Report Number: WUCS-86-07

1986-03-01

A Systolic Parsing Algorithm for a Visual Programming Language

Adam W. Bojanczyk and Takayuki Dan Kimura

In this paper we consider a problem of parsing a two-dimensional visual programming language Show and Tell on a two-dimensional array of processors. A program in Show and Tell is a bit-mapped, two-dimensional pattern satisfying a certain set of grammatical rules. The pattern consists of partially ordered set of rectilinear boxes and arrows distributed over the space of nxn pixel area. The corresponding directed graph, the box graph, where boxes are nodes and arrows are directed edges, may not have a cycle in a Show and Tell program. The cycle detection is the most computationally intensive stage of the... **Read complete abstract on page 2.**

A Systolic Parsing Algorithm for a Visual Programming Language

Adam W. Bojanczyk and Takayuki Dan Kimura

Complete Abstract:

In this paper we consider a problem of parsing a two-dimensional visual programming language Show and Tell on a two-dimensional array of processors. A program in Show and Tell is a bit-mapped, two-dimensional pattern satisfying a certain set of grammatical rules. The pattern consists of partially ordered set of rectilinear boxes and arrows distributed over the space of nxn pixel area. The corresponding directed graph, the box graph, where boxes are nodes and arrows are directed edges, may not have a cycle in a Show and Tell program. The cycle detection is the most computationally intensive stage of the parsing section of a Show and Tell program. We propose to exploit the concept of systolic array in parsing of Show and Tell programming language. A given bit pattern is mapped onto nxn array of mesh connected processors with one pixel assigned to one processing element. We show an algorithm for cycle detection which runs in time proportional to the size of the box graph. The complexity of any individual processor is independent on n, the parameter describing the size of the array.

A SYSTOLIC PARSING ALGORITHM FOR A VISUAL PROGRAMMING LANGUAGE

Adam W. Bojanczyk and Takayuki D. Kimura

WUCS-86-07

March 1986

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

As appeared in 1986 Proceedings of Fall Joint Computer Conference, Dallas, November 1986.

ABSTRACT

In this paper we consider a problem of parsing a two-dimensional visual programming language Show and Tell on a two-dimensional array of processors.

A program in Show and Tell is a bit-mapped, two-dimensional pattern satisfying a certain set of grammatical rules. The pattern consists of a partially ordered set of rectilinear boxes and arrows distributed over the space of nxn pixel area. The corresponding directed graph, the box graph, where boxes are nodes and arrows are directed edges, may not have a cycle in a Show and Tell program. The cycle detection is the most computationally intensive stage of the parsing section of a Show and Tell program.

We propose to exploit the concept of systolic array in parsing of Show and Tell programming language. A given bit pattern is mapped onto nxn array of mesh connected processors with one pixel assigned to one processing element. We show an algorithm for cycle detection which runs in time proportional to the size of a box graph. The complexity of any individual processor is independent on n, the parameter describing the size of the array.

1. Introduction

Thanks to the recent advancement of VLSI technology, high resolution graphics capabilities have become a dominant computer interfacing mechanism for end users. Efficient man-machine communication can be achieved through a graphic interface because of its high bandwidth. Visual programming[1] is a new concept in software engineering that takes advantage of such high bandwidth for designing better software environments. In a visual programming language, a two-dimensional pattern (or an icon) is used to represent various programming concepts, such as iteration, concurrency, recursion, and so forth. One key problem in visual programming language design is to find a formal syntax of such a two-dimensional language and a parsing algorithm based on the formalism.

A systolic array[2] is a collection of identical processing elements (PE, a CPU with local memory), mesh-connected in a two dimensional form. Each PE communicates with its neighbors by passing messages through a thin-wire. There is no shared memory among the PE's. The concept of systolic array was proposed to take advantage of the reduced hardware cost due to the VLSI technology advancement. Many systolic algorithms are designed for numeric, database, and textual applications.

We propose in this paper another application area; a parsing of two dimensional graphic programming language. A program in a such a language is a bit-mapped two-dimensional pattern satisfying a certain set of grammatical rules. The systolic parsing problem is to construct a systolic array such that when a visual program is mapped onto the array with one PE assigned to one pixel, the array can decide whether the pattern satisfies the grammatical rules or not. It is desirable for the array to make a decision in linear time to the size of the pattern. It is also desirable that the complexity of each PE is independent of the size of the array.

Show and Tell™ Language[3] (STL) is a visual programming language designed for novice computer users such as school children and implemented on the Apple® Macintosh™ personal

computer. A STL program is entered through mouse clicking and it consists of a partially ordered set of boxes and arrows. Arrows and boxes may not form a cycle in a STL program. The cycle detection is algorithmically the most complex component of the parsing section of the STL system.

As a part of our efforts to find formal specification methods for visual programming languages (syntax and semantics), we are investigating possible systolic parsing algorithms for STL. In this paper we will present a design of a systolic array that can detect a cycle in a 'box graph' in linear time to the size of the graph, where box graph is an abstraction of a STL program capturing features relevant to the detection of cycles.

In the next section we will briefly introduce STL to provide the background for the problem. Section 3 will define the problem in terms of a formal definition of box graph and systolic array. Section 4 and 5 will describe the results, i.e., the design of a finite state machine that can detect a cycle in linear time to the size of the box graph. Section 6 will conclude with possible extensions of the results.

2. Show and Tell Language

Keyboardless programming is one of the main goals of the STL design. A STL program can be created by using a mouse only. A keyboard is needed only for entering texual or numeric data. STL is designed for computer users who are not familiar with keyboarding.

An STL program consists of nested boxes connected by arrows. Loops and cycles are not allowed. A box may be empty or may contain a data value, an icon which is a name of a system or user-defined operation, or another STL program. Arrows allow data to flow from one box to another. An operation in a box will be executed when and only when all incoming values have arrived at the box. Except for this data dependency there is no inherent sequencing mechanism in STL. The semantic model of STL is based on the concept of dataflow. However, since there is

no loop in an STL program, once an operation is executed and the result is registered in an empty box, the value will never change. There are no side effects. Thus, STL is a functional parallel programming language.

