Homework 10

CS/ECE 374 B

Due 8 p.m. Tuesday, December 10

Question 1: From each clause according to its ability to be satisfied . . . . 10 points

A disjunctive normal form formula is the converse of CNF; i.e., it is an or of a number of clauses where each clause is an and of some terms. E.g.:

$$(x \wedge y \wedge z) \vee (z \wedge \overline{y} \wedge w) \vee (x \wedge \overline{z})$$

DNF-SAT is the analog problem of CNF-SAT: given a DNF formula f determine if there is a satisfying assignment of the corresponding variables that renders the formula true.

(4) (a) Design and analyze an efficient algorithm for DNF-SAT.

Solution: A DNF formula f is satisfiable if and only if there is at least one clause in f that is satisfiable. To check if a clause is satisfiable, it suffices to verify that a term and its negation does not appear in the clause. We can go through f in any order and check whether there exists a clause that is satisfiable. If so, we report TRUE. Otherwise, we report FALSE. Since the algorithm works in a single pass, the running time of the algorithm is linear in the size of f. ■

(4) (b) Demonstrate a reduction from 3SAT to DNF-SAT and analyze its runtime. (Hint: use the distributive law.)

Solution: Let f be the input 3CNF formula to 3SAT. Use the distributive law to convert f to its DNF form f'. In particular, we create 3^n DNF clause by picking one term from each of the 3CNF clauses. For example, given a 3CNF formula:

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

we produce the DNF formula:

$$
\begin{aligned}
f' = &(x_1 \wedge \bar{x_1} \wedge \bar{x_2})\vee \\
&(x_1 \wedge \bar{x_1} \wedge \bar{x_4})\vee \\
&(x_1 \wedge \bar{x_1} \wedge x_5)\vee \\
&(x_1 \wedge x_4 \wedge \bar{x_2})\vee \\
&(x_1 \wedge x_4 \wedge \bar{x_4})\vee \\
&(x_1 \wedge x_4 \wedge x_5)\vee \\
&(x_1 \wedge x_5 \wedge \bar{x_2})\vee \\
&(x_1 \wedge x_5 \wedge \bar{x_4})\vee \\
&(x_1 \wedge x_5 \wedge x_5)\vee \\
&(x_2 \wedge \bar{x_1} \wedge \bar{x_2})\vee \\
&(x_2 \wedge \bar{x_1} \wedge \bar{x_4})\vee \\
&(x_2 \wedge \bar{x_1} \wedge x_5)\vee \\
&(x_2 \wedge x_4 \wedge \bar{x_2})\vee \\
&(x_2 \wedge x_4 \wedge \bar{x_4})\vee \\
&(x_2 \wedge x_4 \wedge x_5)\vee \\
&(x_2 \wedge x_5 \wedge \bar{x_2})\vee \\
&(x_2 \wedge x_5 \wedge \bar{x_4})\vee \\
&(x_2 \wedge x_5 \wedge x_5)\vee \\
&(x_3 \wedge \bar{x_1} \wedge \bar{x_2})\vee \\
&(x_3 \wedge \bar{x_1} \wedge \bar{x_4})\vee \\
&(x_3 \wedge \bar{x_1} \wedge x_5)\vee \\
&(x_3 \wedge x_4 \wedge \bar{x_2})\vee \\
&(x_3 \wedge x_4 \wedge \bar{x_4})\vee \\
&(x_3 \wedge x_4 \wedge x_5)\vee \\
&(x_3 \wedge x_5 \wedge \bar{x_2})\vee \\
&(x_3 \wedge x_5 \wedge \bar{x_4})\vee \\
&(x_3 \wedge x_5 \wedge x_5)
\end{aligned}
$$

f' is logically equivalent to f, so we can complete the reduction by passing f' to the DNF solver from part (a) and returning the result. The reduction produce 3^n terms each of which can be constructed in linear time, resulting in $O(n3^n)$ runtime complexity. ■

(2) (c) Why does this not prove that $P = NP$?

