CSEE 4840

Embedded Systems

Dijkstra's Shortest Path in Hardware

Under the Guidance of – Prof. Stephen Edwards

Ariel Faria (af2791)
Michelle Valente (ma3360)
Utkarsh Gupta (ug2121)
Veton Saliu (vs2519)

Columbia University
Spring 2015

Contents

1. Introduction

Here, we present a parallel implementation of Dijkstra's shortest path algorithm. The algorithm was implemented in SystemVerilog in the Altera Quartus II 13.1.1 environment. The target for the design is the Altera Cyclone V System on Chip.

1.1. Motivation

A single-source shortest path (SSSP) problem requires finding the path of minimum total weight, given a source vertex s and a destination vertex t in a given n-vertex, m-edge directed graph G with the given edge weights. This is one of the most fundamental and extensively studied problems in Computer Science and Network Optimization. The most well-known and classic algorithm to solve the SSSP is Dijkstra's Algorithm.

The algorithm finds offline and real time applications in various areas such as path planning in mobile robots, telecom networks and segmentation in image processing. In particular the motivation for our work was "maze routing" – as an example, a specific application of such an algorithm could be in CAD where conductive tracks on a printed circuit board have to be routed between pins of components without overlapping or crossing other tracks.

Dijsktra's algorithm in software is implemented largely with the help of a priority queue, which is a sequential data structure. As the number of nodes in a network increase, the advantages of parallelizing this queue become apparent, given the quadratic growth of the algorithm. We wanted to exploit this and implement an efficient parallel version of the Algorithm on reconfigurable hardware.

1.2. Dijsktra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$. Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u. In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.. The pseudo code below gives the implementation of Dijslktra's algorithm and the execution of the algorithm can be seen in an example in Figure 1.

```
function Dijkstra(Graph, source):

dist[source] ← 0 // Distance from source to source
prev[source] ← undefined // Previous node in optimal path initialization

for each vertex v in Graph: // Initialization
if v ≠ source // Where v has not yet been removed from Q (unvisited nodes)
```

```
            dist[v] ← infinity // Unknown distance function from source to v
            prev[v] ← undefined // Previous node in optimal path from source
        end if
        add v to Q // All nodes initially in Q (unvisited nodes)
    end for

    while Q is not empty:
        u ← vertex in Q with min dist[u] // Source node in first case
        remove u from Q

        for each neighbor v of u: // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]: // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u
            end if
        end for
    end while

    return dist[], prev[]

end function
```

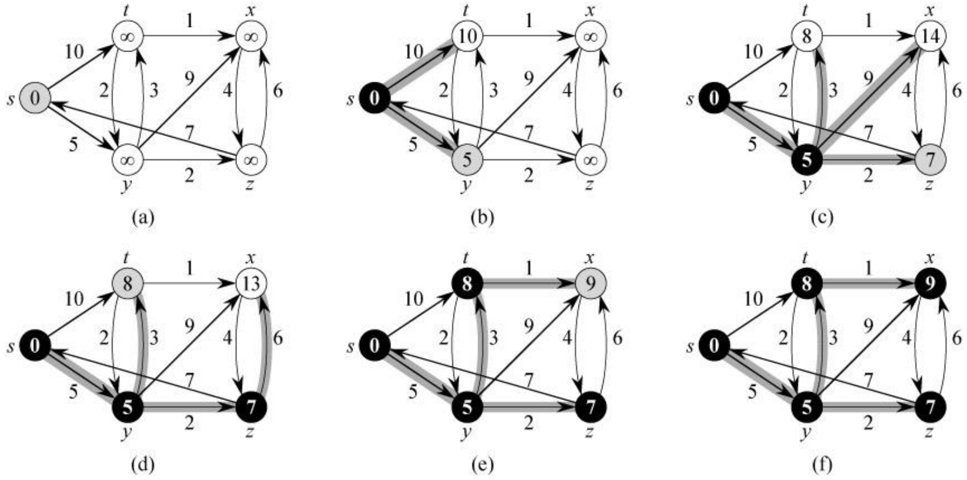


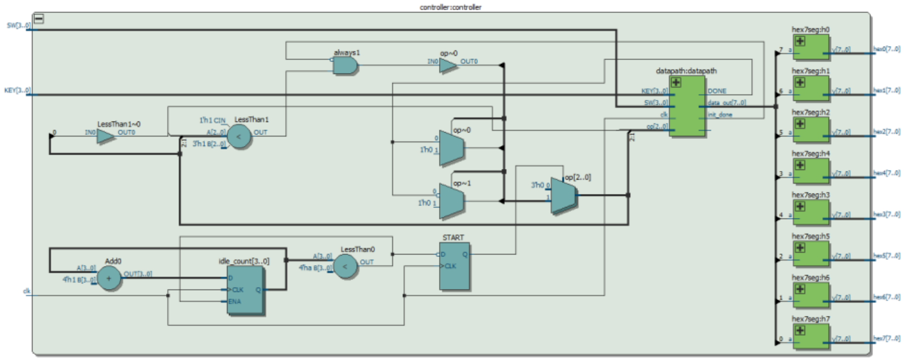
Figure 1. The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority

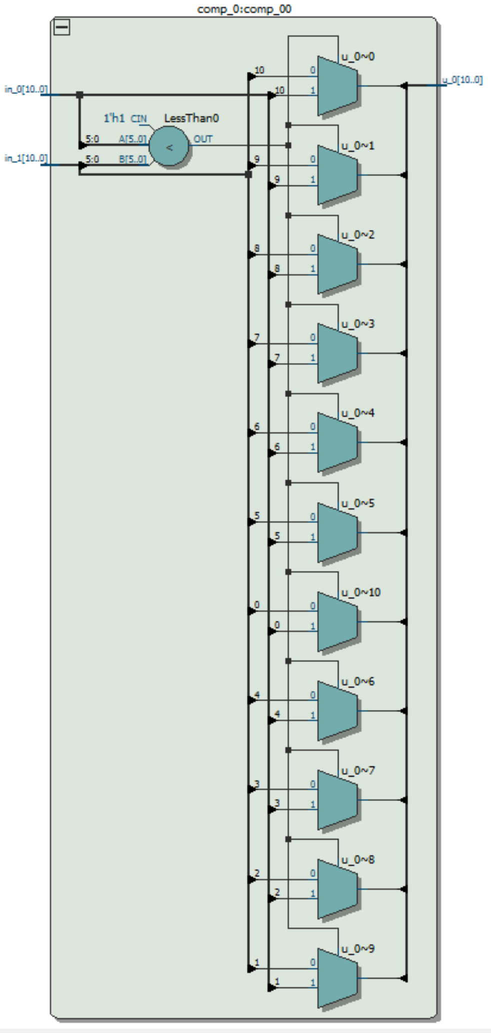
2. Software Prototype

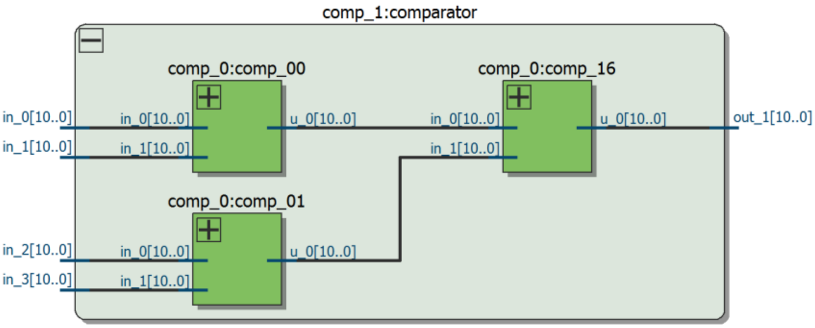
Learning from the previous works, we implemented the algorithm in software to help us understand the algorithm and verify what may be translated to the hardware. Development of the software prototype was done in two stages; first we implemented a classic sequential code. We ran this code on graph networks that were created from random generate mazes. After gaining credibility we used the structures similar to what we use in the hardware and ran it on the same graphs.

3. Architecture

Dijkstra's algorithm utilizes four data items in its operation – the distance of each node to the source node, each node's visited status, the optimal previous node to each node, and the maze graph information. For simplicity, the hardware architecture was initially designed to solve a maze with no more than 32 nodes, and the data path resources as well as the memory architecture were optimized for a node network of this size. The memory organization for the above mentioned data items is illustrated in Figure X. To store distance, previous, and visited information, two M10K dual port memory blocks with 5 bit address buses were designated for each. The word length of the distance memory was selected to be 8 bits, which was found to be sufficient to cover the range of distances for this sized network, and the previous information for each node was encoded with 5 bits. Only 1 bit was required to store visited information for each node (1 for visited, 0 for not visited). Dual port RAMs were utilized so that information could be retrieved and updated for up to four different nodes in parallel. Given that a node can have a maximum of four neighboring nodes (above, below, to the left, or to the right), it is not needed to access information about more than four nodes at a time. In contrast, four dual port memory modules are allocated for the graph information. Each row of the memory blocks corresponds to a different node in the network (the first row corresponded to the source node, the second row to node 1, etc.) and each module corresponded to a particular neighbor to the current node. The four modules made it possible to reference each neighbor in an organized and intuitive fashion as well as access graph information about all neighbors in parallel. The word width of each graph memory module is 11 bits, with the five most significant bits referencing the index of the neighboring node and the six least significant bits encoding the path lengths between this neighbor and the current node.







