Cellular Automata Implementation of DNA-based Computing

David RENAULT
University of Memphis, Tennessee
Computer Science Department, Dunn Hall

Summer 1999

Introduction

The most recent evolutions in terms of computation power lead to the quest for more and more parallelism in the way we are programming. Apart from the capabilities of programming based on Türing machines, another heuristic has been raised which is supposed to provide a high "built-in" parallelism : Molecular Computing. As a matter of fact, molecules have a high level of interaction with each other, thus inducing an important power of computation, assuming you are able to control it. Molecular Computing aims at harnessing these individual molecules for computational purposes.

Nevertheless, even if Molecular Physics provide numerous means to deal with such molecules, effective work remains in some way unreliable and expensive compared to the standards of programming. The aim of this work is to design a platform, based on cellular automata, which could realistically simulate the chemical reactions happening in a test tube. For this kind of interaction, cellular automata are well adapted, since molecular interactions are governed by local data.

This paper first gives a general overview of the latest developments in the domain of Molecular Computing, and the purposes of such a work, then it describes how to build a cellular automata simulating a DNA test tube, and the fundamental questions raised. The following sections examine *Edna+*, a program, designed on these basis, and the results obtained with our simulations.

Contents

1 Overview of Molecular Computing

The first major work based on Molecular Computing is due to Leonard Adelman [1] in 1994, who designed an experiment to solve an instance of a well known $\mathcal{NP}$-Complete problem, The Travelling SalesMan, as known as the Hamiltonian Path Problem. Creating a way to transform his graph in strands, and linking each edge and each vertice with a special DNA strand, he managed to find the solution to the problem using only a linear amount of *biological steps* or $BEPS$ (thus opposing $MIPS$).

Soon, new proposals were made to design a molecular computer, as in [2] and [7], discussing of the various advantages of such a computer. As a matter of fact, the biological experiment per second rate (or $BEPS$ rate) is far less than the values reached by computers nowadays, by a factor of 10^6, yet over 10^{17} strands can be used in a single liter of water, which leads to a significantly higher power of computation, in theory.

Lila Kari, Gherghe. Paŭn and al. showed different universality results about this new form of computation, proving in [6] different results about the characterization of languages and recursively enumerable languages with H-systems [1] and in [4] that it was possible to simulate the evolution of a Turing machine with the same systems. Other research [3] showed that it is possible to break DES [2] with DNA experiments. These computations remain though in the theoretical domain.

The main results found in Molecular Computing are described in two overviews, [8] and [5], and explore the evolution of this domain of Computer Science since Adelman's experience. Though, we will explain here some fundamentals so as to understand the basics of working with DNA strands.

1.1 Fundamentals on DNA

DNA sequences are in fact single or double stranded (helicoidal) structures composed of four different nucleotides : Adenine A, Guanine G, Thymine T and Cytosine C. These nucleotides pair with one another, following the rules :

$$A = T$$

and

$$G \equiv C$$

... thus forming Watson-Crick pairs. These molecules are compatible by pairs, establishing *hydrogen bonds* between them. There are two shared electrons exchanged for an $A-T$ pair, and three for a $G-C$ pair. Each DNA strand has an orientation, denoted from $5'$ to $3'$, and only strands in opposite directions may hybridize.Notice that in the same case, two DNA strands moving in the same direction won't hybridize.

For example :

$$\begin{array}{ccc} 3' & AAACTGGAG & 5' \\ & + & \\ 5' & TTTGACCTC & 3' \end{array} \Rightarrow \begin{array}{c} AAACTGGAG \\ TTTGACCTC \end{array}$$

Moreover, two strands may hybridize even when the nucleotides are not fully compatible. And a long enough single strand with the ight nucleotides might hbridize with itself. The hybridization rules are intricate and we shall discuss them further later.

[1] Systems which have the abilities of combination of DNA strands

[2] Data Encryption Standard

1.2 Operations performed

The basic operations performed in a test tube, and described as "doable" by molecular chemistry are :

Separate Given a test tube containing DNA strands T, and a string s on $\{A, G, T, C\}$, one may divide the test tube into $+(T, s)$ and $-(T, s)$, the first one containing the strands containing the subsequence s and the second one the remaining strands,

Amplifiy Given a test tube containing DNA strands, Polymerase Chain Reaction (PCR), is able to copy these strands and produce a given number of copies in the same tube. This operation is the less reliable of the ones described here ,

Length By Gel Electrophoresis, this operation is able to sort the different strands in a tube by length,

Detect Detect if a tube contains at least one strand or not,

Merge Merge two test tubes,

Some of these operations are not *reliable*. The *separation*, which is the most used operation in Molecular Computing leads to high experimental error rates, but these rates might be curbed by repeating the operation. The *amplify* operation tends to be error-prone, since errors may happen during the copy of the strands. The *detect* operation also gives some problems to get safe.

1.3 Adelman's experiment : a simple example

A simple way to understand how Molecular Computing works at a practical level is to describe Adelman's experience on the Travelling Salesman Problem, as shown in figure 1.

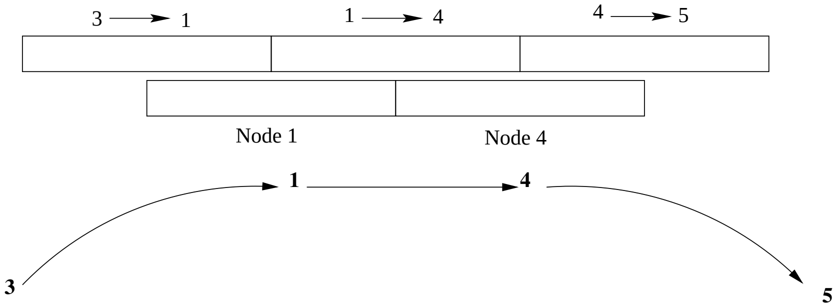


Figure 1: Adelman's experience

The starting and ending nodes have special encodings, so as to differentiate them from the others. Then each node u is described by a special 20-mer (DNA single strand of length 20), and each edge $(u \longrightarrow v)$ is the concatenation of two 10-mers, the first one fully compatible with the last 10 nucleotides of u and the second one fully compatible with the first ten nucleotides of v. Thus, these strands may hybridize as shown in figure 1, and the hybrid obtained is composed of :

- a list of nodes for one strand
- a list of links between these nodes for the other strand

... thus symbolizing a path in the graph.

In this experiment, Adelman placed all the strands in a test tube, at given concentrations, and let them hybridize. We then separate the test tube to find all the double strands containing the starting node, the ending node, and all the nodes of the graph. Then we sort the strands by length, so as to find out if there's one whose length is equivalent to the number of nodes in the graph. If such a strand exist and still appears in the tube at experimentable concentrations, then the experience proves true : the Hamiltonian path exists and is encoded by this strand.

