INFORMATIQUE THÉORIQUE ET APPLICATIONS

S. VENKATASUBRAMANIAN
KAMALA KRITHIVASAN
C. PANDU RANGAN

Algorithms for weighted graph problems on the modified cellular graph automaton

Informatique théorique et applications, tome 23, n° 3 (1989), p. 251-279

<http://www.numdam.org/item?id=ITA_1989__23_3_251_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 23, n° 3, 1989, p. 251 à 279)

ALGORITHMS FOR WEIGHTED GRAPH PROBLEMS ON THE MODIFIED CELLULAR GRAPH AUTOMATON (*)

by S. VENKATASUBRAMANIAN (1), KAMALA KRITHIVASAN (1)
and C. PANDU RANGAN (1)

Communicated by J. BERSTEL

Abstract. – *A Modified Cellular Graph Automaton* (MCGA) *is formulated and algorithms for solving weighted graph problems viz., minimum weight spanning tree construction (based on Kruskal's* [5] *sequential algorithm) and single source shortest paths (based on Dijkstra's* [6] *sequential algorithm) are described on it. The original Cellular Graph Automaton* (CGA) *of Wu and Rosenfeld* [3, 4] *is modified by the introduction of a second type of Finite State Automaton* (FSA) *on the edges of the input d-graph. The equivalence of the* MCGA *to the* CGA *with respect to the acceptance of graph languages is shown.The memory size of the* FSA *in our* MCGA *is made proportional to the area (i. e., the total number of nodes) of the input graph, as suggested by Wu and Rosenfeld* [3, 4]. *It is shown how the above two modifications to the original* CGA *facilitates construction of simpler and faster algorithms for solving weighted graph problems.*

Résumé. – *On définit la notion d'automate cellulaire modifié pour graphes* (ACMG) *et on décrit, dans ce cadre, des algorithmes pour des graphes pondérés, comme le problème de l'arbre recouvrant minimal, ou le problème des plus courts chemins à partir d'une origine commune.La modification par rapport à l'automate cellulaire original de Wu et Rosenfeld consiste en l'introduction d'un deuxième type d'automates finis qui est associé aux arêtes du graphe.*

1. INTRODUCTION

Von Neumann's book on *Theory of Self-Reproducing Automata* was the pioneering work on cellular automata. A. R. Smith [1] studied in detail the acceptance powers of the one-dimensional cellular automata and iterative automata. The iterative automaton is slightly different from the cellular automaton in the sense that the input string is fed into the automaton from one end of the chain of finite state automata. The cellular graph automata

(*) Received January 1987, revised September 1988.
(1) Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India. Net Address: –uunet! shakti! shiva! rangan

of Wu and Rosenfeld [3] (CGA, hereafter) incorporates the input pattern into the initial configuration of the underlying graph.

The CGA, which is more general than the 1-*D* cellular automata or 2-*D* cellular arrays, was first studied by Rosentiehl [2] under the term "intelligent graphs", apparently because the network of finite state automata (FSA, hereafter) can measure the properties concerning its own underlying structure. He presented algorithms to find Eulerian paths, spanning trees and Hamiltonian cycles on this "intelligent graph" where all nodes have fixed degree. Wu and Rosenfeld [3] introduced special kind of single degree nodes that are always in "quiescent" state, thus enabling a node to have degree less than the maximum possible number for that FSA. The graph obtained after the introduction of such single degree nodes is called a *d*-graph, where each node has *d* arcs emanating from it and each arc end at a node is given a distinct number between 1 and *d*. Such *d*-graphs have also been studied by Mylopoulos [8], as mentioned in Wu and Rosenfeld [3]. The two papers on cellular Graph Automata by Wu and Rosenfeld [3, 4] give a fairly comprehensive treatment of the CGA which can accept various kinds of graph structures of bounded degree and also measure their various graph properties viz., radius, area, centre, cut-nodes, blocks, etc. Their formalism of having only single type of FSA at all the nodes of the input *d*-graph was not found to be particularly suitable for solving problems related to digraphs, weighted graphs etc., where the edges also bear some special properties e. g., direction, weights, labels, etc. It was found that algorithms on this CGA for solving problems like finding fundamental cutsets, rooted acyclic graph recognition, minimum weight spanning tree construction, single source shortest paths, etc., become unduly complicated and inefficient. This has led us to propose a slight modification to their original CGA, which is the introduction of a second type of FSA on the edges of the input *d*-graph. These edge automata take care of the special properties viz., direction of digraph edges, costs of the weighted edges, etc., so that the algorithms become simpler and faster. We have also achieved improvement in the time complexities of the algorithms by making the memory size of the FSA proportional to the area (*i. e.*, the total number of nodes) of the input graph. This was actually suggested by Wu and Rosenfeld themselves as a case for further research, and accordingly we observe its effect of speeding up the algorithms.

We first define our modified CGA in the next section. Section 3 establishes the equivalence (w. r. t. acceptance power) of CGA and MCGA. In section 4, we give details of certain high level descriptions of actions that are frequently used in our algorithms. In section 5, we present the actual algorithms on

our modified cellular graph automaton for solving various weighted graph problems.

2. DEFINITION OF THE MODIFIED CELLULAR GRAPH AUTOMATON

2.1. Graphs

2.1.1. *A general graph*

A graph in the context of our discussions is described by a five tuple (N, A, f_N, f_E, g) where:

N is a finite non-empty set of the nodes of the graph;

A is a collection of pairs of distinct elements of N, called the set of arcs or edges;

$f_N: N \to L_N$ is a function that maps each node in N to a label in L_N;

$f_E: A \to L_E$ is a function that maps each arc in A to an edge label in L_E;

$g: N \times N \to Z$ is the neighbourhood ordering function that defines the ordering of the immediate neighbours of a node. That is, if $g(n, m) = i$, then we say that m is the i-th neighbour of n. "g" is a partial function because it is not defined for all the elements of $N \times N$. Note that "g" is defined for only those pairs where the second component node is a neighbour of the first component node. L_N and L_E are finite non-empty label sets.

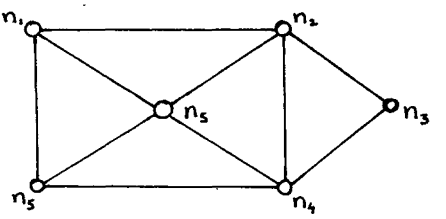
2.1.2. *d-bounded graph*

A d-bounded graph is a graph in which the degree of every node is bounded by the integer "d". In our context, d is a constant and it is independent of the total number of nodes in the graph. For example, figure 1 *a* illustrates a 4-bounded graph.

2.1.3. *d-graph*

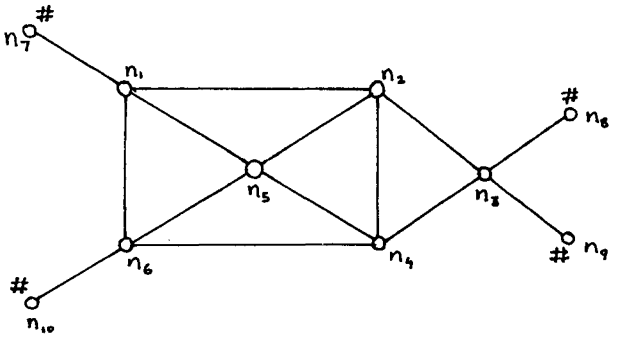
To any d-bounded graph, we can add suitable number of dummy nodes and introduce new edges between the nodes of the graph that have degree less than "d". We introduce the edges in such a way that each non-dummy node has degree d. Since the dummy nodes introduced have all degree one, g may be extended for the dummy nodes as follows: $g(x, m) = 1$ where x is a dummy node and m is adjacent to x.

