CS1510 Reduction Problems 19, 20, & 23, Parallel Problems 2 & 3

Rebecca Negley, Sean Myers

November 2, 2011

19. Show by reduction that if the decision version of the SAT-CNF problem has a polynomial time algorithm then the decision version of the 3-coloring problem has a polynomial time algorithm.

    Solution: We will reduce the 3-coloring decision problem to the SAT-CNF decision problem. We are given an input graph G in the 3-coloring problem. We want to tranform this graph into a CNF formula that is satisfiable if and only if the graph is 3-colorable. Then, we input this formula into SAT-CNF decision and return its true/false output as the output of the 3-coloring problem. To transform the graph: For each vertex v, create terms v_1, v_2, and v_3 to go into the CNF formula. If v_1 is true, v is color 1. If v_2 is true, v is color 2. If v_3 is true, v is color 3. Clearly, we need one and only one of these three terms to be true. To ensure that one of them is true, we create the following clause: $(v_1 \vee v_2 \vee v_3)$. We cannot have 2 of these be true because v can only be one color, so we also include: $(\bar{v_1} \vee \bar{v_2}) \wedge (\bar{v_1} \vee \bar{v_3}) \wedge (\bar{v_2} \vee \bar{v_3})$. If two or three of the terms representing v are true, one or all of these clauses will be false. We then have that if $(v_1 \vee v_2 \vee v_3) \wedge (\bar{v_1} \vee \bar{v_2}) \wedge (\bar{v_1} \vee \bar{v_3}) \wedge (\bar{v_2} \vee \bar{v_3})$ is satisfiable, we can assign one color to v. Also, if we can assign one color to v, we can satisfy these clauses by setting the term corresponding to that color to true and the other two terms to false. We do this with every vertex and AND all the clauses together. Since we do a constant amount of work for each vertex, this takes linear time.

    Now we must account for the edges in the graph to prevent adjacent vertices from being the same color. Say we have adjacent vertices v and w and color 1. We do not want v and w to both be color 1. This means we want $\overline{(v_1 \wedge w_1)} = (\bar{v_1} \vee \bar{w_1})$. We also do not want them both to be color 2 or both to be color 3, so in total for this edge we have: $(\bar{v_1} \vee \bar{w_1}) \wedge (\bar{v_2} \vee \bar{w_2}) \wedge (\bar{v_3} \vee \bar{w_3})$. If this is satisfiable, the two vertices must be different colors. Is it possible to assign different colors to the vertices but have the formula unsatisfiable? No, because if we set the colors corresponding to each vertex and the assigned colors to true and all others to false, the formula will be true. We do this with all edges (with number of edges at most quadratic) and AND all of the clauses (including the ones from above) together. We then have that the CNF formula is satisfiable if and only if one color can be assigned to every vertex such that no pairs of adjacent vertices are the same color. Hence, this formula is satisfiable if and only if the graph is 3-colorable. Since we converted the graph to the CNF formula in polynomial time, if we have a polynomial time algorithm for SAT-CNF decision, we have a polynomial time algorithm for the 3-coloring decision problem.

20. In the dominating set problem the input is an undirected graph G, the problem is to find the smallest dominating set in G. A dominating set is a collection S of vertices with the property that every vertex v in G is either in S, or there is an edge between a vertex in S and v. Show that the dominating set problem is NP -hard using a reduction from the vertex cover problem

    Solution: We will reduce the vertex cover problem to the dominating set problem. A graph G is input into the vertex cover problem. We want the smallest set of vertices such that each edge is incident to a vertex in the set. Create a graph G' originally equal to G. If vertices are allowed to be

of degree 0 (isolated), remove them from G' as they cannot be in the vertex cover. It takes at most quadratic time to do this if we must check adjacency. We need to be sure that at least one vertex is chosen to represent every edge. To do this, for every edge in G, we create a third vertex in G' and create edges between the new vertex and both of the ones incident to the original edge. Then, we input G' into the dominating set problem to get the smallest dominating set of G'. We can convert the smallest dominating set of G' into the smallest vertex cover of G in the following way: If a vertex in the dominating set of G' was originally in G, include that vertex in the vertex cover. If a vertex in the dominating set was one of those added to G', add either adjacent vertex (choose arbitrarily) into the vertex cover.

