CompSci 330 Design and Analysis of Algorithms
Homework 9, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: **Tuesday**, December 3, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

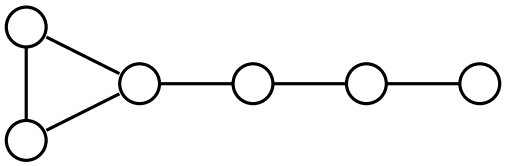
Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Kite). A *kite* is a graph on an even number of vertices, say $2n$, in which n of the vertices form a clique and the remaining n vertices are connected in a "tail" that consists of a path joined to one of the vertices of the clique. An example of a kite with 6 vertices is diagrammed below.



Given a graph and a goal g, the KITE problem asks whether there is a subgraph which is a kite and which contains $2g$ nodes. Prove that KITE is NP-complete. Your reduction should be from one of the following problems: INDEPENDENT SET, VERTEX COVER, CLIQUE, HAMILTONIAN CYCLE, or HAMILTONIAN PATH.

Solution 1. *Proving NP.* First, observe that KITE is in NP: Given a graph G and a subgraph $G' \subseteq G$, we can count the number of vertices in G' to ensure it equals $2g$ in linear time. We can identify the maximum degree node v_{max} in linear time (note that such a node should be the part of the clique that connects to the tail). There should be $g-1$ nodes connected to v_{max}, each of which is connected to all others including v_{max}, which we can easily check in, say $O(|V|^2)$ time. Finally, v_{max} should connect in the "tail", a single path, which we can check via a single DFS or similarly in linear time.

Proving NP-hardness. To prove that KITE is NP-hard, we reduce from CLIQUE. Given a graph $G = (V, E)$ and an integer $k > 0$, we construct an instance $G' = (V', E'), g \geq 0$, of KITE as follows:

(i) Create a copy of G.

(ii) For each vertex $v \in V$, connect to v a path P_v of length k. Paths of every vertex are independent. The resulting graph is G'.

(iii) The goal for the KITE instance is $g = k$ (meaning we want a kite with $2g$ vertices).

This reduction is polynomial in the size of G, as step i takes $O(|V|+|E|)$ time, step ii takes $O(|V|^2)$ time (since $k \leq |V|$ or the answer is immediately no), and step iii takes $O(1)$ time.

We claim that G has a clique of size k if and only if G' has a kite of size $2g$.

Suppose G has a clique of size k, call it C. Then $C \subseteq G'$ by construction, and every vertex in C has a tail of length k. Choosing any one tail results in a kite of size $2k = 2g$.

Conversely, suppose G' has a kite of size $2g$, call it K. K includes a clique of size g, which must be a subset of V, since the only additional vertices added to G' were in the independent tails. This is then a clique of size $g = k$.

Problem 2 (Contracts). You work for a company that takes consulting jobs but has more contracts on offer than they can manage. Specifically, you have n contract offers numbered $1, 2, \ldots, n$, each of which earns a value of v_i dollars and requires a work time of w_i person-hours. Given these, a financial goal g, and a time bound t, the CONTRACTS problem asks whether there is a set of contracts C with total work time at most t (that is, $\sum_{i \in C} w_i \leq t$) and value at least g (that is, $\sum_{i \in C} v_i \geq g$).

For example, if the following three contracts are available, then given $t = 10$ and $g = 600$, the answer would be TRUE, because choosing contracts 1 and 2 would suffice. However, if $t = 10$ but $g = 700$, the answer would be FALSE because no set of contracts with total work time at most 10 gets total value at least 700.

Contract (i)	Value v_i	Work w_i
1	200	4
2	400	5
3	500	9

Prove that this problem is NP-complete. Your reduction should be from one of the following problems: 3SAT, SUBSET SUM, or PARTITION.

Solution 2. *Proving NP.* First, observe that CONTRACTS is in NP. A solution is a set of contracts C of size at most n. We can calculate the total value and total work time in $O(n)$ time, then compare these to the goal g and bound t in $O(1)$ time.

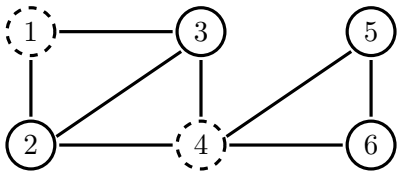
Proving NP-hardness. To prove NP-hardness, we reduce from SUBSET SUM. For every element x_i of the input set X to subset sum, create a contract i with equal value and weight $v_i = w_i = x_i$. For a target k to the SUBSET SUM problem, set $t = g = k$ in the CONTRACTS problem. This reduction takes just $O(n)$ time as it merely needs to iterate through the input set.