Now the resulting graph will have nodes with degree d or one. We recall once again that only the newly introduced nodes (dummy nodes) have degree 1 and all of them have a reserved label #. The label # is not used for any



other node that has degree d. We will call the resulting graph as a d-uniform graph or a d-graph in short.

(*Fig.* 1 *b* illustrates the resulting 4-uniform graph of the 4-bounded graph given in *fig.* 1 *a.*)



2.1.4. *Underlying graph*

The d-bounded graph obtained after the deletion of all the vertices labeled # in a d-graph Γ is also referred to as the underlying graph $U(\Gamma)$.

2.2. Modified cellular d-graph automaton

2.2.1. *Informal description*

2.2.1.1. *General description*

Our modified cellular graph automaton, which is working on a d-graph, will have only two kinds of finite state automata (FSA), with one kind of FSA placed at every vertex of the input graph and the other kind on every edge. We shall denote by M_{NA} the FSA that is placed on the vertices and by M_{EA} the FSA placed on the edges. In the earlier models, the automaton placed at a vertex can "know" the state of the automaton placed on each of its neighbouring nodes. But, in our model, we use an edge automaton to pass "information" between the two node automata placed on the end vertices. We represent a typical state of the edge automaton as a 3-tuple where the second and last component denote the states of the node automata placed at the end vertices of the given edge and the first one is related to the edge itself. We shall now turn to the informal description of how the node automata and the edge automata actually work.

2.2.1.2. *Node automaton*

A node automaton's next state is dependent upon its current state and the states of the d edge automata that are placed on the d edges incident on the node. Recall that a typical state of an edge automaton is a 3 tuple; the first one pertains to the information related to the edges and the second and third components are the states of the end vertices. Therefore, the current node's state may occur as the second or third component in the state of an edge automaton. If the current node's state occurs as the second component then its neighbour's state will occur as the third component and vice versa. Thus, we see that at any given node if we want to access the state of a neighbouring node, then we must look at the second or third component of the state of the corresponding edge automaton depending upon the given node's state occurring as third or second component of the state of the edge automaton. Observe that the "g" function provides an order to the neighbours of a node and that every (non-dummy) node has d neighbours. Also note that g is not symmetric, *i.e.* $g(n, m)=i$ need not imply that $g(m, n)=i$. We specify that the i-th neighbour's state occurs in the corresponding edge automaton's state as k-th component ($k=2$ or 3) by the pair (i, k). For any given node, the sequence $(1, k_1)$, $(2, k_2)$, . . . , (d, k_d) defines a "neighbourhood state position vector" or in short a "neighbourhood vector". The neighbourhood vectors do not change and are determined once for all by the input d-graph.

Let u and v be adjacent nodes in the d-graph. Fix u. Now v might be the i-th neighbour of u and u might be the j-th neighbour of v. We may incorporate this mutual order in the neighbourhood vector of u by replacing the pair (i, k_i) by (i, k_i, j). Such a sequence of triplets are called mutual neighbourhood vector of u. In fact, the first component of the tuple in the above definitions can be dropped as they are implied by the order of tuples in the vector. Henceforth, we will assume that the neighbourhood vectors have the form $(2, 3)^d$ and mutual neighbourhood vectors to have a form $((2, 3) \times Z_d)^d$, where Z_d is the set $(1, 2, \ldots, d)$.

2.2.1.3. *Edge automaton*

We shall now see what is actually involved in the transition of an edge automaton. Recall that a second kind of FSA is placed on each of the edges of the input d-graph. Actually, this is the place where our MCGA differs from the original CGA of Wu and Rosenfeld [3, 4]. In their CGA, each node automaton is directly connected to d other neighbouring node automata whereas in our case it is connected to the d incident edge automata. In order to retain the properties of the original CGA (while trying to improve it), it has become necessary that the intervening edge automaton (between every pair of node automata) communicate the states of the end nodes to each other. To achieve this, the state of an edge automaton is always in the form of a 3-tuple in which:

(i) The first component gives the information relating to the actual state of the edge automaton and in fact, might itself be a 2-tuple (w_e, q_e) where w_e is the constant label or weight that is to be associated with the edge automaton throughout the life of the edge automaton, and q_e is used to record the information concerning that edge that changes with time.

(ii) The second and third components directly indicate the states of the two end nodes and thus enable them to know each others' state.

The transition of an edge automaton actually involves the storing of the end nodes' states in the respective (2nd or 3rd) components as well as an updation (if any) of q_e in the first component. As a given node automaton knows the state of a neighbouring node automaton only through the state of the intervening edge automaton, the alternation of the transition phases of the edge automata and node automata is necessitated. Thus, in each time step, our MCGA executes two phases of transition, wherein,

(i) the first phase corresponds to that of all the edge automata of the MCGA, and

(ii) the second phase corresponds to that of all the node automata of the MCGA.

2.2.1.4. *Initial configuration*

The initial configuration for the MCGA consists of the input d-graph with each of the node automata and the edge automata bearing a label from the sets L_N and L_E respectively. The node automata placed on the "dummy nodes" that have a degree one have their initial (unchanging) state as "#". The corresponding edge automaton (whose one end node has a label "#") also bears a label "#" that also does not change with time. (Recall the notation introduced in definition 2.2.1.3. In fact, # is nothing but W_e of definition.) This is because the pendant ("dummy") node and the corresponding pendant edge do not form a part of the underlying graph that actually has the labels assigned to edges and nodes.

2.2.2. *Formal definitions*

The MCGA is a 5-tuple $(\Gamma, M_{NA}, M_{EA}, H_{NA}, H_{EA})$ where:

Γ is the input d-graph

H_{NA}—is the neighbourhood vector defined for all the nodes in Γ. The first component of the i-th element in this vector indicates to the node automaton which component (2nd or 3rd) of the i-th edge automaton is to be considered in its transition function and the second component gives the mutual neighbourhood number, *i.e.*, if $H_{NA}(n)\ (i, 2)=k$, then this node n is the k-th neighbour of its i-th neighbour.

M_{NA}—is the first kind of FSA (Q_{NA}, δ_{NA}) placed at each of the nodes. The transition function is:

$$\delta_{NA}: Q_{NA} \times Q_{EA}^{d} \times H_{NA} \to Q_{NA}$$

(if $q_{NA}^{t}=\#$, then $q_{NA}^{t+1}=\#$ for all instants, $t=1, 2, \ldots$).

H_{EA}—is the neighbourhood vector for the edge automaton. For a given edge automaton e, the two elements of $H_{EA}(e)$ give the two arc end numbers of this edge with respect to the two adjacent nodes.

M_{EA}—is the second kind of FSA (Q_{EA}, δ_{EA}) placed on the edges of Γ. The transition function is:

$$\delta_{EA}: Q_{EA} \times Q_{NA}^{2} \times H_{EA} \to Q_{EA}.$$

Usually,

$$\delta_{EA}((q_e, q_1, q_2), q_1', q_2', (t\,1, t\,2)) = (q_e', q_1', q_2').$$

A configuration of M is a pair of mappings, one corresponding to the nodes and another for the edges. The mappings assign states to the nodes and edges and we denote them by c_N and c_E respectively. Thus,

$$c_N : N \to Q_{NA}$$

and

$$c_E : A \to Q_{EA}$$

and the configuration itself is denoted as $c = (c_N, c_E)$.

At the end of a time step, the configuration of the MCGA might change from, let us say, c to c' and we denote that by $c \mid - c'$. We also denote the initial configuration by c_0 and the configuration resulting at the end of the t-th time step by c_t. The corresponding functions are denoted by $c_{N,t}$ and $c_{E,t}$.

