COMS20010 — Problem sheet 8

You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and videos instead. (Then come back to the sheet when you need to revise for the exam!) I'll make solutions available shortly after the problem class. As with the Blackboard quizzes, question difficulty is denoted as follows:

- $\star$ You'll need to understand facts from the lecture notes.
- $\star\star$ You'll need to understand and apply facts and methods from the lecture notes in unfamiliar situations.
- $\star\star\star$ You'll need to understand and apply facts and methods from the lecture notes and also move a bit beyond them, e.g. by seeing how to modify an algorithm.
- $\star\star\star\star$ You'll need to understand and apply facts and methods from the lecture notes in a way that requires significant creativity. You should expect to spend at least 5–10 minutes thinking about the question before you see how to answer it, and maybe far longer. Only 20% of marks in the exam will be from questions set at this level.
- $\star\star\star\star\star$ These questions are harder than anything that will appear on the exam, and are intended for the strongest students to test themselves. It's worth trying them if you're interested in doing an algorithms-based project next year — whether you manage them or not, if you enjoy thinking about them then it would be a good fit.

If you think there is an error in the sheet, please post to the unit Team — if you're right then it's much better for everyone to know about the problem, and if you're wrong then there are probably other people confused about the same thing.

This problem sheet covers week 9, focusing on network flows and NP-hardness.

1. [$\star\star$] In lectures, we showed how to use edge capacities to simulate vertex capacities in flow networks. Explain how to do this the other way round, using vertex capacities to simulate edge capacities. In other words, given a flow network (G, c_E, s, t) with source s, sink t and edge capacity function c_E, explain how to construct a network (G', c_V, s', t') with a vertex capacity function c_V such that flows in (G, c_E, s, t) correspond to flows in (G', c_V, s', t') of the same value, and vice versa.

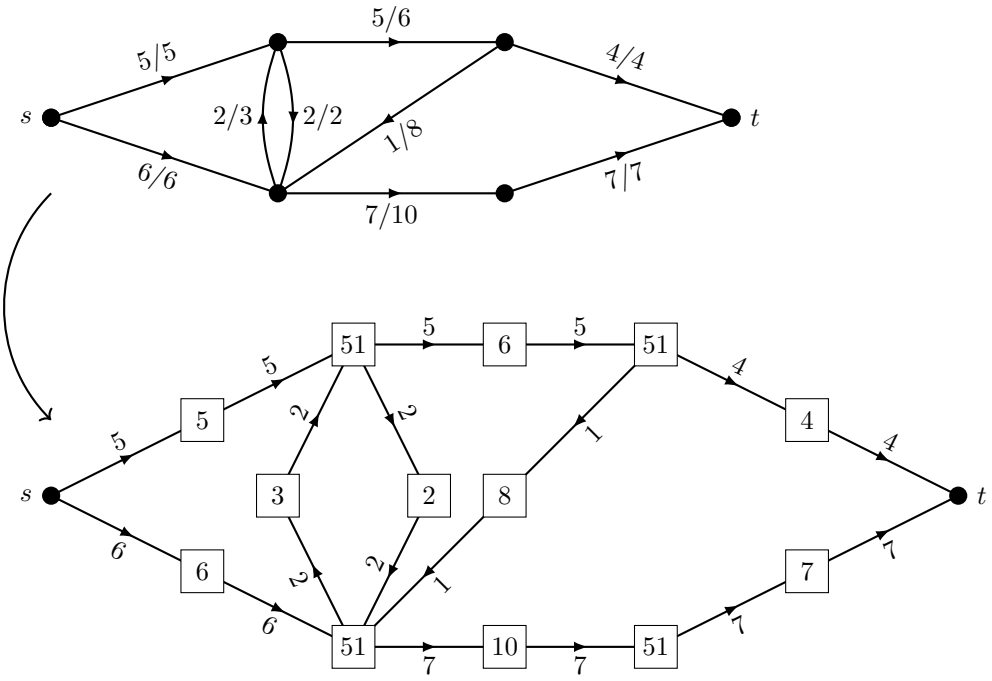
Solution: To simulate vertex capacities using edge capacities in lectures, we introduced a vertex gadget; here, to simulate edge capacities using vertex capacities, we will use an edge gadget. Let (G, c_E, s, t) be as in the question, and form (G', c_V, s', t') from (G, c_E, s, t) as follows:

- For each edge $e = (u, v)$ in G, add a new vertex x_e and replace (u, v) by the pair of edges (u, x_e) and (x_e, v). (This operation is known as *subdividing* e.)
- Assign each new vertex x_e capacity $c(e)$, and assign each vertex in $V(G) \setminus \{s, t\}$ capacity $\sum_{e \in E(G)} c(e)$.
- Take $s' = s$ and $t' = t$.