The main asset of such an experience, for a graph $G = (V, E)$, where $n = |E|$ and $l = |V|$, is to use only $l + \alpha$ BEPS (α is a constant), which runs in linear time, in opposition to "conventional" algorithms which are still unable to perform such a task in polynomial time. Detractors argue that the number of strands used in this algorithm is really important, and could grow up to $l!$ depending on the graph.

2 Creating a cellular automaton

The aim of this report is to create a cellular automaton which simulates two main aspects of DNA strands :

- Motion aspect : The strands shall be able to move freely in the automata, with random movements as if driven by Brownian motion.
- Hybridization aspect : The strands shall be able to interact with each other, following the chemical rules of hybridization as closely as possible.

We will give better definitions of these two aspects in the next section. The main question will in fact be : what properties of these aspects is the cellular automaton capable of preserving ?

2.1 Cayley Graphs

Let $G = (V, E)$ be a graph where V is a set of vertices and $E \subset (V \times V)$ is the set of edges. The *neighborhood* of a given vertex i is the set $\Gamma(i)$ of all vertices j adjacent to i. The cardinality $|\Gamma(i)|$ of this neighborhood is called the *degree of i*. A graph is said to be *connected* if any two of its vertices can be connected by a path. A graph is *regular* if all vertices have the same degree.

To each vertice of the graph, we are going to give a color, defined as follow : let X be a set of *colors*, which is supposed to be symmetrized, which means :

- each color A has an inverse A^{-1} in X : $\forall A \in X, \exists A^{-1} \in X$
- this inverse is defined so that : $\forall A \in X, (A^{-1})^{-1} = A$

An *edge-colored* graph consists of a graph in which all vertices have a color in X. We are going to work with regular edge-colored graphs.

Definition 1 (Fundamental Neighborhood). *Given a regular edge-colored graph G, a* fundamental neighborhood *N_G of Γ is a finite edge-colored oriented graph, which has a distinguished origin λ.*

A given graph may have many different fundamental neighborhoods. However, a regular graph is said to have a *strongly regular coloring* if and only if it possesses a fundamental neighborhood N_G and :

- For all vertices $i \in G$, the neighborhood $\Gamma(i)$ is isomorphic to N_G by an isomorphism f_i
- This isomorphism maps i to λ : $f_i(i) = \lambda$
- f_i preserves the colors of the edges
- The color of the arc (i, j) under f_i is the inverse of the color of the arc (j, i) under f_j.

Theorem 1 (Cayley Graph). *A connected graph has a strongly regular coloring if and only if it is the* Cayley graph *of a group*

Whereas this definition is very obscure, we can see a Cayley Graph as something really simple. Concretely, a Cayley graph is a connected regular graph which has special properties. The neighborhood can be seen as sown in figure 2, which displays a simple Cayley Graph (in fact a $\mathbb{Z} \times \mathbb{Z}$ infinite grid).

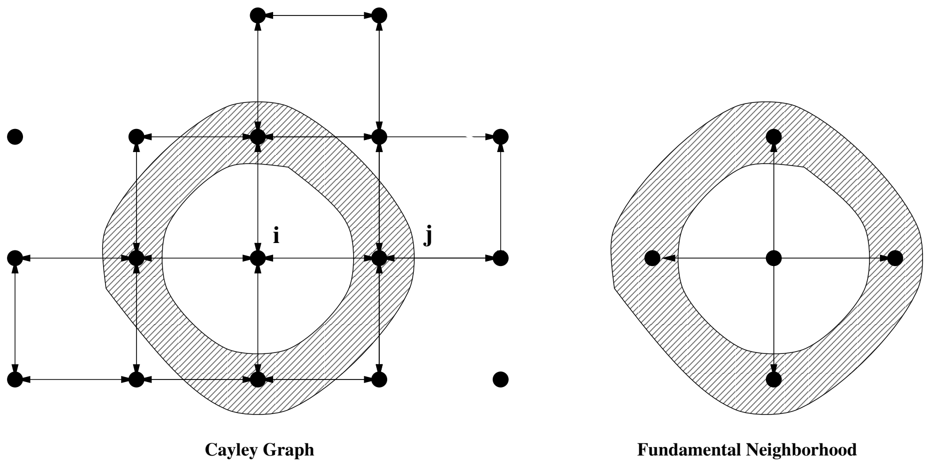


Figure 2: Neighborhood of a graph

Since every vertex has the same number of neighbors, we can see the neighbors of i as $\Gamma_1(i), \ldots \Gamma_n(i)$ where $n = |\Gamma|$. Since we only need the name of the vertex and the number of the generator to define an edge, we can call these neighbors $\Gamma_1, \ldots \Gamma_n$. These neighbors are elements of the set of colors X, all distinct since we have a Cayley Graph (the isomorphism gives the certainty that they are all distinct). Notice that they are completely independent from i.

When you want to refer to a neighbor of i, you can use " $i\Gamma_k$ " where Γ_k leads you to the right neighbor. You can then move into the graph using this description of the edges. The main point is that each time you move from i to j using Γ_k, you can move back from j to i using $(\Gamma_k)^{-1}$. And $(\Gamma_k)^{-1}$ belongs to the neighborhood of j because of the last law of the isomorphism.

From every node, you can reach any other node ith a path constituted of a list of colors, or *generators* :

$$i \longrightarrow j : \Gamma_{k_1}\Gamma_{k_2} \ldots \Gamma_{k_n}$$

Finally, you can define group relations in the set of colors X (the two sets X and N_G might be confounded), for example *commutativity* :

$$\forall m, n \in \Gamma^2, \Gamma_m \Gamma_n = \Gamma_n \Gamma_m$$

The example of the figure 2 is a two-dimensional grid, $\mathbb{Z} \times \mathbb{Z}$, a very simple infinite Cayley Graph. Each vertice or node has 4 neighbors. The fundamental neighborhood is drawn on the figure. we can call the different colors or direction in the following way :

$$North, South, East, West$$

... or N, S, E and W for simplification. Each direction has an opposite direction, that is to say $N^{-1} = S$ and $E^{-1} = W$. Moreover, the graph is commutative. This graph can be folded in a two-dimensional torus, $\frac{\mathbb{Z}}{n\mathbb{Z}} \times \frac{\mathbb{Z}}{m\mathbb{Z}}$.

