An FPGA Based Graph Coloring Accelerator

L.M. Pochet, M. Linderman, R. Kohler, and S. Drager
Air Force Research Laboratory/Information Directorate/Rome Research Site

Abstract

This paper describes a Field Programmable Gate Array (FPGA) based graph coloring acceleration architecture. The approach described maximizes the use of local communication while still ensuring a complete search of the solution domain. A complete search of the solution domain has previously been ensured only through more global approaches. The graph coloring accelerator architecture can be used to speed up routing for Wave Division Multiplexing fiber optic communications systems and multi-hop radio communications.

I. Introduction

The Graph Coloring Accelerator Architecture was developed to improve the speed and efficiency with which Latin Square problems could be solved. Specifically, the targeted problem was Wave Division Multiplexing routing in fiber optic communications systems.

The Graph Coloring Accelerator Architecture was implemented on a field programmable gate array (FPGA). FPGAs allow a user access to all of the benefits of Application Specific Integrated Circuits (ASICs) without the costs and long turnaround time. Targeting an FPGA allows the designer to experiment with the accelerator architecture. The designer is able to vary architecture dependent parameters and test the impact of the changes on the actual hardware in a matter of hours instead of the weeks required to spin new ASIC designs. For example, the node processor cache size may be varied over some range allowing determination of the optimum size to occur based on results from the physical hardware. The choice of implication rules could also go through this type of experimentation.

The JAVA Hardware Description Language (JHDL) was used to design the Graph Coloring Accelerator Architecture for FPGA implementation. JHDL, developed at Brigham Young University (BYU) [5], allows greater flexibility in the parameterization of models than conventional VHDL. Architecture refinement is also simpler than when using the more behavioral VHDL. The structural basis of JHDL allows a designer to know exactly what hardware will be instantiated from the code written. JHDL minimizes or avoids common problems, such as differences between simulation and synthesized design often experienced with other HDLs when migrating from simulation to an actual chip. Finally, JHDL allows the full read back of the FPGA to be annotated on the circuit schematic. This allows simulation-like debugging to take place on the target hardware.

A Latin Square of order N is comprised of an n-by-n array of N symbols in which every symbol occurs exactly once in each row and column of the array [1]. The Latin Square involves the use of two sets of blocks, one of which is organized by rows and the other by columns. Latin-square designs have the character of a double randomized block design, where the experimental variable is confounded with the row and column interactions. Figure 1 is an example of a Latin Square.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 3 & 0 & 2 \\ 3 & 2 & 1 & 0 \\ 2 & 0 & 3 & 1 \end{bmatrix}$$

Figure 1: Latin Square Example.

The ability to complete Latin Squares is useful in many scheduling and routing applications. One such useful application of the Latin Square is the routing of Wave Division Multiplexing (WDM) fiber optic communication systems. These passive, all optical systems allow up to N wavelengths to be applied to each input without any output contention [3]. Optical systems that are routed with the use of Latin Squares are referred to as Latin Routers.

In a Latin Router, each row represents a separate routing node. Each routing node is able to accept N different inputs simultaneously. Each of these inputs is represented as a column in the Latin Router. Each input wavelength to a node will be routed to the node indicated by its column position. In Figure 1, for example, wavelength 2, when applied to the input of node 4 will be routed to node 1.

II. Accelerator Architecture

A. Problem Description

Completing a Latin Square is an NP complete task that has been shown to take considerable compute time on large (N>30) problems [4]. As N increases, the number of nodes required to solve for completion of the Latin Router increases to N^2. Thus as N grows, the time required to completely and correctly finish the Latin Router increases dramatically.

Not only is the time required to complete a Latin Square dependent on N, but also the completion time depends on the number of preset nodes. Preset nodes in a Latin Square correspond to pre-existing requirements on a scheduling system. A relatively sparse graph with few presets will have many solutions, and will be solved relatively quickly. The solution of a partially colored graph will result in a valid assignment of symbols to nodes or a determination that no such assignment exists.

The maximum number of nodes that may be preset in a Latin Square is 50%. After the 50% preset point is reached, all other nodes may be implicated. Implication takes place when a node has only one valid color remaining available to it, the node then selects this color to be its selected color until a backtrack is required. Further, as the number of preset nodes approaches 50%, the graph may also be solved relatively quickly. This is because there are fewer combinations to try before a solution is found or it is discovered that no solution exists.

