunning
R-tree fitness assignment based on S-tree, *s*, t^v
Exit criteria reached?
yes
Best R-tree
no
R-tree fitness sharing
R-tree elitism
R-tree tournament selection
R-tree crossover
R-tree mutation
Hesitation > max?
yes
no
Increase t^s and t^v
R-tree Population Size = *M* *Generaton = i+1*

Figure 3.9: A schematic view of the S-tree/R-tree based GP model toward self-replication

are adjusted due to fitness sharing (Section 3.4.5). The R-tree elitism is performed so that it is ensured the elite R-rtree in the new population will be at least as good as before (Section 3.4.8). However, if the elite R-rtree did not improve in the current generation, the hesitation is increased. The entire population is fully ordered based on the final fitness value of each R-tree. Tournament selection is performed and *M/2* pairs of parents are selected (Section 3.4.6). Each pair may perform an R-tree crossover before entering the mating pool, as controlled by b^r_c (Section 3.3.2.1 and 3.4.7). Each R-tree in the mating pool may be further mutated, as controlled by b^r_m (Section 3.3.2.2 and 3.4.7). If current hesitation has exceeded the specified ξ_{max}, T^s and T^v is incrementally increased (Section 3.4.3). Then, the R-tree population enter a new GP generation, and the same process repeats as above.

3.5 Initial Experimental Results

The model described above has been tested in a number of experiments [101, 102], and two examples are presented here. Typically, model parameters like the following are chosen: Population Size = 100, R-tree Mutate Probability = 0.45, R-tree Crossover Probability = 0.85, R-tree GP Tournament Size = 2, and Max Hesitation = 200. Success is achieved 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 high performance computer system has 4 components [38]. Figure 3.10(t=0) shows one of the seed structures, consisting of 7 oriented components (29 states), for which the new approach using GP finds

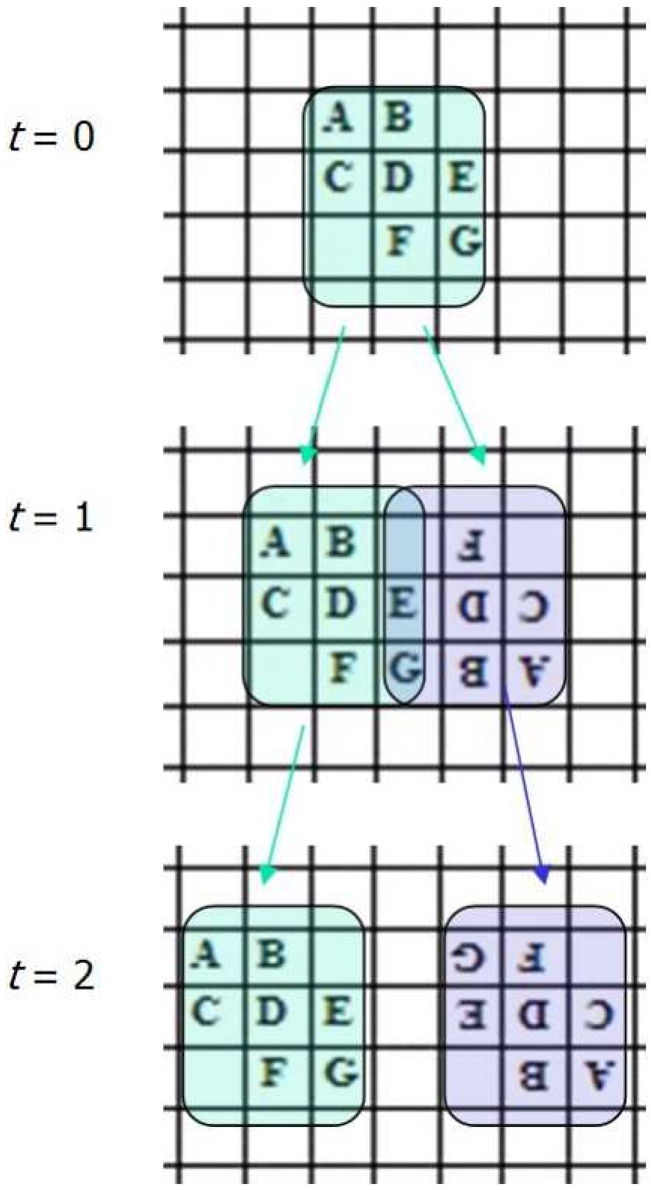


Figure 3.10: Self-replication, example 3.1: the seed, a 7-oriented-component structure, is programed to self replicate in only 2 time steps. In subsequent time steps, each replica attempts to repeat the same action, when collisions result. However, eventually, when enough space is reached, more replicas can be isolated from each other. More details of such a process is illustrated in the next example, shown in Figure 3.12.

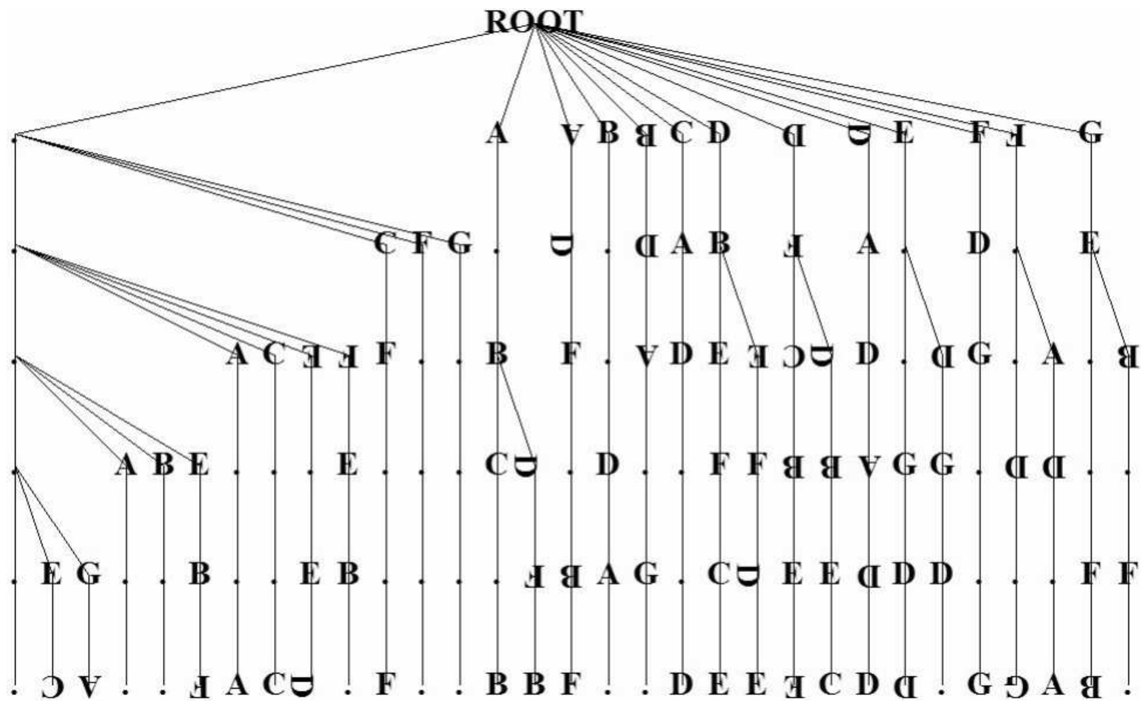


Figure 3.11: Self-replication, example 3.1: the R-tree automatically evolved from the given seed structure with the method established in this chapter. This example R-tree uses the von Neumann neighborhood.

a rule set that allowed the structure to self-replicate. With the resultant R-tree, shown in Figure 3.11, at time t=1 (Figure 3.10), the structure starts splitting (the original seed structure translates to the left while a rotated replica is being born to the right). At time t=2 (Figure 3.10), 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 not been reported before.

Example 3.1 illustrates using the von Neumann neighborhood in the R-tree. All other examples hereafter use the Moore neighborhood instead. To make it easy to visualize the produced structures at each evaluation time step, color codes are used in these figures. A non-isolated seed structure is marked in yellow and isolated seed structure in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step. For example, a second structure of 6 oriented components (25 states) is shown in Figure 3.12 (t=0), and the cells covered by this seed are highlighted at every subsequent time step. As illustrated in Figure 3.12, at t=1, the seed expands to the right, top, and bottom, at light speed (1 cell/each time step), but not to the left. Two contiguous seed replicas are formed. At t=2, the replicas move apart and get isolated. At t=3, each of these isolated replicas repeats the same self-replication action done by the initial seed, and forms 2 sets of contiguous replicas. At t=4, each set attempts to split like at t=2, but the middle ones run into collision and both get nullified. Each of the two replicas found at t=4, which are now more apart than at t=2, repeats the same self-replication action and successfully forms 4 isolated

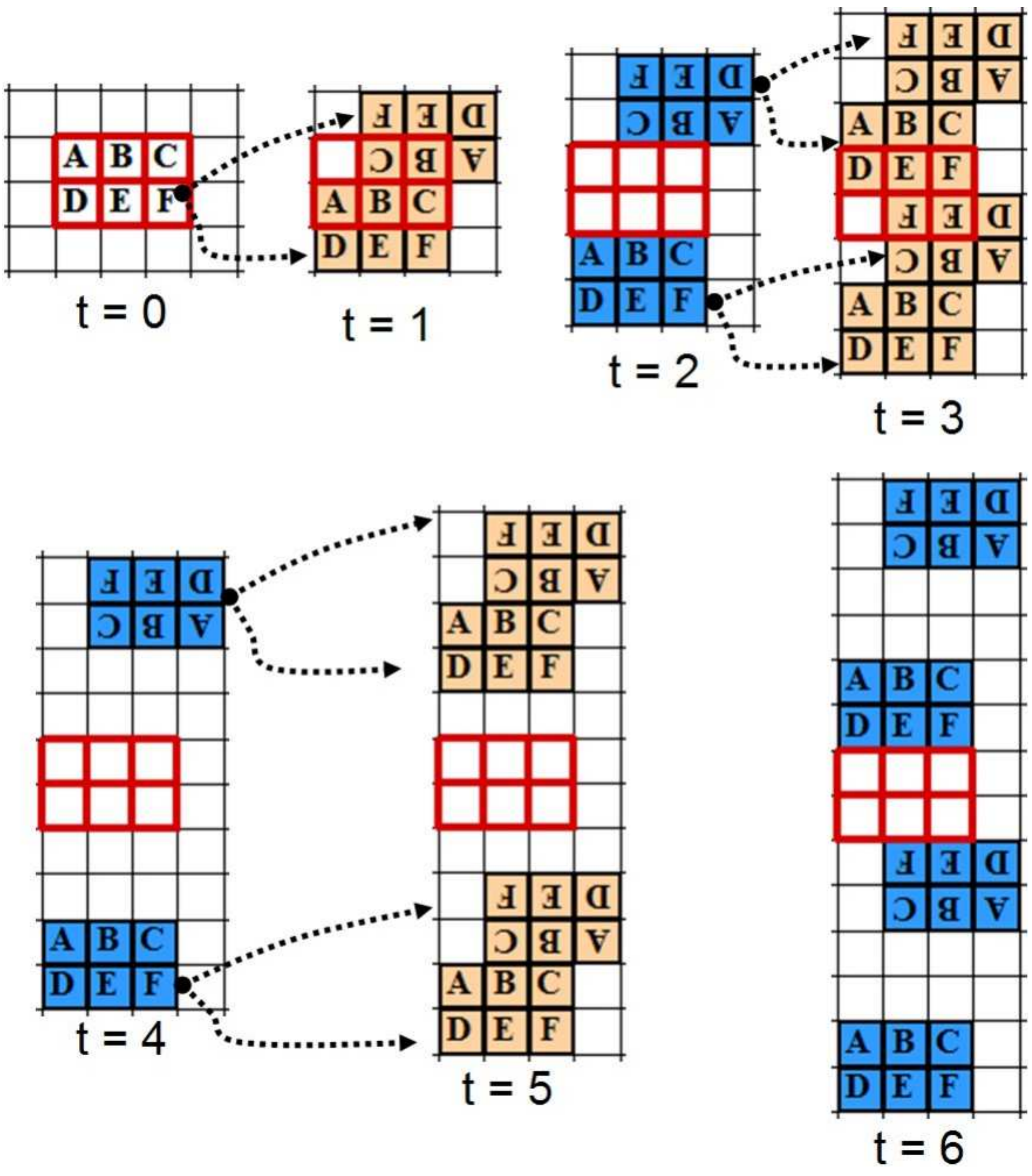


Figure 3.12: Self-replication, example 3.2-1: executing the automatically programmed R-tree against a 6-oriented-component seed structure, from t=0 to t=6. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step.

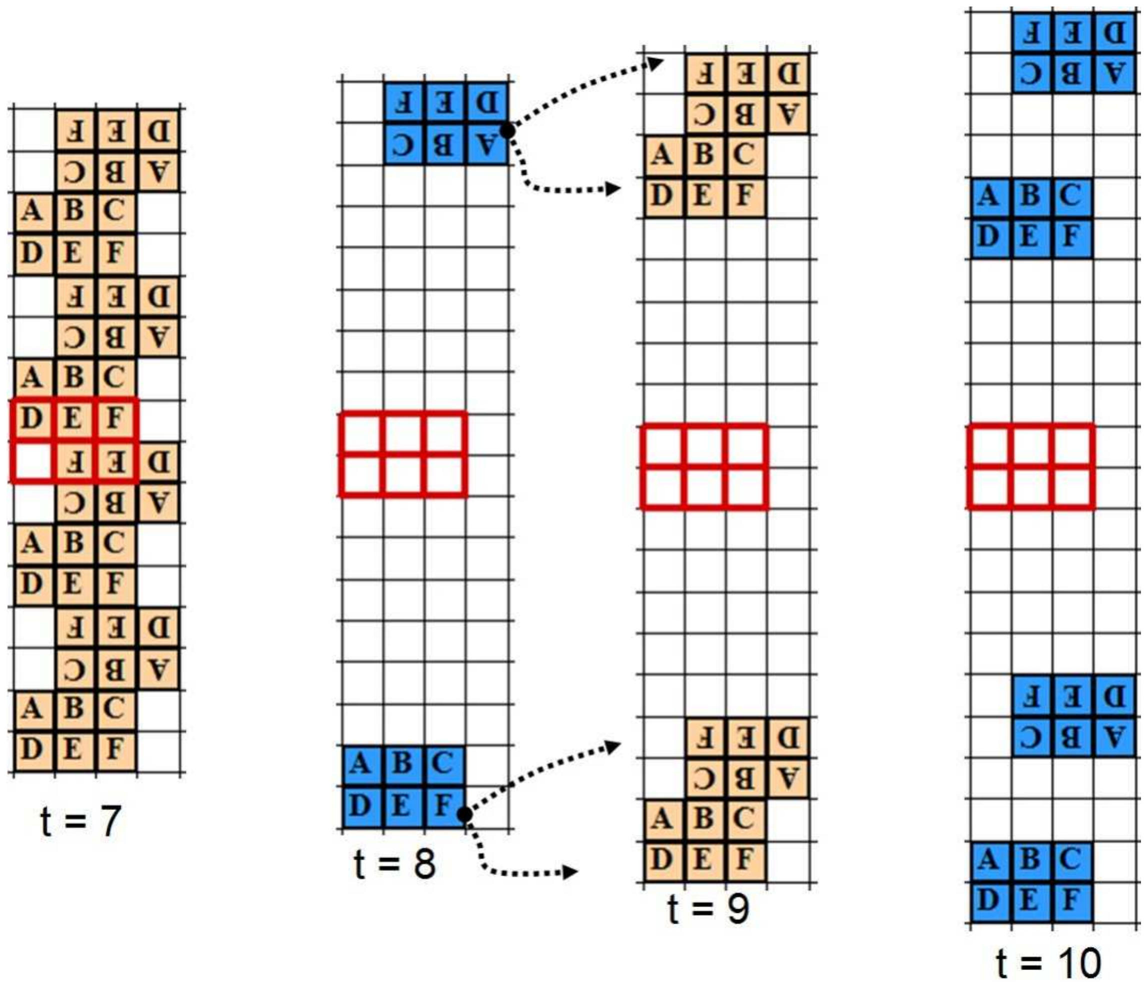


Figure 3.13: Continuation of Figure 3.12 from t=7 to t=10.

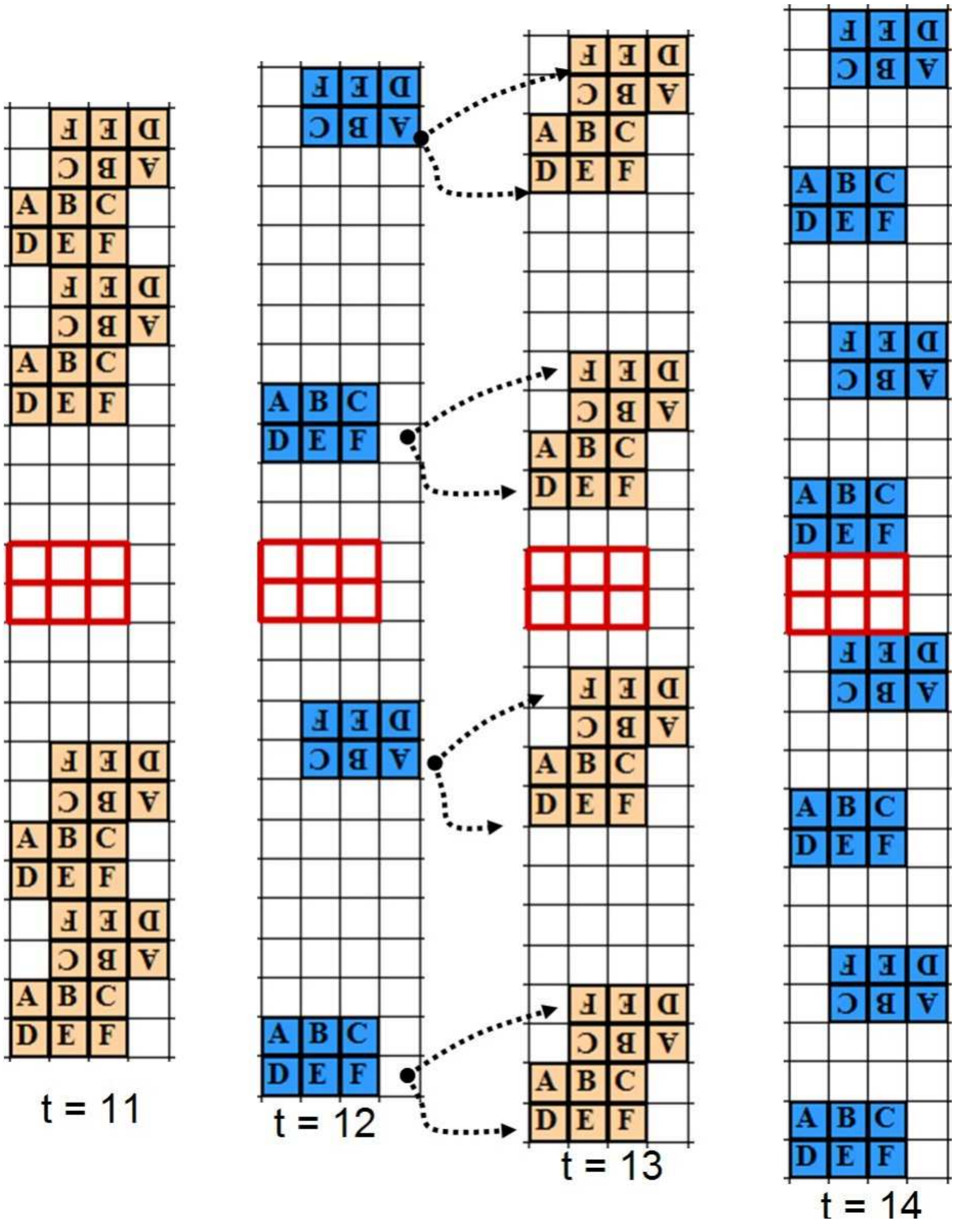


Figure 3.14: Continuation of Figure 3.13 from t=11 to t=14.

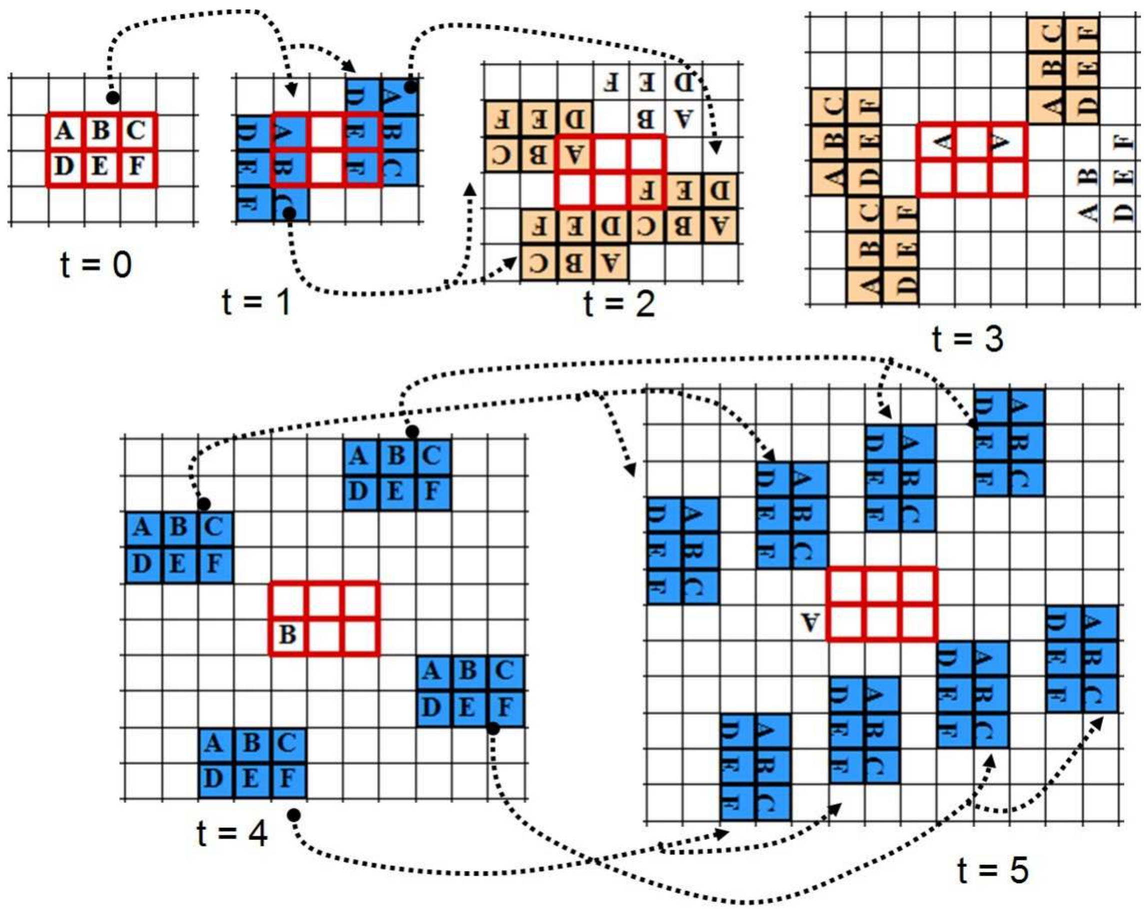


Figure 3.15: Self-replication, example 3.2-2: executing a separately generated R-tree against the same 6-oriented-component seed structure, from t=0 to t=5. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step.

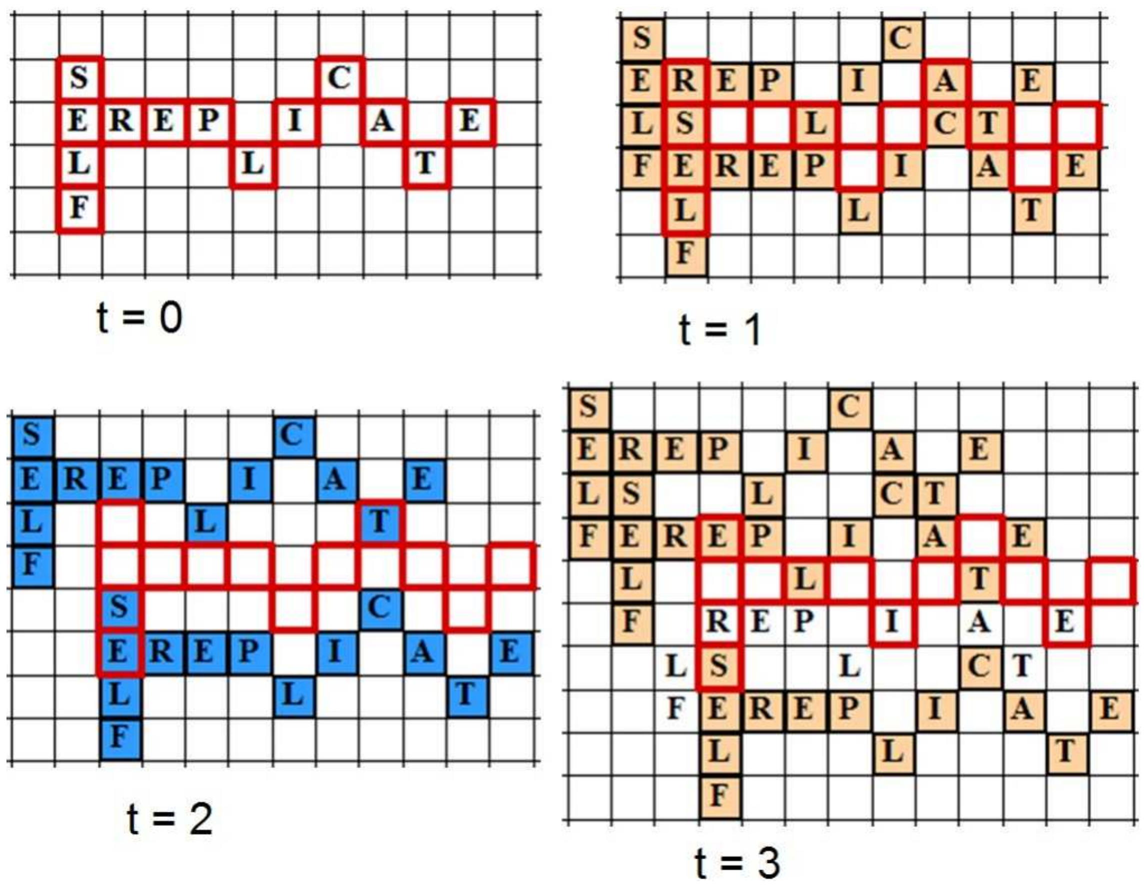


Figure 3.16: Self-replication, example 3.3: executing an automatically generated R-tree against a very large, irregular and arbitrary seed structure, from t=0 to t=4. For illustrative purpose, non-isolated seed structures are marked in yellow and isolated seed structures in blue. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step.

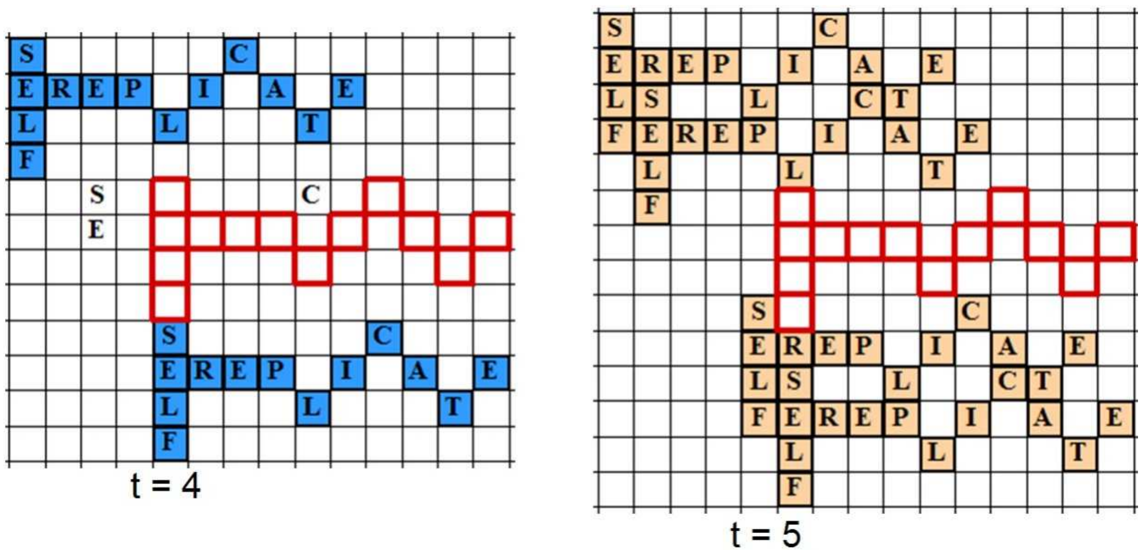


Figure 3.17: Continuation of Figure 3.16 from t=4 to t=5.

replicas at t=6. In the subsequent time steps, as shown in Figure 3.13 and Figure 3.14, each replica attempts to repeat the same self-replication process, interrupted by more collisions, but eventually reaches enough space at t=12, when each of the 4 seed replicas successfully replicates without collision and forms 8 (or 4 pairs) replicas at t=14.

Figure 3.15 shows that it is possible for the same seed structure to self replicate in different ways. A different R-tree is evolved in a separate GP run. Here, very surprisingly, the algorithm found a way to replicate the seed in only one (1) time step. In example 3.2-1 (Figure 3.12), when the active cells already translate at light speed, it took a minimum of 2 time steps to self-replicate. It appears that the result in example 3.2-2 (Figure 3.15) already exceeds light speed, which is not expected. A closer inspection of how the computer discovered strategy manages to do something seemingly impossible reveals that light speed has not been exceeded. Instead, the

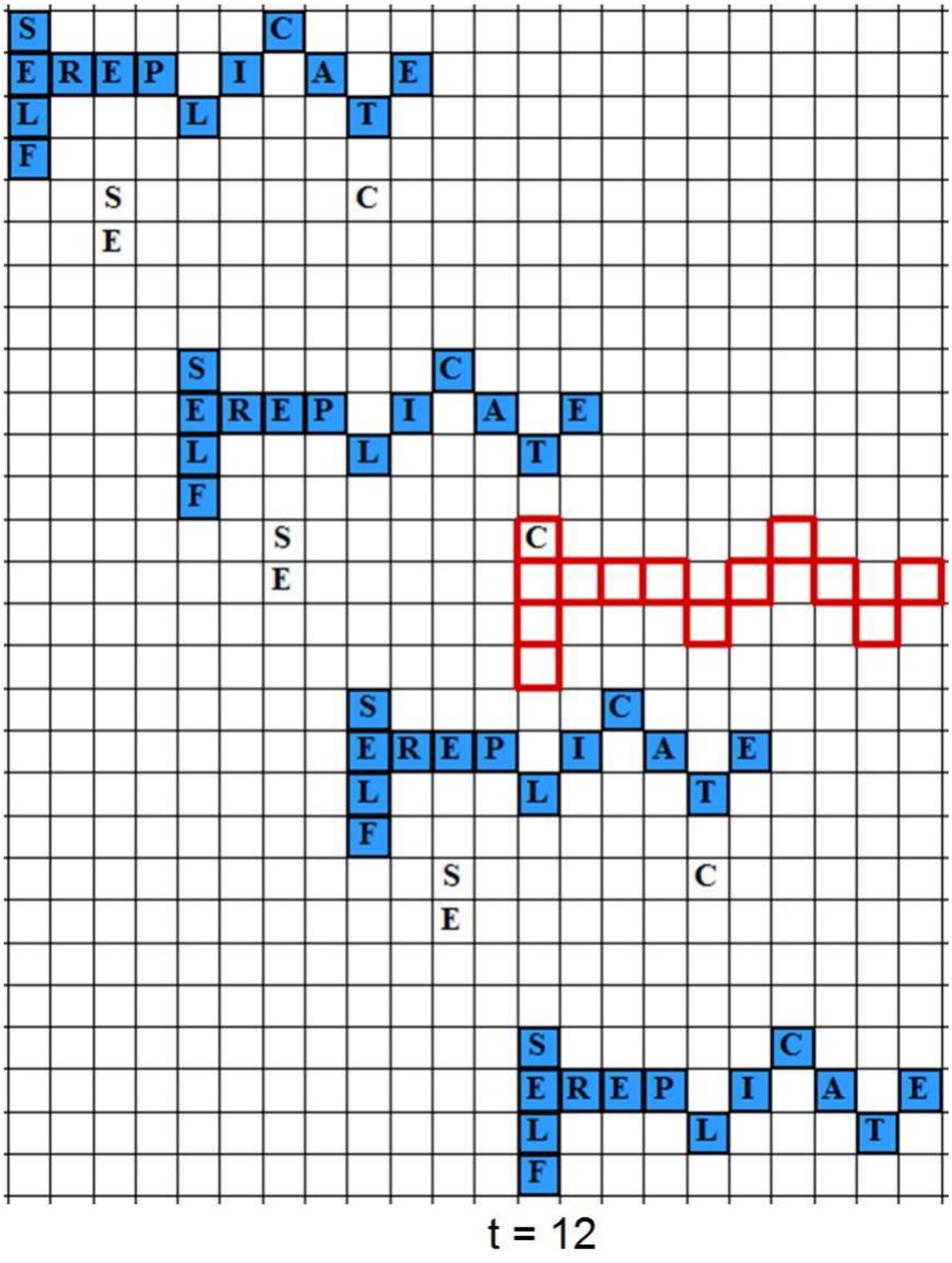


Figure 3.18: Continuation of Figure 3.17 at t=12.

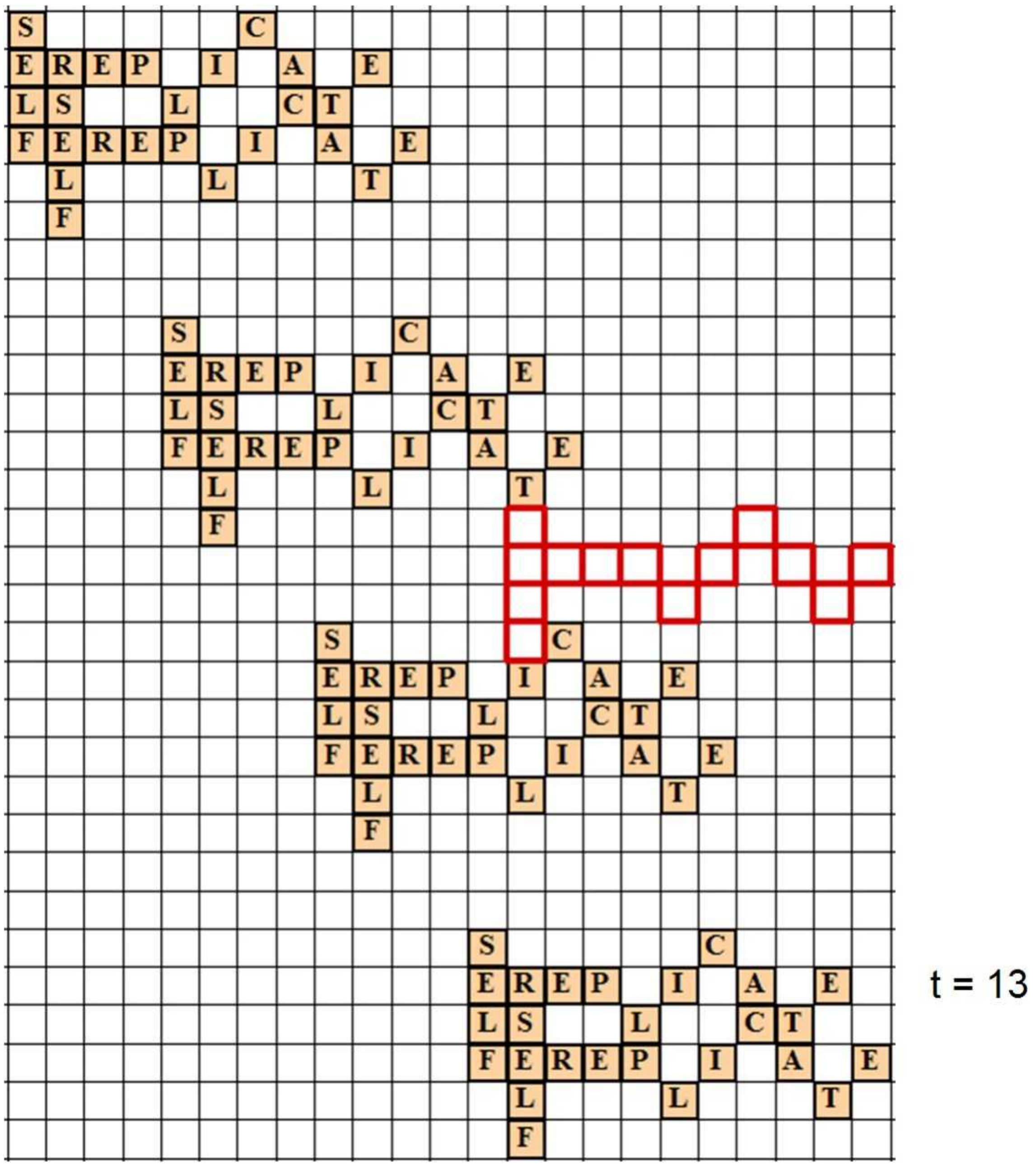


Figure 3.19: Continuation of Figure 3.18 at t=13.

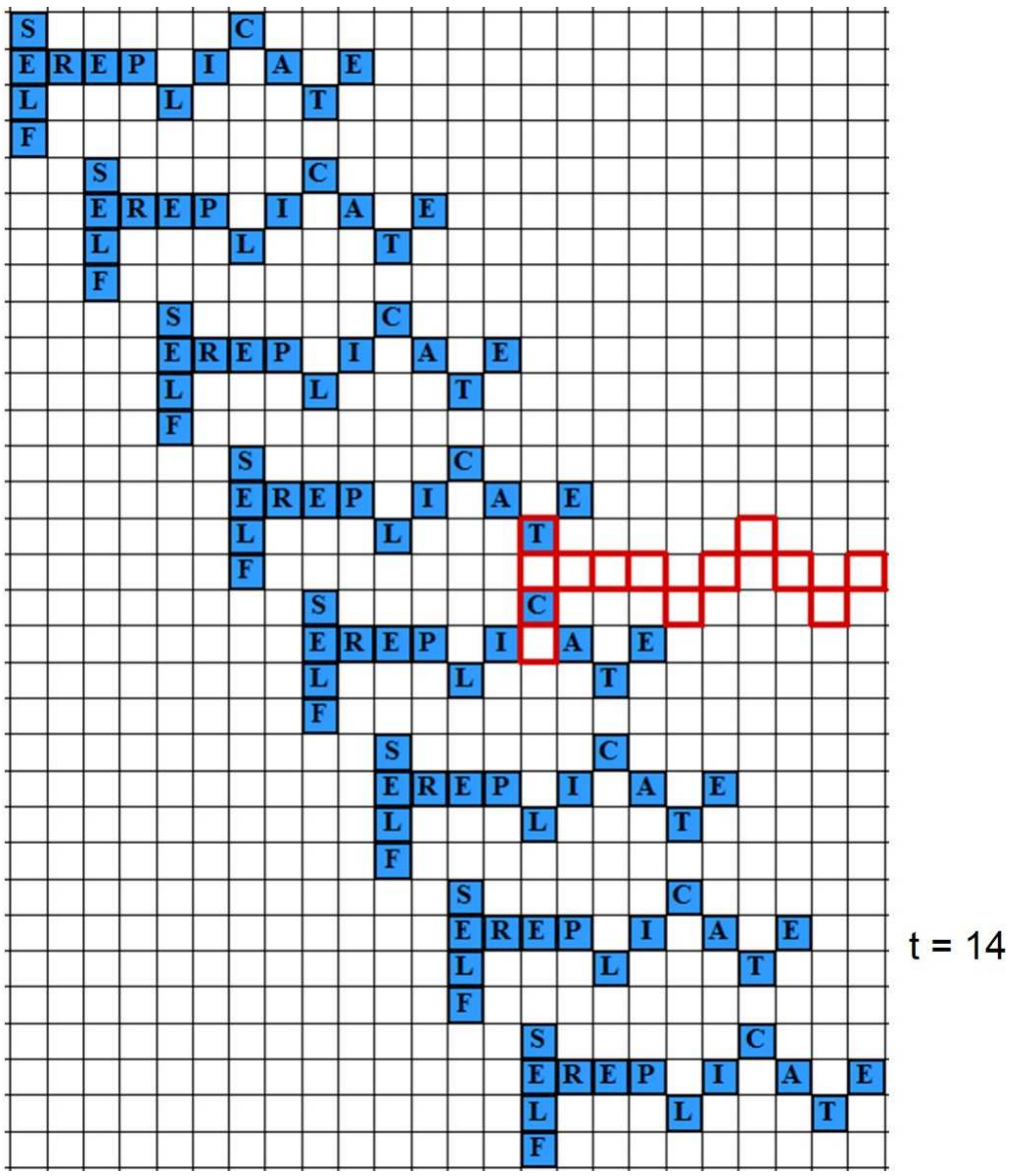


Figure 3.20: Continuation of Figure 3.19 at t=14.

seed structure is rotated before translation, so that the space originally occupied by the seed itself can be more efficiently used, yielding enough space to form a pair of replicas within only one time step.

The third structure to be presented here contains 13 components having an exceptionally large number of states (10 x 4 + 1 = 41 states/cell). It spells like "SELF REPLICATE" and is in purposely made very irregular and arbitrary, as shown in Figure 3.16 (t=0). It is demonstrated from Figure 3.16 to 3.20 that the GP model can nevertheless automatically program such an exceptionally large and irregular structure to self replicate, producing 8 replicas in 14 time steps, in a very consistent pattern.

3.6 Conclusions

This chapter established the S-tree, its properties, and how to generate it, and why it can unambiguously encode/decode arbitrary structures. Further, it introduced the R-tree, its properties, its genetic operators, and why it is more efficient than a linear rule table in evolution and simulation. Then, from these tree-based universal representations, a general genetic programming model for self-replication was established. Given a pre-specified arbitrary structure, it was shown how the S-tree is derived, how the R-tree population is initialized, how each R-tree is simulated, evaluated, and then fitness adjusted, tournament selection is used, and finally evolved, all guided by the goal of programming the R-trees to become better and better expressing the S-tree. Such a model represents a new genetic-programming based CA discovery paradigm, where the S-tree can be viewed as a representation of a global target, the R-tree can be viewed as a representation of local state-transition rules, and thus the CA can be automatically programmed to let the local state-transition rules produce desired global results when they are concurrently executed in each cell. This addresses the difficult issues of CA programming, especially involving self-replication.

Fitness evaluation of local state transition rules in terms of dynamic global behavior in CA, such as self replication, is a very difficult task [37]. Heuristic measures which only exploit partial information (such as component density) in the CA structure were found only useful for very small and simple structures. These heuristic measures assume the specific knowledge of a concrete structure, due to the lack of a universal structure representation which is capable of encoding the full structural information of an arbitrary structure that can be retrieved and exploited by an existing evolutionary system. The introduction of the S-tree and R-tree as a universal encoding mechanism of arbitrary structures/rules gives us an unprecedented ability to perform precise fitness assignment for full or partial matching structures at any stage. More importantly, such an S-tree based fitness measuring approach can be universally applied to arbitrary, future structures without requiring any knowledge about the structure a priori. This is because S-tree encoding has the capability of converting an arbitrary structure into a common tree-based structure, which can be fully exploited by an existing evolutionary system, built prior to any knowledge of the structure, to retrieve the complete structural information encoded in the S-tree, such as details about every Moore neighbor of every component in the structure. By retrieving such complete structural information from the S-tree and comparing it to a given configuration, we can precisely tell how well the current configuration satisfies the expected global computation. This represents a general evolutionary CA programming approach which could be readily applied to solve many other complex problems in CA [79, 4, 2, 74, 23], such as games [6, 14, 18, 19], and behavioral and social problems [20, 25, 17, 73]. Despite the local concurrent computations in CA, this approach provides a feasible genetic programming paradigm to automatically program the CA, even when the desired computation requires global communication and global integration of information across great distances in the cellular space, as in these other applications.