4. Algorithm Implementation

The algorithm proceeds as follows. Node 0, or the source node, is initially selected as the current node. Using the graph information stored in RAM, the distance of each of the neighboring nodes to the current node (let the current node be denoted by the letter u) is assessed. If this distance is greater than the distance of u summed with the path length between node u and its neighbor, the neighbor's distance is updated to equal the distance of u plus the path length. The neighbor's previous node information is also updated by the node ID u. This information is updated in both distance modules over different clock cycles as each distance memory module has only two (not four) independently addressable ports. The distance information for each node was extracted by passing the leading 5 bits of the graph module to the address of one of the distance modules. Once all of the updates have been carried out, the distances of all nodes are passed to a comparator block, which selects the node with the minimum distance, four nodes at a time. The minimum distance (concatenated with its corresponding node index) calculated in each comparison cycle is stored in a register and is then compared against the outcome of the comparison of the next batch of four nodes passed to the comparator block and updated accordingly. The final node stored in the register is then selected as the new u (current node) and the visited bit for this node is set to 1 so that it is not considered in the following updates and comparisons. This process is continued until the target node is visited, at which point the hardware is ready to display previous information over VGA for the node ID that is typed into a C++ program and software driver that communicates with the hardware.

A main advantage of implementing this algorithm in hardware is the ability to employ parallel computation. There were three major ways in which certain aspects of the algorithm were parallelized. First, with the selected memory organization, the node distance comparisons (for the purpose of finding the node with minimum distance) were carried out four nodes a time. That is, the distances of four nodes were compared to each other simultaneously, and a different set of four nodes are compared each cycle until all nodes are sent through the comparator. However, rather than passing visited nodes into the comparator block (we do not want a visited node to be selected more than once to be current node), the highest possible distance (binary 11111111) is instead passed to make it impossible for that node to be selected. Other operations carried out in parallel (four nodes at a time) are the evaluations and updates of the neighbor node distances and previous information. Lastly, sequential operations in the algorithm were often interleaved, with separate multi-cycle operations overlapping in time.

5. Software Functions

The portion of this project that is implemented in hardware is Dijkstra's shortest path algorithm. The rest of the project is implemented in software using a combination of C++ and python programs. In software, a random maze of selectable dimensions is generated and a graph network is extracted from the generated maze containing the path lengths between all adjacent nodes. Once a graph network has been constructed, it is sent via software driver to the hardware in order to be stored into the graph memory modules two nodes at a time using a 32 bit data block. Additionally, once the hardware has finished computing the shortest path through the maze, the software takes in through manual input the previous data generated by the algorithm and displays the randomly generated maze with a the optimal path highlighted in yellow.

6. Project Plan

6.1. Milestones

1) Milestone 1
    a) Construct the maze

    b) Generate a graph
    c) Implement Dijkstra’s algorithm in C
    d) Verify the software implementation of the algorithm

2) Milestone 2

    a) Implement the algorithm in hardware
    b) Send graph data from the software to the hardware
    c) Write a test-bench to verify the hardware implementation

3) Milestone 3

    a) Display the maze as well the optimal path
    b) Perform debugging on the design
    c) Compare performance of the hardware implementation against the software.

6.2. Lessons Learned

During the course of the design, the team learned several keys lessons. We gained a strong appreciation for the importance of frequent and thorough testing throughout the design. Quickly building SystemVerilog code without performing intermediate testing resulted in very arduous and time consuming debugging. Due this lack of testing, the team was at several times forced to rewrite substantial portions of code as bugs were very difficult to track down. Additionally, we gained a strong appreciation for implementing operations and algorithms with low cost, efficient hardware. This was important for reducing computation times, reducing on chip area and power usage, and allowing for scalability.

6.3. Individual Roles

While we were each responsible for our deliverables as per the milestone targets the approach to the design of the algorithm on FPGA was highly collaborative. Because of the vastness of the verilog code for the project each of us contributed to parts of HDL when needed. However, Ariel and Veton made notable contributions to the HDL code in the later stages of the project. Michelle and Utkarsh worked on the software and driver. We pointed our mistakes and corrected our errors at all stages and at all parts of the project.

