

An amorphous algorithm for learning ordered sequences: memorizing the multiplication table

Panayotis A. Skordos

PAS@AI.MIT.EDU

*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology,
Cambridge MA 02139 USA*

Abstract

An amorphous algorithm is presented that memorizes ordered sequences of tokens such as the multiplication table, telephone numbers, and arbitrary alphanumeric strings. The amorphous model consists of a set of information-processing nodes which are identical, independent, asynchronous, message-passing between neighbors, and in unspecified physical locations. For simplicity communication between neighbors is assumed to occur via wires and not via radio. The nodes absorb incoming tokens of information using time-controlled accepting and rejecting rules of behavior, and during this absorbing process the nodes make logical connections between them that have the structure of a tree. Retrieval occurs by matching all parts of a stored sequence. The amorphous program is presented, and a simulator is developed to test the program. Future plans are given how to extend the present approach to other memory tasks and more complex computations.

1. Introduction

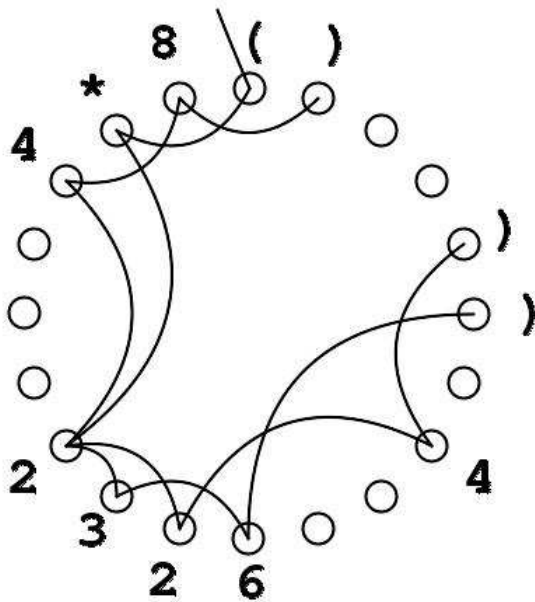
The phenomenon of emerging computation from an underlying set of independent agents is an important topic in many diverse areas of science and engineering. In particular in Computer Science the topic is very promising because processors will continue to become smaller and cheaper to produce; while at the same time the task of programming and organizing a great multitude of processors will remain a difficult task. One approach of handling an immense multitude of processors is the approach of emerging computation with independent information-processing units that are asynchronous and whose physical locations are unspecified. This is the field of amorphous computing (Abelson, Allen, Coore, Hanson, Homsy, Jr., Nagpal, Rauch, Sussman, & Weiss, 2001; Weiss, Knight, & Sussman, 2004).

Another area which is at the forefront of science, and is related somewhat to amorphous computing is understanding the human brain. In the brain a great multitude of individual neurons are involved to produce a global computation, and the cells operate asynchronously with each other. Also the locations of the neurons are not on a regular grid, but instead they are flexible, moving slightly and growing, and making new connections. Of course, the brain is much more complex than an analogy with amorphous computing, but there is an overlap between the two fields that can be useful. By developing amorphous algorithms for simple memory tasks, we will understand better the general principles involved. And in combination with other disciplines, this may help some day to understand better certain aspects of the brain's memory. In addition, the amorphous algorithms will have interesting applications on amorphous computing hardware. This is the motivation of the present study.

The task of storing and retrieving ordered sequences of names and numbers such as the multiplication table is explored. An amorphous program is presented that memorizes sequences of single-digit numbers or characters, and composite objects such as multi-digit numbers and alphanumeric strings. The task is simple and useful both on its own and as a subpart of more complex memory tasks.

Section 2 introduces the high-level amorphous model used here, and then presents the algorithm. The main ideas are how to form a tree in an amorphous manner, how to make parent-child connections between nodes, and how to exploit time-controlled accepting and rejecting rules of behavior. Section 3 describes a serial simulator that is used to test the amorphous algorithm. Sections 4 and 5 present examples of storing and retrieving ordered sequences. Section 6 discusses the connectivity between nodes, and presents the results of simulation experiments. The last section 7 presents ideas for extending the present approach to other memory tasks and more complex computations.

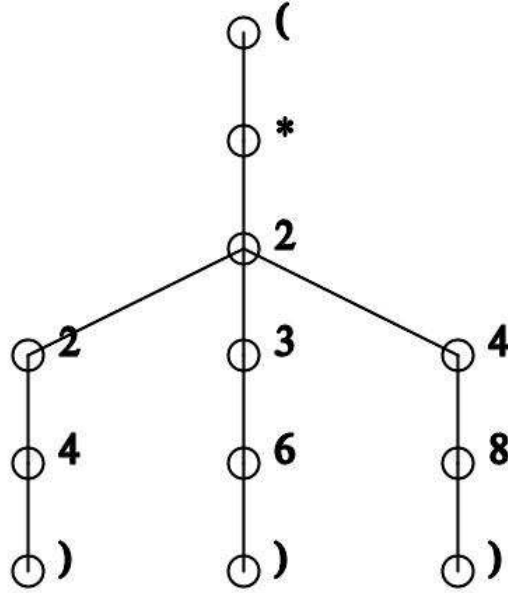
Figure 1: Twenty nodes store the sequences “(* 2 2 4)”, “(* 2 3 6)”, “(* 2 4 8)”.



