General Purpose Cellular Automata Programming

Wei Huang

TR #02-03
March 7, 2002

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, IA 50011-1040, USA

General Purpose Cellular Automata Programming

by

Wei Huang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2001

Graduate College
Iowa State University

This is to certify that the Master's thesis of

Wei Huang

has met the thesis requirements of Iowa State University

Major Professor

For the Major Program

DEDICATION

To my parents and my wife, for their love!

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I am grateful to the many people who helped me in my research and the writing of this thesis.

I would like to give my special thanks to Dr. Hui-Hsien Chou for his guidance, patience, mentoring, and support throughout the preparation of this thesis. It is he who led me to the cellular automata research field. His insightful advice has been invaluable to my graduate study in Iowa State University. I would also like to thank Dr. Leavens and Dr.Gu for their kindness to serve on my committee and to offer their help at any time.

Jun Zhang reviewed chapters 1 and 2 of this thesis, and made several valuable suggestions. Robert Hansen and Michael Burman also helped me out of some grammar errors. I really appreciate their help.

This work was supported in part by NSF SGER Grant IIS-0001542. I want to thank NSF for providing the funding of this research.

ABSTRACT

As cellular automata are becoming popular in many research areas, the need for an easy-to-use system for cellular automata programming is becoming greater. Traditionally, cellular automata transition functions were manually depicted in a tabular format, which is often time-consuming and error prone. A more promising approach is to design a general-purpose cellular automata programming environment.

In this thesis, a new cellular automata simulation environment, jTrend, is introduced. jTrend was developed on the Java platform for cellular automata exploratory research. With a built-in high-level programming language and an easy-to-use graphical user interface, jTrend has become one of the most powerful cellular automata simulators, and can be used for most one- and two-dimensional cellular automata simulations. The object-oriented design and performance optimization techniques used in jTrend provide high flexibility and fast simulation speed.

jTrend has been used to study some real world problems in cellular automata. Solutions for two important problems, *bubble sort* and *satisfiability* (SAT), have been implemented using jTrend. Their experiment results suggest that it may be advantageous to solve problems using cellular automata, and jTrend provides a foundation to test such ideas.

CHAPTER 1. INTRODUCTION

1.1 Cellular Automata Modeling

1.1.1 What are cellular automata models?

Conceived by John von Neumann [21] and Stanislaw M. Ulam [32] in the 1940's, cellular automata models provide a framework to study the behavior of complex, adaptive systems formally.

In essence, cellular automata are discrete dynamic systems whose behavior is completely specified in terms of a local-only relation. Each cellular automaton consists of a grid of cells that can be in one of *N* finite states at any time. Each cell determines its next state based on the states of its neighbor cells (including itself) using a set of pre-defined rules (called transition functions). The states of all cells are updated simultaneously and independently of one another in discrete time steps. One cellular automata example, the *Game of Life*, is shown in Figure 1.1 to depict the mechanism of cellular automata in action.

The Game of Life model was invented by John Conway in 1970 [14] to investigate the basic process in living systems. Each cell in this model can be in either of two states: *alive* or *dead*. The state of each cell changes from one generation to the next, governed by he following rules:

1. If a live cell has two or three live neighbors, it survives.
2. If a live cell has less than two or more than three live neighbors, it dies.
3. If a dead cell has exactly three live neighbors, it is born.

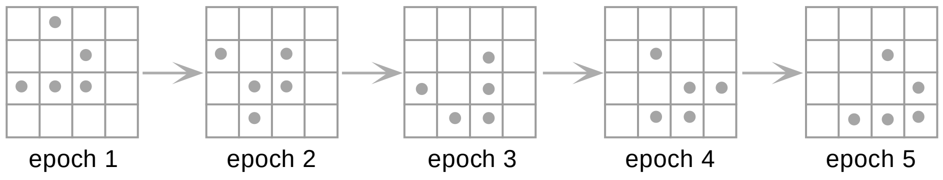


Figure 1.1 Conway's Game of Life example

By the repetitive application of these rules, any given initial pattern of a collection of living cells in a universe of dead cells can develop over time and produces various configuration changes. Figure 1.1 shows how one initial pattern, named *glider*, evolves over five generations. Comparing the patterns at epoch 1 and epoch 5, we note that the glider gradually moves to the lower right corner.

1.1.2 Applications of cellular automata

Since the original work of John von Neumann and Stanislaw M. Ulam, cellular automata have been applied to the study of many real-world phenomena, including adaptation, optimization, simulation, competition, and evolution, etc.

One of the most attractive topics in cellular automata research is self-replication, a challenging subject that was first studied by John von Neumann. It was this study that motivated him to design the cellular automata model. Instead of focusing on the physical realization of self-replication, his study tried to capture the fundamental information processing mechanism underneath self-replications. Such study aims at building systems that are much more autonomous in the future than we can possibly build today. The first self-replication model was designed by John von Neumann in a two-dimensional cellular automaton with 29-state cells. After his pioneering work, self-replication research conducted by other researchers has led to smaller and simpler systems that can be readily implemented on a computer ([6], [19], [2], [23]). Instead of designing the self-replication structure by hand, Chou et al. took a different approach [3]. They presented a model in which self-replication structures can emerge from a set of individual structures that are distributed in the cellular space randomly. To some extend, the results of this study suggest that it is possible to use self-replication structures to study the origin of life. Recently, the research of how to program self-replication structure to do other jobs instead of just self-replicating itself has become popular. This research has led to self-replication structures that can be used to solve other real-world problems, such as SAT [4].

Because of its unique features, cellular automata are also proved to be useful discrete models of dynamical systems theory. A systematic use of cellular automata in this area was pioneered by Stephen Wolfram [30]. The dynamical system theory studies the collective phenomena such as ordering, chaos, fractals, etc., and cellular automata provide a rich collection of representative models where these phenomena can be studied with relative ease.

In addition to modeling phenomenal aspects of our world, cellular automata also provide a way to model the laws of physics directly. Conducted by Edward Fredkin and Tommaso Toffoli first, this research focuses on the formulation of *information-preserving* computational models. In particular, some differential equations of physics, such as the heat, wave and Navier-Stokes equations, have been well studied using cellular automata as simulation models. The detailed introduction about the applications of cellular automata in this field can be found in [27].

To sum up, cellular automata have an increasingly important role as conceptual and practical models of self-replication and discrete dynamical systems. Research in this field will give us better understandings of dynamic phenomena in complex systems with discrete time, discrete space, and a discrete set of state values.

1.2 Existing Cellular Automata Programming Environments

Since cellular automata have been used in many research areas, the need for an easy-to-use system for cellular automata programming is greater than ever. Traditionally, cellular automata programming was conducted manually. To produce desired phenomena, an investigator had to convert his experimental transition function for a cellular model into a mapping table. He then needed to create computer programs that do the simulation and produce results based on the table. He might need to repeat his experiments several times using different tables until the result had succeeded or it appeared that his model was not promising. Without the help of a computer, the job of table creation is tedious and almost impossible to be followed by a human for complex models. In addition, some tasks, such as visualization and result analysis, make the job to conduct cellular automata experiments even more difficult and time-consuming if each investigator has to create their own set of experimental software tools.

Since these circumstances arise quite naturally in connection with cellular automata research, a few software packages have been designed and made available. In the following subsections, three of the existing general-purpose cellular automata simulators are introduced briefly. A comprehensive review of existing simulation systems can be found in [31].

1.2.1 CAM-6

CAM-6 [27] is a commercially available cellular automata machine, which is accessible for PC users. This machine was originally developed to fulfill MIT's internal research needs by the MIT Laboratory for Computer Science. CAM-6 consists of two components: simulation hardware and control software. The hardware component is a module that plugs into a slot of IBM-PC and does most of the simulation work at high speed. The control software for CAM-6 is written in Forth, a semi-high level postfix programming language, and runs on the IBM-PC with 256K of memory.

In CAM-6, users are allowed to specify the rules for cells' behavior by using the Forth language. Internally, these rules are converted into a rule table by the software component and stored in the CAM-6 hardware module. During the simulation, results can be visualized with colored dots, or pixels, on a monitor.

CAM-6 is the first general purpose cellular automata simulation system that is widely available. It is also the first simulator to use hardware acceleration and use a high-level programming language to specify the rule set. Many unique ideas of CAM-6 were adopted by other cellular automata simulation environments later.

1.2.2 Cellular

Different from CAM-6, *Cellular* [9] is a cellular automata programming environment without a specific hardware component, therefore it is easier to deploy. It consists of the following components: *Cellang, avcam,* and *cellview*. *Cellang* is a programming language that is associated with a compiler *cellc*; *avcam* is a simulator used for cell evaluation; and *cellview* is a graphic viewer. Compiled *Cellang* programs can be run on *avcam* with a separate input data file for initial cellular space configuration. The results of simulation can either be fed into *cellview* and viewed graphically, or passed through a custom filter for statistical purposes.

Each program written in *Cellang* can be divided into two main parts: a cell description and a set of statements. The cell description determines the dimension of a cellular space, the field(s) each cell contains, and the bit depth in each field. Statements in *Cellang* are used to decide a cell's next state. Three kinds of statements are available in *Cellang*: *if*, *forall* and *assignment*. The *assignment* statement in *Cellang* is different from other high-level languages, since it provides conditional assignment. The *forall* statement is similar to the loop statement in other high-level languages. The variable that used as an index within this statement will iterate through a range of values. The *if* statement is the same as those in other languages.

Two special variables are maintained in the *Cellular* simulator. `time` is a system defined variable indicating the number of iterations that have been executed. It is updated automatically by C*ellular*. This variable allows field value changes that depend on time. `random` is another pre-defined variable used to provide a uniformly distributed random number in each cell.

The predefined variable `cell` is special and refers to the cell that is under consideration. Assignment to `cell` is the only way to alter the value of the current cell. Furthermore, the new value of `cell` will not become available until the beginning of the next cycle, thus reading `cell`'s value after an assignment to it will obtain its old value, not the assigned new value.

In *Cellang*, neighbors can be arbitrarily referenced using a relative indexing format. In this format, the relative position of the desired neighbor is placed within square brackets ([]). For example, `[1,0]` represents the east neighbor of current cell and `[-1,-1]` for the northwest neighbor. To access the field values of these neighbors, users can place a dot (.) and field name after a relative index. For example, expression `[0,1].dir` will return the *dir* field value of the southern cell.

A new feature introduced in the latest version of *Cellang* is *agents*. Agents are designed to manage the complexity involved in moving a value from one cell to another. Compared with traditional cellular automata techniques, agents allow such movements to be specified more clearly and easily. Additional language features of *Cellang* can be found in [10].

1.2.3 CAMEL/CARPET

Traditionally, most cellular automata simulation environments were implemented on sequential computers, such as CAM-6 and *Cellular*. But in essence, cellular automata are intrinsically parallel. Cellular automata models run in a parallel way because an identical set of rules is applied simultaneously to all cells during each iteration. This characteristic makes parallel computer an ideal platform for the implementation of high-performance cellular automata simulators

CAMEL ([12], [13]) is an interactive parallel programming environment that uses the cellular automaton both as a model for parallel computation and as a tool to model and simulate complex dynamic phenomena. The elaborate design of CAMEL helps to hide the underlying architectural issues from users while offering the computing power of a parallel computer.

CAMEL is composed of a set of *macrocell* processes that run on each processing element of the parallel machine, and a *controller* process running on a master processor. Each *macrocell* process simulates several elementary cells of the cellular automaton, and makes use of the underneath communication system to handle the data exchange among them. All *macrocell*s execute the same local rule set in parallel, under the coordination of the *controller* process. With this design, users only need to specify the transition function for a single cell in the cellular space without considering other cells.

Similar to *Cellular*, CAMEL comes with a programming language called CARPET, which is used for programming cellular algorithms in the CAMEL environment. A CARPET program is composed of a declaration part and a body program. The declaration part, which is similar to the cell description component in *Cellang*, specifies the structure of the automata space, such as its dimension, and determines the information stored in each cell. The body program contains usual statements similar to the *C* language, and a set of special functions to access and modify the states of a cell and its neighborhood. CAMEL converts the body program automatically into a binary code, and distributes the binary code to each processor for simulation.

In CARPET, fields of the current cell can be referred by `cell_field`. To guarantee the synchronous cell updating in cellular automata, the value of fields can only be modified by the `update` function.

Compared with *Cellang,* CARPET generalizes the concept of neighborhood. In CARPET, users are allowed to define a *logical neighborhood* by a `neighbor` declaration. The neighbors of current cell can be assigned specific names through the `neighbor` declaration. Cells can refer to field values of its neighbors using these names. For example, the von Neumann neighborhood can be defined as follows [13]:

```
Neighbor Neumann[4] ([0, -1]North, [1, 0]East,
                     [0, 1]South,  [-1, 0]West);
```

With this declaration, the *dir* field of the northern cell could be referred as `North.dir` instead of `[0,-1].dir` as is required in *Cellular*

Similar to the variable `time` in *Cellang*, CARPET allows users to access the number of iterations that have been executed through a predefined variable `step`. A random number function, `random`, is also provided in CARPET. In addition, CARPET allows cells to access the value of the coordinates X, Y and Z of a cell in the automaton through the `Getx`, `Gety`, and `Getz` functions. These functions are useful for users to design heterogeneous models, in which different cells have different transition functions.

1.3 The Trend Simulation Environment and its Language Features

1.3.1 Why is Trend needed?

Although existing simulation systems are very capable in doing many general-purpose cellular automata programming, they are still inadequate for our requirements. In particular, we need a system that allows us to specify the information stored in each cell easily, a backtracking mechanism to trace back to previous cellular automata space configurations for debugging code, and a high level cellular automata programming language which can exploit the rotational symmetry of the cellular automata space. For these reasons, a new cellular automata simulation environment, Trend, was developed.

Trend combines most capabilities of available cellular automata simulation systems and introduces new language features. Compared with others, Trend has some major advantages for doing simulations of complex cellular automata models without increasing the complexity of programming too much.

The Trend programming language takes a high-level structured approach and is modeled after the most popular programming language *C*. Some language features of Trend will be discussed in the next section. Detailed description of the Trend language can be found in [3].

Since the Trend language contains most modern programming language constructs, it allows algorithms expressed using it to be very complex, yet still readable for users. This greatly extends the power of cellular automata programming when compared with manually created cellular automata simulation systems. A Trend language compiler is bundled with the Trend simulator. For each Trend program, the Trend compiler generates a virtual machine code that can be executed by its simulation engine. The working process of the Trend simulation environment is shown in Figure 1.2.

1.3.2 Trend language features

Data types are the basic constructs of a programming language. Trend[1] introduces three primitive data types: `int`, `nbr`, and `fld`. The first one, integer data type, is common in other programming languages. An `int` variable is a 32-bit integer and that can be used to store cell states temporarily. `nbr` and `fld` are two special data types introduced in Trend specific to cellular automata programming. `nbr` is used to denote the position of a neighbor cell, such as north, south, etc. `fld` denotes fields within each cellular automata cell. When combined with the `nbr` data type, `fld` can uniquely specify a particular field within a particular neighbor cell.

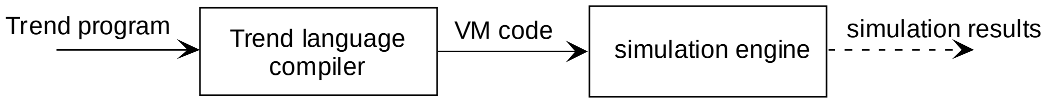


Figure 1.2 The working mechanism of the Trend simulation environment

[1] In this thesis, Trend is used to refer to either the simulation environment or the programming language. Readers should be able to differentiate its uses from the context.

Most arithmetic and logical operators in *C* are available in Trend, such as `+`, `*`, `++` (increase), `%` (remainder), `&` (logical and), `<=` (equal or less than), etc. Function declarations are also allowed in Trend.