SYSTEM VERILOG

Datapath

```
module datapath(
	input logic clk,
	input logic [2:0] op,
	input logic [3:0]  KEY,
	input logic [3:0]  SW,
	input logic [31:0]  data_in,
	input logic  write,
	input		chipselect,
	output logic [31:0] data_out,
	output logic init_done,
	output logic update_done,
	output logic DONE);

//Distance Memory Parameters
logic [7:0] dist_data_1a, dist_data_1b, dist_data_2a, dist_data_2b;
logic [4:0] dist_addr_1a, dist_addr_1b, dist_addr_2a, dist_addr_2b;
logic [7:0] dist_read_1a, dist_read_1b, dist_read_2a, dist_read_2b;
logic dist_we_1a, dist_we_1b, dist_we_2a, dist_we_2b;


logic [4:0] prev_data_1a, prev_data_1b, prev_data_2a, prev_data_2b;
logic [4:0] prev_addr_1a, prev_addr_1b, prev_addr_2a, prev_addr_2b;
logic [4:0] prev_read_1a, prev_read_1b, prev_read_2a, prev_read_2b;
logic prev_we_1a, prev_we_1b, prev_we_2a, prev_we_2b;

logic visited_data_1a, visited_data_1b, visited_data_2a, visited_data_2b;
logic [4:0] visited_addr_1a, visited_addr_1b, visited_addr_2a, visited_addr_2b;
logic visited_read_1a, visited_read_1b, visited_read_2a, visited_read_2b;
logic visited_we_1a, visited_we_1b, visited_we_2a, visited_we_2b;

logic [10:0] graph_data_1a, graph_data_1b, graph_data_2a, graph_data_2b, graph_data_3a,
graph_data_3b, graph_data_4a, graph_data_4b;
logic [4:0] graph_addr_1a, graph_addr_1b, graph_addr_2a, graph_addr_2b, graph_addr_3a,
graph_addr_3b, graph_addr_4a, graph_addr_4b;
logic [10:0] graph_read_1a, graph_read_1b, graph_read_2a, graph_read_2b, graph_read_3a,
graph_read_3b, graph_read_4a, graph_read_4b;
logic graph_we_1a, graph_we_1b, graph_we_2a, graph_we_2b, graph_we_3a, graph_we_3b,
graph_we_4a, graph_we_4b;

distance_memory
dist_1(.data_a(dist_data_1a),.data_b(dist_data_1b),.addr_a(dist_addr_1a),.addr_b(dist_addr_1b),.we_a(
dist_we_1a),.we_b(dist_we_1b),.q_a(dist_read_1a),.q_b(dist_read_1b),.*);
distance_memory
dist_2(.data_a(dist_data_2a),.data_b(dist_data_2b),.addr_a(dist_addr_2a),.addr_b(dist_addr_2b),.we_a(
```

```
dist_we_2a),.we_b(dist_we_2b),.q_a(dist_read_2a),.q_b(dist_read_2b),.*);

previous_memory
prev_1(.data_a(prev_data_1a),.data_b(prev_data_1b),.addr_a(prev_addr_1a),.addr_b(prev_addr_1b),.we_a(prev_we_1a),.we_b(prev_we_1b),.q_a(prev_read_1a),.q_b(prev_read_1b),.*);

previous_memory
prev_2(.data_a(prev_data_2a),.data_b(prev_data_2b),.addr_a(prev_addr_2a),.addr_b(prev_addr_2b),.we_a(prev_we_2a),.we_b(prev_we_2b),.q_a(prev_read_2a),.q_b(prev_read_2b),.*);


visited_memory
visited_1(.data_a(visited_data_1a),.data_b(visited_data_1b),.addr_a(visited_addr_1a),.addr_b(visited_addr_1b),.we_a(visited_we_1a),.we_b(visited_we_1b),.q_a(visited_read_1a),.q_b(visited_read_1b),.*);

visited_memory
visited_2(.data_a(visited_data_2a),.data_b(visited_data_2b),.addr_a(visited_addr_2a),.addr_b(visited_addr_2b),.we_a(visited_we_2a),.we_b(visited_we_2b),.q_a(visited_read_2a),.q_b(visited_read_2b),.*);

graph_memory
graph_1(.data_a(graph_data_1a),.data_b(graph_data_1b),.addr_a(graph_addr_1a),.addr_b(graph_addr_1b),.we_a(graph_we_1a),.we_b(graph_we_1b),.q_a(graph_read_1a),.q_b(graph_read_1b),.*);
graph_memory
graph_2(.data_a(graph_data_2a),.data_b(graph_data_2b),.addr_a(graph_addr_2a),.addr_b(graph_addr_2b),.we_a(graph_we_2a),.we_b(graph_we_2b),.q_a(graph_read_2a),.q_b(graph_read_2b),.*);
graph_memory
graph_3(.data_a(graph_data_3a),.data_b(graph_data_3b),.addr_a(graph_addr_3a),.addr_b(graph_addr_3b),.we_a(graph_we_3a),.we_b(graph_we_3b),.q_a(graph_read_3a),.q_b(graph_read_3b),.*);
graph_memory
graph_4(.data_a(graph_data_4a),.data_b(graph_data_4b),.addr_a(graph_addr_4a),.addr_b(graph_addr_4b),.we_a(graph_we_4a),.we_b(graph_we_4b),.q_a(graph_read_4a),.q_b(graph_read_4b),.*);

comp_1 comparator
(.in_0(comp_in1),.in_1(comp_in2),.in_2(comp_in3),.in_3(comp_in4),.out_1(comp_out));

/*---------------------Memory Initialization---------------------------*/

logic [5:0] counter;
logic phase;
logic [4:0] u;
logic [4:0] prev_u;
logic [7:0] dist_u;
logic init_done_0, init_done_1;

logic [10:0] comp_in1, comp_in2, comp_in3, comp_in4;
logic [10:0] comp_out;

logic [5:0] num_nodes;
```

```
logic corner_flag;

logic update_1,update_2;

assign init_done = init_done_0 & init_done_1;

logic [3:0] test;

logic [9:0] graph_counter;

logic [3:0] op_count;
logic [4:0] update_count;
logic [4:0] iter_count;
logic [3:0] update_vector;
logic [10:0] node_min;
logic [8:0] comp_count;
logic [8:0] start_count;
logic [8:0] comp_reg1, comp_reg2, comp_reg3, comp_reg4;
logic update_min;

assign test[0] = (graph_read_1a[0] | graph_read_1a[1] | graph_read_1a[2]
                                        | graph_read_1a[3]  | graph_read_1a[4] | graph_read_1a[5]);

assign test[1] = (graph_read_2a[0] | graph_read_2a[1] | graph_read_2a[2]
                                        | graph_read_2a[3]  | graph_read_2a[4] | graph_read_2a[5]);

assign test[2] = (graph_read_3a[0] | graph_read_3a[1] | graph_read_3a[2]
                                        | graph_read_3a[3]  | graph_read_3a[4] | graph_read_3a[5]);

assign test[3] = (graph_read_4a[0] | graph_read_4a[1] | graph_read_4a[2]
                                        | graph_read_4a[3]  | graph_read_4a[4] | graph_read_4a[5]);


        logic [5:0] counter_graph;

always_ff @(posedge clk)
begin
        if(op == 3'd1) //Initialization
        begin
                DONE <= 1'b0;
                if (counter < 6'd32)
                begin
                        if(phase == 1'b0)
                        begin
                                u <= 5'd0;
                                dist_we_1a <= 1'b0;
                                dist_we_2a <= 1'b0;
                                visited_we_1a <= 1'b0;
                                visited_we_2a <= 1'b0;
```

```
                prev_we_1a <= 1'b0;
                prev_we_2a <= 1'b0;

                dist_addr_1a <= counter[4:0];
                dist_addr_2a <= counter[4:0];
                visited_addr_1a <= counter[4:0];
                visited_addr_2a <= counter[4:0];
                prev_addr_1a <= counter[4:0];
                prev_addr_2a <= counter[4:0];

                visited_data_1a <= 1'b0;
                visited_data_2a <= 1'b0;
                prev_data_1a <= 1'b0;
                prev_data_2a <= 1'b0;

                if(counter == 6'd0)
                        begin
                                dist_data_1a <= 8'b00000000;
                                dist_data_2a <= 8'b00000000;

                        end
                else
                        begin
                                dist_data_1a <= 8'b11111111;
                                dist_data_2a <= 8'b11111111;
                        end
                phase <= 1'b1;
        end
        else if (phase == 1'b1)
        begin
                dist_we_1a <= 1'b1;
                dist_we_2a <= 1'b1;
                visited_we_1a <= 1'b1;
                visited_we_2a <= 1'b1;
                prev_we_1a <= 1'b1;
                prev_we_2a <= 1'b1;
                phase <= 1'b0;
                counter <= counter + 6'd1;
        end
end
else
begin
        dist_we_1a <= 1'b0;
        dist_we_2a <= 1'b0;
        visited_we_1a <= 1'b0;
        visited_we_2a <= 1'b0;
        prev_we_1a <= 1'b0;
        prev_we_2a <= 1'b0;
        init_done_0 <= 1'd1;
```

```
        update_done <= 1'b0;
end

//reading num nodes
        if (chipselect && write && data_in[31])
                num_nodes <= data_in[5:0];
        else
                num_nodes <= num_nodes;

//reading graph data

if (graph_counter < 10'd32)
begin
        if (chipselect && write && !data_in[31] && !data_in[30])
                begin
                        graph_data_1a <= data_in[10:0];
                        graph_addr_1a <= graph_counter[4:0];

                        graph_data_2a <= data_in[25:15];
                        graph_addr_2a <= graph_counter[4:0];

                        graph_we_1a <= 1'b0;
                        graph_we_2a <= 1'b0;
                        graph_we_3a <= 1'b1;
                        graph_we_4a <= 1'b1;
                end
        else if (chipselect && write && !data_in[31] && data_in[30])
                begin
                        graph_data_3a <= data_in[10:0];
                        graph_addr_3a <= graph_counter[4:0];

                        graph_data_4a <= data_in[25:15];
                        graph_addr_4a <= graph_counter[4:0];

                        graph_we_3a <= 1'b0;
                        graph_we_4a <= 1'b0;
                        graph_we_1a <= 1'b1;
                        graph_we_2a <= 1'b1;

                        graph_counter <= graph_counter + 10'd1;
                end
        else
                begin
                        graph_data_1a <= graph_data_1a;
                        graph_addr_1a <= graph_addr_1a;

                        graph_data_2a <= graph_data_2a;
                        graph_addr_2a <= graph_addr_2a;
```

```
                    graph_data_3a <= graph_data_3a;
                    graph_addr_3a <= graph_addr_3a;

                    graph_data_4a <= graph_data_4a;
                    graph_addr_4a <= graph_addr_4a;

                    graph_we_1a <= graph_we_1a;
                    graph_we_2a <= graph_we_2a;
                    graph_we_3a <= graph_we_3a;
                    graph_we_4a <= graph_we_4a;
                end
        end
        else if (graph_counter == 10'd32)
        begin
            graph_we_1a <= 1'b0;
            graph_we_2a <= 1'b0;
            graph_we_3a <= 1'b1;
            graph_we_4a <= 1'b1;

            graph_data_3a <= graph_data_3a;
            graph_addr_3a <= graph_addr_3a;

            graph_data_4a <= graph_data_4a;
            graph_addr_4a <= graph_addr_4a;

            graph_counter <= graph_counter + 10'd1;
        end
        else
        begin
            init_done_1 <= 1'b1;

            graph_we_1a <= 1'b0;
            graph_we_2a <= 1'b0;
            graph_we_3a <= 1'b0;
            graph_we_4a <= 1'b0;
        end


    end//-------------------------------------------------------------------------------------




    else if(op == 3'd2) //--------------------UPDATE DISTANCE AND PREVIOUS
    begin//0
        if(u != 5'd15)
        begin//check done
        if(update_count == 5'd0)
        begin//0a           ----            Set u to visited
```

```
        node_min <= 11'd2047;
        u <= u;
        start_count <= 9'b0;
        comp_count <= 9'd0;

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;

        visited_addr_1a <= u;
        visited_addr_2a <= u;

        //test
        dist_addr_1a <= graph_read_1a[10:6];
        dist_addr_1b <= graph_read_2a[10:6];
        dist_addr_2a <= graph_read_3a[10:6];
        dist_addr_2b <= graph_read_4a[10:6];

        visited_data_1a <= 1'b1;
        visited_data_2a <= 1'b1;

        //fetch dist of u
        dist_addr_1a <= u;

        //enforcing
        dist_we_1a <= 1'b0;
        dist_we_1b <= 1'b0;
        dist_we_2a <= 1'b0;
        dist_we_2b <= 1'b0;
        visited_we_1b <= 1'b0;
        visited_we_2b <= 1'b0;
        prev_we_1a <= 1'b0;
        prev_we_2a <= 1'b0;
        prev_we_1b <= 1'b0;
        prev_we_2b <= 1'b0;



        update_count <= update_count + 5'd1;

end//0a
else if(update_count == 5'd1)
begin//0b       ----            Set visited_we high to set u to visited

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;
```

```
		visited_addr_1a <= u;
		visited_addr_2a <= u;

		visited_data_1a <= 1'b1;
		visited_data_2a <= 1'b1;

		visited_we_1a <= 1'b1;
		visited_we_2a <= 1'b1;

		//fetch dist of u
		dist_addr_1a <= u;
		dist_u <= dist_read_1a;

		//enforcing
		dist_we_1a <= 1'b0;
		dist_we_1b <= 1'b0;
		dist_we_2a <= 1'b0;
		dist_we_2b <= 1'b0;
		visited_we_1b <= 1'b0;
		visited_we_2b <= 1'b0;
		prev_we_1a <= 1'b0;
		prev_we_2a <= 1'b0;
		prev_we_1b <= 1'b0;
		prev_we_2b <= 1'b0;




		update_count <= update_count + 5'd1;

	end//0b


	else if(update_count == 5'd2)
	begin//0c
		visited_addr_1a <= u;
		visited_addr_2a <= u;

		visited_we_1a <= 1'b0;
		visited_we_2a <= 1'b0;

		graph_addr_1a <= u;
		graph_addr_2a <= u;
		graph_addr_3a <= u;
		graph_addr_4a <= u;


		//fetch dist of u
```

```
        dist_addr_1a <= u;
        dist_u <= dist_read_1a;

        //enforcing
        dist_we_1a <= 1'b0;
        dist_we_1b <= 1'b0;
        dist_we_2a <= 1'b0;
        dist_we_2b <= 1'b0;
        visited_we_1b <= 1'b0;
        visited_we_2b <= 1'b0;
        prev_we_1a <= 1'b0;
        prev_we_2a <= 1'b0;
        prev_we_1b <= 1'b0;
        prev_we_2b <= 1'b0;


        update_count <= update_count + 5'd1;

end//0c
else if(update_count == 5'd3)
begin//0d
        //fetch visited info about each neighbor
        visited_addr_1a <= graph_read_1a[10:6];
        visited_addr_1b <= graph_read_2a[10:6];
        visited_addr_2a <= graph_read_3a[10:6];
        visited_addr_2b <= graph_read_4a[10:6];

        //fetch distance of each neighbor
        dist_addr_1a <= graph_read_1a[10:6];
        dist_addr_1b <= graph_read_2a[10:6];
        dist_addr_2a <= graph_read_3a[10:6];
        dist_addr_2b <= graph_read_4a[10:6];

        //set prev address ready for writing
        prev_addr_1a <= graph_read_1a[10:6];
        prev_addr_1b <= graph_read_2a[10:6];
        prev_addr_2a <= graph_read_3a[10:6];
        prev_addr_2b <= graph_read_4a[10:6];

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;

        update_count <= update_count + 5'd1;
end//0d
else if(update_count > 5'd3 && update_count < 5'd7)
begin//0e
```

```
                graph_addr_1a <= u;
                graph_addr_2a <= u;
                graph_addr_3a <= u;
                graph_addr_4a <= u;

                //fetch visited info about each neighbor
                visited_addr_1a <= graph_read_1a[10:6];
                visited_addr_1b <= graph_read_2a[10:6];
                visited_addr_2a <= graph_read_3a[10:6];
                visited_addr_2b <= graph_read_4a[10:6];

                //fetch distance of each neighbor
                dist_addr_1a <= graph_read_1a[10:6];
                dist_addr_1b <= graph_read_2a[10:6];
                dist_addr_2a <= graph_read_3a[10:6];
                dist_addr_2b <= graph_read_4a[10:6];

                //set prev address ready for writing
                prev_addr_1a <= graph_read_1a[10:6];
                prev_addr_1b <= graph_read_2a[10:6];
                prev_addr_2a <= graph_read_3a[10:6];
                prev_addr_2b <= graph_read_4a[10:6];

                //update prev and dist of neighbors if they are not visited or if their distance is greather than dist(u)+path weight
                if((visited_read_1a == 1'b0) && (dist_read_1a > (dist_u + graph_read_1a[5:0])))
                begin
                    update_vector[0] <= 1'b1;
                end
                else
                begin
                    update_vector[0] <= 1'b0;
                end

                if((visited_read_1b == 1'b0) && (dist_read_1b > (dist_u + graph_read_2a[5:0])))
                begin
                    update_vector[1] <= 1'b1;
                end
                else
                begin
                    update_vector[1] <= 1'b0;
                end

                if((visited_read_2a == 1'b0) && (dist_read_2a > (dist_u + graph_read_3a[5:0])))
                begin
                    update_vector[2] <= 1'b1;
                end
                else
                begin
```

```
					update_vector[2] <= 1'b0;
				end

				if((visited_read_2b == 1'b0) && (dist_read_2b > (dist_u + graph_read_4a[5:0])))
				begin
					update_vector[3] <= 1'b1;
				end
				else
				begin
					update_vector[3] <= 1'b0;
				end

//				if(iter_count == 5'd2)
//					update_count <= update_count;
//				else
//					update_count <= update_count + 5'd1;

				update_count <= update_count + 5'd1;
			end//0e
			else if(update_count == 5'd7)
			begin//0f

				graph_addr_1a <= u;
				graph_addr_2a <= u;
				graph_addr_3a <= u;
				graph_addr_4a <= u;

				//fetch distance of each neighbor
				dist_addr_1a <= graph_read_1a[10:6];
				dist_addr_1b <= graph_read_2a[10:6];
				dist_addr_2a <= graph_read_3a[10:6];
				dist_addr_2b <= graph_read_4a[10:6];

				//set prev address ready for writing
				prev_addr_1a <= graph_read_1a[10:6];
				prev_addr_1b <= graph_read_2a[10:6];
				prev_addr_2a <= graph_read_3a[10:6];
				prev_addr_2b <= graph_read_4a[10:6];

				//set write enables low
				dist_we_1a <= 1'b0;
				dist_we_1b <= 1'b0;
				dist_we_2a <= 1'b0;
				dist_we_2b <= 1'b0;

				prev_we_1a <= 1'b0;