One can define a MCGA with distinguished node as the one for which a particular state in the set Q_{NA} is marked as distinguished, which is recognisable by all FSA, and the node bearing this as the label at the initial step is called a "distinguished node". This unique label usually does not change for that node and is used to initiate many algorithms. A configuration c_k at the end of the k-th time step is said to be a terminal configuration if the node automaton with a distinguished label D enters into a final state.

One significant improvement on the size of the input symbols for the node automaton in our MCGA is noted below as a lemma.

LEMMA 2.2.1: *If the state of every node automaton is always in the form of a d-tuple, (storing in its i-th component the information relevant to its i-th neighbour,) then the total number of the symbols for the original* CGA *is* d^2. *However, for the* MCGA, *the number of input symbols is just* $3d$.

Proof: Usually, the node automata send some signals to one or more of their d neighbours. In that case, their states are in the form of a d-tuple where each component contains some information relevant to the corresponding neighbour. For the original CGA, the node automaton has to consider the d-tuple from each of its d neighbours before getting into its next state. This means that each node automaton has to process d^2 input symbols at every transition step. However, we shall see how an additional edge automaton between every pair of node automata reduces this number by an order of magnitude (with respect to "d"). An edge automaton, by using the information provided by $H_{EA}(e)$ (the neighbourhood vector for the edge automaton), can register in its second and third components only those components of the d-tuples of its end nodes that will be relevant to each other.

For example, define the neighbourhood vector for a given edge automaton e as $H_{EA}(e)=(t1, t2)\in Z_d^2$ for each edge automaton e, where e is the $t1$-th incident edge on its first end node and $t2$-th incident edge on its second end node. Assume that the state of each node automaton is in the form of a d-tuple denoted as

$$q_{NA}=(q_1, \ldots, q_d)$$

where q_i is the information relevant to the i-th neighbour of that node. Then, in the case of the original CGA of Wu and Rosenfeld, the transition of a node automaton is typically of the form

$$\delta(q_{n\,1}, \ldots, q_{nd}), ((q_{1,\,1}, \ldots, q_{1,\,d}), \ldots, (q_{d,\,1}, \ldots, q_{d,\,d})), H(n))=(q_{1,\,k\,1}, q_{2,\,k\,2}, \ldots, q_{d,\,kd})$$

where $ki=H(n)$ (i), that is, n is the ki-th neighbour of its i-th neighbour. Thus, the node automaton at n has to process d^2 input symbols at every transition step.

However, in our MCGA, the transition of the edge automata precedes that of the node automata. The transition of the edge automaton placed on e is:

$$\delta_{EA}((q_e, q_{1,\,t\,1}, q_{2,\,t\,2}), ((q'_{1,\,1}, q'_{1,\,2}, \ldots, q'_{1,\,d}), (q'_{2,\,1}, q'_{2,\,2}, \ldots, q'_{2,\,d})), H_{EA}(e))=(q_e, q'_{1,\,t\,1}, q'_{2,\,t\,2})$$

so that, for node n,

$$\delta_{NA}((q'_{n\,1}, q'_{n\,2}, \ldots, q'_{nd}), (q'_{e\,1}, q'_{e\,2}, \ldots, q'_{ed}), H_{NA}(n))=(q''_{n\,1}, q''_{n\,2}, \ldots, q''_{nd})$$

where each q_{ei} is in the form of a 3-tuple.

Thus, the intervening edge automaton reduces the number of input symbols for a node automaton in this case from a possible d^2 in the case of the original CGA to $3d$ in our MCGA. This leads to the reduction in the size of each time step.

2.3. Modified cellular d-graph languages

A modified cellular d-graph acceptor is an MCGA

$$\mathbb{M}=(\Gamma, M_{NA}, M_{EA}, H_{NA}, H_{EA})$$

with a distinguished node such that M_{NA} is a finite state acceptor specified by the 4-tuple

$$(Q_{I,NA}, Q_{NA}, \delta_{NA}, F_{NA}),$$

where (i) (Q_{NA}, δ_{NA}) is an FSA of the node defined as in the previous section; (ii) $Q_{I,NA}$ is the set of initial states. $Q_{I,NA} = L_N^d$, and (iii) $F_{NA} \subseteq Q_{NA}$ is the set of final states.

An initial configuration of $\mathbb{M}$ is denoted as c_0 and it consists of: (i) the function $c_{N,0}$ that maps N, the set of nodes of the input d-graph Γ into L_N, the set of node labels of Γ, and (ii) the function $c_{E,0}$ that maps A, the set of edges in Γ into L_E, the set of edge labels of Γ.

A configuration of the modified cellular d-graph acceptor $\mathbb{M}$, at the end of m time steps, denoted as c_m, is said to be a terminal configuration whenever the finite state acceptor M_{NA} placed on the node with a distinguished label "D" (appearing as a part of its state) has entered into a final state at that instant. Symbolically, if n is a node whose corresponding node automaton bears the label "D" in its state, then, $c_{N,m}(n) \in F_{NA}$.

$$\mathbb{M} = (\Gamma, M_{NA}, M_{EA}, H_{NA}, H_{EA})$$

accepts the d-graph $\Gamma = (N, A, f_N, f_E, g)$ if there is a finite sequence of configurations $c_0, c_1, \ldots, c_m$, such that $c_0 = (f_N, f_E)$ (recall that f_N and f_E are the labeling functions of Γ equivalent to $c_{N,0}$ and $c_{E,0}$ respectively) is an initial configuration, c_m is a terminal configuration $(m > 0)$ and $c_i \mid - c_{i+1}$ for $0 < i < m$ as defined above. For a given finite state acceptor on the nodes

$$M_{NA} = (Q_{I,NA}, Q_{NA}, \delta_{NA}, F_{NA}),$$

and the FSA on the edges $M_{EA} = (Q_{EA}, \delta_{EA})$, we can define the class of modified cellular graph acceptors determined by M_{NA} and M_{EA} as

$$\mathbb{C}(M_{NA}, M_{EA}) = \{ \mathbb{M} \mid \mathbb{M} = (\Gamma, M_{NA}, M_{EA}, H_{NA}, H_{EA}) \}$$

where H_{NA} and H_{EA} are neighbourhood vectors as defined earlier.

The language of d-graphs accepted by $\mathbb{C}(M_{NA}, M_{EA})$ is the set

$$\mathscr{L}(M_{NA}, M_{EA}) = \{ \Gamma \mid \mathbb{M} = (\Gamma, M_{NA}, M_{EA}, H_{NA}, H_{EA}) \in \mathbb{C}(M_{NA}, M_{EA}) \text{ accepts } \Gamma \}.$$

A d-graph Γ is accepted by $\mathbb{C}(M_{NA}, M_{EA})$ if and only if $\Gamma \in \mathscr{L}(M_{NA}, M_{EA})$.

3. EQUIVALENCE OF MCGA AND CGA

Let us denote the language accepted by an MCGA as L_{MCGA} and that by a CGA as L_{CGA}. We shall make use of this notation in our proofs for the equivalence of MCGA and CGA. Also, throughout this section, whenever *NA* and *EA* are used, they shall refer to node automaton and edge automaton respectively.

In the following Lemma, we shall denote the class of MCGA determined by a pair of FSA, *i.e.*, $M_{1\,N}$ placed on the nodes and $M_{1\,E}$ placed on the edges, as $C(M_{1\,N}, M_{1\,E})$. We shall also denote by $C(M_{2\,N})$ the class of CGA determined by the FSA on the nodes, *i.e.*, $M_{2\,N}$.