The number of backtracks, and thus the number of incorrect guesses, reaches a maximum when approximately 40% of the nodes are pre-assigned. A backtrack occurs when a contradiction is arrived at, after the contradiction is discovered, the guess that caused the contradiction is undone. This data was collected empirically by Gomes [**4**] and is shown in Figure 2.

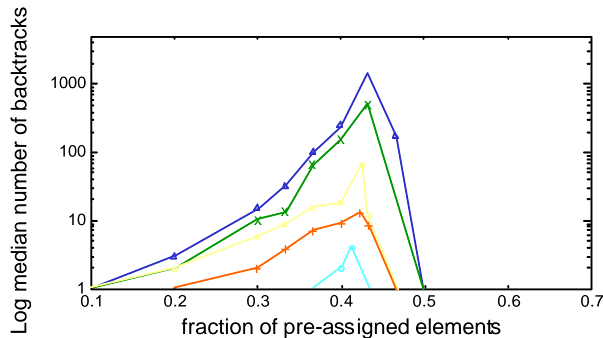


Figure 2: The Complexity of Quasigroup Completion (Log Scale)

Local search methods are usually able to solve partially colored graphs faster, due to the increased parallelism available. However, local search methods cannot assure that all possibilities have been tried. This presents a problem, as local search methods are not able to tell when there is no possible solution to a graph. On the other hand, global search methods assure that all possibilities are tried. However, global search methods are generally slower because less parallelism can be exploited.

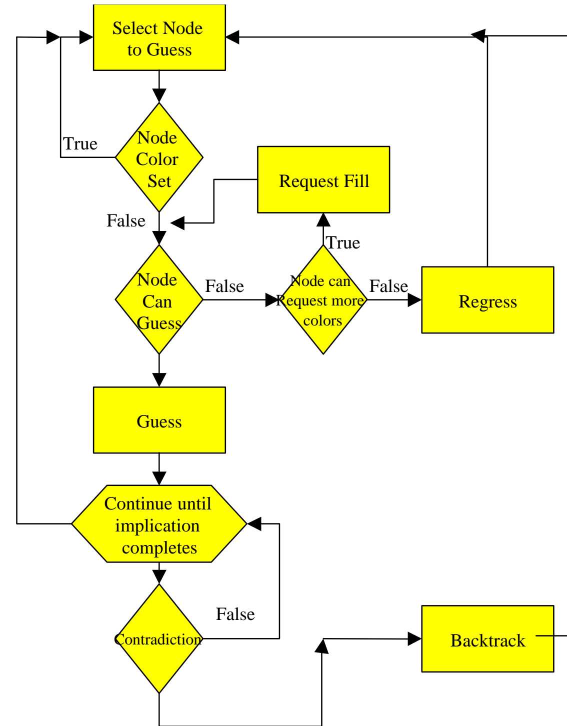


Figure 3. Flow diagram of graph coloring algorithm.

B. Algorithm Overview

There were two design goals for the Graph Coloring Accelerator Architecture. First, the architecture must be scaleable for problem sizes of N=1 to N=40. Second, the algorithm must maintain as much parallelism as possible.

A high level flow diagram of the graph coloring algorithm process is shown in Figure 3. The algorithm works by exploring the search space starting from an initial partial coloring of the nodes. Through a recursive sequence of guesses, implications and backtracking, either a solution is found or it is proved that no solution exists.

The state of the graph is comprised of the states of the nodes of the graph. The state of a node is exactly one of PREASSIGNED, GUESSED, and UNASSIGNED. Nodes in either of the first two states have been assigned exactly one color. The color was assigned to the node by the initial conditions, subsequent guesses of previously UNIMPLICATED nodes, or implication of previously UNASSIGNED or GUESSED nodes. UNASSIGNED nodes may still be assigned one or more colors based upon the implication rules implemented. The set of colors that may be assigned to an UNASSIGNED node comprise the set of all colors minus the colors that have been proved to contradict the color assignments to the PREASSIGNED and GUESSED nodes.

