Solutions to Problem Set 9

1. (Sipser 8.25) An undirected graph is bipartite if its nodes can be divided into two sets such that all edges go from a node in one set to a node in the other set. Show that a graph is bipartite iff it does not contain a cycle that contains an odd number of nodes. Let $BIPARTITE = \{\langle G \rangle \mid G \text{ is a bipartite graph}\}$. Show that $BIPARTITE \in \mathbf{NL}$.
[25 points]

   Solution: Dividing the graph into two sets is the same as 2-coloring the graph such that all the edges are between vertices of different color. If the graph has an odd cycle, then the odd cycle in particular cannot be 2-colored and hence the graph cannot be bipartite. If the graph does not contain an odd cycle, we consider the DFS tree (or forest, if it is not connected) of the graph and color the alternate levels (say) red and blue. By construction, all the tree edges are between vertices of different color. Suppose an edge not in the tree connects two vertices of the same color. But these vertices must be connected by a path of even length in the tree and this extra edge will create an odd cycle, which is not possible. Hence, the above 2-coloring is a valid one which proves that the graph is bipartite.

   We show that $BIPARTITE \in \mathbf{coNL}$ or $\overline{BIPARTITE} \in \mathbf{NL}$, which suffices since $\mathbf{NL} = \mathbf{coNL}$. However, $\overline{BIPARTITE}$ is precisely the set of all graphs which contain an odd cycle. We guess a starting vertex v, guess an (odd) cycle length l and go for l steps from v, guessing the next vertex in the cycle at each step. If we come back to v (we can remember the starting vertex in logspace), we found a tour of odd length. Since any odd tour must contain an odd (simple) cycle, we accept and declare that the graph is non-bipartite.

2. (Sipser 8.23) Define $UCYCLE = \{\langle G \rangle \mid G \text{ is an undirected graph that contains a simple cycle}\}$. Show that $UCYCLE \in \mathbf{L}$. (Note: G may not be connected.)
[20 points]

   Hint: We can try to search the tree by always traversing the edges incident on a vertex in lexicographic order i.e. if we come in through the ith edge, we go out through the $(i+1)$th edge or the first edge if the degree is i. How does this algorithm behave on a tree? How about a graph with a cycle?

   Solution: Note that the above process performs a DFS on a tree and we always come back to a vertex through the edge we went out on. However, if the graph contains a cycle, there must exist at least one vertex u and at least one starting edge (u, v) such that if we start the traversal through (u, v), we will come back to u through an edge different that (u, v). Hence, we enumerate all the vertices and all the edges incident on them, and start a traversal through each one of them. If we come back to the starting vertex through an edge different than the one we started on, we declare that the graph contains a cycle. Since we can enumerate all vertices and edges in logspace and also remember the starting vertex and edge using logarithmic space, this algorithm shows $UCYCLE \in \mathbf{L}$.

3. We define the product of two $n \times n$ boolean matrices A and B as another $n \times n$ boolean matrix C such that $C_{ij} = \vee_{k=1}^{n} A_{ik} \wedge B_{kj}$. (We think of 0 as `false` and 1 as `true` for this problem.)

(a) Show that boolean matrix multiplication can be done in logarithmic space.

(b) Using repeated squaring, argue that A^p can be computed in space $O(\log n \log p)$.

(c) Show that if A is the adjacency matrix of a graph, then $(A^k)_{ij} = 1$ if and only if there is a path of length at most k from the vertex i to vertex j and is 0 otherwise.

(d) Use the above to give an alternative proof that $\mathbf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$.

[35 points]

SOLUTION:

(a) To compute C_{ij}, we maintain a counter t, read A_{it}, B_{tj}, compute $A_{it} \wedge B_{tj}$ and take an 'OR' with $\vee_{k=1}^{t-1} A_{ik} \wedge B_{kj}$ which we can keep stored. Since maintaining the counter takes logarithmic space and all boolean operations take constant space, we can compute C in logspace.

(b) To multiply k matrices, we generate the result entry by entry, by running a counter t and generating the itth entry in the product of the first $k-1$ and the tjth entry in the last matrix. Inductively, we need to maintain k counters which can be done in $O(k \log n)$ space. Finally, note that using repeated squaring, we can compute A^p using $O(\log p)$ matrices, which are different powers of A. To generate each of these matrices, we just need A and a single counter. Hence the total space needed is $O(\log p \log n)$.

(c) We argue this by induction. The case $k = 1$ is easy since $A_{ij} = 1$ iff there is an edge between i and j. Assuming it to be true for $k-1$, note that $A^k_{ij} = \vee_{t=1}^{n} A^{k-1}_{it} \wedge A_{tj}$, which is 1 iff for some t, $A^{k-1}_{it} = 1$ and $A_{tj} = 1$. However, this means there is a path of length $k-1$ from i to t and an edge from t to j, which gives a path of length k from i to j.

(d) Since PATH is **NL**-complete and we can check if there is a path (which can be of length at most n) between any two vertices in a graph by computing A^n in $O(\log^2 n)$ space, $\mathbf{NL} \subseteq \mathsf{SPACE}(\log^2 n)$.