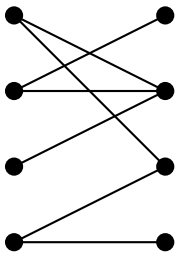
Flows in (G, c_E, s, t) correspond bijectively to equal-valued flows in (G', c_V, s', t'); if f is a flow in (G, c_E, s, t), then for all edges $e = (u, v) \in E(G)$, the corresponding flow f_V in (G', c_V, s', t') maps both (u, x_e) and (x_e, v) to $f(e)$. To see that f_V a flow if and only if f is, we note that:

- The flow f_V is conserved at all vertices x_e by construction;
- The flow f_V is conserved at a vertex $v \in V(G)$ if and only if it is conserved at v in f;
- The flow f_V obeys vertex capacity constraints at all vertices in $V(G)$ by construction;
- The flow f_V obeys the capacity constraint at a vertex x_e if and only if f obeys the capacity constraint at the edge e.

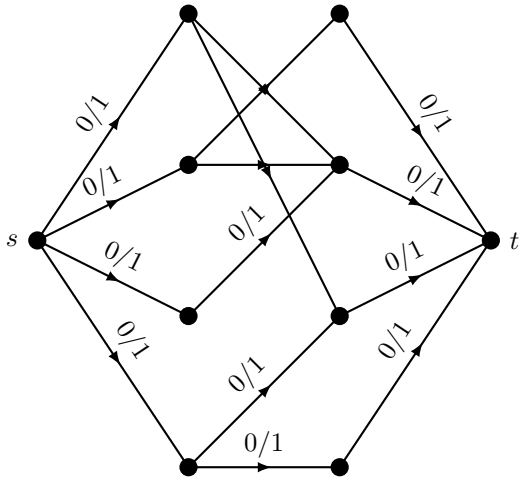
f_V and f have the same values, so the construction is valid. An example is shown below.



2. (a) [⋆⋆] Turn this (bipartite) matching problem into a flow network problem using the reduction described in lectures.



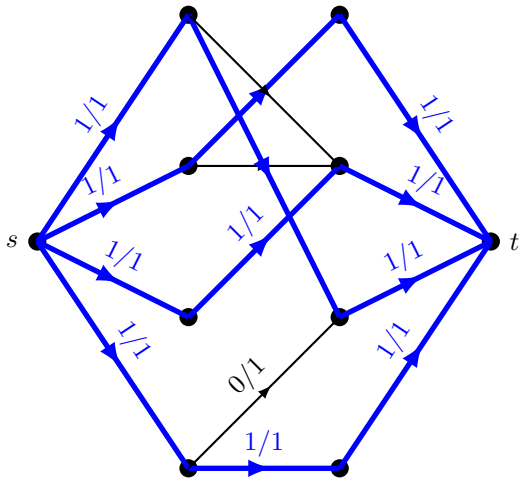
Solution: All we do here is add a new source and target vertex, and then make all the edges directed with capacity 1.



Note that there are a couple of edges drawn here without a 0/1 label. This is just because they don't fit on the image, but should be there.

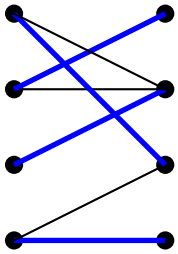
(b) [⋆⋆] Use Ford-Fulkerson to calculate a maximum flow for your flow network from part (a).

Solution: A maximum flow here is



(c) [⋆⋆] How does this maximum flow give you a maximum matching for the earlier bipartite graph?

Solution: We can take the edges which are part of the maximum flow and also part of our bipartite graph to extract a maximum matching.