An empty box can be filled with a data object. Some empty boxes are used for communication with the environment of the program. They are called the *base boxes* of the program and are depicted by a thicker box frame. They correspond to formal parameters of a subroutine in traditional programming languages. The STL interpreter executes a STL program by filling the base boxes with appropriate solution values.

An STL program can be named by a user defined icon. Any Macpaint™ picture can be used as a name. When a box containing a user defined icon is evaluated by the STL interpreter, the program whose name is the icon will be evaluated, after the incoming data values are moved into the base boxes of the called program. An icon in STL is a subroutine name. Recursive definition of a program is also allowed.

An example of STL program is given in Figure 1. The program defines the factorial function recursively. Note that the program has one input and one output parameter. It consists of four boxes one of which contains five boxes. The inconsistent box, which is defined as any box that contains conflicting information such as "5 flowing into 0", is shaded by the interpreter. The data flows from the top to the bottom of the graph. The name of the program is given on the upper left corner of the editing window. The leftmost column provides the editing tools for program construction. Any box that overlaps with other boxes or forms a cycle will be rejected by the editing program of the STL system.

The language is implemented on the Apple® Macintosh™ personal computer. Using the editing tools provided by the STL system, a user constructs a program on the editing window through a sequence of mouse clicking and dragging. In order to draw a box, for example, the user drags the mouse from the upper-left corner of the box, causing a mouse-down event, to the lower-right

corner, causing a mouse-up event. The editing program in the STL system recognizes these two mouse events and constructs an internal data structure representing the box, provided that the box is acceptable, i.e., the box does not overlap with other boxes nor forms a cycle with the existing arrows.

The current STL system does not have capability of parsing a bit mapped image as a STL program. For example, if the factorial program of Figure1 is constructed by MacPaint™, and the resulting MacPaint file is pasted onto the editing window of the STL system, the current STL editor program will interprete the image as a background (comment) image of some other program, and will not be able to parse the bit image as a STL program. Thus, *programming in MacPaint* is not possible with the current STL system.

In this paper we try to solve the problem of finding a parsing algorithm for two-dimensional representation of STL programs. One approach is to find a formal syntax for STL and to construct a parsing algorithm based on the formalism. Our efforts in this direction is reported in a separate paper[4]. Another approach is to construct an algorithm which can accept all and only images that represents a legal STL program. We will take the second approach in this paper.

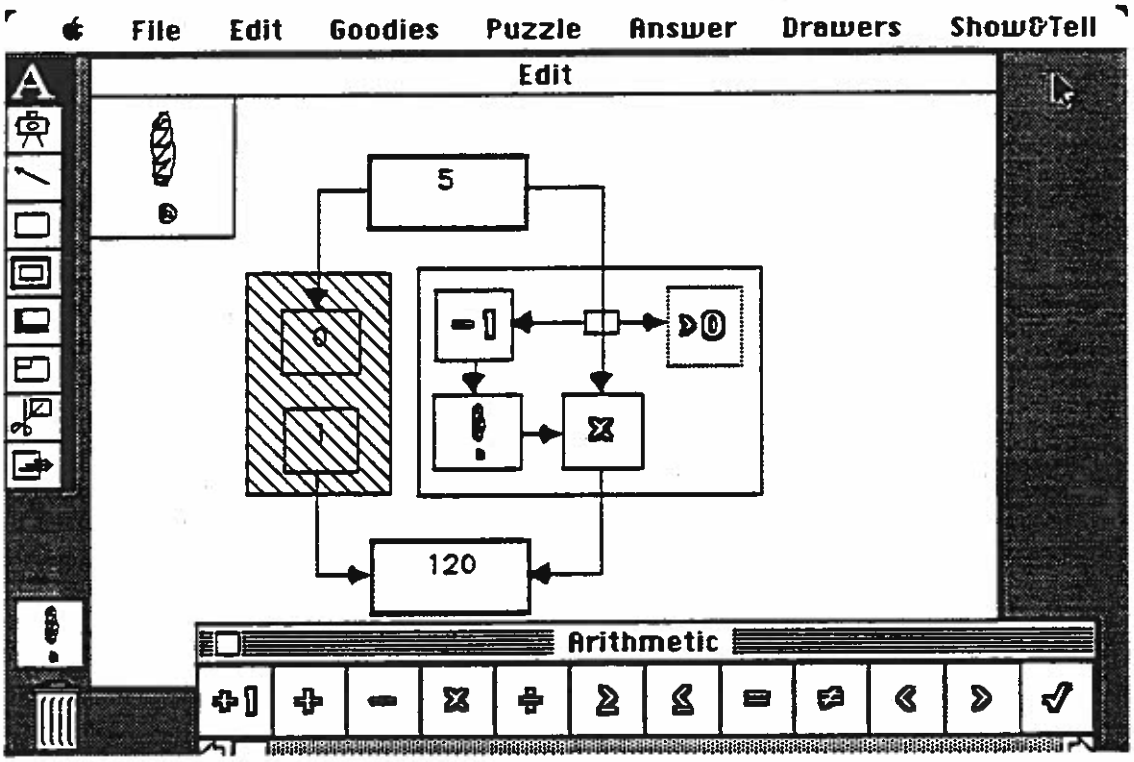


Figure 1: A Recursive STL Puzzle(Factorial Function)

3. Problem Definition

In this section we will define the problem. Our solution to the problem will be given in the next section. First we will define the concept of *box graph* as a syntactic abstraction of STL program. Then, we will define the concept of *systolic array* as a model of parallel computer architecture. Finally we will define the problem of detecting a cycle in a box graph by an systolic array in linear time to the size of the box graph.

3.1 Box Graph

A box graph is a collection of boxes and arrows geometrically distributed over the space of n x n pixel area. No two boxes overlap with each other. No box contains another box. Every arrow starts from the boundary of one box and ends on the boundary of another box. Arrows may intersect with one another, but they don't branch out. No two arrows intersect with a box at the same location. There must be no loops or cycles in a box graph, i.e., there is no sequence of arrows that connects a box to itself. Boxes and arrows are composed of horizontal and vertical line segments. No diagonal lines exist in a box graph. An example of box graph is given in Figure 2. Note that a box graph is an abstraction of a STL program in hiding the type and the content of each box. Thus, in a box graph no nesting of boxes exist. There is no semantics associated with a box graph.