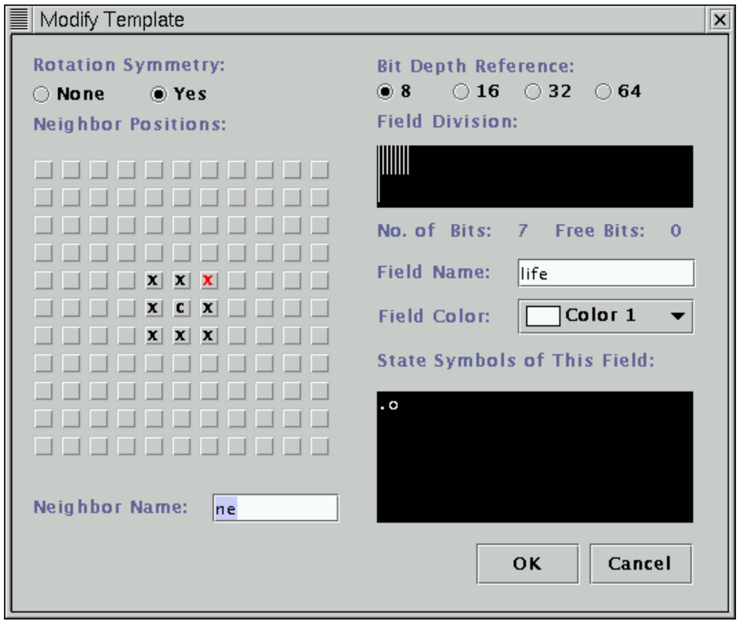
Unlike *Cellang*, Trend programs do not contain any cell description. A cell description, which specifies the information of neighborhood and cell fields, can be defined "on the fly" by the user using the template window (shown in Figure 1.3) of the Trend simulator. Field and neighborhood names defined in the template window are used as *semi-reserved* words in a Trend program. Users can use these names to refer to a particular neighbor and field of the current cell. The Game of Life rules rewritten in Trend [3], shown below, demonstrate some language features of Trend:

```
default life = life;

int count;
nbr y;

count = 0;

over each other y:
     if(y:life) count++;
if(count < 2|| count > 3)
     life = 0;
if(count == 3)
     life = 1;
```

In Game of Life, we defined two variables, `count` and `y`. `count` is an `int` variable used as a temporary storage for the counting of alive neighbors. As an `nbr` variable, the second variable `y` stores a neighbor position index.

Since Trend allows users to configure neighbor and cell information in the template window, that makes the neighborhood reference quite simple in Trend programs. For example, in the Game of Life, a cell can refer to the *life* field of its neighbors using `y:life,` without any additional specification.

One special feature of Trend is in the control flow statements. In addition to providing traditional control flow statements such as `if` and `while`, Trend introduces the `over` statement. The `over` statement in Trend is used to scan over all neighbors for a current


Modify Template

Rotation Symmetry: None [✓ Yes]

Neighbor Positions:

Neighbor Name: ne

Bit Depth Reference: [✓ 8] 16 32 64

Field Division:

No. of Bits: 7 Free Bits: 0

Field Name: life

Field Color: Color 1

State Symbols of This Field:

```
.o
```

Figure 1.3 The template design window of jTrend

cell. It takes the following form [3]:

```
over each {other} nbr_variable statement
```

In essence, the `over` statement is similar to the loop statements in other languages. It loops through all neighbors of a cell, including the center cell itself. During every loop, the `over` statement assigns each neighbor position index into the `nbr_variable`, which can be used in the following statement. The tag "`other`" is optional. It can be added to exclude the center cell in the scanning process. For example, the following code segment in Game of Life determines how many neighbors are alive:

```
count = 0;
over each other y:
    if (y:life) count++;
```

For visualization purposes, each field value in Trend is mapped to an ASCII character. For example, if character '`O`' is defined as the second symbol for the *life* field in the template window, then literal '`O`' represents the value 1 on screen during simulation. Similarly, when '`O`' is assigned to the *life* field in a Trend program, the value of *life* field is set to 1. Symbolic representations make simulation results more readable and real-world problem solving easier.

1.4 Our Projects for Trend Improvement and Portability

For any general-purpose simulation environment, its availability is a critical factor. The existing cellular automata simulators have different kinds of limitations that prevent them from becoming convenient programming environments for users. For example, the CAM-6 requires a special hardware component. Without purchasing this component, it is impossible for users to use CAM-6. Even though Trend does not require such specific hardware, there are still other limitations of Trend. In particular, Trend was designed for specific platform ─ Unix. For users who do not have the access to Unix machines, they cannot use Trend. Additionally, the user interfaces of Trend on Unix were developed using Motif, a commercial graphical user interface (GUI) library. This dependence prevents the Trend source code from generating executable programs on machines where Motif has not been licensed. This limits Trend's widespread availability as well.

Even though the Trend language is more capable than many existing cellular automata programming languages, it still has some limitations. For example, variables in Trend have to be declared before their usage; there is no bracket around a function body, etc. These limitations make us believe that it is time to develop a new version of the Trend simulator, not only for portability reasons, but also for improved language features.

Nowadays, Java is becoming a popular cross-platform programming language. Any Java program is guaranteed to run on different platforms if users install the Java Runtime Environment (JRE), which can be freely downloaded from Sun Microsystems Inc. In addition, Java provides a complete graphical library for users. The abundant graphical components provided in Java make porting Trend's interfaces to Java comparably easy. These benefits make Java our top candidate for extending Trend cellular automata programming language and its simulation environment to the other platforms other than Unix.

We name the new system jTrend, meaning that it is a Java-based Trend. For the rest of this thesis, Trend refers to the Unix version and jTrend denotes the Java version.

1.5 Organization

The rest of this thesis is organized as follows. In Chapter 2, the software organization of jTrend is introduced, with some discussion of implementation issues. New features that are not available in Trend before but added in jTrend are mentioned in this chapter as well. By using the new jTrend environment and the improved Trend programming language, solutions for two computational problems, bubble sort and satisfiability (SAT) problems, are presented in Chapter 3 with some experiment results. Finally, in Chapter 4, we summarize the achievements and make some conclusions.

CHAPTER 2. jTrend ─ A JAVA-BASED CELLULAR AUTOMATA SIMULATOR

jTrend is a general-purpose cellular automata simulator that was created to support the development and experiment of cellular automata research. With its high-level, general-purpose programming language and the flexibility in the neighborhood definition and data field allocation, jTrend is suitable for most one- and two-dimensional cellular automata simulations.

For any well-designed cellular automata simulator, the following three issues must be addressed carefully. The first issue is *software organization*. For a complex simulation system, the software design determines the maintainability and flexibility of the whole system. The second one is *graphical user interfaces* (GUIs). A user-friendly cellular automata simulator should provide integrated, easy-to-use interfaces that allow on-going monitoring and interactive control of simulations. The last issue is s*imulation performance*. Performance is a critical issue for simulation systems, especially for jTrend that was implemented on the Java platform.

This chapter covers the above issues in jTrend. The software organization of jTrend is introduced first in Section 2.1, including its major software modules and their relationship. The rest of Section 2.1 discusses the design of these modules in detail. Section 2.2 is devoted to introducing major GUI components of jTrend. To make that section more complete, some screen shots of these GUI components are presented as well. Section 2.3 discusses the strategies that are used to optimize the simulation performance of jTrend. Finally, the availability of jTrend is discussed in Section 2.4.

2.1 Software Organization of jTrend

Basically, jTrend can be divided into two major modules: Trend language compiler and GUI components. The relationship between them is shown schematically in Figure 2.1.

Trend compiler is a stand-alone module used to compile Trend source code (i.e. rule sets). Upon receiving a “compiling” command from users, a GUI component invokes the compiler module to compile the Trend source code. If the compiling is successful, the

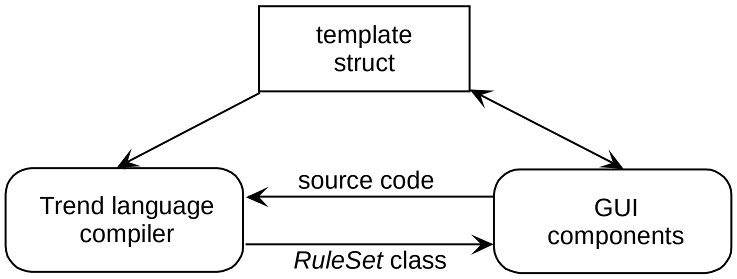


Figure 2.1 Interactions between Trend language compiler and GUI components

source code is converted into an executable Java class, *RuleSet*, and returned to the GUI components as a transition function for cell evaluation. GUI uses the *RuleSet* class to calculate the next state value for each cell in the cellular automata space. If the compiling fails, the compiler will report errors and indicate the positions where errors happen in the source code. According to Figure 2.1, a template struct is referenced by both the GUI module and the Trend language compiler. jTrend stores the information that is related to cell configuration, such as the definitions of cell neighborhood and data bits allocation, in this struct.

As we can see in Figure 2.1, the interfaces between the GUI module and the compiler module are very simple. That allows these two modules to be developed independently as long as the interfaces between them are maintained well. In the next two subsections, the designs of Trend compiler and the GUI components are presented in detail.

2.1.1 Trend language compiler

Because the simulation performance is primarily decided by the efficiency of the binary code generated by the compiler, the compiling module has become a crucial part of jTrend. Originally, the output of Trend compiler on Unix is a parser tree. To decide the next state for each cell, the evaluation component has to follow the parse tree and calculate values for some nodes using recursive function calls. But in Java, the operations involved in function calls are time-consuming. Thus, the interpreted parse tree approach is unacceptable for jTrend, which runs on an interpreted environment already. To improve the evaluation performance, we decided to compile the source code directly into a JVM class. A JVM class can be loaded and executed by the Java virtual machine directly.

Obviously, this solution is far better than the previous one and will improve the simulation performance significantly.

Basically, the jTrend compiler[2] consists of two parts: a *lexical analyzer* and an *iterative parser*. As shown in Figure 2.2 ([1]), the lexical analyzer takes a stream of characters and produces a stream of tokens that become the input to the following phase. When the parser obtains a string of tokens from the lexical analyzer, it will verify that the string can be generated by the grammar for the Trend language. The parser will report any syntax error in an intelligible fashion. If there is no error during compiling, the parser will use the Oolong [11] JVM toolkit to generate a JVM class object, *RuleSet*, and return it to the GUI module. With the current cell's neighborhood configuration as the input, the GUI module executes the *evaluate* method of the *RuleSet* object for each cell. The return value from the *evaluate* method will become the next state value for the current cell.

To parse context dependent literals, the lexical analyzer of jTrend compiler was written manually. On the other hand, the *Java_CUP* [18], a Java parser constructor, was used to generate the parser automatically based on a Yacc-like specification.

2.1.2 The GUI module

The organization of jTrend GUI components is shown in Figure 2.3, with a dashed rectangle around it. Note that this figure is similar to Figure 2.1, except that it provides more detailed information about the interactions between different modules. The information flows between different components are shown in this figure using arrow lines.

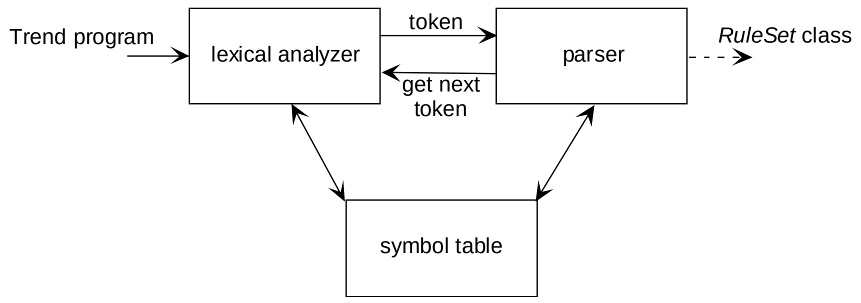


Figure 2.2 The lexical analyzer and parse of Trend language compiler

[2] The jTrend compiler is solely developed by Dr.Hui-Hsien Chou.

As we can observe in Figure 2.3, the GUI components of jTrend are divided into three categories: *user interaction components, graphical visualization components* and *evaluation components*. The user interaction components provide interactive interfaces between users and jTrend. In jTrend, users can change the template of a cellular automata space, specify the values of cells and modify the rule set, etc. Additionally, jTrend allows users to configure the appearance of the simulation environment and monitor the simulation process during run time. All of these functions are provided through the user interaction components of jTrend. The graphical visualization components are responsible for presenting the simulation results to users. The evaluation components in the GUI module calculate the next state for each cell based on current states of its neighbors (including itself), using the *RuleSet* class that is returned from the compiler module. The visualization components then convert the new cell values into graphical symbols that are defined by users in the template window. Finally, these symbols are drawn on the cellular automata space.

According to Figure 2.3, the cell evaluation and the graphical visualization are separated from each other. In addition to maintaining clear interfaces between different components, there are other reasons to take this approach in the implementation of jTrend.

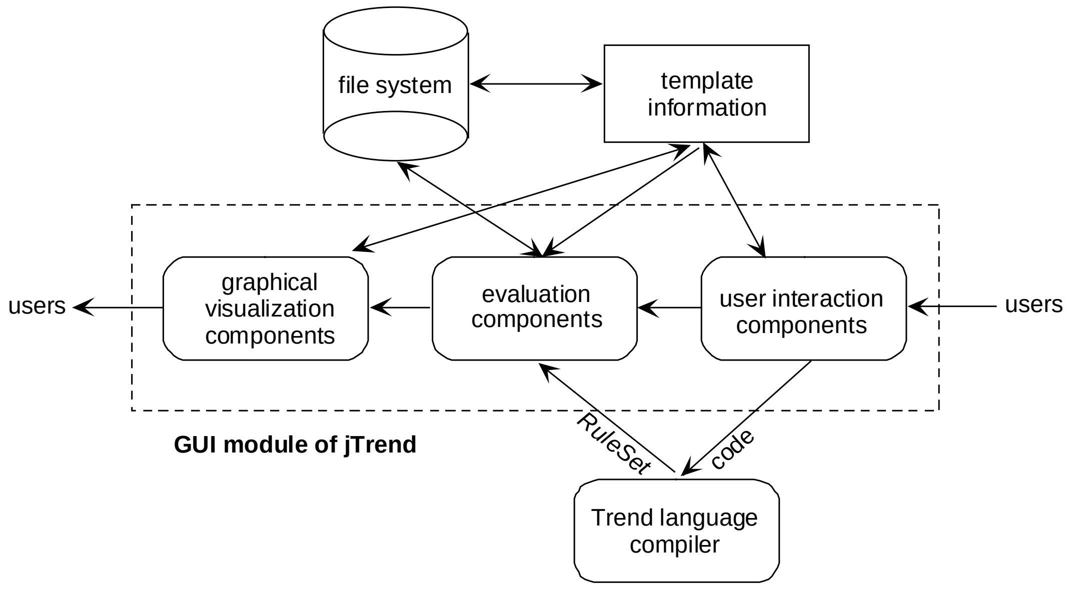


Figure 2.3 Software organization of jTrend GUI module

The first reason is for system optimizations. For a cellular automata simulator, the graphical visualization is always a bottleneck of the entire system. Separating the visualization from cell evaluation allows us to optimize the graphical updating without worrying about altering other parts. The optimization of graphical visualization in jTrend will be introduced in Section 2.3.

Another reason for separating these components is to provide more flexibility in jTrend. For example, there is a new feature in jTrend that allows users to specify a refresh rate for the graphical updating. This rate forces the graphical module to update the simulation results on the main window at a particular frequency. If the evaluation and graphical visualization components are combined together, there is no way to do the evaluation and graphical updating at different rates.

It should also be pointed out that, all information that is required in simulation, including template, cellular space configuration and rule set, can be saved to the file system for later use. For users that are working on complex cellular automata models, this feature allows them to design their models incrementally.