The implication rules are a specified function over the state of the graph that monotonically decreases size of the color sets of the UNASSIGNED nodes. The function is applied iteratively until the function no longer changes the state of the graph or a contradiction is detected. An UNASSIGNED node indicates a contradiction when its set of possible colors is reduced to the empty set. In the absence of a contradiction, a stable state (denoted S*) is reached when further application of the implication function (denoted *I*) does not change the state of the graph (i.e. $I(S^*) = S^*$). The implemented algorithm waits until the graph reaches a steady state before another guess is made, although this is not generally a requirement.

Our implementation assigned colors to the nodes in a fixed order. While this is not generally necessary, it simplifies the control logic. In particular, it makes recursion simple and guarantees that if the first assigned node has no valid color assignment, then no solution exists. Other 'local' searches do not make this assumption. While this often results in a faster solution to the problem, it makes it more difficult to prove that no solution exists because it is difficult to prove that all possibilities have been tested.

If a contradiction is detected, the last ASSIGNED node is assigned a different color and implication is performed. If a contradiction is again detected, the node is assigned yet another untried color, and the process is repeated. If a stable state is reached, then the search recurs one level deeper as an UNASSIGNED node is selected and assigned a color. If all possible colors have in turn have been assigned to a node, and each has resulted in a contradiction, then the algorithm backtracks up to the previous level of recursion. We call this regression.

Our implication function is computed at each node by removing from its set of possible colors any color assigned to another node in the same row or column. Additional implication rules (currently unimplemented) include detection of colors excluded from all nodes within a row or column save one. There are several others implication rules that could be implemented to allow nodes to implicate sooner but this would increase the complexity of the node processor. The architecture of the FPGA requires a tradeoff between computational complexity and number of nodes that can simultaneously be implemented since the computational resources are fixed.

Like the related problem of transitive closure, the order of implication among the nodes does not affect the final state, and therefore the implication of one node can occur in parallel with implications of others. If a contradiction exists, only the location of the contradiction may change, not the occurrence of one. In the absence of a contradiction, the state of the nodes after implication is the same regardless of the order of implication.

C. Design Description

An FPGA architecture was developed comprising of an array of (N x N) small node processors for each node of the Latin Square, an edge controller for each row and column of the graph, and a graph master to manage host interaction and the guessing process. Each edge controller keeeps a LIFO (Last In First Out) list of guesses and the implications caused by the guesses. Use of the guess LIFO simplified backtracking by enabling the edge to step back in time to state where no contradictions existed. Figure 4 shows a high level schematic of the architecture where circles represent edge control, rectangles represent node processors, and the graph master is represented with a triangle.

The communication mesh is a simple 2D toriodal mesh of unidirectional busses. These busses comprise a tag identifying the data being transferred and a data bus $\log_2 N$ bits wide. There are two bus cycles. On the first cycle (referred to as the column focus), the nodes listen for communications from the column edge processor. At the same time, node processors can send information to the row edge processor. The second cycle (row focus), node processors receive from the row edge processor, and transmit to the column edge processor. The cycles are synchronized so the node receives data from the row processor on one cycle, and the column processor on the other, giving the appearance of continuous communication with an edge processor. Edge to node communication consists of remove, backtrack and guess broadcasts as well as memory loads directed at individual nodes.

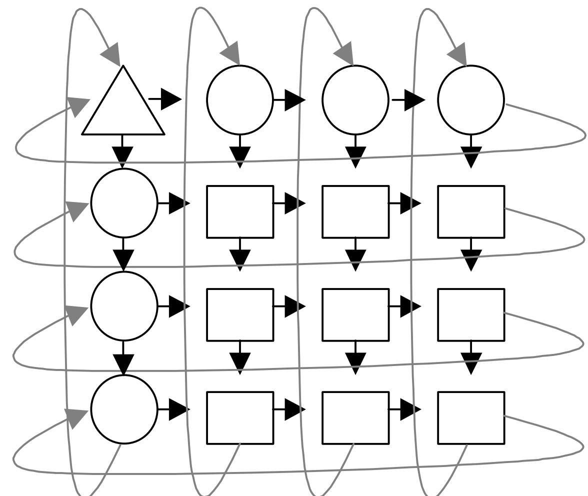


Figure 4. High level view of Latin Square coloring architecture

The bus that connects the edge processors is comprised of a tag and an N bit wide data bus. The data representation

chosen does not scale as well as desired, but considering that there are only 2N edges, and a considerable speed up in node memory fill calculation, the use of an N-bit wide data bus was an acceptable tradeoff. Edge to edge communication consisted of node processor memory fill calculations, as well as guess and backtrack control. Using an integer representation would have required one cycle for each color transmitted to the edge controller creating the fill packet to be sent to the node processor. A one hot representtion allowed all available colors to be transmitted in one cycle, it also simplified the union operation required to create a fill packet. To create a fill packet, the edge controller must perform a union of colors available to the node from its row controller and its column processor.