				prev_we_1b <= 1'b0;
				prev_we_2a <= 1'b0;
				prev_we_2b <= 1'b0;
```

```
update_vector <= update_vector;

if(update_vector[0] == 1'b1)
begin
        dist_data_1a <= dist_u + graph_read_1a[5:0];
        prev_data_1a <= u;
end
else
begin
        dist_data_1a <= dist_read_1a;
        prev_data_1a <= prev_read_1a;
end

if(update_vector[1] == 1'b1)
begin
        dist_data_1b <= dist_u + graph_read_2a[5:0];
        prev_data_1b <= u;
end
else
begin
        dist_data_1b <= dist_read_1b;
        prev_data_1b <= prev_read_1b;
end

if(update_vector[2] == 1'b1)
begin
        dist_data_2a <= dist_u + graph_read_3a[5:0];
        prev_data_2a <= u;
end
else
begin
        dist_data_2a <= dist_read_2a;
        prev_data_2a <= prev_read_2a;
end

if(update_vector[3] == 1'b1)
begin
        dist_data_2b <= dist_u + graph_read_4a[5:0];
        prev_data_2b <= u;
end
else
begin
        dist_data_2b <= dist_read_2b;
        prev_data_2b <= prev_read_2b;
end

update_count <= update_count + 5'd1;
//update_count <= update_count;
```

```
end//0f
else if(update_count == 5'd8)
begin//0g

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;

        //fetch distance of each neighbor
        dist_addr_1a <= graph_read_1a[10:6];
        dist_addr_1b <= graph_read_2a[10:6];
        dist_addr_2a <= graph_read_3a[10:6];
        dist_addr_2b <= graph_read_4a[10:6];

        //set prev address ready for writing
        prev_addr_1a <= graph_read_1a[10:6];
        prev_addr_1b <= graph_read_2a[10:6];
        prev_addr_2a <= graph_read_3a[10:6];
        prev_addr_2b <= graph_read_4a[10:6];

        //set write enables high
        //dist_we_1a <= 1'b1;
        //dist_we_1b <= 1'b1;
        //dist_we_2a <= 1'b1;
        //dist_we_2b <= 1'b1;

        //prev_we_1a <= 1'b1;
        //prev_we_1b <= 1'b1;
        //prev_we_2a <= 1'b1;
        //prev_we_2b <= 1'b1;

        if(update_vector[0] == 1'b1)
        begin
                dist_data_1a <= dist_u + graph_read_1a[5:0];
                prev_data_1a <= u;
                prev_we_1a <= 1'b1;
                dist_we_1a <= 1'b1;
        end
        else
        begin
                dist_data_1a <= dist_read_1a;
                prev_data_1a <= prev_read_1a;
                prev_we_1a <= 1'b0;
                dist_we_1a <= 1'b0;
        end

        if(update_vector[1] == 1'b1)
```

```
		begin
			dist_data_1b <= dist_u + graph_read_2a[5:0];
			prev_data_1b <= u;
			prev_we_1b <= 1'b1;
			dist_we_1b <= 1'b1;
		end
		else
		begin
			dist_data_1b <= dist_read_1b;
			prev_data_1b <= prev_read_1b;
			prev_we_1b <= 1'b0;
			dist_we_1b <= 1'b0;
		end

		if(update_vector[2] == 1'b1)
		begin
			dist_data_2a <= dist_u + graph_read_3a[5:0];
			prev_data_2a <= u;
			prev_we_2a <= 1'b1;
			dist_we_2a <= 1'b1;
		end
		else
		begin
			dist_data_2a <= dist_read_2a;
			prev_data_2a <= prev_read_2a;
			prev_we_2a <= 1'b0;
			dist_we_2a <= 1'b0;
		end

		if(update_vector[3] == 1'b1)
		begin
			dist_data_2b <= dist_u + graph_read_4a[5:0];
			prev_data_2b <= u;
			prev_we_2b <= 1'b1;
			dist_we_2b <= 1'b1;
		end
		else
		begin
			dist_data_2b <= dist_read_2b;
			prev_data_2b <= prev_read_2b;
			prev_we_2b <= 1'b0;
			dist_we_2b <= 1'b0;
		end

		update_count <= update_count + 5'd1;

	end//0g
	else if(update_count == 5'd9)
	begin//0h
```

```
        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;

        //fetch distance of each neighbor
        dist_addr_1a <= graph_read_1a[10:6];
        dist_addr_1b <= graph_read_2a[10:6];
        dist_addr_2a <= graph_read_3a[10:6];
        dist_addr_2b <= graph_read_4a[10:6];

        //set prev address ready for writing
        prev_addr_1a <= graph_read_1a[10:6];
        prev_addr_1b <= graph_read_2a[10:6];
        prev_addr_2a <= graph_read_3a[10:6];
        prev_addr_2b <= graph_read_4a[10:6];

        //set write enables low
        dist_we_1a <= 1'b0;
        dist_we_1b <= 1'b0;
        dist_we_2a <= 1'b0;
        dist_we_2b <= 1'b0;

        prev_we_1a <= 1'b0;
        prev_we_1b <= 1'b0;
        prev_we_2a <= 1'b0;
        prev_we_2b <= 1'b0;

        //***MIGHT WANT TO ENFORCE INPUTS TO PREV AND DATA

        update_count <= update_count + 5'd1;
end//0h
else if(update_count == 5'd10)
begin//0i                     update next 2 elements in array

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
        graph_addr_4a <= u;

        //fetch distance of each neighbor
        dist_addr_1a <= graph_read_3a[10:6];
        dist_addr_1b <= graph_read_4a[10:6];
        dist_addr_2a <= graph_read_1a[10:6];
        dist_addr_2b <= graph_read_2a[10:6];

        //set prev address ready for writing
        prev_addr_1a <= graph_read_3a[10:6];
```

```
		prev_addr_1b <= graph_read_4a[10:6];
		prev_addr_2a <= graph_read_1a[10:6];
		prev_addr_2b <= graph_read_2a[10:6];

		//set write enables low
		dist_we_1a <= 1'b0;
		dist_we_1b <= 1'b0;
		dist_we_2a <= 1'b0;
		dist_we_2b <= 1'b0;

		prev_we_1a <= 1'b0;
		prev_we_1b <= 1'b0;
		prev_we_2a <= 1'b0;
		prev_we_2b <= 1'b0;

		update_count <= update_count + 5'd1;

	end//0i
	else if(update_count == 5'd11)
	begin//0j

		graph_addr_1a <= u;
		graph_addr_2a <= u;
		graph_addr_3a <= u;
		graph_addr_4a <= u;

		//fetch distance of each neighbor
		dist_addr_1a <= graph_read_3a[10:6];
		dist_addr_1b <= graph_read_4a[10:6];
		dist_addr_2a <= graph_read_1a[10:6];
		dist_addr_2b <= graph_read_2a[10:6];

		//set prev address ready for writing
		prev_addr_1a <= graph_read_3a[10:6];
		prev_addr_1b <= graph_read_4a[10:6];
		prev_addr_2a <= graph_read_1a[10:6];
		prev_addr_2b <= graph_read_2a[10:6];

		//set write enables low
		dist_we_1a <= 1'b0;
		dist_we_1b <= 1'b0;
		dist_we_2a <= 1'b0;
		dist_we_2b <= 1'b0;

		prev_we_1a <= 1'b0;
		prev_we_1b <= 1'b0;
		prev_we_2a <= 1'b0;
		prev_we_2b <= 1'b0;
```

```
            if(update_vector[2] == 1'b1)
            begin
                dist_data_1a <= dist_u + graph_read_3a[5:0];
                prev_data_1a <= u;
            end
            else
            begin
                dist_data_1a <= dist_read_1a;
                prev_data_1a <= prev_read_1a;
            end

            if(update_vector[3] == 1'b1)
            begin
                dist_data_1b <= dist_u + graph_read_4a[5:0];
                prev_data_1b <= u;
            end
            else
            begin
                dist_data_1b <= dist_read_1b;
                prev_data_1b <= prev_read_1b;
            end

            if(update_vector[1] == 1'b1)
            begin
                dist_data_2a <= dist_u + graph_read_1a[5:0];
                prev_data_2a <= u;
            end
            else
            begin
                dist_data_2a <= dist_read_2a;
                prev_data_2a <= prev_read_2a;
            end

            if(update_vector[2] == 1'b1)
            begin
                dist_data_2b <= dist_u + graph_read_2a[5:0];
                prev_data_2b <= u;
            end
            else
            begin
                dist_data_2b <= dist_read_2b;
                prev_data_2b <= prev_read_2b;
            end

            update_count <= update_count + 5'd1;

        end//0j
        else if(update_count == 5'd12)
        begin//0k
```

```
graph_addr_1a <= u;
graph_addr_2a <= u;
graph_addr_3a <= u;
graph_addr_4a <= u;

//fetch distance of each neighbor
dist_addr_1a <= graph_read_3a[10:6];
dist_addr_1b <= graph_read_4a[10:6];
dist_addr_2a <= graph_read_1a[10:6];
dist_addr_2b <= graph_read_2a[10:6];

//set prev address ready for writing
prev_addr_1a <= graph_read_3a[10:6];
prev_addr_1b <= graph_read_4a[10:6];
prev_addr_2a <= graph_read_1a[10:6];
prev_addr_2b <= graph_read_2a[10:6];

//set write enables high
//dist_we_1a <= 1'b1;
//dist_we_1b <= 1'b1;
//dist_we_2a <= 1'b1;
//dist_we_2b <= 1'b1;

//prev_we_1a <= 1'b1;
//prev_we_1b <= 1'b1;
//prev_we_2a <= 1'b1;
//prev_we_2b <= 1'b1;

if(update_vector[2] == 1'b1)
begin
        dist_data_1a <= dist_u + graph_read_3a[5:0];
        prev_data_1a <= u;
        prev_we_1a <= 1'b1;
        dist_we_1a <= 1'b1;
end
else
begin
        dist_data_1a <= dist_read_1a;
        prev_data_1a <= prev_read_1a;
        prev_we_1a <= 1'b0;
        dist_we_1a <= 1'b0;
end

if(update_vector[3] == 1'b1)
begin
        dist_data_1b <= dist_u + graph_read_4a[5:0];
        prev_data_1b <= u;
        prev_we_1b <= 1'b1;
```

```
                dist_we_1b <= 1'b1;
        end
        else
        begin
                dist_data_1b <= dist_read_1b;
                prev_data_1b <= prev_read_1b;
                prev_we_1b <= 1'b0;
                dist_we_1b <= 1'b0;
        end

        if(update_vector[0] == 1'b1)
        begin
                dist_data_2a <= dist_u + graph_read_1a[5:0];
                prev_data_2a <= u;
                prev_we_2a <= 1'b1;
                dist_we_2a <= 1'b1;
        end
        else
        begin
                dist_data_2a <= dist_read_2a;
                prev_data_2a <= prev_read_2a;
                prev_we_2a <= 1'b0;
                dist_we_2a <= 1'b0;
        end

        if(update_vector[1] == 1'b1)
        begin
                dist_data_2b <= dist_u + graph_read_2a[5:0];
                prev_data_2b <= u;
                prev_we_2b <= 1'b1;
                dist_we_2b <= 1'b1;
        end
        else
        begin
                dist_data_2b <= dist_read_2b;
                prev_data_2b <= prev_read_2b;
                prev_we_2b <= 1'b0;
                dist_we_2b <= 1'b0;
        end


        update_count <= update_count + 5'd1;
end//0k
else if(update_count == 5'd13)
begin//0l

        graph_addr_1a <= u;
        graph_addr_2a <= u;
        graph_addr_3a <= u;
```

```
			graph_addr_4a <= u;

			//fetch distance of each neighbor
			dist_addr_1a <= graph_read_3a[10:6];
			dist_addr_1b <= graph_read_4a[10:6];
			dist_addr_2a <= graph_read_1a[10:6];
			dist_addr_2b <= graph_read_2a[10:6];

			//set prev address ready for writing
			prev_addr_1a <= graph_read_3a[10:6];
			prev_addr_1b <= graph_read_4a[10:6];
			prev_addr_2a <= graph_read_1a[10:6];
			prev_addr_2b <= graph_read_2a[10:6];

			//set write enables low
			dist_we_1a <= 1'b0;
			dist_we_1b <= 1'b0;
			dist_we_2a <= 1'b0;
			dist_we_2b <= 1'b0;

			prev_we_1a <= 1'b0;
			prev_we_1b <= 1'b0;
			prev_we_2a <= 1'b0;
			prev_we_2b <= 1'b0;

			node_min <= 11'd2047;

			update_count <= update_count + 5'd1;

			comp_count <= 9'd0;

		end//0l
		else if(update_count == 5'd14)
		begin//0m

			comp_reg1 <= comp_count[4:0];
			comp_reg2 <= comp_count[4:0]+5'd1;
			comp_reg3 <= comp_count[4:0]+5'd2;
			comp_reg4 <= comp_count[4:0]+5'd3;

			//fetch visited info about each neighbor
			visited_addr_1a <= comp_count[4:0];
			visited_addr_1b <= comp_count[4:0]+5'd1;
			visited_addr_2a <= comp_count[4:0]+5'd2;
			visited_addr_2b <= comp_count[4:0]+5'd3;

			//fetch distance of each neighbor
			dist_addr_1a <= comp_count[4:0];
			dist_addr_1b <= comp_count[4:0]+5'd1;
```

```
			dist_addr_2a <= comp_count[4:0]+5'd2;
			dist_addr_2b <= comp_count[4:0]+5'd3;

			//fetch distance of each neighbor
			prev_addr_1a <= comp_count[4:0];
			prev_addr_1b <= comp_count[4:0]+5'd1;
			prev_addr_2a <= comp_count[4:0]+5'd2;
			prev_addr_2b <= comp_count[4:0]+5'd3;

			dist_we_1a <= 1'b0;
			dist_we_1b <= 1'b0;
			dist_we_2a <= 1'b0;
			dist_we_2b <= 1'b0;




			update_count <= update_count + 5'd1;

			//update_count <= update_count;

		end//0m
		else if(update_count > 5'd14 && update_count < 5'd18)//----------------------------
COMPARISON---------------------------------
		begin//change to 2 cycles

			//test
			prev_addr_1a <= comp_reg1;
			prev_addr_1b <= comp_reg2;
			prev_addr_2a <= comp_reg3;
			prev_addr_2b <= comp_reg4;


			//fetch visited info about each neighbor
			visited_addr_1a <= comp_reg1;
			visited_addr_1b <= comp_reg2;
			visited_addr_2a <= comp_reg3;
			visited_addr_2b <= comp_reg4;


			//fetch distance of each neighbor
			dist_addr_1a <= comp_reg1;
			dist_addr_1b <= comp_reg2;
			dist_addr_2a <= comp_reg3;
			dist_addr_2b <= comp_reg4;


			dist_we_1a <= 1'b0;
			dist_we_1b <= 1'b0;
```

```
	dist_we_2a <= 1'b0;
	dist_we_2b <= 1'b0;

if((visited_read_1a == 1'b1) || (comp_reg1 == u) || (dist_read_1a == 8'd0))
begin
	comp_in1 <= 11'b11111111111;
end
else
begin
	comp_in1[10:6] <= comp_reg1;
	comp_in1[5:0] <= dist_read_1a[5:0];
end

if((visited_read_1b == 1'b1) || (comp_reg2 == u) || (dist_read_1b == 8'd0))
begin
	comp_in2 <= 11'b11111111111;
end
else
begin
	comp_in2[10:6] <= comp_reg2;
	comp_in2[5:0] <= dist_read_1b[5:0];
end

if((visited_read_2a == 1'b1) || (comp_reg3 == u) || (dist_read_2a == 8'd0))
begin
	comp_in3 <= 11'b11111111111;
end
else
begin
	comp_in3[10:6] <= comp_reg3;
	comp_in3[5:0] <= dist_read_2a[5:0];
end

if((visited_read_2b == 1'b1) || (comp_reg4 == u) || (dist_read_2b == 8'd0))
begin
	comp_in4 <= 11'b11111111111;
end
else
begin
	comp_in4[10:6] <= comp_reg4;
	comp_in4[5:0] <= dist_read_2b[5:0];
end

if(comp_out[5:0] < node_min[5:0])
	update_min <= 1'b1;
else
	update_min <= 1'b0;

update_count <= update_count + 5'd1;
```

```
        end//




        else if(update_count == 5'd18)
        begin//0n

            if(update_min == 1'b1)
            begin
                node_min <= comp_out;
            end
            else
            begin
                node_min <= node_min;
            end

            if(comp_count < 9'd12)
            begin
                comp_count <= comp_count + 9'd4;
                update_count <= 5'd14;
                //update_count <= update_count;
            end
            else
            begin
                comp_count <= 9'd0;
                update_count <= update_count + 5'd1;
            end



//          dist_we_1a <= 1'b0;
//          dist_we_1b <= 1'b0;
//          dist_we_2a <= 1'b0;
//          dist_we_2b <= 1'b0;

        end//0n

        else if(update_count == 5'd19)
        begin//0o


            graph_addr_1a <= u;
            graph_addr_2a <= u;
            graph_addr_3a <= u;
            graph_addr_4a <= u;

            //u <= comp_out[10:6];
```

```
                u <= node_min[10:6];
                comp_count <= 9'd0;

                //fetch visited info about each neighbor
                visited_addr_1a <= graph_read_1a[10:6];
                visited_addr_1b <= graph_read_2a[10:6];
                visited_addr_2a <= graph_read_3a[10:6];
                visited_addr_2b <= graph_read_4a[10:6];


                //fetch distance of each neighbor
                dist_addr_1a <= graph_read_1a[10:6];
                dist_addr_1b <= graph_read_2a[10:6];
                dist_addr_2a <= graph_read_3a[10:6];
                dist_addr_2b <= graph_read_4a[10:6];



                iter_count <= iter_count + 5'd1;
                update_count <= 5'd0;


                //update_count <= update_count;

        end//0o
        else
                DONE <= 1'b0;
end//check done
else//Algorithm done
begin//cd0
        DONE <= 1'b1;
end//cd0
end//0
else if(op == 3'd0)//before starting the algorithm
begin
        DONE <= 1'b0;

        dist_we_1a <= 1'b0;
        dist_we_1b <= 1'b0;
        dist_we_2a <= 1'b0;
        dist_we_2b <= 1'b0;

        visited_we_1a <= 1'b0;
        visited_we_1b <= 1'b0;
        visited_we_2a <= 1'b0;
        visited_we_2b <= 1'b0;

        prev_we_1a <= 1'b0;
        prev_we_1b <= 1'b0;
```

```
        prev_we_2a <= 1'b0;
        prev_we_2b <= 1'b0;
    end
    else//when algorithm is done
        begin
            DONE <= 1'b1;

//          dist_addr_1a <= 5'd0;
//          visited_addr_1a <= 5'd0;
//          prev_addr_1a <= 5'd0;
//
//          dist_we_1a <= 1'b0;
//          dist_we_1b <= 1'b0;
//          dist_we_2a <= 1'b0;
//          dist_we_2b <= 1'b0;
//
//          visited_we_1a <= 1'b0;
//          visited_we_1b <= 1'b0;
//          visited_we_2a <= 1'b0;
//          visited_we_2b <= 1'b0;
//
//          prev_we_1a <= 1'b0;
//          prev_we_1b <= 1'b0;
//          prev_we_2a <= 1'b0;
//          prev_we_2b <= 1'b0;

            graph_addr_1a <= SW;
            graph_addr_2a <= SW;
            graph_addr_3a <= SW;
            graph_addr_3a <= SW;
        end
end

always_comb
begin
    if (KEY[3] == 1'b0)
        data_out[7:0] = graph_read_1a[7:0];
    else if (KEY[2] == 1'b0)
        data_out[7:0] = graph_read_2a[7:0];
    else if (KEY[1] == 1'b0)
        data_out[7:0] = graph_read_3a[7:0];
    else if (KEY[0] == 1'b0)
        data_out[7:0] = graph_read_4a[7:0];
    else
    begin
        data_out[7:3] = 5'b00100;
        data_out[2:0] = op;
    end
end
```

```
endmodule
```