Solution: Because the reduction in the previous part is not a polynomial-time reduction; the overall complexity of the CNF algorithm that uses the linear-time DNF-solver is exponential. ■

Question 2: Solomonic decision problem .... 10 points

(3) (a) Suppose you are given an algorithm partitionable, given a set X of positive integers, determines whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. (partitionable returns True or False). Design an analyze an "efficient" (see below) algorithm that computes such a partition if it exists. In other words, on an input X you should return two sets A and B such that $X = A \cup B$, $A \cap B = \emptyset$, and $\sum A = \sum B$, or return an error if such partition does not exist.

Your algorithm should call partitionable as a subroutine; its efficiency will therefore depend on the efficiency of partitionable. You should analyze both the amount of work that is done within your algorithm, and the number of calls to partitionable that are made, expressing both in asymptotic terms (e.g., "The algorithm performs $\Theta(N^2)$ work and makes $\Theta(N)$ calls to partitionable"). Your algorithm should run in polynomial time under the assumption that partitionable runs in polynomial time.

Solution: To simplify the presentation, call a partitioning of X into A, B with $\sum A = \sum B$ an equal partition. Let $X = \{x_1, \ldots, x_n\}$ and let A_i, B_i be the partition of the first i elements of X. Observe that we can extend A_i, B_i to a full equal partition of X if and only if the list $\{\sum A_i - \sum B_i, x_{i+1}, \ldots, x_n\}$ is partitionable. ($\Longrightarrow$: take $A'_i = A \setminus A_i, B'_i = B \setminus B_i$, then A'_i, B'_i is a partition of $\{x_{i+1}, \ldots, x_n\}$ and $\sum A'_i + \sum A_i - \sum B_i = \sum B'_i$, so $A'_i \cup \{\sum A_i - \sum B_i\}, B'_i$ is an equal partitioning of $\{\sum A_i - \sum B_i, x_{i+1}, \ldots, x_n\}$. $\Longleftarrow$ let A', B' be an equal partition of $\{\sum A_i - \sum B_i, x_{i+1}, \ldots, x_n\}$ and assume without loss of generality that $\sum A_i - \sum B_i \in A'$. Then $A_i \cup A' \setminus \{\sum A_i - \sum B_i\}$ and $B_i \cup B'$ form an equal partitioning of X, since

$$\sum\left(A_i \cup A' \setminus \{\sum A_i - \sum B_i\}\right) = \sum A_i + \sum A' - \sum A_i + \sum B_i = \sum A' + \sum B_i = \sum B' + \sum B_i$$

.)

We can use this to build up the sets A and B one at a time; we can try adding an element to our current set A and call partitionable to check whether this leads to a correct construction.

```
def partition(X):
    A = []
    B = []
    if not partitionable(X):
        return "not␣possible"
    cur_A_sum = 0
    cur_B_sum = 0
    while X not empty:
        x = X.pop # remove one element from X
        # tentatively add x to A and check whether this can be extended
        # to a full partition
        if partitionable(X + [cur_A_sum + x − cur_B_sum]):
            # valid
            A.append(x)
            cur_A_sum += x
        else:
            # can't be extended to a full partition, we need to add x to B
            B.append(x)
            cur_B_sum += x
    return A, B
```

This solution performs $\Theta(n)$ work since we go through X once and perform constant work in each iteration, other than the call to partitionable. Likewise, there are n calls to partitionable. Assuming that partitionable runs in polynomial time, this algorithm also runs in polynomial time.

■

(3) (b) Design and analyze an efficient algorithm for 2PARTITION: given a set X of $2n$ integers, partition them into n disjoint pairs such that the sum of each pair is equal, or report that no such partition is possible. I.e., given X with $|X| = 2n$, create sets $S_1, \ldots, S_n$ such that $|S_i| = 2$, $S_i \cap S_j = \emptyset$ for $i \neq j$, $X = \bigcup_{i=1}^{n} S_i$ and $\sum S_i = \sum S_j$ for any $1 \leq i, j \leq n$.