D. Component Detail

This section describes in detail each component (Node Processor, Edge Controller, and Master) of the Latin Square coloring architecture. It will be shown how design decisions support the overall direction and evolution of the design.

1. Node Processor

The Node processor is a minimal processor able to request memory fills, implicate when only one color remains in its set of possible colors, backtrack when its set of possible colors is empty, guess a color from its memory, and remove colors from its local list. It is connected to adjacent nodes through a unidirectional node bus that starts and ends at the edge controller. A simple view of the node processor can be seen in Figure 5.

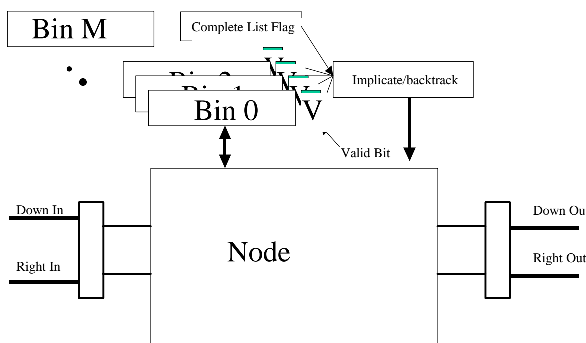


Figure 5. Functional diagram of a node processor.

Bins are used to hold the current subset of possible colors available to the node. Each bin holds a color and a bit indicating whether the bit is valid or not.

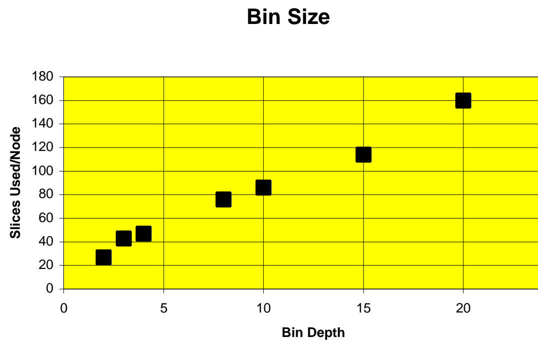
An implicate/backtrack controller counts the number of valid bits and requests an implication when the complete list flag is thrown and there is only one valid color remaining. It requests a backtrack when the complete list flag is thrown and there are no colors remaining in the bins, this indicates there are no colors available for the node.

As described earlier, there are two parts to a communication cycle, column focus and row focus. During the column focus stage, information sent from the column controller arrives at the node and is processed; the node also sends information toward the row processor. On the other half of the cycle, the opposite happens, data is retrieved from the row controller and sent to the column controller. When the node bus is idle, a node is free to send requests on the node bus; otherwise, the input is passed through to the next node on the node bus. Requests written to the node bus must first pass through all nodes below (or to the right of) the sending node before arriving at the edge controller.

The most important consideration during the node processor's design was scaling. Graphs too large to fit on a single FPGA will be time multiplexed with partial row column blocks completed in a windowing technique to be described later. Each node processor, and the required interconnect between node processors, grows as a function graph size grows. Effort was made to minimize the impact of graph size on node processor size.

The interconnect between each node processor and from each edge processor to the adjacent node processor grows O(Log2(N)). Colors are sent one at a time across the node bus. The most demanding task for the node bus is the memory fill, which is a constant value regardless of the size of the graph and requires multiple colors to be sent serially. Node processors require memory fills to update their bins with colors available to the node when the current set stored locally has been exhausted through remove operations. For remove requests, only one color, the color to be removed, is sent

Node processor memory (also known as a bin) also grows as the graph size increases. Node memory holds a subset of the complete list of available colors, this subset contains M colors where M is less than or equal to N. Each node also holds a flag indicating whether the list of M colors is a complete list, or a partial list, of possible colors. As colors are removed from the list due to implication, more are requested from the edge controllers. This paging process involves both row and column controllers. The set of colors available to the node requesting data is the intersection of colors available to the row and column. Each edge controller contains a list of colors available to the row or column it controls. A row controller receives a fill request, and sends a list of available colors to the column controller. The column controller performs a union of the two sets and sends the first M colors to the requesting node serially through the node-wise bus.