2. High-level amorphous model

The present amorphous model is more abstract than the typical amorphous computer in recent works such as (Coore, 1999; Nagpal, 2001; Weiss, 2001; Rauch, 2004). In particular it is assumed that communication between amorphous processors occurs via wires and not via radio waves. This simplifies the model and the programming. However, it may be possible to build the equivalent of communication via wires on an amorphous computer that uses radio communication, by using the techniques in (Nagpal & Coore, 1998; Nagpal, 1999) to

Figure 2: Tree of sequences “(* 2 2 4)”, “(* 2 3 6)”, “(* 2 4 8)” of figure 1.

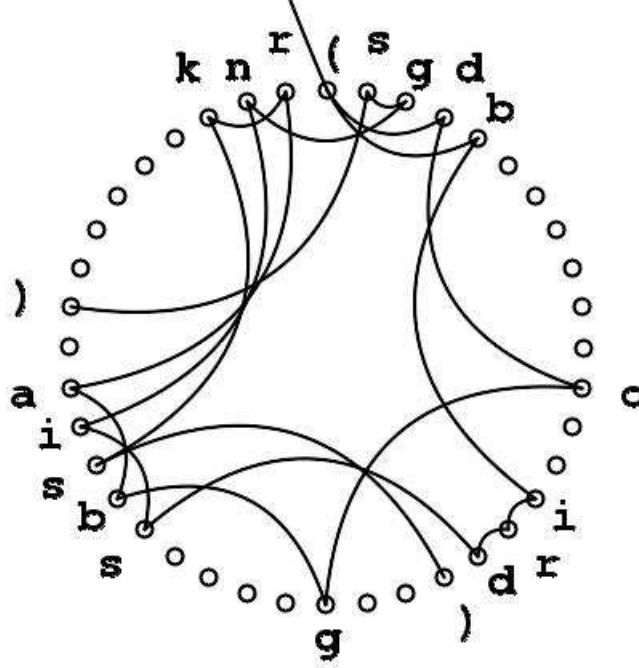


assign unique local identifiers to nearby amorphous processors or group of processors. Also, the techniques in (Nagpal & Coore, 1998) for grouping a set of processors into one node can give robustness against processor failure.

We start with a set of independent nodes that execute locally the same program without global synchronization. The nodes communicate asynchronously with each other via messages sent over wires, but the precise location of the nodes is left unspecified. Every node has a fixed set of neighbors that it communicates with, and together all the nodes constitute a graph. Information enters and exits the graph through one boundary node, which has an extra connection to the outside world. A node typically sends the same message to all its neighbors, but it can also communicate selectively with one neighbor if needed, such as its parent. The assumption of communication via wires implies that a node can talk selectively with a neighbor if needed, and also that there are no issues of message collision.

Neighboring nodes make logical connections between them in order to store information. When two nodes make a connection, one node becomes the parent (upstream) and the other node becomes the child (downstream). A typical scenario is as follows. If a node has already stored a token of information in its memory, and receives a new token from its parent, then it tries to make a connection with one of its free neighbors. The neighbor that accepts the connection, becomes its child and absorbs the new token. If a node has used all of its free neighbors and can not make a connection, then there is a bottleneck (section 6), and the information can be lost. Therefore the total number of nodes and the number of neighbors per node must be large enough so that there are always free nodes to connect to.

Figure 3: Forty nodes store the sequences “(dog barks)” and “(bird sings)”.