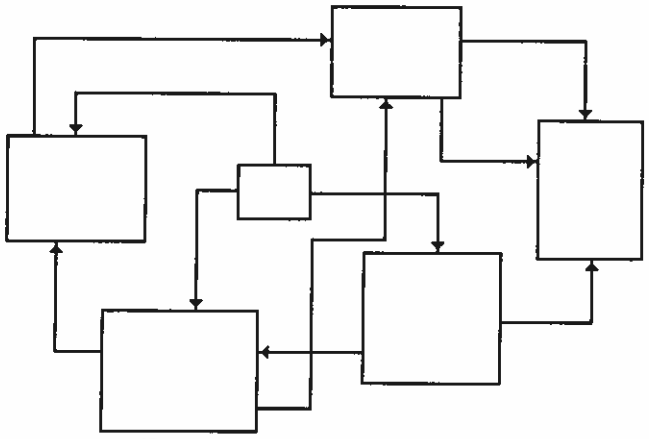


Figure 2: An Example of a Box Graph

The exact definition of a well formed box graph as a bit image will be given in the section 3.3.

3.2 Systolic Array

A systolic array is a collection of n x n processing elements (PE: a CPU with local memory), mesh-connected in a two dimensional form. See Figure 3. Each processor is a simple finite state machine with local memory registers. The memory size of a single processor is independent on n, the parameter describing the size of the array. The processors communicate only with their four nearest neighbors, i.e., north, east, south and west neighbors. Knowing what operations the processors must perform in order to solve a problem, we define a *time unit* to be the maximal time that is necessary for a processor to perform the most time consuming operation together with loading and unloading its registers. Like the memory size, the duration of the time unit is independent on n. A synchronization mechanism allows processors to exchange data at time instants separated by integer multiples of a time unit. We assume that processors are microprogammable. This allows us to change the set of operations performed by each processor when necessary. This approach is similar to one adopted in the PSC project[5].

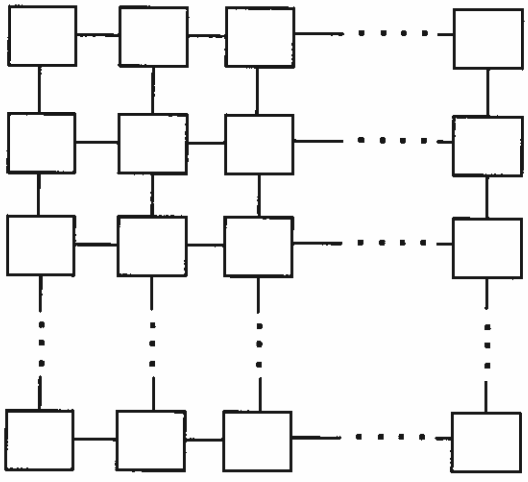


Figure 3: A Systolic Array

3.3 Problem

To design a systolic array of size n x n that can decide whether a given bit pattern is a well formed box graph or not, in $O(n^2)$ steps with constant memory requirement for each PE, where the bit pattern is mapped onto the array with one pixel assigned to one PE.

A well formed box graph is defined as follows:

(1) A box and an arrow consists of line segments. A line segment is one pixel wide. A line segment may be horizontal or vertical, but never diagonal. Two parallel line segments must be separated by at least one pixel. An arrow does not start at a corner of a box.

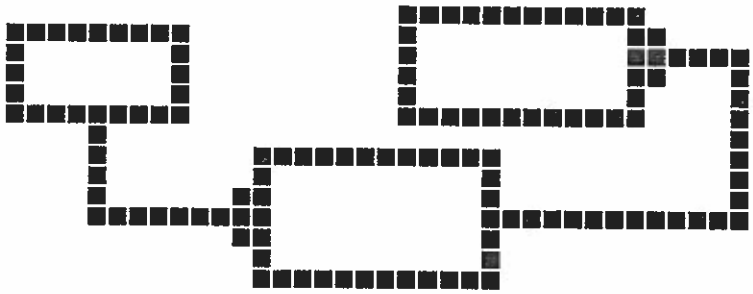


Figure 4: A legal box graph

The following configurations are not allowed.

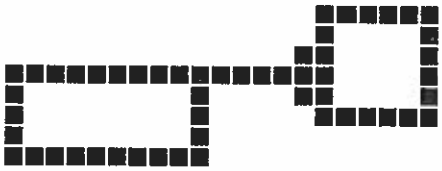


Figure 5: Illegal Box Arrow Combination

(2) A box is a rectangle enclosed by line segments. A box contains nothing and it has at least one empty pixel inside the enclosure.

Figure 6: The Smallest Box

(3) An arrow head is represented by extra two pixels as follows:

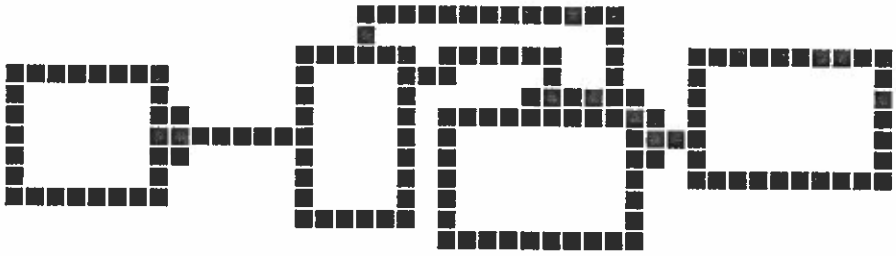


Figure 7: Legal Arrow Heads

The following configurations are not allowed:

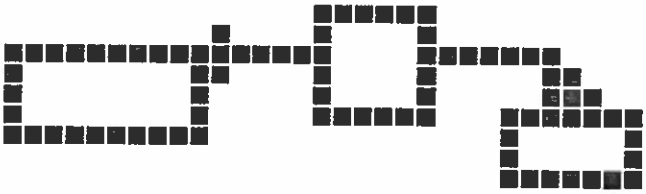


Figure 8: Illegal Arrow Heads

(4) There must be no cycle. The following is an example of cycle:

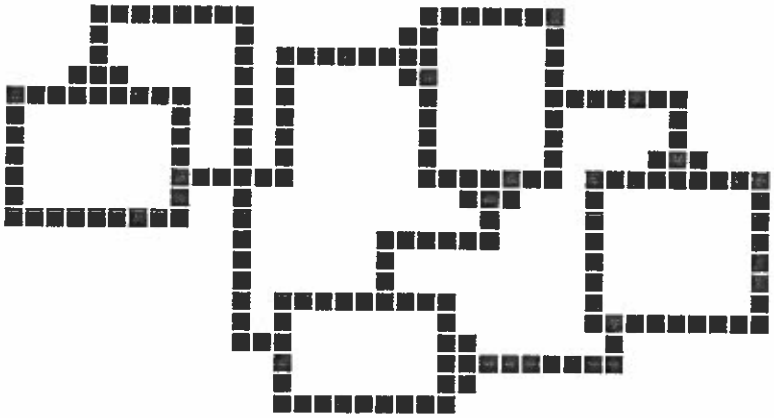


Figure 9: An Example of a Cycle in a Box Graph

3.4 Representation of the Box Graph

A box graph is made up of boxes and arrows (connecting directed lines). Arrows and boxes in turn are made up of pixels, single points of rectilinear grid. Although as individuals the points are indistinguishable, they represent different elements of boxes and arrows, i.e., corners, T-junctions (which are arrow tails), cross junctions, straight line segments and arrow heads. These elements form *basic building blocks* for any box or connecting arrow.

Decision of what a given non-empty pixel represents can be made locally by examining its neighbourhood. The neighbourhood about a pixel consists of all the pixels in a 3x3 window whose center is the given pixel. The pattern of the neighbourhood determines the role played by each non-empty central pixel. Figure 10 illustrates possible representation of basic buildings blocks where 1's denote non-empty pixels, 0's empty or background pixels and X's either of these two. There is also unique pattern code associated with each pattern.

Note that all fifteen patterns are mutually exclusive.

There is a restriction that no arrow can start from a box corner. This restriction excludes the possibility that a T-junction could represent a box corner and a tail of an arrow starting from that corner, or that a cross could represent a box corner and tails of two arrows starting from that corner. See Figure 11 for illustration.

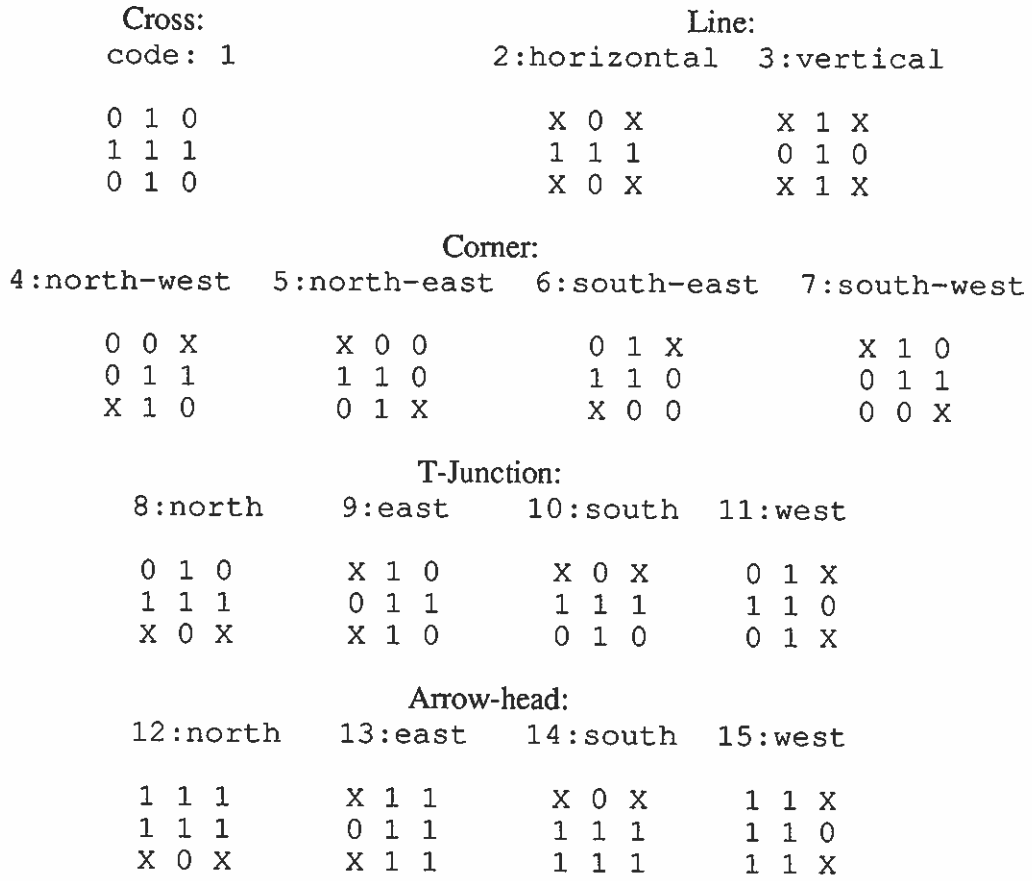


Figure 10: Basic Building Blocks

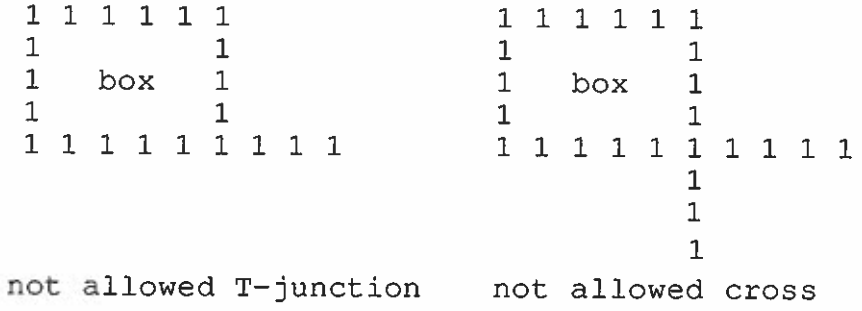


Figure 11: Illegal Patterns

The representation of the arrow head involves two additional non-empty pixels which are superfluous, they simply help to identify the arrow-head. This is not the only possible identification. There are many other but all, including the one considered here, have some drawbacks.