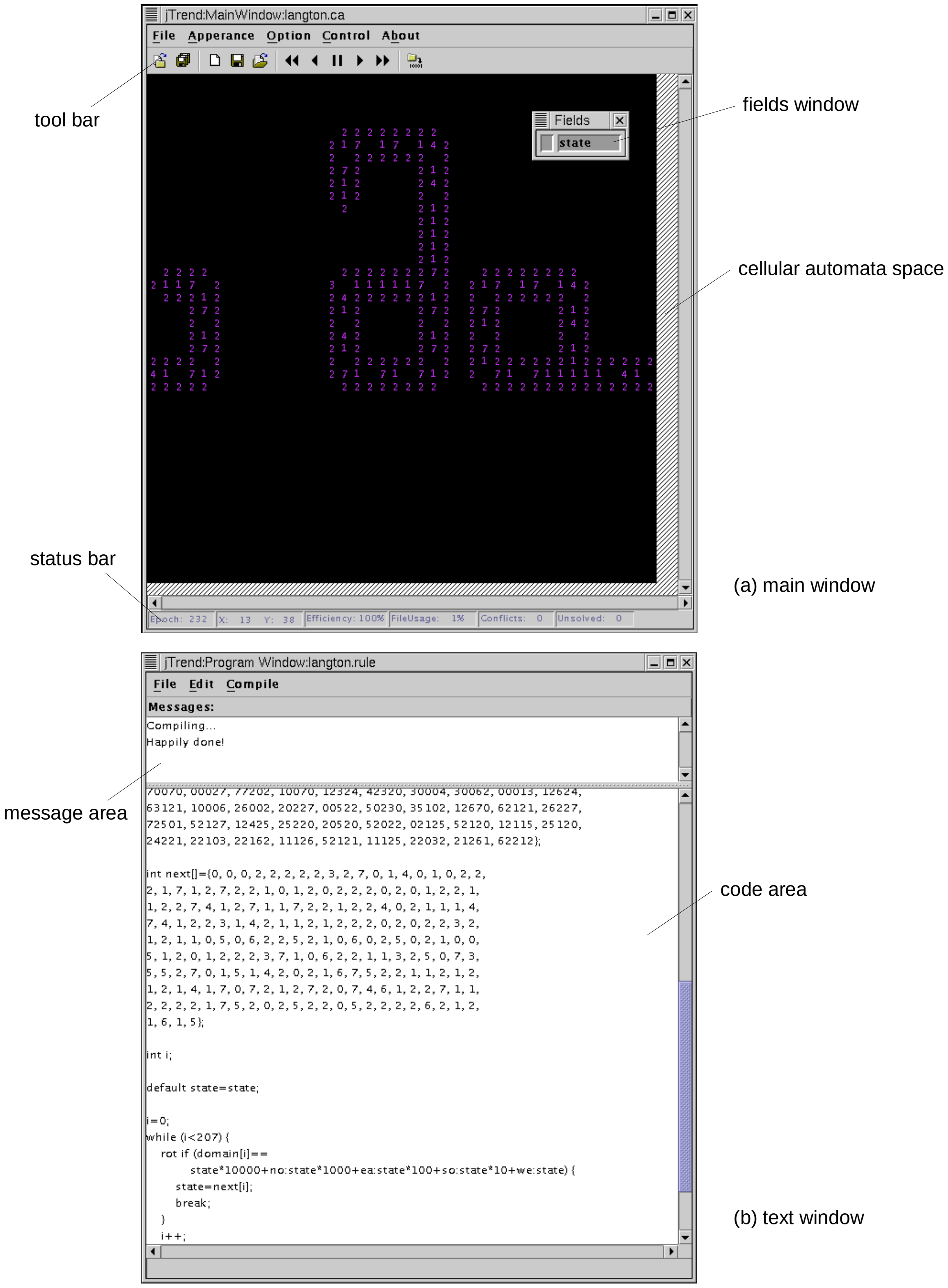
2.2 Major GUI Components and their Functions

In this section, major GUI components of jTrend, including their functions, are introduced in detail. This section focuses on the new graphical features that are special and useful to users who are using jTrend. To limit the size of this thesis, other components of jTrend will not be covered here. Interested readers can find the usage information of jTrend in its software release document.

2.2.1 Main window and text window

As an integrated cellular automata simulator, jTrend allows users to configure their working area and edit their source code at the same time. jTrend presents users two windows, the *main window* and the *text window* (Figure 2.4).

Like most text editors, the text window of the simulator allows users to load a previously designed Trend program, or input new rules into the editing area. As we



```
Epoch: 232   X: 13   Y: 38   Efficiency: 100%   FileUsage: 1%   Conflicts: 0   Unsolved: 0
```

```
Compiling...
Happily done!
```

```
70070, 00027, 77202, 10070, 12324, 42320, 30004, 30062, 00013, 12624,
63121, 10006, 26002, 20227, 00522, 50230, 35102, 12670, 62121, 26227,
72501, 52127, 12425, 25220, 20520, 52022, 02125, 52120, 12115, 25120,
24221, 22103, 22162, 11126, 52121, 11125, 22032, 21261, 62212};

int next[]={0, 0, 0, 2, 2, 2, 2, 2, 3, 2, 7, 0, 1, 4, 0, 1, 0, 2, 2,
2, 1, 7, 1, 2, 7, 2, 2, 1, 0, 1, 2, 0, 2, 2, 2, 0, 2, 0, 1, 2, 2, 1,
1, 2, 2, 7, 4, 1, 2, 7, 1, 1, 7, 2, 2, 1, 2, 2, 4, 0, 2, 1, 1, 1, 4,
7, 4, 1, 2, 2, 3, 1, 4, 2, 1, 1, 2, 1, 2, 2, 2, 0, 2, 0, 2, 2, 3, 2,
1, 2, 1, 1, 0, 5, 0, 6, 2, 2, 5, 2, 1, 0, 6, 0, 2, 5, 0, 2, 1, 0, 0,
5, 1, 2, 0, 1, 2, 2, 2, 3, 7, 1, 0, 6, 2, 2, 1, 1, 3, 2, 5, 0, 7, 3,
5, 5, 2, 7, 0, 1, 5, 1, 4, 2, 0, 2, 1, 6, 7, 5, 2, 2, 1, 1, 2, 1, 2,
1, 2, 1, 4, 1, 7, 0, 7, 2, 1, 2, 7, 2, 0, 7, 4, 6, 1, 2, 2, 7, 1, 1,
2, 2, 2, 2, 1, 7, 5, 2, 0, 2, 5, 2, 2, 0, 5, 2, 2, 2, 2, 6, 2, 1, 2,
1, 6, 1, 5};

int i;

default state=state;

i=0;
while (i<207) {
   rot if (domain[i]==
         state*10000+no:state*1000+ea:state*100+so:state*10+we:state) {
      state=next[i];
      break;
   }
   i++;
```

Figure 2.4   The Main Window and Text Window of jTrend

mentioned earlier, the program written by users is passed to Trend compiler when users decide to compile their source code. The compiling message, no matter successful or not, is shown in the message area of the text window.

The main window in jTrend allows users to decide the configuration of their simulation environment or to load a previously designed configuration from a file. In the cellular automata space displayed in the main window, users can also modify values of different cell fields. Additionally, the main window of jTrend presents two widgets, toolbar and status bar, to users. The toolbar on the top of main window allows users to manipulate frequently used functions. The status bar is continually updated to help users monitor the progress of on-going simulations. These two widgets help jTrend to be more convenient to use. The detailed functions of toolbar and status bar are described in Section 2.2.4.

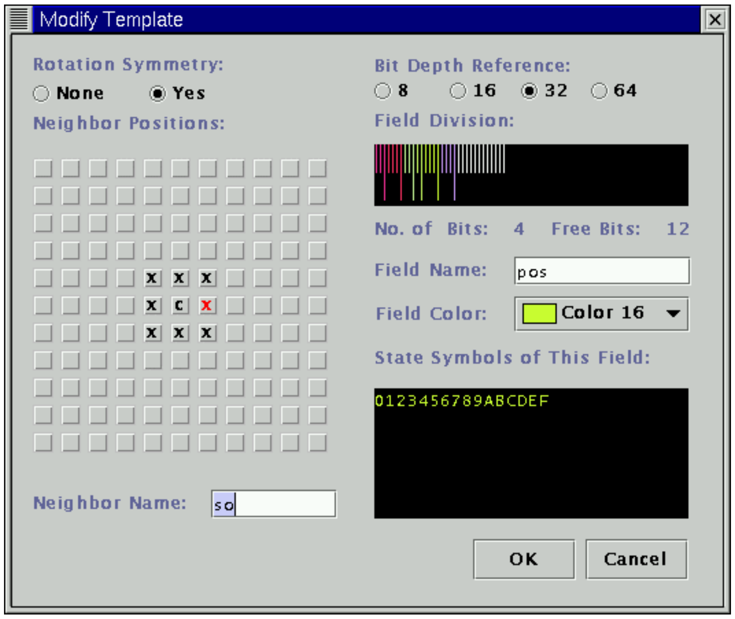
2.2.2 Template design window

Since jTrend is very flexible, the neighborhood layout and cell data bit allocation can both be determined in the *template design window*, instead of being specified at the beginning of the source code like in Cellang, CAMEL.

The template design window is shown in Figure 2.5. Basically, this dialog can be divided into two parts. The left portion of the window is a set of choices related to the neighbor definition of current cellular automata model. The right portion is used for field information definitions in a cell.

An 11 x 11 neighborhood matrix on the left part of the template dialog is used for neighborhood position definitions. The cell with a ‘C’ mark is the central cell. According to the definition of cellular automata, central cell is always required to be included in the neighborhood template. The center cell always represents the current cell when an evaluation is being executed. Users can define other neighbor cells by clicking on other cells and specify their names. As described in the previous chapter, all neighbor names will become semi-reserved words in Trend program. Users can use these names to access cell’s neighbors in a Trend source code.

In the right part of the template dialog, a set of choices regarding the field definition such as field bits, field names and field colors etc are available to users for configuration.


Modify Template

Rotation Symmetry: None [✓ Yes]

Neighbor Positions:

Neighbor Name: so

Bit Depth Reference: 8 16 [✓ 32] 64

Field Division:

No. of Bits: 4 Free Bits: 12

Field Name: pos

Field Color: 🟩 Color 16

State Symbols of This Field:

0123456789ABCDEF

Figure 2.5 The template design window of jTrend (duplicate of Figure 1.3)

In addition to showing the bit allocation for each field, jTrend also show how many free bits available for use within the designated bit depth. That will help user to decide the tradeoff between the field count and their data bit lengths. The maximum cell bit depth is 64, which is two integers long in a popular 32-bit computer system.

In addition to bit information, users can also configure the field symbols for each field in the template window, if its bit length is less than 8. Different field values can be visualized by the field symbols specified here. The reason to use symbols in cellular automata is to make simulation results more readable and closer to real world when jTrend is being used to solve real-world problems.

2.2.3 Floating field window

As we can see in Figure 2.5, the data bits of cells are always divided into different fields with different colors in jTrend. Unfortunately, the symbols of these fields in one cell will overlap with each other on the working area in the main window. That makes it hard for users to view each individual symbol clearly. To manage this problem, we need to allow users to decide which fields to be shown. In jTrend, there is a floating *field window* (Figure 2.6), which is associated with the main window. This floating window is

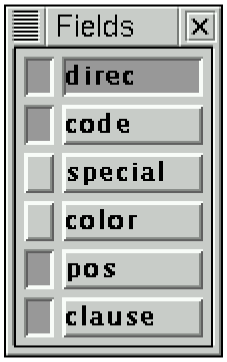


Figure 2.6 The floating field window of jTrend

designed to control the display and editing of cellular automata fields in the working area.

The right column of this field window lists the field names defined by users. Any field that is selected in this column will become the current *focus field.* The main function of focus field is to decide which field is associated with the field editing popup menu. This popup menu is used to edit the field values in the working area of main window. To allow editing, the focus field content will always be displayed in the cellular space of the main window. At any time, there is only one focus field. A new focus field selection will replace the previous focus field.

To the left of the name list is a set of check boxes. Users can decide which data fields are necessary to be displayed at any time by clicking these check boxes. In Figure 2.6, four different data fields (*direc*, *code*, *pos* and *clause*) are selected by the user in the left column. Combined with the focus field, these four fields will be displayed in the cellular space of the main window.

2.2.4 Toolbar and status bar

The toolbar (shown in Figure 2.7) in the main window provides users fast access to commands that are used most frequently, such as the template loading, the program compiling and the simulation controls, etc. Users can benefit from using this facility because this facility will help to save users' time when modeling complex systems.

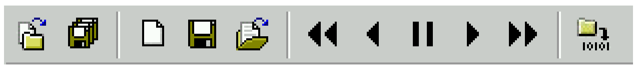


Figure 2.7 The toolbar in the main window of jTrend

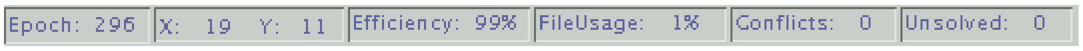

Epoch: 296 | X: 19 Y: 11 | Efficiency: 99% | FileUsage: 1% | Conflicts: 0 | Unsolved: 0

Figure 2.8 The status bar in the main window of jTrend

As shown in Figure 2.8, the status bar of jTrend provides users with the internal information, such as simulation iteration number (i.e. epoch), trace file usage, and unsolved cell number etc. The meanings of these flags are outlined in Table 2.1

2.2.5 Controls of cell evaluation

As a user-friendly simulation environment, jTrend provides two kinds of evaluation mechanisms. The first one is *step evaluation*. The "Forward one step" command provided allows users to trace every simulation step. Thus, users can spot an error, figure out the error position in the rule set and correct it instantly. However, this command is not an appropriate choice for long-term simulations. Because it requires constant clicking to move forward, jTrend provides a "Fast forward" command, which starts the simulation continuously until users click the "Pause" button. All available control buttons in the tool bar are shown in shown in Figure 2.9.

Another mechanism is *backward tracing*, which provides a backward capability in jTrend. When users are designing their rule sets and notice that the behavior of their simulation is not in the way as they expect, they can backtrack one step to find out what really happened during the simulation. The backward tracing mechanism allows this to happen. With this mechanism, debugging code in jTrend becomes much easier.

Table 2.1 The function description of flags in status bar

Flags	**Flag Description**
Epoch	Current epoch number
X, Y	Current coordinates of mouse pointer in cellular automata space
Efficiency	The efficiency of caching mechanism in cell evaluation
FileUsage	The percentage of space size used as trace file
Conflicts	The number of cells with conflict errors
Unsolved	The number of cells with unsolved errors

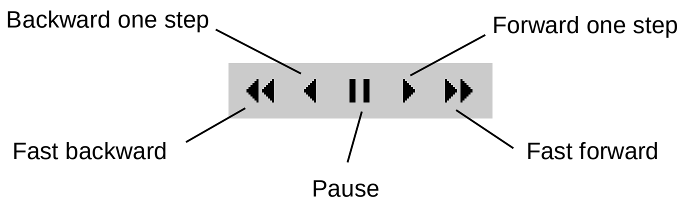


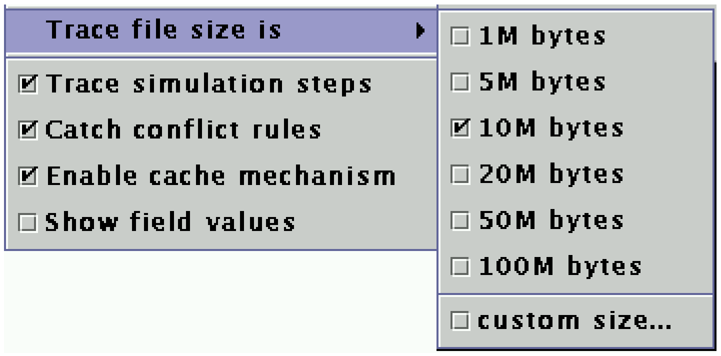
Figure 2.9 Control buttons available in jTrend toolbar

The difference between iterations is stored in a temporary trace file on disk by jTrend. When user want to trace back, the backtracking mechanism uses this file for tracing back to previous epochs. For long-term simulations, the trace file saved on disk may become very large and corrupt the file system eventually. To prevent such circumstances from happening, jTrend only records up to 3000 simulation steps and allows users to specify a limit for the trace file size as (see Figure 2.10).

