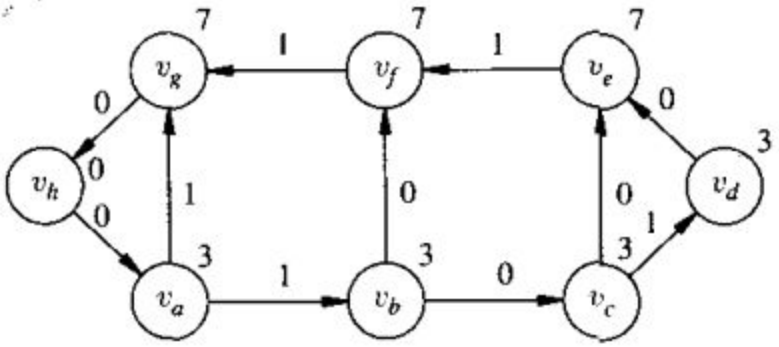
Sean Liu sl3497
Ben Zhu bjz2107

Final Project Report: Retiming Tool

Overview:

This project seeks to read in a input graph, representing some structure with vertices and



edges,
and then retime it to meet certain timing constraints.

We used python, as well as a package networkx to store our graph information. No built in functions were used other than initial graph production.

Each vertex is assigned a certain delay to it, and each edge has a weight that corresponds to register count on it.

To keep track of retiming, we create a retiming vector, with each vertex getting assigned a value for r(vector) depending on the movement of registers relative to it. The theorem we are using says for any node that does not meet timing, r(v) = r(v)+1, which means a register will move from the output to its input.

This corresponds to a 1 in r(v) , and a change in edge weights for the various involved edges. A retiming vector initially is <0,0,0,0,0,0,0> for example, but if the circuit is retimed to meet a certain clock period, then it can change to <1,0,0,0,0,0,1> for example.

In part 1, the method in which confirmed a targeted cycle time against an attainable clock period was done via CP function, taken from Handout #33. This produced a vector of all clock periods for each path for the nodes. This relied on the production of G0, or the subset of the initial graph containing only edge weights of 0. By removing all register paths, this guaranteed the path that would produce the greatest cycle delay moving from register to register.

We produced the phi_init using W and D matrices. W and D represented the full set of weights between any paths or edges, and D the corresponding delay for each path. To find the phi_init, we essentially reduced the W matrix to the subset of items in G0, and then looked for the max delay value in that subset of corresponding weights. The W and D matrices were produced simultaneously, using the Floyd-Warshall to find each shortest path, and find the weights and delays associated with every single path.

We then implemented FEAS, an algorithm that determines if a clock period is feasible. This is done via a simple algorithm, that says for any nodes that do not meet the clock period, perform retiming, with r(v) = r(v)+1. Every node that does not meet the clock period must be touched by the movement of a register. This would then give us a retiming vector if the timing was possible, and an updated graph.

Sean Liu sl3497
Ben Zhu bjz2107

This was used in algorithm OPT2, which aimed to take in the D matrix, sort it for unique values, and then binary search for the minimum delay that was possible. We would run FEAS on every binary searched item, and FEAS would then find the minimum possible clock delay.

In Part 2, we implemented a Linear Programming algorithm, using Simplex, to solve a set of inequalities.

$$\min \quad \mathbf{c}^T \mathbf{r} \quad \text{such that}$$

7.1 $$r_i - r_j \leq w_{ij} \qquad \forall (v_i, v_j) \in E$$

7.2 $$r_i - r_j \leq W(v_i, v_j) - 1 \qquad \forall v_i, v_j : D(v_i, v_j) > \phi$$

The goal was to first find c. C was a vector containing the difference between incoming and outcoming edges of a vertex. This was quite easily accomplished. Then, because we needed to formulate the constraints, we looked for each constraint given some retiming vector. Then, we optimized the 7.2, the second inequality, by eliminating any where D(u,v) - (d(v) or d(u) > phi, because it made no sense to be positive for a register for a path where the subpath would already have a positive on it.

We implemented the simplex by first passing through our Tableau into our basic solve function. This function reads in our Tableau as well as the numerical indices of all the slack variables, and performs the simplest possible solution, by dividing the value in the answer column by the variable's coefficient to find our row solution. In this, we will find infeasible solutions, because slack variables should never be negative, and a solution of p = 0 (or actually, cTr = 0) tells us nothing. Here is where we ran into some issues. We formulated the tableau and updated it each time and eventually produced a final retiming vector, but only when we input the desired minimum clock period that we knew, from part 1, did the retiming vector return correct. When we input other clock timings, our implementation seemed overeager to add registers. As in, it overadded registers, when the timing was already able to be met with a simpler implementation.

We could think of two reasons as to why this was the case. One, in the method in which we implemented the Simplex, we somehow were unable to return a retiming vector of all zeros if no retiming was needed. This is due to the method in which the inequalities are solved - the initial cTr resulted in a solution that always had a negative slack variable, and since we were not allowed those, the basic solution was never allowed either. I believe this had something to contribute to the issues we were seeing.

