CS 6150: HW3 – NP Completeness, Intro to Graphs

HINTS/SOLUTIONS

1. Recall the CNF-satisfiability problem we saw in class, where we are given a formula ϕ, which is an AND of a bunch of clauses, and each clause is an OR of literals (x_i, $\overline{x_i}$, for variables $\{x_i\}_{i=1}^n$). We defined a 3-CNF formula as one in which every clause is the OR of precisely three literals. We showed that the 3-SAT problem, which asks to determine of a 3-CNF formula ϕ is satisfiable, is NP-complete.

   Let us now consider **2-CNF** formulas, in which every clause has precisely two literals. Interestingly, the satisfiability problem for 2-CNF formulas (called 2-SAT) has a polynomial time algorithm! [An example of a 2-CNF formula in 3 variables: $\phi = (x_1 \vee x_2) \wedge (\overline{x_1} \vee x_3) \wedge (x_2 \vee \overline{x_3})$.]

   (a) (2 points) Consider one clause of a 2-CNF formula, say $x_1 \vee x_2$. Prove that it is logically equivalent to the clause $(\overline{x_1} \implies x_2)$. [Those with questions about what this means, please post on the Canvas discussion forum.]

   (Easy to check, by noting that $a \implies b$ is true if either a is false, or b is true.)

   (b) (4 points) Clauses in the *implication* form naturally lead to a directed graph. Consider a graph in which the vertices are literals (thus for n variables there are $2n$ vertices, corresponding to x_i and $\overline{x_i}$, for $1 \leq i \leq n$). Now, for any clause of the form $l_i \implies l_j$, place a directed edge from literal l_i to literal l_j.

   Prove that a 2-CNF formula is satisfiable if and only if there is *no* variable x_i for which there is a path from x_i to $\overline{x_i}$ and a path from $\overline{x_i}$ to x_i. Use this to show that 2-SAT has a polynomial time algorithm.

   For concreteness, the directed graph we construct is the foll: for every $x_i \vee x_j$ in the 2-CNF formula, we have two edges, $(\overline{x_i}, x_j)$ and $(\overline{x_j}, x_i)$. Thus, if there is a path from a literal l_i to l_j, there is also a path from $\overline{l_j}$ to $\overline{l_i}$.
   First, suppose there exists an x_i for which there is a path from x_i to $\overline{x_i}$ and the other way around. If there is a satisfying assignment, then either x_i or $\overline{x_i}$ is true and the other is false, thus there exists a path that goes from TRUE to FALSE – a contradiction.
   Let us consider the other direction, i.e., there is no x_i as above. We need to prove that there exists a satisfying assignment. First, we look for paths from a literal to its negation. For all such paths (say l_i to $\overline{l_i}$), we are forced to set the l_i to be FALSE (and its negation to TRUE). This gives a partial assignment. For all literals that are TRUE, we claim that there cannot be any outgoing edges to unassigned vertices. This is because, if we suppose there is an edge from l_i that is TRUE to l_j that is unassigned, then there is an edge from $\overline{l_j}$ to $\overline{l_i}$, and there must have been a path from $\overline{l_i}$ to l_i (which is why l_i was set to TRUE), and thus there's a path from $\overline{l_j}$ to l_j, meaning that l_j was already assigned (to TRUE) – a contradiction.
   For the unassigned variables, we can make the assignment one by one. Make an arbitrary assignment for one, then assign to TRUE all the literals reachable from the TRUE literal. So also, assign to FALSE all the literals that are 'reverse reachable' from the FALSE literal. This does not lead to contradictions, because it can never happen that both l_j and $\overline{l_j}$ are reachable from l_i. (This would mean that l_i can reach $\overline{l_i}$, which we have assumed does not happen.)
   Running time: this algorithm runs in poly time (we simply need to do n DFS's, essentially computing the *transitive closure*.).

(c) (4 points) Let us now consider a new problem, which we call *2-or-more 3-SAT*. Here, we are given a 3-CNF formula ϕ, and an assignment is said to "2-or-more satisfy" a clause iff *at least two* of the literals in that clause are TRUE for that assignment. For example, the clause $(x_1 \vee x_2 \vee \overline{x_3})$ is "2-or-more satisfied" by the assignment T, F, F, but not by the assignment T, F, T. The problem is now, given a formula, find if there exists an assignment that "2-or-more satisfies" all the clauses.

Prove that *2-or-more 3-SAT* has a polynomial time algorithm.

We observe that $(x \vee y \vee z)$ is 2-or-more satisfied iff $(x \vee y) \wedge (y \vee z) \wedge (z \vee x)$ is satisfied. Thus, we can reduce 2-or-more 3-SAT to 2-SAT, which can be solved in polynomial time, using the algorithm from part (b).

2. (5 points) We mentioned in class that for purposes of complexity, it often helps to think of *decision* versions of problems. We now see a *justification* of this (such results are true for many other problems).