2.2.6 Other special visualization features

To help users debug their program, another new feature, the *tooltip cell information*, is added in jTrend. In the main window of jTrend, users can place the mouse cursor over any cell in the working area to display its field values. The values are displayed in a small pop-up window that looks like a tool tip. One example is shown in Figure 2.11.

Since users may have different appearance preferences, it is a benefit for users to be able to configure the appearance of their simulation environment as they wish. In jTrend, most appearance parameters can be configured in the appearance menu (Figure 2.12). Through the appearance menu, users can decide the refresh rate of simulations, the



Trace file size is ▸

- ☑ Trace simulation steps
- ☑ Catch conflict rules
- ☑ Enable cache mechanism
- ☐ Show field values

- ☐ 1M bytes
- ☐ 5M bytes
- ☑ 10M bytes
- ☐ 20M bytes
- ☐ 50M bytes
- ☐ 100M bytes
- ☐ custom size...

Figure 2.10 jTrend trace file size configuration

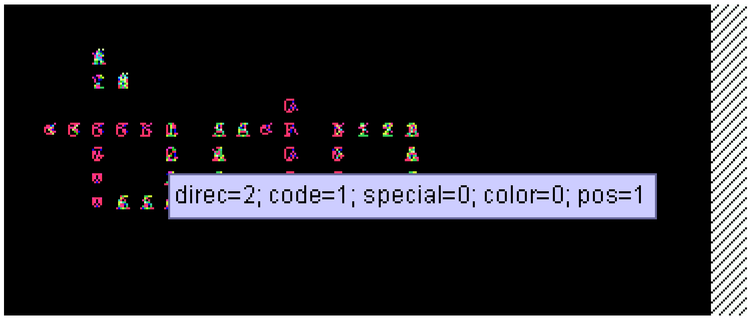


Figure 2.11 One example of the tooltip cell evaluation in jTrend

pixel size of field symbols displayed in the main window, and whether to display quiescent states or not, etc.

2.2.7 Input/Output GUI components

In order to make itself platform-independent and compatible with the existing Trend version, jTrend makes uses of the Java *network file format* to store three different kinds of information, including the template configuration, the cellular automata space configuration, and the Trend language rule set. The files that are saved in the network file format are independent from platforms and could be recognized by jTrend or Trend across different file systems.

The commands that are related to file system in jTrend could be found under the "File" menu item (Figure 2.13) in the main window. Through this menu item, users can load template or rule set files from disk and continue with the simulations. In addition, jTrend

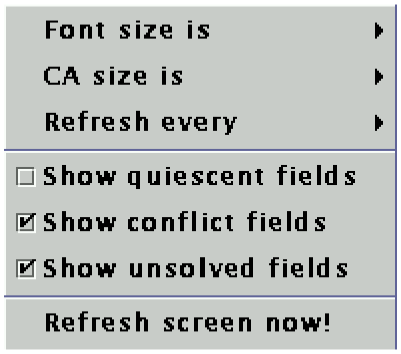


Figure 2.12 The appearance menu of jTrend

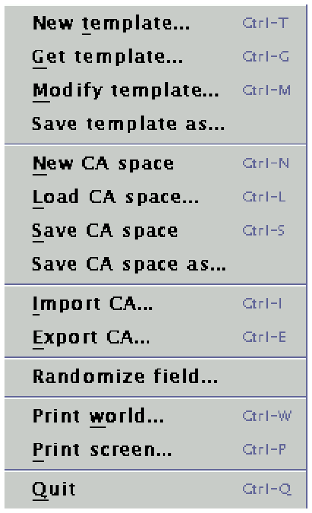
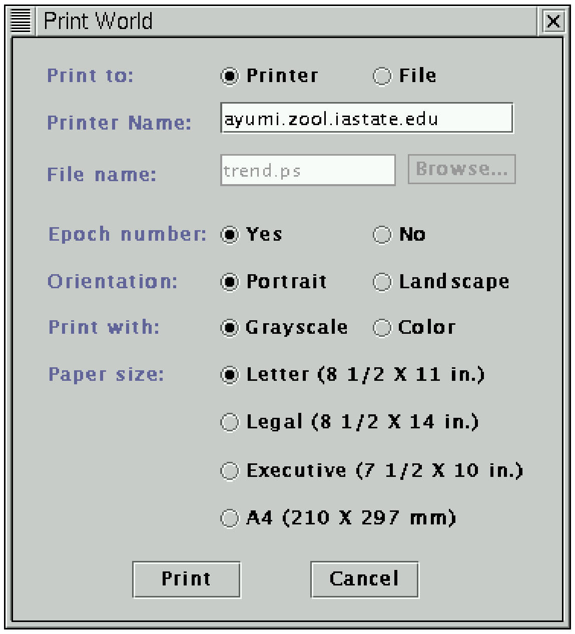


Figure 2.13 The commands under the File menu item in jTrend

allows users to print the configuration of current cellular automata space through two printing commands “Print world” and “Print screen”. The configuration will be converted into a postscript file and sent to a printer for printing. Users can determine other printing options in the printing configuration dialog (shown in Figure 2.14) that is provided by jTrend.


Print World

Print to: [✓ Printer] File

Printer Name: ayumi.zool.iastate.edu

File name: trend.ps Browse...

Epoch number: [✓ Yes] No

Orientation: [✓ Portrait] Landscape

Print with: [✓ Grayscale] Color

Paper size: [✓ Letter (8 1/2 X 11 in.)]
Legal (8 1/2 X 14 in.)
Executive (7 1/2 X 10 in.)
A4 (210 X 297 mm)

Figure 2.14 The printing configuration dialog in jTrend

2.3 Implementation Issues

In the previous section, the functions of major GUI components in jTrend are introduced. Since jTrend is implemented in Java, there are a variety of implementation issues that are not found in the Trend Unix version. Some implementation issues, especially the performance tuning and optimization, are discussed in this section.

2.3.1 Cell evaluation performance tuning

The evaluation module in jTrend calculates the next state for each cell based on the cell's current state and the states of its neighborhood. This computation is conducted using the *RuleSet* class object that is returned from the compiler module (Figure 2.3). The time used in this stage is called *evaluation time.* The evaluation time for each cell might be tiny. But since there are a massive number of cells in a cellular automata space, the evaluation time for these cells turns out to be one of the bottlenecks in jTrend. To improve the simulation speed of jTrend, a three-phase strategy is used to conduct the evaluation for each cell. This strategy is depicted in Figure 2.15.

The first phase is *neighborhood invariant skipping*. According to the definition of cellular automata, the next state of each cell is decided solely by its neighborhood (including itself). If none of its neighbors has been changed, the current cell, which is under evaluation, will stay unchanged. The first phase of cell evaluation module takes advantage of this characteristic. For each cell, we add one flag bit, CHANGE, to indicate

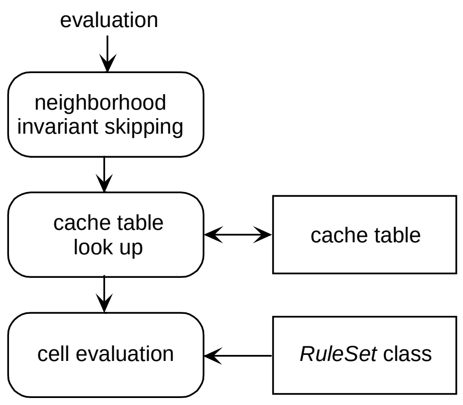


Figure 2.15 The three-phase strategy used in the evaluation module of jTrend

its changing history. If a cell keeps unchanged from the previous epoch to the current epoch, its CHANGE bit is 0. Otherwise this bit is set to 1. Thus, before calculating the next state for a cell, the evaluation module checks the CHANGE bits of all its neighbors. If none of these neighbors has been altered, the cell can be skipped in the evaluation process since its value cannot change in the current epoch.

For a cell that has a different neighborhood in the previous epoch, its next state has to be calculated. Instead of doing cell evaluation using the *RuleSet* class, the evaluation module looks up the cache table and tries to retrieve a state value for current cell. If a recent evaluation that has the same neighborhood as the current cell could be found in the cache table, the evaluation module will use them and the third evaluation phase is unnecessary. The cache table is maintained as a priority tree. Only the least often referred result has the chance to be replaced by new results.

If the second phase does not find a value either, an actual evaluation has to be taken now. The evaluation component will collect the neighbor values of the current cell and pass them to the compiled *RuleSet* class to obtain the next state for the current cell. If the evaluation is successful for a cell, the new result will be stored into the cache table for future references.

Due to the data access locality characteristic of many Trend programs, the evaluation performance can be improved dramatically with this three-phase strategy. Actually, for some Trend rule sets (such as the Game of Life rule) that only modify a small area of the cellular automata space during each epoch, cell evaluation can all be finished in the first two phases (i.e. a 100% hit rate) without going into the third phase after a few iterations.

2.3.2 Graphical interface optimization

For many simulation systems, GUIs provide users an interactive mechanism that is more intuitive than other interface styles. In addition, cellular automat programming can be made more attractive with the visualization of simulation results. From a programmer's perspective, however, GUIs are significantly more complex to implement than other interfaces styles. A badly designed GUI makes itself a bottleneck of the whole system. To improve jTrend, many techniques have been developed to optimize its graphic performance. In the following, we introduce some important techniques used in jTrend.

DIFFERENT bit

As we mentioned in Section 2.3.1, during cell evaluation, a new flag bit, CHANGE, is used to denote whether a cell has been changed between epochs. The CHNAGE bits of a neighborhood are used in the first phase of the three-phase strategy to decide whether it is needed to continue with the cell evaluation. Similarly, for better graphic updating performance, we use the DIFFERENT bit to decide whether a cell needs to be redrawn on the screen. If only a few cells in a cellular automata space are modified, this technique can improve the performance of the graphic module significantly. To show the mechanism of this technique, a scenario where the DIFFERENT bit is used is presented in the following.

In jTrend, users can decide the graphic refresh rate. The refresh rate allows the graphic module to redraw the cellular automata space at a specified frequency instead of each epoch. For example, if the refresh rate is set to 10, the graphic updating module will not repaint the cellular automata working space until the tenth epoch. During this period, the DIFFERENT bit of a cell will not be changed if its value has not been changed at all. At the 10th epoch, the graphic module checks the DIFFERENT bit of a cell. If it is 0, repainting this cell is unnecessary, and the graphic module will skip the cell. If a cell accumulates changes during the past ten epochs, its DIFFERENT bit will be set and it will be redrawn by the graphical module.

One point that should be mentioned is the difference between system simulation repainting and exposure repainting. Unlike the former case that is always triggered by the evaluation module, the latter case only happens when the cellular automata space exposed by a covering window. In the former case, jTrend knows that only cells whose DIFFERENT bits are 1 need to be repainted. But for the latter case, the covered area becomes dirty and has to be cleared and repainted. All cells inside this area needs to be redrawn no matter their DIFFERENT bits are 0 or 1, but after the redraw, they will be reset to zero again.

XOR mode

There are two painting modes in a Java graphics class: *paint mode* and *XOR mode*. In the *paint mode,* everything drawn replaces whatever is already on the screen. For example, if we draw a red square using the *paint mode*, we get a red square, no matter what was underneath. The behavior of *XOR mode* is very different. The idea behind *XOR mode* is that drawing the same object twice will return the screen to its original state. This characteristic makes *XOR mode* a good choice for animation in simulation systems.

In jTrend, each cell can be divided into different fields. Instead of overlapping each other, these fields should be shown at the same time. This requirement makes *XOR mode* painting the choice for drawing the state symbols of different fields.

In fact, the *XOR mode* painting is an efficient solution for graphic updating in jTrend and allows more flexibility. As we mentioned before, jTrend enables users to display some cell fields but turn off the others. To implement this, one alternative solution is to clear the whole cellular automata space and redraw fields that are displayed. Obviously, this is not an efficient approach. The *XOR mode* drawing appears to be more effective in this scenario, because we just need to draw the undesired fields against the existing screen. According to the characteristics of the *XOR mode* painting, these fields will disappear automatically.

Clip rectangle

While developing a GUI application, some simple techniques may have significant performance implications. Using clip rectangles is one of them.

The clip rectangle of a *Graphics* class object in Java is set to the area of the component that is in need of repainting. From a programmer's point of view, this area is the dirty region that requires repainting. jTrend uses this information to narrow the drawing operations. When jTrend receives an exposure event triggered by the system, it knows what areas have become dirty and need redrawing. Instead of clearing the whole cellular automata space that is always larger than the actual dirty area, jTrend only clears the clip rectangle area and redraws cells within this area. This technique leads to smart painting and reduces the graphic repainting overhead.

Canvas space

In jTrend, the actual cellular space size might be larger than the viewable area in the main window, as shown in Figure 2.16. Two scrollbars are provided to allow users view the whole cellular automata space by scrolling.

Since users can only observe part of the cellular automata space, there is no need to render cells within the whole cellular automata space. Only cells within the viewable area should be repainted. jTrend maintains a class named *Canvas* to manage the viewable area repainting. Instead with rendering the whole cellular automata space, this implementation is much more efficient.

Buffered images

For visualization purposes, each state value of a field has a corresponding ASCII symbol defined in the template window. These symbols are used to represent field values on the main window during simulation.

Even though Java provides functions for drawing characters (or strings) on screen directly, it is not efficient to render these symbols this way. The reason is that drawing characters on a graphical component is very inefficient in Java. Further more, some state values have to be represented by a character in different orientations. For example, the

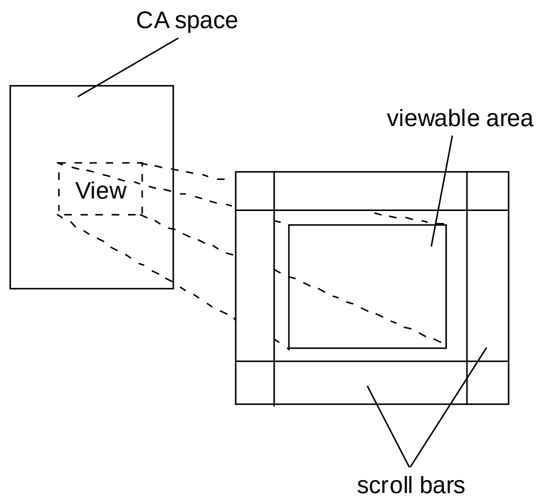


Figure 2.16 The relationship between viewable area and a cellular automata space

Trend value ’>,1’ requires the graphical module to use the character ’>’ in 90^o clockwise rotation. This makes direct character drawing impossible in jTrend.

On the other hand, in Java, rendering an image is more efficient than drawing a string since image rendering is always implemented in native methods. In order to obtain the maximal simulation performance, jTrend uses images for cell symbol rendering. jTrend maintains a dynamically generated image library for all state values defined in the template design dialog such that each symbol has a corresponding pre-created image stored in this library. Every time the template is changed, jTrend will recreate the state symbols in the library. During a simulation, the graphical module of jTrend uses these images to render the cellular automata space instead of drawing characters.

2.4 Availability

jTrend is released as a Java archive (JAR) file. The JAR file format allows a programmer to bundle multiple files into a single compressed archive file for efficient distribution. Typically a JAR file contains the class files and other resources associated with an application. Java allows users to execute the application in a JAR file without extracting its constituent class files. This feature makes JAR file more portable than other releasing formats.

In order to run jTrend, users have to install the Java Runtime Environment (JRE) in his machine. JRE can be downloaded from the Java website of Sun Mircosystems Inc. (http://java.sun.com) for free. jTrend requires JRE not less than version 1.2, since jTrend uses Java Swing, a new graphical toolkit that is only available in JRE version 1.2 or its higher versions.

Because Java is a cross-platform environment, Java programs are guaranteed to “write once, run everywhere”. Therefore, jTrend can be run on most popular operating systems, such as Unix, Windows, Linux, and Mac OS X, etc. jTrend is widely available for users using most major platforms.

The release version of jTrend, including its tutorial and documentation, can be downloaded from the website of Iowa State University Complex Computation Laboratory (http://www.complex.iastate.edu). Some examples can also be found in the website.