The Data Structures:

We stored the synchronous network using networkx. Originally we made our own data structure of dictionaries to store the vertices, edges, weights and delays independently, but this led to issues in following the algorithms exactly as is, and reformulating the graph correctly in

each iteration. Because of this, we utilized a package called networkx which allowed us to play with the synchronous graph far more robustly.

W and D were calculated via Floyd Warshall. FEAS was implemented as described in the two references, which would retime any node that did not meet timing, hopefully discovering if the timing was feasible or not.

Constraints were generated via the two inequalities. We did not have to worry about timing feasibility, but if we had to have, we would need to add an additional constraint of

$$\mathbf{c}^T\mathbf{r} \leq \bar{a} - \sum_i \sum_j w_{ij}$$

Which would allow us to account for timing feasibility.

The simplex and tableau method we followed the tutorials as described. We had to reverse the min problem into a max problem. We also needed to account for the basic solution not being feasible, which we checked for before taking the basic solution and proceeding with the simplex method.

All the algorithms were implemented as carefully and as strict to the examples provided as possible. We produced WD simultaneously, and did all of our algorithms with the same number of iterations in each loop as described in the book. Constraint generation was done with some shortcuts, as we knew that each constraint only involved two of the actual r variables, one being 1 and the other -1. We then also knew that we could simply append on the resulting slack variable, because there would be the edge number of slack variables, and then constraints depending on 7.2, and then cut out the not needed ones as described by the optimization before. Simplex was accomplished by checking for the most negative (smallest) in the p row, so the cTr row, and then finding the smallest ratio in the pivot column of answer/that item in column. And then pivoting around that i,j. We performed this until there were no more columns to pivot over, and then the resulting active variables told us the maximum cTr solution, as well as provided us with a retiming vector.

Technical Questions

a)the point of FW methodology is to essentially create the matrix, and then populate it at each step.

Iteration 0 has all the edges available immediately from the graph, with their various weights. Therefore, the matrix gains information in each i,j column for which the edges are already defined by the network.

At iteration 1, all paths that go through the first vertex are found. If any path located here is shorter than one initially defined via the edges, then the weight may be replaced in the matrix. The solution defined here is simply all the edge network weights, plus the paths that pass through vertex one that have shorter delay, which will overwrite the edge network weight.

At iteration 2, the paths going through vertices 1,2 are found. This builds on previous information. As we know that certain paths we find may have a lower weight than the initially

defined edges, we will follow the path defined at that node to find the next node we're looking for if that path is already defined.

For the ith iteration, all vertices (1, 2 … to i) are checked. If i = number of vertices, then all shortest paths should have already have been found. Otherwise, this algorithm continues to check the paths that pass through the vertices (1 to i) and follow the paths already found to be shortest.
By now, the final solution would have built on all possible shortest paths including every vertex necessary, as well as identifying the optimal path for ones containing certain nodes. The matrix continually gets updated if a shorter path is found. Our W matrix simply returns the weight for every single possible path, and then the shortest path between those.

b) Vertex v_k has a delay of d_k =30. The target clock cycle is 20. The vertex itself has a delay that is greater than the clock period, so there is no way to meet timing constraints.

$$\min \quad \mathbf{c}^T\mathbf{r} \quad \text{such that}$$

$$r_i - r_j \le w_{ij} \qquad \forall (v_i, v_j) \in E$$

$$r_i - r_j \le W(v_i, v_j) - 1 \qquad \forall v_i, v_j : D(v_i, v_j) > \phi$$

We run into an issue here because we can create the constraints for the edges, but for the second set of constraints, when the delay of a vertex is greater than phi already, every single path containing that vertex violates timing. So you have a 2 constraints that contain v_k once, or perhaps more, based on how many edges there are. Then, there are … all paths with v_k in it will violate timing, so any path that contains it will created via the second constraint. However, with our optimization, any D(vi, vj) -d(vi) or d(vj) is still > phi, means that you don't consider any of those constraints, and it all boils down to the constraint exclusively concerning paths eventually reaching v_k. However… these still violate timing, and the issue is, D(u, v) - d(v) > c or D(u,v) - d(u) > c.
Well… if the v_k is either one of the u or the v's, then the condition is always met by subtracting the other node. For some tabulated delay, there is no way d_k +d(u) - d(u) <30… unless some intermediate delays are negative, which isn't the case. There seems to simply not be a way to solve this because the constraints do not have adequate linkages to each other to be solved. No retiming vector is feasible as a result.

c) retiming

Sean Liu sl3497
Ben Zhu bjz2107

i) When you say legal retiming vector.. If it's legal then it is able to meet a certain timing constraint. Generally, we've seen by adding registers, clock cycle goes down -> the trade off of area and timing. Needlessly adding registers certainly affects weight, but if you think about it, by oversaturating with registers, you never increase delay- in fact, G0 gets even more sparse, and delay would never increase. Of course, by overadding registers arbitrarily, rather than retiming, you're pipelining. So we'd have to follow some rules. Using c, we know that certain vertex movements of registers add registers, and some remove registers.