The limited experimental results presented in this chapter show not only that the application of evolutionary methods becomes feasible with this GP approach, but also that a whole new class of self-replicators can be discovered, with larger and arbitrary structures. These structures can be automatically programmed to self-replicate, without the use of a supercomputer. This approach also removes the restriction, often assumed in past CA models, that replicas must have a specific loop-like structure. It represents a significant step forward and a new level of ability in creating self-replicating structures because it qualitatively reduces the time cost of replicator construction compared to past manual methods. Further, replicators created in this fashion are qualitatively different and resemble biological cell mitosis more than do these generated in past studies (universal constructors and loops). Here, automatically discovered replicators can duplicate themselves very quickly, in a fission-like or rotational process. The computer model surprisingly discovered a strategy to allow a large structure to self-replicate within only one time step, representing a speed which was not known previously to be possible. The development of such a novel and time-effective method for generating self-replicating structures opens up the possibility of studying replicator configuration properties in a systematic way. It is to this task that we now turn.

Chapter 4

The Replicator Factory and Properties of Discovered Self-Replicating Cellular Automata Systems

4.1 Overview

In this chapter, I use the encoding mechanisms and GP algorithm presented in the previous chapter as a *replicator factory* to produce a wide range of arbitrary self-replicating structures. Using this approach, one has the ability to automatically generate whole families of self-replicating structures, and to systematically investigate how their properties vary as one systematically varies their features, such as the initial configuration size, shape, symmetry, and allowable states [103]. I then further study how variations of the initial configuration alter the effectiveness of the replicator factory itself in terms of evolutionary cost in synthesizing the replicas as well as removing the debris produced during the self-replication process.

4.2 The Replicator Factory and Debris Cleaning with Multi-Staged GP

Typically, after the desired number of replicas have formed, there exist in the CA space extraneous components which do not belong to any replica. See Figure 4.1 (a) for an example. These extraneous components are called *debris.* If desired, the

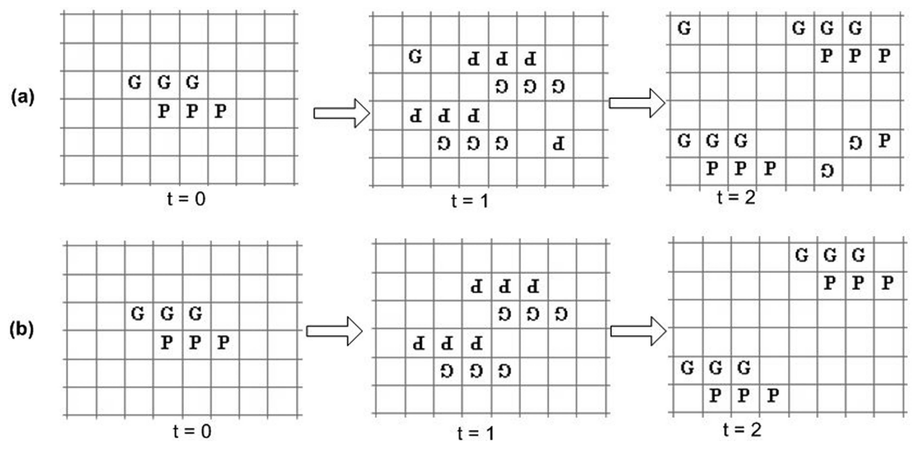


Figure 4.1: An example of the self-replication process, without (a) and with (b) debris removal. The seeds, number of time steps, and resulting replicators in (a) and (b) are identical, but the R-tree used in (a) is produced without encouraging debris removal while the R-tree used in (b) is produced with debris removal encouraged in the fitness function.

replicator factory can also use multi-stage evolution to enable debris removal, where the artificial evolutionary process is performed in two stages, namely a replication stage and a debris removal stage. In the first stage, the focus is on creating replicas and no attempt is made to depreciate extraneous components and partial structures. When the desired number of replica structures are formed, the process enters the second stage. Now the initial fitness evaluation functions are still enforced, but the overall fitness value is lowered an amount proportional to the amount of debris found at the end of self-replication. For example, Figure 4.1 depicts the replication process using an R-tree produced at the end of the first stage and the second stage. The replicators used in this study can be produced in isotropic CA space. However, in simulation done here it was found that debris cannot always be completely removed unless one gives up isotropy, so this requirement is relaxed in the results presented here to permit comparison of replication with and without debris. At the top level, the evolutionary process discontinues if the final hesitation exceeds a pre-specified number of generations (this number adaptively increases as fitness increases), and automatically restarts from a new random population of R-trees (more details in Section 3.4.3). Under these conditions, all seed structures described here successfully result in a fixed set of rules that produce self replication.

4.3 Results: Properties of Evolved Self-Replicating Structures

The replicator factory gives us an unprecedented ability to automatically generate a variety of self-replicating CA structures. In this section I aim to gain a deeper understanding of how variation of evolved self-replicators impacts the effectiveness of my model in terms of both evolution and self-replication. The broad plan is to vary parameters one at a time, perform computational experiments to obtain performance measures of groups of replicators, and inspect if there is a trend between the performance measures and each respective parameter. The performance measures from each experiment are collected as observations, which are then used as sampling data for subsequent linear regression analysis and data visualization. To enhance the compatibility of these sampling data across the varying replicators and minimize the impact of random factors in these observations, the following methods are used. First, let all experiments share same set of genetic programming parameters in each group (typically, a population of 200, tournament size of 3, crossover rate of 0.85, and mutation rate of 0.45, etc.), such that each self-replication experiment executes same genetic operators of the same probabilities. In addition, in each group, each experiment repeats another except for only one replicator property (such as the seed size) which is incrementally changed, all else held constant. Thirdly, for each experiment, instead of using a single measure, collect all of the following in a single observation: 1) stepDup (number of time steps used by the resulting replicator to form the first pair of replicas), 2) evoDup and timeDup (number of GP generations and computation time used during evolution to form the first pair of replicas), 3) evoClean and timeClean (number of GP generations and computation time used in evolution to form the first pair of replicas with all debris removed), 4) ruleDup (size of the evolved R-tree which enables the replicator to form the first pair of replicas), and 5) ruleClean (size of the evolved R-tree that enables the replicator to form the first pair of replicas with all debris removed). The unit of time in timeDup and timeClean measurements is one minute.

4.3.1 Seed Size

In CA model, a structure can simply be viewed as a configuration of contiguous active cells [65, 34]. The number of active cells in the initial replicating structure is called the seed size. I used the replicator factory to systematically investigate how seed size influenced replication of 16 structures having 4 to 56 components (Figure 4.2 shows examples), all else held constant. These structures share the following properties: 1) The number of allowable states is identically 7. These states

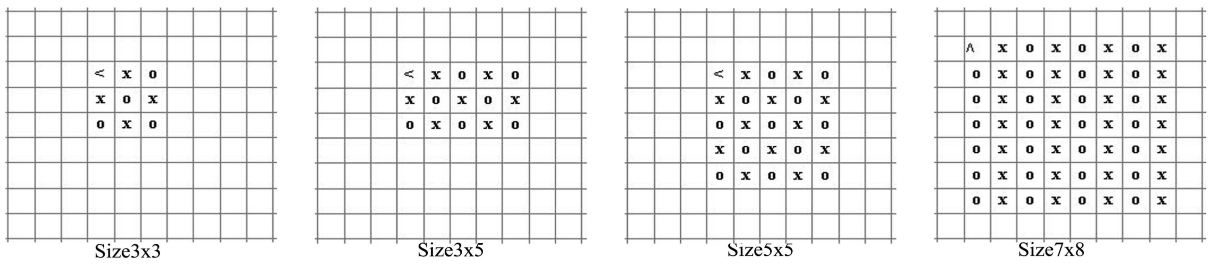


Figure 4.2: Examples of sixteen structures which vary from 4 to 56 components. With the rules evolved by the replicator factory, each of these structures replicates itself within ≤4 steps. See text for details.

Table 4.1: Performance measures for replicators in Figure 4.2.

Category	Width	Length	Size	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
size2x2	2	2	4	2	516	537	3.45	3.717	35	33
size2x3	2	3	6	2	541	550	3.833	3.95	47	47
size2x4	2	4	8	1	38	39	0.25	0.3	25	25
size2x5	2	5	10	2	575	576	4.367	4.383	58	53
size3x3	3	3	9	2	551	557	4.367	4.388	51	43
size3x4	3	4	12	2	582	616	4.733	5.21	70	64
size3x5	3	5	15	1	53	54	0.4	0.417	34	34
size3x6	3	6	18	2	672	706	5.883	6.4	84	83
size3x10	3	10	30	2	1133	1155	9.367	9.683	68	72
size4x4	4	4	16	3	1583	2373	17.8	34.583	117	99
size4x5	4	5	20	4	3177	3237	52.8	54.4	137	105
size5x5	5	5	25	3	1291	1306	15.5	15.817	91	91
size5x6	5	6	30	4	2137	7573	78.1	396.017	171	150
size6x6	6	6	36	4	3029	3029	77.283	77.283	121	121
size7x7	7	7	49	4	2159	2217	59.633	61.9	76	76
size7x8	7	8	56	4	1645	1660	39.017	39.65	130	128

include the quiescent state, 4 states of an oriented component (⟨), and 2 states of 2 non-oriented components (x and o, which are typical states used in manual self-replicating loops). 2) The shape of the structures is either a square or rectangle, of size from 2x2 to 7x8. Each structure contains exactly one oriented component, always in the upper-left position, and then populates alternatively with the two non-oriented components. 3) The size of the seed structures is gradually varied from 4 to 56 components. 4) Each has a rule set (R-tree) that enables it to self replicate with no debris within 1∼4 time steps that was discovered by the replicator factory. As

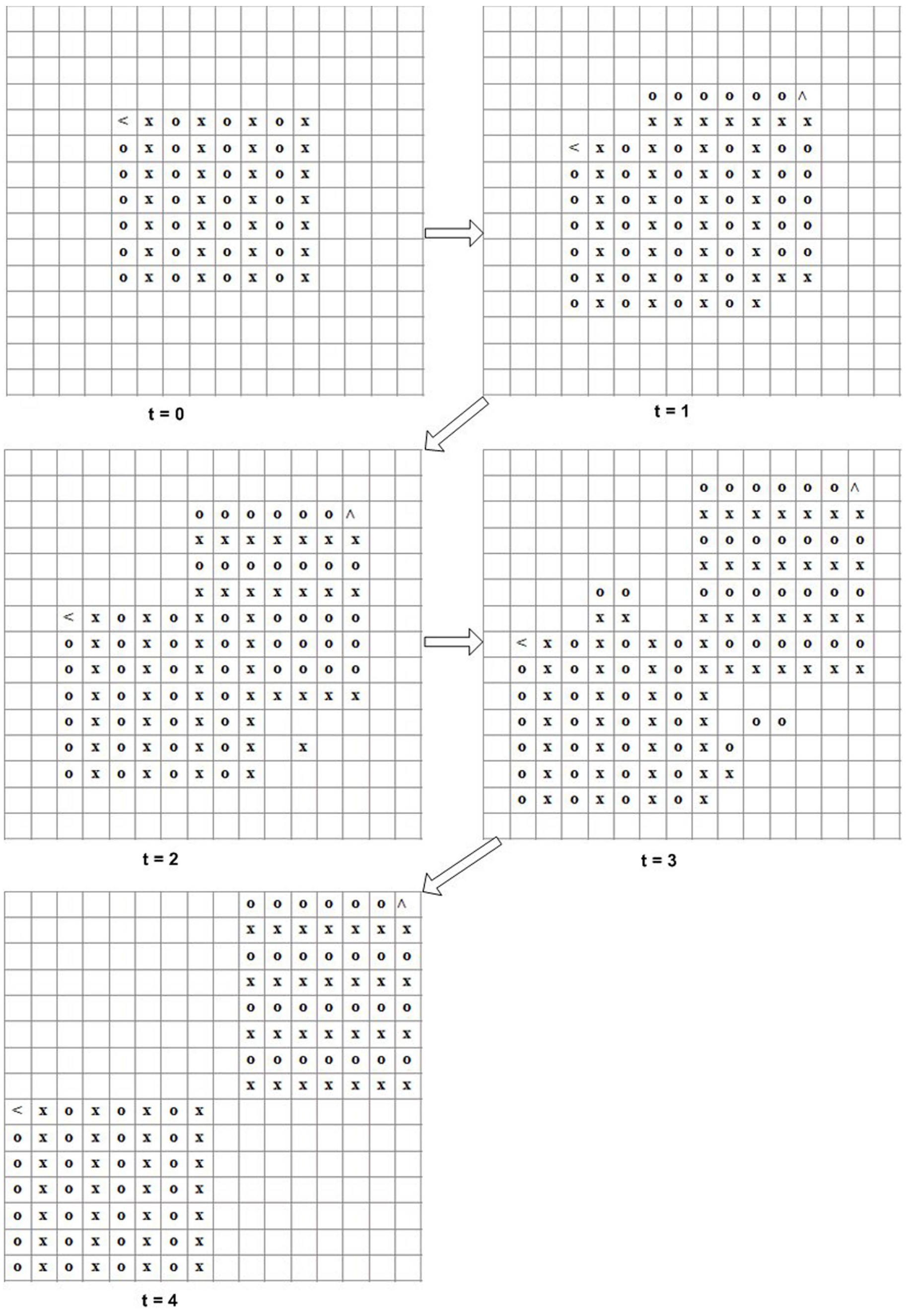


Figure 4.3: The replicator factory evolved an R-tree causing this 7x8 structure (t = 0) to replicate in just 4 time steps. The seed structure splits into two isolated replicators as both of the replicas translate diagonally in opposite directions at light speed (one cell each time step).

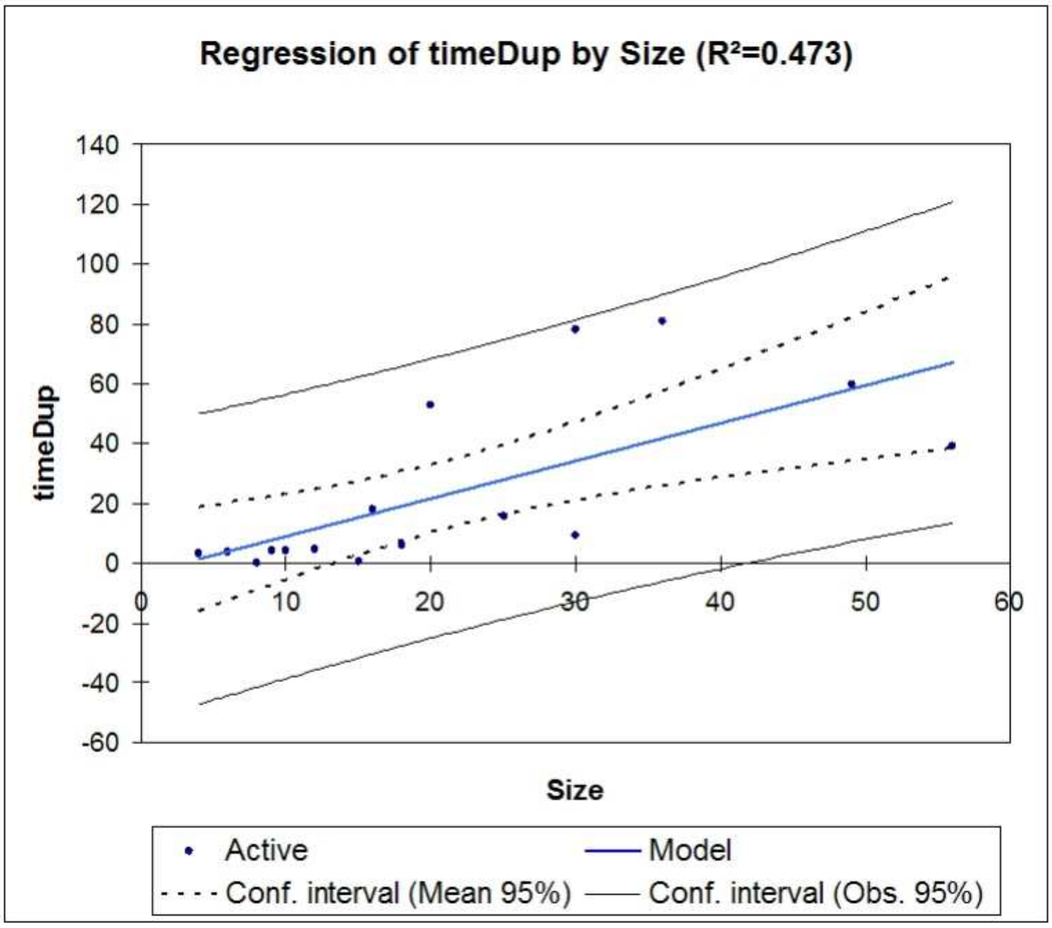
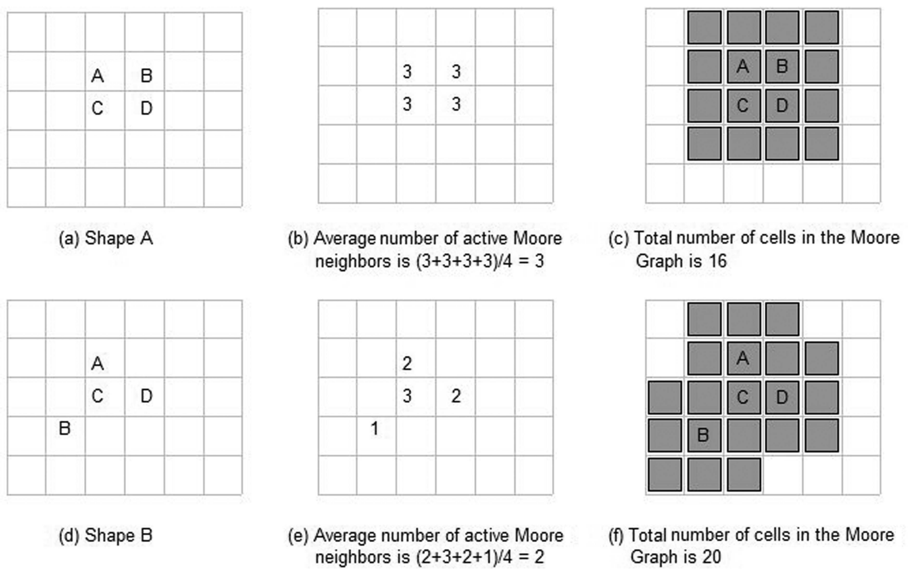


Figure 4.4: The linear regression chart between the seed size and timeDup produced from the sample data shown in Table 4.1. This chart shows the observed values from the table, the regression line and both types of confidence interval. The determination coefficient value R^2 is shown on the chart title [113].

an example, I illustrate the self replication steps of the 7x8 structure in Figure 4.3. It can be seen that the seed structure splits into two isolated replicators in a way when both the parent and child sub-structures translate in opposite directions at light speed (one cell each time step), and hence reach enough CA space to form and isolate the replicators in minimum steps of time. The properties and performance measures of each of these replicating structures are summarized in Table 4.1. The Size column indicates the number of components in each seed structure. Note the tables both the 2x4 and 3x5 structures can self-replicate in only one time step. Even though this seems impossible to human, but this is not error. The R-trees evolved by the replicator factory manage to do that with the same strategy shown in experiment example 3.2-2 in Chapter 3 (Figure 3.15).

From Table 4.1 one can determine whether there is a trend indicating seed size has an effect on the performance vector ⟨stepDup, evoDup, evoClean, timeDup, timeClean, ruleDup, ruleClean⟩. Linear regression is used for this analysis with the results visualized with regression charts. For example, Figure 4.4 plots the relationship between timeDup and seed size. Such a regression chart shows the observations (blue points), the regression line (the fitted model), and two confidence intervals (shown with grey lines): 95% confidence interval around the mean and 95% confidence interval around the observations. The determination coefficient value R^2 is shown on the chart title [113], and is interpreted as the proportion of the variability of the dependent variable linearly explained by the model. The nearer R^2 is to 1, the better the model. In summary, from the chart we can conclude with reasonable confidence that there is a positive relationship between size and timeDup. Similar charts for other measurements are not included here but give similar results.

4.3.2 Seed Shape



A B
C D
(a) Shape A
3 3
3 3
(b) Average number of active Moore neighbors is (3+3+3+3)/4 = 3
A B
C D
(c) Total number of cells in the Moore Graph is 16
A
C D
B
(d) Shape B
2
3 2
1
(e) Average number of active Moore neighbors is (2+3+2+1)/4 = 2
A
C D
B
(f) Total number of cells in the Moore Graph is 20

Figure 4.5: When a sample shape is varied from (a) to (d) above, the correspondingly change in Average Number of Active Moore Neighbors and Moore Graph Coverage is illustrated in (b) and (e), and (c) and (f) respectively. The numbers in (b) and (e) reflect the number of active Moore neighbors of that cell. The shaded cells in (c) and (f) are the cells covered by the Moore Graph of the shown structures. This example shows that a shape of higher complexity tends to reflect lower average number of active Moore neighbors and higher number of cells in the Moore Graph.

Presumably a structure of higher complexity in shape would take longer to evolve toward self-replication. However, to verify this hypothesis through quantitive analysis, there needs a way to measure shape complexity. Two measures applicable to arbitrary structures are used. The first is called Average Number of Active Moore

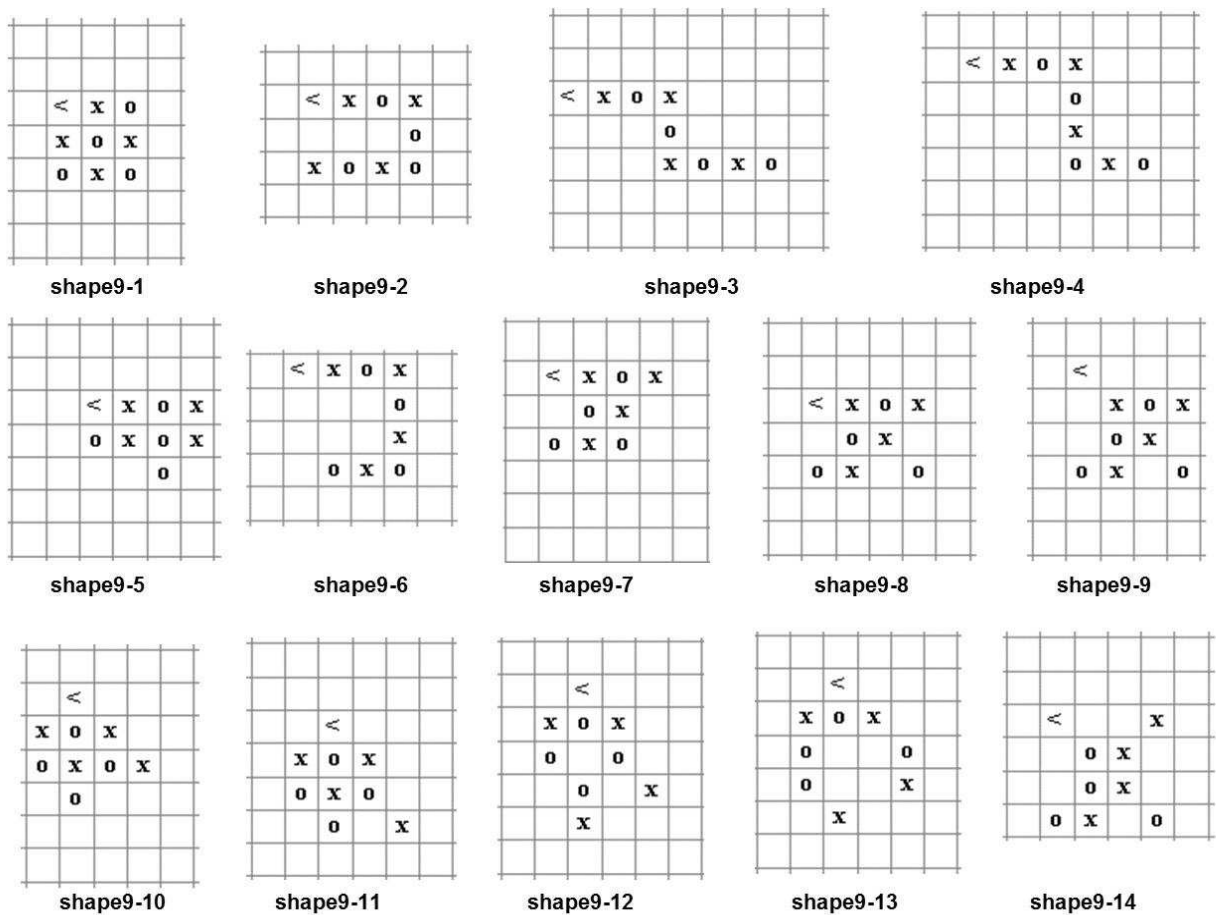


Figure 4.6: Fourteen structures having identical size and allowable states, but varying shapes. Rules to make each self-replicate were produced by the replicator factory.

Neighbors, which can be calculated as follows. First, for each active component in the seed, count the number of active Moore neighbor cells. Then, sum all of them up and divide the result by the seed size. The second measure is the S-tree size (the number of nodes in the S-tree), which is the same as the number of nodes in the Moore graph, because each node in the Moore graph appears in the S-tree precisely once. This parameter can be calculated by counting the number of cells of which at least one of its Moore neighbors is an active component of the seed structure. For

Table 4.2: Performance measures of replicators in Figure 4.6.

Category	avMrNbr	sTSize	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
shape9-1	4.444	25	2	164	179	2.333	2.667	66	63
shape9-2	2.222	30	2	214	234	3.567	4.033	76	76
shape9-3	2.222	33	2	187	213	2.783	3.367	69	65
shape9-4	2.222	33	2	174	185	2.35	2.583	68	69
shape9-5	4	27	2	166	188	2.3	2.833	76	76
shape9-6	2.222	33	2	240	240	4	4	66	66
shape9-7	3.778	28	2	142	151	1.9	2.08	57	49
shape9-8	3.333	30	2	182	196	2.683	3.017	79	79
shape9-9	3.111	33	3	778	778	20.6	20.6	77	77
shape9-10	4.222	29	2	155	158	2.08	2.15	40	43
shape9-11	4	30	2	132	132	1.6	1.6	40	40
shape9-12	2.889	33	3	573	573	13.867	13.867	96	96
shape9-13	2.444	35	3	427	427	10.167	10.167	136	136
shape9-14	2.889	36	3	764	772	20.6	20.833	78	78

example, the seed in Figure 4.5(a) is a 4-component structure which takes a simple square shape. The seed shown in Figure 4.5(d) is identical except that the component B is moved from upper right to lower left position. As a result, the shape becomes irregular and more complex. Correspondingly, Figure 4.5(b) and 4.5e reflect that the Average Number of Active Moore Neighbors has dropped from 3 to 2, indicating that, in more complex shapes, components tend to spread more apart in various ways. At the same time, Figure 4.5(c) and 4.5f show that the Moore graph coverage increases from 16 to 20, indicating more complex shapes tend

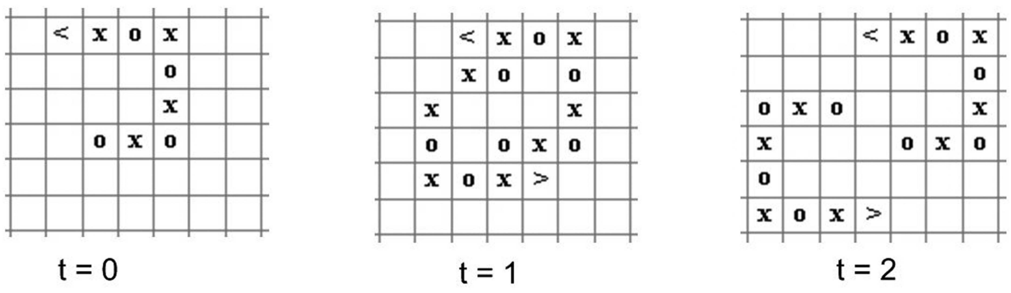


Figure 4.7: Sample replication process for one of the 14 structures of Figure 4.6. Note that the splitting replicators move at light speed toward opposite directions and get isolated in only 2 time steps. Further, the replicators have aligned themselves adaptively according to its own shape in a way that they take the minimum cellular space but nevertheless fully isolate from each other.

to reach more CA space and produce a bigger Moore graph and S-tree. In sum, a lower Average Number of Active Moore Neighbors and higher S-tree size would indicate a higher shape complexity.

To quantitively investigate the impact of seed shape on the effectiveness of the replicator factory, I have chosen 14 structures, shown in Figure 4.6, which share the following properties: 1) the seed size is identically 9; 2) the number of allowable states is identically 7; 3) the shape varies randomly; and 4) each can duplicate based on an R-tree, evolved by the replicator factory, that enables it to self-replicate with no debris within 2∼3 time steps, with an example shown in Figure 4.7. Note that the splitting replicators move at light speed toward opposite directions and get isolated in only 2 time steps. Further, note that the replicators have aligned themselves adaptively according to its own shape in a way that they take the minimum cellular space but nevertheless fully isolate from each other. The shape complexity

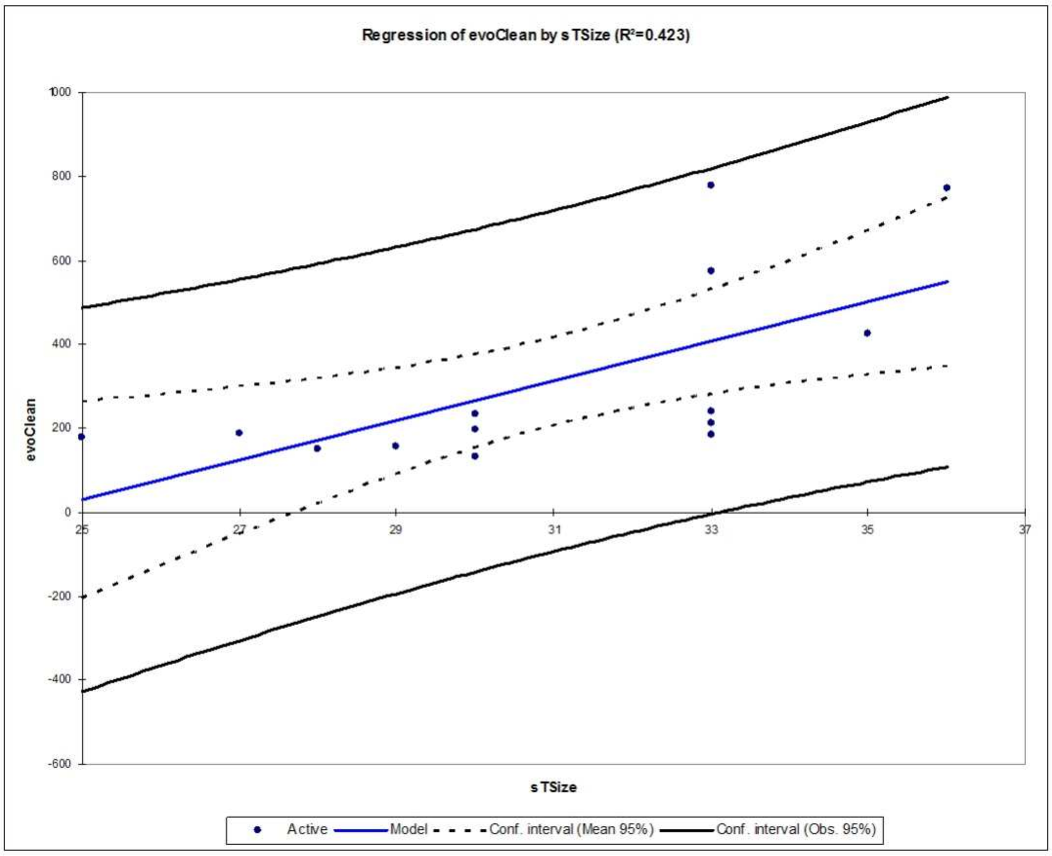


Figure 4.8: Linear regression chart between evoClean and the S-tree size (one of the shape complexity measures) derived from Table 4.2. This chart shows the observed values from the table, the regression line and both types of confidence interval around the predictions. The R^2 value shown on the chart title is interpreted as the proportion of the variability of the dependent variable (evoClean) linearly explained by the model (the S-tree size).

measures (column avMrNbr represents Average Number of Active Moore Neighbors and column sTSize represents the S-tree size mentioned above) and corresponding performance measures are summarized in Table 4.2. Pearson correlation coefficients [100] derived from this table indicate that the increase of S-tree size strongly encourages the increase of evolution cost and number of resulting CA rules required to enable self-replication. On the other hand, the increase of Average Number of Active Moore Neighbors (indicating more compact structure) moderately decreases evolution cost and relatively more strongly decreases the number of resulting CA rules. Linear regression analysis leads to consistent results, with one sample chart shown in Figure 4.8.

4.3.3 Cell States

Table 4.3: Performance measures of 2 sets of replicators shown in Figure 4.9.

Category	numState	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
state5x5-6	6	3	835	854	18.583	19.183	86	86
state5x5-7	7	3	936	1011	15.383	17.65	68	68
state5x5-8	8	3	1165	1358	31.633	38.617	141	140
state5x5-9	9	3	1032	1295	33.833	45.383	101	85
state5x5-10	10	3	1250	1310	29.217	30.017	101	102
state5x5-11	11	3	1487	1567	29.433	32.667	120	106
state5x5-12	12	3	1207	1224	28.333	29	122	97
state5x5-13	13	3	1349	1385	32.567	34.017	115	113
state5x5-14	14	3	1305	1324	32.3	33.067	105	96
state5x5-15	15	3	1287	1504	31.983	39.817	132	139
loop-7	7	2	194	217	3.267	4.033	67	61
loop-8	8	2	166	191	2.45	3.08	48	40
loop-9	9	2	252	290	4.633	5.6	58	50
loop-10	10	2	178	182	2.85	3.1	63	62
loop-11	11	2	133	247	3.583	3.9	58	42
loop-12	12	2	193	314	3.35	6.63	62	58
loop-13	13	2	225	255	5.4	6.366	84	82

Here I choose two separate sets of structures to study how the number of allowable states affects the effectiveness of the replicator factory. The first set,

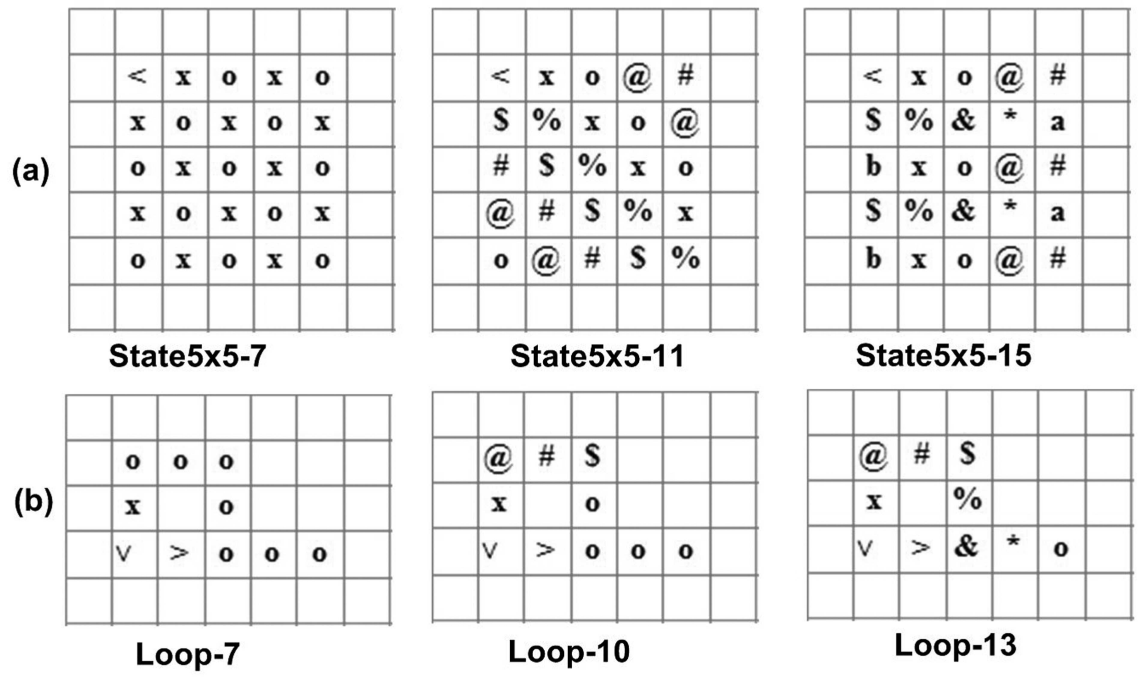


Figure 4.9: Samples of 2 sets of replicators: the first set (a) has size of 25, square shape, and gradually increasing number of allowable cell states (from 6 to 15). The second set (b) has size of 10, loop shape, and gradually increasing number of allowable cell states (from 7 to 13).

for which examples are shown in Figure 4.9(a), includes 10 structures which share the following properties: 1) the seed size is identically 25; 2) the seed shape is identically a 5x5 square; and 3) the number of allowable states varies from 6 to 15. Each structure contains exactly one oriented component, all in the upper-left position, and a different number of non-oriented components corresponding to the total number of allowable states. Note that each new oriented component (regardless of its orientation in the seed) introduces 4 new states, while each new non-oriented component introduces 1 new state. The replicator factory successfully discovered a rule set (R-tree) for each structure that enables it to self replicate with no debris within 3 time steps.

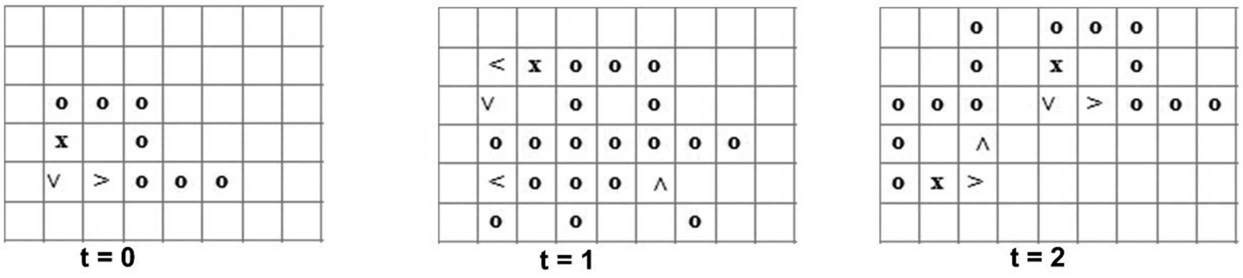


Figure 4.10: The loop structures in set 2, as one example shown here, quickly (within only 2 time steps) reaches enough cellular space to form replicators by moving, sometimes plus rotating (as shown here), both the parent and child structures at the same time. A manually designed loop, however, typically replicates itself by extending a single construction arm while the parent loop itself remains still, and hence usually takes a lot more time steps to reach enough to form a child replicator.

Motivated by past studies of self-replicating loops [9], the second set of structures, examples of which are shown in Figure 4.9(b), includes 7 structures which share the following properties: 1) the seed size is identically 10; 2) the seed shape is identically a 10-component loop; and 3) the number of allowable states varies from 7 to 13. Each structure contains 2 adjacent oriented components, followed by, repeatedly, a different number of non-oriented components corresponding to the total number of allowable states. The replicator factory also successfully discovered a rule set (R-tree) that enables each of these structures to self replicate with no debris within 2 time steps. Replication takes place in a fashion similar to that of the other replicating structures in this paper, which is different from the replication process used in past manually designed self-replicating loops [61, 1, 9]. A manually designed loop typically replicates itself by extending a single construction arm while the parent loop itself remains still, and hence usually takes a large number of time steps to reach enough to form a child replicator. However, the loop structures in set 2, as one example shows in Figure 4.10, quickly (within only 2 time steps here) reaches enough cellular space to form replicators by moving, sometimes plus rotating (as shown here), both the parent and child structures at the same time. The performance measurements from both sets of structures are listed in Table 4.3. The numState column indicates the total number of allowable states of each structure in each set.

A correlation matrix calculated from Table 4.3 suggests the following: 1) the number of time steps required for a structure to self-replicate, whether it is a square or a loop, is not dependent on the number of allowable states; 2) however, the number of allowable states has a strong and positive influence on the evolution cost, computation time, and size of rule set required to enable self-replication. Linear regression analysis done in the same fashion as previously described gives similar results.

4.3.4 Repeated Components

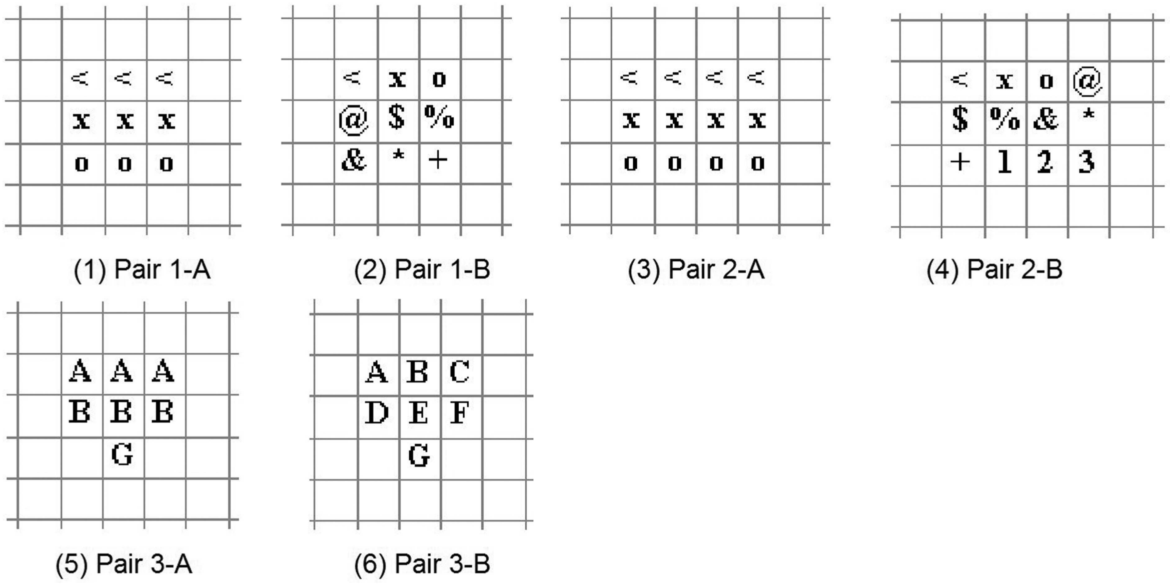


Figure 4.11: Three pairs of replicators used to study the impact of repeated components. Structures in each pair have the same size, shape, number of allowable states, but only the first one (A in each pair) has repeated components.

Table 4.4: Performance measures of the replicators shown in Figure 4.11.