(d) [⋆⋆⋆] Using the max-flow min-cut theorem, prove that if G is a bipartite graph with bipartition

(A, B), and $|A| = |B|$, then G contains a perfect matching if and only if and for all $X \subseteq A$ we have $|N(X)| \geq |X|$. (**Hint:** Remember the reduction given in lectures from finding a maximum matching to finding a maximum flow.)

Solution: If G contains a perfect matching, then each set $X \subseteq A$ is matched to at $|X|$ distinct vertices, so $|N(X)| \geq |X|$. Conversely, suppose $|N(X)| \geq |X|$ for all $X \subseteq A$; then we must show that G contains a perfect matching.

In lectures, we turned G into a flow network by:

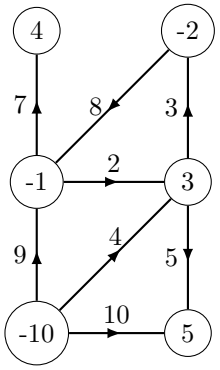
- adding a source vertex s and a sink vertex t;
- directing all edges from A to B;
- adding all possible edges from s to A;
- adding all possible edges from B to t;
- and giving every edge capacity 1.

Let (X, Y) be a cut in this network, so that $s \in X$ and $t \in Y$. Then the capacity of (X, Y) is given by the number of edges leaving X from s plus the number of edges leaving X from A plus the number of edges leaving X from B, so

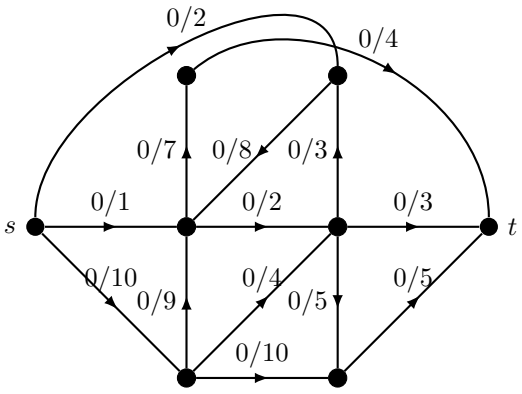
$$\begin{aligned} c^+(X) &= |A \setminus X| + |E(A \cap X, B \setminus X)| + |B \cap X| \\ &\geq |A \setminus X| + |N_G(A \cap X)| - |B \cap X| + |B \cap X|. \end{aligned}$$

By hypothesis we have $|N_G(A \cap X)| \geq |A \cap X|$, so it follows that $c^+(X) \geq |A|$. By the max-flow min-cut theorem, it follows that there is a flow with value at least $|A|$. By the result proved at the end of lecture 23, it follows there is an **integer-valued** flow with value at least $|A|$, which corresponds to a matching with at least $|A|$ edges as shown in lecture 24. Since $|A| = |B|$, this must be a perfect matching as required.

3. (a) [⋆⋆] Reduce this circulation problem to a maximum flow problem using the reduction in lectures.

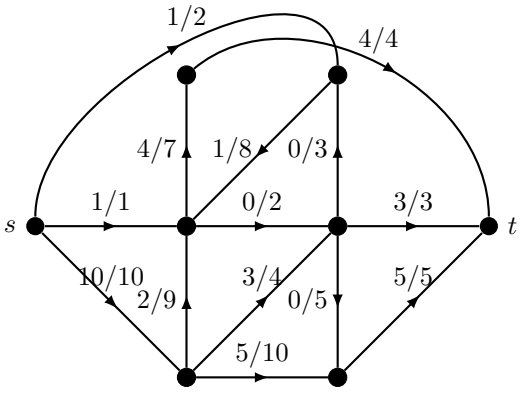


Solution: First, we observe that $S = \{sources\} = \{a, b, f\}$ and $T = \{sinks\} = \{c, d, e\}$. Next, we construct new vertices s, t and connect s to each of our sources, and t to each of our sinks. Finally, we give each edge (s, v) and (v, t) a capacity of $D(v)$. This gives us the following flow network:



(b) [⋆⋆] Use Ford-Fulkerson to find a maximum flow in your reduction problem.

Solution: A maximum flow here looks like



(c) [⋆⋆] Explain how this shows us there is no valid circulation in this network.

Solution: We see that this network has one more unit coming from a source than going out of a sink. I.e. there are 13 units of flow that could come from a source, and 12 units of flow that could go out of a sink. It follows that it is not possible for this flow, or any other flow, to maximise both sources and sinks due to this mismatch.