We claim that an instance of SUBSET SUM has a solution if and only if the constructed CONTRACTS instance has a solution.

Suppose that there is a solution $S \subseteq X$ to SUBSET SUM, then $\sum_{x_i \in X} x_i = k$. Consider choosing the corresponding contracts: Their total value and total work are both equal to $k = t = g$.

Conversely, suppose there is a solution C to the CONTRACTS instance. Then there are a set of items that sum to k corresponding to the chosen contracts.

Problem 3 (Cycle Interrupting). Let $G = (V, E)$ be a connected undirected graph. Say that a subset of vertices $I \subseteq V$ is called *cycle interrupting* in G if every cycle in G contains at least one vertex of I. For example, in the graph below, the dashed vertices 1 and 4 constitute a cycle interrupting set of size 2.



The Cycle Interrupting problem asks, given an undirected graph G and an integer k as input, whether G contains a cycle interrupting set of size at most k. Prove that this problem is NP-complete. Your reduction should be from one of the following problems: Independent Set, Vertex Cover, Clique, Hamiltonian Cycle, or Hamiltonian Path.

Solution 3. *Proving NP.* First, given a graph $G = (V, E)$ and a subset $I \subseteq V$, we can check whether I is cycle interrupting as follows: Remove I and all the edges incident on the vertices of I from G, and let $G \setminus I$ be the resulting graph. By performing a depth first search (DFS) on $G \setminus I$, we can check whether G contains a cycle, i.e., whether DFS finds a back edge. If $G \setminus I$ does not contain a cycle, we return yes, otherwise return no. The total time spent by this procedure in constructing $G \setminus I$ and performing a DFS on the resulting graph is $O(|V| + |E|)$, so we conclude that the problem is in NP.

Proving NP-hardness. We prove NP-Hardness by reducing the vertex-cover (VC) problem to it. Let $(G = (V, E), k)$ be an instance of VC. We construct an instance of Cycle Interrupting $(G' = (V', E'), k')$, as follows: Set $V' = V \cup \{x_e \mid e \in E\}$ and $E' = E \cup \{(x_e, u), (v, x_e) \mid \text{for every } e = (u, v) \in E\}$ (i.e., we replace each edge $e = (u, v)$ in G with a triangle $(u, v), (x_e, u), (x_e, v)$ in G') and $k' = k$. This construction takes just $O(|V| + |E|)$ time.

We claim that G has a vertex cover of size at most k if and only if G' has a cycle interrupting set of size at most k'.

Suppose G has a vertex cover $C \subseteq V$. Let X be a cycle in G'. If X contains an edge $(u, v) \in E$ then $C \cap \{u, v\} \neq \emptyset$ since C is a vertex cover of G. On the other hand, if X contains a vertex x_e for some $e = (a, b) \in E$, then X contains both a and b as (x_e, a) and (b, x_e) are the only edges incident on x_e. However, since C is a vertex cover and $(a, b) \in E$, $C \cap \{a, b\} \neq \emptyset$. We thus conclude that every cycle of G' contains at least one vertex of C. Hence C is cycle interrupting in G'.

Conversely, let $I \subset V'$ be cycle interrupting in G. If I contains any vertex x_e of V' and $e = (a, b)$, we replace x_e with one of a or b arbitrarily. Let $I' \subseteq V$ be the resulting subset of vertices; $|I'| \leq |I|$. We claim that I' is a vertex cover of G. Indeed, let $g = (u, v)$ be an edge of G. Since $(u, v), (v, x_g), (x_g, u)$ is a cycle in G', $\{u, v, x_g\} \cap I \neq \emptyset$. If u or v is in I, then it is also in I'. On the other hand, if $x_g \in I$, then we added one of u or v to I'. Hence, $I' \cap \{u, v\} \neq \emptyset$, i.e., I' is a vertex cover of G of size at most $k' = k$.

We have shown that G contains a VC of size at most k if and only if G' contains a cycle interrupting set of size at most k'. Since the VC problem is known to be NP-Hard, we conclude that the Cycle Interrupting problem is also NP-Hard. Combining this with the above argument that Cycle Interrupting is in NP, we conclude that it is NP-Complete.

Problem 4 (Applied) The DroneFrequency problem is as follows: You are tasked with designing a signal coordination system for a fleet of drones. Each drone operates in a circular patrol zone. Two drones whose patrol zones intersect cannot use the same communication frequency to avoid signal interference. There are three available frequencies, denoted as F_1, F_2, F_3. Given the pairs of drones whose zones intersect, the goal is to determine whether it is possible to assign each drone of the three frequencies without violating the closeness constraint.