CHAPTER 3. TREND PROGRAMMING

As we mentioned before, cellular automata have been used in many research areas. However in these fields, researchers always use cellular automata as simulation models, instead of studying cellular automata itself. There is little research focusing on the relationship between cellular automata and sequential machines, and how to apply cellular automata on traditional computational problems.

In this chapter, new solutions for two traditional problems, *bubble sort* and *satisfiability* (SAT), are illustrated using jTrend. To emphasize the difference between the traditional sequential computation model and the cellular automata model, we focus on how to convert traditional sequential algorithms into cellular automata models. The performance of our new solutions is also discussed in this chapter. The research introduced in this chapter shows us the benefits of cellular automata in solving these traditional problems, and suggests that it may be advantageous to solve other problems using cellular automata in future.

3.1 A Cellular Automata Bubble Sort Algorithm

Sorting is one of the most common activities performed by computers. In order to access the information more efficiently, many algorithms incorporate sorting as an intermediate step to reorder information. In this section, a bubble sort algorithm implemented using the Trend language is introduced. The main idea of this algorithm is to use locally formulated rules that are executed in each member of a numerical array in parallel. In addition to one-dimensional arrays, this algorithm has been extended to sort two-dimensional arrays in a cellular automata space.

3.1.1 Traditional bubble sort

Briefly, sorting is the job of rearranging a list of numeric numbers in decreasing (or increasing) order. Since many applications require the sorting of a large amount of items (such as telephone books, purchasing records, etc.), the efficiency of a sorting algorithm is very important. Many sorting algorithms have been studied before, and bubble sort is one of the most basic sorting algorithms among them.

Traditional bubble sort algorithm runs using two layers of iterations. During each inner iteration, the value of each array element *A* is checked against the value of the element *B* that follows it. If the value of *B* is less than *A*'s value, they will be switched positions. Thus, at the end of each outer iteration, the largest element of the unsorted array is placed in its proper position, and the length of unsorted array is shortened by one. Using induction, we can infer that, if we keep permutating the array using the same condition of switch, eventually this algorithm will lead to a sorted array. The pseudocode of the bubble sort algorithm is listed in the following.

```
for (i=0; i<N-1; i++) {  // The outer loop;
  for (j=0; j<N-1-i; j++) // The inner loop;
    if (a[j+1] < a[j]) {  // if a[j+1] less than a[j], switch their
      temp = a[j];        // positions
      a[j] = a[j+1];
      a[j+1] = temp;
    }
 }
```

As we can see from the pseudocode, the algorithm consists of two nested loops. The main function of the inner loop is to put the maximal number at the end of unsorted array, while the outer loop is used to maintain a variable `i` that indicates the length of sorted array. Obviously, the length of sorted array will increase by one after each outer iteration. In particular, after the first outer iteration, the largest element is placed at the right end of the array `a`, and the variable `i` becomes 1. Similarly, the second largest element is in its proper position on the right after the second outer iteration, and so on. This procedure is repeated until the entire array is sorted. Obviously, the upper bound of the inner loop is `N-1-i`, so the time complexity of the traditional bubble sort algorithm is as follows

$$\sum_{i=0}^{N-1}(N-1-i) = (N-1)+(N-2)+(N-3)+\cdots+1+0 = \frac{(N-1)N}{2} = \mathrm{O}(N^2)$$

Compared with other algorithms such as *quick sort*, the bubble sort is not an efficient approach since it has a worse case time complexity. Even *selection sort* does a better job than bubble sort. However, because of its special characteristics, this algorithm can be easily converted into a cellular automata model.

3.1.2 One dimensional Trend bubble sort

Since a cell in the cellular automata space can only access the information of its adjacent neighbors but not all other cells, this characteristic makes it difficult to convert many traditional sorting algorithms that require maintaining global information about the sorting progress. Fortunately, the bubble sort introduced above is not one of them.

The two global variables used in the traditional bubble sort algorithm can be easily removed. The variable `i` is used to maintain the size of an unsorted list to be checked in the inner loop and the variable `j` is maintained for scanning over the elements of the current unsorted list in succession. The sequential bubble sort requires these two variables in traditional computational model, but a cellular automata model has no need of updating each element in an array sequentially. Actually, cells in the cellular space can execute the switch rules in parallel. Therefore, it is possible to implement the bubble sort on cellular automata without these two variables. This approach, which is called *Trend bubble sort*, is illustrated below.

In the Trend bubble sort, no centralized control is used in sorting the elements of the array. Instead, each element sorts itself by comparing its value with the values of its neighbors. If one element notices its value is less than the value of its left neighbor, this element has to swap its value with this neighbor. Similarly, if its value is greater than the right neighbor's, they have to swap their values as well. This approach will lead to a sorted list eventually. The rules of this algorithm are listed below, in Trend language:

```
default value = value;
default flip = flip;

nbr target;

if(flip == ‘<’) {
  target: = we:;

  if(value != 0 && target:value != 0)
    if(target:value > value)
      value = target:value;

  flip = ‘>’;
}
else {
  target: = ea:;
```

```
    if(value != 0 && target:value != 0)
      if(target:value < value)
        value = target:value;

    flip = ‘<’;
  }
```

The Trend bubble sort rules begin with two default statements. If no rule is applicable for the current cell, the values of the *value* and *flip* fields will be unchanged.

As introduced in chapter 1, each cell in Trend can be divided into different fields to simplify cellular automata programming. In the Trend bubble sort model, each cell is divided into two fields, *value* and *flip,* shown in Figure 3.1. The *value* field contains the integer that is currently being stored in a cell; and the *flip* field is used to determine the comparison target. Because the Trend bubble sort discussed here is one-dimensional sorting, comparisons only happen between the current cell and its horizontal neighbors (i.e. east and west neighbors). Hence the *flip* field uses two special arrow symbols, > and <, to represent the two horizontal comparisons. If the symbol of the *flip* field is ‘>’, then the current cell needs to compare its value with its east neighbor since the arrow is pointing to the east. Similarly, if the symbol is ‘<’, the comparison target is the west neighbor of current cell.

In the Trend bubble sort, if two adjacent numbers in the list are unsorted, they have to be swapped in order to obtain a sorted list. Unlike the traditional bubble sort, each cell accomplishes the swap operation by itself in Trend bubble sort, instead of being a passive recipient of the swap action preformed on them. From the cells’ point of view, to accomplish the swap operation, they have to copy the numbers from each other at the same time. Therefore, the swap operation has two parts and each cell should take care of

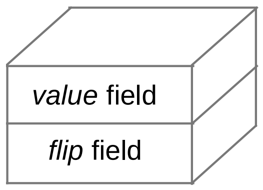


Figure 3.1 The data fields used in the Trend bubble sort

one part. To group cells in pairs, the *fli*p field is used to divide the cells. Each pair consists of two cells whose *flip* fields look like ><. During each iteration, cells in the same pair compare their numbers with each other and copy the numbers from each other, if necessary. In the next iteration, cells will change their *flip* fields and compare their numbers with the neighbor on the other side. For example, if the *flip* field of cell *A* is '>' at the ith epoch, its *flip* field will become '<' at the (i+1)th epoch. An example showing how the cells in the cellular automata space change their *flip* field is presented in Figure 3.2. Obviously, this procedure will lead to a sorted list eventually if we can maintain correct flip patterns for the cells.

The Trend bubble sort is similar to the *odd-even transposition sort* designed for parallel computers with one-dimensional mess architecture [22]. Because the time complexity of the odd-even transposition sort is *O(n)*, the time complexity of the Trend bubble sort is the same. In addition, since [22] has proven that the odd-even transposition sort is an optimal algorithm for parallel models, it can be concluded that the Trend bubble sort is also an optimal sorting algorithm for cellular automata. It should be pointed out that, since the traditional bubble sort is mainly driven by local comparisons between adjacent elements in an array, it could therefore be converted into cellular automata model easily. For other sorting algorithms that use critical globals in rearranging elements such as quick sort, it will be difficult to translate them into cellular automata models.

value	2	7	5	4	1	3	6	0	epoch 1	*value*	2	1	4	0	7	3	5	6	epoch 5
flip	>	<	>	<	>	<	>	<		*flip*	>	<	>	<	>	<	>	<	
value	2	7	4	5	1	3	0	6	epoch 2	*value*	1	2	0	4	3	7	5	6	epoch 6
flip	<	>	<	>	<	>	<	>		*flip*	<	>	<	>	<	>	<	>	
value	2	4	7	1	5	0	3	6	epoch 3	*value*	1	0	2	3	4	5	7	6	epoch 7
flip	>	<	>	<	>	<	>	<		*flip*	>	<	>	<	>	<	>	<	
value	2	4	1	7	0	5	3	6	epoch 4	*value*	0	1	2	3	4	5	6	7	epoch 8
flip	<	>	<	>	<	>	<	>		*flip*	<	>	<	>	<	>	<	>	

Figure 3.2 One example of one-dimensional Trend bubble sort

3.1.3 Two dimensional Trend bubble sort

The Trend bubble sort introduced in the previous section is designed to sort one-dimensional arrays. Because jTrend is a two-dimensional cellular automata simulation environment, this cellular automata sorting algorithm can be revised for sorting two dimensional arrays. Sorting in two-dimensional cellular automata space allows more data to be processed at the same time.

The two-dimensional Trend bubble sort is similar to the one-dimensional one except that the cells' flip field has four different choices instead of only two. These four choices represent four comparison directions (north, south, east, and west) that are shown in Figure 3.3. In addition to comparing with its east and west neighbors, each cell in the cellular space now compares with its north and south neighbors as well. If a cell's value is larger than its east or south neighbors, it will switch its value with these neighbors using the same mechanism of one-dimensional Trend bubble sort. Gradually, the smaller numbers will migrate to the upper-left corner of the cellular automata space while larger numbers move towards the lower-right corner. The rules of two-dimensional Trend bubble sort can be found in APPENDIX A.

3.2 A Migration Cellular Genetic Algorithm for Solving SAT Problems

The satisfiability problem (SAT) is the problem of finding a truth assignment that makes a boolean predicate satisfiable (i.e. evaluated to true.) Since the SAT problem is related to many practical problems such as mathematical logic, constraint satisfaction, etc., efficient methods for solving the SAT problem are very important.

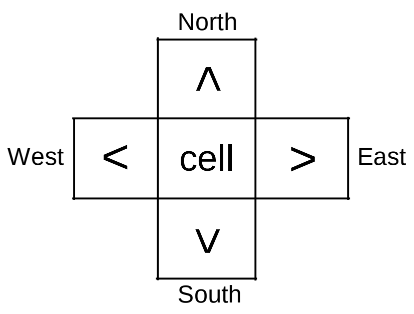


Figure 3.3 The four comparison directions of two-dimensional Trend bubble sort

However, since the SAT problem is NP-hard, there are no polynomial time algorithms that guarantee optimized solutions. That is why certain approximating algorithms are popular alternatives. Among them, the *Genetic Algorithm* (GA) [29] is often used. In this section, we propose a new algorithm, *migration cellular genetic algorithm* (MCGA), for solving SAT problem efficiently by combining the *Cellular Genetic WSAT* (*CGWSAT*) algorithm with a random migration. The experimental results at the end of this section show that MCGA can lead to faster convergence in finding the solutions while maintaining a highly diversified population.

3.2.1 SAT problem and local search solutions

Before we introduce the algorithms for solving SAT problems, it is helpful to provide a brief definition of the SAT problem first. A SAT problem is composed of three basic components [16]:

1. Boolean variables: $x_1, x_2, x_3, \ldots$;
2. A set of literals. A literal in a boolean formula is an occurrence of either a variable (x_i) or its negation ($\neg x_i$).
3. A set of *n* distinct clauses: $C_1, C_2, \ldots, C_n$. Each clause consists of literals combined only by the logical *or* ($\vee$) operators.

A SAT problem is always represented by a predicate in the *conjunctive normal form* (CNF) as $C_1 \wedge C_2 \wedge \ldots \wedge C_n$, where $\wedge$ is the logical *and* operator. The SAT problem is to determine whether there exists an assignment of boolean values to its variables that makes its CNF formula satisfiable, i.e. can be evaluated to true.

Satisfiability problems play a key role in a broad range of research fields, most of which involve solving a combinational optimization problem. Unfortunately, SAT problems are NP-hard problems [7]. The time complexity to decide all satisfiable assignments for a predicate is exponential 2^n, where n is the number of variables involved. Therefore, finding an efficient approximating algorithm for NP-hard problems such as the SAT is of great importance in practical problem solving. Over years, there have been a lot of approximate methods proposed for solving SAT problems. Among them, methods that are based on local search have attracted a lot of attention ([15], [16]).

Local search is an efficient heuristic algorithm devised to solve many NP-hard problems. For most search problems, such as the SAT problem, local search can often achieve many orders of magnitude of performance improvement over brute-force ones.

The main idea behind local search is to find optimal points in a discrete space. Many popular techniques, such as simulated annealing and genetic algorithm, are either local search algorithms or variants of them. One popular local search algorithm for solving the SAT problem is the GSAT method proposed by Selman et al. in 1992 ([24]).

Initially, GSAT starts a greedy local search with a randomly generated truth assignment. From there, this algorithm tries to obtain the largest increase in the total number of satisfied clauses by changing (i.e. ‘flipping’) the assignments of the variables. This procedure is repeated as needed until an answer has been found (with a upper bound MAX-TRIES). The experimental results [24] show that GSAT does outperform some traditional approaches such as backtracking search. However, since GSAT only allows flipping variables that lead to a decrease of the total number of unsatisfied clauses, sometimes it can get trapped within local minima, and never find a solution. To diffuse the information slowly and avoid getting trapped, an extension to the GSAT method, the *random walk SAT (WSAT)* method, was proposed in [25]. WSAT differs from GSAT in the selection of the variables to flip. GSAT only flips the variables that decrease the number of unsatisfied clauses, but with a random walk strategy, WSAT allows the number of unsatisfied clauses to increase with some probability. This random strategy allows WSAT to escape from local minima and find the solutions more quickly.

3.2.2 Cellular genetic algorithms

Genetic algorithms (GAs) are powerful heuristic search strategies based on a simple model of natural evolution ([17], [29]). Starting with a population of individuals (i.e. strings), GA uses *selection* and *reproduction* operators to evolve the individuals that are successful, as measured by a fitness function. The individuals with higher fitness values will be chosen for reproduction with higher probabilities. The reproduction step is accomplished by two genetics operators, *crossover* and *mutation.* Crossover combines two candidate strings to produce new strings with bits from both parents, which allows searches to start from new points. On the other hand, mutation flips bits of the string randomly, which creates new candidate strings. With these two operators, generations of strings can be produced for evaluation until answers are found. Because of its tremendous effectiveness, GA has been used both as tools for solving practical computational problems and as scientific models of the evolutionary processes

Essentially, searches within the GA are parallel because each individual string in the population can be viewed as a separate search. Due to its inherent parallelism, cellular automaton is a perfect model for implementing genetic algorithms. In 1993, D. Whitney first proposed an approach for conducting genetic algorithms using a cellular automata model [27]. In his paper, cellular automata are used as frameworks to enable a fine-grained parallel implementation of GA, which is called the *cellular genetic algorithm.* Each individual string in his genetic algorithm is mapped onto a cell in the cellular automata space. Strings in the cells then select mate partners that have the best fitness in the neighborhood; and then crossover with their partners. The mutation operation is then applied to the new strings after crossover.

Combining cellular genetic algorithm with WSAT, a new algorithm called *Cellular Genetic WSAT* (*CGWSAT*) was proposed by Folino et al ([12], [13]) for solving SAT problems. In CGWSAT, new strings are generated in each cell by crossing with one of their neighbors, while the mutation operation is carried out using the WSAT strategy. The CGWSAT algorithm has been implemented on a CS-2 parallel machine using the cellular automata language CARPET that was introduced in Chapter 1. Experiments show that CGWSAT converges better than both sequential and parallel WSAT methods for all test cases.