4. [⋆⋆] In lectures, we said that every propositional logic formula (such as $\neg x \wedge \neg((y \vee z) \wedge \neg z)$) can be expressed in conjunctive normal form. In other words, for any formula F, there is a CNF formula F' on the same variables which evaluates to `True` if and only if F does. Give an exponential-time algorithm to find such a formula. (**Hint:** Try enumerating the assignments on which F is false.)

Solution: Let F be the input formula. Let $x_1, \ldots, x_n$ be the variables of F, and let $a_1, \ldots, a_t \colon \{x_1, \ldots, x_n\} \to \{\texttt{True}, \texttt{False}\}$ be a list of all assignments of values to those variables under which F is `False`. (We can find $a_1, \ldots, a_t$ in $O(|F| \cdot 2^n)$ time by brute-force checking.) Each assignment a_i corresponds to an AND clause C_i of n literals; for example, the assignment $x \to \texttt{True}$, $y \to \texttt{False}$ and $z \to \texttt{True}$

corresponds to $x \wedge \neg y \wedge z$. The AND clause C_i evaluates to `True` if and only if the values of $x_1, \ldots, x_n$ are chosen as in a_i. We therefore have

$$F = \neg(C_1 \vee C_2 \vee \cdots \vee C_t).$$

In other words, F is `True` if and only if $x_1, \ldots, x_n$ aren't set according to any of the assignments $a_1, \ldots, a_t$.

We now apply De Morgan's laws to turn F into CNF form. First observe that

$$F = \neg C_1 \wedge \neg C_2 \wedge \ldots \neg C_t.$$

Moreover, write $C_i = \ell_1 \wedge \ell_2 \wedge \ldots \ell_n$, where each ℓ_i is a literal. Write $C_i' = \neg\ell_1 \vee \neg\ell_2 \vee \cdots \vee \neg\ell_n$; then again by De Morgan's laws, $\neg C_i = C_i'$. Thus

$$F = C_1' \wedge C_2' \wedge \cdots \wedge C_t'.$$

Each C_i' is an OR clause, so this is in conjunctive normal form. Our algorithm therefore simply enumerates $a_1, \ldots, a_t$, calculates $C_1', \ldots, C_t'$, and outputs the above formula.

5. ⋆⋆⋆ In lectures, we said that search problems are often exactly as hard as their equivalent decision problems. This question gives two examples.

   (a) Give a Cook reduction from the problem of finding a satisfying assignment in a CNF to SAT, which asks whether a satisfying assignment exists. (**Hint:** Try coming up with an exponential-time recursive algorithm, then using the oracle to speed it up.)

   Solution: Suppose we are given a polynomial-time oracle for SAT — let's call it DecideSAT. Let F be a given CNF formula, with variables $v_1, \ldots, v_n$. Let F^+ be the CNF formula that results from setting v_1 to `True`, i.e. removing every clause containing an un-negated copy of v_1 and removing every negated copy of v_1 from F. For example, if

   $$F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4),$$

   then

   $$F^+ = x_4.$$

   Likewise, let F^- be the CNF formula that results from setting v_1 to `False`. If F is satisfiable, then either F^+ or F^- must be satisfiable, so we can proceed as follows. If F is empty, we return the empty assignment. If DecideSAT(F^+) = `True`, then we set x_1 to true and recursively find a satisfying assignment to F^+. Otherwise, if DecideSAT(F^-) = `True`, then we set x_1 to false and recursively find a satisfying assignment to F^-. Otherwise, we return that F is unsatisfiable. This requires at most $2n$ invocations of DecideSAT, so it takes polynomial time.

   (b) Give a Cook reduction from the problem of finding a maximum matching in a graph to the decision problem which asks: given a graph G and an integer k, does G contain a matching of size at least k? (Note that the graph need not be bipartite.)

   Solution: As in the previous question, we can write a polynomial-time subroutine FindMaxSize(G) to find the size of a maximum matching in G by calling our oracle $|V(G)|$ times. Using this, we set out our recursive algorithm FindMatching(G) as follows:

   - If G has no edges, return the empty set.