4. Recognition of the Box Graph

This section describes the procedure for recognizing a box graph which is initially defined by a set of non-empty pixels distributed over a square nxn grid.

The grid is mapped onto the square array of processors with one grid point assigned to one processor. Each processor has a *pixel register* where non-empty pixel is represented by 1 and background pixel by 0.

The recognition of the box graph proceeds in two stages. In the first stage processors recognize the basic building blocks by comparing the patterns of the neighborhoods with the predefined patterns of the basic building blocks. Note however, that basic building blocks still do not uniquely determine elemments of rectangles or arrows. A corner may be the box corner or turning point of an arrow. A line may be a part of an arrow or a side of a box. Basic building blocks together with specification whether they belong to rectangles or arrows form *box graph building blocks*. In the second stageof box graph recognition, the processors uniquely determine the box graph building blocks.

4.1 Identification of Basic Building Blocks

All fifteen patterns defining the basic building blocks (see Figure 10) are stored in local memories of all processors. Processors which correspond to non-empty pixels have to examine their neighborhood in order to determine what building blocks they represent. Succesive shift operations bring pixels from the neighborhood to the center processor. The pattern of the neighborhood is compared to each of the fifteen basic patterns. If there is a match to any predefined pattern, the pattern code is stored in the pattern code register (LPCR) of the center processor and the center processor is marked as initially recognized. However, for some non-empty pixels the pattern of the neighborhood do not match the pattern of any basic building block. This is a side effect of the representation of the arrow-

head. Only the pixels that serve as identifiers of arrow-heads may find themselves in such position.

The resulting ambiguity can be solved in the following way. The processors which were identified as arrow-heads transmit that information to those neighbors which served as markers of the arrow-heads. Because all arrow-heads were found, the markers are not needed any longer and can be erased. The erasing removes ambiguity. Now, by repeating the matching process for all "undecided" processors the identification of basic building blocks is complete, all non-empty pixels know exactly what they represent.

Note that the identification of the basic building blocks takes constant time, independent on n, the array size or even the graph size.

4.2 Identification of the Graph

Once basic building blocks are identified, the array is set to recognize the box graph. Processors have to decide whether they constitute a part of an arrow or a box. In order to make that decision processors must receive enough information possibly coming from the distant regions.

Prior to the computation all processors are in the initial states. In the course of the computation the processors change their states until the final states are assumed. The initial state corresponds to the starting information possessed by a processor which, in case of non-empty pixel, is the pattern code of the basic building block, while for empty pixels the pattern code is zero. The final state means that a processor recognized what element of the box graph it represents. The processor stores the pattern code of the graph building block, which is the code of the basic building block together with the indication whether the building block is a part of an arrow or a box.

For some processors the initial state is also the final state. This is the case with the processors corresponding to the background pixels. Similarly, the processors representing arrow-tails (which are T-junctions), arrow-heads and arrow crossings are in the final state to begin with. All other processors need some additional information before they are able to recognize what element of the box graph they represent.

All decisions (or state changes) are made based on states of some processors, *bounding processors*, lying in the same vertical and horizontal line as the processor making a decision. A bounding processor is such that contains information which may influance states of processors in the same horizontal or vertical line. First of all, any non-empty processor which is in the final state is a bounding processors. Corners, T-junctions, arrow heads and crossings are bounding for a processor which is connected to any of them by a straight line of non-empty processors. Any non-empty processor is bounding for another non-empty processor which is separated from it by a straight line of empty processors. In Figure 12 a possible situation is illustrated. Numbers are codes of basic building blocks. The top, the bottom, the leftmost and the rightmost processors are the bounding processors for the central processor.

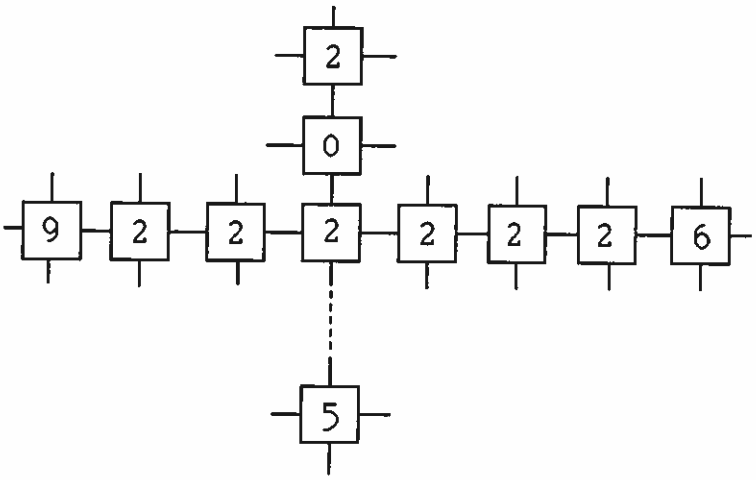


Figure 12: Bounding Processor

The array operates in the synchronous mode. Triggered by the outside control the processors start executing their individual programs. The program to be executed depends on the current state of the processor. All processors pass the data about their current knowledge to the north, east, south and west neighbours. Simultaneously, processors receive similar data from the neighbors. Based on the local data and the data received from the neighbours the processors update their states and also prepare data to be sent to the neighbours in the next cycle.

Each processor has four special registers NR,SR,WR and ER monitoring the states of the bounding processors above, below, to the left and to the right of it. Figure 13 shows some registers which are used for graph building blocks recognition.