Solution: Let $x_{\min}$ and $x_{\max}$ be the smallest and largest elements of X, respectively. In a valid partitioning, $x_{\min}$ and $x_{\max}$ must be paired together, since for any other pairing we will have $x_{\min}, a < b, x_{\max}$. Removing these two elements and recursing, we get that ith smallest element (x_i in the sorted order of X) must be paired with the ith largest element. Thus we simply pair the elements as required and verify that the sums are all equal.

```
def 2partition(X):
    n = len(X) / 2
```

```
X.sort()
# X[−1] −> last element of the list
target = X[0] + X[−1]
pairs = [ (X[0], X[−1]) ]
for i in range(1, n):
    # X[i] −> (i+1)−st element of the list
    # X[−i−1] −> (i+1)st last element of the list
    if X[i] + X[−i−1] == target:
        pairs.append( (X[i], X[−i−1] )
    else:
        return "not␣parititionable"
return pairs
```

The running time of this solution is $\Theta(n \log n)$ since we must first sort X ($\Theta(n \log n)$) and then iterate through it, performing a constant amount of work in each iteration ($\Theta(n)$).

■

(4) (c) Show that the problem 7PARTITION is NP-Hard. 7PARTITION is defined as above, taking a set of $7n$ integers and splitting them into n disjoint sets of size 7 with the sum of each set being equal. You may assume that 3PARTITION is NP-hard.

Solution: Given an arbitrary multiset X containing $3n$ elements. Let $S = \max X$, and Let Y be the multiset of size $7n$ containing X along with $4n$ copies of $T = 3S + 1$, Creating Y is polynomial time process since we just need the sum of X and creating $7n$ elements is linear time.

Claim: X can be 3-partitioned if and only if Y can be 7-partitioned.

Proof: Suppose X is 3-partitioned. Then adding 4 copies of T to each of the partition creates a 7-partition of Y, while ensuring that each partition still has the same sum.

Suppose now that Y is 7-partitioned. We prove that each partition contains exactly 4 copies of T. Suppose otherwise, then by the pigeon hole principle, some partition p_1 has 5 or more copies of T and another partition p_2 has 3 or fewer. Then $\sum p_1 \geq 5T = 15S + 5$, whereas $\sum p_2 \leq 3T + 4S = 13S + 3$. Since $S > 0$, we must have $\sum p_1 > \sum p_2$, therefore this is not a valid 7-partition.

As a result, every partition of Y has exactly 4 copies of T; removing these from each partition leaves us with a 3-partition of X.

The reduction can easily be implemented in polynomial time; therefore, if we could solve $7PARTITION$ in polynomial time, it would indicate that we could solve $3PARTITION$ in polynomial time. Thus, $7PARTITION$ is NP-hard.

■

Question 3: Charon's crossing .... 10 points

Solve problem 42 in Chapter 12 of the textbook. Below is a restatement:

You are given a graph $G = (V, E)$ and the number k. The problem can be described as defining a sequence of subsets of vertices $X_0, \ldots, X_{2m+1}$ with the following properties:

- $X_0 = V$, $X_{2m+1} = \emptyset$
- X_i and X_{i+1} differ by at most k vertices.
- X_{2i+1} is an independent set (in G) for all $i = 0, \ldots, m$
- $V - X_{2i}$ is an independent set (in G) for all $i = 0, \ldots, m$

Less formally, you can think about having two sets X and Y. All vertices start X and at each odd "move" you shift up to k vertices from X to Y, at each odd "move" you shift up to k vertices[1] from Y to X. After every odd

[1]Technically, the formulation in the textbook formulation allows you to shift up to $k - 1$ vertices.

move, X cannot have two vertices that are connected by an edge; after every even move $Y(= V - X)$ cannot contain two vertices that are connected by an edge.

The decision problem CHARON is: given a graph G and a number k, is there a sequence of valid moves that ends with $X = \emptyset$? Your job is to show that CHARON is NP-hard.