- Choose an arbitrary edge $e \in E(G)$, and let u and v be e's endpoints.
- If FindMaxSize$(G - e)$ = FindMaxSize(G), then return FindMatching$(G - e)$.
- Otherwise, return $\{e\} \cup$ FindMatching$(G - u - v)$.

This algorithm runs in $O(|V(G)|^2)$ time, with the bottleneck being the calls to FindMinSize.

As in the previous question, the algorithm works because of a self-reducibility argument. However we choose e, a set of edges M is a matching in G if and only if either $e \notin M$ and M is a matching in $G-e$, or $e \in M$ and $M-e$ is a matching in $G-u-v$. So FindMatching$(G-e)$ returns a matching as large as possible subject to not containing e, $\{e\} \cup$ FindMatching$(G-u-v)$ returns a matching as large as possible subject to containing e. So the algorithm uses the oracle to check whether there's a maximum matching not containing e; if so, it returns that one, and if not, it returns a maximum matching which does contain e.

Note that since there is a polynomial-time algorithm for finding a maximum matching on an arbitrary graph, it would also be technically correct to just use that as your Cook reduction (without calling the oracle at all). As all theorists know, technically correct is the best kind of correct.

6. (a) [⋆⋆] Solve the recurrence

$$a_0 = 1,$$
$$a_1 = \psi := \frac{\sqrt{5}-1}{2},$$
$$a_{n+2} = a_n - a_{n+1} \text{ for all } n \geq 0.$$

Hint: in COMS10007 last year we saw that for recurrences of this form, solutions of the form Ax^n are often a good starting point.

Solution: Guessing a solution of the form Ax^n, as covered in COMS10007 last year, we see that $Ax^{n+2} = Ax^n - Ax^{n+1}$ and hence $x^2 = x = 1 - x$. It follows that

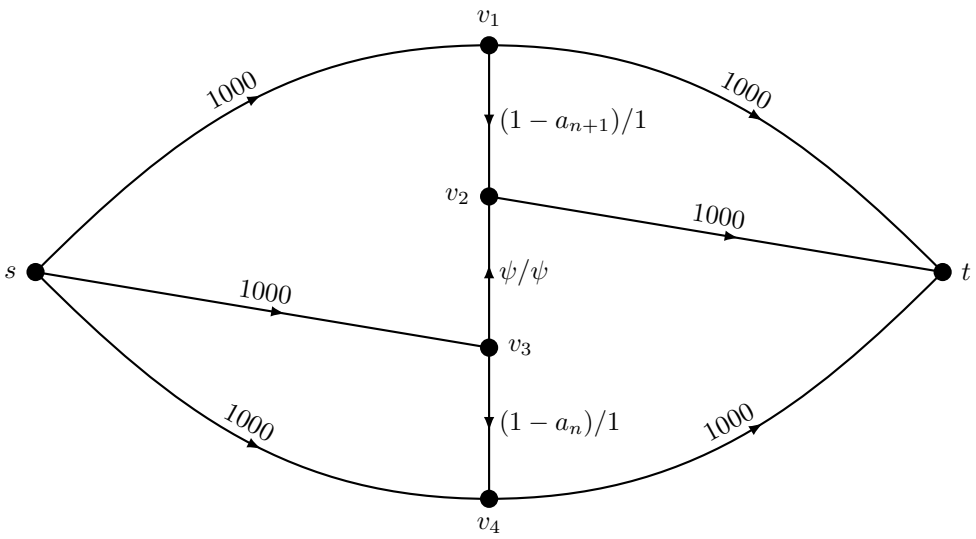
$$x = \frac{-1 \pm \sqrt{5}}{2}.$$

Thus the solution must be of the form $a_n = A\psi^n + B(-\phi)^n$, where $\phi = (1+\sqrt{5})/2$. Substituting in the known values for a_0 and a_1, we see that

$$1 = A\phi^0 + A\psi^0 = A + B,$$
$$\psi = A\phi^1 + B\psi^1 = A\phi + B\psi.$$

Solving these equations simultaneously for A and B yields $A = 1$ and $B = 0$, so $a_n = \psi^n$ for all n.

(b) [⋆⋆⋆] Consider the following flow network and the pictured flow, where the unspecified flows (on the edges incident to s and t) are arbitrary but substantially below the capacity.