As node size increases, there can be fewer nodes on a given sized FPGA. Eventually, node size increases to a point where there are not enough nodes on a given sized FPGA for useful computation. One solution to this problem is decreasing the amount of logic required for node memory bins by decreasing bin depth, M. For example, if M is set to 7 and node size becomes prohibitive because a large Latin Square is being attempted, M can be decreased to 4 to decrease the size of each node. The penalty for this reduction is more numerous and frequent fill requests. Total memory requirements for each node are M x $Log_2(N)$. As N increases, M can be decreased to maintain a constant node size. Decreasing M incurs an increased overhead due to more frequent paging requests from empty node processors, but allows more control over the size of each node processor.

The bins contain the only information specific to the node processor. If filled with different data, the same processor could be used to represent a different node. When coloring a graph too large to fit on a single FPGA, a windowing scheme will be used to clean the memories of each node processor and refill them with colors specific to a different grouping of nodes. Nodes with their colors already set (implicated or guessed) must also be noted and restored when windowing.

2. Edge Controller

Two edge controllers supervise each node processor. One edge controller governs the row information, and one governs the column information. Each edge controller keeps a list of colors available to the node processors in its row or column. This list is N bits long and each bit represents a color. As a color is removed from the list, the corresponding list is changed to '0'. When the color is added back, it is returned to a '1'. The edge controllers also keep a stack of previous lists, each time a guess is made, the previous list is pushed onto the stack. When a regression is requested, the old list is popped off the stack and returned to the edge controller list.

The edge controller's primary function is to handle fill requests from node processors. This action requires input from both edge controllers that supervise the requesting node. The side edge sends the current list of available colors to the top edge through the edge bus that consists of an N-bit wide bus, and a data tag. When the top edge receives the list, it obtains the intersection of its list of possible colors and transmits a fill packet to the requesting node containing up to M colors as well as a flag indicating if the list transmitted is complete or partial. A complete list can be implicated off of once only 1 color remains in memory. A list that is only a partial listing of partial colors cannot be implicated off of.

When windowing to color large graphs, the color stack and the current color list must be stored for each edge controller windowed off chip. After the edge controller's data is stored, the color stack and current list for a new edge are passed in to the edge controller's stack and current free color list. This operation proceeds much as a context switch.

The top edge controllers handle only a small part in the setting of initial conditions. The first fill of each node processor is started at the graph master, not the side edge. The graph master sends a color list consisting of 1 color to the top edge. The top edge treats this as a normal fill request and sends a list consisting of 1 color to the node. The node immediately implicates to the color received.

3. Graph Master

The graph master handles all host-FPGA communication. When migrating to different FPGA platforms, only the graph master must be changed. Host communication consists of obtaining initial conditions, and reporting results.

The graph master also handles all guessing and backtracking. Currently, when the graph master counts a set number of idle cycles on the edge bus, another guess command is sent. A guess command consists of a guess command tag and the row/column of the node to make a guess. Ultimately, it is preferable to detect when the implication process is complete rather then wait a fixed number of cycles. It should be noted that it is not necessary that implication proceed to a stable state before assigning a value to a previously UNASSIGNED node.

The only backtrack control required of the graph master is to ensure that every guess causes a maximum of one backtrack. If multiple contradictions are caused by a single guess, only one backtrack is required to return to a valid state.

The graph master, having the ability to communicate to off chip memory also controls the windowing process to color large graphs. This control involves finding all nodes with set colors, storing all data from the edge controllers windowed out, loading new edge controller data, and filling all nodes with their new lists and preset colors.

E. Results

The program developing the graph coloring architecture described above has existed for six months, the results reported are only preliminary results. No effort has been made to pipeline the design to allow for a faster clock, and it is still able to consistently color only simple graphs. Because no windowing scheme was been implemented to date, the physical constraints of the FPGA used dictate graph size.

The design described above has been targeted for an Annapolis micro Systems WildStar board populated with Virtex 1000 chips containing one million gate equivalents and 32 4k deep block ram units. A graph where N=6 requires 36 node processors, 12 edge controllers, and one graph master was implemented on one of the three available FPGAs. Almost 6,000 slices, and 12 block rams are required for this implementation. Xilinx back end tools report a maximum clock speed of 40 MHz.