As an example, for the classical formulation where $V = \{\text{Goat}, \text{Wolf}, \text{Cabbage}\}$, $E = \{\text{Goat} - \text{Cabbage}, \text{Wolf} - \text{Goat}\}$, and $k = 1$ we have a solution where:

$X_0 = \{\text{Goat}, \text{Wolf}, \text{Cabbage}\}$	initial configuration
$X_1 = \{\text{Wolf}, \text{Cabbage}\}$	move Goat
$X_2 = \{\text{Wolf}, \text{Cabbage}\}$	empty move
$X_3 = \{\text{Cabbage}\}$	move Wolf
$X_4 = \{\text{Goat}, \text{Cabbage}\}$	move Goat
$X_5 = \{\text{Goat}\}$	move Cabbage
$X_6 = \{\text{Goat}\}$	empty move
$X_7 = \emptyset$	move Goat; final configuration

Solution: We reduce from VERTEXCOVER.

Let $G = (E, V)$ be an arbitrary graph. We note from the get-go that the number of vertices in this graph is equal to the size of its minimum vertex cover plus the size of a maximum independent set.

Suppose that G has a minimum vertex cover of size $n \in \mathbb{N}_0$. Based on this, we can note the following pertaining to choices of $k \in \mathbb{N}_0$ for CHARON on this graph:

- If $k > n + 1$, then a simple moveset for CHARON consists of keeping at least the n vertices of a vertex cover in the boat while moving every other vertex across the river one by one, dumping the cover on the other side at the very end of this process. In this way there is always an independent set on the appropriate bank after each move.
- If $k < n$, then when taking *any* k vertices across the river, at least one vertex in the minimum vertex cover will remain on the other side, so that the vertices on the unattended bank do not form an independent set.

We thus have a range $n \leq k \leq n + 1$ on the minimum boat size.

Now suppose we construct a new graph $G' = (E', V')$ from two disconnected copies of G. We can then use a solver for CHARON on G' for each $k = 1, 2, \ldots, |V|$ and identify the minimum k for which CHARON is solvable. Because the two copies of G in G' are disjoint, the minimum vertex cover of G' has size $2n$ (n vertices from each copy), so that the bound becomes $2n \leq k \leq 2n + 1$. We can then halve this inequality to determine the minimum vertex cover of G (if k is even, then $n = k/2$; else $n = (k-1)/2$).

The reduction can easily be implemented in polynomial time, thus we have found a polynomial-time reduction of VERTEXCOVER to CHARON. But VERTEXCOVER is NP-hard, therefore so is CHARON.

(See doi.org/10.1137/110848840 for more discussion of this problem.) ■

Question 4: Semiprime kind of life..10 points (bonus)

Find the prime factors of the following integer:

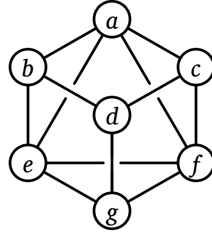
412023436986659543855531365332575948179811699844327982845455626433876445565248426198098870423161841879261420247188869492560931776375033421130982397485150944909106910269861031862704114880866970564902903653658867433731720813104105190864254793282601391257624033946373269391

Solution: This is the RSA-896 challenge: https://en.wikipedia.org/wiki/RSA_numbers#RSA-896, which has been open for nearly three decades. While you can no longer win $75,000 for solving it, you can still impress all your friends if you do manage to solve it! ■

Solved Problem

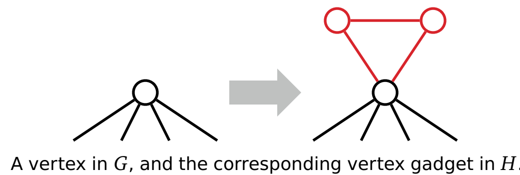
Question 5: Encore. . . . .

A double-Hamiltonian tour in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Prove that it is NP-hard to decide whether a given graph G has a double-Hamiltonian tour.



This graph contains the double-Hamiltonian tour $a \to b \to d \to g \to e \to b \to d \to c \to f \to a \to c \to f \to g \to e \to a$.