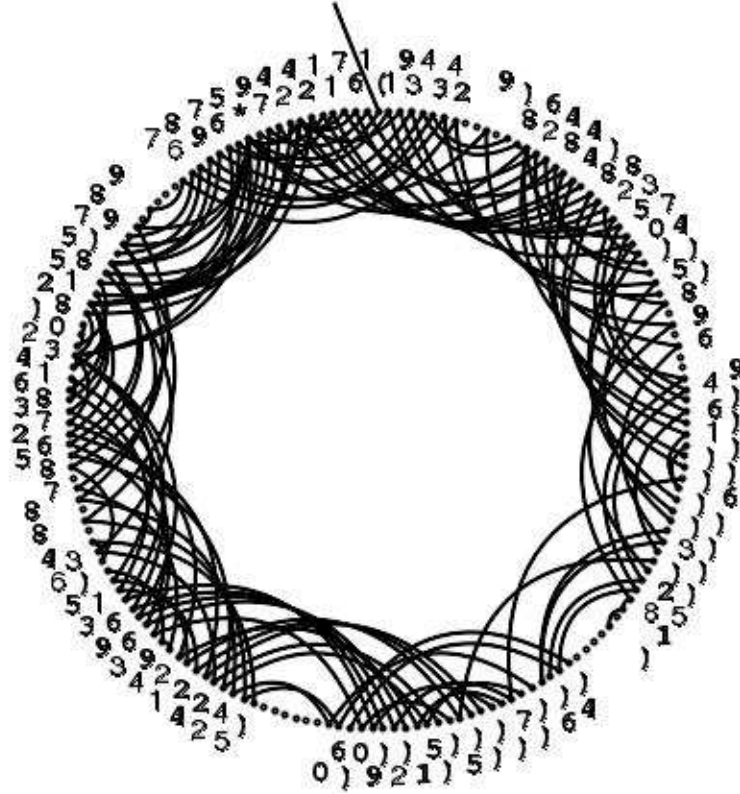


In a different version of the model, we could try to include dynamic creation of new neighbors in order to achieve a better utilization of the nodes and the connectivity of each node. Connectivity is how many free neighbors a node has available to form new connections. Dynamic creation of new neighbors would be useful if some nodes require many more neighbors than the average node. Also, another addition to the model would be the migration of nodes and their connections in order to relieve congestion from busy areas. For simplicity we have chosen to avoid these features in the present study. And it turns out that good results are achieved in the task of storing and retrieving sequences such as the multiplication table by using a large number of simple nodes with a large number of available neighbors for each node. Specific simulation numbers are given in section 6.

2.1 Trees created by amorphous program

To store and to retrieve ordered sequences of tokens, it is convenient to use the structure of a tree. The challenge is to program the creation and the searching of a tree in an amorphous manner. The basic idea that makes this possible in the following amorphous program, is the time-controlled accepting and rejecting of messages by the nodes in order to absorb incoming tokens appropriately and to direct the tokens to the correct paths. In addition, the connection between two nodes distinguishes the nodes into a parent and a child, and this leads naturally to the structure of a tree.

Figure 4: A graph of 175 nodes with connectivity 52 neighbors per node stores the complete multiplication table of 36 sequences from “(* 2 2 4)”, “(* 2 3 6)” up to “(* 9 9 81)”.



Consider a graph of nodes where initially all the nodes are blank. Information enters the graph through a boundary node. When the blank boundary node receives a token, the node stores the token in its local memory and becomes “open” to receive a second token for a limited amount of time and pass it downstream to other nodes. When a second token arrives, the open node makes a connection with a free neighbor which becomes its child, and passes to this child the token. Step by step, the nodes form a tree where each node has one parent upstream and possibly multiple children downstream. The basic rules of behavior can be summarized as follows.

- If a node with empty memory receives a token from upstream, it absorbs the token and it becomes open for a time interval long enough for the next token of an incoming sequence of tokens to arrive. An open-state counter is used to count down a certain number of cycles, specified below. After this time has elapsed, the node is no longer open, and is closed.

- If a node is open and receives a token, it tries to send the token downstream to its children, and it also resets the open-state counter so that it will remain open until the next incoming token. In addition it sends an acknowledgment message to its parent who gave the token. If there are no children or if none of the children accept the token, then the current node makes a new connection with one of its free neighbors in order to pass to it the incoming token.
- If a node is closed and receives a new token from upstream, then it compares the new token to its own token. If the tokens are the same, then the node absorbs the token, becomes open, and sends an acknowledgment upstream to its parent. If the tokens differ, then the node rejects the new token, it sends no acknowledgement upstream, and it becomes closed-non-accepting for some cycles. The time duration is proportional to the length of sequences that can be safely distinguished. However there are also alternative rules of behavior that can handle sequences of unlimited length as explained in section 4.

Finally there are some additional rules of behavior that are used when the nodes perform a retrieval operation, and they are discussed later below.

2.2 The rate of processing

As stated earlier, there is no global synchronization between the nodes. But there is the constraint that the time interval spent by each node to finish its own computational cycle is limited, say not larger than T . And further, all the nodes know this intrinsic value T , and have local clocks so they can wait at most T for messages to be communicated with their connections. In other words, the nodes operate in parallel and asynchronously doing computational steps of duration T .

The asynchronous operation of the nodes is similar to the case of workstations doing a parallel distributed computation of fluid dynamics (Skordos, 1995; Skordos & Sussman, 1995) with one exception. The workstations must wait in locked-mode at the end of each cycle, until they receive messages from all of their neighbors before they can continue to compute the next cycle. In contrast, the nodes of the amorphous model simply proceed to the next cycle after a duration T assuming that no message is coming if no message has arrived within the time interval T .

Consider the following scenario. Node A is open and sees it has received a new token from its parent. In response A sends an acknowledgment to its parent, and sends downstream the new token to its children. These actions are completed in one cycle, a time interval T . Next, A waits for an interval T before proceeding further. During this second interval T , any child has enough time to receive the message and to respond with an acknowledgement if it accepts the new token. At the start of third cycle, node A checks for messages from its children, and if it finds an acknowledgement, it has finished its obligations with regard to this token. Otherwise, node A must make a new connection with one of its free neighbors, say B , and pass the token to B .