2.2 Cellular Automata

For the purposes of our work, we are going to work with a cellular automaton, described as follows :

Definition 2 (Cellular Automata). *A* cellular space *is a pair* (Γ, Q) *consisting of a regular graph* Γ *and a finite-state set* Q *containing a special state* 0*, the* quiescent state. *A* cellular automaton *is a triple* $\langle \Gamma, N, M \rangle$ *consisting of a cellular space* (Γ, Q)*, a finite set* N *of vertices of* Γ *(the neighborhood) and a finite-state machine* M *with input alphabet* $Q^d := Q \times \cdots \times Q$ *(d times) and local transformation function :*

$$\begin{array}{rcl} \delta : Q \times Q^d & \rightarrow & Q \\ (x_i, x_{i+i_1}, \ldots, x_{i+i_d}) & \rightarrow & \delta(x_i, x_{i+i_1}, \ldots, x_{i+i_d}) \end{array}$$

... *where* $d := |N|$ *is the number of cells in the neighborhood N.*

The cellular automata is an appropriate notion for simulating DNA Computing, since it works with a *local rule*. And the evolution of strands in a test tube relies more on the interaction of local data than with the global strands. We can decide to move the strands molecule by molecule, and not strand by strand, and the “strand” object is no longer relevant, since we operate with each cell. We can have hybridizations from a strand with itself, or with many other strands if the rules permit.

Moreover, the couple constituted by a *cellular automaton* and a *Cayley graph* offers great advantages when working with spatial directions. And these spatial directions influence directly the evolution of the strands : two strands are more likely to hybridize if they are moving in opposite directions. So we must be able to give a meaning to the word “opposite” into our test tube.

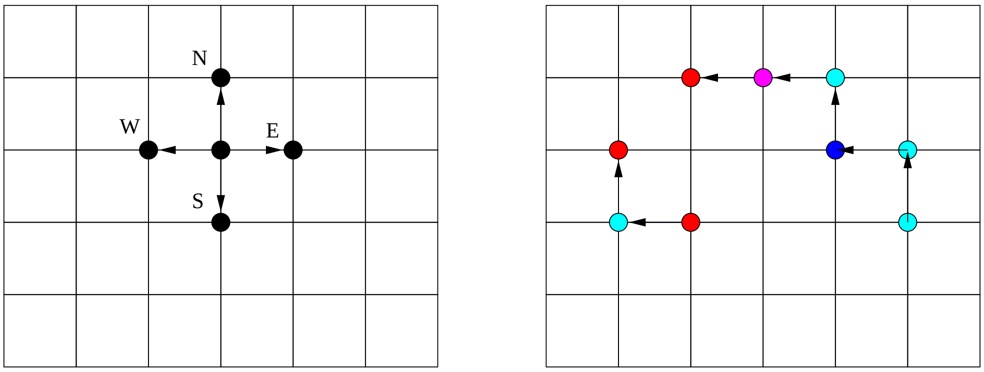
The major local rule of this automaton will be to work in two steps : first, we *move* the nucleotides, then we try to *hybridize* them. We will discuss the two phases separately. The first one is relatively easy because it doesn't change the contents of the tube, whereas the second phase is much more difficult, because we must reflect what is likely to happen into the tube.

3 Simulating DNA reactions in a test tube

This section will describe the different elements of a virtual test tube : its components (the strands, the cells ...) and its operations (motion and hybridization). These elemnts are part of the *Edna+* program, and constitute the `Automata`, `Cell` and `Strand` classes. Another class, the `NodeTable` class contain the spatial data involving the graph.

3.1 The components

Our virtual test tube is a Cayley Graph, as seen on figure 3. A vertice in this graph is called a cell. The links from one cell to another are virtual "directions", that an eventual nucleotide could choose if deciding to move from his current cell. The cells at distance less than 1 form the *neighborhood* of a cell.



A cell and its neighbours

Two strands in the tube

Figure 3: Example of a test tube

Each *cell* possesses the following elements :

- A given set of directions, and a function telling whether two directions are opposite or not.
- A nucleotide, which can be any of the following :

  `ADENINE A` defined as 1. `THYMINE T` defined as 3.
  `GUANINE G` defined as 7. `CYTOSINE C` defined as 15.
  `C_BOND b` defined as 31. `NONE` is defined as 0
  ,

  ... or any combination of two of these. If two non-null nucleotides are combined, they are called *hybrids*, otherwise they are *single*.
- A hybridization level, which tells if the nucleotides in the cell are single (`NONE`), sharing two hydrogne bonds (`WEAK` for an $A-T$ pair), or even three (`STRONG` for a $G-C$ pair), or if these hybrids are forced (`FORCED`, for all other kind of pair),
- A ligation direction, to tell if the cell is ligated into a strand, and if so, the cell it is ligated to,
- Other data to take into account the *temperature* and *salinity* of the cell, so as to perform chemical reactions accurately,

The *strands* are linked lists of adjacent nucleotides. The first one is the *tail*, and the last one, which is ligated to none, is the *header*. Thus the tail decides of the direction. Each strand contains two important things for our purpose :

- A *queued nucleotide*, the nucleotide due to move at the next step of the automaton,
- A *queud c-bond*, the c-bond preceding the former nucleotide,

The *c-bonds* are artifacts of the reaction, used to move and keep the strands together into the tube, and have no chemical activity. For more explanation about these special nucleotides, see the motion algorithm.

3.2 Motion of the strands

During each run step of the automaton, it is possible for one nucleotide of each strand present in the automata to move. The strand moves in a specific order, from tail to header, keeping in mind the next nucleotide to move.

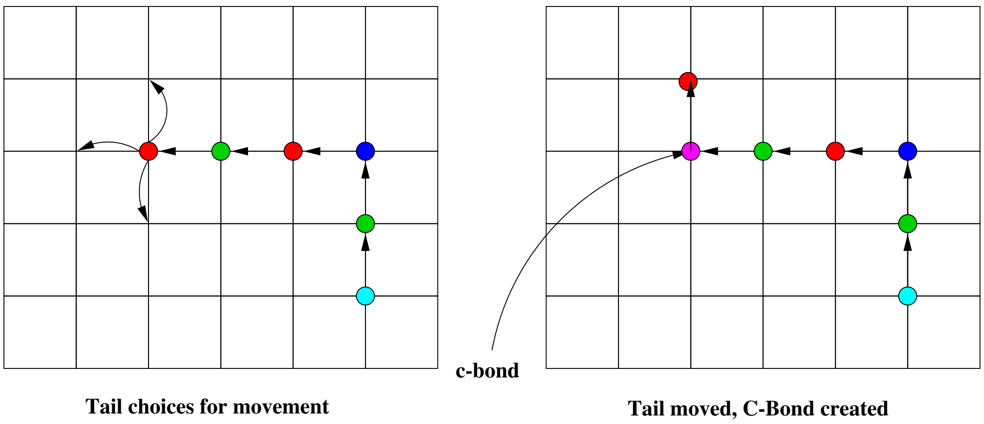


Figure 4: Tail moving towards

The tail determines the direction of movement for the strand, as shown on figure 4. When the tail of a strand moves, it can choose between any direction possible, given that the corresponding cell doesn't already contains a nucleotide, or decide to remain at the same place (including the case where there's no possible direction). If the tail is also a header (ie the strand consists of only one nucleotide) when it moves, then no c-bond is created in the former place of the tail and the tail is requeued to move. However, if the strand is longer than one nucleotide, several actions must be performed. First, a c-bond is created in the former position of the tail. Then the queud cell then points to the next nucleotide in the strand. Finally, the tail is ligated to the c-bond to preserve the integrity of the strand. The queud c-bond points to the just created c-bond.