First, we need to show that this set, call it S, is actually a vertex cover. Suppose there is some edge in G that is not incident to any vertices in S. Then, both of its incident vertices in G and the third vertex created from them in G' were excluded from the dominating set. However, this means that the vertex added to G' for this edge was not in the dominating set or adjacent to any points in the dominating set, a contradiction. Hence, S must be a vertex cover. Suppose there exists some vertex cover S' of G smaller than S. Then, S' must be a dominating set of G'. Suppose not. Consider some vertex v in G' not in S'. Suppose it was one of the added vertices. Since at least one vertex on every edge in G must be in S', v must be adjacent to some point in S' (because it is adjacent to the only two points along one of the edges in G). Suppose $v \in G \backslash S'$. If v is connected to some other point, the other point must be in S' so that the edge is incident to a vertex in S'. If v is allowed to be isolated, it must not be in the vertex cover and was excluded from G'. Hence, all vertices in G' are either in S' or adjacent to a vertex in S', so S' must be a smaller dominating set of G', a contradiction. Therefore, our output must be correct. We created G' in time quadratic in the number of vertices because we went through every edge. We created our output of the vertex cover porblem from the output of the dominating set problem in time linear in the number of vertices because we scanned through every vertex. This gives that if a polynomial time algorithm exists for the dominating set problem, a polynomial time algorithm exists for the vertex cover problem. Since the vertex cover problem is NP-hard, the dominating set problem is therefore NP-hard.

23. In the disjoint paths problem the input is a directed graph G and pairs $(s_1, t_1), ..., (s_k, t_k)$ of vertices. The problem is to determine if there exist a collection of vertex disjoint paths between the pairs of vertices (from each s_i to each t_i). Show that this problem is NP -hard by a reduction from the 3SAT problem. Note that this problem is not easy.

Solution: In order to do this, we must take formula F from 3SAT and convert it into an input for disjoint paths in polynomial time and where the output of 3SAT is 1 if and only if the output to disjoint paths is 1.

To start off, we create x starting vertices, one for each variable in formula F. We will then add 2 vertices to each starting vertex, representing the negated form and the affirmative form of the variable. Each variable and its negation will equal a collection of vertices that it is currently at (will explain more on this later), which will be at most 2, and at least 1. For now, each starts at 1.

For each clause, a "gadget" will be created. A gadget is just a clause broken down into path form in such a way that the properties of the clause are reflected in the gadget, so that way it forms a 1 to 1 relational output among the two algorithms. These gadgets will not be made in parallel, but in pipeline. For each variable that is input to the gadget, any later clause that needs to connect that clause, will connect it from the gadget, not the original vertex. At the end of all the clauses, the last pipe will be the end node t_i which will connect the original variable and its negation from the last gadgets that use those variables. Refer to this figure 1 for a pictoral demonstration.

How do we create a gadget from a clause? Well, we must first see what a clause is made of. The clause is made up of three literals, all or'ed together. What the OR means is that 1 or 2 or 3 of those literals

can equate to true, and the clause will be true. An equivalent way of saying this, is that if 2 or less of the negated literals of the clause are chosen, then the clause will equate to true. So if we are given a clause of (x v y v z), we would create two vertices to connect (x, y, z) to both of these. What does this mean for the disjoint paths problem? It means that there can only be two paths can be created, and it needs to choose another variable path to traverse. Just like in 3-SAT, the only way to satisfy the clause is to choose two or less variables, the same goes for the path problem, it can only choose two paths in a given gadget(clause). Back to the example, the next time we see an x, y or z, we take the two vertices that were produced in the previous gadget made, and feed those as inputs into the new gadget. So at each Gadget, we need to update the collection of vertices that input variables are at, and use those as inputs into the next gadget. This process continues until we run out of clauses, and then we create the last vertices t_i for each literal of a variable, and then connect all the updated variable instances into their respective t_i. This path will then output the correct ouput to the 3SAT problem.

