CS1510 Reduction Problems 17, 18, & 22, Parallel Problem 1

Rebecca Negley, Sean Myers

October 31, 2011

17. Consider the following problem. The input is an undirected graph G and an integer k. The problem is to determine if G contains a clique of size k **AND** an independent set of size k. Recall that a clique is a collection of mutually adjacent vertices, and an independent set is a collection of mutually nonadjacent vertices. Show by reduction that if this problem has a polynomial time algorithm then the clique problem has a polynomial time algorithm.

    Solution: In order to show what the problem is asking, we must reduce the clique problem into the problem described above, let's call it CIS(for Clique and Independent Set). We must first convert the clique problem into a CIS problem, so that the output of the CIS problem is 1 if and only if the output to the clique problem is 1.

    The thing we must realize is that if there is that if we want the output of CIS to be 1 when clique is 1, we must manually insert an independent set of size k at all times. That way, the only thing CIS is looking for is the clique of size k. In order to create an independent set of size k, pick any vertex v in graph G. Add k vertices to graph G. For each vertex added, add an edge from that vertex to vertex v.

    We must now examine the effectiveness of this conversion. Can this algorithm create a false positive, where there isn't a clique of size k, but doing this will create an output of 1 in CIS? No. Since each vertex added only has one edge, it can only have one mutually adjacent vertex (in this case v), and to have a clique size of $k > 2$, more than one vertex has to connect to the vertices added. Since no edge like that is added, it stands that when $k > 2$ there is no clique created . There is one special case we do need to consider, which is a 2 vertex clique, with a 1 vertex graph, which adding this extra vertex would indeed create a clique. In order to fix that special case, run an if to see that the graph size of G is greater than k, if it is return 0 since it is impossible to have a clique of size k when the input vertices is less than k.

    The other case is a false negative, where the input to CIS will produce a 0, when it should've found a clique of size k. This can't happen either, since it will always find an independent set of size k (because we inserted one), which means the only time it can output a 0 is if it can't find a clique of size k. If it can't find it, then it doesn't exist, and can't be a false negative.

    The algorithm would look something like this:

```
Clique(G, k):
    if(G.size ¡ k): return 0
    v = G.get(0)
    for(int i = 0 to k):
        Vertex l = new Vertex
        v.addEdge(l)
    return CIS(G, k)
```

18. Consider the following problem. The input is an undirected graph G and an integer k. The problem is to determine if G contains a clique of size k **OR** an independent set of size k. Show by reduction that

if a problem has a polynomial time algorithm then the clique problem has a polynomial time algorithm.

Solution: We will reduce the clique problem to the given problem. In the clique problem, we are given an undirected input graph G. First, we will let K be the largest value of k such that the given problem with inputs G and k returns true. Since $K \leq v$, with v the total number of vertices, we can begin with $k = v$ and decrement K until the given problem returns true. We must do this no more than v times. We do not know if K is the size of the largest independent set or the largest clique (or possibly both). Choose some vertex w_1 in G arbitrarily. Create G_1' to be G with w_1 (and all corresponding edges) removed. If the given problem returns true with inputs G' and K, choose another vertex w_2. Create G_1'' to be G_1' with w_2 (and all corresponding edges) removed. Continue in this pattern until we reach the first w_j such that the graph $G_1^{[j]}$ (with w_j removed) contains no clique or independent set of size K. Then, we know that w_j is in a clique or independent set of size K. We note this and move on, keeping w_j in our graph. We exclude points until we have only K points remaining, which make up our clique or independent set. Check if any two points are adjacent. If they are, we have a size K clique, and we can return it as our largest clique. (If a larger clique existed, K would have been larger.) Otherwise, we have a size K independent set. We want to make this not be an independent set, so we create $G_2 = G$ and then make a simple path in G_2 that goes through every vertex in the independent set. We continue this pattern until we get a clique. If our clique contains any of our added edges, remove them (but not the rest of the paths). We continue until the largest clique or independent set in the graph is a clique that contains no added edges. We then return this as our largest clique. (No larger clique than the one we find can ever exist because we find the largest possible K at each step.) It is left to show that this algorithm runs in polynomial time if a polynomial time algorithm for the given problem exists. Consider the following rough pseudocode:

```
Clique($G$)
  while true:
    while cliqueIndSet($G$,$K$) = false do:
      $K--$
    $G_1 = G$
    $kSet = \{\}$
    for each vertex $w$ in $G_1$ do:
      if cliqueIndSet($G_1 - w$,$K$) = true do:
        $G_1 = G_1 - w$
      else
        $kSet += w$
    if $kSet[1]$ adjacent to $kSet[2]$ do:
      if added edge in clique do:
        remove added edge
      else:
        return kSet
    else:
      create path in kSet in $G$
```

The first loop will terminate when the largest clique is reached and the method returns. Since it creates a path through the vertices of independent sets, it cuts the sizes of the independent sets in half every iteration. It runs log number of times (which is less than polynomial). Finding K takes at most v time, where v is the number of vertices. When we go through removing all possible vertices in G_1, we consider each vertex once. If a polyonmial time algorithm exists for cliqueIndSet (the given problem), then this loop runs in polynomial time. If we check if an edge has been added, we must check all added edges against each edge in the clique. At most v^2 edges have been added, and there are at most v

vertices in the clique, so this is $\leq v^3$. If we create a path in G, it takes $K < v$ time. Hence, this clique problem algorithm will run in polynomial time if a polynomial time algorithm exists for the problem defined above.