Thereafter the other nucleotides in the strand, when due to move, are forced to move to the cell they are ligated to, which is the queud c-bond, and a c-bond is created at their former place. The queud cell then points to the next nucleotide of the strand. If the nucleotide is a tail, then no c-bond is created and the header of the cell becomes the next nucleotide to move. This movement is illustrated in figure 5.

This algorithm is simple and uses little memory. It moves a complete strand in $\mathcal{O}(l)$ steps, where l is the length of this nucleotide. Even if the movement of a long strand is

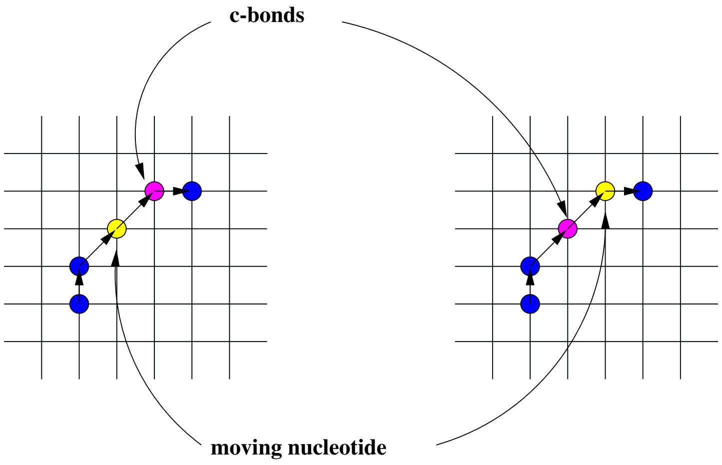


Figure 5: Nucleotide moving towards

slow, the *motion phase* is fast, in $\mathcal{O}(N_{Strands})$ each step. But this movement is a flaw when dealing with very big strands, bcause of their slow global motion (one global move each l steps).

3.3 Hybridization Phase

During each run step of the automata, every strand attempts to hybridize with another strand. Strand hybridization occurs when the three following conditions are met :

1. Compatibility Rule : Two complementary nucleotides (A and T, or C and G) are adjacent (neighbors),
2. Opposite Rule : The nucleotides are moving in opposite directions or either nucleotide is a strand header,
3. The function of hybridization of the two corresponding strands returns a value below a given threshold, telling whether the chemical reaction between the strands is likely to happen and how the strands concretely hybridize into the tube.

For reasons of simplicity, we do not allow strands to hybridize with themselves, hence some patterns of hybridization can not to happen in our model. (cf figure 6)

The two first rules capture the chemical reactions involving two strands. Though, the program doesn't take into account the direction of the strand (form 5' to 3'). The third rule is the most intricate one : it analyzes both strands and finds the "best" way to hybridize them, this way coded by a potential. If the potential is low enough, the two strands may hybridize, otherwise, they keep moving into the tube.

At each step of the automaton, each cell in each strand has the possibility to hybridize with one of its neighbors. If the conditions are met, we compress the two strands by removing all c-bonds due to the movement, then bring them out of the tube, build the new hybridized strand and place it back. The final hybridized strand may hybridize again if it still has single nucleotides inside, and if the hybridization function allows this reaction. (see figure 7)

For some reasons, we can't hybridize "in the tube". The path constituted by the strands may have strange shapes, and in any case, there's a low probability for us not to

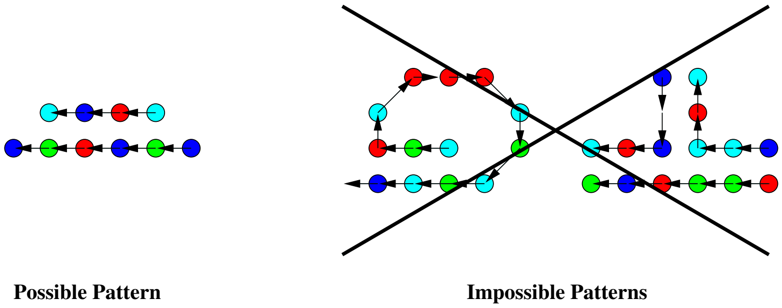


Figure 6: How the strands hybridize - Patterns

be able to place the strand back at its original place (say the place of the bigger strand), if only because there might not be enough place left at this location. So the algorithm looks for a random place inside the tube. Our heuristic assumes that the tube is not filled with strands (the number of nucleotides divided by the number of cells in the space is below 50%), so finding a place for the strand is “easy”.

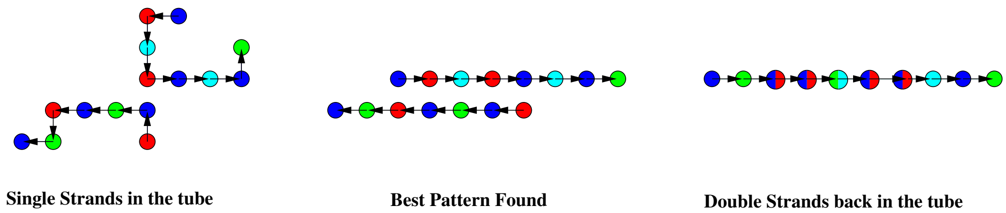


Figure 7: How the strands hybridize - Process

This algorithm for checking possible hybridizations is the slowest of the automata, yet the simplest for memory. Its complexity is in $\mathcal{O}(\sum_{s \in Strands} length(s))$, which takes time. Nevertheless, it fits our purposes. The algorithm of hybridization is in $\mathcal{O}(l_{Big} + l_{Small})$, where l_{Big} is the length of the bigger strand and l_{Small} for the smaller strand.