An algorithm would look something like this:

```
3SAT(formula F):
    Graph G
    List<SomeDataStructureWithANameAndACollection> vars
    for each(letter in F):
        G.add(new Vertex(letter+"start"))
        G.addVertex(new Vertex(letter)).addEdgeFrom(letter+"start")
        G.addVertex(new Vertex("negated" letter)).addEdgeFrom(letter+"start")
        vars.add(new DC(letter, G.getVertex(letter))
        vars.add(new DC("negated"+letter, G.getVertex("negated"+letter))

    for each(clause in F):
        G.add(new Gadget(clause))
        foreach(literal in clause):
            vars.getNegatedform.updateCollection(g.getGadget(clause))

    for each(letter in F):
        G.add(new Vertex("endNode"+letter)
        G.addEdges(vars.get(Letter),"endNode"+letter)
        G.addEdges(vars.get("negated"+letter),"endNode"+letter)
    return path(G)
```

2. You know that lots of famous computer scientists have tried to find a fast efficient parallel algorithm for the following Boolean Formula Value Problem:
INPUT: A Boolean formula F and a truth assignment A of the variables in F .
OUTPUT: 1 if A makes F true, and 0 otherwise.
Moreover, most computer scientists believe that there is no fast efficient parallel algorithm for the Boolean Value Problem. You want to find a fast efficient parallel algorithm for some new problem N. After much effort you can not find a fast efficient parallel algorithm for N, nor a proof that N does not have a fast efficient parallel algorithm. How could you give evidence that finding a fast efficient parallel algorithm for N is at least as hard of a problem as finding a fast efficient parallel algorithm for Boolean Formula Value problem? Be as specific as possible, and explain how convincing the evidence is.

   Solution:In order to show that is just as hard to find a fast efficient parallel algorithm for N, we must reduce Boolean Formula Value Problem to N and have the conversion take less than or equal to poly-log time $O(log_k(n))$. A reduction algorithm would look something like this:

   boolValue(n,p):

Do some kind of conversion to make it so that the output of Boolean formula is 1 if and only if the output of N is
Can use the p processors to do this in parallel time if it is more efficient to do so)
return N(newInput, p);

3. Consider the problem of taking as input an integer n and an integer x, and creating an array A of n integers, where each entry of A is equal to x.

   - Give an algorithm runs in time O(log n) on a EREW PRAM using n processors. What is the efficiency of this algorithm?

     Solution: Since we are using an EREW machine, we cannot have each of the n processors read x at the same time. Instead, we will have one processor read x. At each time interval, each processor that knows the value of x will share the information with one processor that does not know the vaule of x. After $log\ n$ steps, each processor knows the value of x. Since we are storing in an array, each processor can actually store in a different memory location, so we can have each of the n processors write x to a unique array location in one time unit. Then, this algorithm takes $O(logn)$ time. Sequentially doing this problem would take time n, so the efficiency of the algorithm is $E(n,n) = \frac{n}{n*log\ n} = \frac{1}{log\ n}$.

   - Give an algorithm that runs in time O(log n) on a EREW PRAM using $n/log\ n$ processors. What is the efficiency of this algorithm?

     Solution: We can give the information about the value of x to each of the $n/log\ n$ processors in $log(n/log\ n) = O(log\ n)$ time using the same method as above. We can then use $log\ n$ write steps where, at each step, x is written at $n/log\ n$ array locations (one for each processor). Each processor writes to an unfilled location at every time, and no two processors try to write to the same location. Each step takes one time unit, so the whole writing stage takes $log\ n$ time. Hence, this algorithm takes $O(log\ n)$ time. Its efficiency is $E(n, log\ n) = \frac{n}{n/log\ n*log\ n} = 1$.

   - Give an algorithm that runs in time O(1) on a CRCW PRAM using n processors. What is the efficiency of this algorithm?

     Solution: Since this is a CRCW machine, each of the n processors can read x at the same time, taking 1 time unit. Then, they each write to their own memory location in array A, taking another time unit. Hence, this whole algorithm runs in time $O(1)$. Its efficiency is $E(n,n) = \frac{n}{n*1} = 1$.