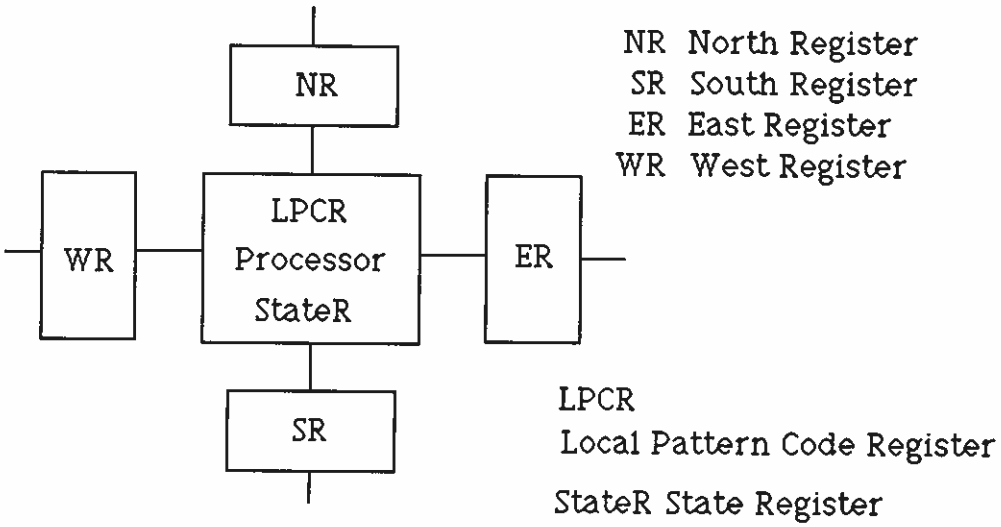


Figure 13: Processor Registers

The StateR register describes the state of the processor. When the processor is not in the final state, the content of StateR is 0. The content is A,B,E,H or T if the processor represtents arrow, box, empty processor, arrow-head or arrow-tail respectively.

The data which is sent by a processor to its neighbors is a concatenation of LPCR and StateR registers of either the processor itself or those of the bounding processors. All incoming messages are evaluated against the information possessed by the processor. A decision is made locally whether the incoming data has higher priority than the information evaluated by the processor. In the case when the incoming data has higher priority it is transmitted further in the same direction it came from. Otherwise, the processor transmits the locally evaluated data.

The data representing empty pixels have the lowest priority. The corresponding processors do not perform any computation but simply shift the incoming data along vertical and horizontal connections. The data representing non-empty pixels which are in the final state have the highest priority. The corresponding processors also do not perform any computation. They ignore the incoming data but keep sending the information what (final) part of the box graph they represent to the immediate neighbors. If both incoming data and local data represent the final states then, as the incoming data is ignored, the local data has precedence. In the case when the incoming data have the same code as the code of the local data, the local data have the higher priority. The data representing line have lower priority than the data representing any other building block. The local data representing any building block other than line have higher priority than any incoming data except when the incoming data represents the final non-empty state and the receiving processor is not in the final state.

4.3 Arrow Identification

If a processor is not in the final state but one of its non- empty neighbors is in the final state, then the final state of the neighbor uniquely determines the final state of the processor. That is if any of the nearest neighbors (north, south, west or east neighbor) represents an arrow then the processor must be a part of that arrow. Similarly, if a neighbor

represents a box then the processor is also a part of that box. Recall that arrow-heads, arrow-tails or crossings are in the final state from the very begining of the computation. As a consequence, any processor which is connected to an arrow-head, arrow-tail or crossing will assume the final state A or B on completion of the first unit of time . Next, one by one, succesive non-empty processors connected to arrow-heads, arrow-tails or crossings by a chain of non-empty processors, will receive data representing final state and they also will be able to determine their final states. In particular the following fact holds.

Lemma 1: Any non-empty processor connected to an arrow-head, arrow-tail or crossing by a straight line of non-empty processors will assume the final state in at most n units of time.

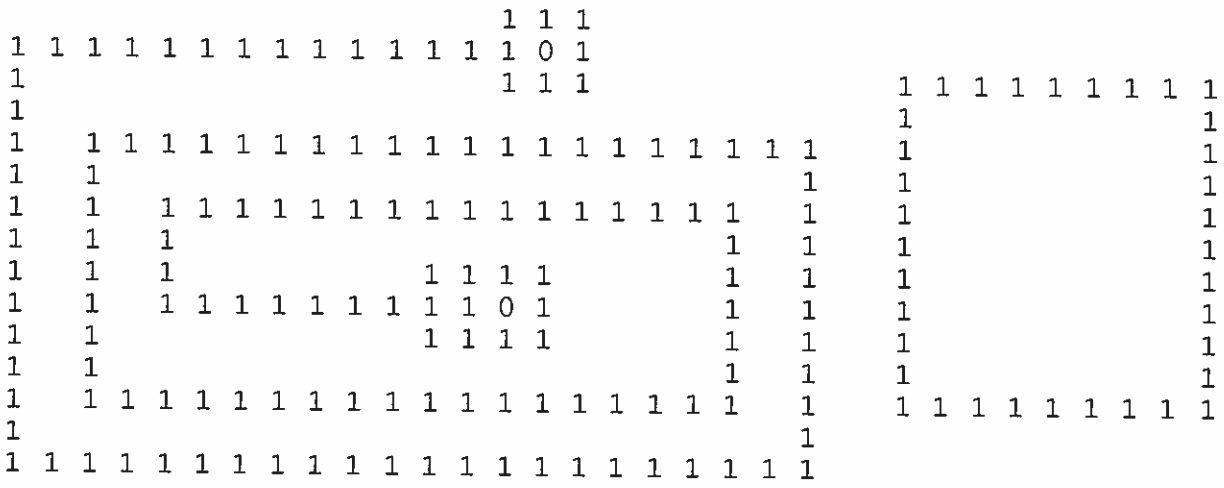
Proof: The lemma follows from the observation that the maximal length of any straight, horizontal or vertical, line is n. QED

Secondly, when a processor is a corner connected by a staight (non-empty) line to another corner of opposite orientation then both corners are parts of an arrow. (Two corners connected by a line have the *same orientation* when traversing the line either only right or only left turns are encountered, otherwise the corners have the *opposite orientation*). Similarly, for a processor representing a line, if both end of the straight part of the line are corners of oposite direction then the line is a part of an arrow. One of the processors between the two corners will receive the information that the opposite bounding processors are corners of opposite orientation. The processor will recognize that it is a part of an arrow. Next the message will spread along the line and eventually will reach the corners. Thus we have the following lemma.