The graph coloring accelerator can be used to color a graph of any preset without re-synthesizing the design circuit. This is achieved by allowing the graph master to preset each node immediately following a reset signal with presets found in off chip memory. Currently, all nodes are preset with either 1 or N colors, presetting to N colors equates to no constraint on the color of the node. This was done to maintain a simple design and will be changed for performance purposes in the future. Presetting a graph of N=6 consumes 140 clock cycles. This is due to a two clock cycle latency added at each node processor and edge controller for data passing through. There is also an N+10 cycle latency for preset information from an edge to the start of the node controller row or column. This is due to latency in the edge controller FIFO stack catching information to send to node processors.

Currently, a naïve guessing scheme is employed. When all implications possible have been completed, a guess is made. This end has been reached when all node busses have been idle for the longest possible propagation time from one node to another. This time is currently 140 clock cycles. The wait time is the worst case cycle count for node bus activity to propagate back to the graph master. The graph master waits for all node busses to be idle for 140 clock cycles before ordering another node to guess. With this in mind, 4,000 clock cycles are required to color an empty graph where there are no contradictions or backtracks. For each graph colored, there is an overhead of approximately 4,200 clock cycles.

F. Future Work

In its current form, the graph coloring accelerator cannot window to color larger graphs than can fit on a single chip. This limits an implementation on a Virtex 1000 (1 million gate equivalent FPGA) to coloring a graph of size N=6. This includes 36 node processors, 12 edge controllers and one graph master.

A windowing scheme will allow extremely large graphs to be colored by time multiplexing the hardware to simulate the existence of more hardware (Figure 7). This approach will also allow multiple FPGA's to be used in parallel to color a graph. Using this technique, a guess would be made and allowed to propagate across the window. Once all implication in the window completed, the window could be shifted by swapping different row or column controllers and allowing the nodes to refill. Implication could then continue on the window even though the guess was made on a different context. There would be a run time penalty because some implications in later windows would cause implications in earlier windows therefore, the windowing process would have to continue for each guess until no more implications were found. This requirement equates to a requirement for a complete window circuit (each node placed on hardware once) where no implication took place.

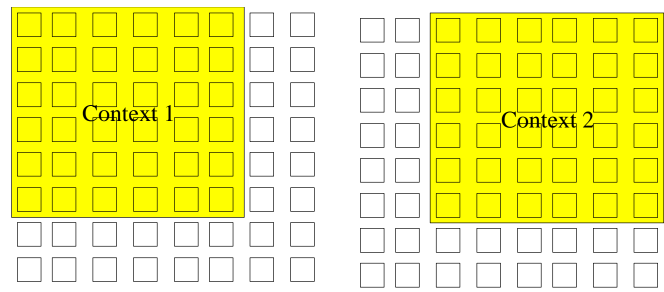


Figure 7. Example of two subsequent windows to complete a Latin Square larger than 6 x 6.

Refinements to the windowing process could be made to group trouble areas of the graph together. For instance, if column 2 and column 43 produced a disproportionate number of backtracks, they could be shifted into the same window to allow contradictions to be found faster, without excessive windowing required. This reordering would allow incorrect guesses to be discarded faster.

The naïve guessing control of the current implementation will also be refined. Guessing colors for highly constrained nodes before guessing for relatively unconstrained nodes would decrease the overall number of incorrect guesses, since fewer guesses are available to highly constrained nodes.

ACKNOWLEDGEMENTS

The Air Force Office of Scientific Research sponsored this research.

REFERENCES

[1] J. Denes and A.D. Keedwell, *Latin Squares and Their Applications*, Academic Press, New York, 1974.

[2] Kichul Kim and V.K.P Kumar, The 16th Annual International Symposium on Computer Architecture, New York: IEEE Press, 1989, pp. 372-379.

[3] Richard Barry and Pierre Humblet, Latin Routers, Design and Implementation, Journal of Lightwave Technology, 1993, pp. 891-899.

[4] Carla Gomes, Selman B., Crato N., Heavy-tailed Distributions in Combinatorial Search, Principles and Practices of Constraint Programming, 1997, pp. 121-135.

[5] Bellows P and Hutchings B, FCCM 1998 Proceedings, 1998, pp 175-184.