LEMMA 3.1: *Given a language* L_{MCGA} *accepted by* $C(M_{1\,N}, M_{1\,E})$, *there exists a corresponding class of* CGA $C(M_{2\,N})$ *that accepts the same language* L_{MCGA}.

Proof: Consider an arbitrary d-graph $\Gamma \in L_{\text{MCGA}}$. Given that $M_1 = (\Gamma, M_{1\,N}, M_{1\,E}, H_{NA}, H_{EA})$, we shall construct the corresponding CGA $M_2 = (\Gamma, M_{2\,N}, H)$ formed with respect to the d-graph. The proof proceeds in three parts. They are:

(i) equivalence of initial configurations of MCGA and CGA;

(ii) simulation of transitions of MCGA by CGA;

(iii) equivalence of terminal configurations of CGA and MCGA.

First, we shall see how the initial configuration of the MCGA M_1 is achievable in an equivalent manner by the CGA M_2.

(i) Equivalence of the initial configurations.

The initial configuration of the MCGA M_1 consists of the node and edge automata in their initial states that correspond to certain labels on the nodes and edges respectively of the input d-graph Γ. We can build a corresponding initial configuration for the CGA $M_2 = (\Gamma, M_{2\,N}, H)$, where the FSA on the nodes carry the labels of the d incident edges apart from their node labels. Thus, the label of each node in the input d-graph Γ occurs in the state of both the end nodes' automata. For example, consider a 3-graph ($d = 3$ for all the non-dummy nodes) that is specified as input to MCGA M_1 and accepted by it. (Note that all the dummy nodes and edges bear a label "#", indicating that the FSA on those edges and nodes do not change their state at any transition step.) The initial state of an *NA* in M_1 is of the form (p, p, p) where p is the label of the node. The initial state of an *EA* is $[(a, q_{0a}), q_{0m}, q_{0n}]$ where a is the label of the edge. Correspondingly, in the case of the CGA M_2, the

initial state of a typical NA is

$$((p, (a, q_{0a})), (p, (b, q_{0b})), (p, (c, q_{0c}))).$$

Thus, each NA of the CGA M_2 incorporates in its i-th component whatever is found in the first component of the 3-tuple of the corresponding i-th incident EA. The second and third components of the EA of M_1 are not included in the NA of M_2 as each NA of M_2 can directly sense the state of its d neighbouring NA.

(ii) Simulation of the transition step.

The transition step of the MCGA involves two phases. In the first phase, the EA undergoes a transition that involves.

(a) copying the states of its end nodes into its second and third components, and

(b) an updation (if any) of the second part of the 2-tuple placed in its first component.

Now, each NA of the CGA M_2 also has two transition steps that correspond to a single step of the MCGA M_1. The first in such a pair of transitions will correspond to the first phase of the M_1's transition. In this step, the 2-tuples placed in each of the components of the NA's d-tuple (that correspond to the first components of the EA's state in M_1) undergo a change in exactly the same manner as the corresponding EA in M_1. In the next step, the other parts of the d-tuple states of the node automata that represent the NA's state in M_1 undergo a transition based on the 2-tuples in the same manner as the NA of M_1.

For example, if δ_{NA} and δ_{EA} are the transition functions of the NA and EA respectively of the MCGA M_1, then their actions can be simulated in two transition steps by the δ of the CGA M_2's NA as follows:

The state of each NA also has a component (apart from the regular d-tuple) that toggles with respect to time to indicate which part of its state (*i. e.*, either the part corresponding to the EA of M_1 or that of the NA of M_1) is to undergo a change at that time step. Assume that if this component has a value zero, then, at that time step, the execution of the transition function results in a change in that part of the NA's (of M_2) state that corresponds to the d incident EA of the MCGA M_1. This change is actually accomplished by executing the δ_{EA} of M_1 on all the d 2-tuples found in NA's (of M_2) state. In the next time step, the toggle variable assumes a value one and hence in this time step, the NA of the CGA M_2 directly executes the δ_{NA} of M_1 on that part of its state that corresponds to the NA of M_1.

(iii) Equivalence of terminal configurations.

It can be easily seen that once: (*a*) the initial configuration, and (*b*) the individual transition steps of the MCGA M_1 are simulated in an equivalent manner by the CGA M_2, then, whenever the distinguished node (with a special label $D \in L_N$) of the MCGA M_1 enters a final state at an instant "t" then the corresponding NA of the CGA M_2 also enters its final state at a time instant "$2t$".

Thus, the CGA $M_2 = (\Gamma, M, H)$ accepts the input d-graph Γ. Since this holds good for any $\Gamma \in L_{\text{MCGA}}$, we state that for a given d-graph language L_{MCGA} accepted by a class of MCGA $C(M_{1N}, M_{1E})$ determined by the pair of FSA M_{1N} and M_{1E} placed on the nodes and edges respectively of the input d-graphs, then there exists a corresponding class of CGA $C(M_{2N})$ determined by the node automaton M_{2N} that also accepts the same language L_{MCGA}.

In the following Lemma, the class of CGA determined by the FSA on the nodes M_{1N} is referred to as $C(M_{1N})$. Also, the class of MCGA determined by the pair of FSA M_{2N} and M_{2E} placed on the nodes and edges respectively of the input d-graph is referred to as $C(M_{2N}, M_{2E})$.

LEMMA 3.2: *Given a d-graph language* L_{CGA} *accepted by* $C(M_{1N})$, *there exists a corresponding* $C(M_{2N}, M_{2E})$ *that accepts the same language* L_{CGA}.

Proof: Consider an arbitrary d-graph $\Gamma \in L_{\text{CGA}}$. The CGA $M_1 = (\Gamma, M_{1N}, H)$ and the corresponding MCGA $M_2 = (\Gamma, M_{2N}, M_{EA}, H_{NA}, H_{EA})$ are formed with respect to this d-graph Γ. In line with the proof of Lemma 3.1, we proceed by showing how the (i) initial configuration, (ii) the transitions, and (iii) terminal configuration of the CGA M_1 are simulated by the MCGA M_2.

(i) Equivalence of the initial configurations.

In the case of the CGA M_1, the initial configuration consists of each NA bearing the label of its corresponding node (an element of the set of labels L, as described in Wu and Rosenfeld [3, 4]). Note that there is no label for the edges at all in the case of the d-graphs input to the CGA. Thus, what the MCGA M_2 does is to include these node labels into the corresponding NA's state and the first component of each EA's 3-tuple contains a quiescent symbol (as there is no label for the edges).

(ii) Simulation of transition steps.

At each time step, the transition of the CGA M_1 involves a change in state of each NA of M_1 depending upon the state of its d neighbours. The

MCGA simulates this action of M_1 in the two phases of its transition step as follows:

(*a*) The first component of the *EA*'s 3-tuple is always in a quiescent state and does not change with respect to time.

(*b*) In the first phase of the transition step, each *EA* of M_2 just includes the states of its two end nodes (which are the same as that of the *NA* of the CGA M_1) into its second and third components.

(*c*) In the next phase, each *NA* of M_2 takes the relevant component from the state of each of its d incident *EA* and undergoes a transition in the same way as the corresponding *NA* of the CGA M_1.

Thus, the two phases of the MCGA M_2's transition completely simulate a single transition of the CGA M_1.

(iii) Equivalence of the terminal configurations.

It can be easily seen that once (*a*) the initial configuration, and (*b*) the transition steps of the CGA M_1 are simulated in an equivalent manner by the MCGA M_2, then, whenever the *NA* with a special distinguished label $D \in L$ enters a final state in the CGA M_1, then the corresponding *NA* of the MCGA M_2 also enters its final state, accepting the same input d-graph Γ.

