COSC 311: ALGORITHMS
HW5: NP-COMPLETENESS
Solutions

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, $S_1, \dots, S_m$
- 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?

Solution: Our goal is to take an arbitrary instance of Independent Set and take polynomial time to turn it into an instance of Set Packing. We start with our input to Independent Set: an undirected, unweighted graph $G = (V, E)$ and a positive integer k. Our reduction is as follows:

- For every edge $(u, v) \in E$, create one element $x_{u,v}$ and add it to U.
- For every vertex $v \in V$, create one subset $S_v = \{x_{u,v} \in U : (u, v) \in E\}$. That is, we create a subset for vertex v that includes all of the elements in U corresponding to edges incident to v in the original graph.
- Set k in our Set Packing instance to be equal to k in our Independent Set instance.

We need to show that "yes" instances of Independent Set map to "yes" instances of Set Packing, and "yes" instances of Set Packing map to "yes" instances of Independent Set.

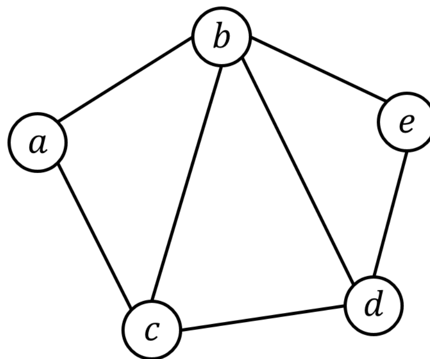
1) Suppose we start with an instance of Independent Set in which it is possible to find a set S of at least k vertices that are all independent. Then for every vertex $v \in S$, choose the corresponding subset S_v in our set packing instance. Consider any two subsets S_u and S_v that were chosen in this way, and suppose that subsets S_u and S_v have an element in common. Then there must have been an edge between u and v in the original input graph to independent set. But this contradicts the

fact that u and v were part of set S, and therefore independent. Hence S_u and S_v have no elements in common, and so the collection of subsets $\{S_v : v \in S\}$ is a collection of at least k subsets such that no two subsets intersect.

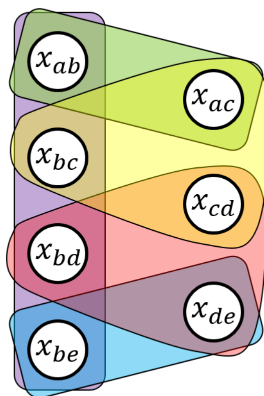
2) Suppose our instance of Set Packing contains a collection C of at least k subsets such that no two subsets intersect. Then consider nodes u, v such that $S_u, S_v \in C$. Suppose there is an edge between u and v in our original graph. Then element $x_{u,v}$ must exist in our Set Packing instance, and is included in both S_u and S_v. But this contradicts the fact that $S_u, S_v \in C$, since their inclusion in the Set Packing solution means that they have no elements in common. Hence edge (u, v) must not exist. This tells us that $\{v : S_v \in C\}$ forms an independent set of at least k nodes in the original graph.

Our reduction involves creating one element for each edge in our graph (of which there are m) and one subset for each node in our graph (of which there are n). We can do this in time $O(n + m)$.

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



Solution: We end up with the following instance of Set Packing, with $k = 2$. There is a set packing of size 2 in this instance: for example, we could choose the yellow set and the blue set. This corresponds to an independent set of size 2 in our original graph: nodes c and e.



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.

Solution: Our certificate is a subset S of k jobs. Given the original set J of jobs and a proposed solution S, it is easy to check in time $O(k^2)$ whether any jobs in the set S conflict by running through all pairs of jobs, and for each pair of jobs running through all pairs of intervals. Since $k \leq n$, this takes time $O(n^2)$ (assuming that the number of intervals per job is upper bounded by a constant; if we remove this assumption and simply say that the number of possible time intervals is some number m, then checking for conflicts in this way takes time $O((nm)^2)$). Since we can define a polynomial time verifier for Multiple Interval Scheduling, we know 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?