22. Consider the following problem. The input is a graph $G = (V, E)$, a subset R of vertices of G, and a positive integer k. The problem is to determine if there is a subset U of V such that

    - All the vertices in R are contained in U, and
    - the number of vertices in U is at most k, and
    - for every pair of vertices x and y in R, one can walk from x to y in G only traversing vertices that are in U

    Show that this problem is NP-hard using a reduction from Vertex Cover. Recall that the input for the vertex cover problem is a graph H and an integer l, and the problem is to determine whether H has a vertex cover of size l or not. A vertex cover S is a collection of vertices with the property that every edge is incident on at least one vertex in S.

    Solution: In order to show that this problem is NP-hard, we must show that Vertex Cover reduces to this problem, which we will call *subsetU*. In order to reduce this problem, we must convert the input of vertex cover to *subsetU* in such a way that *subsetU* will produce an output of 1 if and only if vertex cover were to produce a 1.

    The first thing we do is set create vertices for every edge in the Vertex Cover H. We will put these vertices in R, which is a subest of G. We will put all the actual vertices of H into G, and not R. The edges that form graph G are created like so: For each edge in H, connect the two vertices that are in G, to their corresponding edge vertex in R. The last thing we need to do is set k as the sum of all the edges + l. All that can be done in polynomial time.

    The way that *subsetU* is structured, all of R must be in U, or bilateral-ly: every edge in H must be in the set R. This means that $k - R = l$, or we only have l vertices left to choose for the graph of U, and the only thing left to choose are the vertices of H, since all the edges have been chosen. By definition of *subsetU*, it must be that all of R must somehow connect to eachother. We set it up so that there is no edge going from R to R, only R to G. This means that if every vertex in R is going to be connected, it must be through l vertices in G, or it is not possible. In terms of the vertex cover, this means that all edges must be satisfied using l vertices, which is the very definition of a vertex cover. This means that if there is a vertex cover in H using l vertices, then there is a *subsetU* using k vertices (where $k = R + l$). The converse is also true in the same way: There is only a vertex cover if there is a *subsetU*, due to the way we set the edges in G.

    The pseudocode would look something like this:

```
VertexCover(H, l):
    if(H.size ¡ l): return 0
    for(Vertex i in H)
        G.addVertex(i)
    for(Edge i in H)
        R.addVertex(i)
        G.addEdge(i.getConnection(0), r.vertex(i))
        G.addEdge(i.getConnection(1), r.vertex(i))
    k = l + R.size
    return subsetU(G,R,k)
```

1. Consider the problem of computing the AND of n bits.

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

  Solution: Consider the parallel algorithm computing and$(b_1, \ldots, b_n)$ defined by the recursive algorithm and$(b_1, \ldots, b_n)$ =AND(and$(b_1, \ldots, b_{n/2})$,and$(b_{n/2+1}, \ldots, b_n)$). Even though we have n processors, we only need to make use of $n/2$ (and using more will just take us longer). We assign each processor two of the input bits. Each processor computes the AND of its input bits. Then, in the next step, the output from each processor is ANDed with the output of some other processor (using half as many processors as the step before). This continues for $log\ n$ steps, each taking 1 time unit, until we have the desired quantity. The efficiency of this algorithm is $E(n,n) = \frac{n}{n*log\ n} = 1/log\ n$.

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

  Solution: Assign each of the $n/log\ n$ processors initially to two of the input bits. We then initially include $2n/log\ n$ of the input bits. Have each processor compute the AND of its two bits. Then, for half of those processors, assign each to its output bit and the output bit of some other processor (that is in the half we have excluded) so that every processor output is paired with some other processor output exactly once. Assign each one of the processors we have excluded to two more of the input bits. Now, we include an additional $n/log\ n$ of the original input bits. Again, we have each processor take the AND of its 2 input bits. We repeat until we have included all input bits, taking approximately $log\ n$ steps. Then, we take an additional $log\ n$ steps to compute the total AND exactly as in the solution above. The total time then is $O(log\ n)$. The efficiency of this algorithm is $E(n, log\ n) = \frac{n}{(n/log\ n)*log\ n} = 1$.

- Give an algorithm that runs in time $O(1)$ using n processors on a CRCW PRAM.

  Solution: One processor initially assigns some shared memory location to 1. Then assign each bit to a processor. Each processor reads its bit and if it is 0, it assigns 0 to a shared memory location. If it is 1, it does nothing. (Because 1 AND X = X.) Hence, this utilizes the CW capability of the CRCW machine. Since each processor works in parallel, the time is $O(1)$. The efficiency of this algorithm is $E(n,n) = \frac{n}{n*1} = 1$.