The problem of the hybridization function is critical, because this function decides “how” to hybridize two strands, and prevents some undesirable hybrids, for example hybrids where only one nucleotide hybridize. The two functions implemented into *Edna+* are the following :

- The *h-distance* : given two different strands, we compute all possible shifts between the two strands and find the ones where the number of matches between the strands is maximal (A–T and G–C pairs). Then we choose the shift which minimizes the number of incompatible strands. For example :

Shift															*Matches*	*Best*
				A	*A*	*A*	*A*	*T*	*T*	*T*	*T*					
−3	*A*	*T*	*C*	*G*											0	
−2		*A*	*T*	*C*	*G*										0	
−1			*A*	*T*	*C*	*G*									1	⋆
0				*A*	*T*	*C*	*G*								1	⋆
1					*A*	*T*	*C*	*G*							1	⋆
2						*A*	*T*	*C*	*G*						1	⋆
3							*A*	*T*	*C*	*G*					0	
4								*A*	*T*	*C*	*G*				1	⋆
5									*A*	*T*	*C*	*G*			1	⋆
6										*A*	*T*	*C*	*G*		1	⋆
7											*A*	*T*	*C*	*G*	1	✓

- The *free energy* : given two different strands, we compute all possible shifts that allow only compatible strands to hybridize. Then, considering the part of each strand that is supposed to hybridize, we compute the *free energy* of this sub-strand. Two following nucleotides in a strand can be linked to a Free Energy value. We sum these values on the sub-strand to obtain the total *free energy* of the hybrids. For example :

Shift											*Energy*
			A	*A*	*A*	*A*	*T*	*T*	*T*	*T*	
−2	*T*	*T*	*T*	*T*							E_{AA}
−1		*T*	*T*	*T*	*T*						$E_{AA}+E_{AA}$
0			*T*	*T*	*T*	*T*					$E_{AA}+E_{AA}+E_{AA}$

The free energy of the nucleotides $E_{n_1n_2}$ is a numerical value given by experiments on real DNA strands. Obviously, these values are the same whether you choose the bigger or the smaller strands of the two. The more nucleotides hybridize, the more energy is raised, then longer hybrids are more likely than shorter ones. This function only allows *perfect* hybrids.

4 *Edna+* : The Program

This section will describe the program following the previous model and algorithms. This program is called *Edna+*, and gathers the previous elements into one single software. This program was designed in *Visual C++*, and uses some of the *MFC* classes. The main core of the program (the main window, and the backbone of the classes used by *Windows*) has been generated automatically by the *Microsoft* Visual C++ compiler.

This version of *Edna+* is not entirely safe, though the most important problems with the interface have been solved now. Yet some cases might not have been tested. This program has been implemented in collaboration with a student of the Universit of Memphis, Evan DRUMWRIGHT, who wrote the major part of the `Automata class`, under the direction of professor Max H. Garzon.

4.1 Class Description

This program is written with object oriented rules, so each class is separated from the others, and has its own functions. We can divide this set of classes into two parts.

First, the computing parts, which consist of the test tube, the nucleotides, and the table of strands. Into these classes are the main data and the functions simulating the cellular automata :

- The `NodeTable class`, and its subclass the `Node class`, handle the spatial informations about the test tube. Each Node has a table of neighbors, and the main class contains the data about the Cayley Graph.
- The `Automata class`, and its subclass `Cell class`, contain the main functions performed by the automata, motion and hybridization, with all the cellspace, containing the data about the nucleotides and the cells, and the hybridization functions
- The `Strand class` handes the table of strands, their placement and removal and other basic operations upon strands.
- The `CEdnaPlusView`, `CEdnaPlusDoc`, `CMainFrame` and `CEdnaPlus classes` are generated autmatically by the compiler when creating a new *Windows* application. These classes deal with the input/output, the drawing of the test tube, and the operations with the user.
- The `CEdnaPlus class` is the main class of the application, and contains the main objects as the Automata and the Nodetable.

For further detail on the classes in *Edna+*, see Appendix A and B.

4.2 Main functions

Here is a quick description of the main functions of *Edna+*, acting as an User Manual. We describe each element present in the menu of the application.

① The FILE Menu

The different options in the FILE menu allow you to handle test tubes and sets of strands. It also contains a list of the most recent files used.

The NEW option opens a dialog which allows you to create a new test tube. The graph files have "`.gr`" extensions. There are four choices of possible graphs, yet the user is able to create his own graphs (see Appendix).

- 2D grid with four or eight neighbors,
- 3D grid with six neighbors or 26 neighbors,

In each case the neighbors correspond to the cardinal directions (North, South ...) to which one can add the different quadrants (North-East ...) to enhance the graph. This option doesn't open the graph created, though.

The OPEN and CLOSE options are the normal functions. When opening another graph while one is already loaded, the previous one is closed before loading the new one.

The LOAD STRANDS... and SAVE STRANDS... options deal with set of strands (files with `.set` extensions). *Edna+* allow the user to save the current set of strands, and to load it afterwards. When saving, he is prompted whether he wants to save the locations of the strands or not. When saving the locations, one must use the same test tube when reloading the set of strands. In the other case, the strands are put back randomly into the tube, which allows you the changing of test tube. If there's not enough place left, the strands won't be placed at all. The loading functions adds the strands in the set into the curent tube, without removing the ancient strands.

② The SETTINGS Menu

This menu is used to set the general options of *Edna+*, and to insert the strands into the test tube. Some options may not be available if no test tube is loaded before.

The PLACE STRAND option opens a dialog box which allow you to insert a strand (it can be accessed by hitting 'INS'). You may choose to put the strand randomly, or to give a location where to put the header. The location is a number corresponding to a node in the tube. The remaining of the strand is placed at random into the tube.

The RANDOM STRAND option places a strand of length varying between 2 and 12 into the tube, with random nucleotides. The CLEAR SPACE option removes all nucleotides from the tube.

The following functions customize the general options of the application :

- SET NUMBER OF STEPS allows you to choose the number of steps of the automata each time you run the WALK ONE STEP function. The CUSTOMIZE NUMBER OF STEPS function allows you to pick any number from 1 to 500.
- CUSTOMIZE MAGNIFICATION allows you to enlarge or reduce the size of the automata on the screen. The *Auto Mag* button on the dialog box computes a best value for the screen and the test tube. When you raise the value, the tube becomes smaller, and when you lower it, you enlarge the tube.
- SET SIZE OF NUCLEOTIDES sets the size of the nucleotides on the screen, from Small to Large. Small nucleotides make it difficult to distinct the hybrids when twwo nucleotides hybridize.
- SET HYBRIDIZATION FUNCTION is self-explanatory. The TEST HYBRIDATIONS... opens a dialog allowing to virtually pick up two strands inrto the tube and try to hybridize them. This action has no effect on the tube itself.
- SET THRESHOLDS... allows the user to set the different thresholds for each hybridization function, and to enable or disable two rules of hybridization : the COMPATIBILITY RULE and the OPPOSITE RULE.
- NOTIFY OPTIONS allows the user to be warned when a fully hybridized strand of a given length appears into the tube. If running, the application will stop and a message box will appear.

③ The EDIT Menu

The SELECT STRAND... option allows the user to open a list of the strands into the tube, and to select and/or delete one strand. The length of each strand is shown on the right of the list. The two following options in the menu allow to DELETE a strand or toggle the HIGHLIGHT option for this strand . These two options will only function if a strand had previously been selected.

