doi:10.1093/comjnl/bxhXXX

Dynamic Neighbourhood Cellular Automata*

Stefan Dantchev

Department of Computer Science, Durham University
Email: s.s.dantchev@durham.ac.uk

We propose a definition of Cellular Automaton in which each cell can change its neighbourhood during a computation. This is done locally by looking not farther than neighbours of neighbours and the number of links remains bounded by a constant throughout. We suggest that Dynamic Neighbourhood Cellular Automata can serve as theoretical model in studying Algorithmic and Computational Complexity issues of Ubiquitous Computations. We illustrate our approach by giving an optimal, logarithmic time solution of the Firing Squad Synchronisation problem in this setting.

Received XX Xxxx 200X; revised XX Xxxx 200X

1. INTRODUCTION

General background The concept of Cellular Automata (CA) goes back to works by Von Neumman and Ulam in the early 1950's [1]. A CA is a discrete structure consisting of identical cells arranged in a regular way (such as a line, a two dimensional square grid, etc.), and which compute synchronously. Each cell is a deterministic finite-state automaton whose transition function depends on its own state as well as on the states of the neighbours. Thus one can think of a CA as a massively parallel system, which consists of very simple building blocks than can interact only locally. Von Neumman himself was motivated by his own question of existence of self-reproducing machines and proved that there is a CA which can perform a universal computation. Since then CA have received much attention from theoretical point of view as well as have been used for modelling and simulations [2]. It is now known that even very simple one-dimensional CA can exhibit a very complex global behaviour [3], and therefore issues such as reversibility, conservation laws, limit sets, decidability questions, universality and topological dynamics of CA have been extensively studied [4].

Motivation In our paper, we focus on Algorithmic and Computational Complexity aspect of a certain less-studied kind of CA, namely one in which a cell can dynamically change its neighbourhood by linking to neighbours of neighbours. One can argue that such a model, which we call Dynamic Neighbourhood Cellular Automaton (DNCA), can serve as a basis for studying computational resources in Ubiquitous Computation - a new, quickly developing area which is one of the UKCRC grand challenges [5]. To illustrate the advantages of DNCA over the classical CA, we show an exponential speed-up of the former over latter in solving a classical synchronisation task, the Firing Squad problem. More precisely, we give an algorithm that solves it in optimal time $\Theta(\log n)$ on a DNCA consisting of n soldiers.

Previous work It seems that Rosenfeld and Wu were the first to consider a CA that can dynamically change links between cells. While earlier work [6] was on recognition of certain regular structures of the underlying graph, the subsequent paper [7] contained several algorithms for transforming a regular link structure to another. The presentation of [7] is however rather informal. A more serious drawback of it is, in our opinion, that the authors did not recognise the importance of the potential speed-up of a CA with dynamic links over a traditional one. For instance, they presented a linear (in the number of cells) algorithm for converting a line into a balanced binary tree. We show in the present paper that this can be done in logarithmic time, which is optimal (up to a constant factor). As a matter of fact, Rosenfeld and Wu used the firing squad problem as a subroutine in their construction. We have had different motivation and quite the opposite approach - we suggest the use of creating a balanced binary tree as a building block for faster computations, in particular for solving the firing squad in optimal, logarithmic time.

*A preliminary version of this paper was presented at the bCS08 Visions of Computer Science Conference, held on September 22-24, 2008.

More recently, Dubacq [8] reconsidered the dynamic neighbourhood in the context of CA and compared several different models. He also proved a quite general "synchronisation" theorem, which included solving the firing squad problem in logarithmic time. While Dubacq's dynamically reconfigurable CA have been rigorously defined and the importance of the the speed-up (logarithmic versus linear) have been acknowledged, his model has a serious drawback in that it allows for a cell to be seen by an unbounded number of other cells. Firstly, this does not look reasonable from a practical point of view - a processor sending information to everyone else in a single unit of time. Secondly, it allows for a trivial solution of the firing squad problem by simply pointing everyone to the general - this can trivially be done in logarithmic time. In contrast to this, our definition bounds the number of connections, which a cell can have at any time, by a constant (the same holds in the model of Rosenfeld and Wu).