Solution: We prove the problem is NP-hard with a reduction from the standard Hamiltonian cycle problem. Let G be an arbitrary undirected graph. We construct a new graph H by attaching a small gadget to every vertex of G. Specifically, for each vertex v, we add two vertices $v^\sharp$ and $v^\flat$, along with three edges $vv^\flat$, $vv^\sharp$, and $v^\flat v^\sharp$.



A vertex in G, and the corresponding vertex gadget in H.

I claim that G has a Hamiltonian cycle if and only if H has a double-Hamiltonian tour.

$\Longrightarrow$ Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by replacing each vertex v_i with the following walk:

$$\cdots \to v_i \to v_i^\flat \to v_i^\sharp \to v_i^\flat \to v_i^\sharp \to v_i \to \cdots$$

$\Longleftarrow$ Conversely, suppose H has a double-Hamiltonian tour D. Consider any vertex v in the original graph G; the tour D must visit v exactly twice. Those two visits split D into two closed walks, each of which visits v exactly once. Any walk from $v^\flat$ or $v^\sharp$ to any other vertex in H must pass through v. Thus, one of the two closed walks visits only the vertices v, $v^\flat$, and $v^\sharp$. Thus, if we simply remove the vertices in $H \setminus G$ from D, we obtain a closed walk in G that visits every vertex in G once.

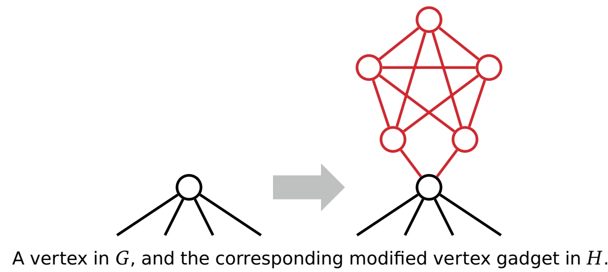
Given any graph G, we can clearly construct the corresponding graph H in polynomial time.

With more effort, we can construct a graph H that contains a double-Hamiltonian tour **that traverses each edge of H at most once** if and only if G contains a Hamiltonian cycle. For each vertex v in G we attach a more complex gadget containing five vertices and eleven edges, as shown on the next page. ■

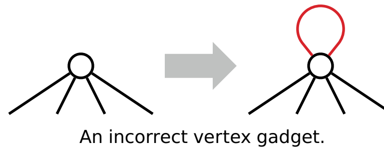
Common incorrect solution (self-loops): We attempt to prove the problem is NP-hard with a reduction from the Hamiltonian cycle problem. Let G be an arbitrary undirected graph. We construct a new graph H by attaching a self-loop every vertex of G. Given any graph G, we can clearly construct the corresponding graph H in polynomial time.

Suppose G has a Hamiltonian cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$. We can construct a double-Hamiltonian tour of H by alternating between edges of the Hamiltonian cycle and self-loops:

$$v_1 \to v_1 \to v_2 \to v_2 \to v_3 \to \cdots \to v_n \to v_n \to v_1.$$

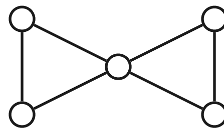


A vertex in G, and the corresponding modified vertex gadget in H.



An incorrect vertex gadget.

On the other hand, if H has a double-Hamiltonian tour, we cannot conclude that G has a Hamiltonian cycle, because we cannot guarantee that a double-Hamiltonian tour in H uses any self-loops. The graph G shown below is a counterexample; it has a double-Hamiltonian tour (even before adding self-loops) but no Hamiltonian cycle.



This graph has a double-Hamiltonian tour.

Rubric (for all polynomial-time reductions): 10 points =

+ 3 points for the reduction itself
  - For an NP-hardness proof, the reduction must be from a known NP-hard problem. You can use any of the NP-hard problems listed in the lecture notes (except the one you are trying to prove NP-hard, of course).
+ 3 points for the "if" proof of correctness
+ 3 points for the "only if" proof of correctness
+ 1 point for writing "polynomial time"

- An incorrect polynomial-time reduction that still satisfies half of the correctness proof is worth at most 4/10.
- A reduction in the wrong direction is worth 0/10.