Since Γ is an arbitrary element of the set L_{CGA}, we can state that the class of MCGA $C(M_{2N}, M_{2E})$ determined by the pair of FSA whose actions are as described above, accepts the same d-graph language L_{CGA} that is accepted by a given class of CGA $C(M_{1N})$.

Combining the above two Lemmas, we can state our equivalence theorem as follows:

THEOREM: MCGA *and* CGA *are equivalent with respect to the acceptance of d-graph languages.*

4. DETAILS OF CERTAIN HIGH LEVEL DESCRIPTIONS

4.1. Introduction

We now give brief explanation for certain high level descriptions of actions viz., sending and receiving signals, waiting for a signal, etc., in terms of the actual transitions of the node automaton (the action of the edge automaton is similar). Since states are represented as tuples, we see that we can interpret certain components of the tuple to represent some particular kind of information. Since every node automaton has complete knowledge about the states of all the neighbouring nodes, we immediately conclude that every node

knows all the information available with its neighbours. Now, the node automaton can either ignore the information thereby blocking its propagation or move the information into its own state thereby making the information available to some other node automaton. This phenomenon is what we informally refer to as "flow of information" or "propagation of signals". This global arrangement is predetermined for that particular MCGA designed for executing the desired algorithm on an input d-graph.

4.2. Sending and receiving signals

In this case, the state of each node automaton is in the form of a 3-tuple $[q, (s'_1, \ldots, s'_d), (s''_1, \ldots, s''_d)]$ where the second and third components consist of d signals to be sent to and received from d neighbours respectively. Each signal is of the form "name (arg)" where "arg" is the (optional) information to be communicated to various nodes. For sending a signal S to the i-th neighbour, the node automaton n places it on the i-th position of the second component of its state. The corresponding edge automaton, during its phase of operation, removes S from that position and places it in its own state. The node automaton at the other end then finally transfers the contents of the state of the edge automaton to the k-th position of the third component of its own state (where n is its k-th neighbour). Thus, sending and receiving signals involves nothing but transferring certain predefined positions of the state from a node automaton to an edge automaton and then from the edge automaton to the other adjacent node automaton.

4.3. Storing son and father nodes

For a given node, in order to remember which one of its neighbours is its father and which are its sons, it maintains an additional d-tuple $(s_1, \ldots, s_d)$ where s_i is equal to

(i) F, if the i-th neighbour is its father,

(ii) S, if the i-th neighbour is its son,

(iii) N, if the i-th neighbour is a non-special neighbour.

(This additional d-tuple is maintained as part of the first component.)

4.4. Waiting for a signal

This is achieved by entering a state $q_{\text{wait (name)}}$ where "name" is the id of the signal which the node automaton is waiting to receive. The node automaton performs no action as long as the "name" is not available in the third

component of its state. It scans that d-tuple at every time step to determine whether "name" is present. If yes, it gets into $q_{\text{wait (over)}}$ state and performs the required actions. $q_{\text{wait (name)}}$ is kept as part of the first component.

4.5. Relaying a signal

Section 4.1 described how a signal is transferred from one node to another. Now, we shall briefly describe the specific actions of the node automata that result in the traversal of a signal over a path containing more than one edge. This action is referred to as "relaying". For relaying a signal without doing or undergoing any modifications, the node automaton just transfers that signal from the d-tuple in its third component to the requisite positions of the d-tuple in the second component of its state that correspond to the neighbours that signal is to be sent to. This action is carried out at each of the intermediate nodes of the path traversed by the signal that results in its relaying.

5. ALGORITHMS

5.1. Introduction

It may be recalled here that section 2 presented the definition of our MCGA and the language it accepts. Section 3 established the equivalence of MCGA and CGA. It is evident that the size of the sets Q_{NA} (the set of states of the node automaton) and Q_{EA} (the set of states of the edge automaton) determine the amount of "memory" associated with those automata. When we make use of the cellular graph automata consisting of these two kinds of FSA to solve certain graph theoretic problems (as in Wu and Rosenfeld [3, 4]), it is preferable to have the "memory" (or, the number of distinct states in the sets Q_{NA} and Q_{EA}) of the automata to be proportional only to the degree "d" of the input d-graphs. It can be easily seen that such an algorithm that requires the FSA to have memory size that is function of only "d" will facilitate a given MCGA (designed for a particular "d") to be used on an input d-graph of any size to obtain a solution. (Here "size" refers to the area *i.e.*, the total number of nodes in the input d-graph). This kind of space complexity is easily achieved for certain graph problems like obtaining the area of the input d-graph, finding the radius, etc. The algorithms for solving these problems are given in Wu and Rosenfeld [3, 4]. However, for certain other more complex graph problems, especially those like the weighted graph problems that are considered in this paper, it has not been possible to

achieve a space complexity that is independent of the area (total number of nodes) of the input d-graph. This may be due to the inherent complexity of the problem. Also, if one tried to limit the memory of the automata to be proportional only to "d", it was found that the number of time steps required for the termination of such algorithms was either exponential with respect to the area or a polynomial of a high degree. Hence, it has been found more reasonable to assume that the memory of the FSA involved is also proportional to the area of the input d-graph, actually a small function of it [e. g., memory log (area)]. This has also been suggested by Wu and Rosenfeld [3, 4]. It is to be also noted that in our case of weighted graph problems, the edge automata should have their memory size that is proportional to the largest weight possible in the input d-graph.

In this section, we present four algorithms for solving certain graph theoretic problems viz., distinct labeling of all the nodes of an input d-graph, distinct labeling of all the edges of a weighted d-graph based on a global sorting of their weights, minimum weight spanning tree construction and single source shortest path for weighted d-graph, in that same order. Throughout this section, the edge automaton will be abbreviated as EA and the node automaton as NA.

5.2. Distinct labeling of all nodes of a d-graph

5.2.1. *The algorithm*

In this section, we present an algorithm on our MCGA that assigns a distinct label for all the nodes in the input d-graph Γ.

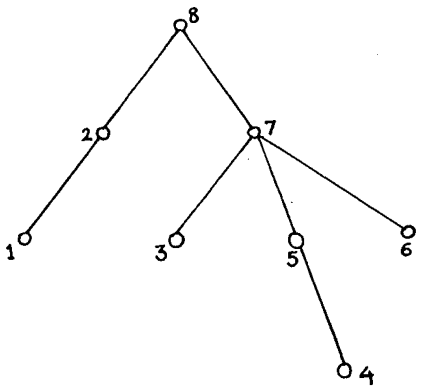
The main idea behind this algorithm is the numbering of the nodes based on a postorder traversal of a breadth first spanning tree of the input d-graph.

We briefly give the principle behind the algorithm before presenting the actual decentralised version. Given an arbitrary tree in which the internal nodes may have none, one or any number of sons, the postorder traversal of that tree is defined as follows:

(i) visit the subtrees from the one with the least sequence number to the one with the highest (the sequence number of a subtree, in the case of the d-graphs that we will be concerned with, is the same as the arc end number of the corresponding edge and it lies between 1 and d);

(ii) visit the root node.

Note that the above rules of traversal are to be applied recursively in the tree. Consider as an example the tree shown in the figure 2 whose post-order numbering of all the nodes is also given. With respect to the figure, it can be



noted that 2 is the immediate left cousin of 7 (in this case it is brother straightaway) and 3 is the immediate cousin of 4, etc. Also note that the total number of nodes in the subtree rooted at the node 7 is equel to 5 whereas it is 1 for the node 4. Thus, it can be easily seen that given any arbitrary tree, the postorder number of any node in that tree is equal to the sum of the total number of nodes in the subtree rooted at that node and the post order number of the immediate "left" cousin of this node. Thus, the main purpose of this algorithm is to make available these two informations to each of the nodes in the BFST so that they can label themselves with the sum of these two numbers. The BFST is constructed during the forward pass of the "CONSTRUCT" signal. Each node records the total number of nodes in its subtree during the backward pass of the "REPLY" signals. After the "REPLY" signals reach the root node, it initiates transmission of "LABEL" signals towards the leaves which inform each node of the postorder number of its immediate left cousin. Thus, this algorithm leads to correct numbering of nodes.