Rest of the paper is organised as follows. In section 2, we give the formal definition of what we call Dynamic-Neighbourhood Cellular Automata (DNCA). We have tried to keep it as simple as possible yet general enough. There are different ways in to extend the definition, which we discuss in section 4. The proof of the main result, a solution of the Firing Squad problem in $\Theta(\log n)$ time, is given in section 3. Finally, we discuss possible directions for further research in section 4.

2. DEFINITIONS

The formal definition of DNCA and some related concepts is as follows.

DEFINITION 2.1. DNCA *is a quadruple* $(Q, P, \delta, \mathcal{C})$ *where*

(i) Q *is a finite set of states;*
(ii) P *is a finite sets of ports;*
(iii) $\delta : Q^{|P|+1} \to Q \times \left(\{\varepsilon\} \cup P \cup P^2\right)^{|P|}$ *is the transition function and*
(iv) $\mathcal{C}$ *is a (potentially infinite) set of cells.*

Interconnection function is a function $\eta : \mathcal{C} \times P \to \mathcal{C} \times P \cup \{\perp\}$ *such that* $\eta(\eta(C,p)) = \langle C, p\rangle$ *for every* $C \in \mathcal{C}$ *and* $p \in P$ *such that* $\eta(C,p) \neq \perp$.

A (global) state of the DNCA then is a pair $\left(Q^{|\mathcal{C}|}, \eta\right)$.

The intuitive meaning of the definition is as follows. We have a set of identical Deterministic Finite-state Automata (DFAs) that we call cells. The cells are "named" by the elements of some set $\mathcal{C}$. Usually, we take $\mathcal{C}$ to be some countable set, e.g. the natural numbers $\mathbb{N}$. In the rest of the paper, however, we shall be mainly concerned with time complexity and, thus, we shall assume than only finitely many cells numbered from 1 to n are active - here n plays the role of "input size". We note that the cell names have no relevance to the actual computation of a DNCA as a cell is a DFA, which cannot hold $\log n$ bits required to memorise a name. All the cells have a common state set Q and the same transition function δ which depends on the cell's own state as well as on the states of up to $|P|$ neighbouring cells which are available through the cell's ports named by elements of some finite set P.

The links are formally defined with the help of the interconnection function: if two cells C_1 and C_2 are connected through ports p_1 of C_1 and p_2 of C_2 the interconnection function should consistently say so, i.e. $\eta(C_1, p_1) = \langle C_2, p_2\rangle$ and $\eta(C_2, p_2) = \langle C_1, p_1\rangle$. A port p of a cell C may be left loose - this is reflected by $\eta(C,p) = \perp$ and may be used to define external input/output to the DNCA even though we ignore this issue throughout the paper. Note that this definition allows for loops, i.e. $\eta(C,p) = \langle C, p\rangle$.

The transition function δ takes the current state of a cell C together with the current states of its (at most)$|P|$ neighbours and then returns a new state plus $|P|$ port changes, each of them being of the following three kinds: ε (meaning the port is left loose), p (meaning connect to the cell which is currently connected to the port p of C) or $\langle p_1, p_2\rangle$ (meaning connect to the cell which is currently connected to port p_2 of the cell which is connected to port p_1 of C - i.e. switch a connection from a neighbour to a neighbour of a neighbour). All these changes that produce a new global state from the current one happen synchronously in parallel. It could happen that the new interconnection function is inconsistent, i.e. the condition $\eta(C,p) \neq \perp$ and $\eta(\eta(C,p)) \neq \langle C, p\rangle$ for some $C \in \mathcal{C}$ and $p \in P$ - we shall treat such a situation as run-time error in which case the computation of the DNCA fails.

3. FIRING SQUAD IN OPTIMAL, LOGARITHMIC TIME

Firing Squad problem is a synchronisation problem for a line of finite automata (i.e. one-dimensional CA) proposed by Myhill in 1957, first solved by McCarthy and Minsky, and appeared in print in [9]. The problem itself is as follows. The first (leftmost) cell is a general and all the others are soldiers. At some point in time, the general is placed in a special state "fire when ready". The computational task is to all soldiers into a "fire" state simultaneously at some later time, and it must be the first time that any of them has fired.