3.2.3 Migration cellular genetic algorithms (MCGA)

In the cellular genetic algorithm, the string in a cell can only recombine with strings in its immediate neighborhood. For two strings that are far away from each other in a cellular space, numerous crossover and mutation steps have to happen before these two strings get the chance to mate. Obviously, this local interaction property of cellular automata prevents strings from migrating quickly, and it slows down the speed of finding the solutions. Since cells only interact with their adjacent neighbors, this algorithm tends to get trapped in local optima and will not lead to diversified solutions.

One possible improvement to these problems might be to expand the neighborhood where the recombination can occur. However, the cellular automata definition limits the size of the neighborhood (for example, in jTrend, the maximal neighborhood size is 11x11.) A larger neighborhood, even when available, still cannot solve these problems satisfactorily. We propose a new approach, the *migration cellular genetic algorithm (MCGA),* for solving SAT problems effectively without the need to expand neighborhood. With the help of a migration mechanism, MCGA has the ability to escape from local optima. This algorithm has been implemented using the jTrend cellular automata simulator.

In detail, MCGA is based on the CGWSAT algorithm but introduces a new migration strategy that allows strings to migrate in the cellular automata space. In MCGA, individual strings are mapped onto the two-dimensional cellular space and each cell in the space can only carry at most one string at any time. MCGA only uses some cells to store strings. Cells that are not occupied by any string are used as pathways for strings to migrate. MCGA uses a fitness function, which is defined as the number of unsatisfied clauses, to evaluate strings. A string whose fitness value is zero is a solution to the SAT problem because it has no unsatisfied clauses. Such strings are named as *solution strings* in this thesis. To prevent solution strings from contaminating other individuals, the migration strategy of MCGA only allows strings whose fitness value is **not** zero to move in the cellular automata space. Once these migrating strings find solutions, they will stay in the same cells forever.

With the new migration strategy, strings in MCGA can move around in the cellular space. Therefore they have more chances to crossover with other strings. That helps to maintain the diversity of the solution population. In addition, since the strings with fitness zero are fixed in the same cell of the cellular space, MCGA should be able to prevent these strings from influencing other strings. This technique is also helpful in improving the diversity of MCGA algorithm.

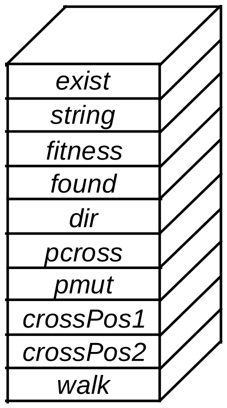
3.2.4 The implementation of MCGA

The pseudo-code of the MCGA algorithm is listed below. Each cell decides its next state according to this algorithm in parallel. This algorithm has been implemented on jTrend using the Trend programming language, which can be found in APPENDIX B.

```
if(cell has a string) {
  if(cell is not a solution) {
    if(does not need to migrate) {
      partner = DecidePartner(north, northeast, east,
                southeast, east, southwest, west,
                northwest);
      if(partner is not center cell) {
        if(pcross > CROSS_RATE)
          Crossover(cell, partner);
        Mutation(cell);
        fit = Evaluate_fitness(chrom);
        Update(cell_chromosome, chrom);
        Update(cell_fitness, fit);
      }
    }
    else {
      Move_around(cell);
    }
}
```

MCGA adopts the Von Neumann neighborhood, with each cell divided into ten data fields that is shown in

Figure 3.4. The function description of each field is also shown in this figure. At the beginning of a simulation, the cellular automata space is initialized with a population of strings along with their fitness values. During each iteration, a string with non-zero fitness value looks around its neighborhood and decides it crossover partner according to the



Field Name	Field Function Description
exist	determine whether there is a string existing in the current cell
string	the string in the current cell
fitness	the fitness value for current string
found	determine whether the string is the solution for current predicate
dir	the direction for cell with fitness value 0 to migrate
pcross	the random number of crossover rate
pmut	the random number of mutation rate
crossPos1	the first point for two-point crossover operator
crossPos2	the second point for two-point crossover operator
walk	the random number of random walk rate

Figure 3.4 The fields defined in MCGA and their function description

fitness values of neighbors. In the cellular genetic algorithm, cells only crossover with the neighbor that has the lowest fitness value. As a result, that approach tends to get trapped in local minima. To eliminate this problem, our MCGA algorithm
adopts the WSAT algorithm in crossover selection and allows cell to mate with strings that have a higher fitness value with some probability. This strategy is called *downward move*. It allows MCGA to escape from the local optima and continue the searching from a new starting point. The crossover operation used by MCGA is a two-point crossover, as shown in Figure 3.5. Current cell choose one of the two offspring as its new string. The crossover partner of the current cell does the same thing simultaneously.

For the mutation operation, MCGA adopts the mixed random walk strategy proposed by [25]. With this strategy, the variable that can causes the best decrease in the number of unsatisfied clauses is flipped with a probability `f`; and with probability `(1-f)`, another variable that appears in some unsatisfied clause is picked randomly and its assignment value is flipped.

In MCGA, strings that are not the solutions for a SAT problem are allowed to move in the cellular automata space. The strings use *dir* field to decide whether to migrate, or to do the crossover and mutation operations. To allow strings move around in cellular automata space, a handshake mechanism is used in MCGA. The detail of the handshake mechanism is shown in Figure 3.6.

As we can observe from Figure 3.6, the MCGA algorithm uses the *dir* field to direct its migration operation. Every time when an empty cell *A* finds that its neighbor cell *B* it points at has a partial solution for the SAT problem (i.e., its fitness is not zero), it will inspect the *dir* field of cell *B*. If cell *B*'s *dir* field is also pointing towards *A*, *A* will copy some data from *B*, including *fitness, string, exist*, etc. At the same time, cell *B* will notices

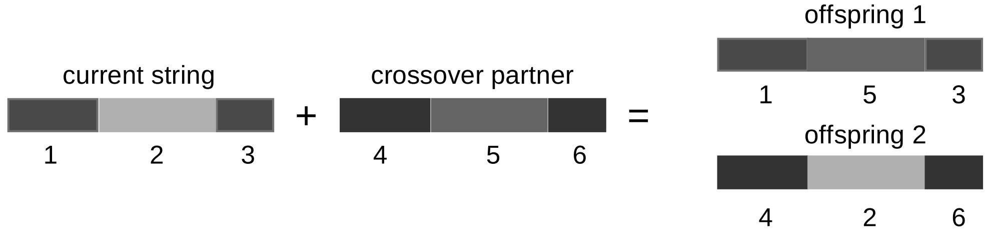


Figure 3.5 The mechanism of two-point crossover in genetic algorithms

that *A* points at itself, and copies the same data fields from cell *A*. After these simultaneous exchanges, one step of the migration operation is achieved. In essence, the migration mechanism of MCGA is similar to the swap operation introduced in the jTrend bubble sort earlier.

With the help of migration, we expect that the MCGA algorithm can escape from the local minima and find more answers faster than CGWSAT. However, the potential problems that can plague this algorithm should also be considered seriously. Premature convergence is one of these problems. For a genetic algorithm that can find a solution quickly, its population might become saturated with similar answers. Thus, the pool of candidate answers would converge to some sub-optimal points in the space being searched, and the diversity of the whole population would become low. In the next section, we compare the convergence speed and diversity of MCGA and CGWSAT. These experiments were conducted with jTrend running on a Linux machine with Pentium III processor and 256 Mbytes of memory.

3.2.5 Experimental results and analysis

Due to the 64-bit cell size limit of jTrend, we can only test SAT predicates with no more than 32 boolean variables. In our experiments, each predicate is composed of 48 clauses with 16 variables, and all predicates are generated randomly by computer. Since there are 16 variables, the possible solutions (i.e. strings) that can make a predicate true are 65536 (2^{16}). However, only a small fraction of these strings are actual solutions for a predicate. For example, the first predicate in our test cases has only 131 solutions out of

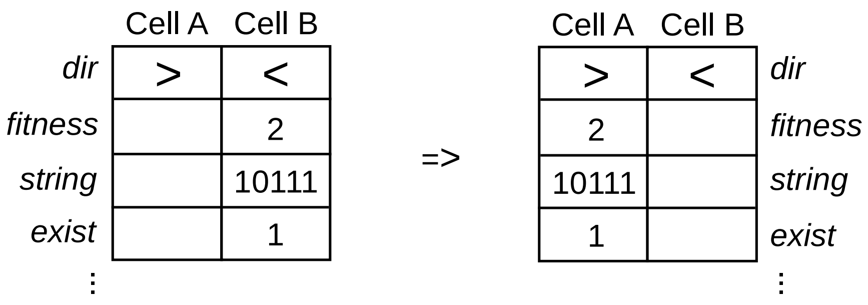


Figure 3.6 The handshake mechanism of MCGA for migrating strings

the 65536 possible strings. To study the distribution of solution numbers, we randomly generated 20 sample predicates and count their solution number. The distribution of solution numbers is listed in Table 3.1. As we can observe, most predicates only have 0 – 200 solutions.

Intuitively, we believe that the size of a cellular automata space can influence the performance of MCGA, because the bigger the cellular automata space size, the sparser the string distribution will be under a fixed population size. The chance for unsolved strings to crossover with others is lower since there might be no strings surrounding them. Due to these reasons, we have to study how the cellular automata space size affects the performance of the MCGA algorithm, and decide the best size for this algorithm before we compare it with CGWSAT.

First, we want to compare the convergence speeds of MCGA that runs under different space sizes. We start with 400 strings in the cellular space. To allow strings with the best fitness value to migrate in the cellular space, there should be extra empty cells for these strings to move. Therefore, the cellular space size must be greater than 20x20 (400). In this experiment, we tested four different space sizes (40x40, 35x35, 30x30, and 25x25.) The results are shown in Figure 3.7. Four predicates are chosen in this experiment according to the solution number distribution of Table 3.1. The solution numbers of these four predicates are 621, 25, 107, and 211 respectively, which represent a typical distribution of these 20 predicates we mentioned earlier.

It is obvious from Figure 3.7 that the smaller the cellular automata space, the faster these initial 400 strings become solutions of a predicate. Since cells in larger cellular

Table 3.1   The solution number distribution of 20 SAT sample predicates

Range	Number of predicates	Range	Number of Predicates
0 – 49	6	250 – 299	0
50 – 99	2	300 – 349	0
100 – 149	5	350 – 399	1
150 – 199	4	400 or above	1
200 - 249	1		

space have fewer chances to crossover, the results are reasonable and seem to follow the predication we made.

To understand how the cellular space size correlates with the diversity of solutions found, we must have a quantitative measurement of diversity. In our experiments, we use *diversity*, which is defined as the percentage of **distinct** solutions existing in a solution population, to measure the quality of solution strings. For example, if there are 400 strings carrying solutions for a predicate, but only 250 strings are unique, then the diversity of the population is 250/400 = 62.5%. Note that we define the diversity to be 100% if the population size is 0.

The diversities with the four different space sizes are shown in Figure 3.8 for the same four predicates. Note that the diversities are the highest (100%) at the beginning of a simulation (epoch 0). As the simulation continues, the diversities decrease and approach a fixed value eventually. The reason for such behaviors is the crossover operation of cellular genetic algorithms. In the traditional genetic algorithm, whenever crossover happens

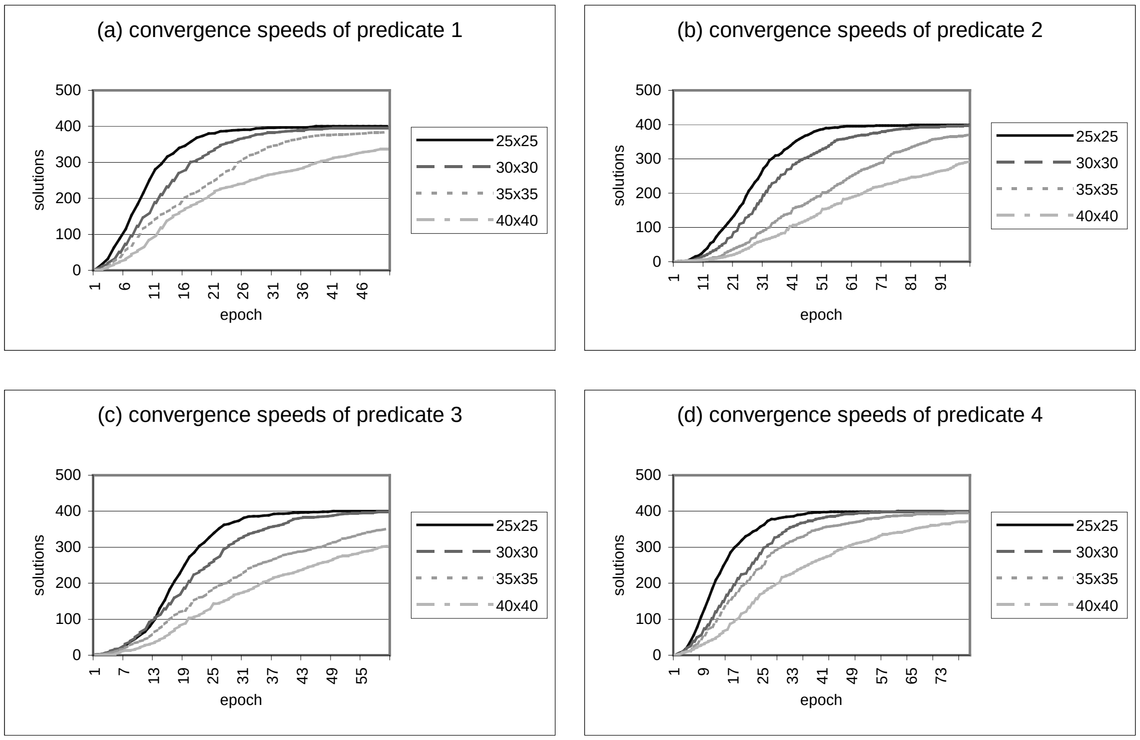


Figure 3.7 The convergence speeds of MCGA with different space sizes

between two strings, each of them will pick one offspring. Therefore, all information is still kept in the population. However, in cellular genetic algorithms, only one offspring is kept in the current cell. The other offspring is tossed away, so the total information in the gene pool decreases. That tends to make the whole population becoming the same. Toward the beginning of a movable cellular genetic algorithm, the effect of crossover operations is relatively small because few crossover operations will happen at the onset. At this time, it is easier for strings to evolve independently, and the diversity can be maintained at a higher level. As the simulation goes on, the effect of crossover operation will gradually become obvious. Every strings exchange information with its neighbors through crossover. Even though mutation can alleviate this unification effect a little bit, it is still possible for the whole population to evolve in the same direction, and the diversity will suffer. Eventually, when all strings have become solutions, the diversity value will be constant. It should be pointed out that, according to the definition, the diversity is 100% when the population size is 0. That is why diversities are always 100% at epoch 0 in Figure 3.8.