Lemma 2: The recognition of a segment of an arrow between two corners of opposite orientation takes no more than 2n units of time.

Proof: The constant two comes from the fact that the information about two corners is combined in one of the processors lying in between and next propagated back to the corners. QED

The only hard situation is when following a line only corners of the same orientation are met. This can happen with a box or a *spiral*, see Figure 14 below.



spiral box

Figure 14: Spiral and Box.

A box has the property that its interior is "empty". Spiral on the other hand has one or more *coils* inside. This makes all but the innermost coils of a spiral easy to recognize. However, the innermost coil must end in a box so it is easy to recognize anyway. The procedure is the following.

As we mentioned before, any non-empty processor which lies on a line connecting two corners of the same orientation is either a part of a box or an arrow. However, if there is a coil inside then any corner of the inner coil will betray that.

Consider a processor which is the part of the outer coil and which lies on the same vertical or horizontal line as the corner of the inner coil. After at most n units of time (which is the

maximal distance in vertical and horizontal direction in the array), the processor will have the information that two of its bounding opposite processors represent corners of the same orientation, i.e., the processor represents either a part of a box or a spiral. However, the processor will also receive the message that there is a corner inside. Hence, the processor must represent a part of a spiral. This is conclusive, the processor assumes the final state. That final state is communicated along vertical and horizontal lines. Passing through all nonempty processors the information eventually reaches the two bounding corners which in turn can now decide that they are parts of an arrow. The procedure is simultaneously executed for all four sides of all outer coils. As a consequence we have the following lemma.

Lemma 3: Any spiral can be recognized in no more than 4n units of time.

Proof: All outer coils can be recognized in time at most 2n. This is because at most n units of time are necessary to propagate data from all three corners to the processor which makes the final decision. Additional n units of time are needed to propagate the message back to the corners. The innermost coil has the length at most 4n and its end must be an arrow-head or an arrow tail. As the end of the innermost coil is in the final state from the begining of the computation this fact will propagate along the innermost coil in time equal to the length of the coil which is at most 4n. QED

4.4 Box Identification

It remains to find a way for recognizing boxes. Without loss of generality, consider processors forming upper and left side of a box. Sides are bounded by corners of the same orientation with the upper-left corner being the only common point. Note that a box is convex and "empty" inside. Thus all processors lying on the upper and left sides have their *counterparts* on the lower and the right sides, only empty processors separate the opposite sides.

Now consider any north-west corner. This processor initiates search for a box. A signal is sent from the north-west corner to all processors forming upper and left sides (including bounding corners) to check on their counterparts. The signal propagates along both sides until it reaches the corners.

If any of the two corners has the opposite orientation than the north-west corner, that corner immediatelly concludes that the line is a part of an arrow. The corner assumes the final state, terminates the search and propagates the information back to the north-west corner. All processors in between, one by one, assume the final states. Next, the north-west corner assumes the final state and transmits this information to its neighbors. Now, the information spreads along the second side, changing the states of all processors on that side which are not in the final states into the final states.

On the other hand, if the corner has the same orientation that the north-west corner, the search continues. The corner which received the signal from the north-west corner (and has the same orientation as the north-west corner) replies by sending boolean value true back to the north-west corner. This boolean variable stops at each processor it passes. Each processor which the boolean variable visits holds the boolean variable until the check on the processor counterpart is finished. If the result of the check indicates that the counterpart may belong to the same box as the processor, the boolean variable is passed along the line, towards the north-west corner, unchanged. Negative check indicates that the processor is a part of an arrow. This is the final decision overruling other actions the processor may contemplate to take. The search is aborted. The information is sent to the neighbors and transmited further in succesive units of time. Eventually all processor on both sides starting from the north-west corner are informed.

In case when all checks are positive, the north-west corner will receive the boolean variable true. The search along one side is positive. If also the search along the second side is positive this means that the interior of the structure is empty. As only a box has this

property, the structure must a box. The north-west corner assumes the final state marking the corner as the box corner. The information is transmited to the neighbors which will propagate it further until all processors forming the box are reached. The recognition of the box is now completed.

Lemma 4: The recognition of a box takes no more than 4n units of time.

Proof: The message from the north-west corner must travel to the south-east corner and back to the north-west corner. QED

The main result of this section is the following.

Theorem 1: A well defined box-graph given as a colection of basic building blocks can be recognized in at most 4n units of time on an nxn array of processors The size of the local memory of any processor is independent on n.

5. Cycle Search

Let b be any box in the box graph G. By indeg(b) we denote the number of arrows pointing at the box b and by outdeg(b) the number of arrows originating at the box b.

The algorithm for establishing whether a box graph is acyclic is based on the simple observation that removal of all boxes, together with arrows originating at or pointing to them, for which indeg(b)*outdeg(b)=0 will leave the subraph cyclic if the graph is cyclic. If a graph is acyclic the procedure will render the subgraph empty.

The cycle search is initiated by boxes north-west corners. For every box the computation consits of two local phases, checking phase, which may be repeated number of times for the box, and erasing phase, which when entered is executed only once for that box.

In the erasing phase each north-west corner propagates two control bits, *head* and *tail*, along the box perimeter. The initial values for these bits are zeros. The value of tail is changed to one when at least one arrow-tail is found on the perimeter. Similarly, the value of head is changed to one when at least one arrow-head is found on the perimeter.

After at most 4n units of time (the maximal perimetr length of any box), the north-west corner receives both control bits. The next action depends on the value of the bits. When both are ones this indicates that indeg(b)*outdeg(b)>0, i.e., there is at least one arrow pointing to the box and at least one arrow leaving the box. The north-west corner sets both bits to zero and the checking cycle repeats.