Find a sequence of four augmenting paths such that, if flow is pushed along each path in sequence, then the flow on e_1 will become $1 - a_{n+3}$, the flow on e_2 will stay ψ, and the flow on e_3 will become $1 - a_{n+2}$.

Solution: The four paths are $P_1 = P_3 = sv_1v_2v_3v_4t$, $P_2 = sv_3v_2v_1t$, and $P_4 = sv_4v_3v_2t$. Note that since $a_n = \phi^n$ and $\phi < 1$, we have $a_0 > a_1 > a_2 > \ldots$. The residual capacity of P_1 is therefore $\min\{a_{n+1}, \psi, a_n\} = a_{n+1}$. After pushing flow along P_1:

- The flow along (v_1, v_2) is 1 out of 1.
- The flow along (v_3, v_2) is $(\psi - a_{n+1})$ out of ψ.
- The flow along (v_3, v_4) is $(1 - a_n + a_{n+1})$ out of $1 = (1 - a_{n+2})$ out of 1.

The residual capacity of P_2 is therefore $\min\{1, a_{n+1}\} = a_{n+1}$, and after pushing flow along P_2:

- The flow along (v_1, v_2) is $1 - a_{n+1}$ out of 1.
- The flow along (v_3, v_2) is ψ out of ψ.
- The flow along (v_3, v_4) is $(1 - a_{n+2})$ out of 1.

The residual capacity of P_3 is therefore $\min\{a_{n+1}, \psi, a_{n+2}\} = a_{n+2}$, and after pushing flow along P_3:

- The flow along (v_1, v_2) is $(1 - a_{n+1} + a_{n+2})$ out of $1 = (1 - a_{n+3})$ out of 1.
- The flow along (v_3, v_2) is $(\psi - a_{n+2})$ out of ψ.
- The flow along (v_3, v_4) is 1 out of 1.

The residual capacity of P_4 is therefore $\min\{1, a_{n+2}\} = a_{n+2}$, and after pushing flow along P_4:

- The flow along (v_1, v_2) is $(1 - a_{n+3})$ out of 1.
- The flow along (v_3, v_2) is ψ out of ψ.
- The flow along (v_3, v_4) is $(1 - a_{n+2})$ out of 1.

This is the desired flow, so we're done.

(c) [★★★] Using the previous two parts, prove that the Ford-Fulkerson algorithm need not terminate when edge weights are irrational, and moreover that terminating it early may yield an answer which is very far from optimal.

Solution: Starting from an empty flow on the above network, the Ford-Fulkerson algorithm could choose augmenting paths sv_1v_2t (setting the flow along (v_1, v_2) to 1 out of 1), then $sv_3v_2v_1t$ (setting the flow along (v_3, v_2) to ψ out of ψ and the flow along (v_1, v_2) to $1-\psi = 1-a_1$ out of 1). Since the flow along (v_3, v_4) will be $0 = 1-a_0$ out of 1, this will be exactly the flow pictured above for $n = 0$. The Ford-Fulkerson algorithm could then repeatedly choose the sequence of augmenting paths P_1, P_2, P_3, P_4, without ever terminating, and without the value of the flow ever exceeding $1 + \psi + 1$. It is easy to see that the maximum flow has value at least 2000, so this is very far from optimal.

(d) [★★★] What would go wrong with the above argument if the capacity of (v_3, v_2) were anything other than ψ?

Solution: If the capacity x of (v_3, v_2) were not equal to ψ, then we would need to take $a_1 = x$ rather than $a_1 = \psi$. This would change the boundary conditions of the recurrence from the first part, and in particular would result in $B \neq 0$. Since $\phi > 1$, we have $\phi^n \in \omega(\psi^n)$, and so for sufficiently large odd n the term $B(-\phi)^n$ would dominate and we would have $a_n < 0$. This would break the augmenting paths described in the second part.