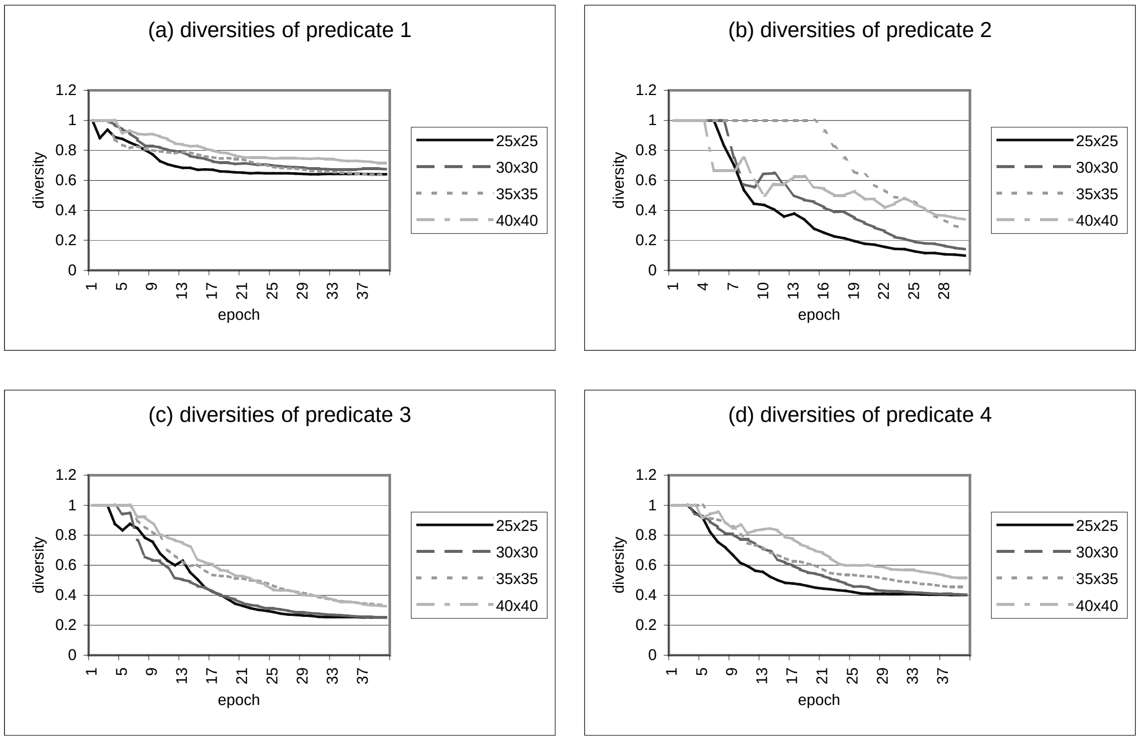


Figure 3.8 The diversities of MCGA with different space sizes

From Figure 3.8, it seem to suggest that larger cellular spaces can maintain populations with higher diversity than the smaller ones. The explanation of this is, the convergence speeds of strings are slower in larger cellular space, which allows strings to have more chances to move faster away and to crossover with other strings. This prevents these strings from finding similar solutions. In addition, we also observe that the diversities of MCGA 35x35 is about the same as, sometimes better than, MCGA 40x40. That suggests that the influence of cellular size on the diversity will become very small when the size is larger than some threshold.

Our main focus is to compare the performance of MCGA and CGWSAT. Based on the observations of Figure 3.8, it can be seen that cellular automata spaces of 40x40 has the best diversities among the four different sizes. It seems that this space size is the best choice for comparing diversity between MCGA and CGWSAT. However, as we

mentioned earlier, MCGA 35x35 has about the same diversity as MCGA 40x40. In addition, the convergence speed of space size 35x35 is better than 40x40. Combining all

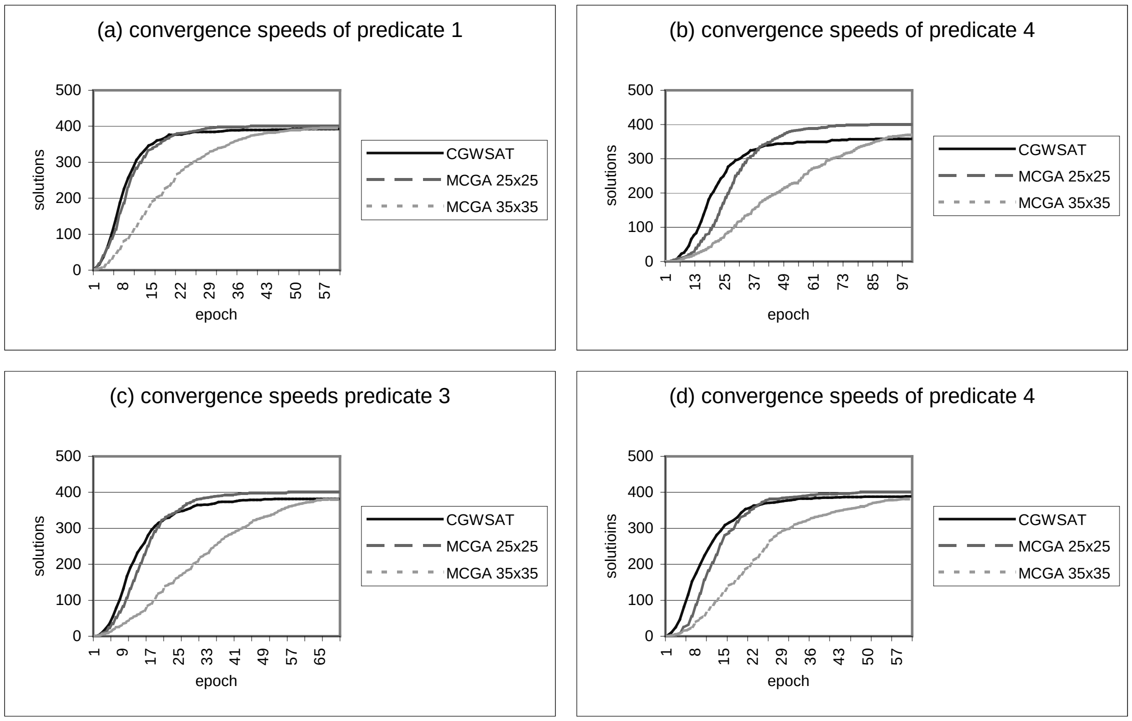


Figure 3.9 The convergence speed comparisons between CGWSAT and MCGA

these facts, we believe that space size 35x35 is a better choice than 40x40 for the MCGA algorithm, if we want to compare the convergence speed and diversity of MCGA and CGWSAT at the same time. Since MCGA 25x25 has better convergence than other space sizes, we also use this size to compare MCGA and CGWSAT in this experiment.

In these experiments, we initialized the cellular automata space with a string population of 400. The crossover operation used in these two algorithms is the same two-point crossover with a crossover probability of 50%. In addition, their mutation rate is the same at 90%. Both MCGA and CGWSAT also use the WSAT random walk strategy with a 50% probability in choosing the random walk direction. The convergence speeds of MCGA and CGWSAT are shown in Figure 3.9. Without lose of generality, we used the same four predicates that are used in the previous

experiments for this comparison. The convergence speeds of MCGA 25x25 are about the same as CGWSAT. But for MCGA 35x35, its convergence speeds are less than the speeds of CGWSAT algorithm in most cases. An explanation of this is the influence of

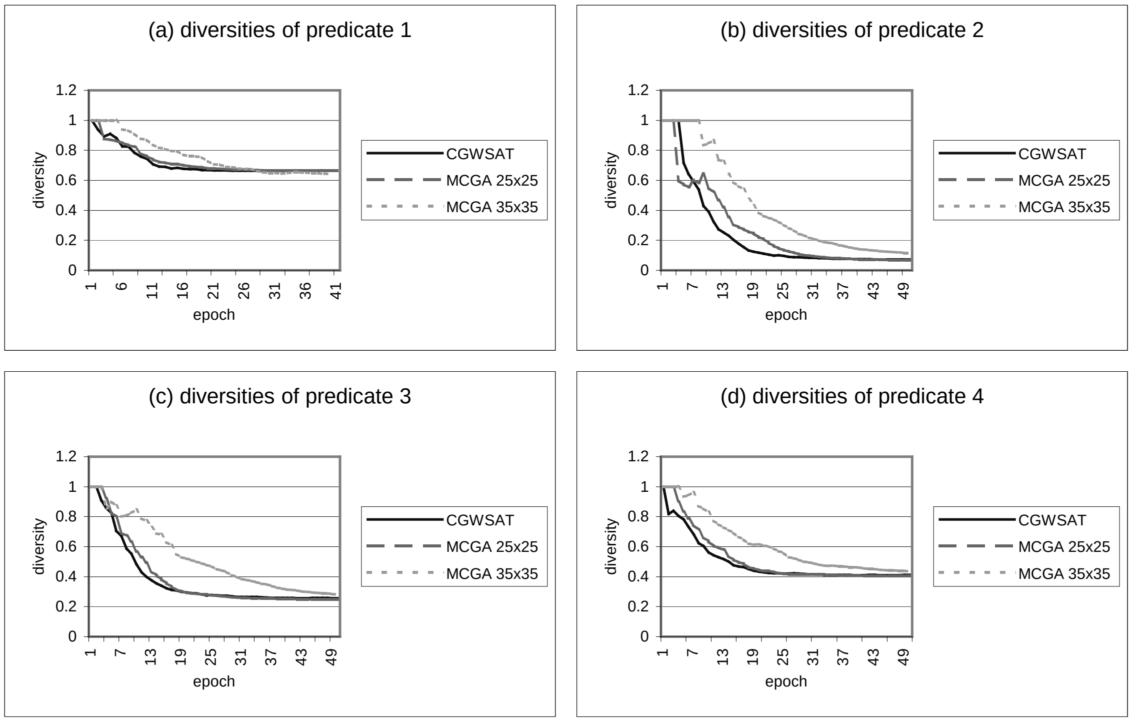


Figure 3.10 The diversity percentage comparisons between CGWSAT and MCGA

cellular automata size we mentioned earlier. The cellular space size used in MCGA is larger than CGWSAT. Since only some of the cells in the MCGA cellular space contain strings, these strings have fewer chances to cross with other strings than in CGWSAT. That slowdowns the convergence speed of MCGA, and makes the convergence speed of MCGA slower than CGWSAT.

We expect the diversity of MCGA to be better than CGWSAT because MCGA disallows solution strings to migrate in cellular space. The experimental results shown in Figure 3.10 follow our expectation, and show that the diversity of MCGA, especially the one with cellular space size of 35x35, is better than the CGWSAT algorithm.

3.2.6 Discussion

Based on all the experiment results presented in the previous section, we can draw the following conclusions. The diversity of solutions in MCGA is better than the diversity of the CGWSAT algorithm. In addition, the MCGA algorithm has a slower convergence speed in finding solutions. If we consider that cellular space size of MCGA is larger than CGWSAT and strings have fewer chances to do crossover, it is not very unreasonable. To increase the diversity of MCGA, we improve MCGA by disallowing strings to cross with the strings that have found the solutions. The experiment results show that, this technique can increase the diversity of MCGA; but it also deteriorates the convergence performance.

With the current implementation of the Migration Cellular Genetics Algorithms (MCGAs), each cell will host a complete string and execute the complicated rules shown in Appendix B. Although it is quite straightforward, it has problems with scalability. When the strings become larger (such as from 32 to thousands), it is impossible for a cell to host the string since the maximal bit depth allowed in jTrend are only 64. A better solution is to design a collaborate framework that allows cells to carry out the complex string manipulation together. In this framework, each cell is designed to be simple and carries a small fraction of the whole string. With this new framework, we can reduce the complexity of each cell, and make it possible to be scaled for lager problems. That will be our future work as discussed in the next chapter.

CHAPTER 4. SUMMARY AND CONCLUSION

Cellular automata are discrete dynamical systems whose step-by-step behavior is completely specified in terms of a local relation. Each cell in a cellular automata space executes the same rule set in parallel using only local interactions. With this intrinsic parallelism, research conducted on cellular automata provides opportunities for the development of scalable algorithms and applications in computational science. In addition, the abstract modeling of cellular automata allows researchers to capture the essential features of systems in which global behaviors arise from the collective effect of numerous simple local interacting components, without the need to physically realize those models in cumbersome fabrications. Cellular automata have been widely used for modeling and simulation of complex phenomena and systems, especially in fluid dynamics, evolution, road traffic flow, artificial life, and quantum physics.

In this thesis, a general-purpose cellular automata simulation environment, jTrend and its implementation issues are introduced. In addition, a bubble sort algorithm for sorting one and two-dimensional arrays using the Trend programming language is also presented. The study of genetic algorithms finally provides a new cellular genetics algorithm, MCGA, for solving SAT problems on a cellular automata model. We compared the convergence speed and diversity of MCGA with the previous CGWSAT algorithm for solving the SAT problem on the jTrend environment. In the following sections, we summarize the major contributions and achievements of this work and discuss some future research possibilities.

4.1 Contributions

In a summary, the major contributions made by this work are as follows:

- A general-purpose cellular automata simulator, jTrend, was developed for the Java runtime environment with high portability.
- An object-oriented, modular design of jTrend makes it more flexible and provides the potential for future performance improvement.
- A three-stage cache strategy implemented in the cell evaluation module of jTrend, which improves the simulation speed significantly.

- One and two-dimensional bubble sort algorithms with an optimal time bound were implemented on cellular automata model.
- A cellular genetic algorithm with string migration capability was designed for solving SAT problems. Performance comparison between the new algorithm and the previous CGWSAT algorithm is presented.

The details of each contribution are discussed below.

4.1.1 A general purpose cellular automata simulator with high portability

jTrend is an integrated general-purpose cellular automata simulator based on the Java runtime environment. In the past, very few general-purpose cellular automata simulation tools were available to researchers. For the few that were developed, either they require special hardware to operate such as CAM-6, or they are available only on some specific platforms, such as Cellang and CARPET. All of these limit the their availability to researchers.

jTrend was implemented on the Java runtime environment. The cross-platform nature of Java makes jTrend widely available to researchers. Almost all major operating systems, such as Windows, Unix, Linux, and Mac OS X can run jTrend. Consequently, more users using different platforms can participate cellular automata research, helping to expand this research domain.

As a general-purpose cellular automata simulator, jTrend also provides all functions necessary to control and monitor the simulations. All cell-related information can be defined by users via an easy-to-use graphical user interface. jTrend also takes care of simulation details and provides backtracking mechanism for debugging. Together with other new features, jTrend turns out to be one of the most powerful cellular automata simulators available today.

4.1.2 Object-oriented system design

Three major software modules in jTrend are the *graphical user interface*, *Trend language compiler*, and *simulation engine*. The graphical user interface manages the interaction between users and the simulation engine. It allows users to control the appearance of their models and monitor its simulation progress. The independently developed Trend language compiler compiles the Trend source program into an executable Java class *RuleSet* on the fly, which is used by the simulation engine to compute new cell values.

The clear job division between these modules makes jTrend highly maintainable and easy to extend in the future. Different modules can be updated without altering the others. For instance, if the Trend programming language needs to be upgraded, only the compiler module has to be modified since the other modules are not related to compiling the Trend programming language.

4.1.3 Three-stage caching strategy

To improve the simulation speed of jTrend, a three-stage strategy is adopted to accelerate the evaluation for each cell. These include *neighborhood invariant skipping, cache table lookup,* and *cell evaluation.* In the *neighborhood invariant skipping* stage, jTrend utilizes a property of cellular automata that a cell will not change as long as its neighbors stay the same. Consequently, this stage can accelerate the simulation speed significantly because most cells remain unchanged during the simulation in many models.

jTrend also maintains a cache table to store the recent evaluation results. Whenever necessary, jTrend lookups a new cell value in the cache table during the *cache table lookup* stage. In many models, jTrend shows to have high cache hit rates. That proves this stage very effective in speeding up simulations. Only when both stages have failed to provide a new cell value will the third evaluation stage come into play. The overall performance of jTrend is therefore much better than previous cellular automata simulators without the two additional skipping/caching mechanisms.

4.1.4 Cellular automata bubble sort algorithm

With the help of jTrend, a bubble sort algorithm with an optimal time bound for sorting numeric integers was implemented in a cellular space. In addition to being able to sort a one-dimensional array, this algorithm was also extended and was able to sort numbers in a two-dimensional array.

In a cellular automata model, there is no notion of updating each element of an array in succession like sequential machines. Additionally, each cell can only see its adjacent neighbors instead of all cells in the whole cellular space. Due to these differences, our new algorithm takes a different approach to sort. Instead of rearranging the array with a global control index, our algorithm only uses local interactions between adjacent cells. Each cell sorts its content by comparing with its neighbors and making a swap operation with them if necessary. Eventually, this local-only approach can lead to a sorted array in the cellular automata space. Due to the intrinsic parallelism of cellular automata, the performance of our algorithm is far better than traditional bubble sort. Its time complexity, *O(n)*, is optimal for parallel models like cellular automata.