The simple divide-and-conquer solution of Moore starts by propagating two signals along the line: a fast signal and a slow one, which moves three times as slow. The fast signal bounces off the end of the line and meets the slow signal in the centre. The middle soldier declares himself a general and with the help of additional two signals, he agrees with the original general that they both give order to fire at the same time, each for his half of the line . Thus, the process recursively continues, halving each sub-line

until each division consists of a single soldier (who becomes general at that moment). Then every soldier fires. It is easy to see that the time required is $O(n)$. Since then, a number of solutions have been found including an absolute optimal ones in terms of time, number of states, communication bits etc. The problem has been generalised to many different topologies (see the survey [10]).

We shall assume throughout the section that we are given a DNCA with n linearly ordered cells. That is, the cell names are the numbers from 1 to n, and there are we successor and predecessor ports defined by $\mathrm{succ}(x) = x+1$ for every $x \in [1 \ldots n-1]$, $\mathrm{succ}(n) = \varepsilon$ and $\mathrm{pred}(x) = x-1$ for every $x \in [2 \ldots n]$, $\mathrm{pred}(1) = \varepsilon$. We also assume that, in the beginning, all the cells are in some *idle* state except for cell 1, which initiates the computation, and cell n, which knows that it is the last in the line.

We start by converting the line into a balanced binary tree (BBT)

Creating the BBT is done at two stages. At the first one, an almost balanced tree is constructed by setting the root to be cell 1 and then recursively constructing two disjoint almost balanced sub-trees - one with a root 2 and containing all even-numbered cells, and the other with root 3 and containing all odd-numbered cell (except 1, of course). The recursive split is made in constant time, by simply linking any cell whose name is ≥ 4 to the neighbour of the neighbour in both directions. Special care is needed in the boundary cases, i.e. the cells that are neighbours of the end(s) or the root(s).

More formally, the first stage is meant to leave any cell in a state of the form $(vertex-type, parent-type)$. Here $vertex-type$ is $leaf$ (a leaf of the tree), $lean$ (a vertex that has a single left child which is a leaf) or $node$ (any other internal vertex, including the root). $parent-type$ is $left$, $right$ (the node is a left, respectively right, child) or $none$ (for the only root of the tree). The process starts by cell 1 going to a state $(root, none)$. A root node links the next two nodes as the left and the right (only if it exists) children:

Algorithm 1 Action performed by a $(root, \star)$

$\mathrm{left}(x) := \mathrm{succ}(x)$

if $\mathrm{State}(\mathrm{succ}(x)) = end$

then $\mathrm{State}(x) := (lean, \star)$

else $\mathrm{State}(x) := (node, \star)$,
$\mathrm{right}(x) := \mathrm{succ}(\mathrm{succ}(x))$

An idle cell should become a left/right root only if its predecessor/predecessor of predecessor has been reached. Otherwise it should link to the neighbour of the neighbour in both directions with special care needed if it is next to an end cell:

Algorithm 2 Action performed by an idle cell

if $\mathrm{State}(\mathrm{pred}(x)) = (node, \star)$

then $\mathrm{State}(x) := (root, left)$,
$\mathrm{par}(x) := \mathrm{pred}(x)$

elseif $\mathrm{State}(\mathrm{pred}(\mathrm{pred}(x))) = (node, \star)$

then $\mathrm{State}(x) := (root, right)$,
$\mathrm{par}(x) := \mathrm{pred}(\mathrm{pred}(x))$

else $\mathrm{pred}(x) := \mathrm{pred}(\mathrm{pred}(x))$

if $\mathrm{State}(\mathrm{succ}(x)) = end$

then $\mathrm{State}(x) := end$
else $\mathrm{succ}(x) := \mathrm{succ}(\mathrm{succ}(x))$

An end cell should become a left/right leaf only if its predecessor/predecessor of predecessor has been reached:

Algorithm 3 Action performed by an "end" cell

if $\mathrm{State}(\mathrm{pred}(x)) = (node, \star)$

or $\mathrm{State}(\mathrm{pred}(x)) = (lean, \star)$

then $\mathrm{State}(x) := (leaf, left)$,
$\mathrm{par}(x) := \mathrm{pred}(x)$

elseif $\mathrm{State}(\mathrm{pred}(\mathrm{pred}(x))) = (node, \star)$

then $\mathrm{State}(x) := (leaf, right)$,
$\mathrm{par}(x) := \mathrm{pred}(\mathrm{pred}(x))$

else $\mathrm{pred}(x) := \mathrm{pred}(\mathrm{pred}(x))$

The figure below is an example of how the first stage works on a 8-element line. The double-circled vertices show the recursive propagation of the root(s). In the end, $1, 2, 3$ are nodes, 4 is a lean, and $5, 6, 7, 8$ are leaves.