④ The RUN Menu

The WALK ONE STEP function launches the automata (accessed by 'SPACE'), for a given number of steps. Each step consists of one hybridization phase, followd by one motion phase. The number of steps might be customed in the SETTINGS menu. The CONTINUOUS RUN option launches the automata (accessed with 'ENTER') continuously, which disables some options in the other menus. To stop it, you must use this option again.

⑤ The View and Help Menu

The two options in the View menu allow you to display the graph or the nucleotides. When the nucleotides are not displayed though, the graph won't be displayed, too. In the help menu you can find a reminder of all the characters used in *Edna+* to refer to the nucleotides. The Help window might be called with 'F1', and is independent from the main window.

5 Experiments

The experiments performed with *Edna+* derive from Adleman's experiment on the Hamiltonian Path problem. We have used different kinds of graphs, and different kinds of encodings to test our program. The experiments show that on large tests *Edna+* does not behave very well, while it can find accurately the solutions with smaller problems.

Edna+ can theoretically sustain a very large Cayley Graph, but big graphs use a huge amount of memory. Although, the algorithms do just depend on the number of strands, and are independent of the size of the graph. Some very big experiments are then "possible" to handle within *Edna+*, given the initial amount of memory to load the graph.

5.1 Techniques Used

The tests realized with our software were instances of the Hamiltonian Path problem. Encodings of the vertices and the edges are placed into a large enough test tube, and then *Edna+* simulates the volution of these strands into the tube. This tests were performed on two different computer in parallel (same test launched on two computers) on one hour each. The computers are PCs 100 MHz with 32Mb RAM. here is a small description of these experiments.

First, the *size of the graph* must be large enough to contain the strands, but small enough to have an average load over 30%. Thus, many "encounters" are possible, leading to more hybridizations. The number of neighbors is critical to avoid deadlocks. Even with eight neighbors, many strands will turn around themselves and finally stop moving and hybridizing. Moreover, these obstacles in the tube increase the chances for other strands to deadlock when hitting them.

The test realized with Adleman's experiments showed that *Edna+* has the ability to handle a large number of nucleotides pretty fast, even if this number seems small compared to "real" experiments. During the tests, the fully hybridized strands were regularly deleted from the test tube, and replaced by other strands from the initial set. The experiments show that if the graph contains many possible paths, we must add more strands encoding the vertices than the edges, because the vertices disappear pretty quickly, while there still remains unused edges.

The experiments show that *Edna+* is working pretty well on all the experiments, even though it doesn't find the solution, many paths are found. The fact that we are working with a probabilistic algorithm makes it difficult to get the solutions of a rather large instance of the problem (probability that the path is the right one below 1%). Though, it simulates pretty accurately the motion and hybridization of the molecules into the test tube. Further experiments in *real* test tubes shall prove the accuracy of the calculation.

5.2 Table of results

This table shows the results obtained with *Edna+*. The two first columns show the number of edges and vertices in the graph tested, and the third value the set of nucleotides uses. The final column tells whether the experiment succeeded or not.

Vertices	Edges	Set	Success
5	4	①	✓
5	8	②	✓
10	9	③	✓
10	13	④	✓
10	17	⑤	Not Found

The sets ① and ② are sets designed and used in real experiments in laboratory. The other sets (namely ③, ④ and ⑤) are original and designed to serve the purposes of *Edna+*.

The first experiments found accurately the Hamiltonian Path, taking more time and more strands when the graph gets bigger. The last experiment wasn't able to find the Hamiltonian path, though it found an important number of paths of length 6, 7 and 8. This experiment had been going for two hours before to stop.

6 Conclusion

Edna+ was realized to simulate accurately the chemical reactions happening in a test tube containing DNA strands. The motion and hybridization of DNA strands have been rendered as accurately as possible, but our main purpose was to keep it simple. The results obtained must now be compared to *real* experiments in laboratory. Though, *Edna+* proved able to manage strands during the tests.

Our main purpose, beyond simulating the DNA strands, was to keep the software as simple as possible, so that further extensions might easily be implemented. Our program fulfills these purposes, and is able to manage small experiments conveniently. It offers another alternative to real DNA strands, and only waits to be tested further.

Files for *Edna+*

The executable file of *Edna+*, all *C++* code and the *Visual C++* contents pof the application might be found in the following directory :

`/home/drenault/fich/Stage99/Exe/`

This report and the pictures it contains might be found at :

`/home/drenault/fich/Stage99/`
`/home/drenault/fich/Stage99/Pics/`

The executable might be launched on a Windows platform, with the pathname :

`/home/drenault/fich/Stage99/Exe/EdnaPlus.exe`

Acknowledgments

This software was developed in collaboration with Evan Drumwright, with which I have been working for one month to set up the backbone of *Edna+*. This program was built from a previous work from Lijun Liu, named *Edna*. All of these were supervised by Dr. Max Garzon from the University of Memphis. Thanks them all for their help.

References

[1] Adleman, L. M. Molecular computation of solutions to combinatorial problems. In *Science,* 226 pp. 1021–1024 (1994).

[2] Adleman, L. M. On constructing a molecular computer. In *DNA Computers,* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol.27, American mathematical Society (1996).

[3] Boneh, D., Dunworth, C., and Lipton, R. J. Breaking des using a molecular computer. In *Technical Report from the University of Princeton, Department of Computer Science* (1999).

[4] Csuhaj-Varjú, E., Freund, R., Kari, L., and Paum, G. Dna computing based on splicing : Universality results.

[5] Garzon, M., and Deaton, R. Molecular computing and programming. In *IEEE Trans. on Evolutionary Computation, in press.* (1999).

[6] Kari, L., Paum, G., Rozenberg, G., Salomaa, A., and Yu, S. Dna computing, matching systems, and universality. In *Turku Center for Computer Science Technical Report No 49* (1996).

[7] Konrad, M. On designs principle for a molecular computer. In *Comm. of the Ass. Comp. Mach. CACM, 28:5, pp. 464–480* (1985).

[8] Pisanti, N. A survey on dna computing. In *Università di Pisa, Technical Report : TR-97-07* (1997).

Appendix A - Automata Class

The automata class represents a space made up of a set number of atomic units (Cells) and strands of nucleotides (Strands). An automata object is constructed with a pointer to the Nodetable (see last section). The object uses the nodetable to determine how many cells are needed, the number of neighbors per cell, which cells are neighboring, and which directions are opposing (ie north and south).