Solution: We create our instance of Multiple Interval Scheduling as follows:

- For each element $u \in U$, create a new one-hour time interval that does not overlap with any previously created intervals (i.e., map the first element of U to 9-10am, map the second element of U to 10-11am, etc).

- For each subset S_j in the Set Packing instance, create a job j that requires the time intervals corresponding to the elements included in S_j.
- Set k in the Multiple Interval Scheduling instance equal to k in the Set Packing instance.

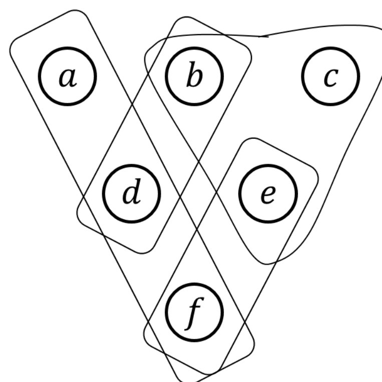
We need to show that "yes" instances of Set Packing map to "yes" instances of Multiple Interval Scheduling, and vice versa.

1) Suppose we start with an instance of Set Packing that contains a collection C of at least k subsets such that no two subsets intersect. Then we can construct a schedule of at least k non-conflicting jobs by including job j in our schedule if subset S_j is in C. To see why this is a valid schedule, assume that two jobs i and j chosen in this way have a conflict. Then there is some time interval that is used by both jobs i and j; this time interval corresponds to an element in U that must therefore be included in both S_i and S_j. But this contradicts the fact that S_i and S_j were both in C. Hence jobs i and j do not conflict. Since there were at least k subsets in our Set Packing solution, there are at least k jobs in our schedule. So we have a "yes" instance of Multiple Interval Scheduling.

2) Suppose our instance of Multiple Interval Scheduling has a valid schedule with at least k jobs. Then we can construct a solution to our Set Packing instance by including subset S_j in our collection of subsets if job j is included in our schedule. To see why this is a valid solution to Set Packing, suppose that two subsets S_j and S_i chosen in this way have an element in common. This element corresponds to a time interval that is required by both job i and job j. But this contradicts the fact that i and j were both part of a valid schedule. Hence subsets S_j and S_i must not have any elements in common. Since there were at least k jobs included in our schedule, there are at least k subsets included in our solution to Set Packing. So we have a "yes" instance of Set Packing.

Our reduction requires us to create one time interval for every element in U, of which there are n, and one job for every subset in our Set Packing instance, of which there are m. For each job, we then run through all n time intervals and add the intervals corresponding to the elements in that job's corresponding subset. This takes time $O(mn)$. Hence our reduction takes time $O(mn)$.

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



Solution: We map our instance of Set Packing to an instance of Multiple Interval Scheduling as follows. Create time intervals:

Element	Time Interval
a	9-10am
b	10-11am
c	11am-12pm
d	12-1pm
e	1-2pm
f	2-3pm

And create jobs:

Job	Required intervals
1	9-10am, 12-1pm, 2-3pm
2	10-11am, 12-1pm
3	10am-12pm, 1-2pm
4	1-3pm

We are looking for a valid schedule of at least $k = 3$ jobs.

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.)

Solution: Let $k_{\max} = \sum_e c(e)$. We can repeatedly call A with different values of k, binary searching between $k = 0$ and $k = k_{\max}$. This will require $O(\log k_{\max})$ calls to A; $\log k_{\max}$ is polynomial in n and m since all edge capacities are polynomial in n and m. So the overall runtime of this algorithm is $O(f_A(n, m) \log k_{\max})$, where $f_A(n, m)$ is the runtime of algorithm $A(G, k)$. Assuming $f_A(n, m)$ is polynomial in n and m, the overall runtime is polynomial.

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.

Solution: We first need to prove that k-Cut is in NP. Our certificate will be a partition of the vertices in E. Given an instance of k-Cut and a proposed solution, we can run through all of the edges in G, add up the capacities of the edges that cross the cut, and check whether the total capacity of the cut is greater than k. This takes time $O(m)$. Since we have defined a polynomial time verifier for k-Cut, we know that k-Cut is in NP.