Controller Module

```
module VGA_LED(input logic        clk,
               input logic            reset,
               input logic [31:0]  writedata,
               input logic            write,
               input                  chipselect,
               input logic [2:0]  address,
                           input logic [3:0]  KEY,
                           input logic [3:0]  SW,
               output logic [7:0] VGA_R, VGA_G, VGA_B,
               output logic           VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
               output logic           VGA_SYNC_n);


  logic [7:0] hex0, hex1, hex2, hex3,
                              hex4, hex5, hex6, hex7;

        logic [31:0] data_out;

        logic [2:0] op;

        always_comb
        begin
                if (data_out[7] == 1'b0)
                        hex0 = 8'b00111111;
                else
                        hex0 = 8'b00000110;

                if (data_out[6] == 1'b0)
                        hex1 = 8'b00111111;
                else
                        hex1 = 8'b00000110;

                if (data_out[5] == 1'b0)
                        hex2 = 8'b00111111;
                else
                        hex2 = 8'b00000110;

                if (data_out[4] == 1'b0)
                        hex3 = 8'b00111111;
                else
                        hex3 = 8'b00000110;
```

```
				if (data_out[3] == 1'b0)
					hex4 = 8'b00111111;
				else
					hex4 = 8'b00000110;

				if (data_out[2] == 1'b0)
					hex5 = 8'b00111111;
				else
					hex5 = 8'b00000110;

				if (data_out[1] == 1'b0)
					hex6 = 8'b00111111;
				else
					hex6 = 8'b00000110;

				if (data_out[0] == 1'b0)
					hex7 = 8'b00111111;
				else
					hex7 = 8'b00000110;

			end

   VGA_LED_Emulator led_emulator(.clk50(clk), .*);

			datapath datapath(.SW(SW), .KEY(KEY), .clk(clk),.op(op), .data_out(data_out),
.init_done(init_done), .update_done(update_done),.DONE(DONE), .data_in(writedata),
.write(write), .chipselect(chipselect));

			logic START, init_done, update_done, DONE;

			logic [3:0] idle_count;

			always_ff @(posedge clk)
			begin
				if (idle_count < 4'd10)
				begin
					START <= 1'b0;
					idle_count <= idle_count + 4'd1;
				end
				else
					START <= 1'b1;
			end

			always_comb
			begin
				if (START == 1'b1)
				begin
					if(op <= 3'd1 && init_done == 1'b0)
						op = 3'd1; //Initialization Signal
```

```
                    else
                    begin//Initialization done
                        if (DONE == 1'b1)
                            op = 3'd4; //DONE Signal
                        else
                        begin
                            op = 3'd2; //not done yet
                        end
                    end
                end
                else
                    op = 3'd0; //Idle Signal
        end

endmodule
```