7. Given literals $x_1, \ldots, x_k$ with $k \geq 2$, a *not-all-equal* or *NAE* clause $\mathrm{NAE}(x_1, \ldots, x_k)$ evaluates to true if and only if the values of $x_1, \ldots, x_k$ are not all equal, i.e. at least one x_i is true and at least one x_i is false. The *NAE-SAT* problem asks, given a formula of the form

$$NAE(x_{1,1}, \ldots, x_{1,k_1}) \wedge NAE(x_{2,1}, \ldots, x_{2,k_2}) \wedge \cdots \wedge NAE(x_{\ell,1}, \ldots, x_{\ell,k_\ell}),$$

whether it is satisfiable — that is, whether there is any assignment of truth values to variables which makes the formula true. In other words, it is like SAT, except that we have NAE clauses instead of AND clauses. (Note we allow literals to appear multiple times in the formula, so we may have e.g. $x_{1,1} = x_{1,2} = x_{2,1}$.)

(a) [★★★★] 3-NAE-SAT is the version of NAE-SAT in which $k_i = 3$ for all $i \in [\ell]$. Give a Cook reduction from NAE-SAT to 3-NAE-SAT.

Solution: Let $\mathrm{NAE}(x_{i,1}, \ldots, x_{i,k_i})$ be a NAE clause. Then in polynomial time, we will replace it with a polynomial-sized AND of 3-NAE clauses in $x_{i,1}, \ldots, x_{i,k_i}$ and a collection of new variables which, for given values of $x_{i,1}, \ldots, x_{i,k_i}$, is satisfiable if and only if $x_{i,1}, \ldots, x_{i,k_i}$ are not all equal. Given this construction, we can turn a polynomial-time algorithm for 3-NAE-SAT into a polynomial-time algorithm for NAE-SAT by applying this construction to each clause in the input, running our 3-NAE-SAT algorithm, and returning the answer.

If $k_i = 2$, then our construction is simple: $\mathrm{NAE}(x_{i,1}, x_{i,2}) = \mathrm{NAE}(x_{i,1}, x_{i,2}, x_{i,2})$. If $k_i = 3$, then the clause is already a 3-NAE clause. Otherwise, we work inductively by expressing our k_i-NAE clause as an AND of smaller NAE clauses. We have

$$\begin{aligned}\mathrm{NAE}(x_{i,1}, \ldots, x_{i,k_i}) = \mathrm{NAE}(x_{i,1}, x_{i,2}, u_1) \wedge \mathrm{NAE}(x_{i,2}, x_{i,3}, u_2) \wedge \ldots \\ \wedge \mathrm{NAE}(x_{i,k_i-1}, x_{i,k_i}, u_{k_i-1}) \wedge \mathrm{NAE}(u_1, \ldots, u_{k-1}),\end{aligned}$$

where $u_1, \ldots, u_{k_i-1}$ are newly-added variables. Observe that if $x_{i,1} = \cdots = x_{i,k}$, then the values of $u_1, \ldots, u_{k-1}$ must all be equal and the clause is unsatisfiable; conversely, if they are not all equal, then we will be able to set some pair of u_i's to different values. Applying the

construction repeatedly to expand out the $(k_i - 1)$-NAE clause if necessary, we end up with a total of

$$\sum_{i=4}^{k}(i-1) \leq (k-3)(k-1) < k^2$$

3-NAE clauses; thus the reduction takes polynomial time and yields a 3-NAE formula of polynomial size, as required.

(b) [⋆⋆⋆⋆] Using the first part of the question, prove that 3-NAE-SAT is NP-complete under Cook reductions.

Solution: Observe that 3-NAE-SAT is a member of NP, since we can test whether or not an assignment is satisfying in polynomial time. Since we have reduced NAE-SAT to 3-NAE-SAT, and we proved in lectures that 3-SAT is NP-complete, it suffices to prove that any polynomial-time algorithm for NAE-SAT would yield a polynomial-time algorithm for 3-SAT. Consider an arbitrary 3-SAT instance

$$F = \bigvee_{i=1}^{\ell}(x_{i,1} \vee x_{i,2} \vee x_{i,3}).$$

We claim this is satisfiable if and only if the NAE-SAT instance

$$F' = \mathrm{NAE}(x_{1,1}, x_{1,2}, x_{1,3}, z) \wedge \mathrm{NAE}(x_{2,1}, x_{2,2}, x_{2,3}, z) \wedge \cdots \wedge \mathrm{NAE}(x_{\ell,1}, x_{\ell,2}, x_{\ell,3}, z)$$

is satisfiable, where z is a new variable added to every clause. Indeed, given a satisfying assignment of F, every clause must contain at least one true literal, so it becomes a satisfying assignment of F' on setting z to `False`. Conversely, given a satisfying assignment a of F', we split into two cases to produce a satisfying assignment of F.