Category	SeedType	ifRepCom	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
Pair 1-A	Square	Yes	2	182	226	3.167	4.433	71	71
Pair 1-B	Square	No	2	216	271	3.317	4.533	67	66
Pair 2-A	Lateral	Yes	2	134	134	2.017	2.017	46	46
Pair 2-B	Lateral	No	3	443	447	10.15	10.25	111	110
Pair 3-A	Irregular	Yes	2	628	646	7.583	7.999	51	46
Pair 3-B	Irregular	No	2	652	652	8.5	8.5	36	36

How does the presence of repeated components in the seed alter the effectiveness of the replicator factory? I have chosen three pairs of structures to investigate this matter, where each pair is a square, rectangle, or irregular shape respectively, as shown in Figure 4.11. The structures in each pair are identical (same shape, size, and number of allowable states), except that only the first structure contains repeated components. For each structure the replicator factory evolved an R-tree which enables the structure to self-replicate without debris within 2∼3 time steps. The performance measures are summarized in Table 4.4, where the SeedType column indicates the type of shapes each pair takes and the column ifRepCom is a Boolean variable, which has a value of YES if there is a presence of repeated components, and NO otherwise.

To visualize the influence of the presence of repeated components on the performance vector ⟨stepDup, evoDup, evoClean, timeDup, timeClean, ruleDup, ruleClean⟩, I generated a rescaled parallel-coordinates-plot, Figure 4.12, from Table 4.4, where the structures are grouped by the value of ifRepCom, and the mean values of each performance measurement of each group is rescaled on a scale of 0∼1 so that all variables are represented on the same scale (for each measurement, 0 corresponds to the minimum and 1 to the maximum). Here it can clearly be seen that on average the presence of repeated components reduces the number of evolution generations, computation time, and number of rules required to enable the structure to self replicate, with or without debris removal. From the derived correlation matrix it can also be seen that the YES value of ifRepCom has negative influence consistently on each item in the performance vector.

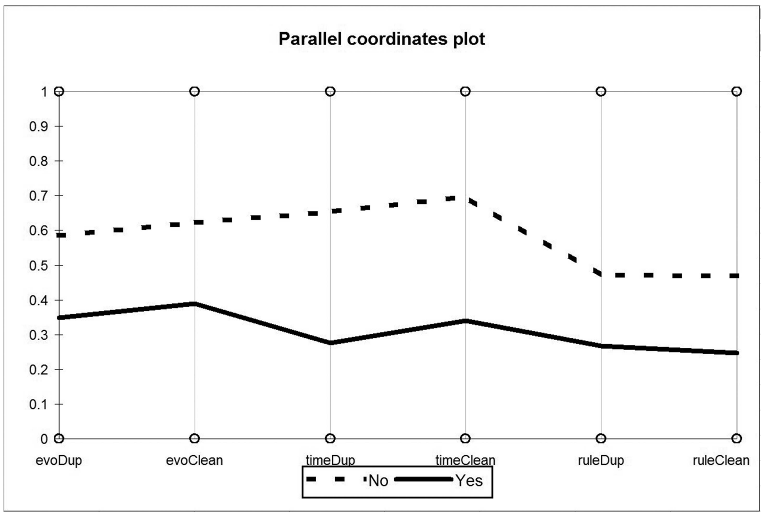


Figure 4.12: Rescaled parallel-coordinates-plot produced from Table 4.4 visualizes the impact of the presence of repeated components. The structures are grouped by the ifRepCom column, and the mean values of each performance measurement of each group is rescaled on a scale of 0∼1 such that all variables are represented on the same scale (for each measurement, 0 corresponds to the minimum and 1 to the maximum). The “Yes” line in the chart represents those observations of which the value of ifRepCom is Yes, and the “No” line in the chart represents those observations of which the value of ifRepCom is No.

4.3.5 Repeated Sub-Structures

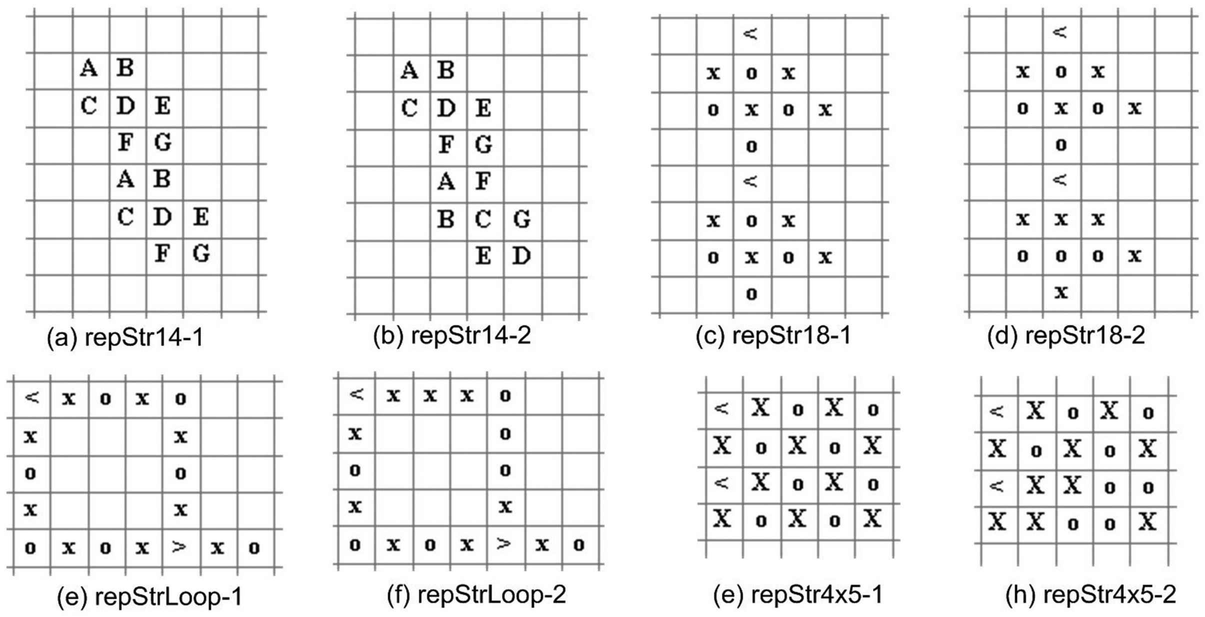


Figure 4.13: Four pairs of replicators: structures in each pair are identical except only the first structure contains repeated sub-structures.

Table 4.5: Performance measures of replicators in Figure 4.13.

Category	Shape	ifRepStr	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
repStr14-1	Irregular	YES	2	4243	4457	115.6	121.017	70	67
repStr14-2	Irregular	NO	2	3854	3930	113.567	115.449	75	70
repStr18-1	Irregular	YES	4	1513	1964	47.433	65.933	99	100
repStr18-2	Irregular	NO	4	33731	34519	1491.12	1526.23	153	150
repStrLoop-1	Loop	YES	3	957	958	41.867	41.933	143	143
repStrLoop-2	Loop	NO	4	3878	3878	115.833	115.833	117	117
repStr4x5-1	Regular	YES	3	1986	2485	49.783	66.8	119	119
repStr4x5-2	Regular	NO	3	2754	2754	146.317	146.317	145	145

How does the presence of repeated sub-structures in the seed alter the effectiveness of the replicator factory? To investigate this matter, as shown in Figure 4.13, I have chosen 4 pairs of structures which share the following properties: 1) the structures in each pair are identical (same size, shape, number of allowable states) except only the first structure contains repeated sub-structures; 2) from pair to pair, the size varies from 14 to 20 components; 3) from pair to pair, the shape varies from irregular, loop, to regular; 3) from pair to pair, the number of allowable states varies from 7 to 29. The replicator factory successfully evolved an R-tree for each structure to self replicate without debris within 2∼4 time steps, with the performance measures summarized in Table 4.5, and with the self-replication process of the first pair of structures shown in Figure 4.14 as an example. In Table 4.5, the Shape column indicates the type of shape each pair belongs to, and the ifRepStr indicates which structure in each pair contains repeated sub-structures.

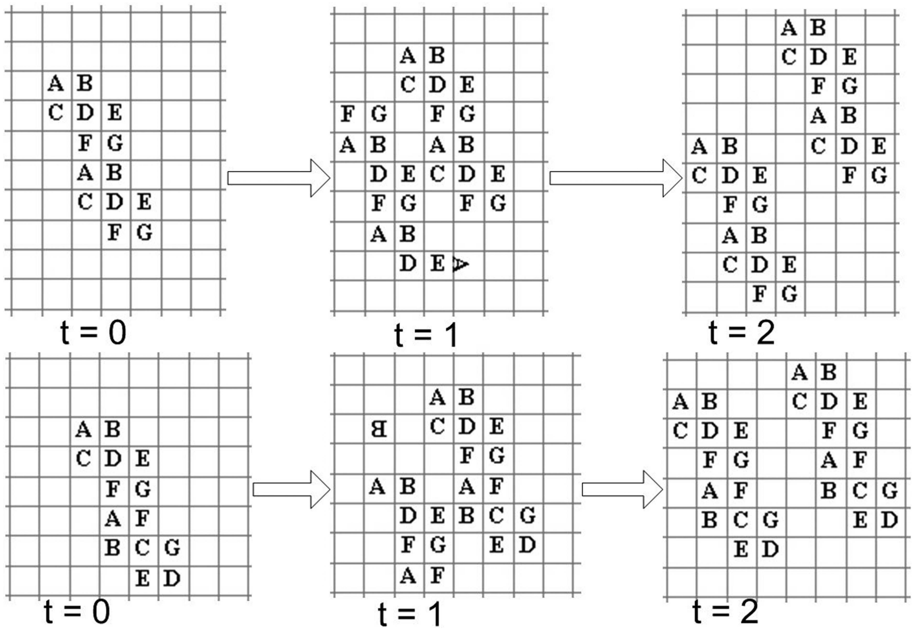
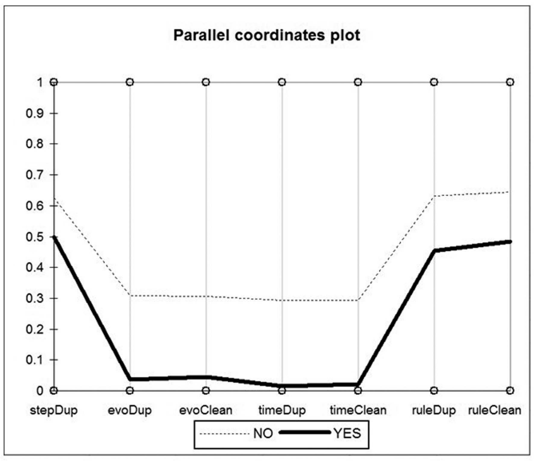


Figure 4.14: Self-replication of the first pair of replicators in Figure 4.13. The upper-left seed has repeated sub-structures. The bottom left seed does not.

To visualize the influence of the presence of repeated sub structures on the performance vector ⟨stepDup, evoDup, evoClean, timeDup, timeClean, ruleDup, ruleClean⟩, I generated a rescaled parallel-coordinates-plot, Figure 4.15 (left), from



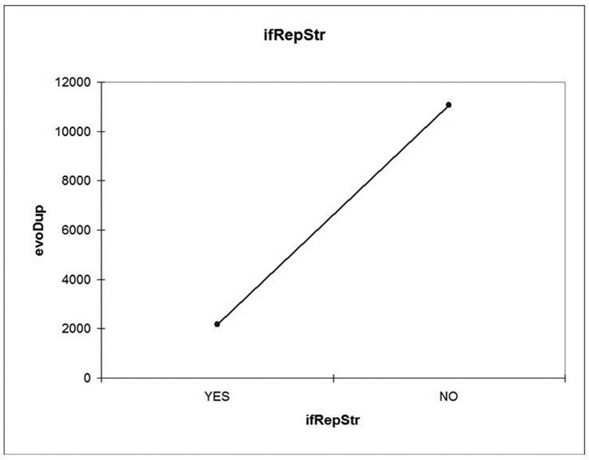


Figure 4.15: Rescaled parallel-coordinates-plot and mean chart visualizing the impact of the presence of repeated sub-structures. The presence of repeated sub-structures consistently lowers each measured evolutionary and replication cost. The "Yes" line in the chart represents those observations of which the value of ifRepStr is Yes, and the "No" line in the chart represents those observations of which the value of ifRepStr is No.

the performance summary shown above, where the structures are grouped by the value of ifRepStr, and the mean values of each performance measurement of each group is rescaled on a scale of 0∼1. Here it can clearly be seen that on average the presence of repeated sub-structures reduces the number of evolution generations, computation time, and number of rules required to enable the structure to self replicate, with or without the debris cleaned. From the derived correlation matrix and a sample linear regression chart shown in Figure 4.15 (right), it can also be seen that the YES value of ifRepStr has inverse influence on the performance measures.

4.3.6 Type of Symmetry

As shown above, repeated sub-structures tend to reduce the evolutionary cost toward self-replication. Does structure symmetry have a similar influence? In 2-D CA space, there are 4 types of symmetries: reflection, rotation, translation, and glide-reflection. I have tested each of these types. For example, each structure in Figure 4.16 has exactly the same sub-structures, size, and states, but with various types of symmetry. Note that translation symmetry is actually the same as repeated sub-structures. The replicator factory has successfully discovered R-trees for each of these structures that enable them to self-duplicate without debris within 2∼3 time steps. The self-replication of two of the structures, one of reflection and the other of glide-reflection symmetry, is illustrated in Figure 4.17. The performance measures of these replicators are summarized in Table 4.6. The TypeOfSymmetry column indicates which symmetry each replicator belongs to.

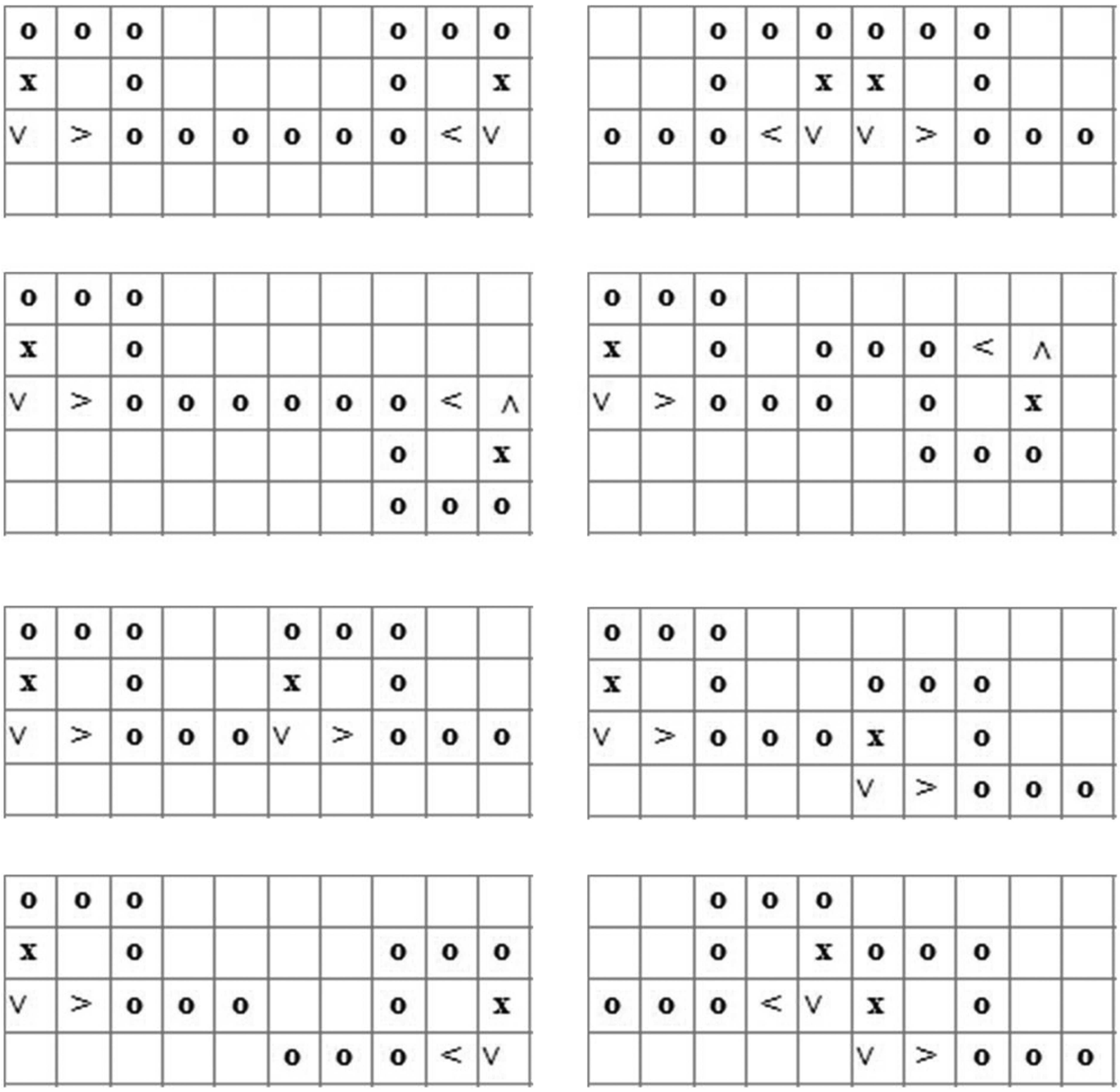


Figure 4.16: Sample seeds with reflection (top row), rotation (2^{nd} row), translation (3^{rd} row), and glide-reflection (last row) symmetry. These structures otherwise have exactly the same sub-structures, size, and number of allowable states.

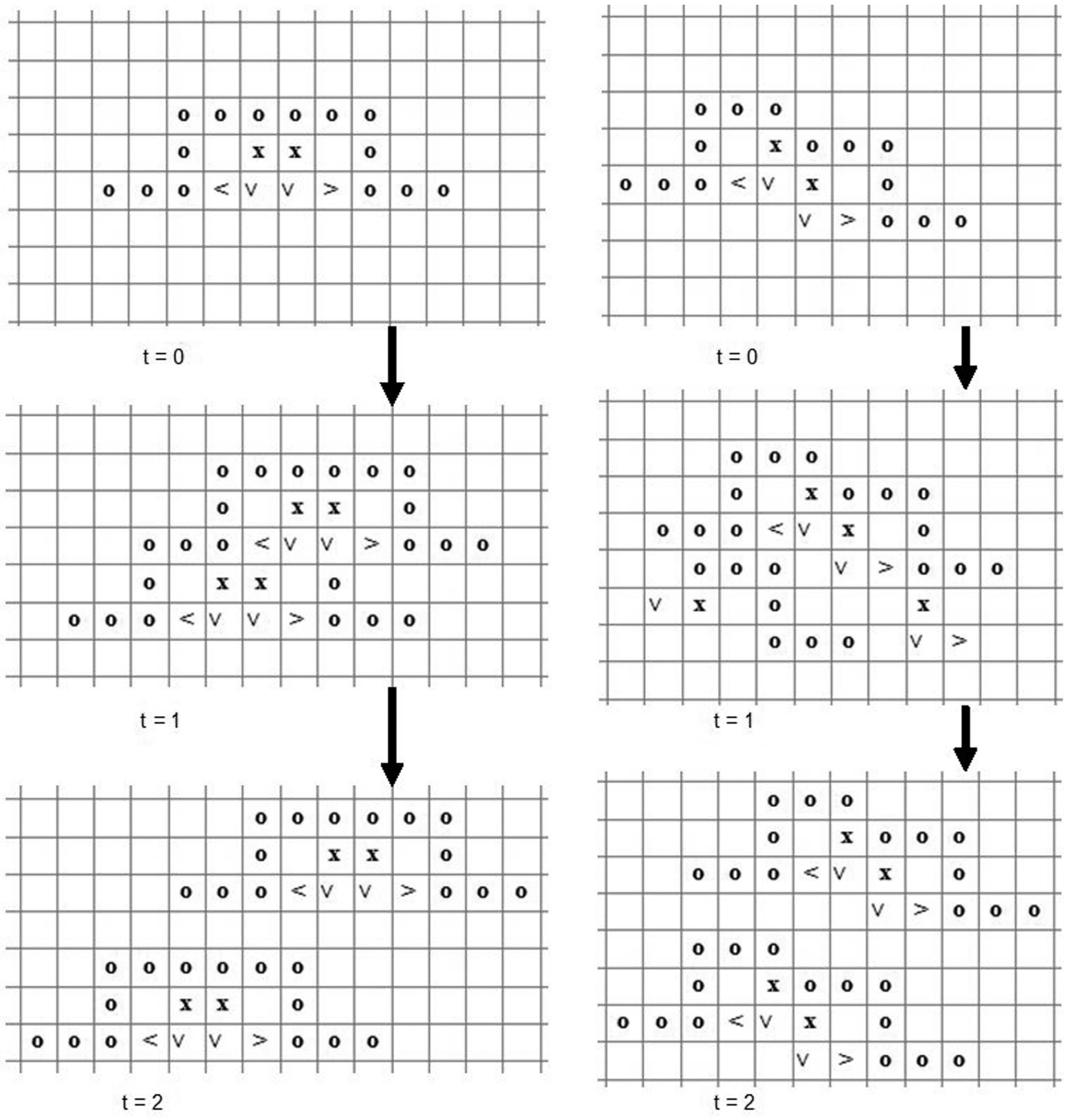


Figure 4.17: Self-replication of two of the structures in Figure 4.16. The structure on the left has reflection symmetry while the one on right has glide-reflection symmetry.

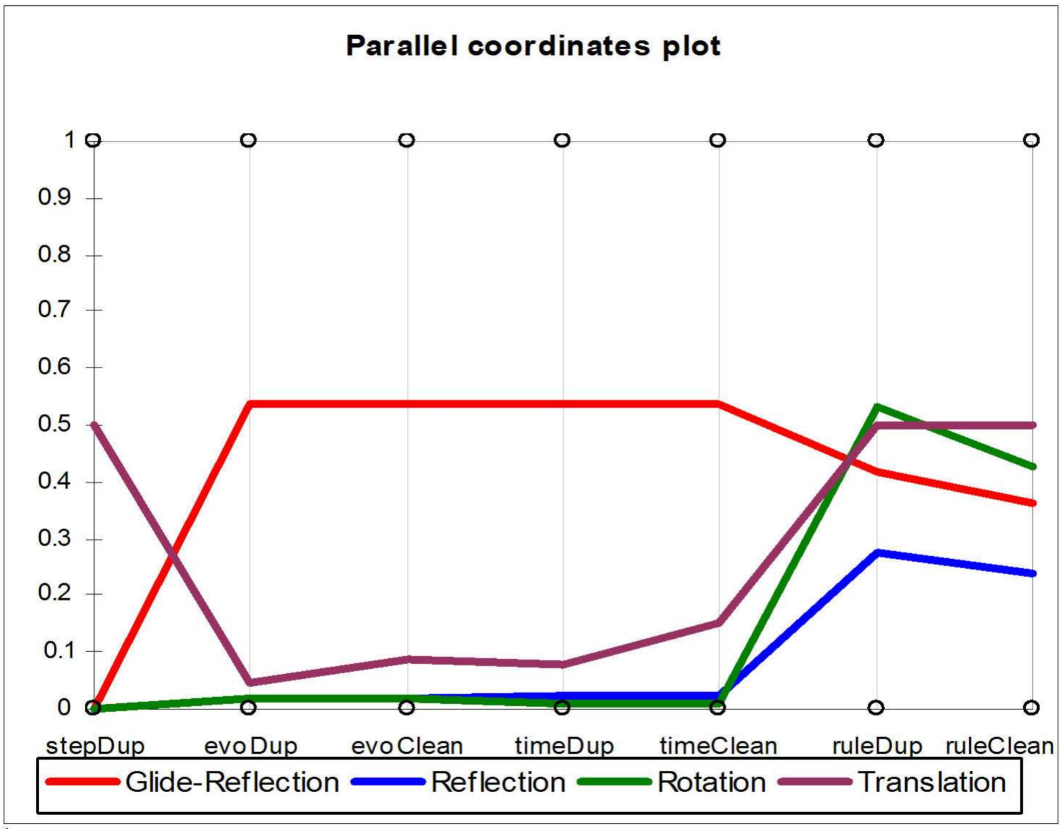
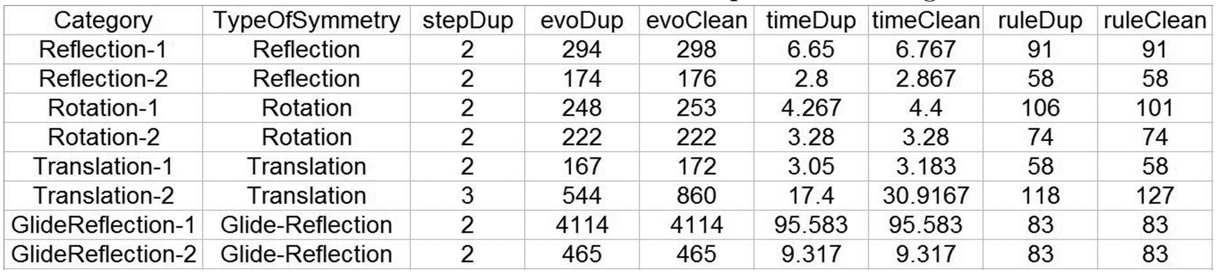


Figure 4.18: Rescaled parallel-coordinates-plot and mean charts visualizing the impact of the presence of four types of symmetry. It seems that glide-reflection often leads to much higher evolutionary cost than other types of symmetry.

Table 4.6: Performance measures of the replicators in Figure 4.16.



Category	TypeOfSymmetry	stepDup	evoDup	evoClean	timeDup	timeClean	ruleDup	ruleClean
Reflection-1	Reflection	2	294	298	6.65	6.767	91	91
Reflection-2	Reflection	2	174	176	2.8	2.867	58	58
Rotation-1	Rotation	2	248	253	4.267	4.4	106	101
Rotation-2	Rotation	2	222	222	3.28	3.28	74	74
Translation-1	Translation	2	167	172	3.05	3.183	58	58
Translation-2	Translation	3	544	860	17.4	30.9167	118	127
GlideReflection-1	Glide-Reflection	2	4114	4114	95.583	95.583	83	83
GlideReflection-2	Glide-Reflection	2	465	465	9.317	9.317	83	83

Again, a rescaled parallel-coordinates-plot is produced from the performance measures, as shown in Figure 4.18. It suggests that on average, evolution and replication cost with the presence of reflection or rotation symmetry is close, if not lower, than with translation symmetry (i.e. repeated sub-structures). However, glide-reflection symmetry, which is a combination of reflection and translation, seems to potentially lead to higher cost. Also note that, the sub-structure in seed translate-1 (third row, left in Figure 4.16) only translates horizontally, while in translate-2 (third row, right in Figure 4.16) the sub-structure translates both horizontally and vertically, which has seemingly caused it to take more time steps and much higher evolutionary cost to self replicate.

4.4 Conclusions

The S-tree/R-tree encoding and replicator factory makes it possible for a broad range of replicating arbitrary structures to be artificially synthesized and self constructed in a systematic fashion. This represents a significant step forward in creating self-replicating structures because it qualitatively reduces the time cost of replicator construction compared to past manual methods. Further, the replicators created in this fashion are qualitatively different from those generated in past studies (universal constructors and loops). Families of replicators demonstrated in this chapter have again suggested that these automatically discovered replicators can duplicate themselves very quickly, in a fission-like or rotational process. Several replicators manage to self-replicate within only one time step, representing a speed which had never been pursued with past replicator models .

Given the large number of self replicating structures the replicator factory has produced, I was able to study the properties of self-replicating systems using GP as the initial configuration's size, shape, symmetry, allowable states, and other factors were systematically varied. It was found that the number of GP generations, computation time, and number of rules required by an arbitrary structure to self-replicate are positively correlated with the number of components, configuration shape, and allowable states in the initial configuration, but inversely correlated with the presence of repeated components or sub-structures, and seed symmetry. These results suggest the properties of the resulting replicators can be predicted in part a priori.

Even though the limited number of experiments and statistical results included in this chapter have revealed many qualitative properties of self-replicating systems, due to the nature of evolutionary computation, it is impossible for this study alone to cover a sufficiently large number of experiments to permit a full spectrum of observations, hence the results presented here merit much further quantitive studies on a broader spectrum of replicators, further leveraging the unprecedented productivity of the replicator factory demonstrated in this chapter.

Chapter 5

Multi-Objective Artificial Evolution toward Task-Performing Self-Replication with Specified Seed Structures

5.1 Overview

As mentioned in Chapters 1 and 2, past studies in self-replication have mainly involved two approaches, the theoretical study of universal constructors and the manual design of simple self-replicating loops. These approaches represent two extremes of the replication complexity scale: on one end there are highly complex but marginally realizable universal constructors, on the other end one finds simple structures that can do nothing but self-replication [65]. It has been realized that a system capable of self replication but not much of anything else would not be very useful. In 1995, Tempesti asked whether it is possible to add additional computational capabilities to simple self replicating loops, hence attaining complex machines that are nevertheless completely realizable [68]. He took the first step in this direction and manually revised a self-replicating loop that resembles Langton's (see Section 2.2.2), but with added capability of attaching an executable program that writes out "LSL", the acronym of the Logic Systems Laboratory. Subsequent studies have also shown that any program, written in a simple yet universal language, can be embedded into self-replicating loops [53]. These works demonstrated that some simple manually-designed self-replicating loops are capable of carrying out some limited "secondary" tasks. However, these past works implementing secondary capabilities all share the following limitations: 1) they are implemented on specific, non-arbitrary, and manually-designed seed structures; 2) they depend on manual, pre-written computer programs; and 3) they are implemented by embedding the pre-written program into the specific seed structure, in other words, at the cost of altering (and changing the complexity of) the seed structure itself.

Previous chapters have presented an approach to creating novel self-replicators, with the use of GP to automatically program cellular automata with arbitrary, initially non-replicating structures. However, it is not clear whether these automatically programed self-replicators are also capable of carrying out a simple secondary task, such as writing out "UM", the acronym for University of Maryland, and if so, whether GP could be used to discover the needed rule sets. Assuming the answer to be yes, there can be seen several possibilities for achieving this. First, after generating a self-replicator with the replicator factory as before, one could pre-write and embed a manual computer program like others, in the generated self-replicating structure. A second possibility is first to generate the self-replicating structure as usual, then subsequently execute a separate GP evolution to automatically program the cellular automata to add the secondary functionality on top of self-replication. A third option is to automatically program the cellular automata toward both self-replication and secondary task performance in parallel with a single GP evolution.

If we could implement the first option, it would prove that self-replicators created by this third approach are at least as sound as the past manually-designed self-replicators, in terms of secondary task performance. On the other hand, if the second option could be implemented, it would indicate not only the novel replicators are as sound as before, but also this new approach has the advantage of overcoming the past limitations listed above. In other words, it would show, given an arbitrary known seed structure, the evolutionary approach can automatically program it toward not only self-replication, but also carrying out a secondary task, without the need for pre-writing and embedding manual programs or altering the complexity of the seed structure itself. Also, this would extend such secondary capabilities to arbitrary seed structures, contrast to the past when only specific seeds were used. Finally, the third option, if possible, would be most ideal, because it would demonstrate everything that the second option can prove, plus it is the least cost, as it also eliminates the need of multiple GP-evolution runs. For this reason, this is the option that this study has chosen to focus on, despite it being apparent that this option would be most challenging at the same time.

Choosing the third option naturally leads to the introduction of multi-objective issues into the replicator factory model. Self-replication and secondary task performance are obviously incommensurable, often competing, objectives. As detailed in Chapter 3, when the CA simulation starts at time t=0, in the entire CA space all cells are quiescent except those active cells forming a single seed structure. When the time proceeds, these active cells can reach more cells, but without exceeding light speed, because of the well accepted CA rule that a quiescent cell fully surrounded by other quiescent cells can only remain quiescent during the following time step. This means with the limited simulation time steps, there is a limited active area for any activity to take place. It follows that self-replication and task performance have to compete for this limited active space. If a duplicated seed structure already occupies an active cell, no other structure can occupy the same cell. On the other hand, since state transitions starting from the same initial configuration are governed by the same rules encoded in a single R-tree, the two objectives have to compete for the rules in the R-tree too. For example, if an R-tree dedicates its rules to producing a configuration in the active area to match a maximum number of seed structures, it obviously gives less chance for distinct structures to occupy the same cells, as required by a secondary task, such as writing out "UM". In Chapter 3 both the evolving R-tree and limited CA space could be dedicated to the optimization of one single objective, i.e., self-replication, with the use of a single fitness function concerning a single S-tree (produced from the seed to be duplicated). In contrast, if we now generalize the realization of a secondary task such as the creation of an arbitrary pre-specified target structure, we would need to introduce a secondary fitness function to measure the performance of the same R-tree in terms of creating the target structure. This secondary fitness function will incorporate a different S-tree, one representing the target structure, instead of the seed structure.

The introduction of distinct objectives and multiple fitness function components brings a new level of complexity to the GP based replicator factory model. What is being sought now is the automated programing of CA rules that produce diversified global behaviors that have to compete and coordinate at the same time, in a shared time and space. In Chapter 3, a single fitness function (see Section 3.4.4) was developed to allow the population to approximate to a single global optimum.

In contrast, now balancing between proximity and diversity becomes the key. To address that, we have to extend the replicator factory model with diversity preserving methods, to assign fitness measures based on multiple S-trees, to enable R-tree tournament selection based on multiple fitness criteria, and to support multi-objective R-tree elitism, as detailed in the following sections.

5.2 Problem Formulation

Let r represent an evolving R-tree, s represent an S-tree for a given structure, p represent an infinite cellular space, and t represent a time step applying r on s in p. As in Chapter 3, define function *δ(r,s,p,t)* to be, after applying r on s recursively in p for t time steps, the number of instances of isolated structures found in p that match s.

Further, let e represent an S-tree for a given "target structure", i.e., a CA configuration (contiguous non-quiescent cells) separate from a self-replicating structure. Define function *η(r,s,e,p,t)* to be, after applying r on s in p recursively for t time steps, the number of instances of isolated structures found in p that match e.

Define a Boolean function *β_δ(r,s,p,t)* as:

$$\beta_\delta(r,s,p,t) = \left\{ \begin{array}{ll} 1, & \text{if } \delta(r,s,p,t) \geq 2 \\ 0, & otherwise \end{array} \right\}, \tag{5.1}$$

and a second Boolean function *β_η(r,s,e,p,t)* as follows:

$$\beta_\eta(r,s,e,p,t) = \left\{ \begin{array}{ll} 1, & \text{if } \eta(r,s,e,p,t) \geq 1 \\ 0, & \textit{otherwise} \end{array} \right\} \tag{5.2}$$

For a time step t', if $\beta_\delta(r,s,p,t') = 1$ and $\beta_\eta(r,s,e,p,t') = 1$, or,

$$(\beta_\delta(r,s,p,t') \times \beta_\eta(r,s,e,p,t')) = 1 \tag{5.3}$$

t' is said to be a sustained time step. In words, by the definition of β_δ and β_η above, a sustained time step is a CA time whenever at least 2 instances of the isolated seed structure and at least one instance of the isolated target structure are found in the CA space.

With the definitions above, we can now formally state the problem as follows:

Given a cellular space p, a seed structure s, an extra structure e, we want to find an R-tree r, which will satisfy the following goals:

1. There exists a time step T_j, for any positive integer j, so that

$$\sum_{t=1}^{T_j} \delta(r,s,p,t) \geq j \tag{5.4}$$

   Note this equation requires that, given enough time steps, we can obtain any desirable number of replicated seed structures. Let's call this the *replicatability* objective.

2. There exists a time step T_k, for any positive integer k, so that

$$\sum_{t=1}^{T_k} \eta(r,s,e,p,t) \geq k \tag{5.5}$$

Note this equation requires that, given enough time steps, we can also obtain any desirable number of pre-specified target structures. Let's call this the *taskability* objective.

3. There exists a time step T_i, for any positive integer i, so that

$$\sum_{t=1}^{T_i}(\beta_\delta(r,s,p,t) \times \beta_\eta(r,s,e,p,t)) \geq i \tag{5.6}$$

Note this equation requires that, given enough time steps, we can find any desirable number of sustained time steps. Let's call this the *sustainability* objective.

5.3 R-tree Fitness Assignment with Multiple S-trees

When the GP evolution starts, R-tree initialization and simulation are carried out in exactly the same way as described in Sections 3.4.2 and 3.4.3 respectively. Section 3.4.4 developed a fitness assignment method based on a single S-tree, where probes exploiting the complete structural information encoded by the given S-tree are tested at all possible locations with all possible orientations, so that the probes producing the best results can be identified and accepted. The number of acceptable probes at each time step is dynamically and adaptively determined in order to avoid the over-probing problem. Most of this methodology can be adopted here, except that we need to address the following differences:

1. Because our goal now is to program the CA to allow both the seed structure

and target structure to occupy the active space at the same time, we will need to perform two types of probing. One uses the seed S-tree to detect configurations matching the seed structure, and the other uses the target S-tree to detect configurations that match the target structure. The former is hereafter referred as *seed probing*, and the latter as *target probing*.

2. Over-probing can now happen not only by accepting too many seed probes, but also by accepting too many target probes, because both appear in same CA space. They are referred as *seed over-probing* and *target over-probing* hereafter.

3. A new problem, which is hereafter referred as *speciation*, can now happen if a relatively high number of accepted seed probes dominate the limited space and prevent target structure from being formed, or vice versa.

4. The order of seed probing vs. target probing becomes important, because any accepted probes will mark the traversed active cells "UNAVAILABLE" to subsequent probes, and thus potentially impact their results.

5.3.1 Determining Acceptable Seed Probes and Target Probes

In order to avoid seed over-probing, target over-probing, and speciation problems, we can modify Strategy 3.1 described in Section 3.4.4.2 as follows:

Strategy 5.1: An evolving R-tree, r, shall start with an initial goal that probes and finds the starting sustained time step $\tilde{t}$, when $\beta_\delta(r,s,p,\tilde{t})\times\beta_\eta(r,s,e,p,\tilde{t})) > 1$, and that only when that R-tree succeeds in finding $\tilde{t}$ can it start to become more aggressive in attempting to accept more seed or target probes. The better it performs in previous evaluation time steps, the faster it can accept more seed and target probes and produce more sustained time steps.

This strategy says that an evolving R-tree can start with a basic goal, programing itself to find a minimum CA space which contains 2 isolated seed structures and 1 target structure. Until this is achieved, it can accept neither more seed probes nor target probes. On the other hand, once the current goal is achieved, it can gradually raise its goal by adaptively adjusting the number of acceptable seed probes (π_{δ}^{t}) and target probes (π_{θ}^{t}) at a coordinated and controlled pace. This process can be illustrated by the following pseudo-code:

```
for R-tree $r$, seed S-tree $s$, target S-tree $e$, and cellular space $p$ {
  $\pi_{\delta}^{0} = 2;$
  $\pi_{\eta}^{0} = 1;$
  at each time step $t > 0$ {
    if $(\delta(r, s, p, t-1) = \pi_{\delta}^{t-1}$ and $\eta(r, s, e, p, t-1) = \pi_{\eta}^{t-1})$ {
      $\pi_{\delta}^{t} = \pi_{\delta}^{t-1} + 1;$
      $\pi_{\eta}^{t} = \pi_{\eta}^{t-1} + 1;$
    }
  }
```

This pseudo-code says that at t=0, the number of acceptable seed probes and target probes is 2 and 1 respectively. At any subsequent time step, if each of these accepted probes finds a perfectly matching structure (e.g., initially, finding 2 isolated seed structures and 1 isolated target structure), the number of acceptable seed probes and target probes can each be increased by 1. Otherwise, they remain the same. Whenever these numbers are adjusted, the new numbers have to be realized, before they can be adjusted again.

Typically, during the early phase of evolution, no R-tree can make any sustained time steps, so no R-tree can raise its goal at any time step. Later, once an R-tree succeeds in making the first sustained time step (i.e., making two perfect isolated seed structure and one isolated target structure), it is allowed to accept 3 seed probes and 2 target probes at the subsequent time step. Then, it can raise these numbers again only when it achieves the said numbers of structures.

5.3.2 Interlaced Acceptance of Seed Probes and Target Probes

As pointed out earlier, the order of seed probing vs. target probing becomes important, because any accepted probes will mark the traversed active cells "UNAVAILABLE" to subsequent probes, and thus potentially impact their results. Based on last section, at time step t, we are allowed to accept π_δ^t (such as 2) seed probes and π_η^t (such as 1) target probes. But how shall we do that? First perform seed probing like in Section 3.4.4.1 and accept the allowed number of best seed probes before performing the target probing, or vice versa, or intermix in certain ways? Either way will obviously introduce human bias into the evolution. To avoid such bias and achieve optimal result, the following strategy is added:

Strategy 5.2: Seed probing and target probing shall be conducted and accepted in an intermixed and interlaced fashion, in an order which is automatically, adaptively, and dynamically determined, so that a more promising probe, regardless of being a seed probe or target probe, can always be accepted before a less promising one, given the fixed number of acceptable seed probes and target probes of a considered R-tree at a given time step.

This strategy says, the order of interlaced probing and probe acceptance can not be arbitrary, and shall not be pre-determined by a person. They shall be conducted in an adaptive order so that a less promising probe can never be accepted in favor of a more promising one, regardless of its type. The following pseudo-code implements such an interlaced probing algorithm:

```
SeedProbesToBeAccepted = $\pi_\delta^t$;
TargetProbesToBeAccepted = $\pi_\eta^t$;
while (SeedProbesToBeAccepted > 0 or TargetProbesToBeAccepted > 0) {
// if there are more seed or target probes to be accepted,
    bestProbe = null; bestResult = 0.0;
    for each cell, $c \in p$ {
        if $c$ is not active, go to next cell; // skip
        if $c$ is marked unavailable, go to next cell; // skip
        // Otherwise, conduct a probe from current cell, with the seed S-tree (4 phases)
        currProbe = probe($s$, $c$);
        // get the scalar measure for the current probe
        currResult = currProbe.result();
        if(currResult $\geq$ 1.0) {
            // perfect match, accept immediately as a seed probe
            bestProbe = currProbe; // store the current one
            break; // no need to probe other cells, exit for
        }
        // did not find a perfect seed probe from this cell, let's try to probe it as
        // a target structure instead, and compare which yields better matching
        // so, conduct a probe from current cell, with the target S-tree (4 phases)
        targetCurrProbe = probe($e$, $c$);
        // get the scalar measure for the current probe
        targetCurrResult = targetCurrProbe.result();
        if(targetCurrResult $\geq$ 1.0) {
            // perfect match, accept immediately as a target probe
            bestProbe = targetCurrProbe; // store the current one
            break; // no need to probe other cells, exit for
        }
        // at this point, did not find a perfect seed probe nor a target probe from this cell
        // so determine if keep it as a best seed probe or target probe, or throw away:
        if(currResult > bestProbe.result()) {
            // if found as a better seed probe, store
            bestProbe = currProbe; // store the current one
            probeAcceptType = seed; // track if it wins as a seed probe or target probe
        }
        if(targetCurrResult > bestProbe.result()) {
            // if found as a better target probe, store
            bestProbe = currProbe; // store the current one
            probeAcceptType = target; // track if it wins as a seed probe or target probe
        }
    } // end of for
    // at this point all cells and all orientations have been probed,
    // best probe identified, either as a seed or a target probe, now accept it.
    if(probeAcceptType == seed) { //accept as a best seed probe
        bestProbe.accept($s$); // this also marks the traversed cells as unavailable
```

```
            //deduct the number of acceptable seed probes
            SeedProbesToBeAccepted = SeedProbesToBeAccepted - 1;
        } else if (probeAcceptType == target) { //accept as a best target probe
            bestProbe.accept($e$); // this also marks the traversed cells as unavailable
            //deduct the number of acceptable target probes
            TargetProbesToBeAccepted = TargetProbesToBeAccepted - 1;
        }
        // go for next acceptable seed or target probe, if still any
    } //end of while
```

This pseudo-code suggests probing each location and orientation both as a seed probe and target probe. If one probe results in a scalar measure of 1.0, immediately accept that probe (also marking the traversed cells as "UNAVAILABLE"), because obviously no other future probes (seed or target) can yield a better matching. On the other hand, if a probe results in a partial matching (a scalar measure less than 1.0), then check if the current probe is better than previous ones. If not, throw it away. If yes, store the current probe as the current best probe. After each available cell and orientation has been probed, accept the stored best probe, either as a seed probe or target probe, and also mark the traversed active cells "UNAVAILABLE" to subsequent probings. Repeat this process until there are no more acceptable seed probes or target probes. Note that, as soon as one considers a new R-tree, or a new evaluation time step, this whole process starts over with all of the marked cells un-marked and available again.