Next we need to show that all problems in NP can be reduced to k-Cut. We will do this by taking a problem that is already known to be NP-complete, namely NAE-3-SAT, and reducing it to k-Cut. Our goal is to take an instance of NAE-3-SAT and turn it into an instance of k-Cut. We will create two nodes for each variable x_i in our NAE-3-SAT instance: one corresponding to x_i and the other corresponding to $\bar{x}_i$. We connect these two nodes by an edge. We also create a "triangle" of edges

in our k-Cut instance for each clause in our NAE-3-SAT instance.

The trick will be to set capacities on our edges so that we can translate a sufficiently high-weight cut in our graph into a NAE-3-SAT truth assignment. We can imagine the cut partitioning our nodes so that the nodes on one side correspond to terms set to true, and nodes on the other side correspond to terms set to false. In order for us to end up with a satisfying assignment, two properties have to be true about our resulting cut:

1. Nodes x_i and $\bar{x}_i$ must be on opposite sides of the cut.
2. In every "clause" triangle, there must be at least one node on each side of the cut.

To accomplish property (1), we must set the capacity on edge $(x_i, \bar{x}_i)$ high enough that this edge must be included in any sufficiently high-valued cut. In order to accomplish property (2), exactly two edges in each "clause" triangle must cross the cut.

We will define our reduction as follows:

- For each variable x_i in the NAE-3-SAT instance, create nodes labeled x_i and $\bar{x}_i$.
- For each variable x_i in the NAE-3-SAT instance, add edge $(x_i, \bar{x}_i)$ with capacity $10m$.
- For each clause (x_i, x_j, x_k) in the NAE-3-SAT instance, add edges (x_i, x_j), (x_i, x_k), and (x_j, x_k), each with capacity 1.
- Set $k = 10nm + 2m$.

We now need to show that "yes" instances of NAE-3-SAT map to "yes" instances of k-Cut, and vice versa.

1) We want to show that if our instance of NAE-3-SAT is satisfiable, then our graph contains a cut of value at least k. Take a satisfying assignment to the NAE-3-SAT instance, and consider the cut in which all terms set to true are on one side, and all terms set to false are on the other. All of the edges $(x_i, \bar{x}_i)$ cross this cut, contributing $10mn$ to the value of the cut. And since we have a satisfying instance of NAE-3-SAT, two edges from each "clause" triangle cross the cut, contributing $2m$ to the value of the cut. Hence the total value of the cut is $10mn + 2m$, so we have a "yes" instance of k-Cut.

2) We want to show that if our instance of k-Cut has a cut of value at least k, then our instance of NAE-3-SAT is satisfiable. We first will show that for all i, nodes x_i and $\bar{x}_i$ must be on opposite sides of the cut. Suppose they were not. Then an upper bound on the value of the cut is the sum of all other edge capacities in the graph: this is $10m(n-1) + 3m = 10mn - 7m < 10mn + 2m$, which contradicts the fact that the graph has a cut of value at least $k = 10mn + 2m$. Hence x_i and $\bar{x}_i$ are on opposite sides of the cut. This means that our cut corresponds to a truth assignment to the variables. Next we need to show that two edges from each "clause" triangle cross the cut. Again, suppose not. Then an upper bound on the value of the cut is $10mn + 2(m-1) + 1 = 10mn + 2m - 1 < 10mn + 2m = k$, which again contradicts the fact that the graph has a cut of value at least k. Hence there must be exactly two edges from each "clause" triangle crossing the

cut (note that all three edges in a "clause" triangle cannot cross the cut). This means that within each clause, there is at least one term on each side of the cut. Hence we have fulfilled the NAE requirement, and we have a "yes" instance of NAE-3-SAT.

We created 2 nodes for each variable and one edge between these nodes: $O(\ell)$. We then created three edges for each clause: $O(k)$. The total runtime for our reduction is $O(\ell + k)$, so we have a polynomial time reduction.