COSC 311: ALGORITHMS
HW5: NP-COMPLETENESS

Due **Monday**, December 10 in class

1 Submission and Collaboration

This assignment is due on Monday, December 10 at the start of class; bring a hard copy of your typed responses to submit in class. **Each problem must be on a separate sheet of paper, with your name and section number at the top of the page.** If you use more than one sheet of paper for an individual problem, please staple together those pages. There is no need for you to rewrite the problems on your submission.

In accordance with this course's intellectual responsibility policy, you are welcome to discuss the problems with other students who are currently taking the course, but you must write up your solutions individually. Please note at the top of your submission with whom you discussed each problem.

2 Problems

Problem 1: Generalizing Independent Set.

The *Set Packing* problem is defined as follows:

Input:

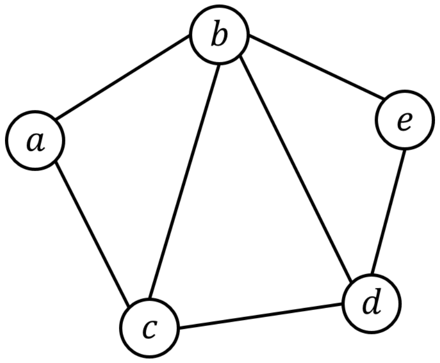
- A set U consisting of n elements
- m subsets of U, $S_1, \ldots, S_m$
- A positive integer k

Output: Is there a collection of at least k subsets such that no two subsets intersect?

The Set Packing problem can be seen as a generalization of Independent Set. In the Independent Set problem, we want to find a subset of nodes such that none of the nodes in our set are connected by an edge. Set Packing doesn't make any assumptions that the elements are organized into a graph structure or that conflicts between elements are explicitly encoded as edges. Nonetheless, both problems involve choosing a conflict-free set of items.

a) Give a polynomial-time reduction from Independent Set to Set Packing. What is the runtime of your reduction?

b) Show how your reduction from part (a) turns the following instance of Independent Set, with $k = 2$, into an instance of Set Packing:



Problem 2: A hard scheduling problem.
In the Interval Scheduling problem, we had a set of jobs, each with a specified start time and finish time, and a single processor; our goal was to schedule as many jobs as possible without any conflicts. We saw in class that the Interval Scheduling problem can be solved in polynomial time using a greedy algorithm that always chooses the job with the earliest finish time.

Suppose that instead of requiring a single interval of processing time, each job instead specifies a *set* of intervals during which it requires the processor. For example, job j might require the processor from 9-10am and from 1-2pm. As before, we have a single processor and we want to schedule as many jobs as possible without conflicts. If you schedule job j, you must give it *all* time intervals that it requires. Hence you cannot schedule any other job from 9-10 or from 1-2, but you could still schedule other jobs at any other time, including the interval from 10am-1pm.

Specifically, the *Multiple Interval Scheduling* problem is defined as follows:

Input:

- A set J of jobs, where $|J| = n$
- For each job $j \in J$, a set of time intervals during which job j requires the processor
- A positive integer k

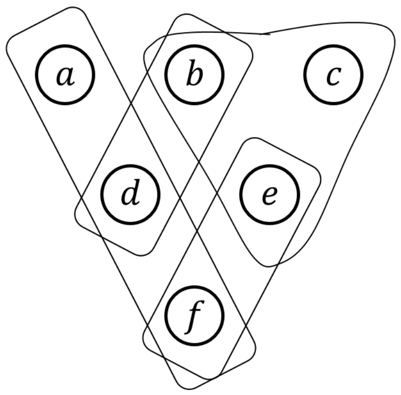
Output: Is it possible to schedule at least k jobs such that no two jobs conflict?

Your job is to show that Multiple Interval Scheduling is NP-complete. Specifically:

a) Show that Multiple Interval Scheduling is in NP.

b) Give a polynomial-time reduction from Set Packing to Multiple Interval Scheduling (as part of your reduction, you should explain how you know that "yes" instances of Set Packing map to "yes" instances of Multiple Interval Scheduling, and vice versa). What is the runtime of your reduction?

c) Show how your reduction turns the following instance of Set Packing, with $k = 3$, into an instance of Multiple Interval Scheduling:



Problem 3: Not-All-Equal 3-SAT.
It turns out that there are many variants on boolean satisfiability problems. We initially decided to work with 3-SAT instead of SAT because its additional structure (requiring each clause to have three terms instead of any number of terms) was slightly easier to work with. In some cases, it helps to consider even further variants on the standard satisfiability problem. One such variant is *Not-All-Equal 3-SAT*. As you might expect based on its name, in Not-All-Equal 3-SAT we are trying to satisfy a set of clauses, each of length 3, where at least one term in each clause must be set to false. Specifically, we have:

Input:

- A set of variables $X = \{x_1, \ldots, x_\ell\}$
- A set of clauses $C_1, \ldots, C_k$, each of length 3

Output: Is there a satisfying assignment such that for each clause, the term assignments are not all equal (i.e., at least one term is set to false)?

For example, consider the conjunction

$$(x_1 \vee \bar{x_2} \vee x_3) \wedge (x_1 \vee x_4 \vee \bar{x_5}).$$

The assignment $x_1 = x_3 = x_4 = x_5 =$ `true`, $x_2 =$ `false` is not a NAE satisfying assignment because in the first clause, all three terms x_1, $\bar{x_2}$, and x_3 are set to true. The assignment $x_3 = x_4 = x_5 =$ `true`, $x_1 = x_2 =$ `false` is a NAE satisfying assignment.

We won't show this on this homework assignment, but it turns out that Not-All-Equal 3-SAT is NP-complete.

Now considering a seemingly unrelated problem: that of finding cuts in a graph. We know how to solve the Min Cut problem efficiently using the Max-Flow Min-Cut Theorem. But what about the flip side? In the *Max Cut* problem, our goal is to find a cut in a graph that has the *highest* capacity possible. We define the k-Cut problem (which is the decision version of Max Cut) as follows:

Input:

- An undirected, weighted graph $G = (V, E)$, where $|V| = n$ and $|E| = m$
- For each edge $e \in E$, a positive integer capacity $c(e)$
- A positive integer k

Output: Is there a cut of G that has capacity at least k?

a) The k-Cut problem is a decision problem, but originally we were interested in the optimization problem Max Cut, in which we're looking for the maximum capacity of any cut in the graph. Suppose we had an algorithm $A(G, k)$ that could solve k-Cut in polynomial time. How could we use A to solve Max Cut in polynomial time in n and m? (You may assume that edge capacities are polynomial in n and m.)

b) Unfortunately, we are unlikely to ever find a polynomial time algorithm to solve k-Cut. Use the fact that NAE-3-SAT is NP-complete to prove that k-Cut is NP-complete.