One can easily prove the following key properties of the ABBT produced by the first stage.

LEMMA 1. *The almost balanced binary tree obtained from a line of n cells is*

(i) *of height* $h = \lfloor \log_2 n \rfloor + 1$.

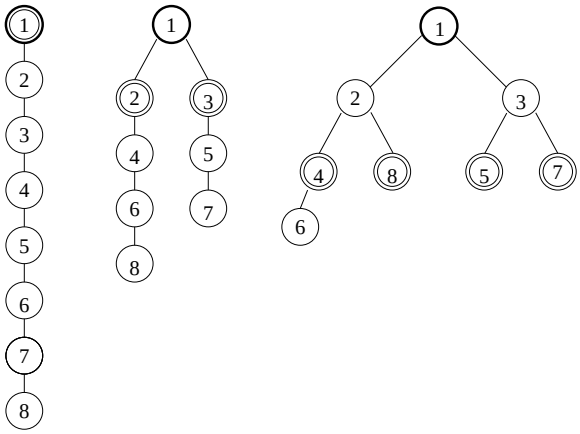

FIGURE 1. Creating an A(lmost)BBT.

(ii) *For every internal node, the difference in the number of vertices in the left sub-tree and the right sub-tree rooted at that none is either zero or one.*
(iii) *Every leaf of the tree is at depth either* h *or* $h-1$.

PROOF. Straightforward bounded induction (from 0 to $h-1$) on the number of rounds, each round being a point in time at which a "root" message gets propagated down. □

The task of the second stage is to find all the leaves at the higher depth $h-1$ and to advise them to pretend that they have a child (at level h). This could easily be done recursively. Given a node with left and right sub-trees L and R, respectively, the height of L is greater than the height of R only if $|L| = |R| + 1 = 2^t$ for some t. In this case all leaves of R should be extended and L should be recursively balanced. Otherwise, L and R have the same height, so they should be recursively balanced.

The implementation of this stage is better described in terms of message passing. It is initiated by the unique root of ABBT who sends a message *balance* to both his children. This message is passed downwards by ordinary nodes until it reaches a lean or a leaf. A lean is a root of sub-tree of size two, so it replies back to its parent by a message *power* whose informal meaning is "my sub-tree is of size which is a perfect power of two". A leaf replies back to its parent by a message *balanced* whose meaning is "my sub-tree is of perfectly balanced", i.e. of size which is a perfect power of two minus one. Now every node awaits messages from both children. Whenever a node receives two *balanced*, it passes *balanced* to its parent. If a node gets *power* from the left and balanced from the right, every leaf of the right should be extended. This is done by sending a message *extend* to the right child and, at the same time, *power* to the parent. Any other combination of two messages from the children can simply be ignored. An *extend* message is passed downwards until it reaches a leaf that then gets to know that it should pretend it had a child.

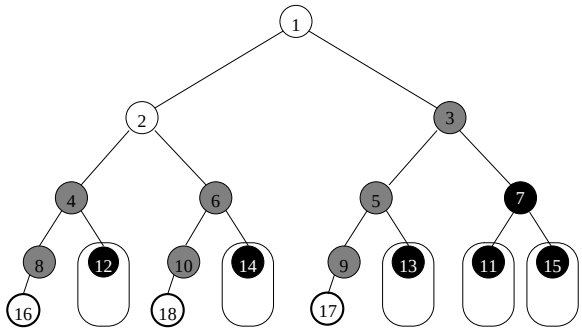

FIGURE 2. Completing the tree

An example with 18 nodes is given below. The black nodes are roots of balanced sub-trees, while the grey ones are roots of sub-trees whose size is a perfect power of two. The grey *node* vertices $3, 4, 5, 6$ sent *extend* message to all leaves in their respective right sub-trees.