For simplicity, it is assumed that node A makes a connection with a neighbor B and passes to B the new token in one cycle T , namely by the end of the third cycle. This means that node A becomes free and is ready to process a new token at the start of the fourth cycle, and the rate of processing is one token every three cycles in this scenario.

One can always increase the interval T to make possible the formation of a new connection in one cycle. Or alternatively, one can use more cycles for the processing of each token. There is freedom in the model to do so. And it may be necessary in order to accommodate a particular hardware implementation, or to model additional interactions between the nodes as discussed in sections 4 and 6.

Finally note that the rate of processing is an important global parameter of the whole system. First, it determines the duration of the open-state counter. Second, a sequence of tokens must arrive at the boundary node of the graph at the same rate as the processing rate, namely one token every three cycles in the above scenario. And third, two different sequences that are sent to the graph must have a time gap between them which is larger than the duration of the open-state counter, namely three cycles. In this way the two sequences will not be mixed together.

3. The simulator

In order to test the amorphous algorithm, a serial simulator is used which has its own timing considerations. To avoid any timing side-effects between the nodes, two copies of the graph of nodes are used, the old and the new. A node reads the state from the old graph, namely its own local state and messages, and applies its actions to the new graph. Thus, there is a time interval between the old and the new graph, which is the local time step or one cycle.

In addition, the nodes are processed in random serial order, to enforce the constraint that one must not depend on the serial simulation order between nodes for messages and computation to complete. And likewise, when a node accesses the list of its neighbors, a random permutation list is used to randomize the accessing order.

The default processing rate in the simulator is one token every three cycles as above. In particular, if an open node A receives a new token, during cycle-1 A reads the token from the old graph, and sends messages to its parent and children, applying them to the new graph. During cycle-2 the parent and the children read the messages and respond accordingly, while node A waits doing nothing. During cycle-3 node A reads the responses of the children, and if necessary, A makes a new connection with some neighbor B .

3.1 The implementation of tokens

The token is the basic unit of information. The beginning or end of an ordered sequence of tokens is denoted with special tokens which act as markers: the left-parenthesis “(“ and the right-parenthesis “)”. For example the sequence of numbers “(1 4 7 2)” enters the boundary node of a graph first with “(“, then “1”, then “4”, and so on, and finishes with the token “)”.

The value of a token is a one byte character. A two-digit number such as “81” is a composite of two tokens, and thus, the tokens must have an extra type field to indicate the continuation of a composite object. The type is atom-continued for the first token “8” and type atom for the second token “1”. The type field is implemented as a bit-flag, so that different combinations of bit-flags can be turned on and off easily. The token is very versatile, and it is used both as the message sent between the nodes and also as the memory stored by the nodes locally.

The use of a pair quantity, a type field and a value field, is a programming convenience. It is easy to see that the pair could be replaced with a single scalar by using an extended encoding. For example, an “atom-continued 2” could be encoded as the integer 12 meaning a “second octave 2” in contrast to an “atom 2” which would be simply 2 meaning a “first octave 2”. Other types could be encoded as third octave, fourth octave, and so on.

Finally another design aspect is the use of a memory register and a transit register for each node. The latter one is used temporarily when a node is pushing downstream an incoming token, and keeps the token in the transit register while waiting to hear from its children. The transit register is also used during a retrieval operation as explained in section 5.

4. Storing a sequence

Let us store a sequence that illustrates the challenge of directing the tokens to the correct path. The sequence “(2 6 3)” is to be stored in a graph that has stored already the sequence “(2 3 4)”. The top node “(“ opens first, and then the node “2” opens under the node “(“. If there is a node containing a “6” underneath the “2”, it opens next; otherwise, the node “2” makes a new connection with one of its free neighbors and passes to this child the token “6”. The interesting event occurs when the next token “3” of the sequence “(2 6 3)” arrives. The graph already has a sequence “(2 3 4)” so it is possible that the incoming token “3” will match the “3” under the “2” and this would be incorrect. Instead, the incoming token “3” must be stored under the node “6” to represent correctly the incoming sequence. There are at least three approaches to deal with this problem, as follows.

- The closed-non-accepting behavior: the node “3” of the already stored sequence “(2 3 4)” becomes closed-non-accepting for a few cycles after the token “6” of the sequence “(2 6 3)” is tried. And thus, the next incoming token “3” is stored under the node “6” to represent correctly the incoming sequence.
- An additional interaction between parent and children: node “2” first tries to find an open path by sending out a tentative token “3” to be accepted only if a child is already open. If there are no replies, then it tries to open a path by sending again the token “3” to all its children to be accepted if a child matches the token. Finally if there are no replies again, then node “3” makes a new connection.
- The remember-my-path approach: node “2” remembers which child absorbed the previous token that passed (during the duration that node “2” is open only), and sends the next token selectively only to this child. In this case “3” is sent only to node “6”.