The algorithm is as follows:

(1) The distinguished node D sends a "CONSTRUCT" signal to all its neighbours.

(2) A node receiving one or more "CONSTRUCT" signals simultaneously marks the sender with the lowest arc end number as its father and then sends a CONSTRUCT signal to all its other neighbours. It ignores any subsequent receipts of CONSTRUCT signals. It sends a REPLY signal to its father node so that it can note the arc end leading to its son node.

(3) A node that receives no "REPLY" signals (leafnode) sends a COUNT (*O*) signal to its father.

(4) An *EA* receiving a COUNT (*s*) signal from one end node adds one to *s* and sends COUNT $(s+1)$ to its other end node, after noting down $(s+1)$ in its state as SUBNO.

(5) A node having more than one son waits still it gets COUNT signals from all its sons, adds up all the arguments and sends the sum in a COUNT (*s*) signal to its father.

(6) A node having K sons maintains a K-tuple of the form $\left(0, S_1, S_1+S_2, \ldots, \sum_{i=1}^{k-1} S_i\right)$ where S_i is the argument of the "COUNT" signal received from its i-th son. It also maintains the sum of the arguments of all the COUNT signals received from its sons, as BASENO $= \sum_{i=1}^{k} s_i + 1$.

(The case $K=0$ will occur at every leaf node of the spanning tree. In such a case, the leaf nodes do not maintain any K-tuple and assign a value of one to their BASENO.)

(7) The root node equates its own label named NODENO to its BASENO.

(8) The root node, after labeling itself sends its entries in the K-tuple, to the corresponding sons in a LABEL (c_i) signal (where c_i is the i-th component in the k-tuple corresponding to its i-th sons.

(9) An *EA* receiving a LABEL (c_i) signal, adds its SUBNO value to c_i before transmitting to the other node.

(10) An internal node receiving a LABEL (*c*) signal subtracts its BASENO from *c*, adds the resultant to each entry in its own k-tuple and then sends the final resultant to each of its corresponding sons in a set of LABEL signals. It sets its own NODENO as "*c*".

(11) A leaf node sends a COMPLETED signal to its father node after determining its NODENO.

(12) An internal node sends a COMPLETED signal to its father only after receiving COMPLETED signals from all its sons.

(13) Whenever the root node receives COMPLETED signals from all its sons, it terminates the process.

It can be seen that steps (1)-(2) construct the rooted tree and store father and sons.

(3)-(7) enter in the memory of a node the number of nodes of the subtree.

(8)-(10) set up the post order numbering.

(11)-(13) terminate the process.

5.2.2. *Complexity analysis*

(*a*) Time steps

The worst case number of time steps required for this algorithm to terminate = height of the tallest BFST in the given graph = diameter of the graph = $O(n)$ (where n = total number of nodes).

It can be seen that worst case rarely occurs and the diameter is mostly a fraction of "n". Hence this algorithm is better than the one suggested by Wu and Rosenfeld [3] where distinct labeling is done using the order of visiting the nodes in a DFST construction. This was found to take always $2n-2$ time steps.

(*b*) Space requirement

As is evident, each EA requires $O(\log n)$ memory while each NA requires $O(d\log n) = O(\log n)$ memory only.

From the foregoing discussion, it can be stated that, THEOREM. There exists a Modified Cellular Graph Automaton denoted as $A_{d,n}$ that distinctly labels all the nodes of a class of graphs that have bounded degree d and area n, in O(diameter) time steps, each node having a memory size $O(d\log n)$.

5.3. Global sorting of edges in a weighted graph

5.3.1. *The algorithm*

We now give an algorithm that assigns consecutive numbers to the weighted edges (non-negative in value, duplicate weights are allowed) according to their position in an imaginary global list formed by sorting the edges according to their weights in an ascending order. The weights are assumed to be non-negative. Two or more edges are allowed to have the same weight.

Here, an imaginary globally sorted list of the weighted edges is assumed to exist. Each edge initially assumes that it forms the first element in that list. But whenever it comes to know of an edge whose weight is less than its own, it shifts its position up by one and so on till it has examined the weights

of all the edges in the graph. The following method is used to inform every edge automaton in Γ about the weight of every other edge automaton that has a non-negative weight. Note that the edge automata and node automata bearing the label "#" do not take part in the algorithm. Each *EA* depending on a certain decision criterion, assigns itself to one of the two end nodes. A particular *NA* knows which subset of its d incdient edge automata have assigned themselves to it. Based on this information, it forms a list (consisting of utmost d weights) that correspond to those edge automata that have associated themselves with this *NA*. Such a packet of edge weights formed at each of the n nodes of the input graph, are transmitted through the branches of the BFST so that a packet formed at a node reaches all the other nodes of the BFST which in turn inform their associated edges about the packets whenever they arrive. A node does not receive the same packet twice because they are transmitted through the circuit-free BFST. Whenever a node receives one or more such packets at an instant, it transmits them to its associated edges, thus enabling every *EA* of the graph to know about the weight of every other *EA*. Then, an *EA* may perform an incrementing operation on its label for every entry that it finds in the packets which is less than its own weight. A node knows that it has received packets from all the nodes of the graph whenever it finds that at a particular instant, it has not received a single packet from any of its BFST neighbours. This is because packets have been transmitted to the node through the BFST tree. At that instant, all the edges assigned to that node will carry the correct rank number in a global sorted list. It accordingly informs its associated edges that termination has occured.

(1) Initially, a BFST is constructed and all the nodes are distinctly labeled using the algorithm in section 5.2.

(2) At the first step after this node labeling is over, each *EA* assigns itself to one or the other of its two end nodes as follows:

(i) If it carries an odd numbered weight, it assigns itself to the end node having an integer label that is smaller of the two end nodes.

(ii) If it carries an even numbered weight, it assigns itself to the end node with the higher label value. This is to uniquely assign an edge to a node.

(3) Each *EA* maintains an integer variable SORTNO which is initialised to 1.

(4) After assigning itself to one of its end nodes, each *EA* sends its weight w to that node in an INITSORT (w) signal.

(5) A *NA* receiving a set of INITSORT signals notes the neighbours from which it received them as its assigned edges, orders the weights in a list according to their neighbourhood number (*i.e.*, of the form $(w_1, w_2, \ldots, w_d)$ where w_i is -1 if the i-th incident edge is not assigned to it). This d-tuple is the packet formed at that node which will be sent to all the other nodes through the branches of the BFST.

(6) Each *NA* initially sends the first list made of the INITSORT signals to all its assigned edges in a SORTOUT signal. It also sends this list to its father and all its son nodes in a TRANSMIT $(n, (w_1, \ldots, w_d))$ signal (where n is its own id).

(7) An *EA* receiving a TRANSMIT signal just transmits it to the other end node.

(8) An *EA* receiving a set of lists in a SORTOUT $((n_1, (w_{11}, \ldots, w_{1d})), (n_2, (w_{21}, \ldots, w_{2d})), \ldots, (n_k, (w_{k1}, \ldots, w_{kd}))$ signal, scans the whole list and increments its SORTNO by 1 for every weight w_{ij} which is less than its own. If a weight happens to be equal to its own, it increments SORTNO only if the corresponding node is less than its own assigned node. (For the first SORTOUT signal, if the weights are the same, the *EA* with the higher neighbourhood number only increments its SORTNO.) The *EA* does not perform the incrementing its SORTNO.) The *EA* does not perform the incrementing action for all other cases. k can be at most d and the maximum value of w_{ij} is the maximum of the edge weights.