5.3.3 Fitness Function for Replicatability

Section 5.3.1 describes how the number of acceptable seed and target probes can be adaptively determined for each R-tree at each time step, and Section 5.3.2 describes how the acceptable seed and target probes can be identified and accepted adaptively in an interlaced fashion, in order to avoid over-probing, speciation, and human bias. Like in Section 3.4.4.1, let r represent an simulated (evaluated) R-tree, s represent an S-tree for a given seed structure, p represent an infinite cellular space, $\bar{c} \in p$ represent a root cell being probed, $h \in$ (1,2,3,4) represent the phase of the current probe, and t represent a time step applying r on s in p. Define function $\kappa_\delta(r,s,p,t,\bar{c},h)$ to be, after applying r on s in p recursively for t time steps, then probing s from $\bar{c}$ in phase h, the number of traversed cells (those not marked "UNAVAILABLE") which match the state of the corresponding node (active or quiescent) as guided by s, and let $\breve{p}_i$ represent the set of cells marked as "UNAVAILABLE" by a previously accepted probe i, we can then write the replicatability fitness function as follows:

$$f_\delta(r) = \sum_{t \in \mathcal{T}^v} (\sum_{n=1}^{n=\pi_\delta^t(r)} (\max_{c \in p,\ c \neg\in \bigcup_{m=1}^{m=n-1} \breve{p}_m} (\max_{h \in (1,2,3,4)} (\frac{\kappa_\delta(r,s,p,t,c,h)}{\lambda(s)})))). \tag{5.7}$$

This equation says that the replicatability fitness of an evolving R-tree is the accumulated result of every accepted seed probe at every evaluation time step. Every accepted seed probe finds a best probe among tested seed probes at every location and every orientation, given the remaining cells that are not marked as "UNAVAILABLE". Obviously an R-tree can now gain a higher replicatability fitness value by either earning a higher number of accepted seed probes, or better results produced from individual accepted seed probes, or both.

5.3.4 Fitness Function for Taskability

In addition to all of the definitions in Section 5.3.3 above, let e represent an S-tree for a given target structure, and define function κ_η*(r,s,p,t,$\bar{c}$,h,e)* to be, after applying r on s in p recursively for t time steps, then probing e from $\bar{c}$ in phase h, the number of traversed cells (those not marked "UNAVAILABLE") which match the state of the corresponding node (active or quiescent) as guided by e. Then based on accepted target probes, we can write the taskability fitness function as follows:

$$f_\eta(r) = \sum_{t \in \mathcal{T}^v} (\sum_{n=1}^{n=\pi_\eta^t(r)} (\max_{c \in p,\ c \neg\in \bigcup_{m=1}^{m=n-1} \breve{p}_m} (\max_{h \in (1,2,3,4)} (\frac{\kappa_\eta(r,s,p,t,c,h,e)}{\lambda(e)})))). \qquad (5.8)$$

This equation says that the taskability fitness of an evolving R-tree is the accumulated result of every accepted target probe at every evaluation time step. Every accepted target probe finds a best probe among tested target probes at every location and every orientation, given the remaining cells that are not marked as "UNAVAILABLE". Obviously an R-tree can now gain a higher taskability fitness value by either earning a higher number of accepted target probes, or better results produced from individual accepted target probes, or both.

5.3.5 Fitness Function for Sustainability

As pointed out in Section 5.2, a sustained time step is an evaluation time step $\tilde{t} \in \mathcal{T}^v$, at which at least two perfectly matching seed probes and at least 1 perfectly matching target probe are accepted, i.e., $\beta_\delta(r, s, p, \tilde{t}) \times \beta_\eta(r, s, e, p, \tilde{t})) > 1$. As shown in Section 5.3.2, the intermixed probing algorithm provides the capability for identifying such perfectly matching seed or target probes during the probing and fitness evaluation process. Hence, we can easily track the number of such accepted and perfectly matching probes at each evaluation time step, for a given R-tree, r. Accordingly, we can easily derive the value of f_θ, which can be denoted as:

$$f_\theta(r) = \sum_{t \in \mathcal{T}^v} (\beta_\delta(r, s, p, t) \times \beta_\eta(r, s, e, p, t)) \tag{5.9}$$

This equation says, the sustainability fitness of an evolving R-tree is the accumulated number of sustained time steps out of the total number of evaluated time steps, which (the latter) is common among all R-trees in a given GP generation.

5.4 R-tree Tournament Selection Based on Multiple Criteria

An essential idea behind the GP paradigm is to let better R-trees have a better chance to be selected into the mating pool and reproduce. In Chapter 3, the meaning of "better" was un-ambiguous, because every R-tree can be fully ranked based on a single fitness measure. In contrast, now each R-tree receives three fitness measures, replicatability f_δ, taskability f_η, and sustainability f_θ, as shown above. Since now different fitness measures can rank the R-trees in different orders, we need to re-define the meaning of "better". An R-tree is said to be "better" than another R-tree if it is not worse in any objective and at least better in one objective. In such a case a superior R-tree $\bar{r}^*$ is also said to *dominate* an inferior one ($\bar{r}$), i.e., formally, an R-tree $\bar{r}^* \in \mathcal{R}$ is said to *dominate* another R-tree $\bar{r} \in \mathcal{R}$, denoted as $\bar{r}^* \succ \bar{r}$, if and only if:

$$\forall i \in \{\delta, \eta, \theta\} : (f_i(\bar{r}^*) \geq f_i(\bar{r})) \tag{5.10}$$

and,

$$\exists i \in \{\delta, \eta, \theta\} : f_i(\bar{r}^*) > f_i(\bar{r}) \tag{5.11}$$

Likewise, an R-tree is said to be "worse" than another R-tree if it is not better in any objective and at least worse in one objective. In such a case the inferior R-tree ($\bar{r}^*$) is also said to be *dominated* by the superior one ($\bar{r}$), denoted as $\bar{r}^* \prec \bar{r}$. If an R-tree is not better, nor worse, than another R-tree, such R-trees are said to be "indifferent". This is shown in Figure 5.1.

Further, an R-tree $\bar{r}^* \in \mathcal{R}$ is said to be *non-dominated*, if and only if it is not dominated by any other R-tree in the search space, i.e.,

$$\neg \exists \bar{r} \in \mathcal{R} : \bar{r} \succ \bar{r}^* \tag{5.12}$$

As shown in Figure 5.1, there can be multiple instances of non-dominated R-trees. Each of these non-dominated R-trees is different from the others, and each represents a best trade-off among replicatability, taskability, and sustainability.

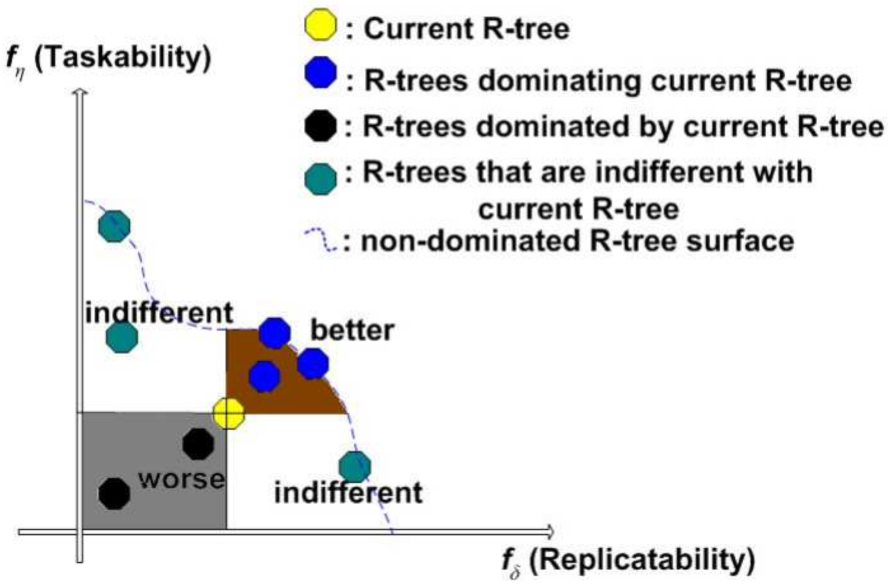

: Current R-tree
: R-trees dominating current R-tree
: R-trees dominated by current R-tree
: R-trees that are indifferent with current R-tree
: non-dominated R-tree surface
f_η (Taskability)
indifferent
better
worse
indifferent
f_δ (Replicatability)

Figure 5.1: With respect to the current R-tree shown in yellow, R-trees in the brown area dominate the current R-tree, the ones in the gray area are dominated by the current R-tree, and the one in white areas are indifferent.

5.4.1 Domination Pressure

Using the definition above, given the multiple fitness criteria, an R-tree can be either "better", "worse", or "indifferent" than another R-tree. However, we still need to derive a scalar value from these domination relationships so that we can use this scalar value to fully rank the population and facilitate the tournament selection algorithm (detailed in Section 3.4.6). The concept of *domination pressure*, denoted as $\phi(\bar{r})$, is thus introduced here. The more an R-tree is dominated by other R-trees, the higher domination pressure it receives, and hence, the less chance it has to reproduce, and vice versa. $\phi(\bar{r})$ is defined as follows:

$$\phi(\bar{r}) = \sum_{\bar{r}_i \in \mathcal{R},\ \ \bar{r}_i \succ \bar{r}} \tau(\bar{r}_i), \tag{5.13}$$

where,

$$\tau(\bar{r}) = |\, \{\bar{r}_j \mid \bar{r}_j \in \mathcal{R} \ \wedge \ \bar{r} \succ \bar{r}_j\} \,| \tag{5.14}$$

Therefore, $\tau(\bar{r})$ represents the total number of other R-trees that R-tree $\bar{r}$ dominates in the population, and $\phi(\bar{r})$ represents the sum of $\tau(\bar{r}^*)$ of all R-trees $\bar{r}^*$ that dominates $\bar{r}$. By this definition, a domination pressure zero ($\phi(\bar{r}) = 0$) means $\bar{r}$ is not dominated by any other R-trees. On the other hand, a high domination pressure means an R-tree is being dominated by many other R-trees (which in turn, dominate many other R-trees). This allows a global criteria to be used in the selection operator without using an aggregation of the absolute values of any individual objective measures.

5.4.2 Distribution Pressure

Domination pressure, as explained above, can be used to drive the R-tree evolution toward good trade-offs among replicatability, taskability, and sustainability. However, as pointed out in the overview section of this chapter, it is critical to keep a good balance between proximity and diversity. Domination pressure helps to address proximity, but not diversity. In order to promote R-trees with greater diversity to have a better chance to reproduce, a second pressure is hereby introduced, namely the *distribution pressure*, as follows.

The idea is to impede R-trees that sample points close to each other into the mating pool at the same time, and hence effectively promote R-trees that are more distributed apart to enter the mating pool. The higher density an R-tree is surrounded by neighboring peers, the less that R-tree shall be selected into the mating pool, and vice versa. Thus, the distribution pressure can be defined as inversely proportional to the density of neighboring peers, which can be estimated with distance to the k-th closest neighbor of each R-tree. This can be done as follows. First, the distance between two R-trees, $\bar{r}$ and $\bar{r}^*$, can be obtained by calculating the Euclidean metric of the fitness vector:

$$\sigma(\bar{r}, \bar{r}^*) = \sqrt{(f_\delta(\bar{r}^*) - f_\delta(\bar{r}))^2 + (f_\eta(\bar{r}^*) - f_\eta(\bar{r}))^2 + (f_\theta(\bar{r}^*) - f_\theta(\bar{r}))^2} \qquad (5.15)$$

Let the population be L. For a candidate R-tree, $\bar{r} \in L$, if we calculate its distance to every other R-tree in L, we get a list of size $(|L| - 1)$. Then sort the list. The k-th value in the sorted list provides a scalar density estimate $(\alpha_{\bar{r}}^k)$ of R-tree $\bar{r}$, where normally $K = \sqrt{|L|}$. The distribution pressure $\psi(\bar{r})$ of R-tree $\bar{r}$ can then be defined as:

$$\psi(\bar{r}) = \frac{a_1}{\alpha_{\bar{r}}^k + b_1}, \tag{5.16}$$

where a_1 and b_1 are configurable constants.

It shall be pointed out that the way the domination pressure and distribution pressure for the R-tree population are calculated here is very much influenced by the fine grained fitness assignment methods used in SPEA2 [84], an existing multi-objective evolutionary algorithm, reviewed in Chapter 2.

5.4.3 Parsimony Pressure

The introduction of domination pressure and distribution pressure in the tournament selection helps to keep a good balance between proximity and diversity. However, for an R-tree to attempt to minimize the domination and distribution pressures at the same time, it can potentially expand itself easily and become very large. Large R-trees tend to consume more computation to evolve, slow down the evolution speed, and reduce the overall performance. Hence, a third controlling pressure, namely, the *parsimony pressure*, $\omega(\bar{r})$, is introduced in order to keep each R-tree as parsimonious as possible. The parsimony pressure is defined as positively proportional to the size of an R-tree $\bar{r}$ (number of nodes in the tree, $\mu(\bar{r})$):

$$\omega(\bar{r}) = a_2\mu(\bar{r}) + b_2, \tag{5.17}$$

where a_2 and b_2 are configurable constants.

In sum, we now can define an aggregated pressure, $\varrho(\bar{r})$,

$$\varrho(\bar{r}) = \omega(\bar{r}) + \psi(\bar{r}) + \phi(\bar{r}), \tag{5.18}$$

so that the tournament selection can be conducted exactly like before (Section 3.4.6), based on $\varrho(\bar{r})$, and hence drive the GP evolution toward diversified optimal trade-offs among replicatability, taskability, and sustainability, while keeping the R-trees as parsimonious as possible.

5.5 Multi-Objective R-tree Elitism

After the multi-objective fitness measurement and diversity preserving tournament selection described above, R-trees can reproduce with the same genetic operators as before, such as crossover and mutation (Section 3.4.7). As a result, it is expected that the offspring R-trees are also diversified. This means we might find a set of R-trees that are not dominated by any other R-trees in the new candidate population. Each of these R-trees represents a currently found best trade-off among replicatability, taskability, and sustainability, and therefore none shall be discarded in favor of another. In Section 3.4.8, we described the R-tree elitism built in the replicator factory, such that the very best or “elite” R-tree can not be expelled from the population in favor of worse individuals. Because now each R-tree in the non-dominated set can be qualified as an “elite” R-tree, we need to enhance the elitism implementation to accommodate the need to prevent each of these elite R-trees from being lost. A common way to deal with this problem in other multi-objective evolution is to maintain a secondary population, the so-called archive, which acts like as an external storage separate from the evolving population [107]. Zitzler and many others have reported that including the members in the archive population in the selection process can significantly improve evolutionary multi-objective search [83]. Consequently, in this study I decided to adopt such a method and introduce a second population into the multi-objective replicator factory, to which the non-dominated R-trees in the original population are copied at each generation. The new population is referred to as the *elite population*, the original population the *optimization population*, and the union of both the *union population*.

5.6 The Resulting Extended Multi-Objective Replicator Factory Model

A schematic view of the extended multi-objective replicator factory model is illustrated in Figure 5.2. In the beginning, the optimization R-tree population is randomly initialized and the elite population is empty. Then, each R-tree in the optimization population is simulated in the given cellular space within the evaluation time steps, and each individual objective fitness measures is derived from the simulation results as described in Section 5.3.3, 5.3.4, and 5.3.5 respectively. Next, based on these multi-objective fitness measures, the domination relationship between each pairs of R-trees is determined. During the first generation, this is done among the R-trees in the optimization population only, not the empty elite population. In subsequent generations, however, it is done among all R-trees in the

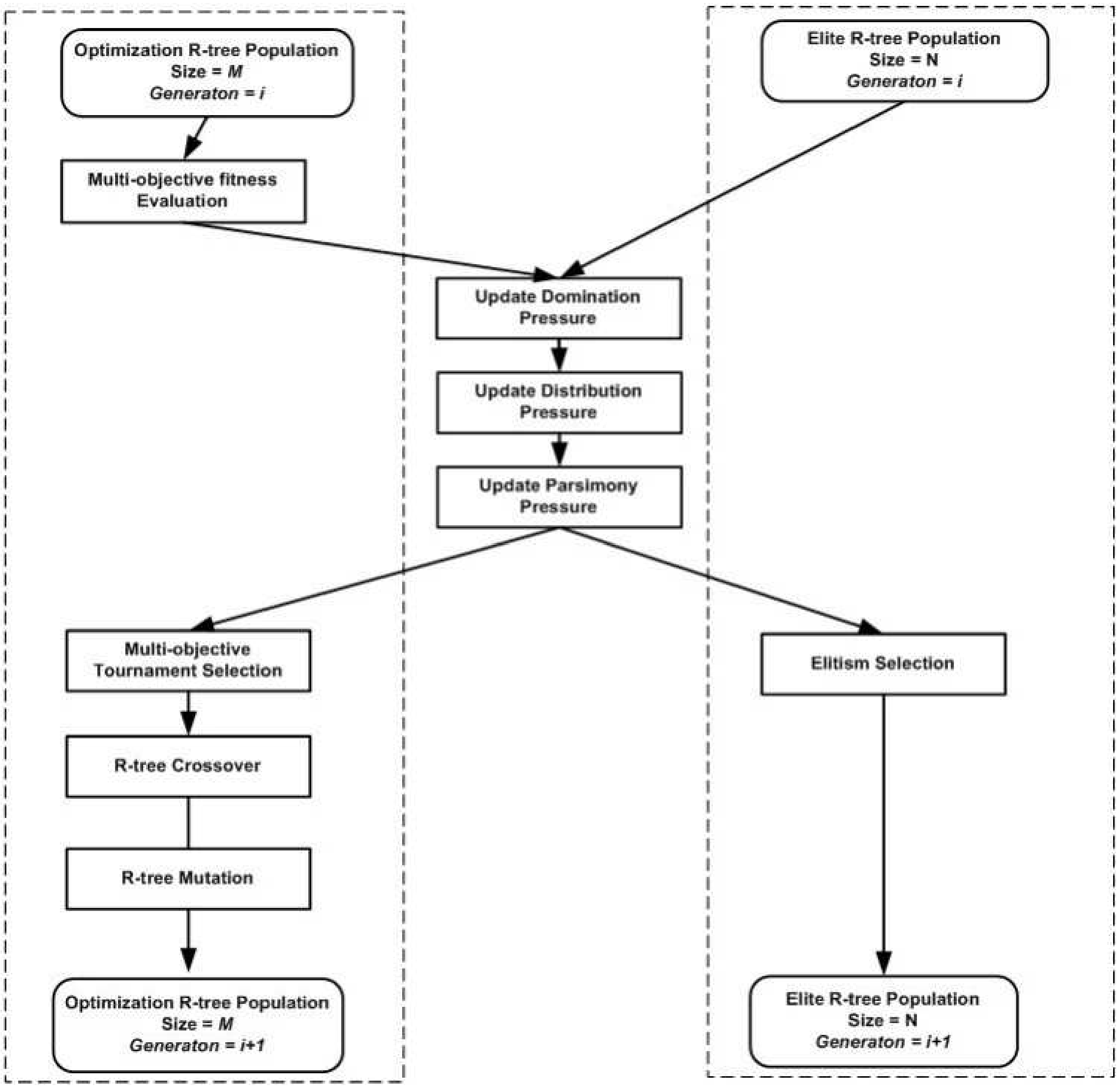

Optimization R-tree Population
Size = *M*
Generaton = i
Multi-objective fitness Evaluation
Elite R-tree Population
Size = N
Generaton = i
Update Domination Pressure
Update Distribution Pressure
Update Parsimony Pressure
Multi-objective Tournament Selection
R-tree Crossover
R-tree Mutation
Optimization R-tree Population
Size = *M*
Generaton = i+1
Elitism Selection
Elite R-tree Population
Size = N
Generaton = i+1

Figure 5.2: A schematic view of the multi-objective replicator factory model.

union population. Then, based on the domination relationship derived from the multiple fitness measures, the domination pressure for each R-tree is calculated in the scope of the union population, according to Section 5.4.1 above. This means, for example, that an R-tree with zero domination pressure is not dominated by any R-tree in the optimization population as well as in the elite population. Likewise, the distribution pressure and parsimony pressure of each R-tree in the scope of the union population are updated, based on the details given in Section 5.4.2 and 5.4.3 respectively.

As shown in Figure 5.2, based on the updated three kinds of pressures, two forms of selection operations are then performed on top of the union population. The first is the mating selection which picks R-trees from the union population using the tournament selection algorithm to form a mating pool. In this tournament selection algorithm (Section 5.4), the probability of an R-tree being picked is inversely proportional to the total pressure that it currently faces, i.e., a sum of its domination pressure, distribution pressure, and parsimony pressure. As a result, the mating pool tends to pick parents from the union population, which are more promising to generate better trade-offs among replicatability, taskability, and sustainability, while keeping the R-trees as diversified and parsimonious as possible. Once the mating pool is formed, the same R-tree genetic operators established in the original replicator factory model can be used to produce a new generation of the optimization R-tree population. This includes the probability based R-tree crossover and mutation operations. Also shown in Figure 5.2, the second kind of selection operation is the elitism selection, which simply copies the R-trees that have zero domination pressure into the elite population.

Once both of the optimization and elite population are updated as described above, the multi-objective replicator factory enters a new generation, and repeats the same process shown above, until either the maximum number of generations has reached, or the user decides the elite population has reached one or more R-trees that provide satisfactory results in terms of replicatability, taskability, and sustainability.

5.7 Task-Performing Self-Replication: Experimental Results

The extended multi-objective replicator factory model described in the previous sections has been implemented with C++, and executed on an IBM T42 ThinkPad laptop. Various combinations of seed structures, target structures, and other configuration items have been tested. I have found that the new model can indeed successfully synthesize diversified R-trees that enable not only self-replication, but also performing simple secondary tasks, such as writing out "UM", the acronym for University of Maryland. Both the seed structure and the target structure to be "manufactured"' by the seed structure can be fairly arbitrary, with the complexity of the extra structure being allowed to significantly exceed that of the seed structure itself. Nevertheless, both self-replication and target structure construction can carry on continuously and sustainably, i.e., any number of duplicated seed structures or target structures can be obtained given enough space and time. The following are some examples of such experimental results. The following parameters were used:

Elite Population size = 10, Optimization Population Size = 90, R-tree Mutation Probability = 0.42, R-tree Crossover Probability = 0.88, R-tree GP Tournament Size = 2, and Max Hesitation = 200.

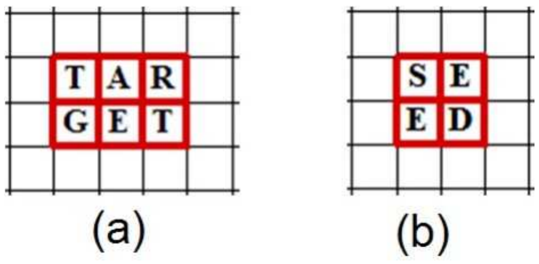
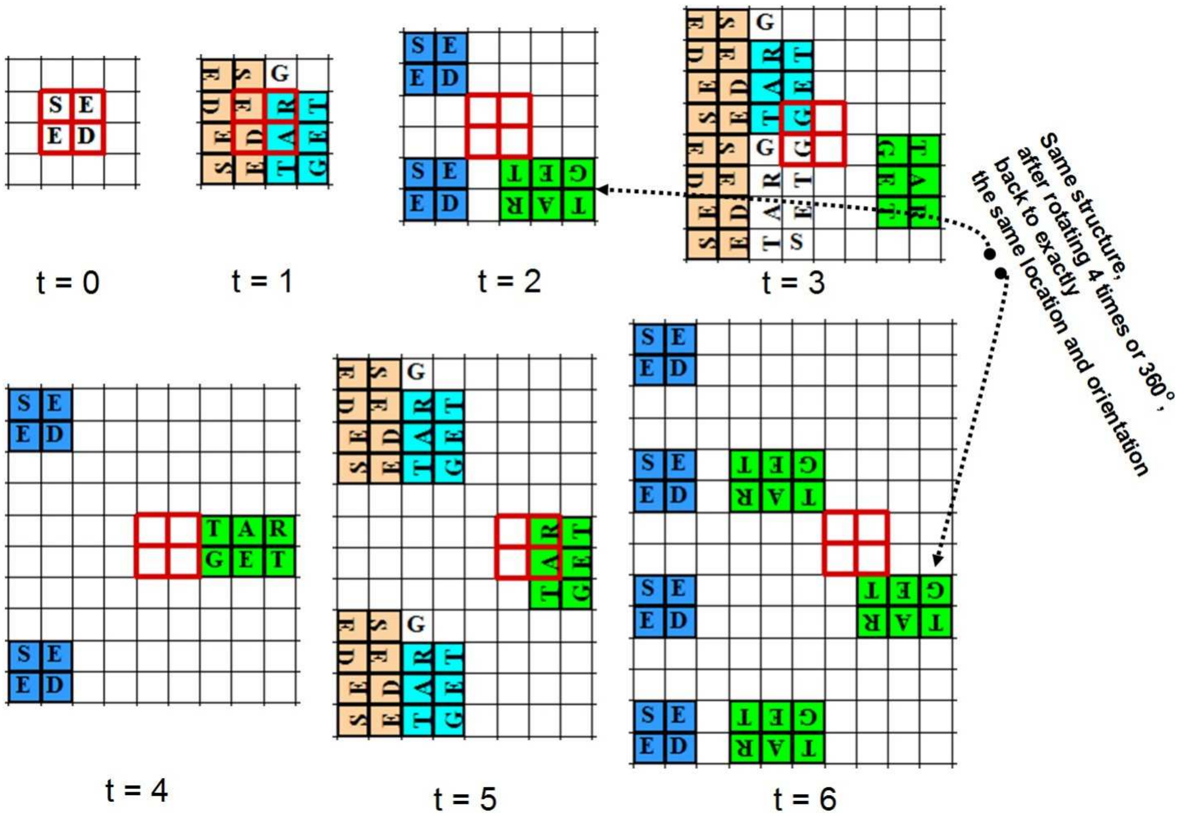


Figure 5.3: Secondary task performance, example 5.1: (a)the pre-specified target structure, which is a 6-component 29-state weak-rotational-symmetric structure, (b)the pre-specified seed structure, which is a 4-component 29-state weak-rotational-symmetric structure. For illustrative purpose, the seed structure is chosen to spell the English word "seed", and the target structure is likewise chosen to spell the English word "target". An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., "manufacture" the target structure as it replicates.

The first example presents a pre-specified seed structure shown in Figure 5.3(b), which is a 4-component (2x2) 29-state weak-rotational-symmetric structure, and also a pre-specified target structure shown in Figure 5.3(a), which is a 6-component (3x2) 29-state weak-rotational-symmetric structure. Each component in the seed and target structure is oriented, and each cell in the cellular space can be either the quiescent state, or one of the 7 oriented components (hence totally 7 x 4 + 1 = 29 states/cell). Note that for illustrative purpose, the seed structure is chosen to spell the English word "seed", and the target structure is likewise chosen to spell the English word "target". Also note that the "target" is more complex than the "seed". An R-tree is sought by the multi-objective replicator factory to program the given seed structure to self-replicate, and perform the given secondary task at the same time.



t = 0
t = 1
t = 2
t = 3
t = 4
t = 5
t = 6
Same structure, after rotating 4 times or 360°, back to exactly the same location and orientation

Figure 5.4: Secondary task performance, example 5.1-1: executing first automatically programmed R-tree against the given seed structure from t=0 to t=6. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

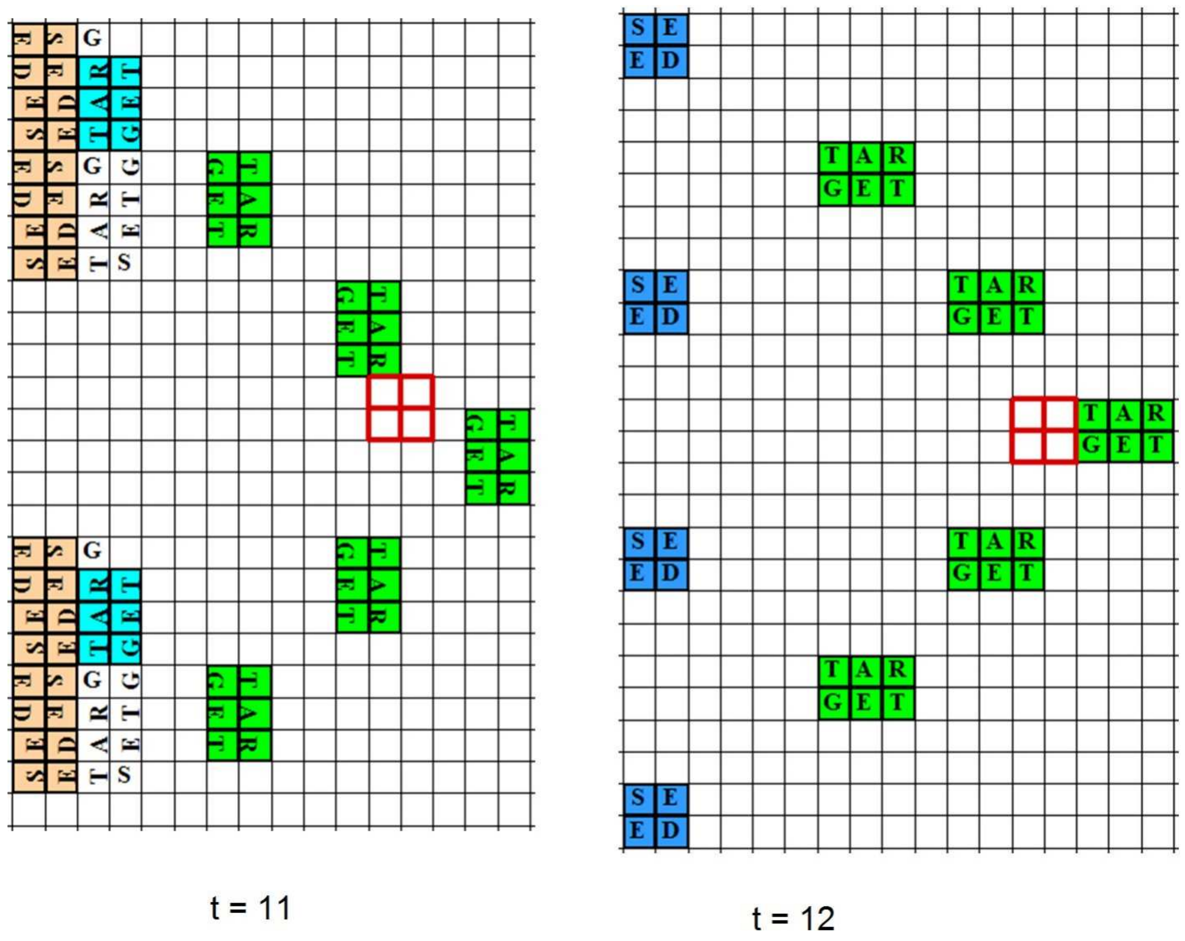


Figure 5.5: Continuation of Figure 5.4 from t=11 to t=12.

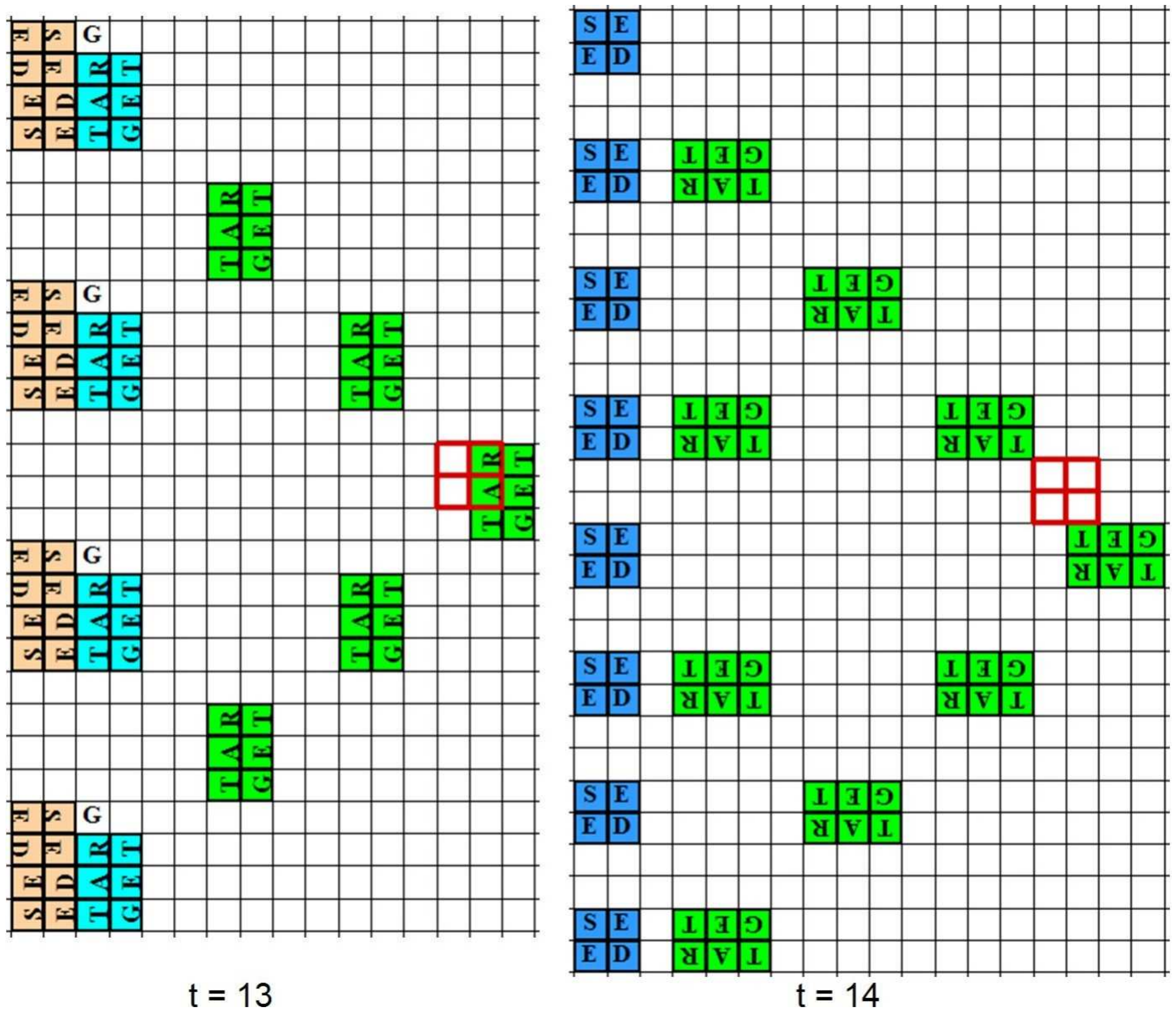


Figure 5.6: Continuation of Figure 5.5 from t=13 to t=14.

The execution results of two of the automatically programmed diversified R-trees with the given target and seed structures are shown from Figure 5.4 to 5.6 and from Figure 5.7 to 5.11 respectively. To make it easy to visualize the produced structures at each evaluation time step, color codes are used in these figures. A non-isolated seed structure is marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. Also, to provide location correlation, the cells covered by the initial seed structure are always highlighted by red edges at any time step. The same visualization convention is used hereafter. For example, in Figure 5.4, the initial configuration at t=0 is shown with the seed cells highlighted with red edges. These highlighted cells are referred as *seed cells* hereafter.

From example 5.1-1 (Figure 5.4), we can observe the following: 1) the configuration expands from the initial seed cells at light speed simultaneously to the left, top, and bottom, but not to the right. Consequently, it can be seen at t=14 (Figure 5.6, right) there are precisely 14 rows above the seed cells, 14 rows below the seed cells, and 14 columns to the left of the seed cells, that contain active cells. 2) As the expansion occurs, it forms a growing vertical "fission-wall" on the left edge, which repeatedly splits into seed replica, and deposits target structures to the right. These target structures, once created, never dissolve in pieces, but keep rotating 90^o at each time step. For example, the first target structure deposited at t=2 (Figure 5.4), after rotating 4 times, or 360^o, resumes exactly the same location and orientation at t=6 (Figure 5.4). Note the same structure can still be found at t=14 (Figure 5.6) at the same location and orientation. Other target structures,

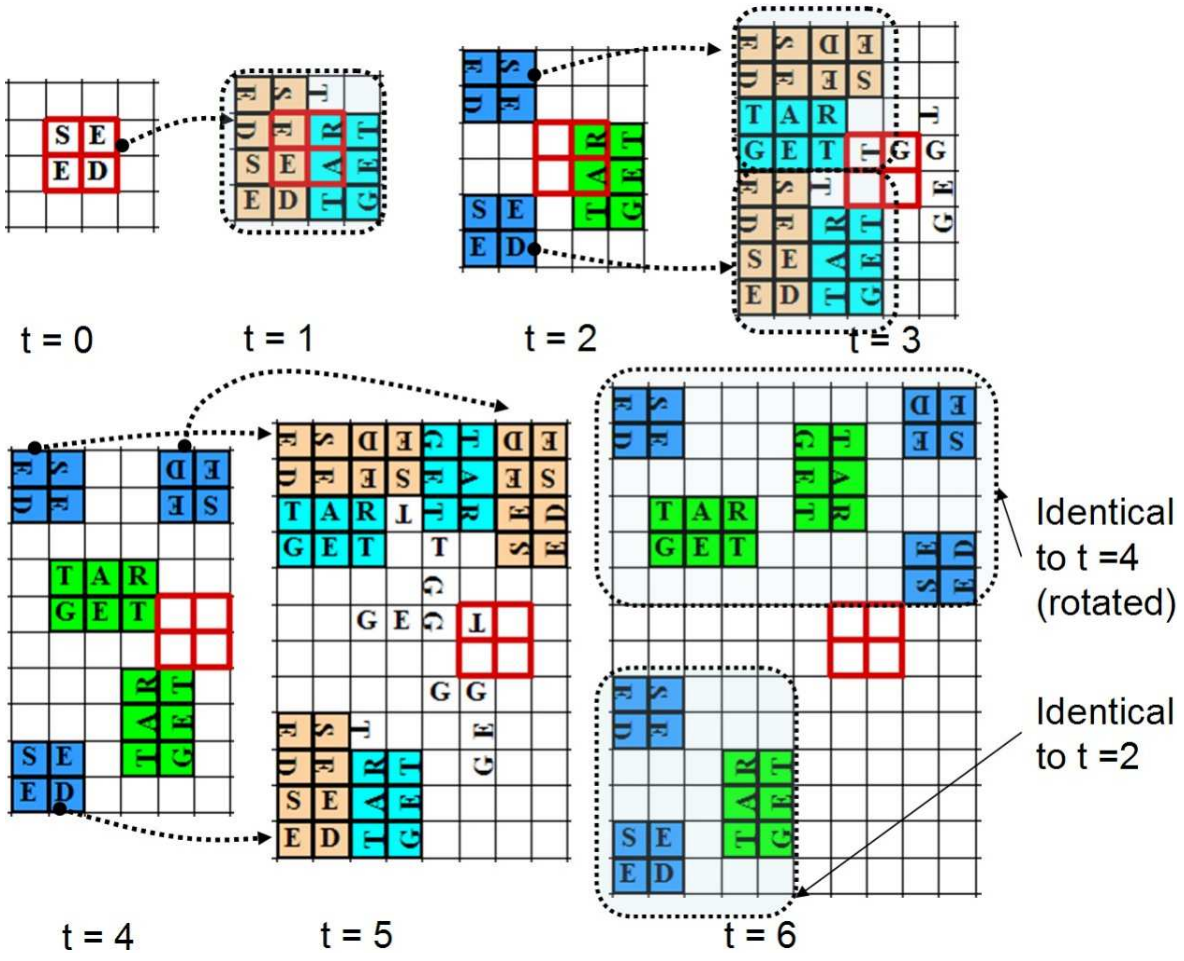


Figure 5.7: Secondary task performance, example 5.1-2: executing second automatically programmed R-tree against the given seed structure from t=0 to t=6. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

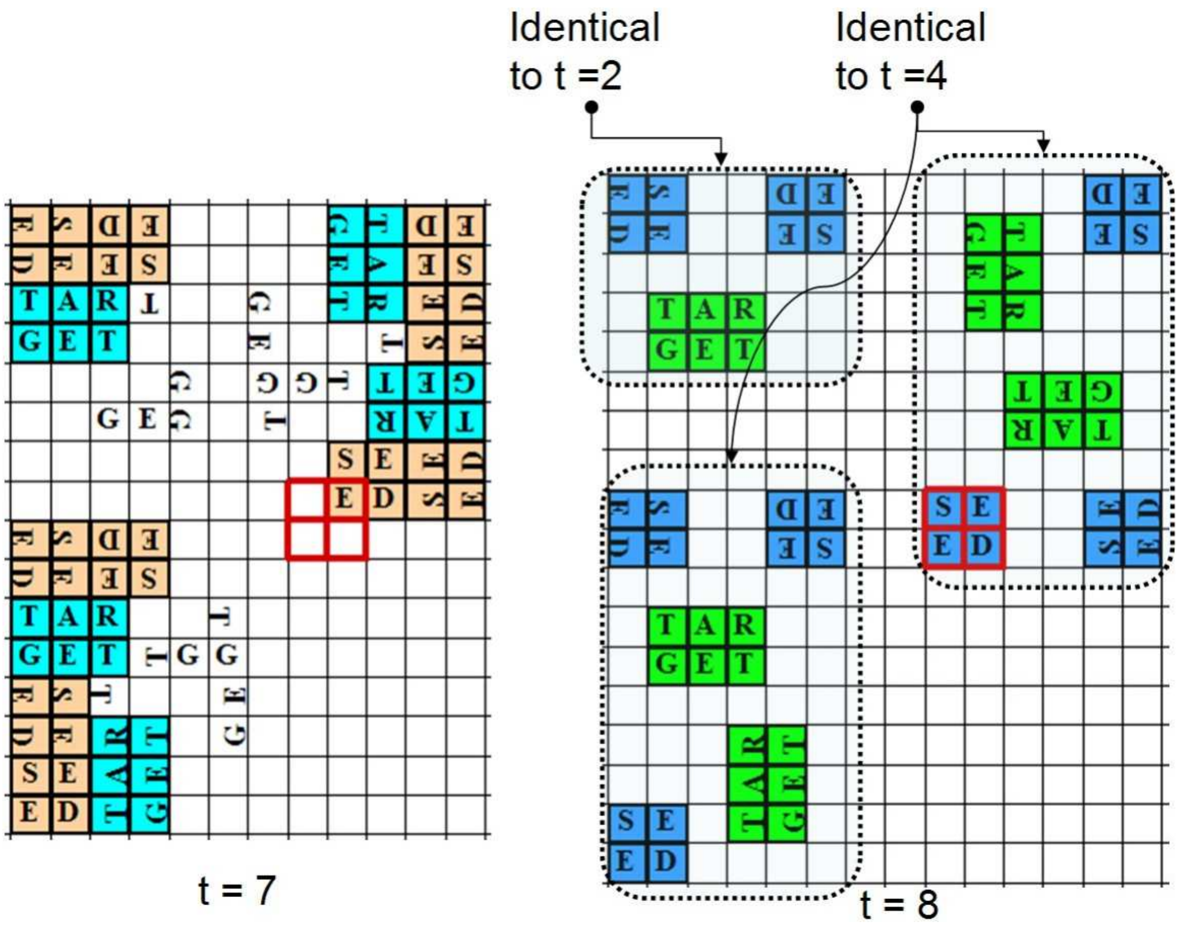


Figure 5.8: Continuation of Figure 5.7 from t=7 to t=8.