4.1.5 Migration cellular genetic algorithm for solving the SAT problem

This thesis proposes a new strategy for solving SAT problems using parallel genetic algorithms. Named as *migration cellular genetic algorithm* (MCGA), this new model is based on the cellular genetic algorithm and random walk SAT (WSAT). By allowing strings to migrate freely around the cellular automata space, this model allows quick escape from local minimal points in the search space. Compared with the CGWSAT algorithm that does not allow migration, this new capability leads to a slower convergence in finding solutions but maintains a much better diversity of solutions found.

Since SAT is an NP-complete problem, solutions for it can lead to potential applications of cellular automata on solving other NP-hard problems, especially combinational optimization problems such as graph coloring, N-Queens, etc. Even though MCGA is still not a polynomial time algorithm for SAT, the parallel nature of this algorithm makes us believe that cellular automata are ideal models for solving many NP-hard problems due to their intrinsic parallelism.

4.2 Future Work

We envision the following possible future topic beyond this work.

4.2.1 Improving the simulation environment of jTrend

Currently, a noticeable limitation of jTrend is its simulation speed, when compared with Unix Trend. Since jTrend was implemented in Java, overhead introduced by the Java runtime environment slows down the simulation of jTrend. Even thought some techniques have been used in jTrend to improve its speed, they do not rule out the possibility to improve it further using other techniques.

According to the profiling results, there are two major bottlenecks within jTrend, *cell evaluation* and *graphic updating*. To improve the simulation speed, a three-stage caching strategy has already been used; but that is still not enough when running complicated rule sets that prevent invariant skipping and cache table lookup to be effective. If we can implement the cell evaluation stage using Java Native Interface (JNI), a technique that allows executing native code, this bottleneck can be eliminated. In addition to JNI, multiple-threading is another possibility that can be added in the future for machines that support multiple CPUs. Multithreaded programming allows program to divide one big task into many small ones and assign these jobs to different threads. Obviously, with this technique, the evaluation time will decrease significantly on a multi-CPU computer.

To reduce the overhead of graphic updating in jTrend, it may be desirable to run jTrend as a background process, totally eliminating its graphical appearances. If jTrend can be run as a background application, the overhead introduced by graphic modules will disappear. This can benefit users who are running long-term simulations using jTrend and do not need to monitor its individual epochs.

4.2.2 Improving the Trend programming language

Another research possibility is to improve the Trend programming language. As a high-level language, Trend is powerful and convenient for users doing cellular automata research. However for some specific cellular automat simulations, this language may not be powerful enough because it does not provide some additional features that are required in these simulations. One example is the floating-point data type. Trend provides users with only three different data types now, *int*, *nbr*, and *fld*. Calculations allowed in Trend are all integer-based. Obviously, these data types are not enough for jTrend to simulate some complex models that require floating point calculations such as a weather model.

Besides more data types, new approaches for data analysis should also be incorporated into the Trend programming language. Currently, if users want to do data analysis, they have to export the content of a cellular automata space into a separate file and use other programs to analyze the simulation results. This process is quite cumbersome and inconvenient for users. If Trend can provide some functions to collect such statistical information automatically, it can save users a lot of time when doing data analysis.

4.2.3 Other cellular automata models

In Chapter 3, two algorithms for solving real-world problems (sorting and SAT) using cellular automata models are presented. Besides these, there are other attractive problems for cellular automata research.

Since cellular automata are intrinsically parallel models, it is useful to use them to solve hard problems that require intensive computation. A lot of research effort has been put into this field, and some cellular automata models have been designed (such as TSP [20]). We anticipate developing new solutions for the following problems on cellular automata: N-Queen, SAT on a self-replication structure, and digital image processing.

As we mentioned before, research on self-replication has attracted lots of attention since cellular automata provide an ideal abstraction for the artificial life research. If we can let a self-replicating loop carry complex codes and allow it to evolve and change based on the fitness of the complex code it carries, much more useful cellular automata behaviors are expected. With genetic algorithms, this approach towards artificial life research may generate many interesting results in the future.

APPENDIX A. TWO-DIMENSIONAL BUBBLE SORT RULE LIST

```
//default rules
default value = value;
default flip = flip;

nbr target; //comparison target

if(flip == ’>,0’) { //flip field is pointing to east
  target: = ea:;
  if(value != 0 && target:value != 0)
    if(target:value < value)
      value = target:value;
}
else if(flip == ’>,1’) { //flip field is pointing to south
  target: = so:;
  if(value != 0 && target:value != 0)
    if(target:value < value)
      value = target:value;
}
else if(flip == ’>,2’) {//flip field is pointing to west
  target: = we:;
  if(value != 0 && target:value != 0)
    if(target:value > value)
      value = target:value;
}
else if(flip == ’>,3’) { //flip field is pointing to north
  target: = no:;
  if(value != 0 && target:value != 0)
    if(target:value > value)
      value = target:value;
}

//change the flip field
if(flip)
  flip = flip  % 4 + 1;
```

APPENDIX B. MIGRATION CELLUAR GENETIC ALGORITHM RULE LIST

```
/**********************default rules******************/
default dir = dir;
default pcross = pcross;
default pmut = pmut;
default fitness = fitness;
default string = string;
default found = found;
default crossPos1 = crossPos1;
default crossPos2 = crossPos2;
default walk = walk;
default exist = exist;
/***********the constants are defined here**************/
int VAR_NUM = 16;
int MAX_CLAUSE = 48;
int MUT_RATE = 14;  // 14/16 == 87.5%
int CROSS_RATE = 8; // 8/16 == 50.0%
int RANDOM_WALK_RATE = 8; // 8/16 = 50.0%

int CLAUSE[] =
{2,16,16384,64,2048,32768,256,32,1,128,2048,1024,8,16384,64,256,2
56,4096,8192,16384,2048,32768,4,4,64,8192,2,16384,4,128,1024,8,16
384,4096,4,32,2048,16,4096,2,4,64,512,2048,16,64,4,8,1024,8192,81
92,16384,2048,128,4096,2048,1024,32,512,2,16,16384,2,512,16,1024,
4096,1024,4096,32768,16384,1024,16384,32768,2,128,512,8192,512,25
6,1,32768,8192,32,16384,16,4,256,32768,2048,512,128,512,32768,163
84,2,8192,16384,4096,64,1024,2048,4096,32768,8192,32768,1,1024,16
384,8192,8,32768,4,4096,2,2048,32768,2048,16384,1024,128,4,2,4,25
6,32768,4,16384,8192,16,32768,4096,512,8192,16,16,1,64,16,16,4096
,1024,8,32};

int VALUE[] =
{1,0,0,1,0,0,1,1,1,0,0,1,1,1,0,1,0,0,0,1,0,1,1,0,0,0,1,1,0,1,0,1,
1,1,0,1,0,0,0,1,1,1,0,0,0,1,1,1,1,1,0,0,1,1,0,0,1,0,0,1,0,1,0,0,1
,1,0,1,1,0,1,1,1,1,1,1,0,1,0,1,0,1,1,0,1,1,0,0,1,0,1,1,0,1,0,1,1,
1,0,0,0,1,0,1,0,1,1,1,1,0,0,1,1,0,1,0,0,1,1,1,1,0,1,0,1,0,1,0,1,1
,1,0,0,1,1,1,0,1,0,0,0,0,1,1};


/******temporary variables are defined here***********/
int tmpFit;
nbr p;
int tmpStr;

/************functions are defined here***************/
int IsSame(int i, j) {
  if(i == 0 && j == 0)
    return 1;
  else if(i >= 1 && j == 1)
    return 1;
  else
    return 0;
}

int EvalFitness(int str) {
  int count;
  int k;
  count = 0;

  for(k = 0; k < MAX_CLAUSE; k++) {
    if(IsSame(str & CLAUSE[3 * k], VALUE[3 * k]) == 1 ||
       IsSame(str & CLAUSE[3 * k + 1], VALUE[3 * k + 1]) == 1 ||
       IsSame(str & CLAUSE[3 * k + 2], VALUE[3 * k + 2]) == 1)
    {
       count++;
    }
  }

  return MAX_CLAUSE - count;
}

nbr DecidePartner() {
  nbr n;
  nbr partner;
  int value;
  partner: = ce:;
  value  = fitness;

  over each other n: {
    if(n:exist == 1) {
      if(n:fitness != 0 && n:fitness < value) {
        partner: = n:;
        value = n:fitness;
      }
    }
  }

  return partner:;
}

void Randomize() {
   dir = ce:dir ^ no:dir ^ ne:dir ^ ea:dir ^ se:dir ^ so:dir ^
         sw:dir ^ we:dir ^ nw:dir;
  pcross = ce:pcross ^ no:pcross ^ ne:pcross ^ ea:pcross ^
           se:pcross ^ so:pcross ^ sw:pcross ^ we:pcross ^
           nw:pcross;
  pmut = ce:pmut ^ no:pmut ^ ne:pmut ^ ea:pmut ^ se:pmut ^
         so:pmut ^ sw:pmut ^ we:pmut ^ nw:pmut;

  crossPos1 = ce:crossPos1 ^ no:crossPos1 ^ ne:crossPos1 ^
              ea:crossPos1 ^ se:crossPos1 ^ so:crossPos1 ^
              sw:crossPos1 ^ we:crossPos1 ^ nw:crossPos1;
  crossPos2 = ce:crossPos2 ^ no:crossPos2 ^ ne:crossPos2 ^
              ea:crossPos2 ^ se:crossPos2 ^ so:crossPos2 ^
              sw:crossPos2 ^ we:crossPos2 ^ nw:crossPos2;

  walk = ce:walk ^ no:walk ^ ne:walk ^ ea:walk ^ se:walk ^
         so:walk ^ sw:walk ^ we:walk ^ nw:walk;
}

int Crossover(int str1, str2) {
  int left, right;
  int mask1, mask2;
  int i, v;

  if(crossPos1 == crossPos2)
    return str1;
  else {
    if(crossPos1 > crossPos2) {
      left = crossPos2;
      right = crossPos1;
    }
    else {
      left = crossPos1;
      right = crossPos2;
    }

    mask1 = 0;
    mask2 = 0;

    v = 1;
    for(i = 0; i < VAR_NUM; i++) {
       if(i >= right || i < left)
         mask1 = mask1 | v;
       v = 2 * v;
    }

    v = 1;
    for(i = 0; i < VAR_NUM; i++) {
       if(i < right && i >= left)
         mask2 = mask2 | v;
       v = 2 * v;
    }

    return ((str1 & mask1) | (str2 & mask2));
  }
}

int Mutation(int str) {
  int t, i, p, j, v, count, position;
  int tmpStr;

  if(walk < RANDOM_WALK_RATE) { //choose the local optimal
    v = 1;
    count = MAX_CLAUSE + 1;
    for(i = 0; i < VAR_NUM; i++) {
      tmpStr = string;
      tmpStr = tmpStr ^ v;

      j = EvalFitness(tmpStr);
      if(j < count) {
        position = i;
        count = j;
      }
      v = v * 2;
    }

    t = 1;
    for(i = 1; i <= position; i++)
      t = t * 2;

    return str ^ t;
  }
  else {
    t = 1;
    p = pmut;

    for(i = 1; i <= p; i++)
      t  = t *2;

    return str ^ t;
  }
}
/******************main body for GA_MOVE*******************/

Randomize();
```

```
if(exist) {
  if(found == 0) {
    rot if(dir == ’>,3’ && no:dir == ’>,1’ && no:exist == 0)
    {
       fitness = 0;
       string = 0;
       found = 0;
       exist = 0;
    }
    else {
       p:= DecidePartner():;

       if(p: != ce:) {
          tmpStr = string;
          if(pcross <= CROSS_RATE)
            tmpStr = Crossover(string, p:string);
          tmpStr = Mutation(tmpStr);

          tmpFit = EvalFitness(tmpStr);

          string = tmpStr;
          fitness = tmpFit;

          if(tmpFit == 0)
            found = 1;
       }
    }
  }
}
else {
  rot if(dir == ’>,3’ && no:dir == ’>,1’ && no:exist == 1 &&
         no:found ==0){
    fitness = no:fitness;
    string = no:string;
    found = no:found;
    exist = no:exist;
  }
}
```

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compiler: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] John Byl. Self-reproduction in small cellular automata. *Physica D,* 34:295-299, 1989.

[3] Hui-Hsien Chou. Self-replicating structures in a cellular automata space. Technical Report CS-TR-3715, Computer Science Department, University of Maryland, College Park, Maryland, 1996.

[4] Hui-Hsien Chou and James A. Reggia. Emergence of self-replicating structures in a cellular automata space. *Physica D*, 110:252-276, 1997.

[5] Hui-Hsien Chou and James A. Reggia. Problem solving during artificial selection of self-replicating loops. *Physica D*, 115:293-312, 1998.

[6] E. F. Cood. *Cellular Automata.* Academic Press, New York, 1968.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing,* 151-158, 1971.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms,* Second Edition. MIT Press/McGraw Hill, 2001

[9] J. Dana Eckart. A cellular automata simulation system, 1998. Available from <http://www.cs.runet.edu/~dana/ca/tutorial.ps> as of 10/27/2001.

[10] J. Dana Eckart. Cellang: language reference manual, 1997. Available from <http://www.cs.runet.edu/~dana/ca/cellang.ps> as of 10/27/2001

[11] Joshua Engel. *Programming for the Java Virtual Machine*. Addison-Wesley, Reading, Massachusetts, 1999.

[12] Gianluigi Folino, Clara Pizzuti and Giandomenico Spezzano. Combining cellular genetic algorithms and local search for solving satisfiability problems. In *Proc. of ICTAI'98 10th IEEE International Conference Tools with Artificial Intelligence,* 192-198, IEEE Computer Society, 1998.

[13] Gianluigi Folino, Clara Pizzuti, and Giandomenico Spezzano. Solving the satisfiability problem by a parallel cellular genetic algorithm. In *Proc. Of the 24th Euromicro Conference*, Vasteras, Sweden, August 1998.

[14] M. Gardner. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American,* 223(4):120-123, 1970.

[15] Jun Gu. Efficient local search for very large-scale satisfiability problems. *Sigart Bulletin,* 3(1):8-12, 1992.

[16] Jun Gu. Global optimization for satisfiability (SAT) problem. In *IEEE Trans. on Data and Knowledge Engineering,* 6(3):361 – 381, 1994.

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

[18] Scott E. Hudson. *Java CUP User's Manual*, 1996. Available from <http://www.mit.edu/afs/sipb/project/java/packages/java_cup> as of 10/02/2001.

[19] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D,* 10:135-144, 1984.

[20] J. A. Moreno, J. G. Santos. A problem-solving environment based on cellular automata. In *Proc. of the 3rd Conference on Cellular Automata for Research and Industry (ACRI 98),* 261-270, Trieste, October 1998.

[21] J. von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Illinois, 1966

[22] Michael J. Quinn. *Parallel Computing: Theory and Practice.* McGraw-Hill Press, New York, 1994.

[23] James A. Reggia, Steven L. Armentrout, Hui-Hsien Chou, and Yun Peng. Simple systems that exhibit self-directed replication. *Science,* 259:1282-1288, Feb. 26, 1993.

[24] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problem. In *Proc. 10th National Conference on Artificial Intelligence (AAAI-92),* 440 – 446, San Jose, CA, July 1992.

[25] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. AAAI-94, 337 – 343*, MIT Press, 1994.

[26] Giandomenico Spezzano and Domenico Talia. CARPET: A programming language for parallel cellular processing. In *Proc. 2nd European School on Parallel Programming Environment (ESPPE 96),* 71 – 74, Alpe d'Huez, April 1996.

[27] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, Massachusetts, 1987.

[28] Darrell Whitley. Cellular genetic algorithms. In *Proc. Fifth Int. Conference on Genetic Algoriths*, 1993.

[29] Darrell Whitley. A genetic algorithm tutorial. Technical Report, Computer Science Department, Colorado State University, Fort Collins, Colorado.

[30] Stephen Wolfram. *Cellular Automata and Complexity.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[31] Thomas Worsch. Programming environments for cellular automata. In *the proceedings of ACRI 96*, 16-18, Milano, October 1996.

[32] Stanislaw Ulam. Random processes and transformations. In *Proc. Int. Congr. Mathem.*, 2:264-275, 1952.