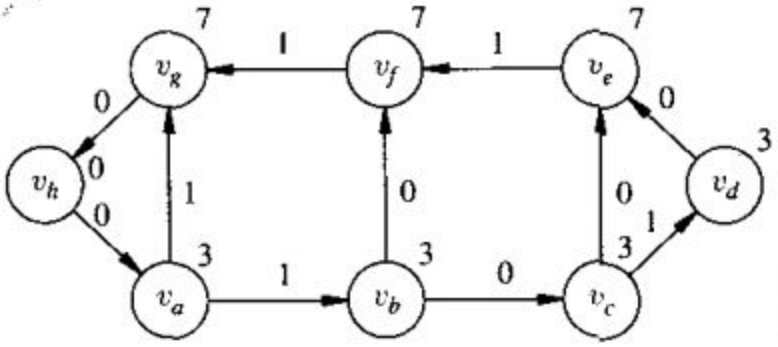


FIGURE 9.10
Retimed network.

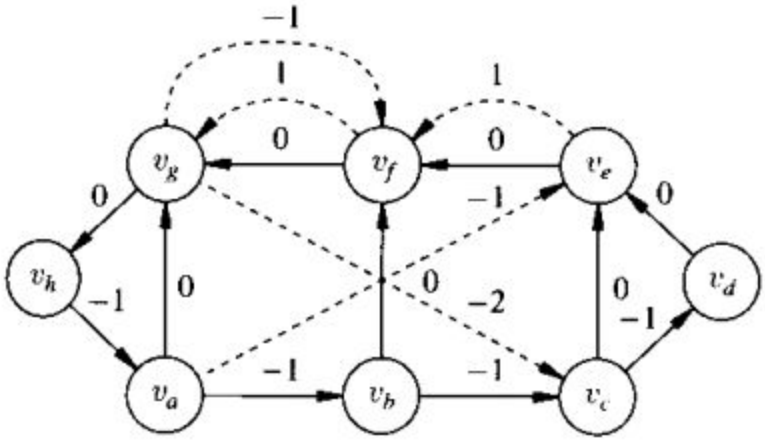


FIGURE 9.11
Constraint graph.

I believe that every retiming vector that is larger than the optimal can still be a feasible solution given it is a legal retiming vector, which it is. Movements across vertices with a c that is negative means that there are more outputs than inputs, which if there is a register movement, then the # of registers will decrease. If C is positive, then the movement will gain registers. So it's totally possible to have different sizes with legal retiming vectors.

ii) Smaller retiming vectors, which might still be legal could be a feasible solution. I don't know if this is always true. The cases seem to be, if the legal retiming vector r_opt is a valid solution, then something smaller requires r_opt to not be the optimal retiming vector -> otherwise that makes no sense, to have a retiming vector that can do better than the optimal solution. However, we've learned that the first set of constraints isn't overly concerned with register count

or area… and the second linear programming method we used minimized area for a given timing. So it's possible, that given some timing, there is an undiscovered more optimal retiming. However, I'm leaning more to the side of it's not always a feasible solution. This is because at the optimal solution, it makes no sense to be able to get a free lunch - to trade for less registers, but the same timing might make sense at a non-optimal solution, but this is already down to the edges of functionality for the circuit. To simply be able to take away a register and expect the same timing makes no sense.

iii) I mean… tons of legal retiming vectors r_x are smaller, and feasible solutions. The vast majority of them do not come close to meeting cycle time demands though.If we're looking at "smaller," as mentioned before, with C, retiming can gain you registers and lose you registers. You simply have to just not gain as many registers as r_opt in retiming. This problem requires meeting phi=13, and… you can meet it by adding registers, but by taking away registers the G0 path delay can only grow larger.

d) Simplex
i)Pivot Choice negative
The pivot column corresponds to the most influential variable, which has the greatest ability to increase the value of the objective function. By pivoting on an item in that column, we guarantee the removal of the matrix item in the objective function row results in the greatest increase of the objective function "p" as notated in the tutorial. We're sticking to positive pivot variables inside the pivot column, so it requires the most negative item in choice of pivot column.

ii) other negative columns
This is like .. retiming the 2nd largest vertex that violates timing constraints. Of course, it fixes some issues, but it's not the optimal approach, and will not maximize cTr in our problem. By arbitrarily not picking to maximize cTr, you are shortchanging yourself of just how large cTr can be.
So in the first one, if you look at the iteration i compared to i-1, the objective cost (what exactly is this term???) of the basic solution at iteration i will be ...less? I have no idea what you mean by objective cost.
If you're speaking in terms of registers, then clearly, from iteration i to iteration i-1, you are still moving in the right direction, but cost will increase. The value we're pivoting around is not the optimal value, and so cost can be added on arbitrarily. It is like advertising to meet a set population, and rather than trying to minimize your costs, you are saying I'm willing to pay a little more to reach the right numbers but I don't care if I overadvertise.

For iteration i versus the cost of using a smaller magnitude pivot, this is clearly not optimal. Pivoting around the best choice ensures that the path we take is the correct decision to make. If you pivot on a smaller magnitude pivot, then costs will be higher, (or eventually they will be higher). This is like saying I can increase production of x by 1 unit for a profit increase, or I can increase production of y by 1 unit for a bigger profit increase, but then picking to increase x. Makes no sense.