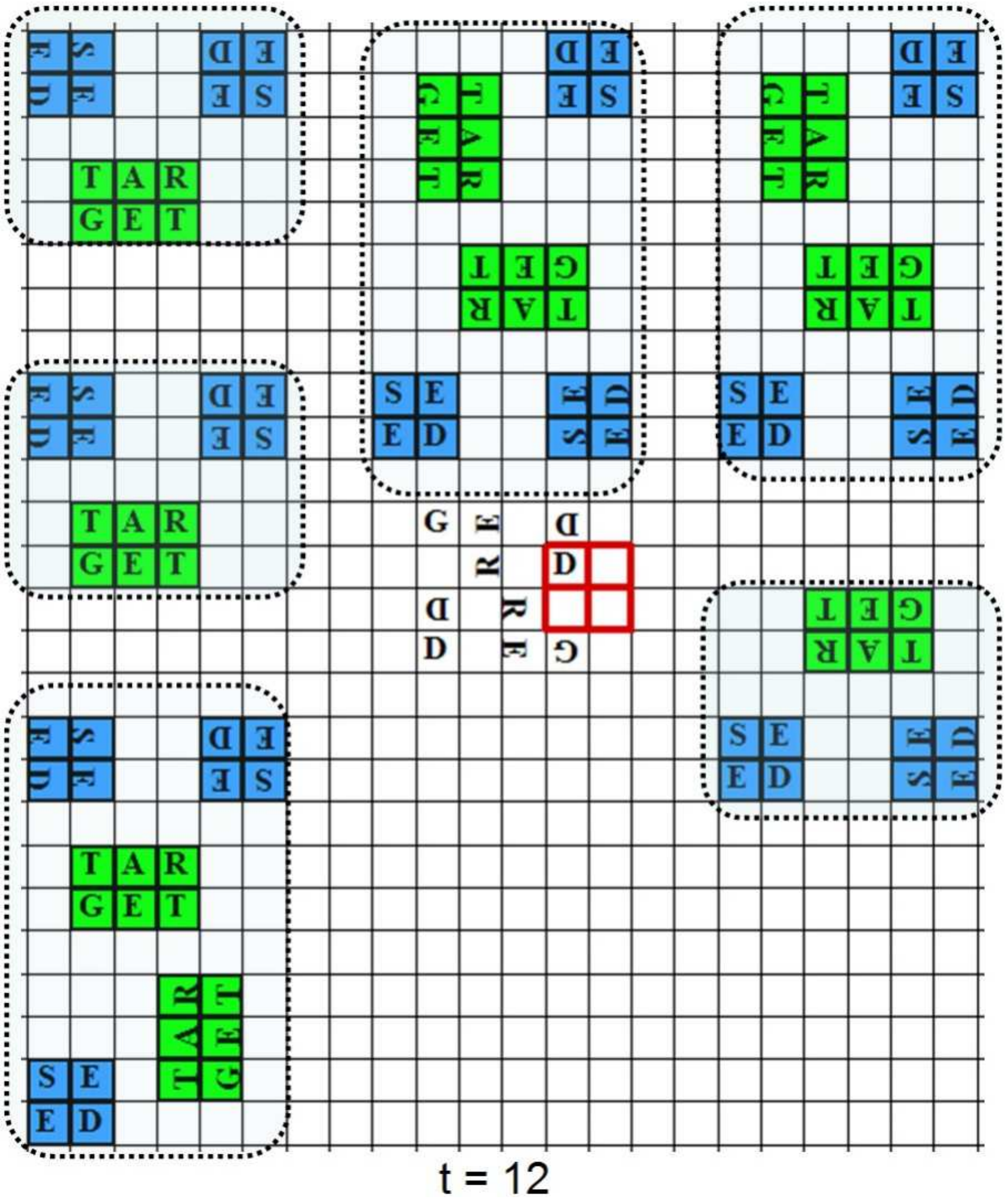


Figure 5.9: Continuation of Figure 5.8 at t=12.

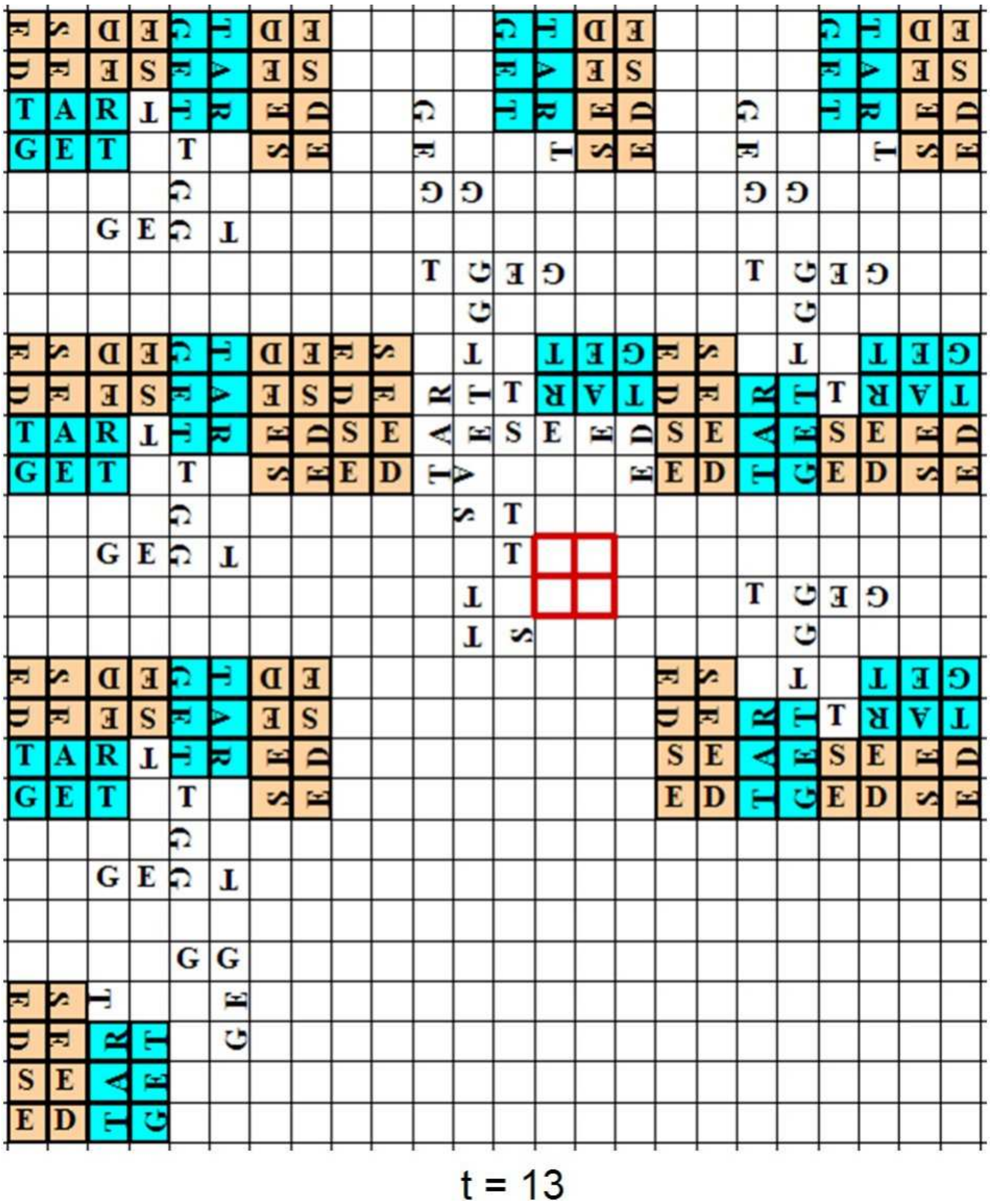


Figure 5.10: Continuation of Figure 5.9 at t=13.

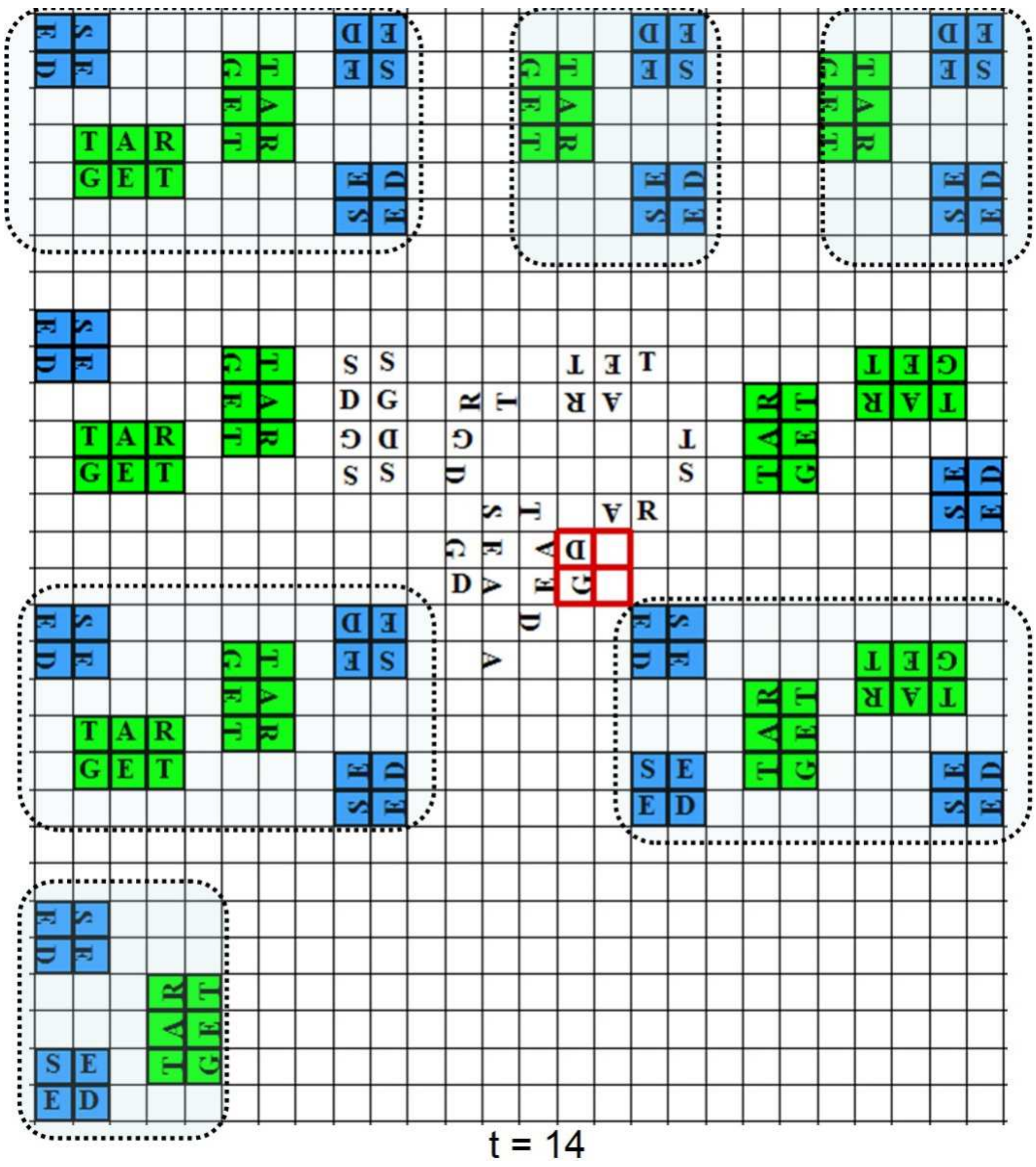


Figure 5.11: Continuation of Figure 5.10 at t=14.

once born and deposited in an expanding triangle area, persist exactly in the same fashion. 3)By the time of t=14 (Figure 5.6), the fission-wall deposits 8 seed replicas on the left edge and 9 target structures in the triangle behind the edge.

In example 5.1-2 (Figure 5.7), the second R-tree discovered a very different strategy to perform the same task. As illustrated in Figure 5.7, at t=1, the seed shown at t=0 transforms into three contiguous structures (2 seeds + 1 target), almost identical to t=1 in example 5.1-1 above (Figure 5.4). However, at t=2, something different is exhibited. Even though the contiguous structures also split into three isolated structures, but in contrast to first R-tree, this time the target structure just translates to the right, without rotating itself. From this point of time on, the strategy which this R-tree takes is very simple. It does nothing else but one thing: let every new seed replica attempt to repeat exactly the same action that the initial seed did at t=1 and t=2. For example, each of the 2 seed replicas found at t=2 transforms into exactly the same sets of contiguous structures at t=3. Note two things here. First, since the seed replica on the top at t=2 is in a rotated orientation, the corresponding set of contiguous structures it forms on the top at t=3 is also rotated. Secondly, when the second seed replica at t=2 transforms into second set of contiguous structures at t=3, it collides with the existing target structure and causes the target structure to fall into pieces. Likewise, the first set of contiguous structures at t=3 (on the top) splits into three isolated structures, exactly the same action and result shown at t=2 (rotated). The second set of contiguous structures at t=3 (at the bottom) also attempts to repeat the same action, but there is not enough room and one seed replica (at the upper position) collides with the target structure from the other set of contiguous structures and got nullified, so only 2 isolated structures are resulted from the second set. Next, each of the three seed replicas found at t=4 repeats exactly the same action, and results in the isolated structures shown at t=6. Note that at t=6 the lower part is identical to t=2 and the upper part is identical to t=4 (rotated). This is because the seed replica at left bottom at t=3 is able to repeat exactly the same action and result as at t=1 and t2, while the 2 seed replicas on the top at t=4 repeats exactly the same action and result as at t=3 and t=4. This pattern repeats recursively, again and again. At t=12 (Figure 5.9), every seed replica still simply repeats exactly the same action as shown at t=13 (Figure 5.10) and t=14 (Figure 5.11). Comparing the two discovered strategies found by the single GP run, each has its own strength and weakness. With first R-tree collision only happens between seed replica themselves. Each target structure avoids collision and manages to persist by rotating itself and quickly getting away from the "mother" seed as soon as it is born. With R-tree 2, target structures do not rotate, collision can happen between seed replicas and target structures, and possibly cause some target structure to fall into pieces. However, it turns out this strategy is more productive, due to the fact that, by allowing target structures to also share the risk of collision, more seed replicas themselves managed to survive, and each survived seed replica, in turn, recursively splits into more target structures. Comparing the results at t=14 (see Figure 5.6 and 5.11), R-tree 1 only produces 8 seed replicas and 9 target structures, while R-tree 2 has produced 17 seed replicas and 13 target structures, given the same number of time steps. These two strategies can be said indifferent, as none can be said better than the other. In real application, if persistence is the priority, than R-tree 1 is choice. On the other hand, if productivity is the priority, R-tree 2 would win.

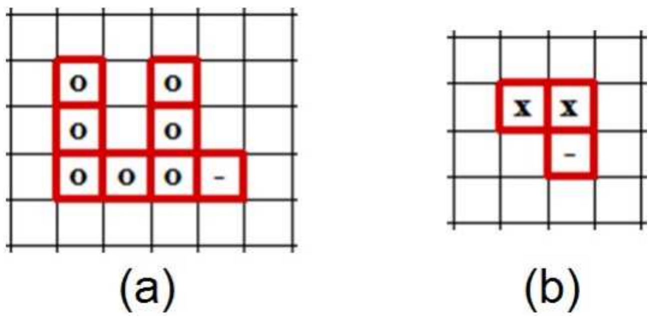


Figure 5.12: Secondary task performance, example 5.2: (a)the target structure, pre-specified as a 8-component 7-state weak-rotational-symmetric structure which writes out an English letter “U”; (b)the seed structure, pre-specified as a 3-component 7-state weak-rotational-symmetric structure. An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., “manufacture” the target structure as it replicates.

Next, a second example experiment is presented with a different seed and target structures. The target structure is pre-specified as a 8-component 7-state weak-rotational-symmetric structure which writes out an English letter “U” (Figure 5.12(a)), the seed structure is pre-specified as a 3-component 7-state weak-rotational-symmetric structure (Figure 5.12(b)). One of the resulted R-trees is picked from the elite population after re-configuring the multi-objective replicator factory model with the new seed and target structures and running the second experiment. The execution result from T=0 to T=14 is presented from Figure 5.13 to Figure 5.17. Here, a clear pattern can also be observed: 1) the configuration

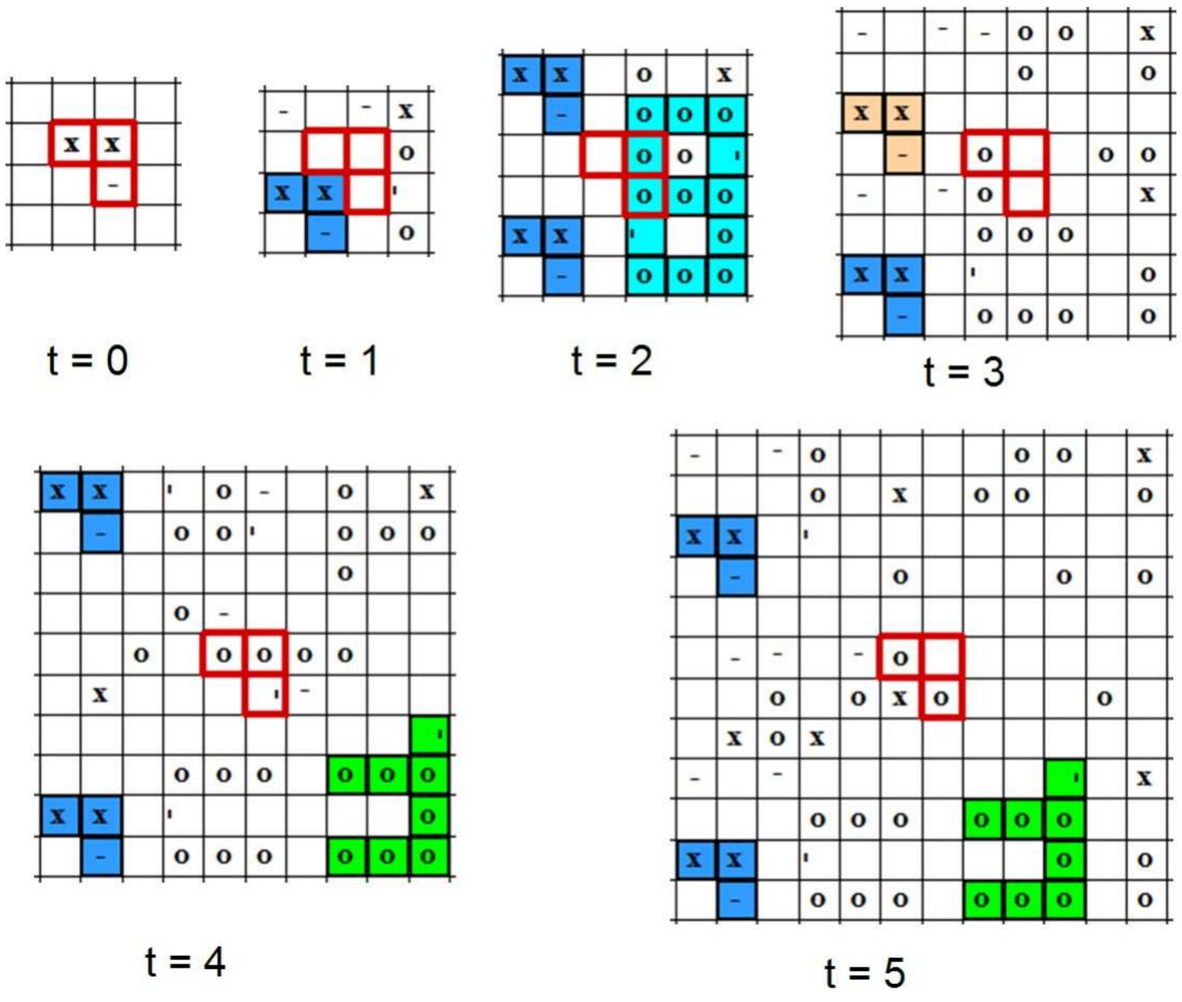


Figure 5.13: Secondary task performance, example 5.2: executing an automatically programmed R-tree against the given seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

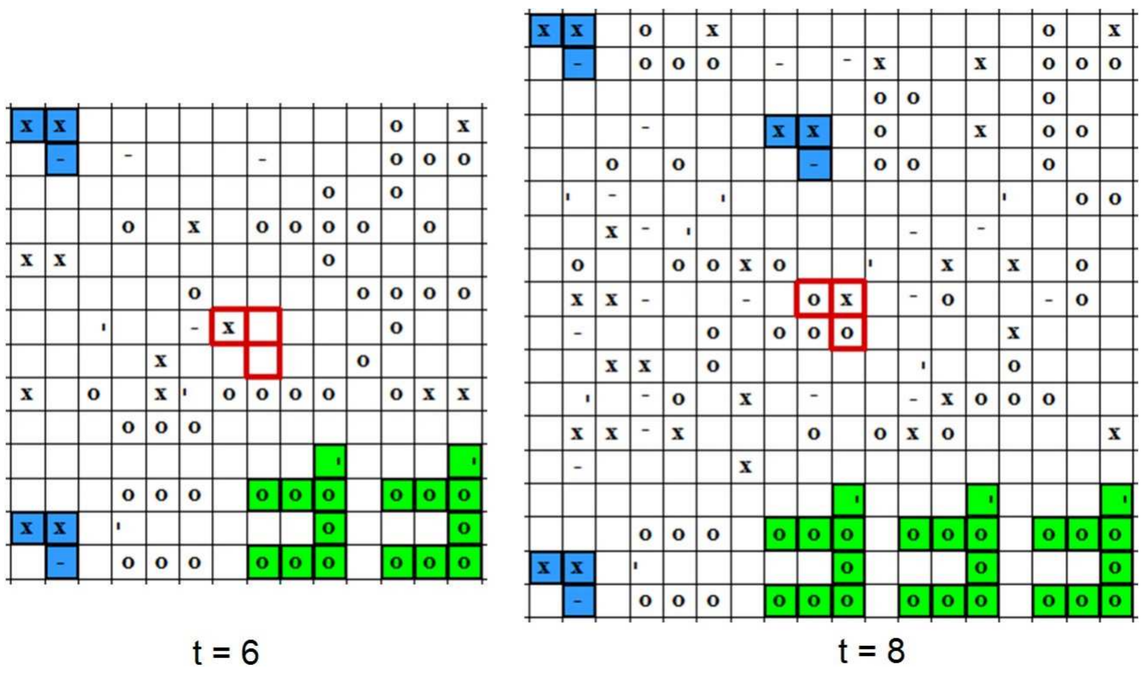


Figure 5.14: Continuation of Figure 5.13 at t=6 and t=8.

expands from the initial seed cells toward all 4 directions (up, right, down, left) at light speed; 2) first seed replica appears at t=1 (Figure 5.13) at left bottom corner, then persists, keeps moving away from the seed cells subsequently; 3) second seed replica appears at t=2 (Figure 5.13), right above the first one, and then keeps moving apart in opposite directions, while also keeps moving away from seed cells; 4)the first target structure is deposited at the right bottom corner at t=4 (Figure 5.13), then persists, and keeps moving at exactly the same pace with the first seed replica. Note that the relative position between the first seed replica and first target structure (as well as the components between them) never changes after t=4 (Figure 5.13); and 5)additional target structures are deposited after the first one, very regularly lined behind each other, and then keep moving exactly at the same pace, all led by the very first seed replica; and 5)by the time of t=14 (Figure 5.17),

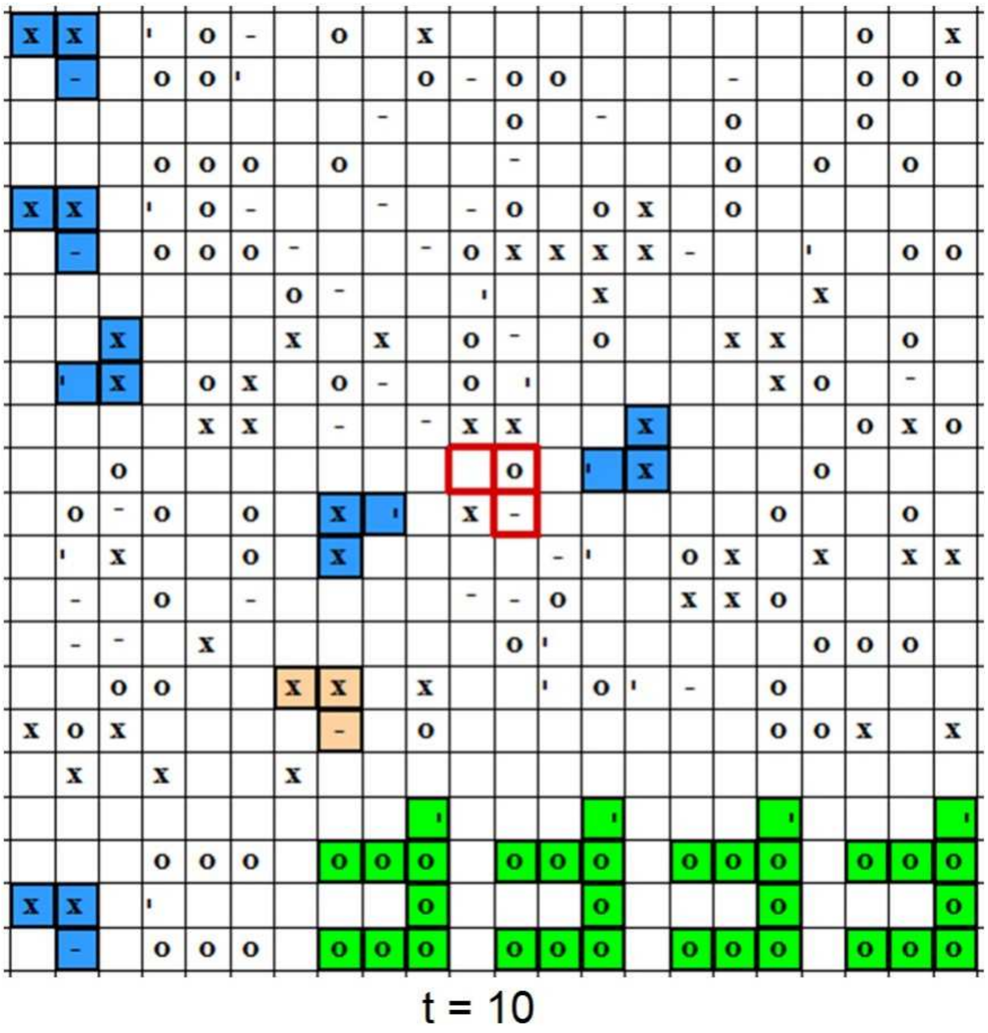


Figure 5.15: Continuation of Figure 5.14 at t=10.

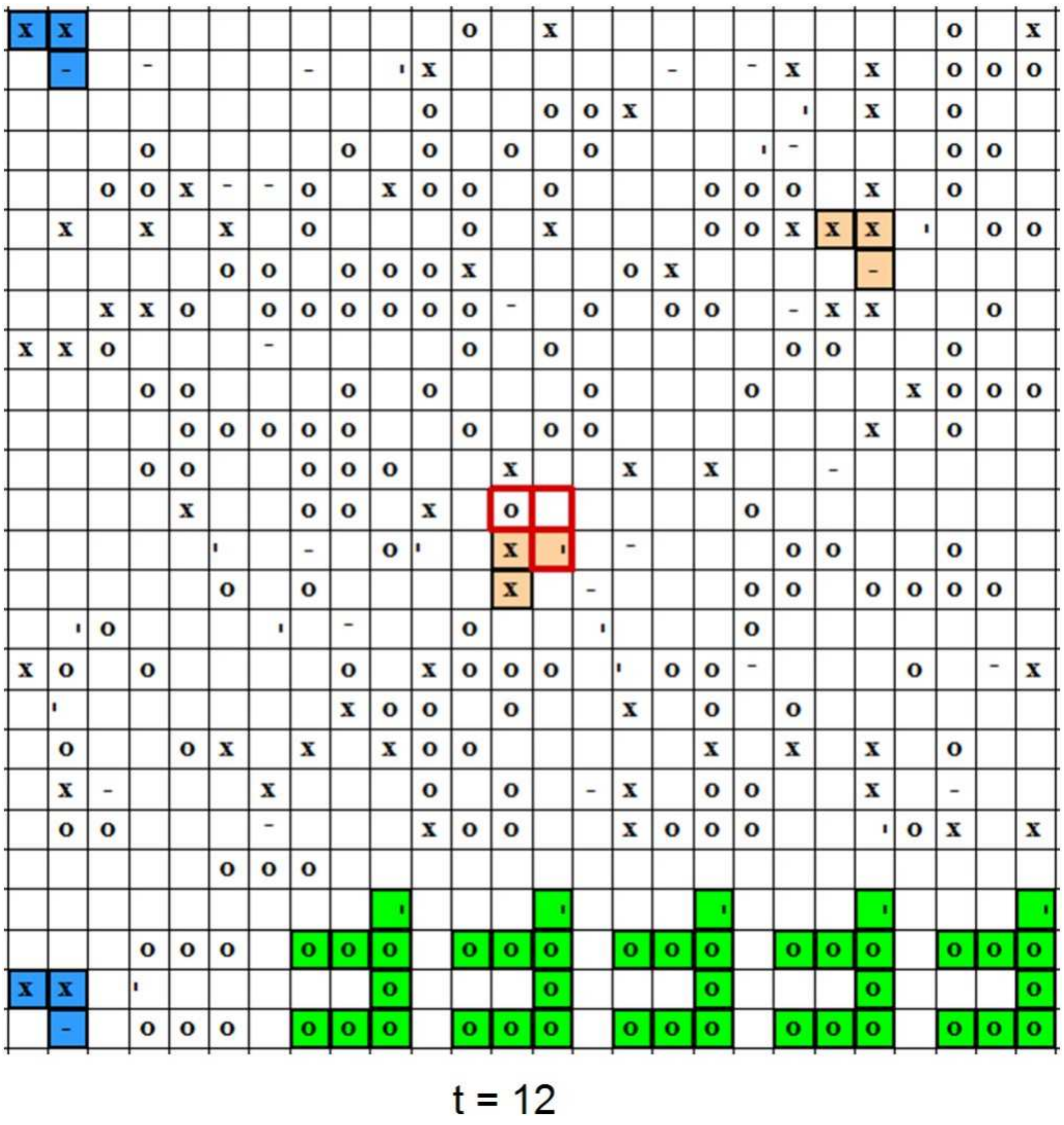


Figure 5.16: Continuation of Figure 5.15 at=12.

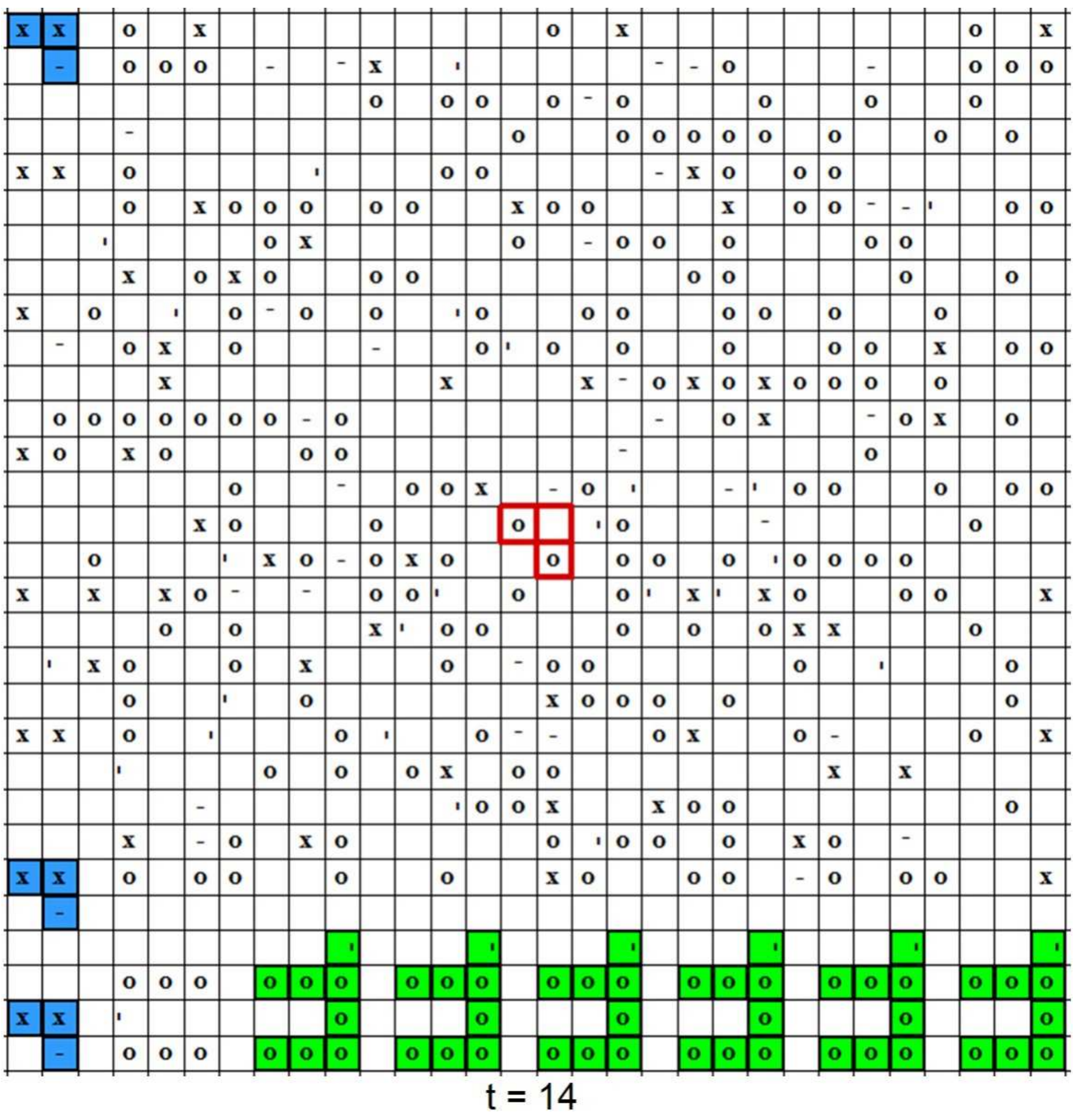


Figure 5.17: Continuation of Figure 5.16 at t=14.

6 such target structures are “manufactured”’, seemingly lining up in a horizontal “assemble line” at the bottom.

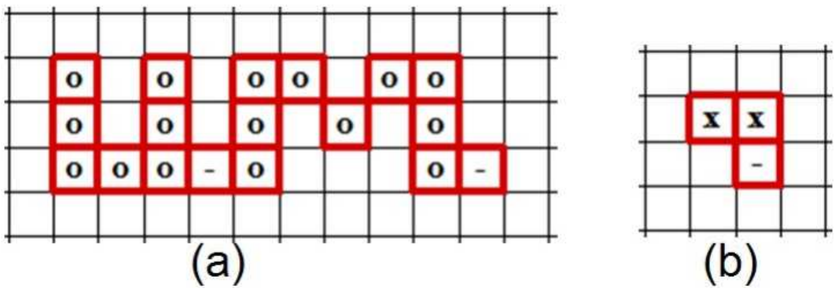


Figure 5.18: Secondary task performance, example 5.3: (a)the target structure, pre-specified as a 18-component 7-state weak-rotational-symmetric structure which writes out “UM”, the acronym for University of Maryland; (b)the seed structure, pre-specified as a 3-component 7-state weak-rotational-symmetric structure, same seed used in example 5.2. An R-tree is sought to program the seed structure not only to self-duplicate, but also to perform a secondary task, i.e., “manufacture” this bigger, more complicated target structure.

For comparison, a third sample experiment is presented here, in which the same seed structure is used as in the second example (Figure 5.18(b)), but a bigger and more complexed target structured is specified. As shown in Figure 5.18(a), the new target structure is a 18-component 7-state weak-rotational-symmetric structure which writes out “UM”, the acronym for University of Maryland. An R-tree automatically programed by the multi-objective replicator factory is executed with the results illustrated from Figure 5.19 to Figure 5.24. It indicates that the same seed structure can be programmed to write out not only “U”, but also “UM”, in a clear and consistent pattern: 1) the configuration also expands from the initial seed cells

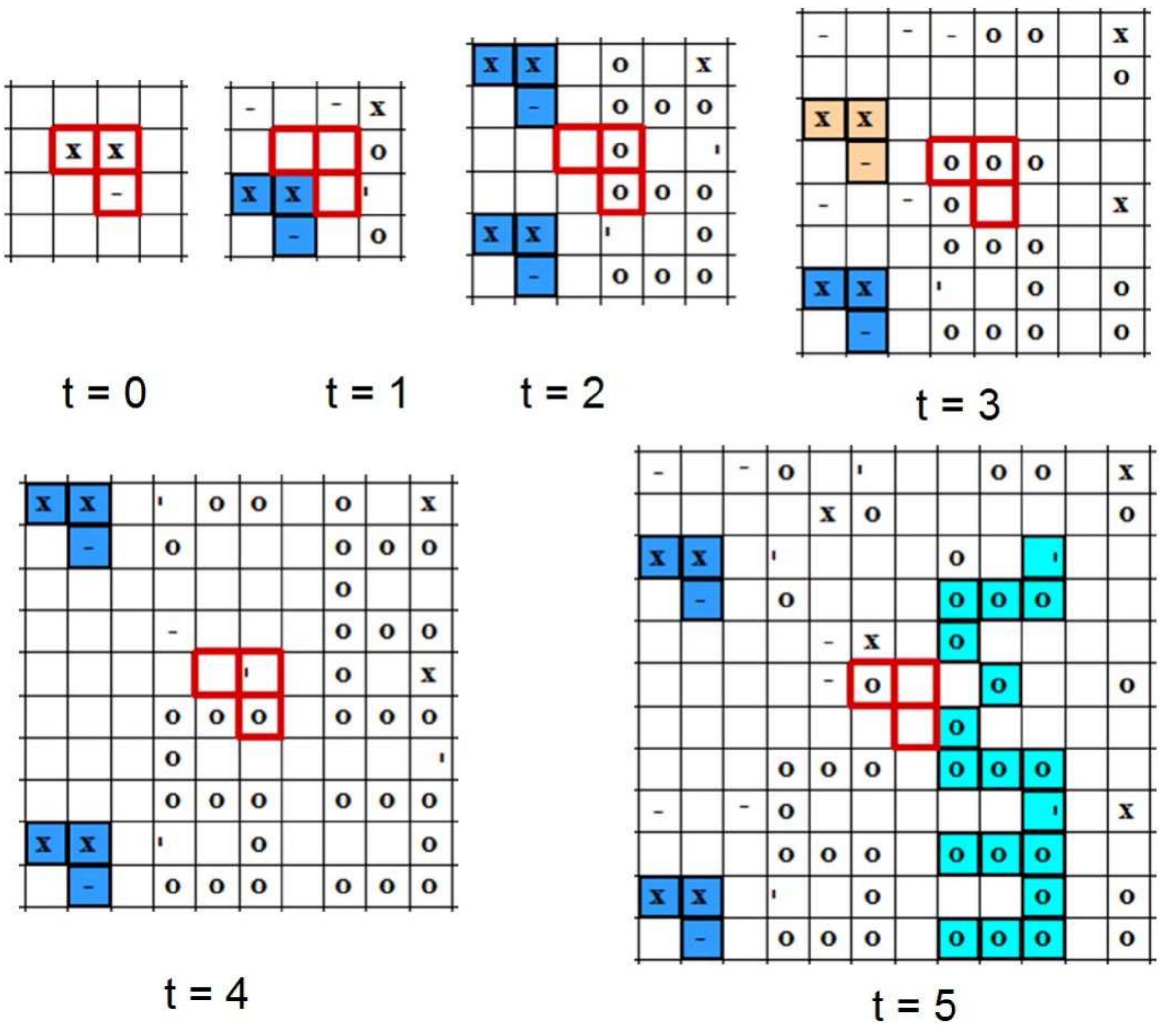


Figure 5.19: Secondary task performance, example 5.3: executing an automatically programmed R-tree against the given seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

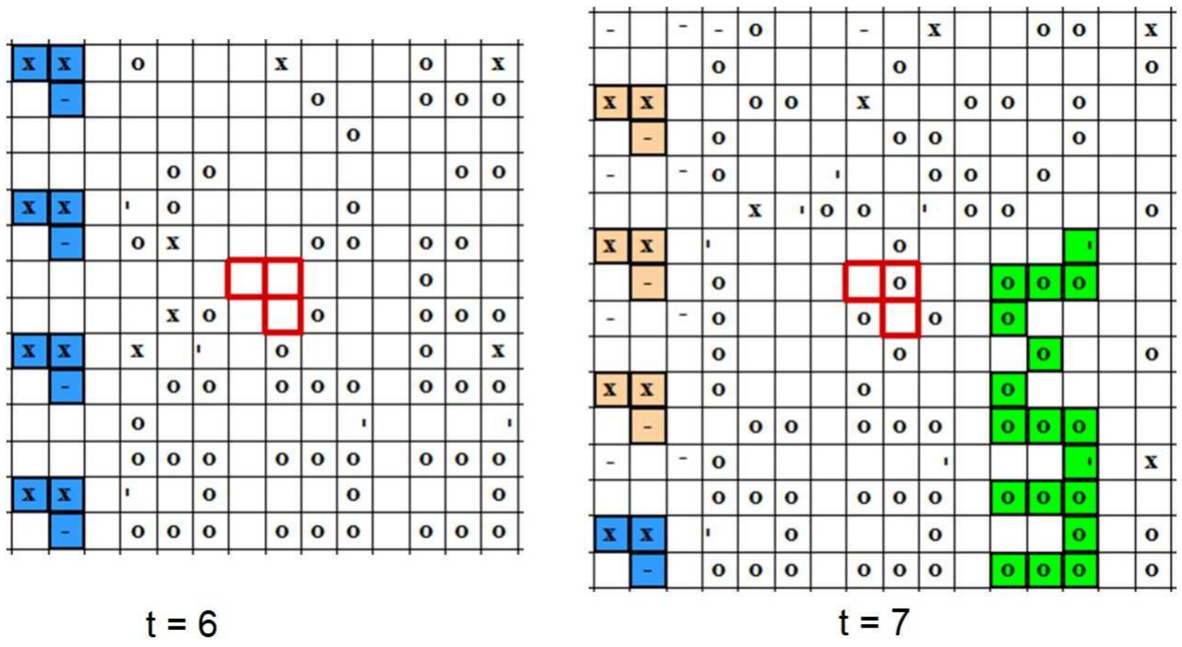


Figure 5.20: Continuation of Figure 5.19 at t=6 and t=7.

