# NATR: A New Algorithm for Tracing Routes

Prasad G. R.
NITK, Surathkal
grprasad_bms@yahoo.com

Dr. K. C. Shet
NITK, Surathkal
kcshet@yahoo.co.uk

Dr. Narasimha B. Bhat
Manipal Dot Net Pvt. Ltd., Manipal
narasim@manipaldotnet.com

*Abstract*— **This paper presents NATR ("New Algorithm for Tracing Routes"), a new shortest path algorithm using reconfigurable logic and has time complexity O(L), where L is shortest path length. It uses ball and string model and is highly parallel and scalable. Unlike most other shortest path algorithms, NATR does not need to find the minimum of nodes/adjacent nodes. Hence its FPGA implementation is faster compared to other FPGA implementations. Preliminary experimental results show that a 17-node NATR runs about 6.3 times faster compared to parallel Bellman-Ford algorithm on Xilinx Virtex II.**

## I. Introduction

Shortest path (SP) problem in graphs is still an active area of research[4], due to the demands for faster SP algorithms by applications like CAD for VLSI[8], robotics[6] and computer networks[3][5]. SP algorithms which run on instruction set based processors, like Dijkstra's[1] algorithm and others[4][7], iterate hundreds of instructions and are sequential in nature, and hence have high computation time. Reconfigurable logic based approaches have been used in the past [3][9] to accelerate SP algorithms. But they are slowed down by the process of finding minimum of nodes/adjacent nodes.

Reconfigurable computing[2] achieves high performance by spatially spreading computation on hardware instead of iterating hundreds of instructions on a processor. Reconfigurable computing has execution time close to ASICs with flexibility to reconfigure. It can be used to efficiently and effectively mimic "natural" solutions: an implementation that replicates the way nature tackles analogous problems.

This paper presents NATR ("New Algorithm for Tracing Routes"), a new SP algorithm using reconfigurable logic and has time complexity O(L), where L is shortest path length. It avoids finding minimum of nodes/adjacent nodes and hence is faster compared to other approaches. It mimics the formation of ball and string model[5]. In NATR, nodes fall down synchronously from the source by comparing their position with positions of adjacent nodes, and stop at shortest distance from source. Given adjacent node's information, a node can move independently and this makes NATR scalable. NATR is intended for large graphs in which $L < N$, where N is number of nodes. NATR assumes undirected graphs and positive integer edge weights.

The rest of the paper is organized as follows. Section 2 explains related work for finding SP, the ball and string model and its formation. NATR and its implementation details are given in Sections 3 and 4 respectively. In Section 5, we implement parallel Bellman-Ford algorithm, as it takes 'P' clock cycles to find shortest path, where 'P' is maximum of number of edges along the shortest paths from source to other nodes. Section 6 presents experimental results and compares NATR with other approaches. Section 7 suggests future extensions and the paper concludes with Section 8.

## II. Related Work

### A. Shortest path algorithms

Dijkstra's algorithm[1] is a popular SP algorithm and has time complexity $O(N^2)$. Let $x_i$ be the current distance of node 'i' from source and D be the adjacency matrix. When there is no edge between nodes 'i' and 'j', $D_{ij}$ is set to large value to indicate infinity. Let 's' the be source and 'd' be the destination. Dijkstra's algorithm is as shown in Figure 1.

```
1. Initialize $x_j$ to $D_{sj}$, where j =1 to N, and $j \neq s$, set $x_s$ as 0.
        Add 's' to set of labeled nodes F={s}
2. Find minimum among $x_j$
        $x_i$=Min($x_j$) for j= 1 to N and node 'j' not in F
3. Add node 'i' to F and update each node's distance using
        $x_j$=Min($x_j$ , $x_i + D_{ij}$ ) where j=1 to N
4. Repeat steps 2 and 3 until destination is reached
```

Figure 1: Dijkstra's algorithm.

Dijkstra's algorithm has been improved using efficient data structures like radix heap and two level radix heap[7], and have time complexities O(M+NlogC) and O(M+NlogC/loglogC), where C is edge weight, M is number of edges and N is number of nodes. A recent improvement[4] has time complexity $O(M+D_{max}\log(N!))$, where $D_{max}$ is maximal number of edges incident at a vertex. Implementation of Dijkstra's algorithm on reconfigurable logic is presented in [3] and this uses a comparator tree to find minimum instead of using a loop. But the algorithm has to repeat steps 2 and 3(Figure 1) until destination is reached and hence its time complexity is O(N). Ralf Moller[6] has reformulated Dijkstra's algorithm and implemented that using the concept of signal propagation, and has time complexity O(L).

### B. Ball and String model(BSM)

*Ball and string* model[5] of a graph is a network of balls connected by strings, where balls and strings represent nodes and edges respectively. For graph in Figure 2, Figure 3 shows equivalent BSM. In BSM, a straight line is a fully stretched string and a curve is a string with slack.

To illustrate formation of BSM from the graph, let v1 be the source. Assume all balls(nodes) are together as shown in Figure 4 and are at a distance of 0 from source. Source is fixed, which is shown by hatching. A fixed ball cannot move down. Keeping source fixed, when other balls are released, they fall down as shown in Figures 5 to 7. Figure 5 shows positions of balls after they fall down by unit distance and at this point no string is stretched to full. Figure 6 shows positions of balls after they fall down by distance of 2. Now, the string between v1 and v3 is stretched to full. Hence v3 cannot fall beyond 2 and gets fixed at 2. After falling by a distance of 3, balls are as shown in Figure 7 and at the end we get BSM as shown in Figure 3. In BSM, all strings along the shortest path are stretched to full and other paths will have one or more slacks.