The first approach is the default in the simulator. It is appealing because the nodes interact with each other in a simple way. However, it has the drawback that it can not handle sequences of unlimited length. In particular, the number of cycles of the closed-non-accepting behavior is proportional to the length of sequences that can be safely distinguished. In the case of a multiplication table where the longest sequence is 7 tokens “(* 9 9 81)”, the closed-non-accepting behavior must be 21 cycles assuming the rate of processing is 3 cycles

per token. Then, the minimum gap between two different incoming sequences must also be 21 cycles.

To deal with sequences of unlimited length, the second and third approaches can be used. The only drawback of the second approach is that it uses an additional interaction between parent and child, and adds two more cycles to the rate of processing. In particular, the rate would change from 3 cycles to 5 cycles per token. Finally, the only drawback of the third approach of “remembering-my-path” is that the nodes can not send the same message to all the children without distinction. It requires nodes to remember, while they are open, which child absorbed the previous token that passed through them, and to communicate selectively with this child only.

5. Retrieving a sequence

To retrieve a sequence, a special token is used which acts as a variable and matches every token. We refer to it as baladeur or jockey token, and we denote it with the special character “?”. When a node compares two tokens, it also checks if the token type is baladeur which means a match always. The following additional rules of behavior are used to handle a baladeur.

- If a baladeur opens a node that has an atom-continued token, then the baladeur is sent automatically further downstream to open the continuation of the composite object.
- After a baladeur opens a path, the subsequent tokens that pass downstream from that node are marked opened-by-baladeur so the nodes will not make new connections to store a sequence that contains a baladeur.
- If the incoming sequence has tokens before the baladeur and these get stored in the graph, then we have an unfinished sequence; namely, a sequence that lacks the end-marker “)”. Unfinished sequences are automatically deleted as follows. If a node has stored a value, and is not open, and has no children, and is not itself an end-marker “)”, then it is a loose node and must free itself on the next cycle. It becomes empty, and it also deletes the upstream connection with its parent, so that successively the unfinished sequence is deleted completely.

The above rules for baladeur are somewhat complex, and a simpler approach might be to operate the graph of nodes in two distinct modes: the learning mode, and the retrieval mode. In the latter one, incoming sequences are only matched, and not stored. Such an arrangement might also provide capability to learn sequences that contain themselves a variable; for example, the sequence “(* ? 1 ?)” could be learned. But this is a topic for future work. In the present algorithm the ordered sequences do not contain variables themselves. The above rules for baladeur make it possible for the nodes to learn or to retrieve based on the incoming sequence only. If the sequence contains no baladeur and is a new sequence, it is automatically learned; otherwise it is matched for retrieval.

A typical scenario for retrieval is as follows. The multiplication sequence “(* 2 3 6)” has been stored, and the questioning sequence “(* 2 ? 6)” comes in. A conduit of nodes opens successively starting with the node “(“, then “*”, then “2”, then all the nodes under

“2” such as 2-3, 2-4, 2-5, etc. Subsequently only one of these paths matches the next token “6”, and finally the end-marker token “)” matches the end of the stored sequence.

When the end-marker “)” of a stored sequence is matched, the reverse flow of the retrieval operation begins that sends upstream all the tokens of the matched sequence to the boundary node. The end-marker node first sends an empty retrieval-marked token upstream to its parent, and then enters a retrieval mode where it will send its own token upstream in the next cycle. A node that receives an empty retrieval-marked token from downstream does the same: it sends an empty retrieval-marked token upstream, and prepares to send its own token upstream in the next cycle. A node remains in retrieval mode and keep sending upstream any tokens coming from downstream as long as there are retrieval-marked tokens coming. If there are none, then the node exits the retrieval mode and becomes idle.

We notice that this retrieval operation creates a sequence of tokens that arrive one token after another. This is done for simplicity, and is different from the incoming sequence which enters the graph at the rate of one token every three cycles. But it is possible to change the retrieval into a frequency of 1 token every 3 cycles if desired. In particular, the nodes must use another counter, say the retrieval-open counter, that counts down 3 cycles. At every third cycle, a node which is in retrieval mode sends upstream the token which is waiting in its transit register, and checks to see if there is another retrieval token coming from its children downstream. If there is one, the new token is placed in the transit register and waits another 3 cycles until it is sent further upstream.

The tables of figures 5 and 6 show the retrieval timing for the sequence “(a b c)” in the two different frequencies: 1 token every cycle and 1 token every 3 cycles. The retrieval starts from the rightmost part of the figure which corresponds to the end-marker node “)” and the tokens move from right to left. The outgoing boundary is at the leftmost part of the figure. The columns labeled with a dash correspond to the links between the nodes. Time increases from top to bottom, one line for every cycle. The symbol “@” is used to denote an empty retrieval-marked message. The superscript notation $a^3, a^2, \dots$ shows the retrieval-open counter that counts down 3 cycles before a token can be sent further upstream.

6. Physical space and connectivity

Figure 1 shows a graph of 20 nodes that store the sequences “(* 2 2 4)”, “(* 2 3 6)” and “(* 2 4 8)” of the multiplication table. In this example, each node has 10 neighbors. For simplicity, the underlying physical space of the nodes is assumed to be one-dimensional periodic, and the neighbors are chosen 5 to the left and 5 to the right of each node. Thus, the space is conveniently drawn as a circle. But for the purposes of the amorphous algorithm, only the connectivity of the nodes matters, and the precise location of the nodes is unspecified.