Consider the Independent-Set problem, in which the input is an undirected graph $G = (V, E)$ and a parameter k, and the goal is to determine if G has an independent set of size k. Suppose we have an oracle $\mathcal{O}$ for solving this decision version of independent set (think of it as a library function that takes input a graph G and k and answers YES/NO).

Prove that there exists an algorithm that can *find* an independent set of size k, if one exists, using a polynomial number of calls to the oracle $\mathcal{O}$, and possibly a polynomial amount of computation of its own.

First run the oracle on G, and ensure there exists an independent set of size k. Set $S = \emptyset$. Next, find a vertex v with the property that the graph obtained by removing v and all its neighbors (call this H) has an independent set of size $k-1$ (if the original graph has an independent set of size k, there must exist such a vertex). We can find such a vertex using n calls to the oracle. Add the vertex to S, and recurse on H, trying to find an independent set of size $k-1$.

The total number of oracle calls is at most $n + (n-1) + \cdots = O(n^2)$.

3. We will introduce some commonly arising problems in CS, and study their complexity. For all reductions, you need to reduce from one of the problems we have seen in class (SAT, 3-SAT, INDEPENDENT SET, VERTEX COVER, SUBSET SUM).

(a) (4 points) Consider the Integer Linear Programming (ILP) problem. We have n variables $x_1, x_2, \ldots, x_n$. An instance of ILP is defined by a set of *linear* constraints, i.e., constraints of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$, where a_i and b are integers. The problem is to determine if there exist **integers** x_i that satisfy all the constraints.

Prove that ILP is NP-hard *[Hint: give a reduction from SAT.]*

Consider some instance Φ of 3-SAT, and suppose it has variables $x_1, \ldots, x_n$, and clauses $\phi_1, \phi_2, \ldots, \phi_m$. Then, we consider the ILP, with variables y_i satisfying $0 \leq y_i \leq 1$, and additionally, for every clause ϕ, we add one constraint, as follows: say the clause is $x_1 \vee \overline{x_2} \vee x_3$, then the constraint we add is $y_1 + (1 - y_2) + y_3 \geq 1$. [I.e., for a negated variable, we have 1 minus the corresponding y variable on the LHS.]

A satisfying assignment to Φ clearly leads to feasible values for y_i in the ILP. It's also easy to see the opposite, because y_i and $(1 - y_i)$ are both $\in \{0, 1\}$, and thus if the sum of three such terms is ≥ 1, at least one must be ≥ 1, implying that the corresponding clause is satisfied.

(b) (1 point) Do you think ILP is NP-complete? [You are not expected to *answer* this. Give two-three lines of thought.]

Since it's NP hard, the question basically asks if ILP is in NP, i.e., is there a poly time "certificate" that a set of integer linear constraints has a feasible solution? This is equivalent to asking, if there is a feasible solution, can the 'bit length' be bounded polynomially in the input size?

This is a nontrivial question, which isn't fully understood.

(c) (2 points) Next, we will study equations over integers *modulo 2*, i.e., where the variables take 0/1 values, and we consider addition and multiplication (mod 2). First, consider linear systems modulo 2. Suppose we a collection of linear equations on n 0/1-variables x_i:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m. \end{aligned}$$

This set of linear equations is said to be *satisfiable* iff there is a 0/1 assignment to the variables x_i such that all the equalities are satisfied (mod 2). Let us call this problem LINEQ(mod2). Does LINEQ(mod2) have a polynomial time algorithm? [Again, answer YES/NO with a couple of lines justification.]

Yes, we can solve it via Gaussian elimination (which works over any field, in particular, integers mod 2).

(d) (8 points) Now, let us study **quadratic** equations (mod 2). Here, we have n variables as above, but the equations now are quadratic. I.e., each equation has the form $\sum_{i,j} c_{ij}x_ix_j = b$, where c_{ij} and b are 0/1 (and addition/multiplication are mod 2). For instance, consider the equation $x_1x_2 + x_3^2 + x_3x_1 = 1$. This is satisfied by the assignment $(1,1,1)$ and $(1,1,0)$, but not by $(1,0,1)$. The QUADEQ(mod2) problem is to determine, given quadratic equations as above, if there is a 0/1 assignment such that all the equations are satisfied. Prove that QUADEQ(mod 2) is NP-complete. *[Hint: for the hardness, reduce from 3-SAT. As an intermediate step, you may want to prove that determining satisfiability of "cubic" equations (mod 2) is NP-hard; you get partial credit for this.]*