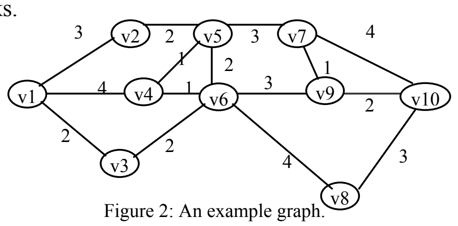

Figure 2: An example graph.

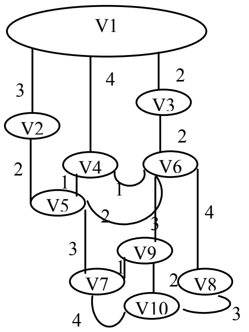

Figure 3: Ball and String model for the graph of Figure 2.

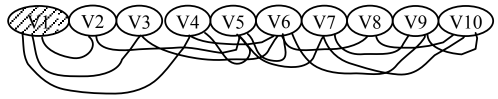

Figure 4: Initially all balls are together and v1 is fixed.

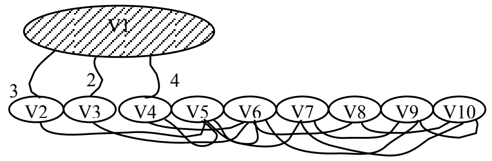

Figure 5: Balls after moving by distance of 1.

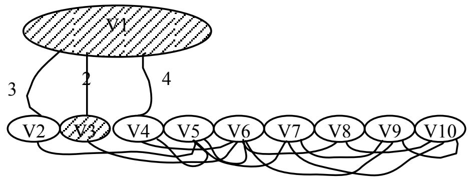

Figure 6: Balls after moving by distance of 2, v3 is fixed.

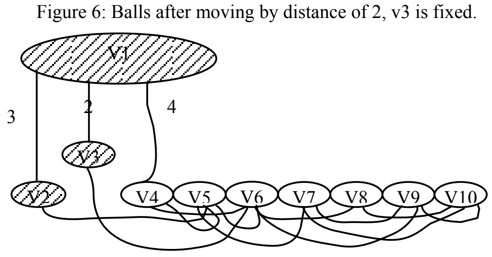

Figure 7: Balls after moving by distance of 3, v2 is fixed.

In [5], BSM is used to rebuild shortest path tree(SPT), whenever SPT gets disturbed due to changes in edge weights. This problem is represented as a linear programming problem and is solved to get new SPT. NATR is a simple approach to find shortest path and it uses reconfigurable logic, and hence is much faster compared to [5].

### C. Problems and opportunities

Most of the existing algorithms are sequential in nature(select nodes one by one) and find minimum of nodes/adjacent nodes, and hence have high computation time.

In BSM, strings will have lengths equal to corresponding edge weights in graph and hence a closer node will have shorter length and gets fixed first, thus eliminating the need for finding minimum. In addition, during the formation of BSM, nodes that are at the same distance from source get fixed in parallel(like, v4 and v6), which overcomes sequential selection of nodes.

## III. NATR

NATR mimics the formation of BSM to find shortest path. In NATR all nodes fall down from source synchronously. They fall under the constraint of not breaking any strings and stop at shortest distance from source. To fall down, a node needs information about adjacent nodes and this consists of adjacent node position, its status(whether node is fixed or movable) and weights on edges connecting the adjacent nodes(from adjacency matrix, D). Given this information, a node can fall down independently and this makes NATR scalable. Each node consists as its information; node position, status flag, step_size and previous_node. Initially for all nodes, position is set to 0 and step_size is set to 1. For all non source nodes flag is set to 0 and for source it is set to 1. Logic behind a node's move is as said below.

- Find next position(new $X_i$) of node 'i' by adding its position value and step_size.
- Find the actual distance($Dist_j$) from each of its adjacent nodes using $Dist_j = X_i - X_j$ where $1 <= j <= N$.
- Compare $D_{ij}$ with $Dist_j$ set flags $C_j$ and $E_j$ indicating less and equal respectively.
- If any of $C_j$ is 1, then move is failure(as string connecting node 'i' and 'j' breaks) and old position is retained.
- If none of the $C_j$'s are 1 then move is successful and position is set to new position found in step1.
- On a successful move, if any $E_j$ is set to 1(string is stretched to full) with corresponding $O_j$ is set to 1, then node 'i' gets fixed through 'j' and 'j' is set as previous_node of 'i' and status flag $O_i$ is set to 1.

Here initially step_size is set to 1. Later at each clock cycle, if move is successful step_size is multiplied by 'k', where 'k' is acceleration factor. Whenever a move is failure, if step_size >=k then it is divided by 'k' else step_size is set to 1. Step_size is varied to accelerate node's move. NATR with k=1, takes exactly L clock cycles and hence its time complexity is O(L). For k=1, the moves are as shown in Figures 4 to 7, and at the end it looks as shown in Figure 3. With k > 1, NATR takes less than L clock cycles to find shortest path, for large edge weights. But for small edge weights it takes more than L clock cycles due to excessive failed attempts. So, for small edge weights (<5) k=1 will be efficient. Figures 8 to 10 show the moves in NATR for k=2. In first clock cycle all nodes move by distance of 1 as shown in Figure 9 and