It can be shown that DroneFrequency is NP-Complete, though that is not the purpose of this applied problem. The underlying structure of problem is common in the real world: assign few resources to all entities subject to pairwise constraints. Thus we desire solving this problem and others that are NP-hard, even if we expect the asymptotic runtime of our algorithm to be worse than polynomial. As we have throughout the entire course, instead of designing a new algorithm from the ground up, we reduce a given problem to another for which results are already known.

3-SAT, and the more general SAT problem (determine if a Boolean formula with n variables and m operations has a satisfying assignment), have been studied extensively since the 60s, with the aim of describing algorithms that are efficient in practice, even in light of the reasonable assumption that no worst-case polynomial-time algorithm exists (i.e., P $\neq$ NP). See the Wikipedia page on SAT solvers for more information[2]. Thus, for this problem, you will implement a polynomial-time reduction from DroneFrequency to 3-SAT. The implementation details are as follows.

A 3-SAT instance is a Boolean formula Φ that is the conjunction of clauses with at most three literals each. For example,

$$\Phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_3 \vee x_4) \wedge (\overline{x_4} \vee \overline{x_2})$$

is a valid 3-SAT instance; in fact, Φ is a YES instance since it has a satisfying assignment: $x_1 = T$, $x_2 = F$, $x_3 = T$, $x_4 = F$. We represent a Boolean function as follows: The literals x_i and $\overline{x_i}$ are each represented by i and -1, respectively, and each clause is represented as an array of at most three integers. For example, the formula Φ above is represented by array

$$[\quad [1, 2, -3], [-2, 3, 4], [-4, -2] \quad].$$

Towards obtaining a SAT-solver-based algorithm for `DroneFrequency`, you will implement an efficient reduction to 3-SAT: Your task is to implement a $O(n + m)$-time algorithm that, given a number of drones n and a list m of pairs of drones whose zones intersect, construct a boolean formula Φ (using the integer representation above) that is satisfiable if and only if there is a valid assignment of frequencies to the drones. For full credit, your solution will need to have an empirical runtime that is within constant factors of an $O(n + m)$-time implementation. (As a consequence, your formula should have size $O(n + m)$.)

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

- **Python.** You should submit a file called `drones.py` to the Gradescope item "Assignment 9 - Applied (Python)." The file should define (at least) a top level function `reduce_to_3sat` that looks like:

[2]In particular, see the Davis-Putnam-Logemann-Loveland (DPLL) algorithm for an elegant (worst-case exponential-time) algorithm for SAT from 1961. Some Duke trivia: Loveland is Professor Emeritus here and was Prof. Owen Astrachan's PhD advisor.

```
def reduce_to_3sat(n:int, pairs:[(u:int,v:int)])
```

that returns an integer-representation of a Boolean formula corresponding to the input as described above, where the drones in the pairs are represented by integers 0 to $n-1$.

- **Java.** You should submit a file called `Drones.java` to the Gradescope item "Assignment 9 - Applied (Java)." The file should define (at least) a top level function `reduceTo3SAT` that looks like:

```
public List<int[]> reduceTo3SAT(int n, int[][] pairs)
```

where `pairs` is a 2D array such that, for all i, `pairs[i]` is a 2-length array such that `pairs[i][0]` and `pairs[i][1]` are integers between 0 to $n-1$ that represent drones too close to be assigned the same frequency. The method should return an integer-representation of a Boolean formula corresponding to the input as described above.

Solution 4. For each drone i we create three variables i_1, i_2, i_3. The high-level idea is that exactly one should be true in a satisfying assignment of the overall formula, corresponding to assigning frequency F_j to drone i if i_j is the one true variable.

There are three kinds of clauses:

For each drone i, we add clause $(i_1 \vee i_2 \vee i_3)$, which is satisfiable if and only if at least one of the variables for drone i is true.

For each drone i we add clauses

$$(\neg i_1 \vee \neg i_2) \wedge (\neg i_2 \vee \neg i_3) \wedge (\neg i_3 \vee \neg i_1),$$

which is satisfiable if and only no two variables i_1, i_2, i_3 are true. In combination with the first kind of clause, a satisfying assignment has indeed exactly one of i_1, i_2, i_3 as true.

Lastly, for each pair of drones (i, j) whose zones interact, we must ensure the drones do not have the same frequency assigned. This is captured by the following clauses:

$$(\neg i_1 \vee \neg j_1) \wedge (\neg i_2 \vee \neg j_2) \wedge (\neg i_3 \vee \neg j_3).$$

Combining all of these clauses into a single 3-SAT instance Φ, we have a correct reduction. Constructing these clauses takes $O(1)$ time per drone and pair of drones whose zones interact, so it takes $O(n+m)$ time overall.