Finally, we point out that the two stages can work in parallel (more precisely the second stage lags a step behind the first one) - it is not hard to see that whenever a node needs to send or receive a message, the relevant links as well as the type of the node has already been established by the first stage. In order to solve the firing squad problem, we can run any known algorithm on all branches in parallel with a speed three times slower than normal - it is easy to see that every signal sent by such an algorithm will reach a soldier who has already established his position (i.e. knows he is the end of the line, or knows he has to simulate an additional fake soldier). The running time is clearly $O(\log n)$. The optimality follows from the fact that, the reach of any cell at time k is at most 2^k, so that the last in the line cannot even see the order to fire in fewer than $\lceil \log n \rceil - 1$ steps. Apart from the running time, there are other resources that can be of interests, namely the number of messages sent by a node as well as the number of times a node needs to re-link. These can easily be counted, and all upper and lower bounds on the resources are summarised below.

THEOREM 3.1. *There is a DNCA algorithm that solves the FSSP on a line of* n *cells in optimal time* $\Omega(\log n)$ *time with each cell having re-linked at most* $O(\log n)$ *times. The number of messages sent,received or generated by a cell is* $\Omega(1)$ *plus the number of messages needed to solve the sequential FSSP on a line of length* $\log n$.

4. CONCLUSION AND FUTURE WORK

We have given a basic definition of Dynamic Neighbourhood Cellular Automata and have shown that a DNCA can exhibit an exponential speed-up over a (static neighbourhood) CA. There are a number of ways to extend the model as well as many other algorithmic and complexity issues to be considered.

(i) Consider cell that are more powerful than deterministic finite-state automata. These could be finite automata whose memory is $\Theta(\log n)$, so that a cell could know its own name or memorise another cell's name. In this case, one may need to restrict in some way the transition function δ, so that a cell is not as powerful as a log-space Turing machine.

(ii) Consider complexity measure other than running time. An example of this, particularly relevant to Ubiquitous Computations could be "energy", i.e. the number of times a cell receives/sends a message or changes its neighbourhood.

(iii) Consider DNCA consisting of non-identical cells, i.e. cells that have different computational power. Make precise the notion of input to and output of a DNCA.

(iv) Consider different algorithmic problems that have been studied in the context of Distributed Computation. These might include Leader Election, Byzantine Agreement etc.

REFERENCES

[1] Von Neumann, J. (edited and completed by A. W. Burks) (1966) *The Theory of Self-Reproducing Automata*. University of Illinois Press.

[2] Ganguly, N., Sikdar, B. K., Deutsch, A., Canright, G., and Chaudhuri, P. P. (2003) A survey on cellular automata. Technical report. Centre for High Performance Computing, Dresden University of Technology.

[3] Wolfram, S. (2002) *A new kind of Science*. Wolfram Media Inc., Champaign, Ilinois, US, United States.

[4] Kari, J. (2005) Theory of cellular automata: A survey. *Theoretical Computer Science*, **334**, 3–33.

[5] Sloman, M., Chalmers, D., Crowcroft, J., Kwiatkowska, M., Milner, R., Rodden, T., and Sassone, V. Ukcrc grand challenges for computing research, ubiquitous computing: Science and design. Available at http://www-dse.doc.ic.ac.uk/Projects/UbiNet/GC/index.html.

[6] Rosenfeld, A. and Wu, A. Y. (1979) Cellular graph automata i and ii. *Information and Control*, **42**, 305–353.

[7] Rosenfeld, A. and Wu, A. Y. (1981) Reconfigurable cellular computers. *Information and Control*, **50**, 64–84.

[8] Dubacq, J.-C. (1994) Different kinds of neighborhood-varying cellular automata. Maîtrise / honour bachelor degree École normale supérieure de Lyon.

[9] Mooore, E. F. (1964) The firing squad synchronization problem. In Moore, E. F. (ed.), *Sequential Machines, Selected Papers*, pp. 213–214. Addison-Wesley, Reading, MA.

[10] Mazoyer, J. (1986) An overview of the firing squad synchronization problem. In Choffrut, C. (ed.), *Automata Networks*, Lecture Notes in Computer Science, **316**, pp. 82–94. Springer.