During the time evolution of a graph of nodes, an important question is how to choose a free neighbor to form a connection. As the nodes absorb incoming information and form a tree, it is desirable to spread-out thinly among the nodes in order to avoid a bottleneck. A bottleneck occurs if a node tries to make a connection, but it has no free neighbors even though the graph of nodes has many free nodes far away from this node. Thus, the algorithm must try to spread the connections among the nodes. There are at least four approaches to consider.

Figure 5: Retrieval timing with 1 token every cycle.

—	(	—	<i>a</i>	—	<i>b</i>	—	<i>c</i>	—	)
.	.	.	.	.	.	.	.	@	)
.	.	.	.	.	.	@	<i>c</i>	)	.
.	.	.	.	@	<i>b</i>	<i>c</i>	)	.	.
.	.	@	<i>a</i>	<i>b</i>	<i>c</i>	)	.	.	.
@	(	<i>a</i>	<i>b</i>	<i>c</i>	)	.	.	.	.
(	<i>a</i>	<i>b</i>	<i>c</i>	)	.	.	.	.	.
<i>a</i>	<i>b</i>	<i>c</i>	)	.	.	.	.	.	.
<i>b</i>	<i>c</i>	)	.	.	.	.	.	.	.
<i>c</i>	)	.	.	.	.	.	.	.	.
)	.	.	.	.	.	.	.	.	.

Figure 6: Retrieval timing with 1 token every 3 cycles for the sequence “(a b c)”. Time increases from top to bottom and tokens move right to left. The columns labeled with dashes correspond to the links between nodes. The symbol “@” denotes an empty retrieval-marked message. The superscript numbers show the retrieval-open counter. The retrieval is shown below only until “b” for brevity.

—	(	—	<i>a</i>	—	<i>b</i>	—	<i>c</i>	—	)
.	.	.	.	.	.	.	.	@	) ³
.	.	.	.	.	.	@	<i>c</i> ³	.	) ²
.	.	.	.	@	<i>b</i> ³	.	<i>c</i> ²	.	) ¹
.	.	@	<i>a</i> ³	.	<i>b</i> ²	.	<i>c</i> ¹	)	.
@	(³	.	<i>a</i> ²	.	<i>b</i> ¹	<i>c</i>	) ³	.	.
.	(²	.	<i>a</i> ¹	<i>b</i>	<i>c</i> ³	.	) ²	.	.
.	(¹	<i>a</i>	<i>b</i> ³	.	<i>c</i> ²	.	) ¹	.	.
(	<i>a</i> ³	.	<i>b</i> ²	.	<i>c</i> ¹	)	.	.	.
.	<i>a</i> ²	.	<i>b</i> ¹	<i>c</i>	) ³	.	.	.	.
.	<i>a</i> ¹	<i>b</i>	<i>c</i> ³	.	) ²	.	.	.	.
<i>a</i>	<i>b</i> ³	.	<i>c</i> ²	.	) ¹	.	.	.	.
.	<i>b</i> ²	.	<i>c</i> ¹	)	.	.	.	.	.
.	<i>b</i> ¹	<i>c</i>	) ³	.	.	.	.	.	.
<i>b</i>	<i>c</i> ³	.	) ²	.	.	.	.	.	.

- Choose a free neighbor that has the highest connectivity: first-order connectivity criterion.
- Choose a free neighbor using a second-order connectivity criterion.
- Choose randomly a free neighbor to connect to.
- Choose a free neighbor that is furthest away from the node.

The first criterion chooses the neighbor that has the largest number of free neighbors. This is the default approach used in the simulator and in all the examples of this paper. If the connectivity of node i is $\mathbf{Con}(i)$ and the index i runs over the free neighbors of a node, then the first-order connectivity criterion can be written as

$$\max_i \mathbf{Con}(i) \quad (1)$$

The second-order connectivity criterion improves a little over the first-order criterion but is more complex, and the gain is not worth the cost in complexity. If the free neighbors of node i are denoted $\mathbf{FreeN}_j(i)$ where j runs over the free neighbors of node i , the second-order density criterion can be written as

$$\max_i \sum_j \mathbf{Con}(\mathbf{FreeN}_j(i)) \quad (2)$$

which means to choose the neighbor i whose possible connections all together have the largest sum of connectivities.

The third approach of randomly choosing a free neighbor is not great, but is not too bad either, as can be seen in figures 7-10.

In considering the fourth approach, it is interesting that the exact opposite approach of choosing a free neighbor closest to the current node, guarantees to reach a bottleneck as soon as possible. Indeed, the fourth approach of choosing a free neighbor furthest away from the current node tends to spread-out the connections, and provides good results but introduces a physical dependency. It assumes that the nodes can compare relative distances between them in order to choose the neighbor furthest away from the current node. Thus this approach is not as general as the others, and should be avoided.