The automata class contains the following data :

***cellspace :** this is a pointer to the array of Cells; once the dynamic space is allocated cellspace[i] is the (i+1)th cell in the array, (ie cellspace[0] is the 1st cell),

***nodetable_ptr :** pointer to the nodetable; necessary for graph information,

***first_strand :** pointer to the first strand in the cellspace (ie pointer to the beginning of the strand list),

automata_strands : number of strands currently present in the automata,

num_nucleos_moved : number of nucleos moved in the present time step; used when passing information back to visual interface,

num_cbonds_created : number of cbonds created in the present time step; used when passing information back to visual interface,

It also manages the following data structures :

The Motion / Hybridization History : The history is used to pass information about changes in the automaton to the program that encapsulates it (in this case the graphical interface). Without the history, the encapsulating program would be extremely slow at posting changes (unless changes were only posted at distant time intervals). The automaton passes motion / hybridization changes to the UpdateHistory function, which updates the appropriate array.

The history consists of three arrays, nucleo_movefrom_array[], nucleo_moveto_array[], and cbond_create_array[]. The size of each of the first two arrays is the Automata's space size, while the cbond_create_array[] is half of the space size (see the Automata constructor). nucleo_movefrom_array[] and nucleo_moveto_array[] are used to determine whether a nucleotide is moved, deleted, or needs to be redrawn (ie the nucleotide has become a hybrid). If a nucleotide moves from a position "to" NULL_ADDR, then that nucleotide has been removed in the automata. If a nucleotide moves from a position to the same position, then that nucleotide has changed type (ie adenine to adenine/thymine).

	nucleo movefrom	*nucleo moveto*	*cbond create*
header nucleotide moves from 0 to 1	0	1	–
non-header nucleotide moves from 0 to 1	0	1	0
c-bond is removed from 0	0	`NULL_ADDR`	–
strand is hybridized at 0,1,2,3	0	0	–
	1	1	–
	2	2	–
	3	3	–

The Strand List : The strand list is a linked list that begins at Automata::first_strand. first_strand points to the first strand in the list. Each strand contains a variable, Strand::next_strand, that points to the next strand in the list. The last strand in the list's next_strand variable points to a null pointer. Note that the next strand in the list can be set or retrieved using Strand::GetNextStrand() and Strand::SetNextStrand().

The Automata class contains the following functions :

- Run Step function :

  move() : handles all strand movement for a time step,

  move_compress_strand() : compresses an excess cbond out of a strand for a time step,

  hybridize() : attempts to hybridize all strands in the automata ,

  UpdateHistory() : updates the movement / hybridization history for the automaton,

- Hybridization functions :

  hybridize_cell() : attempts to hybridize a nucleotide ,

  calc_best_shift() : calculates the best shift and the number of matches in that shift for two strands; returns the maximum number of matches for that shift, and the shift required in ¡shift¿,

  calc_hdistance() : calculates the h-distance and the shift required to obtain that h-distance; returns the h-distance and sets the shift and inverse appropriately ,

  hybridize_compress_strand() : removes all cbonds from a strand in a single time step,

  hybridize_strands() : performs the actual hybridization work (making one strand the hybrid, deleting the other, etc) ,

- Miscelleanous :

  ExistsInArray() : exclusively for use of PlaceStrand(); determines whether we are trying to allocate a new strand over itself; returns true if the value exists in the array, false otherwise ,

  GetNeighbor() : gets the n'th neighbor of a cell (n = 1 .. neighborhood size); exits if 0'th neighbor specified ,

- Strand operaions functions (private) :

  delete_strand() : removes a strand from the strand list; does not remove the nucleotides; returns a Strand pointer to the next strand in the list ,

  strand_len() : returns the length of a strand ,

  StrandNucleos() : returns all of the nucleotides of a strand in nucleos[],

IsStrand() : verifies that a strand pointer exists in the strand list, currently used only by hybridize(); returns true if in the strand list, false otherwise,

find_strand() : finds the strand that contains a cell; returns a Strand pointer ,

- Public Functions :

runstep() : executes a run step,

clear_space() : deletes all strands, nucleotides from the automaton,

PlaceStrandRand() : randomly places a strand in the automaton, positions returned in locations[]; returns false on failure,

PlaceStrand() : given a starting location, randomly places the rest of a strand in the automaton, positions returned in locations[]; returns false on failure,

PlaceStrandExact() : places a strand in the locations specified by locations[], tail will be at locations[0]; returns false on failure,

remove_strand() : removes a strand given an index (0.. [number of strands -1]),

GetStrandLen() : returns the length of a strand given a strand index (see above); exits (-1) if invalid strand index given,

StrandLoc() : returns the location of a strand given a strand index (see above) in locations[]; exits (-1) if invalid strand index given,

GetNumNucleosMoved() : returns the number of nucleotides moved in the last run step,

GetNumCbondsCreated() : returns the number of cbonds created in the last run step,

GetNucleotide() : returns the nucleotide (ie adenine, thymine) in a given cell,

GetHybrid() : returns the hybrid type of a give cell,

GetNucleotides() : returns the nucleotides and hybrids of a strand given an index ,

GetNumStrands() : returns the number of strands in the automaton,

GetVisualUpdate() : returns the contents of the history arrays,

IsHeader() : returns true if a cell is a header, false otherwise ,

Appendix B - The View, Doc and MainFrame classes

The `CEdnaPlusView` , `CEdnaPlusdoc` and `MainFrame` classes are the main class of the program, handling both the automata and the nodetable. They take care of loading the files, displaying the graph and the nucleotides, and handling the interface with the user.

These classes are the backbone of the program. This report won't describe all the classes defined by *Visual C++* to create a new application, but the main functions that are used in *Edna+* are described into this class. It contains six side classes, the dialogs used to run the program :

- `CAboutDlg` : Class automatically created by *Visual C++*, contains the about box, the icon of the program, and the acknowledgements,
- `CCreateGraph` : Dialog used for the creation of Graph files. A further description of the features can be found in Appendix "Loading a Cayley Graph",

- `CInsertDialog` : Dialog allowing to insert a new strand. Random location or Precise location allowed,
- `CSetMagDlg` : Dialog allowing to set the magnification factor,
- `CSetStepDlg` : Dialog allowing to set the number of steps,
- `CStrandList` : Dialog allowing o display the list of the strands in solution,

The program contains a pointer to an Automata object, *automata*, and a pointer to a NodeTable object, *nodetable*. These pointers are static and thus can not be used outside the View Class. The other data contained in the view are :

colors : list of the 16 colors used in *Edna+*,

Continuous : boolean variable decideng whether the automata is running continuously or not,

Counter : counter used to time events ,

FileChosen : name of the file loaded, empty when there's no file loaded. This variable il used to know if the nodetable and automata pointers are loaded or not,

Height, Width : size of the screen,

Magnification : level of magnification of the screen, chosen automatically by the program, might be changed manually,

NbSteps : number of steps chosen to run the program,

Pen, pens, pen2 : pens used to display the nodes. Pen is the black pen for the vertices, pens ae the big pens (mono nucleotide), and pen2 are the small pens(bi-nucleotide),