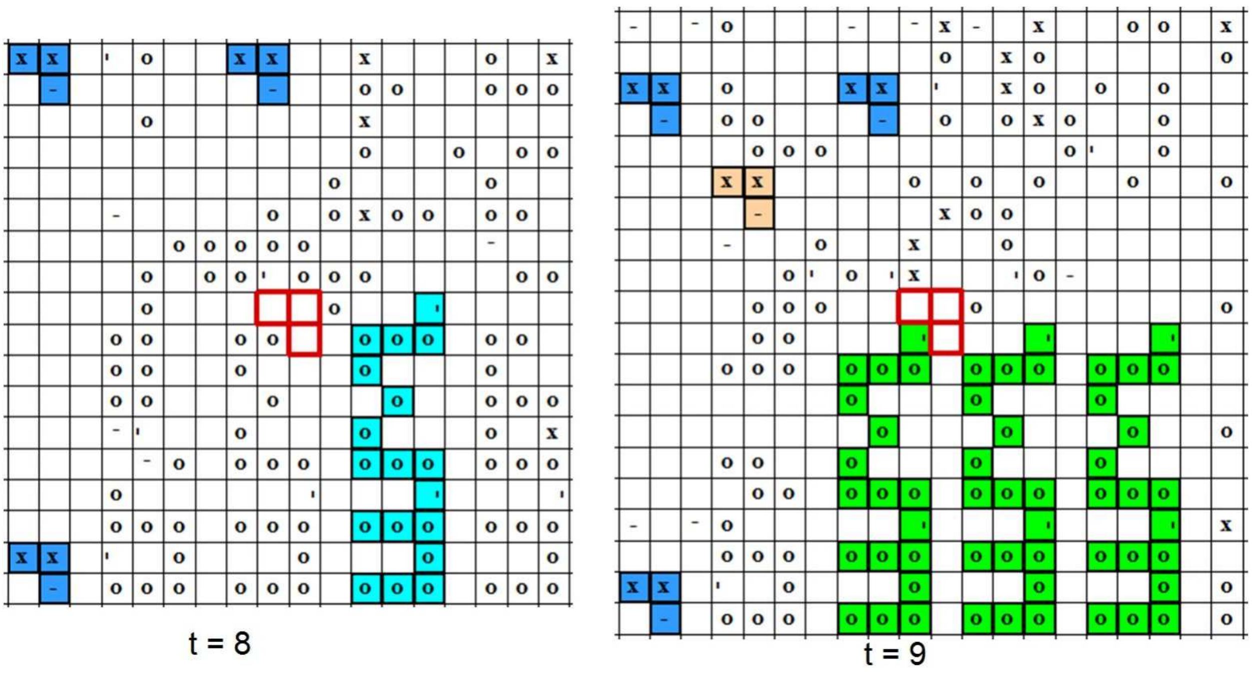


Figure 5.21: Continuation of Figure 5.20 at t=8 and t=9.

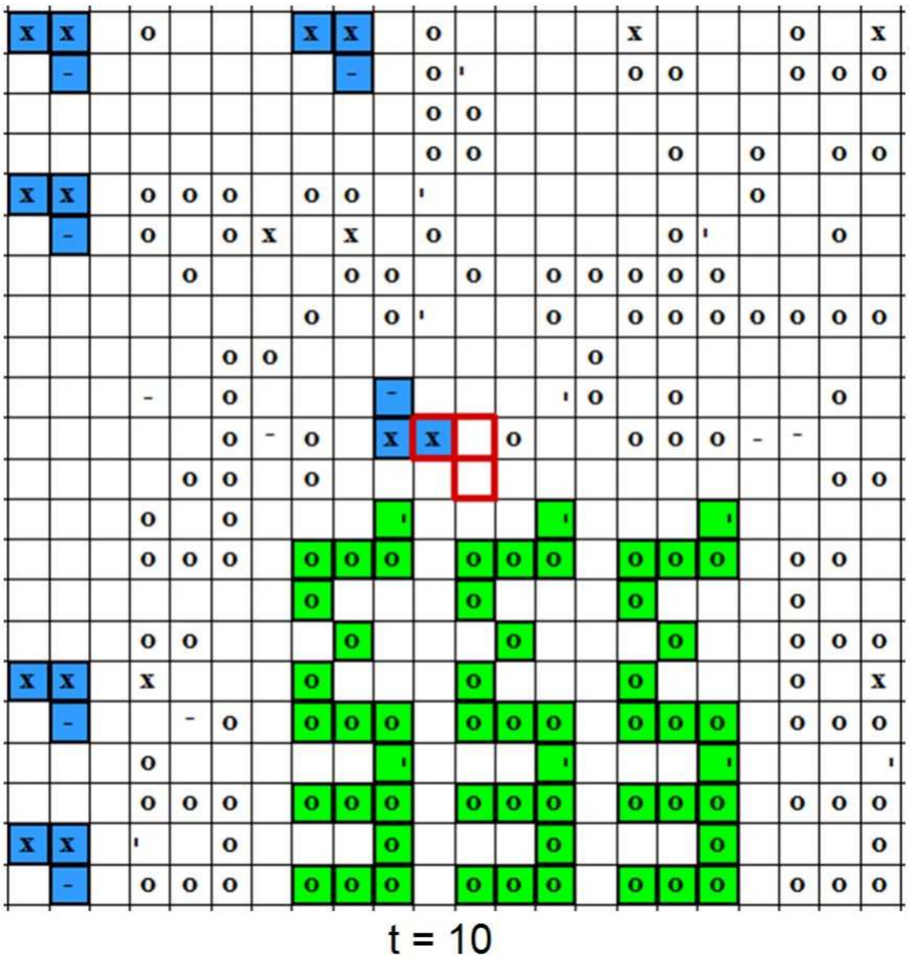


Figure 5.22: Continuation of Figure 5.21 at t=10.

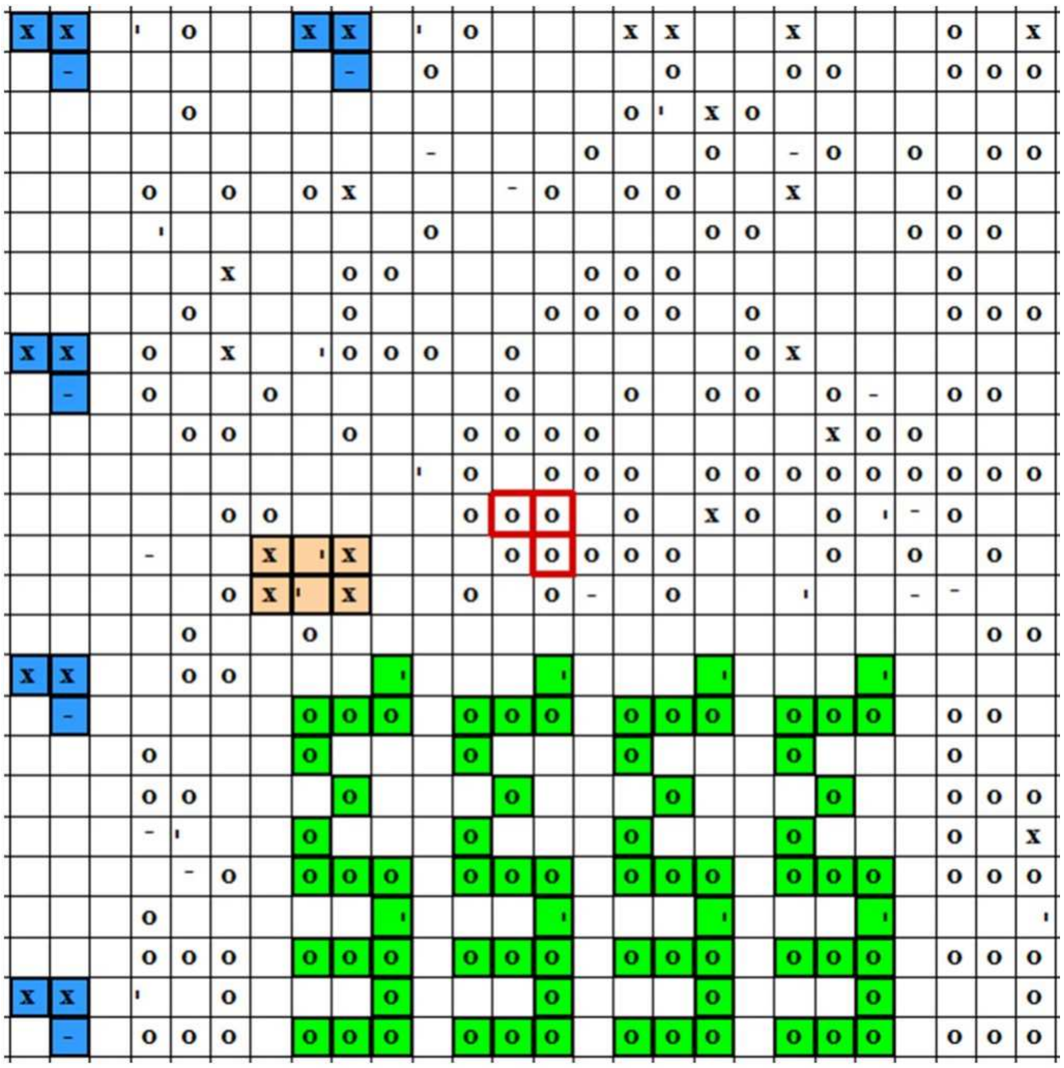


Figure 5.23: Continuation of Figure 5.22 at t=12.

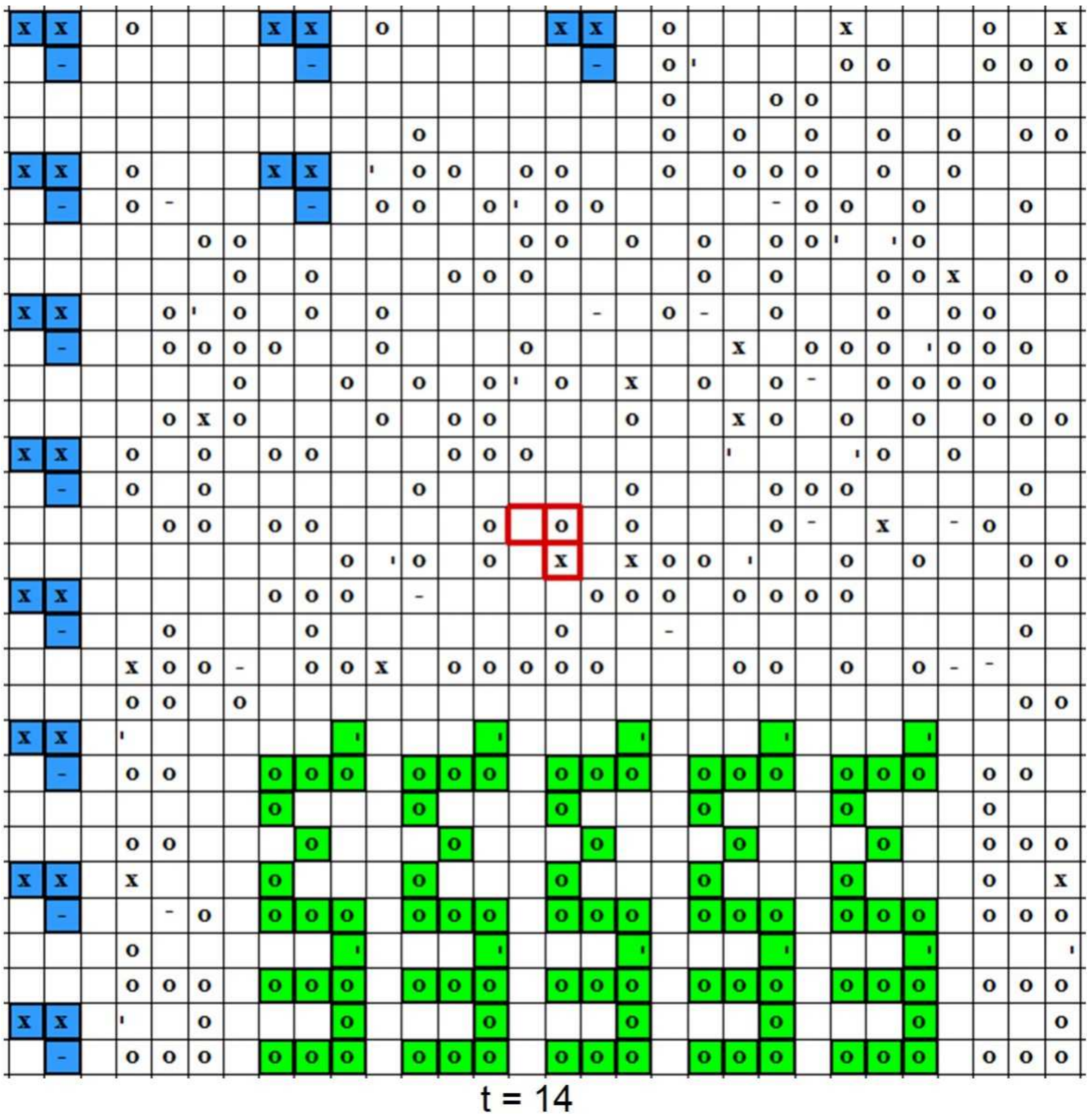


Figure 5.24: Continuation of Figure 5.23 at t=14.

at light speed in all 4 directions; 2) the seed structure keeps replicating, mainly on the left edge (from t=2 (Figure 5.18), 2 on the edge, to t=14 (Figure 5.24), 8 on the edge); 3) the first target structure takes more steps to appear than in the example 5.1 and 5.2 above, due to its increased size; 4) however, once the target structures are formed, they also regularly and persistently line up at the bottom, translating to the left as led the first seed replica.

5.8 Conclusions

Past studies of self-replication in CA models have mainly involved two approaches, the theoretical study of universal constructors which are highly complex but marginally realizable, and the manual design of simple self-replicating loops, which can do nothing but self-replicate [65]. Tempesti took the first step and showed that it is possible to add additional computational capabilities to the simple self replicating loops, such as writing out “LSL”, the acronym of Logic Systems Laboratory, and hence attaining complex machines that are nevertheless completely realizable [68]. Subsequent studies have also shown a simple yet universal language can be embedded into self-replicating loops [53]. These works demonstrated that some simple manually-designed self-replicating loops are capable of carrying out some limited “secondary” tasks, but such capabilities are limited by specific, non-arbitrary, and manually-designed seed structures, the need for writing manual, pre-written computer programs, and the cost of altering (and changing the complexity of) the seed structure itself by embedding the pre-written program into the specific seed structure. A third approach, as established in this study in previous chapters, can create novel self-replicators, with the use of GP to automatically program the cellular automata with arbitrary, initially non-replicating structures. As this chapter further shows, these automatically programed, arbitrary self-replicators are at least as sound as the manually designed self-replicating loops, in terms of the capability of carrying out a simple secondary task, such as writing out "UM", the acronym for University of Maryland. Further, the approach demonstrated here represents a big step forward from the past, because of the following advantages: 1) it generalizes such a secondary computability to arbitrary self-replicating structures, in contrast to the specific manual loops used in the past; 2) it automatically programs the CA to support both self-replication and secondary computation, in contrast to the need for pre-writing manual programs in the past; 3) it carries out such a secondary task with use of the same initial seed structure, in contrast to the past where the seed structure had to be altered by embedding the manual program in it; and 4) both self-replication and secondary computation are programmed in a single GP run, with the possibility of yielding multiple non-dominated solutions, each representing a diversified strategy carrying out the same given task. Several sample experimental results are included to demonstrate the effectiveness of such a novel approach. Consistent patterns are observed from multiple experiments using different seed structures, target structures, or both, indicating when the seed structure keeps self-replicating like a fission-wall, persistent target structures being deposited behind, either lined up and translating at the same pace led by the very first seed replica (see Figure 5.17 and Figure 5.24), or very regularly aligned in an expanding triangle, with each rotating at the same location (see Figure 5.6).

Chapter 6

Multi-Objective Structure/Rule Co-evolution with Unspecified Seed Structures

6.1 Overview

In last chapter, it was demonstrated that we can automatically program arbitrary structures in CA to self-replicate and do secondary task performance at the same time, in a single GP evolutionary process. In such a model, there are two obvious characteristics to be noticed: 1) both the seed structure and the target structure are known prior to evolution as they are pre-specified in the configuration file; and 2) it is only the R-tree that evolves. The S-tree of the seed structure is produced from the pre-specified seed structure and remains constant during the entire evolution process.

This chapter aims to explore further extending the multi-objective replicator factory model to be even more general, i.e., enabling it to carry out a pre-specified task without being given a seed structure. In such a model, only the task to be done is specified (such as a target structure to be "manufactured"). The new model is expected to automatically and evolutionarily search for an appropriate seed structure as well as associated state transition rules, which cooperatively provide the capability of performing the given task as the found seed structure replicates itself.

To seek a solution to a given task, the new model would extend the searching from rule space only to both rule space and structural space. This represents a novel way for creating CA models toward performing a task. When it proves difficult to perform the given task with a specified seed, this new approach allows the replicator factory to automatically search for an alternative suitable seed structure for the given task. We will see an example where this proves useful in writing out "UMD", a much bigger target structure than "U" or "UM", two structures that were tested in Chapter 5.

In such a generalized replicator factory model, the input is a target structure *e*, and the output is a replicator *X*, which can be viewed as an evolved pair of an S-tree and R-tree. The S-tree part, *X(s)*, can be viewed as the static structural aspect of the replicator, and the R-tree part, *X(r)*, can be viewed as the dynamic state-transitional aspect of the replicator. Therefore, here both aspects are to be automatically programmed to successful perform a pre-specified task. This is done by incorporating structure/rule co-evolution in the current model, with the details presented below.

6.2 Problem Formulation

Let *X* represent an evolving replicator, *X(s)* represent the S-tree part in *X*, and *X(r)* represent the R-tree part of *X*, *e* represent an S-tree for a given target structure, *p* represent an infinite cellular space, *t* represent a time step applying *X* in *p*. With the same functions defined in Section 5.2, we can now formally state the problem as follows:

Given a cellular space p, and an extra structure e, we want to find a replicator X which will satisfy the following goals:

1. There exists a time step T_j, for any positive integer j, so that

$$\sum_{t=1}^{T_j} \delta(X(r), X(s), p, t) \geq j \tag{6.1}$$

   This is the replicatability objective of structure/rule co-evolution with an unspecified seed structure.

2. There exists a time step T_k, for any positive integer k, so that

$$\sum_{t=1}^{T_k} \eta(X(r), X(s), e, p, t) \geq k \tag{6.2}$$

   This is the taskability objective of structure/rule co-evolution with an unspecified seed structure.

3. There exists a time step T_i, for any positive integer i, so that

$$\sum_{t=1}^{T_i} (\beta_\delta(X(r), X(s), p, t) \times \beta_\eta(X(r), X(s), e, p, t)) \geq i \tag{6.3}$$

   This is the sustainability objective of structure/rule co-evolution with an unspecified seed structure.

6.3 Replicator Initialization

Before structure/rule co-evolution starts, a population of replicators needs to be initialized. Each replicator has an S-tree part and an R-tree part, so the initialization of each population member includes initializing the S-tree and R-tree at the same time.

6.3.1 S-tree Initialization with Random Structures

A random structure initialization algorithm presented below takes two configurable control parameters, the initial seed size (ν) and density control (ρ). First, a center (empty) cell is selected in the cellular automata space. Then an active state is randomly selected with uniform probability. This cell now becomes the root of the structure. Next, starting from the root, each active cell attempts to expand the structure by recursively "reconstructing" (assigning states to) its Moore neighbors, controlled by ρ, until the structure size reaches ν. This is done as follows. First, the root cell attempts to reconstruct each of its eight Moore neighbors. To reconstruct a Moore neighbor, either a quiescent state or an active state can be selected, with the probability of an active state to be selected being ρ. If an active state is to be selected, such a state is picked with uniform probability (among all active states). In either case, each of these reconstructed cells are marked with flag "RECOVERED". Next, in the order of how they were reconstructed, each of these active Moore neighbors, in turn, recursively attempts to reconstruct its own Moore neighbors. Note that it is possible that some of the Moore neighbors are already marked “RECOVERED”, since they are already reconstructed by previous active components. These neighbors are left untouched. Only the remaining neighbors are reconstructed in the same way. This recursive Moore neighbor reconstructing process continues until the initial structure size ν is reached. Note that, a higher ν will result in a bigger initial seed structure. On the other hand, a higher ρ results in a simpler and more condensed shape, and a low ρ results in a more expanded and complex shape. Once the random structure construction is complete, an S-tree can be immediately derived, exactly in the same way as before (Section 3.2.2).

6.3.2 R-tree Initialization for Co-evolution

For each member of the replicator population, the R-tree part can be initialized exactly as before, as described in Section 3.4.2.

6.4 Replicator Simulation

Once the S-tree part is converted into the seed configuration (t=0), the CA simulation with the R-tree part can be carried out exactly the same as in Section 3.4.3.

6.5 Replicator Fitness Evaluation and Tournament Selection

Once the replicator is simulated with a given number of time steps, its replicatability, taskability, and sustainability fitness measures can be calculated exactly the same as in Section 5.3, using the seed S-tree from the evolving replicator, and the target S-tree from the target structure. Domination pressure, distribution pressure, and parsimony pressure can also be derived as before. Tournament selection based on these criteria picks diversified replicators to form the mating pool, as described in Section 5.4.

Note that here the fitness of an S-tree itself is not directly evaluated. The tournament selection is based on the fitness values of the replicators, not their S-tree parts alone. The fitness of a replicator is evaluated by the configurations it produces with respect to self-replication and task-performance, as a collective result of executing its R-tree part on its S-tree part, which are both evolved. However, should it be desired in a future application, it is possible to add a fitness component that evaluates the fitness of its S-tree part separately, in terms of the structure it evolves and represents, with respect to the meaning in the application domain. In such scenarios, the new fitness component can be integrated into the tournament selection process, such as the parsimony pressure concerning an R-tree, the replicator's other part.

6.6 Replicator Evolution

Once the mating pool is formed, the candidate replicators can reproduce. For example, a pair of parent replicators can re-combine with their S-tree parts, R-tree parts, or both, at the same time. After such crossover operations, both the S-tree part and R-tree part of a candidate replicator can be further subject to the mutation operation. Since the fitness of a resulting offspring replicator is evaluated by the global CA behaviors collectively produced by the new S-tree and R-tree part, the co-evolution of the S-tree and R-tree in the evolving replicators explores the rule space and structural space cooperatively toward the successful performance of a given task.

6.6.1 Evolution in Structural Space with S-tree Genetic Operators

The S-tree representation of structures converts the evolution of an arbitrary CA structure into the evolution of a tree structure, which has already been well studied under genetic programming. This is based on the S-tree's property that the encoding and decoding between a structure and an S-tree is unambiguous in both directions. A specific structure can be unambiguously reconstructed from an evolved S-tree through the reconstruction of the Moore neighborhood of each active cell in the structure. In a way, the S-tree can also be viewed as a rule tree which governs how to reconstruct the Moore neighborhood for every component in the structure.

The existing genetic programming operators can readily be applied to S-trees. One point crossover between two S-trees is equivalent to exchanging sub-structures between two structures, and one point mutation is equivalent to local modification to the structure. The point where crossover or mutation is performed on the S-tree can be randomly selected from all edges with uniform probability.

Figure 6.1 shows an example of structure evolution using a one-point GP crossover operator. Initially, S-tree$_1$ (Figure 6.1(a), left) represents Structure$_1$ (Figure 6.1(b), left), and S-tree$_2$ (Figure 6.1(a), right) represents Structure$_2$ (Figure

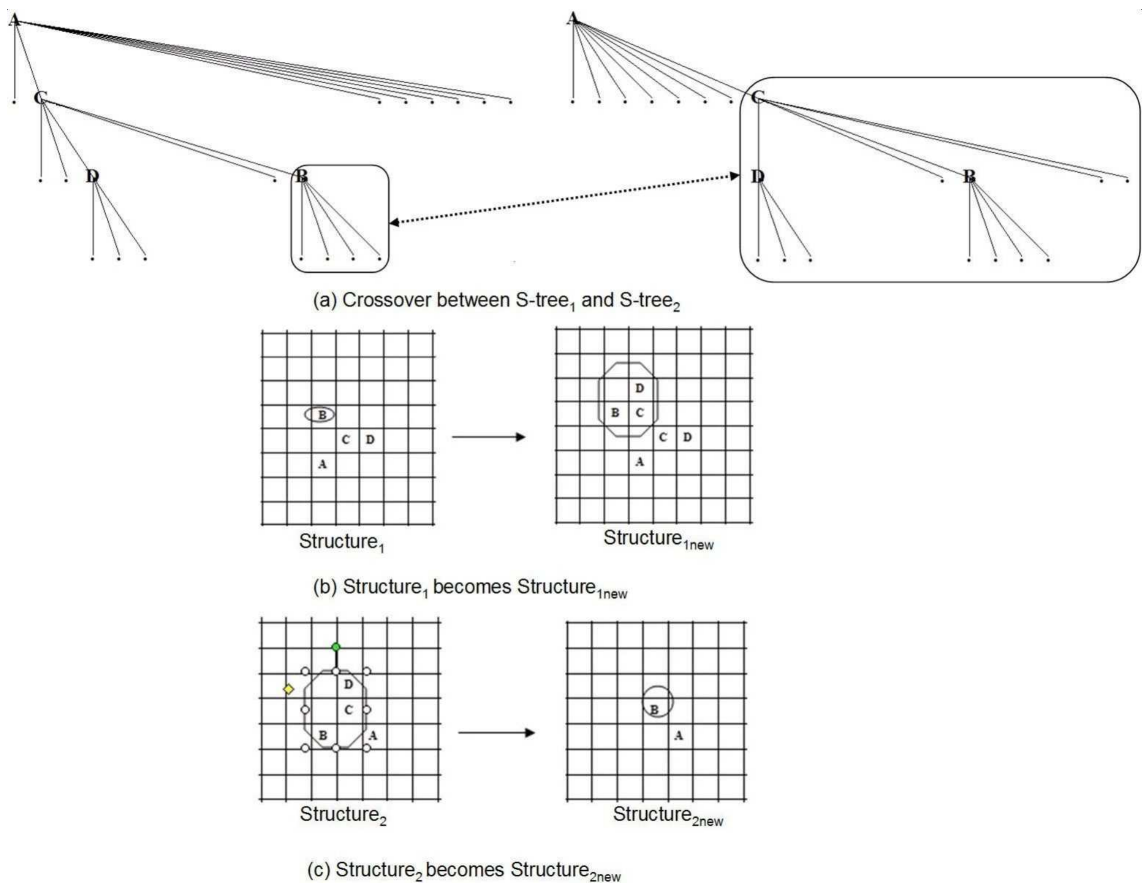

A
C
D
B
(a) Crossover between S-tree$_1$ and S-tree$_2$
Structure$_1$
Structure$_{1new}$
(b) Structure$_1$ becomes Structure$_{1new}$
Structure$_2$
Structure$_{2new}$
(c) Structure$_2$ becomes Structure$_{2new}$

Figure 6.1: An example of structure evolution with one-point S-tree crossover. A crossover takes place between S-tree$_1$ (a, left) and S-tree$_2$ (a, right) with the shaded sub trees exchanged. Correspondingly, Structure$_1$ (b, left) and structure$_2$ (c, left) are now replaced by structure$_{1new}$ (b, right) and structure$_{2new}$ (c, right), respectively. The resulting new S-trees after the crossover operation now represent the new structures respectively.

6.1(c), left). A crossover takes place between S-tree$_1$ and S-tree$_2$ with the shaded sub trees exchanged. Correspondingly, Structure$_1$ (Figure 6.1(b), left) and Structure$_2$ (Figure 6.1(c), left) are now replaced by Structure$_{1new}$ (Figure 6.1(b), right) and Structure$_{2new}$ (Figure 6.1(c), right), respectively. Note that the sub-structures being exchanged are circled for illustration. The resulting new S-trees after the crossover operation now represent the new structures. It can be seen that the sub-structure is not always exactly copied over to the new structure. It is potentially reorganized as the new neighborhood requires. Also note that Structure$_{1new}$ is much larger and more complex than either Structure$_1$ or Structure$_2$, and at the same time Structure$_{2new}$ becomes a diminished structure containing only two components. This raises a risk that the resulting structure after repeated crossovers may become unbounded large or diminished small, both of which are undesirable for the evolution to converge. Hence, like in the R-tree crossover operation (Section 3.3.2.1), the more restrictive homologous one-point crossover genetic operator instead is chosen. The essential idea is almost the same as in the R-tree homologous crossover (see Section 3.3.2.1 for details), except the definition of "mismatch" is different here. In S-tree homologous one-point crossover, 2 nodes (each from a different parent) are not a "mismatch" as long as both are either quiescent or active (meaning the states themselves do not have to exactly match). As a result, S-tree homologous one-point crossover allows searching a much smaller space and inclemently converging toward to common sub-structures which can not be modified again without the mutation operator. The S-tree homologous one-point crossover operator is less restrictive than the R-tree homologous one-point crossover operator, so that structural crossover can take place on two components as long as the topology of their Moore neighbors match each other. This was done because otherwise, it is very difficult to get the structural evolutionary process started because there are so few matches between the initial random structures.

The offspring S-tree resulting from crossover or mutation operations may contain nodes which have more, or less, child nodes than needed to reconstruct their Moore neighbors. This is similar to the R-tree evolution when a “missing rule” or “obsolete rule” scenario occurs and the R-tree is allowed to repair itself by generating new rules as needed or pruning obsolete rules. The same operations can be done for S-tree evolution. If any node in an offspring S-tree is found missing child nodes needed to fully reconstruct its Moore neighborhood, a new child node can be added to repair the S-tree. Likewise, if any node is found containing more child nodes than it needs, the extraneous child nodes can be pruned.

6.6.2 Evolution in Rule Space with R-tree Genetic Operators

For the R-tree part to evolve, exactly the same genetic operators detailed in Section 3.4.7 can be used.

As a result, each candidate replicator in the evolving population is subject to any combination of the following 4 genetic operators: homologous S-tree crossover, S-tree mutation, homologous R-tree crossover, and R-tree mutation, and each is associated with an independent controlling probability (b^s_c, b^s_m, b^r_c, and b^r_m). By controlling the values of each of these probabilities, we can let the replicator explore the structural and rule space in different paces. A typical combination used in the subsequent experiments is ($b_c^s = 0.1$, $b_m^s = 0.02$, $b_c^r = 0.8$, and $b_m^r = 0.4$).

6.7 The Resulting Generalized Replicator Factory Supporting Structure/Rule Co-evolution

With the capability to initialize a replicator with both of its S-tree and R-tree parts randomly generated, as well as allowing both to simultaneously evolve subsequently with S-tree and R-tree genetic operators, the previously established multi-objective replicator factory model can be naturally further generalized to a multi-objective co-evolution model where only the target structure needs to be pre-specified, and diversified replicators can be concurrently sought, with each replicator in the set resulting in a different seed structure and cellular rule set combinations, which reflect different strategies to perform the common given task. The schematic view of such a generalized model is illustrated in Figure 6.2. Similar to the previous model shown in Figure 5.2, there also co-exist two populations, one optimization population and one elite population. However, the members in both populations are no longer R-trees, but replicators, which are coupled pairs of co-evolving S-tree and R-tree. In other words, each evolving member in the population may represent a different seed structure and associated different state transition rules. In the beginning, the optimization population is initialized with pairs of randomly generated S-trees and R-trees (Section 6.3), and the elite population is empty. Then, each replicator is simulated in the CA space, first by reconstructing the initial CA configuration (t=0)

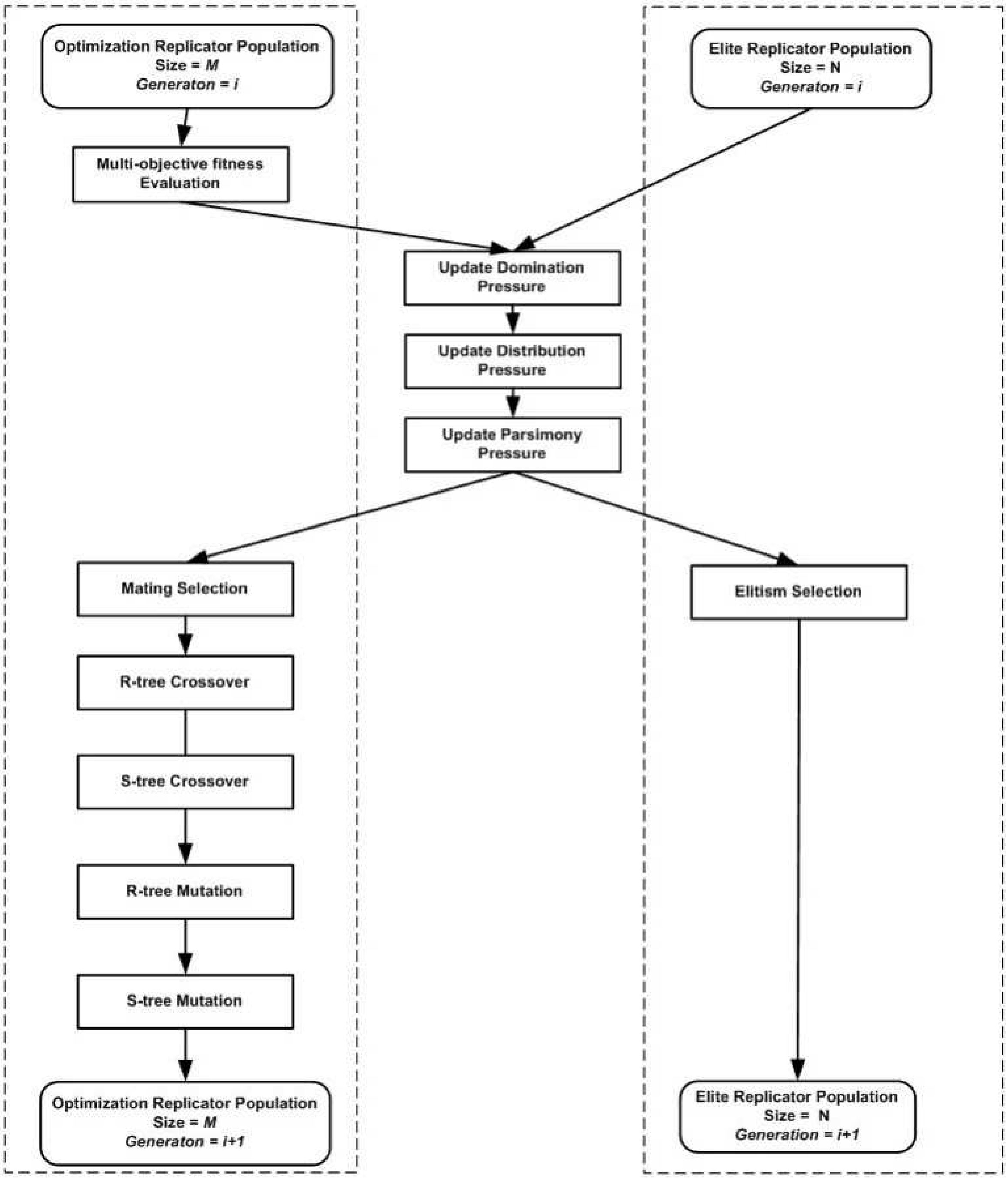

Optimization Replicator Population
Size = M
Generaton = i
Elite Replicator Population
Size = N
Generaton = i
Multi-objective fitness Evaluation
Update Domination Pressure
Update Distribution Pressure
Update Parsimony Pressure
Mating Selection
Elitism Selection
R-tree Crossover
S-tree Crossover
R-tree Mutation
S-tree Mutation
Optimization Replicator Population
Size = M
Generaton = i+1
Elite Replicator Population
Size = N
Generation = i+1

Figure 6.2: A schematic view of the generalized multi-objective co-evolving replicator factory model.

from the evolved S-tree part, then by executing its evolved R-tree part recursively from the initial configuration with a given number of simulation time steps (Section 6.4). With the presence of replicatability, taskability, and sustainability objectives, fitness measures are evaluated based on the simulation results, in the same way described in Sections 5.3.3, 5.3.4, and 5.3.5. Then, based on these multiple fitness measures, the domination relationship, domination pressure, distribution pressure, and parsimony pressure are derived exactly the same as before (Sections 5.4.1, 5.4.2, and 5.4.3).

Next, the mating selection picks replicators from the union population using a tournament selection algorithm to form a mating pool. In this tournament selection algorithm, the probability for an replicator to be picked is inversely proportional to the total pressure it currently faces, i.e., a sum of its domination pressure, distribution pressure, and parsimony pressure. As a result, diversified combinations of parsimonious S-tree/R-tree that are better approximation to better trade-off among replicatability, taskability, and sustainability have a better chance to enter the mating pool and produce offspring replicators. The mating between two replicators now involves not only the crossover of its R-trees, but potentially the crossover of the S-trees as well, even though the crossover of R-tree and S-tree is based on separate probabilities (Section 6.6). Since both probabilities are configurable model parameters, the evolution of R-tree and S-tree can be controlled to take place in different and desired paces. Likewise, after the mating operation, both the R-tree and S-tree are allowed to mutate, based on separate and controllable probabilities. The elite selection operation is done in the same way as before (Section 5.6).

6.8 Structure/Rule Co-evolution: Experimental Results

This section presents some experimental results showing it is indeed possible to automatically program the cellular automata with unspecified seed structures with the capability of both replicating and performing a pre-specified secondary task as the discovered structure replicates. In fact, diversified S-tree/R-tree combinations can be found concurrently in a single GP run, with each reflecting a different strategy, using different seeds and rules to perform the same task. Typical model parameters used are: Elite Population Size = 10, Optimization Population Size = 90, R-tree Mutation Probability = 0.42, R-tree Crossover Probability = 0.88, R-tree GP Tournament Size = 2, Initial Seed Size = 5, Seed Density Control = 0.8, S-tree Crossover Probability = 0.1, S-tree Mutation Probability = 0.02, and Max Hesitation = 200.

In the first co-evolution experimental example, a target structure is pre-specified as shown in Figure 6.3 (a), 6-component 29-state weak-rotational-symmetric structure. Comparing with task performance experiment 5.1 presented in the previous chapter (Figure 5.3), the same target structure is used here, but no seed structure is pre-specified. Replicators are sought with novel structures and rules which are capable of "manufacturing" the target structure. Structure/Rule co-evolution took place and produced a set of replicators with both the S-tree and R-tree evolved collectively. Two of the results are shown below. The first replicator produced an S-tree reflecting a seed structure shown in Figure 6.3 (b), which is a 3-component 29-state weak rotational symmetric structure, and an R-tree reflecting execution

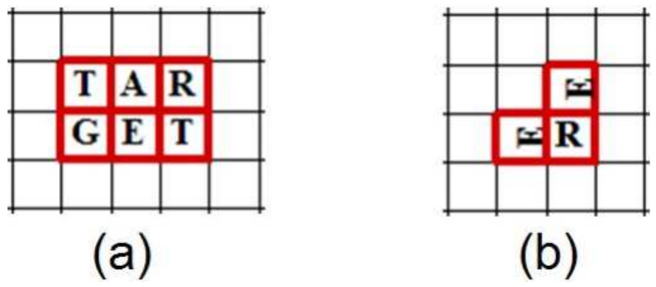


Figure 6.3: Structure/Rule co-evolution, example 6.1-1: (a)the target structure, pre-specified as a 6-component 29-state weak-rotational-symmetric structure, same target structure used in secondary task performance example 5.1 (Figure 5.3); (b)an automatically discovered seed structure, which is a 3-component 29-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree.

results in the cellular space as shown from Figure 6.4 to Figure 6.6. Meanwhile, the second replicator produced an S-tree reflecting a seed structure shown in Figure 6.7 (b), which is a 3-component 29-state weak rotational symmetric structure, and an R-tree reflecting execution results in the cellular space as shown from Figure 6.8 to Figure 6.11.