Suppose we have an instance Φ of 3-SAT, with variables $x_1, \ldots, x_n$, and clauses $\phi_1, \ldots, \phi_m$. Let us consider a system of quadratic equations in variables $y_1, \ldots, y_n$, and additional variables z_{ij}, for $1 \le i, j \le n$ (explained below). Consider some clause $\phi = (x_1 \vee \overline{x_2} \vee x_3)$. Corresponding to it, we write the equation:

$$(1 - y_1)y_2(1 - y_3) = 0.$$

This is satisfied iff either $1 - y_1 = 0$ or $y_2 = 0$, or $1 - y_3 = 0$, which is equivalent to ϕ being satisfiable. (This can be formalized in a natural way.)
The only catch is that we have obtained a *cubic* equation. This can easily be made quadratic by introducing new variables z_{ij}, and adding the constraint $z_{ij} = y_iy_j$, and using z_{ij} instead of the product.
Note: One of the students gave a "nicer" solution than the above, by introducing new variables c_{i1}, c_{i2}, c_{i3} for every clause i, and adding the equation $c_{i1}y_1 + c_{i2}(1-y_2) + c_{i3}y_3 = 1$. If $y_1 = 1 - y_2 = y_3 = 0$, this equation cannot be satisfied no matter what the $c_{i1}, \ldots$ are! The converse is also easy to check.

4. (a) (3 points) Let G be a graph in which all vertices have degree ≥ 2. Show that G has a cycle. Argue from first principles.

Start with any vertex, take one of the edges out of it, mark that edge. At the next vertex, take any unmarked edge out, mark that edge, and keep going. Since the degree is ≥ 2, if we reach a vertex that has never been reached, there is always a 'way out'. Eventually, we have to run out of vertices, and thus we reach a vertex that has already been reached. This implies that we have found a cycle.

(b) (2 points) Does there exist an undirected graph on 10 nodes with degrees $2, 3, 4, 4, 7, 1, 4, 5, 3, 2$ respectively?

No. The sum of the degrees is precisely twice the number of edges. In this case, the degrees add up to an odd number.

(c) (5 points) Recall that a directed graph $G = (V, E)$ is said to be strongly connected if for every $u, v \in V$, there is a directed path from u to v and vice versa. Given such a graph G, give an algorithm that finds a cycle of odd length, or answers that such a cycle does not exist. Your algorithm should run in time $O(|E| + |V|)$.

Start with any vertex u, color it red. Now we start a BFS from u, coloring vertices in alternately blue and red (based on the distance from u). Since G is strongly connected, for any starting vertex u, we end up coloring all the vertices. Now, check if there exists an edge ij both of whose end points are of the same color. If there isn't then we have found a two-coloring of the graph, and thus there is no odd cycle.
Thus, suppose there is an edge (i, j). We have two cases: either i and j are both red, or both blue. First, say they are red. Now, do a BFS from j. Again, the strong connectivity implies that we can reach all the vertices. Now, the path from j to u (the initial start vertex) must be even, or else we have found an odd cycle. In this case, the path from $j \to u \to i$ is of even length, and thus, together with the edge (i, j), we have an odd cycle. A similar argument can be made in the case that both i, j are blue (in this case, the path from $j \to u$ must be odd). [The mild technicality above is that an edge may be repeated. One can argue that the cycle then can be decomposed into two cycles, at least one of which has odd size, and we may potentially have to recurse.]
Overall, we did precisely 2 BFS's, and thus the run time is $O(m + n)$.

5. (10 points) Consider the common conundrum of a frequent traveler: you want to travel from place A to place B sometime during the week. Suppose you are given departure and arrival times of every flight in the world, during that week. The sole goal of the traveler is to minimize the *total travel time*, i.e., flying time plus time spent at intermediate airports. Suppose that for every connection, there must be a 10 minute gap between the scheduled arrival into that airport and the scheduled departure from that airport (time taken to connect).

   Suppose there are n airports total, and m flights, give a polynomial time algorithm to find the schedule with the smallest total travel time (as defined above). Give an analysis of the running time of your algorithm. *[Hint: construct an appropriate graph and run Dijkstra's algorithm.]*
   [Source: Jeff Erickson's Exercises]

   The following example will illustrate the solution. Suppose cities are labeled $A, B, \ldots$. Each flight is a 4-tuple (origin, destination, start time, arrive time). We create a graph in which every vertex is a tuple (city, time, 'type'). The 'type' can either be 0 or 1 (labels correspond to departure and arrival resp.). If there is a flight departing at city X at time t, we have a vertex $(X, t, 0)$, and if there's a flight arriving at city X at time t, we have a vertex $(X, t, 1)$. The edges in the graph are the following: for every flight (X, Y, t_1, t_2), we have an edge from $(X, t_1, 0)$ to $(Y, t_2, 1)$, of weight $t_2 - t_1$. Further, as long as $t' - t \geq 10$ (minutes), we have an edge from $(X, t, 1)$ to $(X, t', 0)$ of weight $t' - t$ (this corresponds to the transit time condition).

   Finally, we have two special nodes u, v. u has an edge to all nodes of the form $(A, t, 0)$, for some t (fixed A, the intended origin), with weight 0. t has an edge *from* all nodes of the form $(B, t, 1)$, for different t, again with weight 0.

   The shortest $u - v$ path gives the desired time.

   The size of the graph is clearly linear in the input size (input consists of the list of flights). Thus the running time is $O(m \log m)$.