(9) A *NA* receiving a set of "TRANSMIT" signals retransmits each of them to its tree neighbours. This is done in such a manner as to avoid the same neighbour node that sent this signal from receiving it again (a sort of "reflection" of the TRANSMIT signal). This could be achieved by sending the node identity along with a transmit signal. It also sends all of the received packets (d-tuples) in a SORTOUT signal to its associated edge automata.

(10) A *NA* terminates the process locally by sending a "FINISH" signal to all its associated nodes at an instant after the one in which it finds that it has not received a single TRANSMIT signal from any of its BFST neighbours.

(11) An *EA* receiving a FINISH signal knows that it has examined the weights of all the edges in the graph and hence the value of SORTNO at that instant gives the required distinct label to that edge that corresponds to its own position in a globally sorted list of edges arranged in an ascending order of their weights.

It should be noted that a node sends the TRANSMIT signal to all its BFST neighbours except the one from which it was received. But it sends the SORTOUT signal only to its associated edge automata.

It can be seen that steps (1)-(3) consist of initialisation, (4)-(9) describe how the flow of weights is transmitted to each edge automaton and (10)-(11) terminate the process.

5.3.2. *Complexity analysis*

(*a*) Time steps

The initial node labeling algorithm (see section 5.2) requires 0 (diameter) time steps. The worst case time required for subsequent sorting = time required by a TRANSMIT signal to reach all the nodes = 0 (diameter).

Hence, the worst case time complexity of the whole algorithm = 0 (diameter).

(*b*) Space requirement

(i) Message Length: worst case measure of message length = length of the longest SORTOUT or TRANSMIT signal = $d \times (d+1) \times \log \max (w_{\max}, n)$ (where $w_{\max}$ is the largest of the edge weights).

(ii) Memory: each *EA* needs space to store the message to be transmitted. Hence memory requirement is $0(d^2 \log \max (w_{\max}, n))$. It will also be the memory requirement for node automata. From the foregoing discussion, we can state that

THEOREM: *There exists a* MCGA $A_{d,n}$ *that labels all the edges of the class of weighted graphs that have bounded degree d and area n, with distinct integers that correspond to their position in a global list formed by sorting the edges according to their weights in an ascending order in O (diameter) time steps, each node and edge automaton having a memory O* $(d^2 \log \max (w_{\max}, n))$.

5.4. Construction of the minimum weight spanning tree

5.4.1. *The algorithm*

We now present an algorithm that makes use of the global sorting of edges achieved in the previous algorithm to construct the MWST for a weighted graph in which all the edges bear positive integral weights. This algorithm is based on the sequential algorithm due to Kruskal [5].

We first briefly describe Kruskal's algorithm for the construction of MWST of a weighted graph that serves as the basis for our distributed algorithm on

the MCGA. Kruskal's algorithm consists of the following steps:

(*a*) Sort the edges of the weighted graph to form a list in an ascending order.

(*b*) Examine the i-th edge occuring in this list (only after examining all the previous $i-1$ edges) as to whether it can be included in the MWST or not. The i-th edge can be included as a branch in the MWST only if either one or both the end nodes of this edge are not yet included into the MWST or if both the end nodes are already included into the MWST then this edge can be included if it does not form a fundamental circuit with the existing branches of the MWST. Otherwise, if that edge forms a circuit with some branches of the current MWST, then its weight will be greater than all the other edges (branches) in the fundamental circuit and hence it will become a chord of the MWST. Thus the complete spanning tree would have been constructed as soon as all the edges in the weighted graph have been examined. In our algorithm, the position number of an edge in a global list formed by sorting the weighted edges in the graph in an ascending order is stored in the corresponding edge automaton itself using the algorithm in section 5.3. Following Kruskal's algorithm closely, a node whose position number is i should examine whether it can be included into MWST or not only after the fate (branch or chord) of all the previous $(i-i)$ edges have been decided. In case one or both the end nodes of an edge automaton are not yet included into the MWST, it takes only one time step to decide about the inclusion of that edge into the MWST. However, in our distributed case (as against the sequential algorithm of Kruskal), it will take (in the worst case) n time steps to decide whether an edge whose both end nodes have been already included into the MWST, can become a branch of the MWST or not. This is because, it takes, in the worst case, $(n-1)$ time steps to know whether a circuit is formed with existing branches of the MWST wherein all such branches have a weight less than its own. In that case, this edge is not included as a branch in the MWST. If we were to strictly follow Kruskal's sequential algorithm, then an i-th edge automaton can start examining whether it can include itself into the MWST or not only after $(n-1)\times(i-1)$ time steps from the start of the algorithm. This means that the algorithm will take $O(e\times n)$ time steps. In order to reduce this, we have specified in our algorithm that an i-th edge automaton automatically starts examining at the i-th instant itself without waiting for all its previous $(i-1)$ edges to decide about their inclusion. Whenever an i-th edge automaton finds that both its end nodes are already in the MWST, then it sends a special signal to search for any fundamental circuit formed by it and hence waits for the return of this signal for n time

steps. While this edge automaton is in the "wait" state, the next edge automaton labeled $(i+1)$ will start its examination at the $(i+1)$-th instant itself. If it also finds that both its end nodes are already included in the MWST, then it also sends its own circuit search signal. Thus, while the time complexity is reduced by this method, the number of circuit search signals increases. However, one can see that the correctness of the algorithm is not hampered by this modification of Kruskal's algorithm. We now present the actual algorithm.

(0) All the edges are distinctly labeled according to their weights using the algorithm in section 5.3.

(1) The distinguished node having label n synchronises the initiation of the process by communicating to each of the *EA* the time instant at which they have to all start the TIME counter through a SYNC(n) signal [where n is the total number of nodes in $U(\Gamma)$]. The argument of SYNC signal decrements at each time instant.

(2) Each *EA* starts incrementing the time counter at an instant t time steps after its first receipt of a SYNC(t) signal.

(3) An *EA* with SORTNO as i waits till its TIME counter reaches i. It then determines the state of its adjacent nodes. If one or both the end nodes are still having their INCLUDED variables not set to 1, it sets its BRANCH to 1 and sets the INCLUDED of both the nodes to 1. Otherwise, if both the nodes are already included, it sends a CIRCUIT-FIND (i) signal to the node with lower id (note that all the nodes are distinctly labeled initially using the algorithm given in section 5.2). It then goes into a WAIT state for n more steps.

(4) A *NA* receiving a CIRCUIT-FIND signal sends it to all the neighbours other than the one from which it received.

(5) An *EA* receiving a CIRCUIT-FIND (i) signal sees whether i is greater than its own SORTNO. If so, it transmits it to the other end node. Otherwise, it blocks that signal from propagating.

(6) An *EA* in its WAIT state, receiving its own CIRCUIT-FIND (i) signal through the other end node, sets itself as a CHORD of the MWST.

(7) An *EA* that has "WAIT" ed for n time steps without getting back its CIRCUIT-FIND signal, sets itself as a BRANCH of the MWST. This can be achieved by having the *EA* count upto n by having a counter as part of the state.

(8) The algorithm terminates automatically when the e-th edge determines whether it belongs to the tree or not, which in the worst case will take $(e+n)$ time steps from the instant the TIME counter was started at every EA.

It can be easily seen that steps (0)-(2) initialize the process, (3)-(7) describe the algorithm, and (8) terminates the process.