From Figure 6.4 to Figure 6.6, it can be seen that, when executing the first S-tree/R-tree combination resulted from the co-evolution, at t=1, the initial seed structure turns into a composite structure composing two seed replicas sitting on top of a target structure, which splits into 3 isolated structures at t=2. This fission process deposits the first target structure at the bottom, which then persists, but keeps translating away from the seed cells to the bottom direction, without rotating

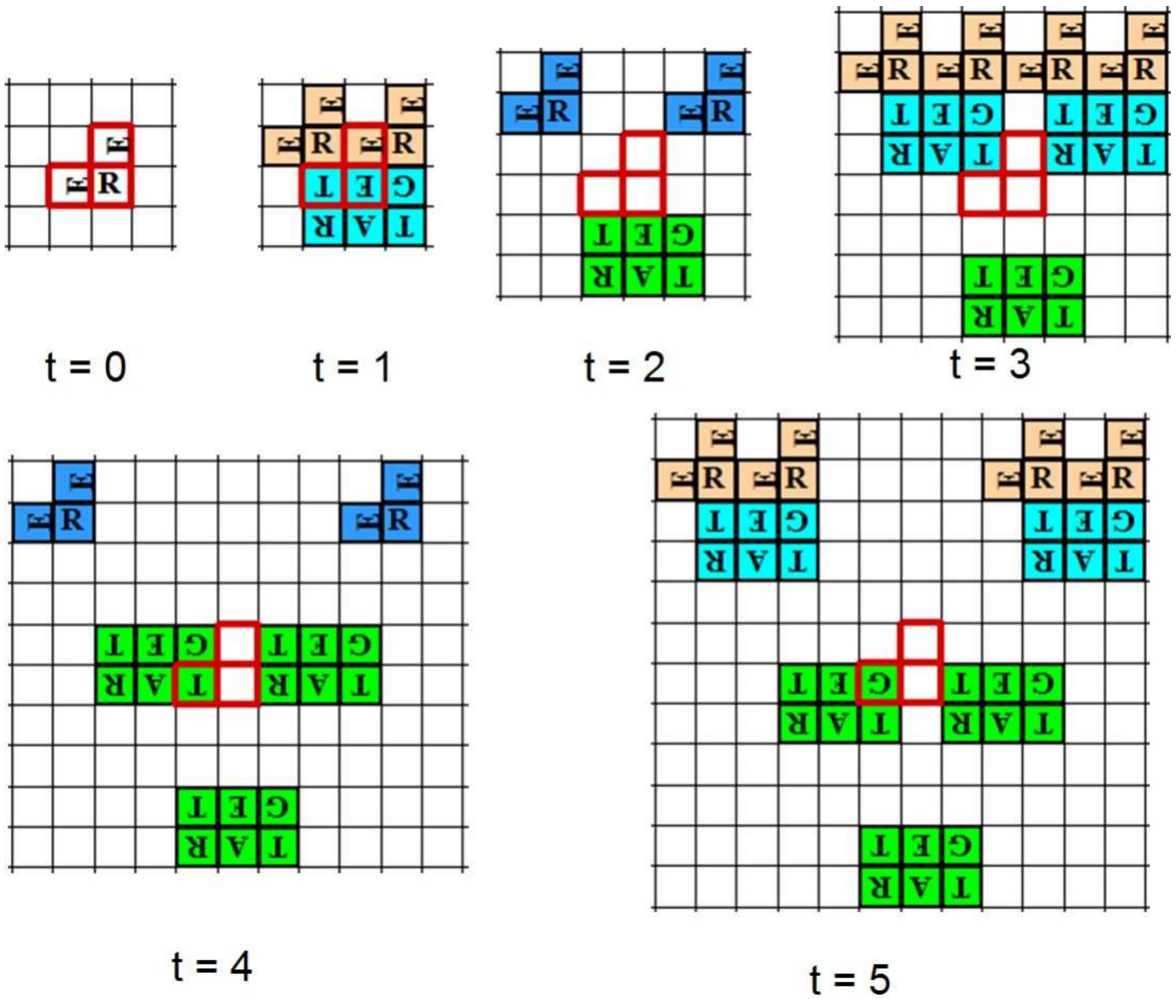


Figure 6.4: Structure/Rule co-evolution, example 6.1-1: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

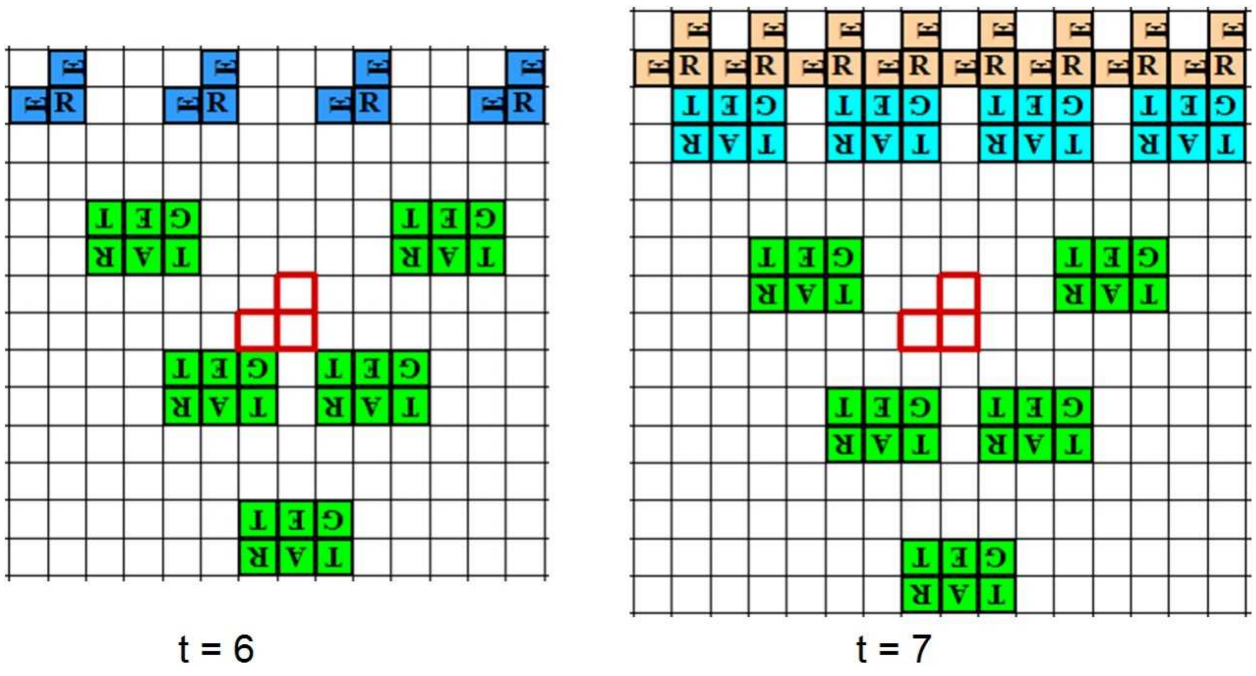


Figure 6.5: Continuation of Figure 6.4 from t=6 to t=7.

itself. More instances of the same composite structure are formed on the top edge (like a horizontal fission wall), which split into more and more seed structures, and deposit more and more target structures below the fission wall. Each of these target structures, once born, persists and keeps translating away from seed cells to the bottom direction. By the time of t=14 (Figure 6.6), there are 8 seed replicas formed on the top, and 19 target structures deposited below them at the same time.

Figure 6.8 to Figure 6.11 illustrates the execution result of a different evolved S-tree/R-tree combination. It can be seen that, the replicator factory has discovered a very different and very interesting way to complete the same task. Here, an equal number of seed replica and target structures can be created and deposited on a spiral curve. At t=1 (Figure 6.8), the seed structure shown at t=0 transforms into an isolated pair of seed and target structures. This pair of structures then persist

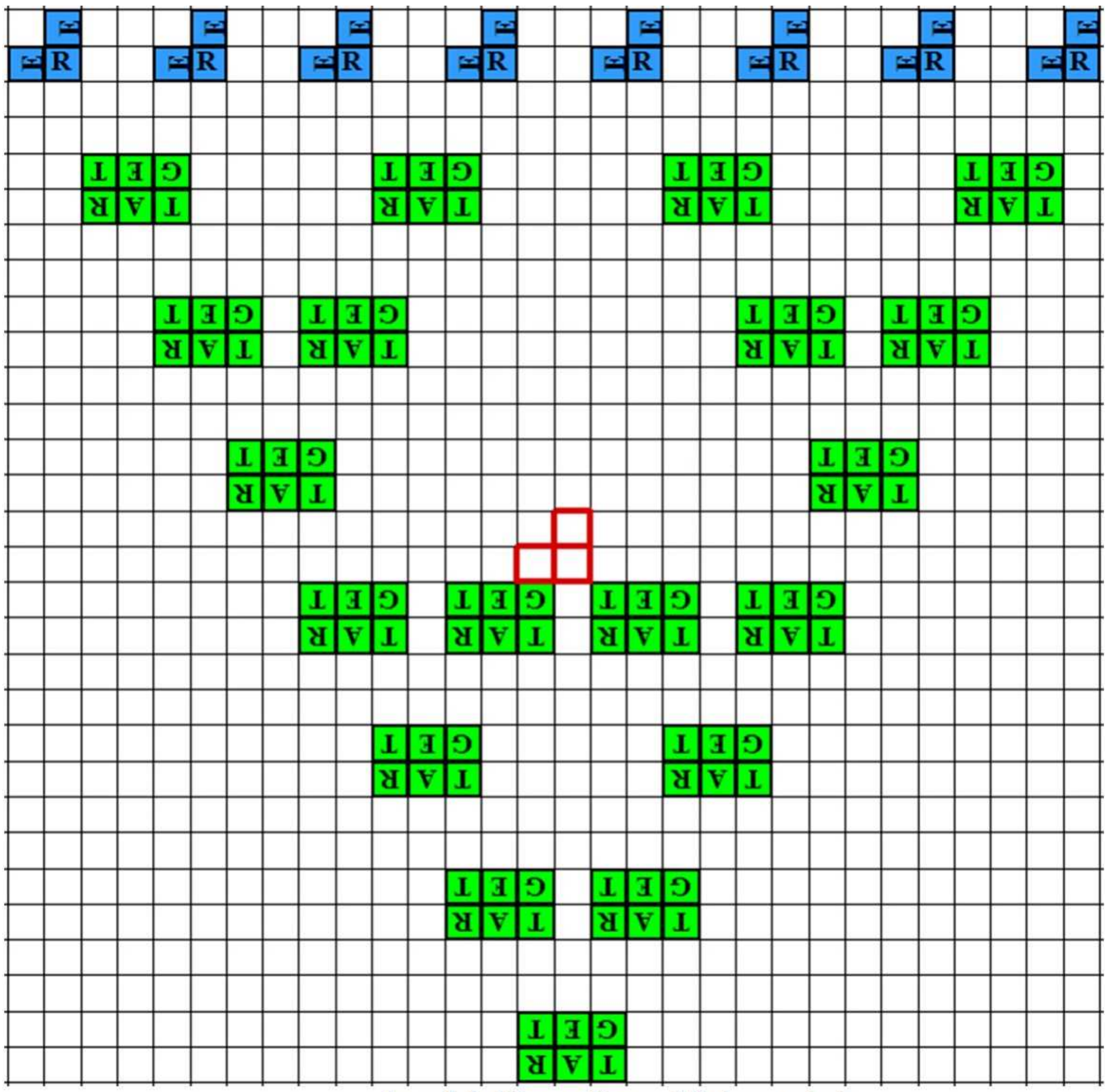


Figure 6.6: Continuation of Figure 6.5 at t=14.

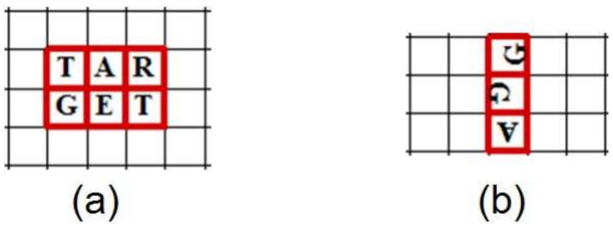


Figure 6.7: Structure/Rule co-evolution, example 6.1-2: (a)the target structure, same target structure as in example 6.1-1 above (Figure 6.3); (b)a different automatically discovered seed structure, which is also a 3-component 29-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree.

and keep spinning 90^o at every subsequent time step. For example, at t=5, it can be seen they resume exactly the same location and orientation as at t=0, after rotating 4 times, or 360^o. The same can be said for t=9, 13, and 17, etc., as shown in Figures 6.10 and 6.11. More importantly, while this pair of structures spin in the center, seemingly they "emit" and launch endless "material" into a spiral curve, which transform into equal and increasing number of seed and target structures, all lined up on this spiral trail. Note that at t=17, 21 seed replicas and 21 target structures can be found on the spiral trail. The spiral can extend infinitively and produce any number of seed and target structures.

In the second co-evolution experiment example, a target structure is pre-specified as shown in Figure 6.12 (a), which is same target structure used in task performance example 5.2 (Figure 5.12) in Chapter 5. A separate run of structure/Rule co-evolution took place and produced a set of evolved replicators of var-

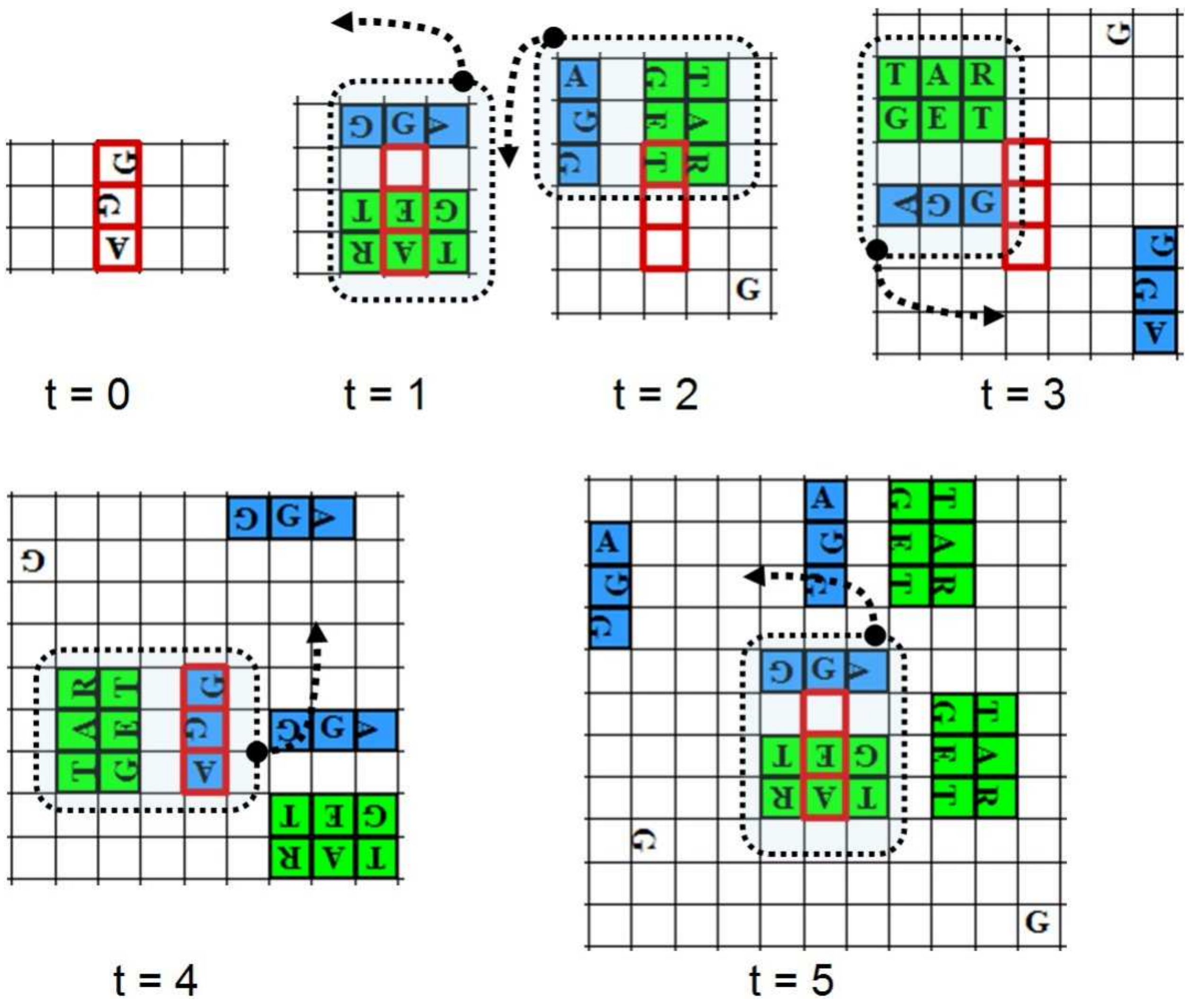


Figure 6.8: Structure/Rule co-evolution, example 6.1-2: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

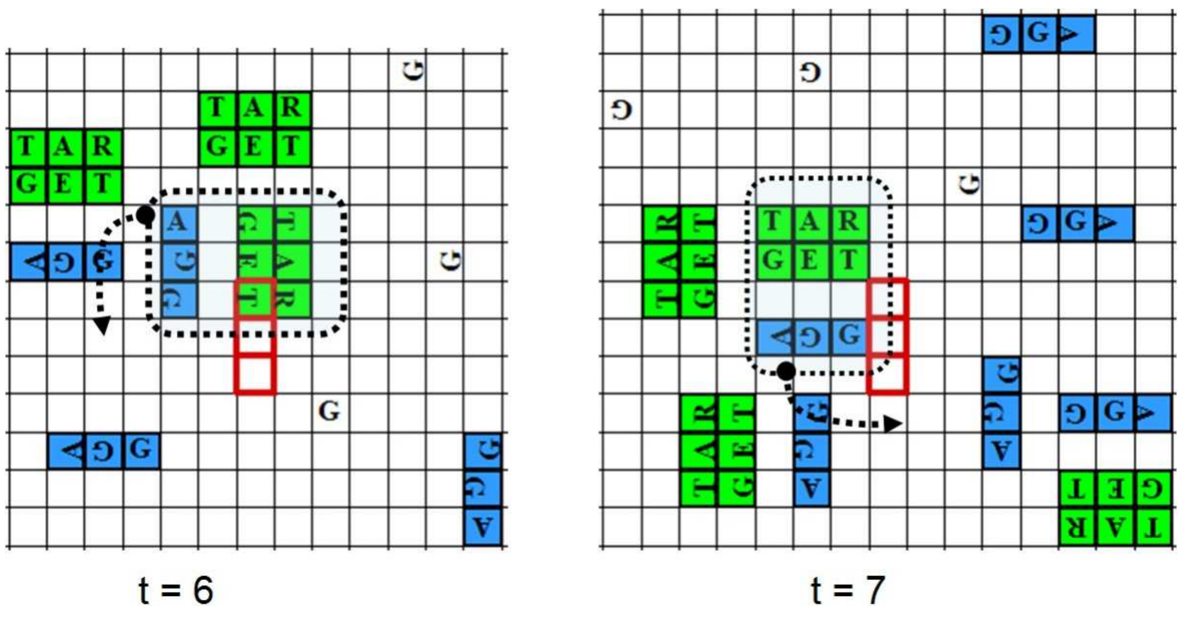


Figure 6.9: Continuation of Figure 6.8 at t=6 and t=7.

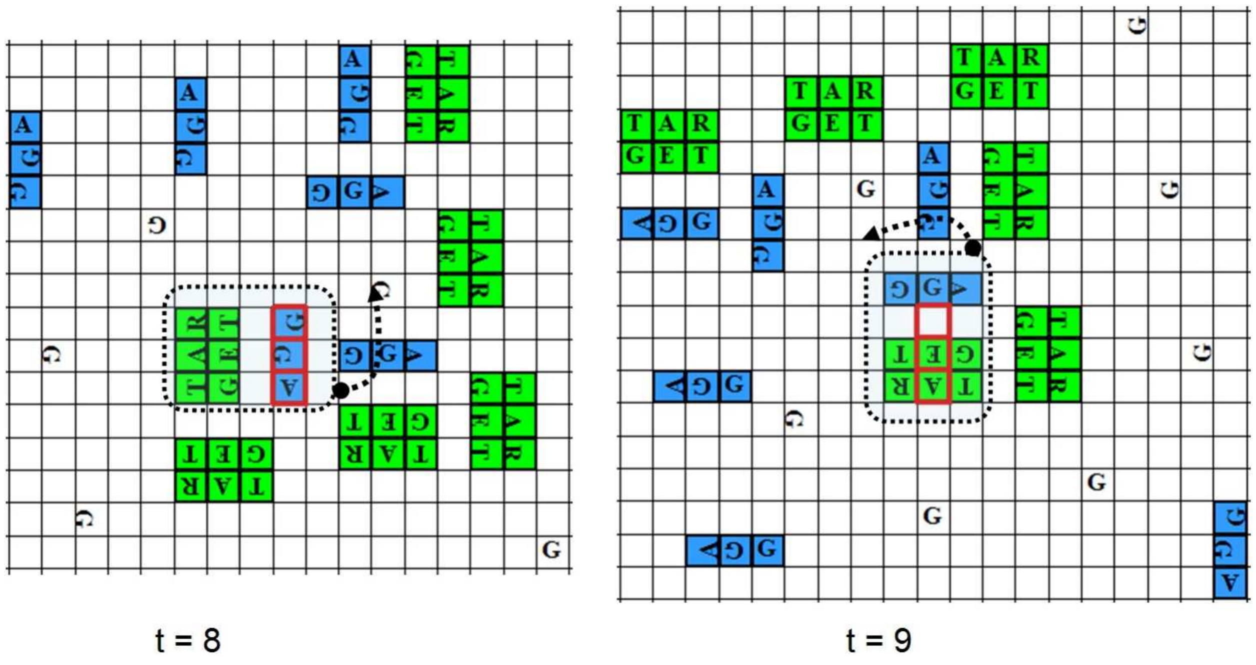


Figure 6.10: Continuation of Figure 6.9 at t=8 and t=9.

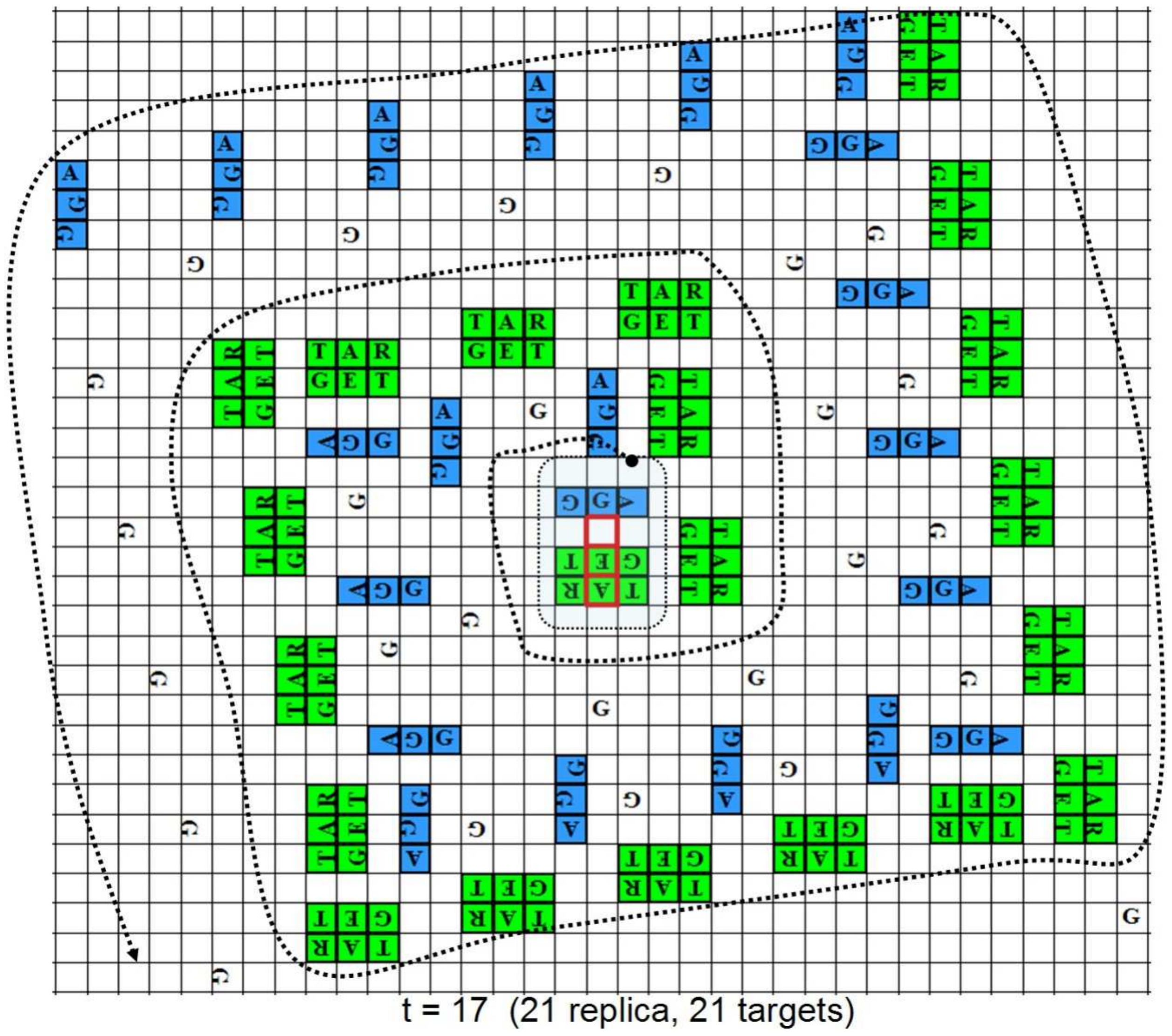


Figure 6.11: Continuation of Figure 6.10 at t=17.

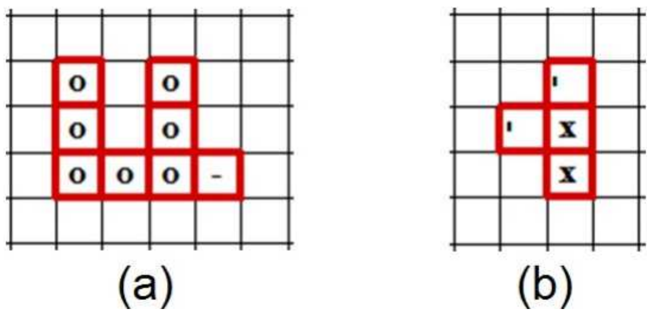


Figure 6.12: Structure/Rule co-evolution, example 6.2: (a)the pre-specified 8-component 7-state target structure, same target structure used in task performance example 5.2 (Figure 5.12); (b)an automatically discovered seed structure, which is also a 4-component 7-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., “manufacture” the given target structure, as the found seed structure self-replicates, as guided by the found R-tree.

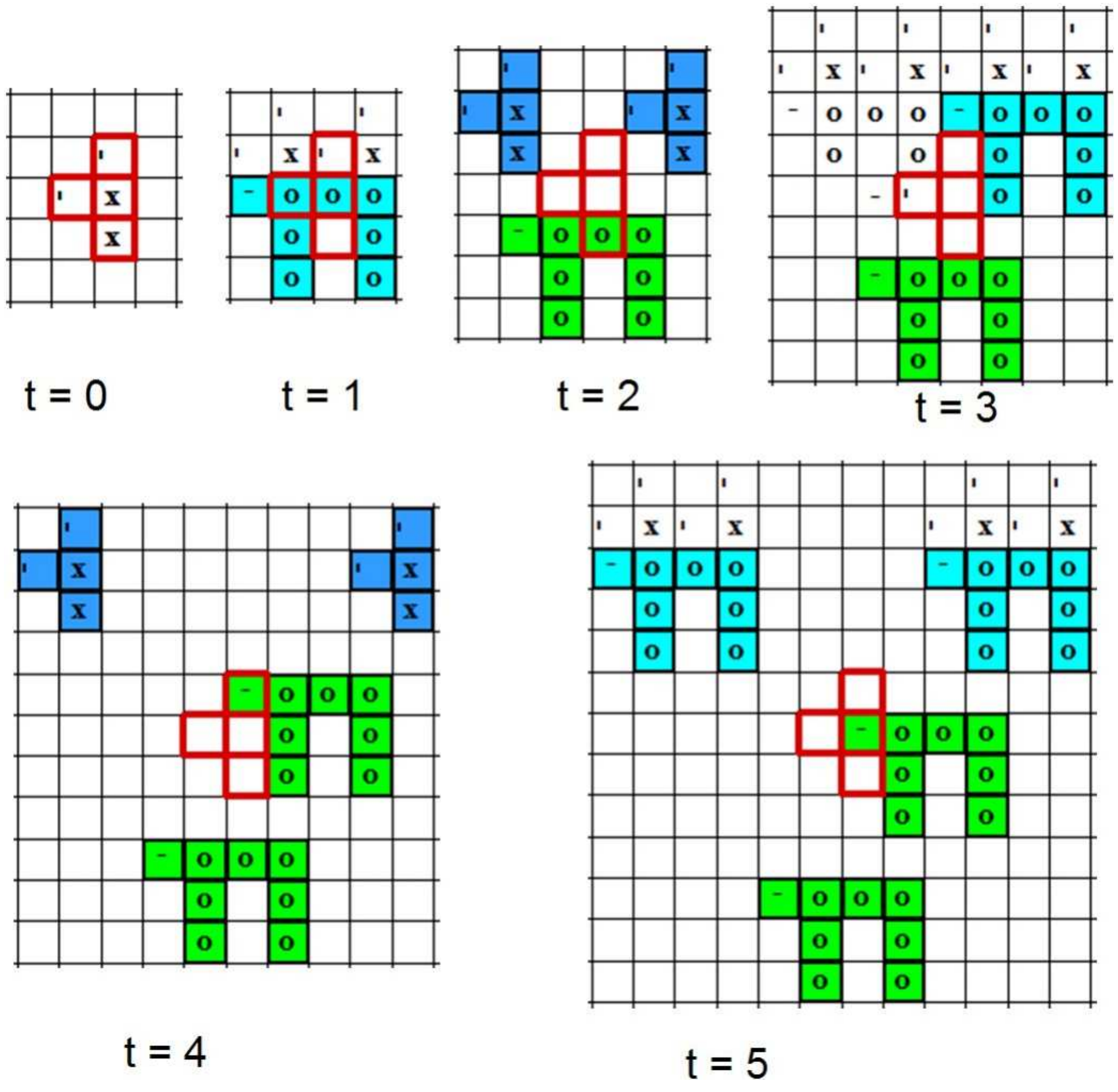


Figure 6.13: Structure/Rule co-evolution, example 6.2: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

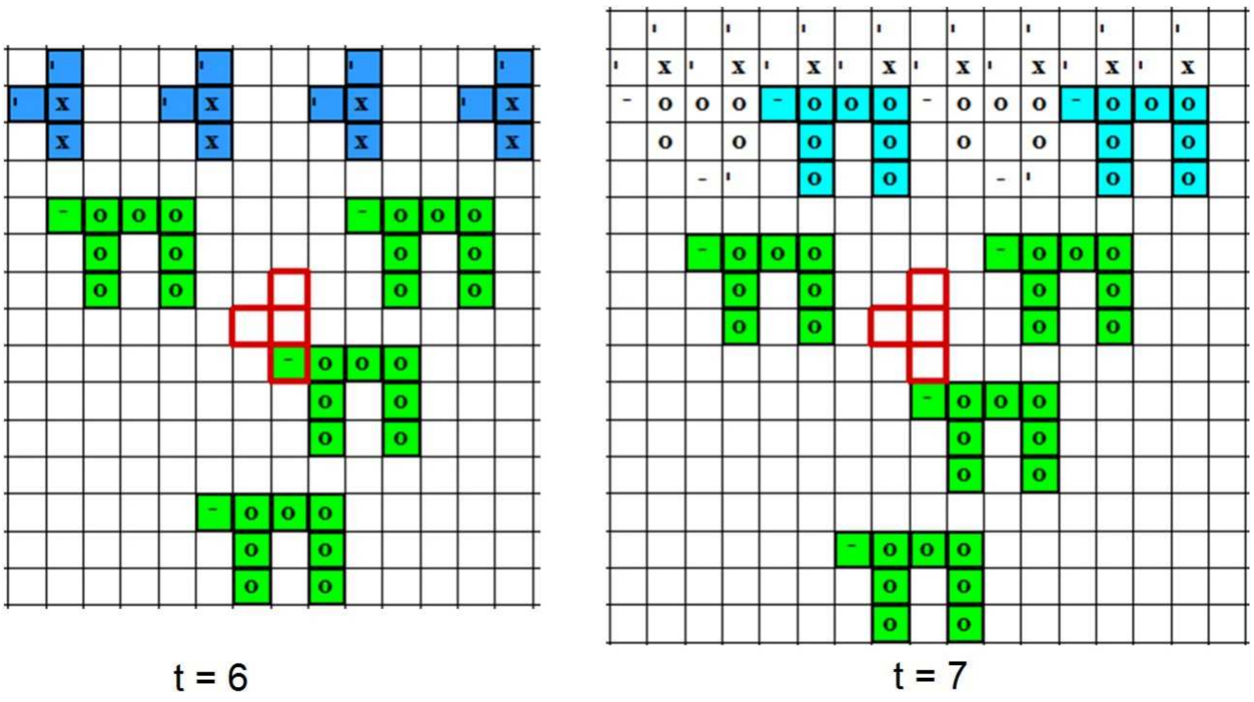


Figure 6.14: Continuation of Figure 6.13 at t=6 and t=7.

ious S-tree/R-tree combinations, with one sample result shown here. The evolved replicator produced an S-tree reflecting a seed structure shown in Figure 6.12 (b), which is a 4-component 7-state weak rotational symmetric structure, and an R-tree reflecting execution results in the cellular space as shown from Figure 6.13 to Figure 6.16. It can be seen that this bigger target structure can also be constructed by the discovered structure in a way which closely resembles structure/rule co-evolution experiment 6.1 above, i.e., the seed structure gets replicated on the top, and target structures get deposited below. Note by the time of t=14, 8 seed replicas and 14 target structures are present in the configuration. This process can carry on infinitively, and produce any number of seed or target structures.

In the third co-evolution experiment example, a target structure is pre-specified

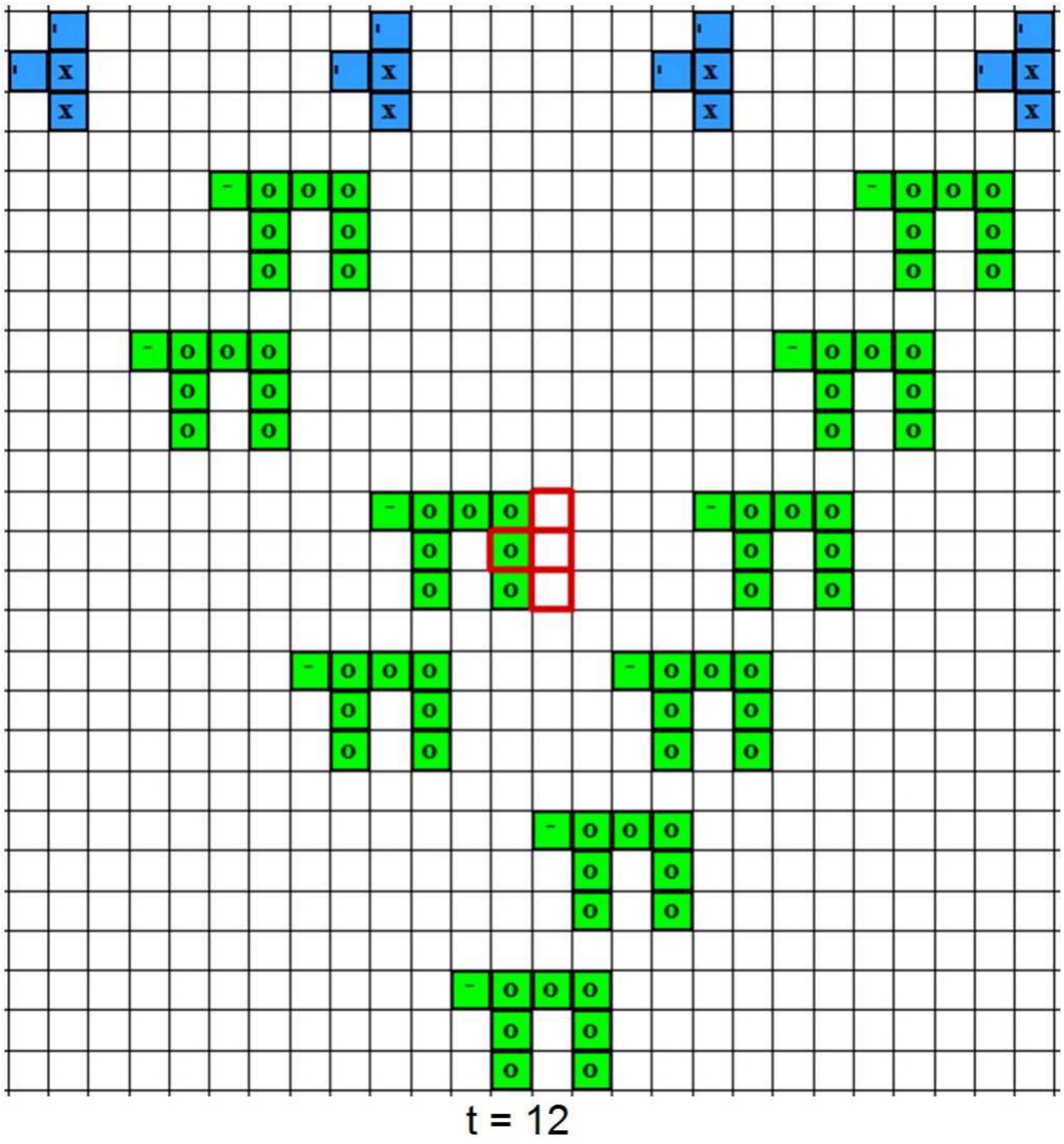


Figure 6.15: Continuation of Figure 6.14 at t=12.

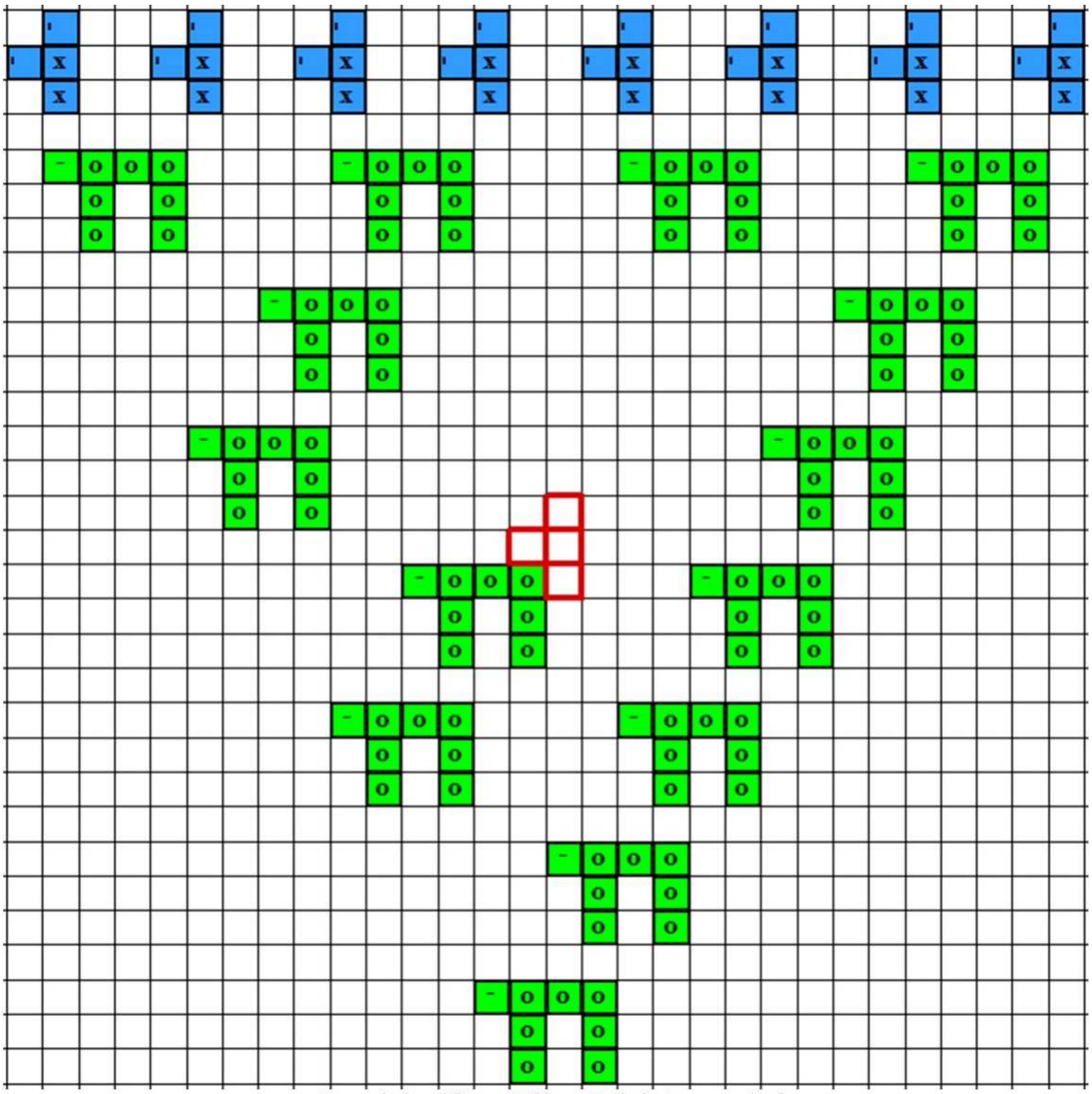


Figure 6.16: Continuation of Figure 6.15 at t=14.

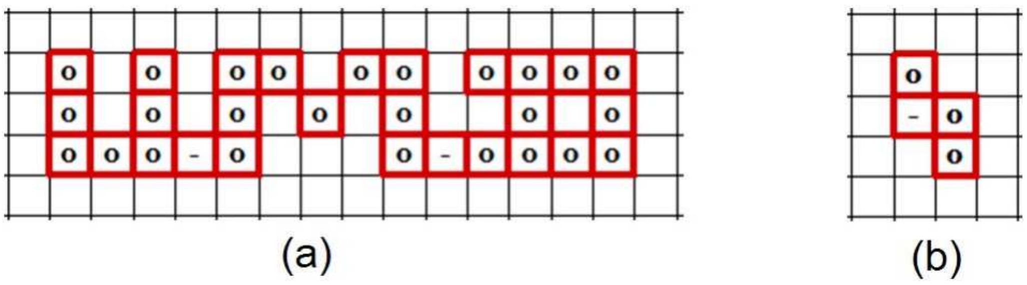


Figure 6.17: Structure/Rule co-evolution, example 6.3: (a)the pre-specified target structure, a very big 28-component 7-state weak-rotational-symmetrical structure that writes out "UMD", the domain name of University of Maryland; (b)an automatically discovered seed structure, which is a 4-component 7-state weak-rotational-symmetric structure. Various S-tree/R-tree combinations were sought to perform a pre-specified task, i.e., "manufacture" the given target structure, as the found seed structure self-replicates, as guided by the found R-tree.

as shown in Figure 6.17 (a), which is a large and complex structure composed of 28 7-state components, and which writes out "UMD", the domain name of University of Maryland. The same pre-specified seed structure, which succeeded in writing out the "U" and "UM" target structures in task performance experiment 2 and 3 in Chapter 5 (see Figure 5.12 and 5.18), was tested and found unable to produce this "UMD" structure. However, by extending the search into structural space with the approach presented in this chapter, a novel structure is discovered which successfully self-replicate, and produce this large structure at the same time, as illustrated below. The structure shown in Figure 6.17 (b), is a 4-component 7-state weak rotational symmetric structure. The R-tree evolved with this structure is executed against the shown structure and produce results shown from Figure 6.18 to Figure

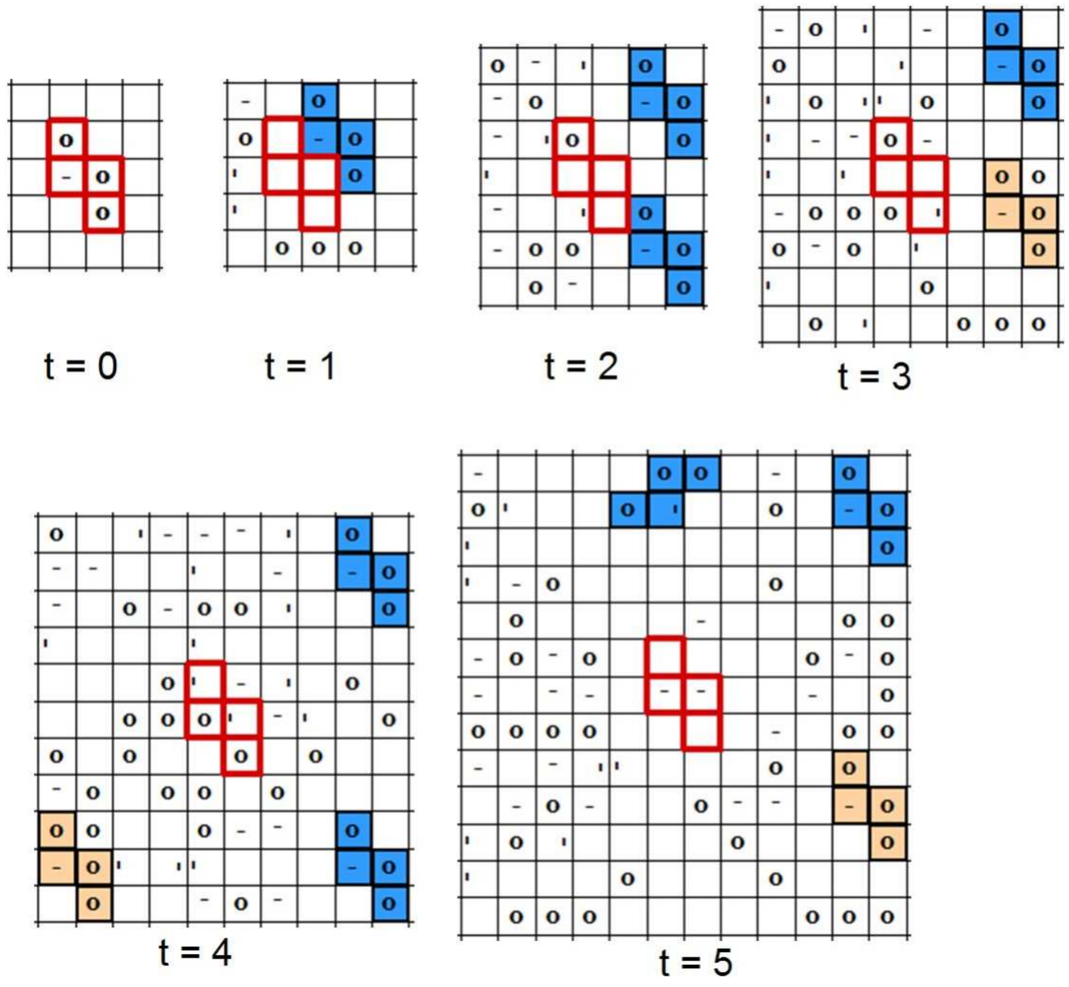


Figure 6.18: Structure/Rule co-evolution, example 6.3: executing an automatically programmed R-tree against the automatically discovered seed structure from t=0 to t=5. Color code convention: non-isolated seed structure marked in yellow, non-isolated target structure in cyan, isolated seed structure in blue, and isolated target structure in lime. The cells covered by the initial seed structure are also always highlighted by red edges at any time step to provide a location correlation.