In the case when indeg(b)*outdeg(b)=0 the north-west corner initiates erasing phase. A single control bit *erase* is propagated along the perimeter and further along all arrows leaving or pointing to the box up to the point where another box is met. When a processor receives the control bit 'erase' it marks itself as erased and transmits the 'erase' bit to all non-empty and not erased neighbors. When the 'erase' bit is received by a processor belonging to another box, which must be an arrow-head or arrow-tail, the processor replaces the content of LPCR register, which is the code of an arrow-head or an arrow-tail, by the code of a line having the same direction as the direction of the side the processor represented originally. This operation removes one arrow-head or arrow-tail from the second box, possibly changing the product of indeg and outdeg for that box. The (local) erasing phase terminates at this point, however a new (local) erasing may start at the second box.

It is clear that if the original box-graph is acyclic, the cycle search procedure will terminate leaving all non-empty processors marked as erased. The execution time is proportional to the total number of marked processors. We now prove this assertion. To do that we will use the standard graph terminology. In addition, by the *cardinality of a box* b, card(b), we mean the number of processors forming the box b, and by the *cardinality of an arrow*

(b1,b2), card((b1,b2)) we mean the number of processors forming the arrow (b1,b2). Finally, by cardinality of the graph G, card(G), we mean the total number of non-empty pixels forming the graph G. We are ready to prove the main result of the paper.

Theorem 2:

If a box-graph is acyclic then the cycle serch procedure terminates in time at most 3*card(G).

Proof: Without loss of generality we can assume that G is connected.

As G is acyclic there is at least one box s, a source box, such that indeg(s)=0 and outdeg(s)>0. Similarly, there is at least one box t, a terminal box, such that outdeg(t)=0 and indeg(t)>0.

Consider all source boxes b(1,1),b(1,2),..,b(1,i_1), and all arrows starting at these boxes. The arrows point at boxes b(2,1),b(2,2),...,b(2,i_2). Choose the source box for which the sum of the cardinality of the box and the cardinality of the longest arrow staritng at that box is maximal. Let it be the box b(1,1) and the arrow (b(1,1),b(2,1)). From the description of the cycle search procedure it follows that after 2*card(b(1,1))+card((b(1,1),b(2,1))) units of time all boxes b(1,1),b(1,2),..,b(1,i_1) and all arrows which started at these boxes are erased. (We take cardinality of the box b(1,1) twice as there are two traverses along the box perimeter, in the first travers the product of outdeg and indeg is calculated, and in the second, the erasing phase is processed).

Some of the boxes b(2,1),b(2,2),..,b(2,i_2) in the resulting subgraph may now become the source boxes for that subgraph and some may have been partially or fully erased. Again, consider all source boxes b(3,1),b(3,2),..,b(3,i_3) and all arrows originating at these boxes in the current subgraph. Note that the intersection of the sets {b(2,1),b(2,2),..,b(2,i_2)} and {b(3,1),b(3,2),..,b(3,i_3)} may be empty. Nevertheless, in the current subgraph there is at least one source box. Otherwise the subgraph would have been cyclis which is impossible

as the graph itself is acyclic. Choose the box and the arrow starting at that box for which sum of the cardinalities is maximal. Let it be the box b(3,1) and the arrow (b(3,1),b(4,1)). After 3*card(b(3,1))+card((b(3,1),b(4,1))) units of time all boxes b(3,1),b(3,2),.., $b(3,i_3)$ and all arrows which started at these boxes are erased. (We count the cardinality of the box b(3,1) three times as now there may be up to three trverses along the box perimeter. In the first travers the check for the product of outdeg and indeg may still be negative. In the second travers the fact that indeg=0 is discovered, and finally in the third travers the erasing phase is processed).

As there are no cycles in the graph, the procedure must terminate on one of the terminal boxes. This happens after

M=[2*card(b(1,1))+card((b(1,1),b(2,1)))]+...+

[3*card(b(2*m-1,1))+card((b(2*m-1,1),b(2*m,1)))]

units of time. But M<3*card(G) which completes the proof. QED

In the worst case the procedure takes order of n**2 units of time. The reason for this is that the longest "non-empty" path cannot exceed the overall number of processor, which is exactly n**2.

If there is a cycle in the graph, the procedure will not terminate without the intervention of the outside control. However, we know that after at most 3*n**2 units of time the acyclic graph would have been erased. Thus after that time, if there are still not erased processors in the array, the graph must have cycles.

6. Conclusions

We introduced a concept of a box graph and proposed a systolic algorithm for an nxn array of processors which dedects cycles in a well defined box graph. The algorithm is synchronous and its behavior in the worst case is proportional to n**2. The memory

requierement is constant per each processor. When a graph is acyclic the algorithm can recognize that fact in time proportional to the cardinality of the graph. However, it cannot discover a cycle until full time" n**2 is reached. It is an open question whether it is possible to detect cycle in time, say, proportional to the cardinality of a box graph. This question is related to the another one. We assumed that the array is synchronous. It is not known to the authors whether the cycle search procedure could terminate if the array operated in asynchronous manner.

In our considerations we assumed that the initial box graph does not have flaws and that no arrow starts or ends at box corners. The question is whether these assumption can be relaxed while maintaining the execution time proportional to cardinality of the graph.

7. References

[1] G. Raeder. "A Survey of Current Graphical Programming Techniques," *IEEE Computer*, August 1985, pp 11-25.

[2] H.T. Kung. "Why Systolic Architecture?," *IEEE Trans. Computer* 15(1):37-46.

[3] P. McLain and T. D. Kimura. *Show and Tell User's Manual.* Technical Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, March 1986.

Show and Tell is a trademark of Computer Services Corporation.

[4] W.D. Gillett and T.D. Kimura. Parsing Two-Dimesional Languages. Manuscript submitted to COMPSAC86, Chicago, October 1986.

[5] A.L. Fisher, H.T. Kung, L.M. Monier, and Y. Dohl. "The Architecture of a Programmable Systolic Chip," J. VLSI and Computer Systems 1(2):153-169 (1984).