5.4.2. *Complexity analysis*

(a) Time steps.

As noted in step (8) above, the worst case time complexity $= O(e+n) = O(dn) = O(n)$.

(b) Space requirement.

After determining its SORTNO, the EA does not even need to store its weight in its memory and hence the worst case space requirement $= O(\log(\text{SORTNO})_{\max}) = O(\log(dn)) = O(\log n)$. But initially finding SORTNO requires each edge automaton to have more memory.

From the foregoing discussion, we can state that, THEOREM. There exists a MCGA $A_{d,n}$ that constructs the minimum weight spanning tree for the class of weighted graphs with bounded degree d and area n, in $O(n)$ time steps, each node and edge automaton having a memory of $O(d^2 \log \max(w_{\max}, n))$.

5.5. Single source shortest paths

5.5.1. *The algorithm*

This algorithm for the MCGA is based on the Dijkstra's [6] sequential algorithm for finding shortest paths from a given node to all other nodes in a weighted graph whose edge weights are positive integers.

We shall first briefly describe the sequential algorithm due to Dijkstra. Initially all the nodes are assigned with ∞ as their temporary label except the single source node that bears a permanent label of zero. In the next step, all the nodes that are neighbours to the single source node alter their temporary labels such that they are equal to the distance value in the corresponding incident edges. The minimum of all the existing temporary labels is found out and the node with such a minimum is given the next permanent label. Subsequently, new distance values are calculated for all the nodes that are neighbours of the last node that was given a permanent label. If the new distance value is less than the existing temporary label of a node, then its temporary label is altered to that distance value. A global minimum is again obtained to find the next node with the permanent label. Ultimately, all the nodes will get their permanent label that is nothing but the value of

the shortest distance between that node and the single source node. We now give our distributed algorithm on the MCGA that is based on this sequential algorithm.

(1) After labeling themselves distinctly using the algorithm in section 5.2, each *NA* maintains an integer variable CURRENTMIN initialised to ∞.

(2) The distinguished node *D* (from which the paths have to be determined) starts the process by setting its CURRENTMIN to zero and sending an UPDATE(0) to all its neighbours.

(3) An *EA* receiving an UPDATE(*i*) signal adds its own weight to the argument *i* and then transmits it to the other node.

(4) An *NA* reveiving a set of UPDATE (*i*) signals from its neighbours for the first time, sets its CURRENTMIN to the minimum of the values received and sets its UPSTREAM-NODE to the corresponding sender. Depending upon which node sends the UPDATE signal which modifies the CURRENTMIN value, the value of UPSTREAM-NODE is changed. The finer details are left out. Ties are resolved in favour of the lowest arc end number. The purpose of storing the variable UPSTREAM-NODE is to indicate the path that corresponds to the current shortest distance value. Ultimately, at the end of radius (*D*) time steps, when all the nodes have obtained their shortest paths to the source node, it is this variable that will serve to indicate the actual path to be traced. As mentioned in Deo [6], it can be seen that the union of all the shortest paths is an arborescence rooted at the source node and hence the value of UPSTREAM-NODE at a given node at the end of the algorithm will also correspond to its farther node in the arborescence.

(5) Any subsequent receipt of UPDATE (*i*) signals by a *NA* results in the following actions:

(i) It finds the minimum of all the distance values received.

(ii) If that minimum is greater than or equal to the CURRENTMIN stored in its state, it ignores all the signals and does not retransmit any of them.

(iii) If that minimum is less than the CURRENTMIN, it resets the value to the new minimum and also alters the UPSTREAM-NODE accordingly. It also retransmits the new minimum to all its neighbours in an UPDATE (s_{min}) signal.

(6) The distinguished node terminates the algorithm L_c time steps after it had initiated the process (where L_c = length of the longest chain in the graph from the distinguished node *D*, which can be safely assumed to be *n*, the total number of vertices in the graph).

It can be seen that step (1) initialises the process, steps (2)-(5) describe the algorithm, and step (6) terminates the process.

5.5.2. *Complexity analysis*

(*a*) Time steps.

As noted in step (6) above, the worst case time complexity of the above algorithm $= O(n)$.

(*b*) Space requirement.

Each *EA* needs $O(\log w_{max})$ size memory and each *NA* needs $O(\log s_{max})$ size memory where

w_{max} = maximum of the weights of all the edges,

s_{max} = largest possible distance value in the graph.

We shall state the above mentioned results in the form of a theorem and give an indication of the proof of correctness of our algorithm.

THEOREM: *There exists a* MCGA $A_{d,n}$ *that finds the single source shortest paths for the class of weighted graphs that have bounded degree d and area n, in* $O(n)$ *time steps, with each EA having a memory* $O(\log w_{max})$ *and each NA having a memory* $O(\log s_{max})$ *where* w_{max} *and* s_{max} *are as defined above.*

Proof: The value of the variable CURRENTMIN maintained at every node at any instant corresponds to the shortest distance value communicated to that node upto that instant. If the shortest path from a node to the source node consists of k edges, then it takes k time steps for the UPDATE signal to reach that node that will result in CURRENTMIN assuming its minimum possible value. Thus, the CURRENTMIN values always go on decreasing with respect to time till they attain a value that corresponds to the shortest distance to the source node. Subsequent receipt of UPDATE signals will not affect its value as they will always be greater than this minimum.

6. CONCLUSIONS

We have now seen various algorithms for solving problems concerning weighted graphs on the Modified Cellular Graph Automaton. The basic algorithm for labeling of all nodes distinctly has time complexity of O (diameter) which is faster than the labeling of the nodes with a depth first numbering using the DFST algorithm of Wu and Rosenfeld whose time complexity is O (area). An MCGA algorithm based on Prim's [6] sequential algorithm for MWST construction on the MCGA was found to require

$O(n^2)$ time steps. But the algorithm given in this paper is based on Kruskal's [5] sequential algorithm for MWST construction and takes only $O(e) = O(n)$ time steps. This algorithm requires the edges to be in sorted order with respect to their weights which was done on the MCGA using the algorithm in section 5.3 that also takes $O(n)$ time. This one on signle source shortest paths (based on Djikstra's [6] sequential algorithm) that takes $O(L_c)$ time illustrates the ease of solving weighted graph problems on the MCGA. As for the future work, it would be interesting to devise algorithms on this MCGA for more weighted graph problems viz., Travelling Salesman Problem, maximal weighted matchings, etc., and analyse their time requirements.

REFERENCES

1. A. R. Smith, *Cellular Automata and Formal Languages, in* Proceedings, 11th SWAT, Vol. III, 1970, pp. 216-224.
2. P. Rosentiehl, J. R. Fiksel and A. Holliger, *Intelligent Graphs: Networks of Finite Automata capable of solving Graph Problems, in* Graph Theory and Computing, R. C. Read Ed., 1972, pp. 219-265, Academic Press, New York.
3. A. Wu and A. Rosenfeld, *Cellular Graph Automata I,* Information and Control, Vol. 42, 1979, pp. 305-329.
4. A. Wu and A. Rosenfeld, *Cellular Graph Automata II,* Information and Control, Vol. 42, 1979, pp. 330-353.
5. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Design and Analysis of Computer Algorithms,* Addison Wesley, 1974.
6. N. Deo, *Graph Theory with Applications to Engineering and Computer Sciences,* Prentice Hall, 1974.
7. A. Rosenfeld *et al., Sequential and Cellular Graph Automata,* Journal of Information Sciences, 1980.
8. J. Mylopoulos, *On the relation of Graph Grammars and Graph Automata, in* Proceedings, 13th SWAT, 1972, pp. 108-120.