6.22. It is shown that, even though it took more steps for such a big target structure to appear, it can still be made while the found seeds self-replicate.

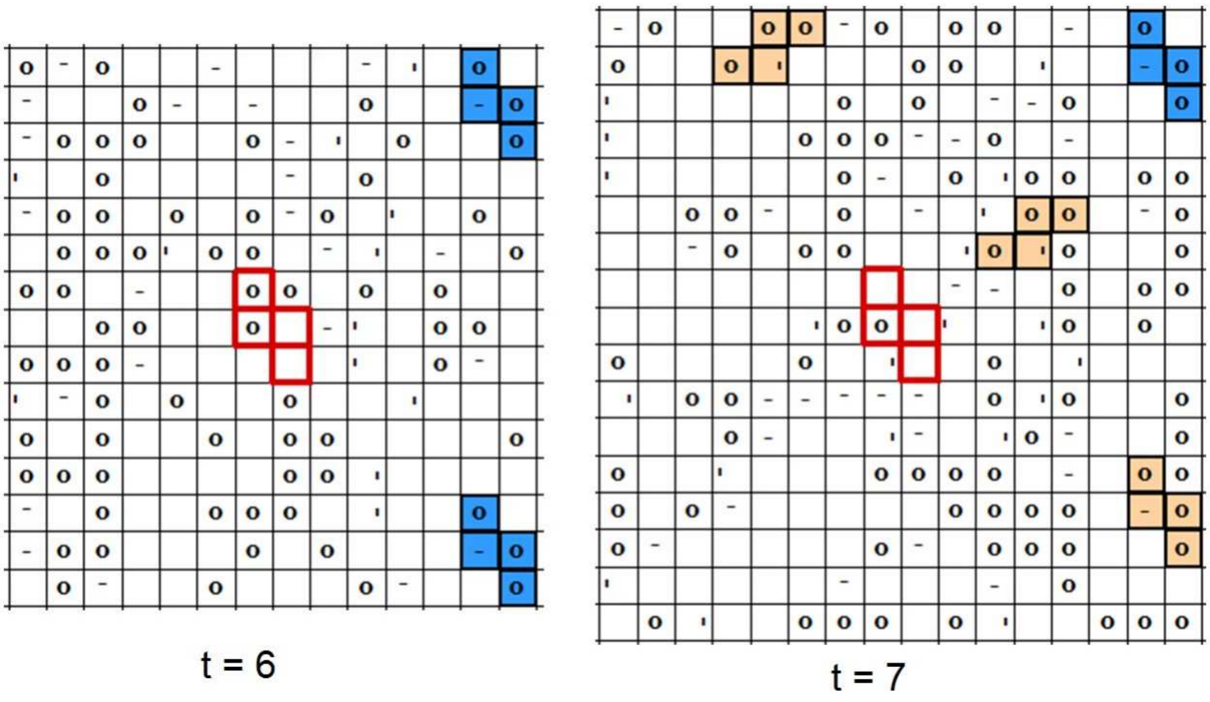


Figure 6.19: Continuation of Figure 6.18 at t=6 and t=7.

6.9 Conclusions

This chapter further extends the multi-objective replicator factory model to be even more general, i.e., enabling it to discover its own replicators that also carry out a pre-specified task without being given a seed structure. In such a generalized replicator factory model, the input is a task to be performed, such as a target structure to be constructed, and the output is a replicator, of which both the seed structure and associated CA rules are to be automatically programmed toward the successful performance of the given task. To implement such a general model, for

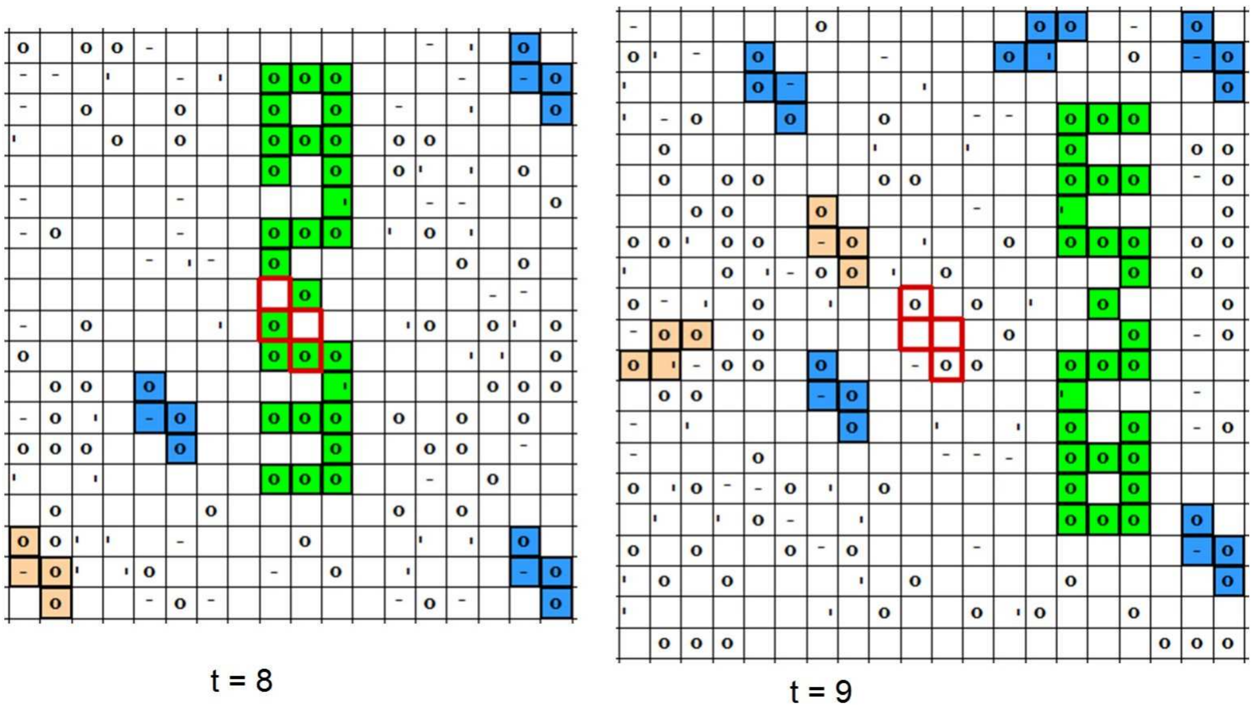


Figure 6.20: Continuation of Figure 6.19 at t=8 and t=9.

the first time, structure/rule co-evolution is introduced and established, on top of the multi-objective replicator factory model.

The details presented in this chapter demonstrate how S-trees can be initialized with random structures, how S-trees can evolve with GP genetic operators, how evolved S-trees can be re-constructed as CA structures which can then be simulated by executing the associated concurrently evolving R-trees, and how multi-objective fitness measures can be evaluated from the simulation results collectively produced by the co-evolving structure/rules. It is then shown that this approach effectively extends the exploration from the rule space only to both structural space and rule space, in the seeking of multiple non-dominated solutions to a pre-specified task.

The experimental results produced from the generalized multi-objective co-

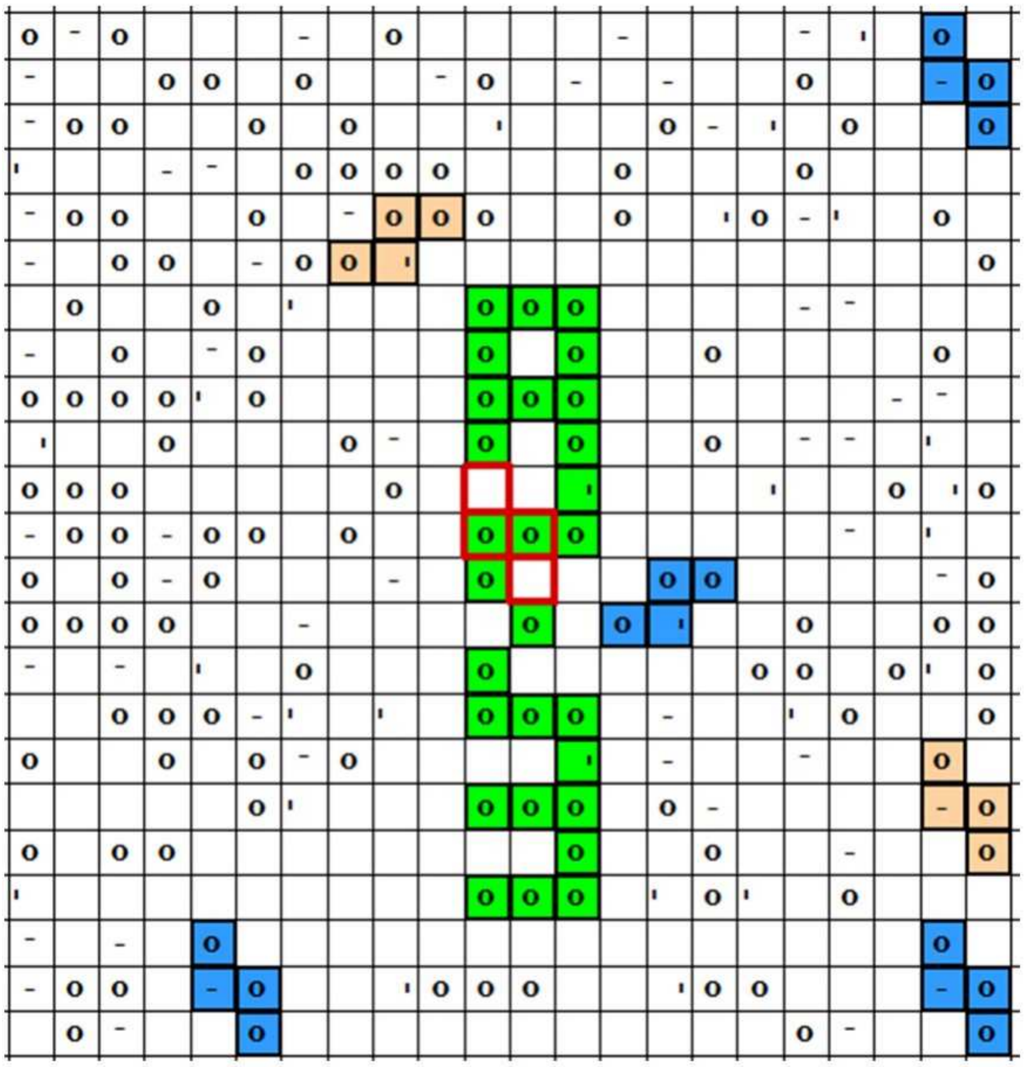


Figure 6.21: Continuation of Figure 6.20 at t=10.

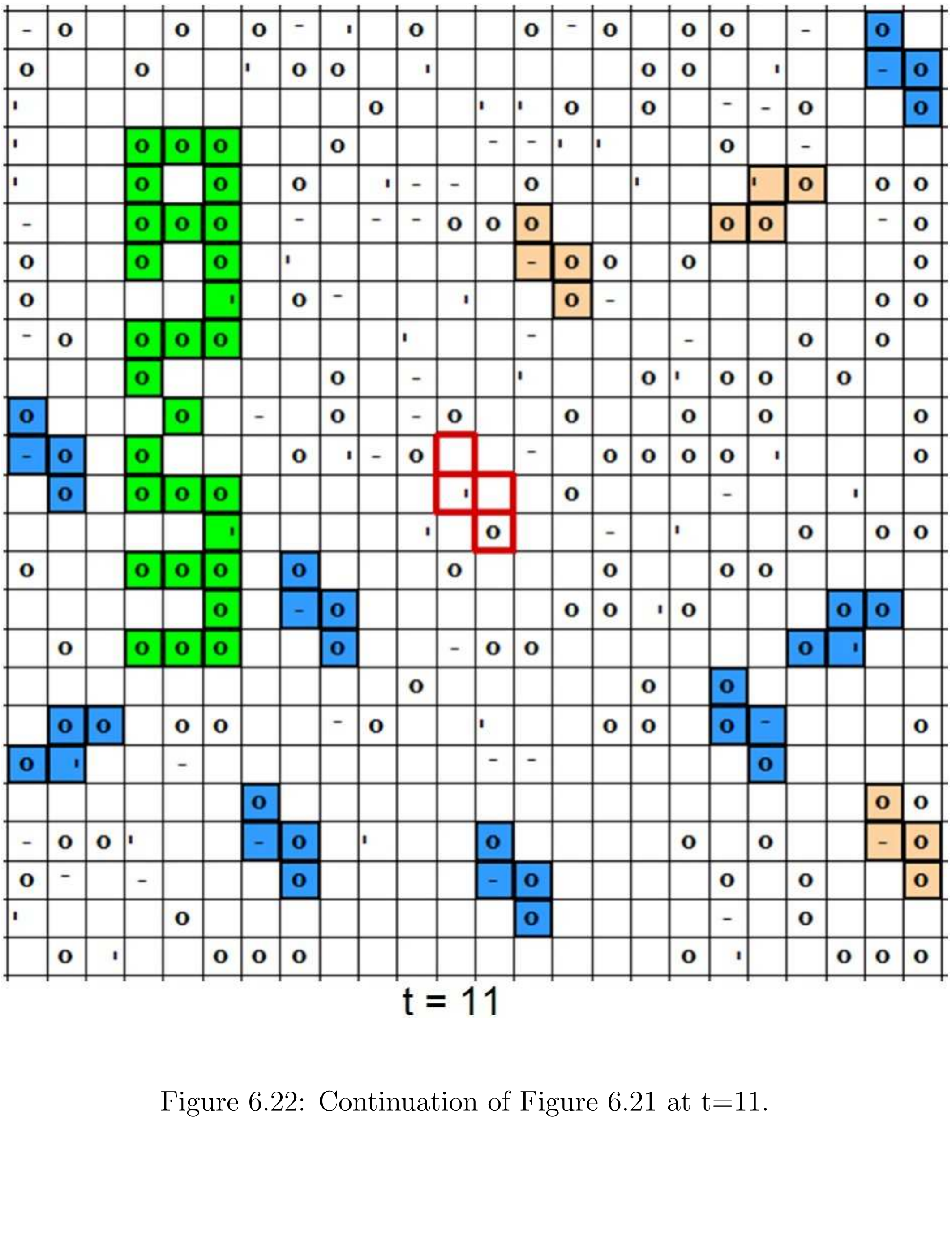


Figure 6.22: Continuation of Figure 6.21 at t=11.

evolution model have provided some results showing that it is indeed possible to automatically program the cellular automata with unspecified seed structures that will evolve the capability of “manufacturing” arbitrary target structures at the same time when the found structures replicate. This indicates that multiple diversified solutions can concurrently be yielded, with each solution reflecting different strategies, in terms of different seed structures/rules combinations, in order to perform the same given task. Typically, the seed structure keeps replicating and deposits persistent target structures behind. The new model is found capable of writing out persistent “UMD”, with multiple automatically programmed seeds, even though it was previously found that a pre-specified seed structure used in Chapter 5 experiments was unable to do the same. A small seed structure may be easier to replicate, but potentially needs more effort to “manufacture” a superior target. Likewise, a larger seed structure may require less time steps to produce a target, but requires more effort for itself to replicate. This dilemma potentially makes it hard to pre-specify an optimal seed structure to perform a given task. However, the general approach presented in this chapter makes it possible for alternative seed structures to be automatically approached through evolution.

In sum, the results presented in this chapter show that structure/rule co-evolution is possible, capable of producing interesting results, and merits much further study. Many variations and their effects can be further investigated, such as 1) adopting other types of S-tree genetic operators, such as domain-based or supervised genetic operators, for structure evolution; 2) introducing domain-based fitness measures for structure evolution, and/or 3) introducing multi-staged co-evolution so that in different stages S-tree and R-tree genetic operators can be favored with different weights.

Chapter 7

Conclusions and Future Work

Given the local concurrent computations in CA, in the past it has proved very difficult in general to manually program their transition function when the desired computation requires global communication and behavior [1], as with self-replication. This difficult issue of creating models of self-replicating structures in CA has greatly limited the number of different self-replicating structures designed and studied to date and contributed to the lack of a systematic study of their properties. To address this difficult issue, past studies have focused on mainly two approaches. One is creating CA systems capable of universal construction with self-replication thus being only as a special case. These so-called universal constructors are highly complex and marginally realizable, and have mainly been used for theoretical studies [48, 71, 11]. The second approach focuses on manual design of simpler and smaller self-replicating loops [9, 34, 57], which are manually crafted to do nothing but self-replication. The resultant CA models do not allow arbitrary structures to replicate; instead, they share the same restriction inherited from Langton's self-replicating loop: requiring the structure to be a simple, square (or rectangular) shape to enable their replication [61].

This research presented in this dissertation is the first work adopting GP for discovery of self-replication models in CA. Not only was it demonstrated that GP can be used to automatically program CA to produce self-replication of arbitrary structures, but also a whole class of new replicators were discovered, which are qualitatively different from past models. Using GP to automatically program arbitrary, initially non-replicating structures to self-replicate removes the restriction that replicas must have a specific loop-like structure, qualitatively reducing the time cost of replicator construction compared to past manual methods, and creating replicators that resemble biological cell mitosis more than they do replicators discovered in past studies (universal constructors and loops). Here, automatically discovered replicators can construct themselves very quickly, in a fission-like, rotational, and/or spiral process. Surprisingly, some of these discovered replicators can self-replicate within only one time step, representing a speed which was not known a priori to even be possible (Figure 3.15).

Especially, this study created an unambiguous and universal tree-like representation for both arbitrary structures and rule tables, which made the efficient and effective application of GP possible in CA programming, even though these tree encodings do not represent conventional sequential computer programs as in traditional GP. The structure tree can be viewed as a representation of a desired global configuration, and the rule tree can be viewed as the representation of local state-transition rules. Local rules can now efficiently evolve in the form of trees, and receive fitness measures simply evaluated in terms of how well, when they are simulated in CA spaces, they express the desired global configuration. Previous heuristic fitness functions do not provide precise and universal fitness measuring for local CA rules in terms of produced global results, as they only exploit partial structural information (such as component density) and rely on the specific knowledge of the given structure [37]. Thus this work provides an unprecedented demonstration of how precise fitness assignment can be used for full or partial matching structures at any GP stage.

More importantly, the tree based evolution and fitness measuring approach presented here can be universally applied on arbitrary, future structures without requiring any knowledge about the structure a priori. This is because of the new approach's capability of converting an arbitrary structure into a common tree-based structure, which can be fully exploited by an existing GP evolutionary system, built prior to any knowledge of the structure, to retrieve the complete structural information encoded in the S-tree, such as details about every Moore neighbor of every component in the structure. By retrieving such complete structural information from the tree encoding, in the efficient way that is allowed by representation of a structure as a MST (minimum spanning tree), and comparing them to a given configuration, we can tell precisely how well the current configuration satisfies the expected global computation. Thus, this approach provides a feasible and efficient GP programming model toward global CA computation, despite the local concurrent computations in CA.

The development of the time-effective method for generating self-replicating structures presented here opened up the possibility for studying replicator configuration properties in a systematic way. It proved possible to automatically generate whole families of self-replicating structures, allowing one to systematically investigate the properties of replicating CA structures as one varies the initial configuration, size, shape, symmetry, and allowable states. This work showed that the number of GP generations, computation time, and number of rules required by an arbitrary structure to self-replicate are positively correlated with the number of components, configuration shape, and allowable states in the initial configuration, but inversely correlated with the presence of repeated components or sub-structures, and seed symmetry. This leads to the conclusion that the properties of the resulting replicators can be predicted in part a priori. In summary, the anticipated impact of this study includes the creation of an automated approach for creating novel self-replicators, the discovery of a whole new class or family of self-replicators of arbitrary structures, and a deeper knowledge of the properties of self-replicating cellular automata models.

After satisfying the primary goal of this study — establishing that genetic programming provides a powerful method for creating CA models of self-replication — this dissertation research extended its study to the examination of whether the new approach can also support finding self-replicators that perform simple secondary tasks. In the past, Tempesti and subsequently others showed that it is possible to add additional computational capabilities to simple self replicating loops, allowing them to carry out some limited "secondary" tasks [68, 53]. However, these past replicators with secondary capabilities were limited in that they are implemented on specific, non-arbitrary, and manually-designed seed structures; they depend on manual, pre-written computer programs; and they are implemented by embedding the pre-written program into the specific seed structure, in other words, at the cost of altering (and changing the complexity of) the seed structure itself. The results shown in this study suggest that self-replicators that are created by GP for arbitrary initial structures, are also capable of carrying out a simple secondary task, such as writing out "UM", the acronym for University of Maryland. Compared to the past methods, the GP approach developed in this dissertation has the following advantages: 1) it generalizes producing replicators that carry out a secondary task to arbitrary self-replicating structures, from just the specific manually-designed loops used in the past; 2) it automatically programs the CA to support both self-replication and any needed secondary computation, in contrast to the need for pre-written manual programs as in the past; 3) it carries out such a secondary task with the use of the same initial seed structure, in contrast to in the past where the seed structure has to be altered by embedding a manually-written program in it; and 4) both self-replication and secondary computation are programmed in a single GP run, with the possibility of yielding multiple non-dominated solutions, each representing a different strategy for carrying out the same given task. For example, it was shown that multiple simple structures can "manufacture" structures much more complex than themselves, such as writing out "UM" consistently and continuously, at the same time as replication, without involving pre-writing and embedding manual programs. This represents another impact of this dissertation, i.e., introducing multi-objective evolution into self-replication CA models for the first time and providing a new approach to creating arbitrary self-replicators capable of concurrent task performance without requiring embedding manual programs.

Finally, this is also the first work to create a structure/rule co-evolution system, on top of the multi-objective GP paradigm, to discover, without being given seed structures, replicators that could also perform a pre-specified secondary task. In such a model, only the task to be done was specified. The new model was able to automatically search for and find an appropriate seed structure as well as associated state transition rules, which cooperatively provide the capability to perform the given task as the seed structure replicates itself. To seek a solution to a given task, the new model extended the search from rule space only to both rule space and structural space. This represents a novel way for creating CA models toward task performance. The experimental results indicate that multiple diversified solutions can concurrently be yielded, with each solution reflecting different strategies in terms of different seed structures/rules combinations. Typically, the seed structure keeps replicating and deposits persisting or moving target structures behind. The new model was found capable of writing out "UMD", with an automatically programmed seed, even though previous efforts to manually create a pre-specified seed structure were unsuccessful. Thus, when it is hard to pre-specify an optimal seed structure to perform a given task, alternative seed structures may sometimes be automatically discovered through evolution using GP.

The approach formulated in this work uses a uniform tree representation to encode a global structural configuration and local interaction rules. It defines genetic operators for one or both representations, and then lets one or both efficiently evolve, automatically guided by fitness measures simply evaluated in terms of how well, when the rule-representation is simulated, it expresses or exhibits the global structural representation. This represents a general evolutionary CA programming approach which can be readily applied to solve many other complex problems in cellular automata [79, 4, 2, 74, 23], such as games [6, 14, 18, 19], behavioral and social problems [20, 25, 17, 73].

Among the many issues that might be examined in the future, several appear to be of particular importance. These include the further development of programmable self-replicators for real applications, and a better theoretical understanding of both the time/space complexity of the GP paradigm and the principles of self-replication in cellular automata spaces. Domain-based fitness functions and novel S-tree genetic operator, could also be further introduced and examined in the context of the co-evolution system described in this dissertation. More general and flexible cellular automata environments, such as those having non-uniform transition functions or novel interpretations of transition functions, merit exploration. It has already proved possible, for example, to create simple self-replicating structures in which a cell can change the state of neighboring cells directly [36]. Also, from the perspective of realizing physically self-replicating devices, exchange of information between the modeling work described here and ongoing work to develop self-replicating molecules/nanotechnology is important. Closely related to this issue is ongoing investigation of the feasibility of electronic hardware directly supporting self-replication [51, 87]. If these developments occur and progress is made, I foresee a productive future for the development of a technology of self-replicating systems.

Bibliography

[1] 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.

[2] Per Bak, "*How nature works: the science of self-organized criticality*", Springer-Verlag New York, Inc., 1996.

[3] Banzhaf, W., Nording, P., Keller, R.E., Francone, F.D. (1997), *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*, Morgan Kaufmann

[4] Yaneer Bar-Yam, "*Dynamics of Complex Systems*", Addison-Wesley, 1996.

[5] S. C. Benjamin and N. F. Johnson. A Possible Nanometer-scale Computing Device based on an Adding Cellular Automaton. *Applied Physics Letters,* 70(17):2321-2323, 1997.

[6] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning ways for your Mathematical Plays*, volume 2. Academic Press, 1984.

[7] J.-L. Beuchat and J.-O. Haenni. "von Neumann's 29-state cellular automaton: A hardware implementation." *Logic Systems Laboratory*, 1997. (submitted for publication).

[8] William R. Buckley and Amar Mukherjee. Constructibility of Signal-Crossing Solution in von Neumann 29-state Cellular Automata. *5th International Conference in Computational Science, proceesings, part II.* 395-403. 2005.

[9] J. Byl. "Self-Reproduction in small cellular automata." *Physica D*, Vol. 34, pages 295-299, 1989.

[10] Chou H, Reggia J. Problem solving during artificial selection of self-replicating loops. *Physica D 115*, 1998, 293-312.

[11] E. F. Codd. *Cellular Automata.* Academic Press, New York, 1968.

[12] Cramer, Nichael Lynn (1985), "A representation for the Adaptive Generation of Simple Sequential Programs" in *Proceedings of an International Conference on Genetic Algorithms and the Applications*, Grefenstette, John J. (ed.), CMU

[13] J. P. Crutchfield and N. H. Packard. Symbolic Dynamics of Noisy Chaos. *Physica D*, 7:201-223, 1983.

[14] K. Culik and S. Dube. An Efficient Solution of the Firing Mob Problem. Theoretical Computer Science, 91:57-69, December 1991.

[15] A. Deutsch and S. Dormann. Cellular Automaton Modeling of Biological Pattern Formation, volume in press. Birkhuser Boston Inc., 2003.

[16] Michael P. Fourman. Compaction of symbolic layout using genetic algorithms. In John J. Grefenstette, editor. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 141-153,1985.

[17] Judy Frels, Debra Heisler, James Reggia, and Hans-Joachin Schuetze. A Cellular Automata Model of Competition in Technology Markets with Network Externalities. *5th International Conference in Computational Science, proceesings, part II.* 378-385. 2005.

[18] M. Gardner. The Fantastic Combinations of John Conway's New Solitaire Game 'Life'. *Sci. Am.*, 223:120-123, 1970.

[19] M. Gardner. On Cellular Automata Self-reproduction, the Garden of Eden and the Game of 'life'. *Sci. Am.*, 224:112-117, 1971.

[20] R. J. Gaylord and L. D'andra. *Simulating Society: A Mathematica Toolkit for Modelling Socioeconomic Behavior*. Springer: New York, NY, 1998.

[21] Goldberg, David E (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*

[22] D. M. Gordon. The Development of Organization in An Ant Colony. *American Scientist*, 83:50-57, Jan-Feb 1995.

[23] Hermann Haken, *Advanced Synergetics: Instability Hierarchies of Self-Organizing Systems and Devices*, Springer Verlag, 1983

[24] Harvey, Inman(1992), Species Adaptation Genetic Algorithms: A basis for a continuing SAGA, in '*Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*', F.J. Varela and P. Bourgine (eds.), MIT Press/Bradford Books, Cambridge, MA, pp. 346-354.

[25] R. Hegselmann and A. Flache. Understanding Complex Social Dynamics: A Plea for Cellular Automata Based Modelling. *Journal of Artificial Societies and Social Simulation*, 1(3), June 1998.

[26] Holland, J.H., *Adaptation in Natural and Artificial Systems*,University of Michigan Press, USA, 1975

[27] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd Edition. Addison-Wesley

[28] E. A. Jackson. *Perspective of Non-Linear Dynamics 2*. Cambridge University Press, 1991.

[29] Koza, John (1992), *Genetic Programming: On the Programing of Computers by Means of Natural Selection*

[30] Koza, J.R. (1990), Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, *Stanford University Computer Science Department technical report STAN-CS-90-1314*. A thorough report, possibly used as a draft to his 1992 book.

[31] Koza, J.R. (1994), *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press

[32] Koza, J.R., Bennett, F.H., Andre, D., and Keane, M.A. (1999), *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann

[33] Langdon, W. B., Poli, R. (2001), *Foundations of Genetic Programming*, Springer-Verlag

[34] Langton C, Self-Reproduction in Cellular Automata, *Physica D*, 10, pp. 135-144, 1984.

[35] C. G. Langton. "Studying artificial life with cellular automata." *Physica D*, Vol. 22, pages 120-149, 1986.

[36] J. D. Lohn and J. A. Reggia. "Discovery of self-replicating structures using a genetic algorithm." *Proceedings of 1995 IEEE International Conference on Evolutionary Computation* (ICEC'95), pages 678-683, 1995.

[37] J. D. Lohn. "Automated discovery of self-replicating structures in cellular space automata models." Dept. of Computer Science Tech. Report CS-TR-3677, Univ. of Maryland at College Park, August 1996.

[38] Lohn J, Reggia J. Automated discovery of self-replicating structures in cellular automata.' I*EEE Trans. Evol. Comp.*, 1,1997, 165-178.

[39] F. B. Manning. An Approach to Highly Integrated, Computer-maintained Cellular Arrays. *IEEE Trans. on Computers*, C-26:536-552, 1977.

[40] R. D. McLeod, P. Hortensius, R. Schneider, H. C. Card, G. Bridges, and W. Pries. CALBO - Cellular Automaton Logic Block Observation. In Canadian Conference on VLSI, November 1986.

[41] Daniel Merkle, Martin Middendorf, and Alexander Scheidler. Modelling Ant Brood Tending Behavior with Cellular Automata. *5th International Conference in Computational Science, proceesings, part II.* 412-419. 2005.

[42] Mitchell, Melanie, (1996), *An Introduction to Genetic Algorithms*, MIT Press, Cambridge, MA Addison-Wesley

[43] Morita, K, Imai K. Self-reproduction in a reversible cellular space. T*heoretical Comp. Sci.*, 168, 337-366, 1996.

[44] Morita K, Imai K. Simple self-reproducing cellular automata with shape-encoding mechanism. In Langton C, Shimohara K, eds, P*roc Fifth Intl Workshop on Synthesis and Simul. Living System*, MIT Press, 1997, 450-457.

[45] G. Mrugalski, J. Rajski, and J. Tyszer. Cellular Automata-based Test Pattern Generators with Phase Shifter. *IEEE Trans. on CAD*, 19(8):878-893, August 2000.

[46] Nehaniv, C. L. 2002. Evolution in Asynchronous Cellular Automata, In R.K. Standish, M.A. Bedau, and H.A. Abbass(eds.) *Proceedings of the Eighth International Conference on Artificial Life*, MIT Press, pp. 65-78.

[47] J. von Neumann, "The General and Logical Theory of Automata." In [Taub61], pp. 288-328, 1951.

[48] J. von Neumann. Theory of Self-Reproducing Automata. University of Illinois Press, Illinois, 1966. Edited and completed by A. W. Burks.

[49] H. Nishio and Y. Kobuchi. Fault Tolerant Cellular Space. *J. Comput. Syst. Science*, 11:150-170, 1975.

[50] S. Omohundro. Modeling Cellular Automata with Partial Differential Equations. *Physica D*, 10:128-134, 1984.

[51] Freitas R & Merkle R, *Kinematic Self-Replicating Machines*, Landes, 2004.

[52] L. S. Penrose. “Self-reproducing machines.” *Scientific American*, Vol. 200, No. 6., pages 105-114, June 1959.

[53] Perrier J, Sipper M, Zahnd J. Toward a viable self-reproducing universal computer. *Physica D*, 97, 335-352, 1996.

[54] U. Pesavento. “An implementation of von Neumann’s self-reproducing machine.” *Artificial Life Journal*, Vol. 2, No. 4, pages 337-354, 1995. The MIT Press, Cambridge, MA.

[55] Poli R and Langdon W, Schema theory for genetic programming with one-point crossover and point mutation, *Evolutionary Computation*, 6, 231-252, 1998.

[56] J. Rebek, Jr. “Synthetic self-replicating molecules.” *Scientific American*, Vol. 271, No. 1, pages 48-55, July 1994.

[57] Reggia J, Armentrout S, Chou H, Peng Y, Simple Systems That Exhibit Self-Directed Replication, *Science*, 259, 1282-1288, 1993.

[58] Richards F, Meyer T, Packard N. Extracting cellular automaton rules directly from experimental data, *Physica D*, 45, 1990, 189-202.

[59] Salzberg C, Antony A, and Sayama H: Complex genetic evolution of self-replicating loops, Artificial Life IX: *Proc. of the Ninth Internat. Conf. on the Simulation and Synthesis of Living Systems*, 262-267, 2004, MIT Press.

[60] H. Sayama. Introduction of Structural Dissolution into Langton’s Self-Reproducing Loop. *Artificial Life VI: Proceedings of the Sixth International Conference on Artificial Life*, C. Adami, R. K. Belew, H. Kitano, and C. E. Taylor, eds., pp.114-122, Los Angeles, California, 1998, MIT Press.

[61] Sayama H. (2000). Self-replicating worms that increase structural complexity through gene transmission, *Proc. Seventh Intl Conf Artificial Life*, Bedau, M, McCaskill, J., Packard, N, Rasmussen, S., eds., 21-30, MIT Press.

[62] Schaffer, J. (1985). Multiple objective optimization with vector evaluated genetic algorithms. *Genetic Algorithms and their Applications: Proceedings of the First International Conference on Genetic Algorithms*, 93100.

[63] J. Signorini. “How a SIMD machine can implement a complex cellular automaton? A case study: von Neumann’s 29-state cellular automaton.” In *Supercomputing ’89: Proceedings of the ACM/IEEE Conference*, pages 175-186, 1989.

[64] B. K. Sikdar, N. Ganguly, A. Karmakar, S. Chowdhury, and P. Pal Chaudhuri. Multiple Attractor Cellular Automata for Hierarchical Diagnosis of VLSI Circuits. *In Proc. of Asian Test Symposium*, pages 385-390, November 2001.

[65] Sipper, M. (1998). Fifty years of research on Self-Reproduction: An overview. *Artificial Life*, 4, 237-257.

[66] M. Sipper, D. Mange, and A. Stauffer. “Ontogenetic hardware.” *BioSystems*, Vol. 44, No. 3, pages 193-207, 1997.

[67] Smith, S.F. (1980), *A Learning System Based on Genetic Adaptive Algorithms*, PhD dissertation (University of Pittsburgh)

[68] Tempesti G. A new self-reproducing cellular automaton capable of construction and computation.’ In F. Morn, A. Moreno, J. J. Merelo, and P. Chacn, editors, *ECAL’95: Third European Conference on Artificial Life*, volume 929 of Lecture Notes in Computer Science, 555-563. Springer-Verlag, 1995.

[69] T. Toffoli and N. Margolus. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press, Cambridge, Mass, 1987.

[70] P. Tzionas, P. Tsalides, and A. Thanailakis. A New Cellular Automaton-based Nearest Neighbor Pattern Classifier and its VLSI Implementation. *IEEE Trans. on VLSI Implementation*, 2(3):343-353, 1994.

[71] Vitanyi P. M. B. 1973 “Sexually reproducing cellular automata” *Mathematical Biosciences* 18: 23-54.

[72] P. M. B. Vitanyi. “Genetics of reproducing automata.” In: *Proc. 1974 Conference on Biologically Motivated Automata Theory*, IEEE, New York, N. Y., 1974, pages 166-171.

[73] Ruili Wang and Mingzhe Liu. A Realistic Cellular Automata Model to Simulate Traffic Flow at Urban Roundabouts. *5th International Conference in Computational Science, proceesings, part II.* 420-427. 2005.

[74] Gerard Weisbuch, “*Complex Systems Dynamics*”, Addison Wesley Longman, 1991.

[75] Stephen Wolfram, ed., Theory and Applications of Cellular Automata, *World Scientific: Singapore*, 1986.

[76] S. Wolfram, *Cellular Automata and Complexity*, Addison-Wesley, Reading, Mass, 1994.

[77] S. Wolfram. Universality and Complexity in Cellular Automata. *Physica D*, 10:1-35, 1984.

[78] S. Wolfram. Undecidability and Intractability in Theoretical Physics. *Phys. Rev. Lett.*, 54:735-738, 1985.

[79] S. Wolfram. *A New Kind of Science*. Wolfram Media, Inc, 2002.

[80] Ray, T. S. 2001. Overview of Tierra at ATR. In: "Technical Information, No.15, Technologies for Software Evolutionary Systems". ATR-HIP. Kyoto, Japan.

[81] Srinivas, N., Deb, K.: Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation.* 2 (1994) 221.248

[82] Deb, K., Agrawal, S., Pratap, A., et al.: A Fast Elitist Nondominated Sorting Genetic Algorithm for Multiobjective Optimization: NSGA-II. *Evolutionary Computation.* 2 (2002) 182.197

[83] Zitzler, E., Thiele, L.: Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach. *Evolutionary Computation.* 1 (1999) 257.271

[84] Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In: *EUROGEN 2001*, Athens, Greece (2001)

[85] Knowles, J.D., Corne, D.W.: Approximating the Nondominated Front Using the Pareto Archived Evolution Strategy. *Evolutionary Computation.* 2 (2000) 149-172

[86] Knowles, J.D., Corne, D.W.: M-PAES: A Memetic Algorithm for Multiobjective Optimization. In: *Proceedings of the 2000 congress on evolutionary computation.* Piscataway, NJ: IEEE Press (2000) 325.332

[87] Mange D., Goeke M, Madon D, et al. Embryonics, *Towards Evolvable Hardware*, Springer Verlag, 1996, 197-200.

[88] K. Imai, T. Hori, and K. Morita. Self-reproduction in three-dimensional reversible cellular space. *Artificial Life*, 8:155-174, 2002.

[89] Hutton, T.J. (2004). A functional self-reproducing cell in a two-dimensional artificial chemistry. In J. Pollack et al., eds., *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, 444-449. 290 (6) (June), 65-75.

[90] T. J. Hutton. Evolvable self-replicating molecules in an artificial chemistry. *Artificial Life*, 8:341-356, 2002.

[91] Andre Stauffer and Moshe Sipper. Interactive Self-Replicating, Self-Incrementing and Self-Decrementing Loops. *Artificial Life* VIII, Standish, Abbass, Bedau (eds)(MIT Press) 2002. pp 53-56.

[92] Stauffer, A., and Sipper, M. 2002. An interactive self replicator implemented in hardware. *Artificial* Life,8:175-183.

[93] Ewaschuk, R., Turney, P. Self-Replication and Self-Assembly for Manufacturing. *Artificial Life Journal.* Volume 12, Issue 3. Summer 2006. pp. 411-433

[94] Seeman, N.C. (2003). DNA in a material world. *Nature*, 421 (January 23), 427-431.

[95] Seeman, N.C. (2004). Nanotechnology and the double helix. *Scientific American*, 2004 Jun;290(6):64-9, 72-5

[96] Arbesman, S. (2004). Erg: *A Computational Energetics as an Approach to the Study of the Origins of Life.* Senior Thesis, Computer Science Department, Brandeis University.

[97] NASA, (1980) in Chapter 5: Replicating Systems Concepts: Self-Replicating Lunar Factory and Demonstration, pp. 198-335. http://www.islandone.org/MMSG/aasm/

[98] Tempesti, Gianluca ; Mange, Daniel ; Mudry, Pierre-Andr ; Rossier, Jol ; Stauffer, Andr. Self-replication for reliability: bio-inspired hardware and the embryonics project. *Proceedings of the 3rd conference on Computing frontiers* (2006), p. 199-206, 2006

[99] D. Mange, M. Sipper, A. Stauffer, G. Tempesti. "Towards Robust Integrated Circuits: The Embryonics Approach". *Proceedings of the IEEE*, 88(4), April 2000, pp. 516–541.

[100] Lehmann E.L (1975). Nonparametrics: Statistical Methods Based on Ranks. Holden-Day, San Francisco.

[101] Pan Z. and Reggia J. Evolutionary Discovery of Arbitrary Self-Replicating Structures. *Lecture Notes in Computer Science*, Vol. 3515, V. Sunderam et al(eds.), 2005, 404-411.

[102] Pan Z. and Reggia J. Artificial Evolution of Arbitrary Self-Replicating Structures. *Int. J. of Unconventional Computing*, Vol. X, 1-19. 2005.

[103] Pan Z. and Reggia J. Properties of Self-Replicating Cellular Automata Systems Discovered Using Genetic Programming. *Advances in Complex Systems*, Vol. 10, No. supp01(August 2007).

[104] Gina M. B. Oliveira, Jos C. Bortot and Pedro P. B. de Oliveira. Multiobjective evolutionary search for one-dimensional cellular automata in the density classification task. *Artificial Life VIII*, Standish, Abbass, Bedau (eds)(MIT Press) 2002, pp202-206

[105] Coello, C. 1999. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information Systems* 1(3):269-308.

[106] Oliveira, G.; de Oliveira, P.; and Omar, N. 2000. evolving solutions of the density classification task in 1D cellular automata, guided by parameters that estimate their dynamic behavior. In Bedau, M. A.; Mc-Caskill, J. S.; Packard, N. H.; and Rasmussen, S., eds., *Alife VII*, 428-436. Cambridge, Mass.: MIT Press.

[107] Hisao Ishibuchi , Kaname Narukawa, Comparison of evolutionary multiobjective optimization with rference solution-based single-objective approach, *Proceedings of the 2005 conference on Genetic and evolutionary computation*, June 25-29, 2005, Washington DC, USA

[108] C.A.C. Coello and G.B. Lamont (Eds.). Applications of Multiojective Evolutionary Algorithms. *World Scientific*, 2004.

[109] K. Deb and S. Jain. Running performance metrics for evolutionary multiobjective optimization. In *SEAL*, pages 13–20, Singapore, Nov. 2002.

[110] E. Zitzler, L. Thiele, M. Laumanns, C.M. Fonseca, and V.G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Trans. Evolutionary Computation*, 7:117 – 132, 2003.

[111] Bleuler, S. Brack, M. Thiele, L. Zitzler, E. Multiobjective genetic programming: reducing bloat using SPEA2. Proceedings of the 2001 Congress on Evolutionary Computation. Volume: 1, On page(s): 536-543

[112] B. L. Miller and D. E. Goldberg, Genetic algorithms, tournament selection, and the effects of noise, Complex Syst., pp. 193212, June 1995.

[113] Jobson J. D. (1999). Applied Multivariate Data Analysis: Volume 1: Regression and Experimental Design. Springer Verlag, New York.