First Level Comparator

```
module comp_0(input logic[10:0] in_0, input logic[10:0] in_1, output logic[10:0] u_0);

always_comb begin
if(in_0[5:0] <= in_1[5:0])
   u_0 = in_0;
else
   u_0 = in_1;
end

endmodule
```

Second Level Comparator

```
module comp_1 (
   input logic [10:0] in_0,
   input logic [10:0] in_1,
   input logic [10:0] in_2,
   input logic [10:0] in_3,
   output logic [10:0] out_1);

   logic [10:0] u_0;
   logic [10:0] u_1;

   comp_0 comp_00(.in_0(in_0),.in_1(in_1),.u_0(u_0));
   comp_0 comp_01(.in_0(in_2),.in_1(in_3),.u_0(u_1));
```

```
    comp_0 comp_16(.in_0(u_0),.in_1(u_1),.u_0(out_1));

endmodule
```

Distance Memory Module

```
// Quartus II Verilog Template
// True Dual Port RAM with single clock

module distance_memory
#(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=5)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	// Port A
	always @ (posedge clk)
	begin
		if (we_a)
		begin
			ram[addr_a] <= data_a;
			q_a <= data_a;
		end
		else
		begin
			q_a <= ram[addr_a];
		end
	end

	// Port B
	always @ (posedge clk)
	begin
		if (we_b)
		begin
			ram[addr_b] <= data_b;
			q_b <= data_b;
		end
		else
		begin
			q_b <= ram[addr_b];
		end
```

```
	end

endmodule
```