StrIndex : number of the current strand,

tight, tight2 : variables used only for display,

ViewDis, ViewGr : booleans variable decideing whether the user has chosento see the complete display, only the nucleotides or nothing at all. Note that when you choose `ViewDis = FALSE`, then nothing happen to `ViewGr`,

The following functions are inserted into these classes :

- The Interface functions :

  OnDestroy : deallocate the variables *automata* and *nodetable*, then exits the application,

  OnDraw : draw the graph and the nucleotides, each time the window changes,

  OnFileClose : closes the current file and deallocate the pointers,

  OnFileNew : loads the CCreateGraph dialog and creates a new file, but doesn't load it on the screen,

  OnFileOpen : opens a file and loads it,

  OnTimer : clock used to run the automata, and to highlight the strands ,

- The functions running the automata :

OnRunContinuousRun : runs continuously the automata, a second call will stop tha automata. The "SPACE" shortcut has been assigned to this function,

OnRunWalkOneStep : runs *NbSteps* of the automata. The "ENTER" shortcut has been assigned to this function,

- The Settings functions :

OnSettingsClearSpace : clears the entire space from nucleotides,

OnSettingsCustomMagnification : loads the CSetMagDlg dialog and allows the user to choose his own magnification level ,

OnSettingsCustomnumberofsteps : loads the CSetStepDlg dialog and allows the user to choose the number of steps ,

OnSettingsDeleteStrand : loads the CStrandList dialog and allows to delete or highlight strands in a list,

OnSettingsPlaceSatrand : loads the CInsertDialog dialog and allows to place a strand randomly or not,

OnSettingsRandomStrand : places a strand randomly, with size chosen between 2 and 12, and random nucleotides (no c-bonds),

- The Drawing functions :

ClearSpace : clears the entire space from nucleotides,

convert : converts the coordinates into the screen , given a magnification level,

DrawCell : $(Node * nod, intType, intWay) \longrightarrow \emptyset$
draws a single cell, at the place given by the Node nod. the Type variable is the type of the nucleotide, as given by the automata (Adenine, Guanine ...). The Way is either it is a headr or not This function also redraws the neighbors after havng drawn the nucleotide,

DrawDblCell : $(Node * nod, intType, intType2, intWay) \longrightarrow \emptyset$
draws a double nucleotide cell,with the same variables, and a second type to decide of the color. The two color are assumed not to be the same from the SetColor function,

DrawGraph : draws the entire graph,

DrawNeighbors : $Node \longrightarrow \emptyset$
draws the neighbors of a given node,

DrawNucleotides : redraws the entire nucleotides of the graph,

FastDrawNucleotides : gets the arrays described in the Automata Class, and redraws only what changed on the graph. Much faster than the previous function,

getcoords : $Node \longrightarrow double*$
converts a node into its coordinates in space,

HighLightCell : highlights a givern cell,

SetColor : $(intCell, intType) \longrightarrow (intRet1, intRet2)$
From a given Cell and the type of the nucleotides inside, is able to find the colors used to drawing the cell. The cell number is given so as to tell the difference between hybrids and single nucleotides (conflict between AA and T for example),

transform : matrix transformation supposed to be able to make rotations around the axis,

UnDrawCell : redraws a blank circle in place of the cell,

The colors used in the program are defined at the begining of this file, and might be changed at will. The difference between a header or not a header is the tone of the color (each color from 0 to 7 has its dark tone which is from 8 to 15, found by adding eight). The color types are defined by `ADCOL`, `GUCOL` ... The void color is `VDCOL`, used to undraw the cells. The highlight color `HGCOL` is used to highlight the cells.

The different types of nucleotides are defined as NORMAL (single nucleotide non header), HEADER (single nucleotide header), and DOUBLE and DBLHDR which are the same for double nucleotides. Beware that the `DrawCell()` and `DrawDblCell()` functions only accept certain kinds of nucleotides.

The `ADJUST` variable is a manual magnification adjustment level, which allows to calculate automatically a good magnification of the graph, knowing the size of the screen and the maximal coordinate of the graph. The `PATH` variable allows to load files in a certain directory.

Appendix C - Creating a Cayley Graph File

The user may design his own files for test tubes in *Edna+*. These files must absolutely be in text format, as in the NotePad for example. Here are the rules to write a valid `.gr` file for our program :

1. The first line must contain the number of nodes of the graph, then a return
2. Then, each line will contain data about the node. The node will be loaded in the order of the file, line by line. The order of the nodes has vry few importance
3. For each line, first write the coordinates of your node in the space, separated by a blank space or any character different of a number. there must be three coordinates
4. Then write the number of neighbors that this node has in the current graph
5. Write a value here : 0 means inside the graph (all neighbors willbe displayed), 1 means on the border (no neighbor displayed) and 2 means in a corner (only the first neighbors will be displayed)
6. Then write the list of the numbers of the neighbors, as they are in the file. If this list is not of the right length, the graph won't be valid. This list must order the nodes in the right way, as described in the "Class Automata" section.
7. The line will have the following aspect :

   ```
   (Xcoord,Ycoord,Zcoord) N Edge Neighb1 ... NeighbN
   ```

8. Then restart for each node in your graph

9. The last line contains the data about the directions in your graph. There you will put a list of numbers which tells which neighbor is the opposite of the current one. For example, if your generators are $(e_1, e_2 \ldots e_n)$, then the list will be

$$(e_1^{-1}, e_2^{-1}, \ldots e_n^{-1})$$

The following rules must be respected when you create such a file :

- The lines beginning with `#` are supposed to be comment and won't be read
- No blank line will be allowed, only comments
- The program won't support wrong files. If your file is not valid, you might check the output written when loading the file. This is the `output` file in the same directory.
- The program may load wrong files successfully, but if data are false, then it may crash.
- Numbers can be negative
- Any other character than a number, '-' and a return will be ignored. You may write your files as you want for readability.

A simple example of file designed this way is `Test.gr`, which is a 2D grid, with height and width 3. The file contains the following lines :

```
# Test for Graphs Reading
# These lines are commentaries and won't be read
# First line is for the number of elements
10
# Then each line gives the links with other cells as follow
# (x_coord,y_coord) NbLinks  LinksList
(0 0 0) 2 0 1 3
(-100 0 0) 3 0 0 2 4
(-100 -100 0) 5 0 1 3 5 7 9
(0 -100 0) 3 0 0 2 8
(-200 0 0) 2 0 1 5
(-200 -100 0) 3 0 2 4 6
(-200 -200 0) 2 0 5 7
(-100 -200 0) 3 0 2 6 8
(0 -200 0) 2 0 3 7
(-100 -100 -100) 0 1 2
```

This file correspond to the following graph :