The first-order connectivity criterion is implemented in the simulator by having all the nodes keep track of how many free neighbors they have at any given time. And when a node is looking for a neighbor to connect to, it chooses the neighbor with highest connectivity. For simplicity, the choosing is done automatically by the underlying system, and is not simulated directly. However, it is possible to simulate directly: when a node seeks a connection, it first polls its neighbors to read their connectivity. Then it chooses a node with the highest connectivity, and makes the chosen node its child.

Note that this important step, where a node is making a new connection with a chosen child, is an example of selective communication between nodes. The initiating node only communicates with the neighbor that is going to become its child. The other occasion we have seen where selective communication takes place between nodes, is when a node sends a message upstream to its parent.

Figure 7: Simulations that try to store the full multiplication table of 36 sequences. The horizontal axis shows different connectivities, and the vertical axis shows different number of nodes. A table entry of “0” means success that all 36 sequences are stored, “1” means one sequence is left out, and so on. In the first table, we choose a free neighbor that has the highest connectivity.

.	40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70
150	5	9	6	8	6	7	5	7	5	5	4	4	3	4	2	4
175	7	5	5	4	6	1	0	2	2	0	0	2	0	0	0	0
200	6	7	4	3	1	1	0	1	0	0	0	0	0	0	0	0
300	7	6	3	5	2	1	0	0	0	0	0	0	0	0	0	0

Figure 8: Choose a free neighbor using the second-order connectivity criterion.

.	40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70
150	5	5	4	3	5	4	3	6	3	5	2	4	3	4	2	2
175	3	0	3	0	1	0	0	0	1	0	0	0	1	0	0	0
200	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
300	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 9: Choose randomly a free neighbor to connect to.

.	40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70
150	15	12	12	11	10	10	7	6	5	3	3	3	6	3	1	1
175	15	16	14	10	13	13	6	7	3	1	6	3	2	1	1	1
200	13	18	8	10	15	6	7	11	6	3	3	1	3	3	1	1
300	15	15	13	13	6	11	7	6	3	5	1	0	8	1	7	3

Figure 10: Choose a free neighbor that is furthest away.

.	40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	70
150	5	5	6	6	4	4	2	1	1	2	1	3	3	2	2	1
175	3	3	5	4	1	2	0	2	0	0	0	0	0	0	0	0
200	0	3	4	3	0	1	0	0	0	0	0	0	0	0	0	0
300	0	3	4	3	0	0	0	0	0	0	0	0	0	0	0	0

6.1 Simulation experiments

In order to test the different approaches of choosing a neighbor, experiments were done on the task of storing the full multiplication table of 36 sequences from “(* 2 2 4)”, “(* 2 3 6)” up to “(* 9 9 81)”. Figures 7-10 tabulate the results of these experiments by showing the smallest initial connectivity and the fewer nodes that are needed to store all the multiplication sequences without a bottleneck. The initial connectivity is uniform among the nodes in all cases. The horizontal axes show connectivity and the vertical axes show number of nodes. A table entry of “0” means success that all 36 sequences are stored, “1” means one sequence is left out, and so on. We can see that the first-order connectivity criterion performs very well.

Figure 1 shows one of the smallest possible graphs that can store the full multiplication table of 36 sequences from “(* 2 2 4)”, “(* 2 3 6)” up to “(* 9 9 81)”. This graph has 175 nodes and initial uniform connectivity of 52 neighbors per node. After all the sequences have been stored, there are only 25 nodes left out of 175, which corresponds to node utilization of 85 per cent.

7. Conclusion and future directions

We have seen an amorphous program that organizes a set of independent processing nodes to store and to retrieve ordered sequences of names and numbers. This includes simple tokens such as one-byte characters and one-digit numbers, and composite objects such as multi-digit numbers and general alphanumeric strings.

The basic idea is the time-controlled accepting and rejecting rules of behavior, and the fact that a connection between two nodes distinguishes the nodes into parent and child which gives rise to the structure of a tree. As information enters the graph, the nodes make logical connections between them to direct the information down the correct paths. To retrieve a stored sequence, a special variable token is used that matches every token. Also the end-marker of a stored sequence is used in a special way to initiate the sending of the retrieved sequence upstream to the boundary node. It is possible to retrieve a sequence of the multiplication table by trying any of the following combinations “(* 2 3 ?)”, “(* 2 ? 6)”, “(* ? 3 6)”, and even “(? 2 3 6)”.

One limitation of the present program that is left for future work, is how to handle non-unique or multiple retrievals. For example, in the case of the multiplication table the pattern “(* ? ? 12)” matches more than one sequence. Also, if the stored data has multiple branches such as “(dog runs)” and “(dog barks)”, then the pattern “(dog ?)” produces multiple results. The present system garbles the multiple results on retrieval because it has no mechanism for choosing one of the multiple sequences and blocking the others, or for being able to produce all matched sequences at the output one after the other.