Graph Memory

```
// Quartus II Verilog Template
// True Dual Port RAM with single clock

module graph_memory
#(parameter DATA_WIDTH=16, parameter ADDR_WIDTH=5)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	// Port A
	always @ (posedge clk)
	begin
		if (we_a)
		begin
			ram[addr_a] <= data_a;
			q_a <= data_a;
		end
		else
		begin
			q_a <= ram[addr_a];
		end
	end

	// Port B
	always @ (posedge clk)
	begin
		if (we_b)
		begin
			ram[addr_b] <= data_b;
			q_b <= data_b;
		end
		else
		begin
			q_b <= ram[addr_b];
		end
	end
```

```
endmodule
```

Previous Memory Module

```
// Quartus II Verilog Template
// True Dual Port RAM with single clock

module previous_memory
#(parameter DATA_WIDTH=5, parameter ADDR_WIDTH=5)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	// Port A
	always @ (posedge clk)
	begin
		if (we_a)
		begin
			ram[addr_a] <= data_a;
			q_a <= data_a;
		end
		else
		begin
			q_a <= ram[addr_a];
		end
	end

	// Port B
	always @ (posedge clk)
	begin
		if (we_b)
		begin
			ram[addr_b] <= data_b;
			q_b <= data_b;
		end
		else
		begin
			q_b <= ram[addr_b];
		end
```

```
	end

endmodule
```

Visited Memory Module

```
// Quartus II Verilog Template
// True Dual Port RAM with single clock

module visited_memory
#(parameter DATA_WIDTH=1, parameter ADDR_WIDTH=5)
(
	input [(DATA_WIDTH-1):0] data_a, data_b,
	input [(ADDR_WIDTH-1):0] addr_a, addr_b,
	input we_a, we_b, clk,
	output reg [(DATA_WIDTH-1):0] q_a, q_b
);

	// Declare the RAM variable
	reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

	// Port A
	always @ (posedge clk)
	begin
		if (we_a)
		begin
			ram[addr_a] <= data_a;
			q_a <= data_a;
		end
		else
		begin
			q_a <= ram[addr_a];
		end
	end

	// Port B
	always @ (posedge clk)
	begin
		if (we_b)
		begin
			ram[addr_b] <= data_b;
			q_b <= data_b;
		end
		else
		begin
			q_b <= ram[addr_b];
		end
	end
```

```
endmodule
```

SOFTWARE

Main C++ Code

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <stdlib.h>
#include "graph.hpp"
#include <stdio.h>
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

using namespace std;

int vga_led_fd;

/* Write the contents of the array to the display */
void write_segments(unsigned int segs)
{
  vga_led_arg_t vla;
  int i;
  for (i = 0 ; i < VGA_LED_DIGITS ; i = i+1) {
    vla.digit = i;
    vla.segments = segs;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
      perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
      return;
    }
  }
}
```

```
int main()
{
	bool graph_bool[r][c];
	int graph_int[r][c];
	vector<Node> nodes;
	char * test;
	int i = 0;
	int j = 0;
	vga_led_arg_t vla;
	static const char filename[] = "/dev/vga_led";
	unsigned int message;
	unsigned int dumbNode1 = 0;
	unsigned int dumbNode2 = 1073741824;

	printf("\nLabyrinth Userspace program started\n");
	if ( (vga_led_fd = open(filename, O_RDWR)) == -1)
	{
		fprintf(stderr, "could not open %s\n", filename);
		return -1;
	}



	int numberNodes;

	ifstream File;
	File.open("random_maze.txt");
	char output[100];
	if (File.is_open()) {
		while (!File.eof()) {

			File >> graph_bool[i][j] ;
			j++;
			if(j == c)
			{
				i +=1;
				j = 0;
			}


		}
	}

	File.close();

	Graph graph = generateGraph(graph_bool, r, c);

	numberNodes = graph.numberNodes();
```

```
        cout << "First node to enter: "<< numberNodes -1 << endl;

        ofstream outFile;

        outFile.open("maze_output.txt");

        nodes = graph.getNodes();

        for(int i = 0; i < r; i++)
        {
                for(int j = 0; j < r; j++)
                {
                        graph_int[i][j] = graph_bool[i][j];
                }
        }

//  for(int i = 0 ; i < nodes.size();  i++)
//  {
//    int x = nodes[i].getX();
//    int y = nodes[i].getY();
//    graph_int[x][y] = 2;

//  }

int x = nodes[0].getX();
int y = nodes[0].getY();

graph_int[x][y] = 2;

x = nodes[numberNodes - 1 ].getX();
y = nodes[numberNodes - 1 ].getY();

graph_int[x][y] = 2;

unsigned int ** neighbors = graph.getNeighbors();


        message = 2147483648 + numberNodes;
        write_segments(message);

        for(int i = 0; i < numberNodes; i++)
        {
                write_segments(neighbors[i][0]);
                write_segments(neighbors[i][1]);
        }

        for(int i = numberNodes; i < 32 ; i++)
        {
```

```
		write_segments(dumbNode1);
		write_segments(dumbNode2);
	}



	char option;
	cout << "Enter best path? (y/n)" << endl;
	cin >> option;
	if(option == 'y')
	{
		unsigned int path_node1;
		unsigned int path_node2;

		cout << " Enter the nodes in the best path: " << endl;
		cout << "Node index: " << endl;
		cin >> path_node1;
		write_segments(path_node1);

		cout << "Node index: " << endl;
		cin >> path_node2;
		write_segments(path_node2);

		while(path_node1 != -1)
		{
			int x1 = nodes[path_node1].getX();
			int y1 = nodes[path_node1].getY();
			int x2 = nodes[path_node2].getX();
			int y2 = nodes[path_node2].getY();
			graph_int[x1][y1] = 2;
			graph_int[x1][y1] = 2;


			if(y1 == y2)
			{
				if(x1 < x2)
				{
					int i = x1 ;
					while(i != x2)
					{
						graph_int[i][y1] = 2;
						i++;
					}

				}
				else
				{
					int i = x2 ;
					while(i != x1)
```

```
                {
                    graph_int[i][y1] = 2;
                    i++;
                }
            }
        }
        if(x1 == x2)
        {
            if(y1 < y2)
            {
                int i = y1 ;
                while(i != y2)
                {
                    graph_int[x1][i] = 2;
                    i++;
                }

            }
            else
            {
                int i = y2 ;
                while(i != y1)
                {
                    graph_int[x1][i] = 2;
                    i++;
                }
            }
        }
        path_node1 = path_node2;
        cout << "Node index: " << endl;
        cin >> path_node2;
        if(path_node2 == -1)
            break;
        write_segments(path_node2);
    }
}


x = nodes[0].getX();
y = nodes[0].getY();

graph_int[x][y] = 2;

x = nodes[numberNodes - 1].getX();
y = nodes[numberNodes - 1].getY();

graph_int[x][y] = 2;

for(int i = 0; i < r ; i++)
```

```
	{
		for(int j = 0 ; j < c; j++)
		{
			outFile << graph_int[i][j] << " ";
		}

		outFile << "\n";
	}

	ofstream nodeFile;

	nodeFile.open("nodes.txt");

	for(int i =0; i < numberNodes; i++)
	{
	int x = nodes[i].getX();
	    int y = nodes[i].getY();
	    nodeFile << x << " " << y << "\n";

	}

	nodeFile.close();

}
```

Display Maze

```
import numpy
from numpy.random import random_integers as rand
import matplotlib.pyplot as plt
from matplotlib import colors


def main():
	f = open("maze_output.txt")
	f2 = open("nodes.txt")
	nodes = {}

	g = numpy.genfromtxt(f)
	cmap = colors.ListedColormap(['black', 'white', 'yellow'])
	bounds=[0,1,2]
	norm = colors.BoundaryNorm(bounds, cmap.N)

	fig = plt.figure(figsize=(10,10))
	ax = fig.add_subplot(111)
```

```
	line = f2.readline()
	i = 0
	while(line != ""):
		y, x = line.split(" ")
		nodes[i] = (x,y)
		line = f2.readline()
		i = i + 1

	j = 0
	for node in nodes:
		x = nodes[node][0]
		y = nodes[node][1]
		ax.annotate(str(j), xy=(0,-0.4), xytext=(x,y), fontsize = 5, color = 'red', weight = 'bold')
		j = j + 1

	plt.imshow(g,cmap=cmap,interpolation='nearest')
	plt.savefig('maze.png')
	plt.xticks([]),plt.yticks([])
	plt.show()
	plt.savefig("output2.png")

