**CS 180** Homework 8

**Problem 1**

We can prove this by first representing the knowing relationships with a bipartite graph. Since knowing is mutual, we do not have directed edges in this graph; we just construct an edge of capacity 1 from a boy to a girl if they know each other. Then we convert this problem into a network flow problem by adding a source node with edges of capacity 1 to every boy node. We then add a sink node with incoming edges of capacity 1 from every girl node. Running Ford Fulkerson to find the max flow would find a perfect matching in this graph. We could then just run FF k times, where each time, we only remove the edges from boy to girl in that round so we can have unique matchings.

This algorithm is also backed by Hall’s Theorem stating that a bipartite graph has a perfect matching if for any subset of the left side, S,the number of right side nodes S’s nodes connect to, N, is greater than |S|. Then since every boy knows k girls and vice versa, there must be k perfect matchings.

**Problem 2**

Going from a Hamiltonian Path to a Hamiltonian Cycle, we can add an extra node vn+1 that is connected to every other node in the graph. If the original graph has a Hamiltonian Path from node v0 to vn, then the new graph with this new node has a Hamiltonian Cycle from vn+1 to v0.

From a Hamiltonian Cycle we can arbitrarily choose a single edge (u, v) and append two additional nodes to it, u’ and v’. Vertice u’ would only connect to u and v’ similarly just to v. Then there is a Hamiltonian path if and only if this new graph has a Hamiltonian Cycle with the edge we first chose (u, v).

**Problem 3**

1. To solve this formula, we just need to make sure one clause is true because then the disjunction of all the clauses is true. To do this, we can traverse the clauses. If a clauses contains both a boolean and its negative, then it can’t satisfied. Or else, it can be satisfied by making all non-negated variables true and making all negated variables false. Doing this would run in linear time actually, which is a subset of a polynomial time algorithm.
2. Using the distributive law deceivingly sets us up for a polynomial time algorithm. Given a CNF problem with at most 3 literals per clause, converting it to a DNF problem would actually take exponential runtime, generating at most 3 resulting clauses for every clause in the CNF. For example (A ∨ B ∨ C) → (A) ∨ (B) ∨ (C) and (A ∨ B ∨ C) ∧ (D ∨ E ∨ F) → 9 clauses because of the 3 distinct literals in the first clause distributed among the other 3 distinct literals in the second clause. Even if some of the literals were not distinct, like if we had A and ~A, this would still give an upper bound or big O of 3C where C is the number of clauses.

**Problem 4**

Given an undirected graph G and a positive integer, we can derive a new directed graph from G by replacing every edge (a, b) of G with a pair of oppositely directed edges (a, b) and (b, a). This reduction to the new graph, G’, takes polynomial time. Then we prove the vertex cover of size k exists in G only if there is a subset of k nodes in G’ such that when those nodes are removed, so are all cycles. To start, we assume there is a vertex cover of k nodes in G. If we remove these k nodes and all incident edges from the new G’ graph, we get a graph with no cycles. This is because for every directed edge (a, b) in G’, either a or b or both were removed because at least one of the two must’ve been in the cover. Therefore, the resulting graph has vertices that share no edges between one another and so there are no cycles.

**Problem 4**