Some applications of the present approach include the following. One can generalize the memorization of ordered sequences to memorization of arbitrary nested LISP lists using the discrimination network described in the appendix of (de Kleer, Doyle, Rich, Jr., & Sussman, 1978). This is because a discrimination network produces an unwrapped representation of an arbitrary nested LISP list as an ordered sequence. For example, the LISP list that prints as “(3 (2 . 1))” becomes “down 3 down 2 up 1 up nil” and can be stored easily as an ordered sequence in the present system.

Another application of the present approach is to lookup name-value pairs. For example, one can use sequences of the form “(lookup name object)” where object is another sequence, perhaps the discrimination network “down 3 down 2 up 1 up nil”. Then, a lookup request “(lookup name ?)” matches the whole object associated with the given name. The matching could be done in an analogous way to the composite token of section 3.1. We group the sequence “down 3 down 2 up 1 up nil” using markers such as left-right parentheses, and we use a new flag, an object-continuation flag, to indicate that the baladeur “?” must be sent automatically further down to match all the parts of a continued object until the object finishes.

Finally another application may be the semantic thread memory of Vaina and Greenblatt (Vaina & Greenblatt, 1979) which is popular in many systems such as (Stamatoiu, 2004) to learn commonsense knowledge about the world. A semantic memory contains symbols linked together in a way that captures their meaning, and the basic idea of thread memory is to organize the symbols into ordered sequences of names that have an ordering from general to more specific. It is easy to see that such ordered sequences of names can be stored and retrieved using the present amorphous program. However, this is not the end of the story.

Thread memory includes techniques for manipulating and retrieving the thread memory. Storage by itself is passive, while the benefits of thread memory arise from the operations described in (Vaina & Greenblatt, 1979) that thread memory facilitates. To apply the present approach to thread memory, we must first enhance the amorphous system to provide more utilities at the node level that can serve the operations of thread memory. And in addition we must design a planning unit that interacts with the amorphous memory by reading and writing into the memory. The planning unit could also be an amorphous program. It is a challenging topic for future work to build interacting graphs of different types of nodes: one set of nodes for the memory, and another set of nodes for the planner.

Acknowledgements

The author wishes to thank Professor Gerald Jay Sussman for many important suggestions, ideas, and comments on this paper.

References

- Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Jr., T. F. K., Nagpal, R., Rauch, E., Sussman, G. J., & Weiss, R. (2001). Amorphous computing. *Communications of the ACM (also MIT Artificial Intelligence Memo 1665, August 1999)*, 43(5).
- Butera, W. J. (2002). *Programming a Paintable Computer*. Media Lab, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Coore, D. (1999). *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

- de Kleer, J., Doyle, J., Rich, C., Jr., G. L. S., & Sussman, G. J. (1978). *AMORD A Deductive Procedure System*. MIT Artificial Intelligence Memo 435, Massachusetts Institute of Technology.
- Nagpal, R. (1999). *Organizing a Global Coordinate System for Local Information on an Amorphous Computer*. MIT Artificial Intelligence Memo 1666, Massachusetts Institute of Technology.
- Nagpal, R. (2001). *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Nagpal, R., & Coore, D. (1998). *An Algorithm for Group Formation and Maximal Independent Set in an Amorphous Computer*. MIT Artificial Intelligence Memo 1626, Massachusetts Institute of Technology.
- Newell, A., & Simon, H. A. (1972). *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall.
- Rauch, E. (2004). *Diversity in Evolving Systems: Scaling and Dynamics of Genealogical Trees*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Skordos, P. A. (1995). Parallel simulation of subsonic fluid dynamics on a cluster of workstations. In *Proceedings of High Performance Distributed Computing 95, 4th IEEE Int'l Symposium, Pentagon City, Virginia (also MIT Artificial Intelligence Memo 1485, November 1994)*.
- Skordos, P. A., & Sussman, G. J. (1995). Comparison between subsonic flow simulation and physical measurements of flue pipes. In *Proceedings of ISMA 95, International Symposium on Musical Acoustics, Le Normont, France (also MIT Artificial Intelligence Memo 1535, April 1995)*.
- Stamatoiu, O. L. (2004). *Learning Commonsense Categorical Knowledge in a Thread Memory System*. M.S. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Sussman, G. J., Abelson, H., & Sussman, J. (1985, second edition 1996). *Structure and Interpretation of Computer Programs*. MIT Press and McGraw-Hill.
- Vaina, L. M., & Greenblatt, R. D. (1979). *The Use of Thread Memory in Amnesic Aphasia and Concept Learning*. MIT Artificial Intelligence Working Paper 195, Massachusetts Institute of Technology.
- Weiss, R. (2001). *Cellular Computation and Communications using Engineered Genetic Regulatory Networks*. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
- Weiss, R., Knight, T. F., & Sussman, G. J. (2004). *Genetic Process Engineering*. Cellular Computing, Martyn Amos editor, pp.43–73, Oxford University Press.
- Yip, K., & Sussman, G. J. (1996). *A Computational Model for the Acquisition and Use of Phonological Knowledge*. MIT Artificial Intelligence Memo 1575, Massachusetts Institute of Technology.