Case 1: Suppose $a(z) =$ `False`. Then every NAE clause in F' must contain a true literal other than z, so the corresponding AND clauses in F are also satisfied. Thus a yields a satisfying assignment for F.

Case 2: Suppose $a(z) =$ `True`. Note that $\mathrm{NAE}(x, y, z)$ is satisfied if and only if $\mathrm{NAE}(\neg x, \neg y, \neg z)$ is satisfied. For this reason we define the complementary assignment $\overline{a}$, where $\overline{a}(x) =$ `True` if $a(x) =$ `False` and $\overline{a}(x) =$ `False` if $a(x) =$ `True`. Then $\overline{a}$ is a satisfying assignment of F' in which $\overline{a}(z) =$ `False`, and we are back in Case 1.

We have therefore shown that F' is satisfiable if and only if F is satisfiable. Our reduction simply constructs F' from F, then applies our NAE-SAT algorithm to F'.

(c) [⋆⋆⋆⋆] *Monotone 3-NAE-SAT* is the version of 3-NAE-SAT in which all literals are un-negated variables, so we do not allow e.g. a clause $\mathrm{NAE}(x_1, \neg x_2, x_3)$. Using the second part of the question, prove that Monotone 3-NAE-SAT is NP-complete under Cook reductions.

Solution: Observe that Monotone 3-NAE-SAT is a member of NP, since we can test whether or not an assignment is satisfying in polynomial time. Since we have proved that 3-NAE-SAT is NP-complete, it suffices to prove that any polynomial-time algorithm for monotone 3-NAE-SAT would yield a polynomial-time algorithm for 3-NAE-SAT. Consider an arbitrary 3-NAE-SAT instance with variables $v_1, \ldots, v_n$ and formula

$$F = \bigwedge_{i=1}^{\ell} \mathrm{NAE}(x_{i,1}, x_{i,2}, x_{i,3}),$$

where $x_{1,1}, \ldots, x_{\ell,3}$ are literals taken from $\{v_1, \ldots, v_n, \neg v_1, \ldots, \neg v_n\}$. We form a corresponding monotone 3-NAE-SAT instance F' by replacing each variable v_i with a pair of variables (v_i^-, v_i^+), adding a clause $\text{NAE}(v_i^-, v_i^+, v_i^+)$ for each i, replacing each un-negated instance of v_i with v_i^+ and replacing each negated instance with v_i^-. For example, the 3-NAE-SAT formula

$$\text{NAE}(v_1, v_2, \neg v_3) \wedge (\neg v_1, v_2, \neg v_4)$$

would be replaced by the monotone 3-NAE-SAT formula

$$\begin{aligned}&\text{NAE}(v_1^+, v_2^+, v_3^-) \wedge \text{NAE}(v_1^-, v_2^+, v_4^-) \\ &\qquad \wedge \text{NAE}(v_1^-, v_1^+, v_1^+) \wedge \text{NAE}(v_2^-, v_2^+, v_2^+) \wedge \text{NAE}(v_3^-, v_3^+, v_3^+) \wedge \text{NAE}(v_4^-, v_4^+, v_4^+).\end{aligned}$$

We must show that F is satisfiable if and only if F' is satisfiable. Given a satisfying assignment a of F, we can form a satisfying assignment a' of F' by setting $a'(v_i^+) = a(v_i)$ and $a'(v_i^-) = \neg a(v_i)$. Thus the original NAE clauses evaluate to the same values as before, and we have $a'(v_i^+) \neq a'(v_i^-)$ for all i so the new NAE clauses are satisfied.

Conversely, suppose a' is a satisfying assignment of F'. Then since the new NAE clauses are satisfied, we have $a'(v_i^+) \neq a'(v_i^-)$ for all i. We then form an assignment a of F by setting $a(v_i) = a'(v_i^+)$, so that $\neg a(v_i) = a'(v_i^-)$. Thus all the literals in the NAE clauses of F under a have the same values as they did in F' under a', and so F is satisfiable.

Our reduction therefore forms F' from F, applies our Monotone 3-NAE-SAT algorithm to it, and returns the result.