main()
```

Driver

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"

#define DRIVER_NAME "vga_led"

/*
 * Information about our device
 */
struct vga_led_dev {
	struct resource res; /* Resource: our registers */
```

```
	void __iomem *virtbase; /* Where registers can be accessed in memory */
	u32 segments;
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(int digit, u32 segments)
{
	iowrite32(segments, dev.virtbase + digit);
	dev.segments = segments;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	vga_led_arg_t vla;

	switch (cmd) {
	case VGA_LED_WRITE_DIGIT:
		if (copy_from_user(&vla, (vga_led_arg_t *) arg,
				   sizeof(vga_led_arg_t)))
			return -EACCES;
		if (vla.digit > 3)
			return -EINVAL;
		write_digit(vla.digit, vla.segments);
		break;

	case VGA_LED_READ_DIGIT:
		if (copy_from_user(&vla, (vga_led_arg_t *) arg,
				   sizeof(vga_led_arg_t)))
			return -EACCES;
		if (vla.digit > 3)
			return -EINVAL;
		vla.segments = dev.segments;
		if (copy_to_user((vga_led_arg_t *) arg, &vla,
				 sizeof(vga_led_arg_t)))
			return -EACCES;
		break;

	default:
		return -EINVAL;
	}
```

```
	return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_led_fops = {
	.owner		= THIS_MODULE,
	.unlocked_ioctl = vga_led_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
	.minor		= MISC_DYNAMIC_MINOR,
	.name		= DRIVER_NAME,
	.fops		= &vga_led_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
	static unsigned int welcome_message[VGA_LED_DIGITS] = { 0x00000000 };
	int i, ret;

	/* Register ourselves as a misc device: creates /dev/vga_led */
	ret = misc_register(&vga_led_misc_device);

	/* Get the address of our registers from the device tree */
	ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
	if (ret) {
		ret = -ENOENT;
		goto out_deregister;
	}

	/* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
				DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}
```

```
	/* Display a welcome message */
	for (i = 0; i < VGA_LED_DIGITS; i = i+1)
		write_digit(i, welcome_message[i]);

	return 0;

out_release_mem_region:
	release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
	misc_deregister(&vga_led_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_led_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
	{ .compatible = "altr,vga_led" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
	.driver = {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(vga_led_of_match),
	},
	.remove	= __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
```

```
static void __exit vga_led_exit(void)
{
	platform_driver_unregister(&vga_led_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Ariel Faria, Michelle Valente, Utkarsh Gupta, Veton Saliu");
MODULE_DESCRIPTION("VGA Bouncing Led Emulator");
```

Graph Generation Code

```
#include <vector>
#include <iostream>

using namespace std;

const int r = 11;
const int c = 11;

class Node{
private:
    int x;
    int y;
    int index;
public:
    Node(int x, int y, int index)
    {
        this->x = x;
        this->y = y;
        this->index = index;
    }

    int getX()
    {
        return x;
    }

    int getY()
    {
        return y;
    }

    int getIndex()
```

```
    {
        return this->index;
    }

    void showInfo()
    {
        cout << "index: " << index << endl;
        cout << "x: " << x << endl;
        cout << "y: " << y << endl;
    }

};
vector<Node> nodes;

class Graph {
private:
    int** adjMatrix;            // Adjacency Matrix
    int numV;                   // Number of vertices
    //vector<Node> nodes;
public:
    Graph(int V) {
        numV = V;
        adjMatrix = new int*[numV];
        for (int i = 0; i < numV; i++) {
            adjMatrix[i] = new int[numV];
            for (int j = 0; j < numV; j++)
                adjMatrix[i][j] = 0;
        }
    }

    void addEdge(int i, int j, int distance) {
        if (i >= 0 && i < numV && j > 0 && j < numV) {
            adjMatrix[i][j] = distance;
            adjMatrix[j][i] = distance;
        }
    }

    void removeEdge(int i, int j) {
        if (i >= 0 && i < numV && j > 0 && j < numV) {
            adjMatrix[i][j] = 0;
            adjMatrix[j][i] = 0;
        }
    }

    int valueEdge(int i, int j) {
        if (i >= 0 && i < numV && j > 0 && j < numV)
            return adjMatrix[i][j];
        else
            return 0;
```

```
}

void addNode(int x, int y, int index)
{
    Node newNode(x,y,index);
    nodes.push_back(newNode);
}

void showMatrix()
{
    for(int i = 0; i < numV ; i++)
    {
        for(int j = 0; j < numV; j++)
        {
            cout << " " << adjMatrix[i][j] ;
        }
        cout << "\n" ;
    }
}

int numberNodes()
{
    return numV;
}

unsigned int ** getNeighbors()
{
    unsigned int ** neighbors = 0;
    unsigned int ** neighbors32 =0;
    neighbors32 = new unsigned int*[numV];
    neighbors = new unsigned int*[numV];
    int numNeighbor = 0;
    for(int i = 0; i < numV ; i++)
    {
        neighbors[i] = new unsigned int[4];
             neighbors32[i] = new unsigned int[2];
        for(int j = 0; j < numV; j++)
        {
            int bin_index = j * 64;
            if(adjMatrix[i][j] != 0)
            {
                int bin_value = adjMatrix[i][j] + bin_index;
                neighbors[i][numNeighbor] = bin_value;
                numNeighbor++;
            }
        }
        // If doesnt have 4 nodes, complete with itself.
        if(numNeighbor != 4)
        {
```

```
                for(int w = numNeighbor; w < 4 ; w++)
                    neighbors[i][w] = i * 64;
            }

            neighbors32[i][0] = neighbors[i][0] + (32768 * neighbors[i][1]);
            neighbors32[i][1] = neighbors[i][2] + (32768 * neighbors[i][3]);
                 neighbors32[i][1] = 1073741824 + neighbors32[i][1];
            numNeighbor = 0;
        }

        return neighbors32;
    }

    void showNeighbor(int node_num)
    {
        Node node = nodes[node_num];
        string x = to_string(node.getX());
        string y = to_string(node.getY());
        cout << "Neighbors node " << node_num  << "( " << x << "," << y << " )" << ": ";
        for(int i =0 ; i < numV; i++)
        {
            if(adjMatrix[node_num][i] != 0)
                cout << i << ", ";
        }

        cout << "\n";
    }

    void showAllNeighbors()
    {
        for(int i =0 ; i < numV; i++)
        {
            showNeighbor(i);
        }
    }

    vector<Node> getNodes()
    {
        return nodes;
    }

    int ** getGraph()
    {
        return adjMatrix;
    }



    ~Graph() {
        for (int i = 0; i < numV; i++)
```

```
                delete[] adjMatrix[i];
            delete[] adjMatrix;
        }

};

bool isCorner(bool maze[r][c], int i, int j)
{
        int left =0;
        int right = 0;
        int top = 0;
        int bottom = 0;
        int total = 0;

        if(maze[i][j] == 0)
            return false;

        if( j > 0)
            if(maze[i][j - 1] == 1)
                left = 1;

        if( i > 0)
            if(maze[i - 1][j] == 1)
                top = 1;

        if(maze[i][j + 1] == 1)
            right = 1;
        if(maze[i + 1][j] == 1)
            bottom = 1;

        total = right + bottom + top + left;

        if( total == 1 )
            return true;

        else if(total > 2)
            return true;

        else
        {
            if((top == 1 && bottom == 1) || (left == 1 && right == 1))
                return false;
            else
                return true;
        }

}

int numberNodes(bool maze[r][c], int length, int width)
```

```
{
    int countNodes = 0;
    for(int i = 0; i < length; i++)
    {
        for(int j = 0; j < width; j++ )
        {
            if(isCorner(maze, i, j))
            {
                countNodes++;
                Node newNode(i,j,countNodes);
                nodes.push_back(newNode);
            }
        }
    }

    return countNodes;
}

int isConnected(Node node1, Node node2, bool maze[r][c])
{
    int x1 = node1.getX();
    int x2 = node2.getX();
    int y1 = node1.getY();
    int y2 = node2.getY();
    int temp = 0;
    int value = 1;

    if(x1 == x2)
    {
        if(y1 < y2)
        {
            temp = y1;
            value = maze[x1][temp];

            while(temp != y2 && value != 0 )
            {
                temp++;
                value = maze[x1][temp];
                if(isCorner(maze, x1, temp) && temp != y2 && temp != y1)
                    return 0;
            }
            if(temp == y2)
                return (y2 - y1);
            else
                return 0;
        }
        else
        {
            temp = y2;
```

```
            value = maze[x1][temp];

            while(temp != y1 && value != 0 )
            {
                temp++;
                value = maze[x1][temp];
                if(isCorner(maze, x1, temp) && temp != y1 && temp!= y2)
                    return 0;
            }
            if(temp == y1)
                return (y1 - y2);
            else
                return 0;
        }

    }

    if(y1 == y2)
    {

        if(x1 < x2)
        {
            temp = x1;
            value = maze[temp][y1];

            while(temp != x2 && value != 0 )
            {
                temp++;
                value = maze[temp][y1];
                if(isCorner(maze, temp, y1) && temp != x2 && temp != x1)
                    return 0;
            }

            if(temp == x2)
                return (x2 - x1);
            else
                return 0;
        }
        else
        {
            temp = x2;
            value = maze[temp][y1];

            while(temp != x1 && value != 0 )
            {
                temp++;
                value = maze[temp][y1];
                if(isCorner(maze, temp, y1) && temp != x1 && temp != x2)
                    return 0;
```

```
            }
            if(temp == x1)
                return (x1 - x2);
            else
                return 0;
        }

    }

    return 0;
}

Graph generateGraph(bool maze[r][c], int length, int width)
{
    int num_nodes = numberNodes(maze, length, width);

    Graph g(num_nodes);

    int edge;

    for(int i = 0; i < nodes.size(); i++)
    {
        Node temp1 = nodes[i];

        for(int j = 0; j < nodes.size(); j++)
        {
            if(j != i)
            {
                Node temp2 = nodes[j];
                edge = isConnected(temp1,temp2, maze);

                if(edge != 0)
                {
                    g.addEdge(temp1.getIndex() - 1, temp2.getIndex() - 1, edge );
                }
            }
        }

    }

    return g;
}
```
