1 Review

1. Suppose L is an NP-Complete language and has an efficient verifier V. When given an input $x \in L$ and certificate c, V (always / sometimes / never) accepts.

Solution: Sometimes. Let $L = \text{SAT}$. A verifier V for SAT can take a Boolean formula and an assignment of variables and accepts if and only if the Boolean formula resolves to true (for short, T) under the assignment. In this case, the formula is x and the assignment is c. Indeed, for the formula $\phi(v) = v$, we have $V(x = \langle \phi(v) = v \rangle, c = \{v = T\})$ accepts and $V(x = \langle \phi(v) = v \rangle, c = \{v = F\})$ rejects.

2. (True/false): The language

$$L = \{(G, C) : G \text{ is a graph and } C \text{ is a Hamiltonian cycle in } G\}$$

is in P.

Solution: True. An efficient algorithm for this language is exactly the NP verifier for the Ham-Cycle decision problem. It simply checks that C

(a) is a valid cycle in G, and

(b) goes through every vertex exactly once (i.e., it is Hamiltonian).

It is straightforward to do this efficiently (in polynomial time).

2 Search To Decision

3. Recall the knapsack language (decision problem):

$$\textsc{Knapsack} := \left\{ (W[1 \dots n], V[1 \dots n], C, K) : \exists S \subseteq \{1, \dots, n\} \text{ s.t. } \sum_{i \in S} W[i] \le C \text{ and } \sum_{i \in S} V[i] \ge K \right\},$$

where all the numbers $W[i], V[i], C, K$ are ***non-negative integers***, represented in ***binary***.

(In words, the problem asks: given n items where the ith item has weight $W[i]$ and value $V[i]$, does there exist a subset of items of total weight at most C and total value at least K?)

In this problem, assume that there exists an *efficient* algorithm D that decides Knapsack.

(a) Describe an *efficient* algorithm that, given a Knapsack instance (W, V, C, K) as input, uses D to determine the maximum value K^* of a set of items whose total weight is at most C.

Solution: Observe that the maximum value K^* must be between 0 and the total value $\sum_{i=1}^{n} V[i]$ of all items (inclusive). So, we perform a *binary search* within this interval, returning the largest value $\tilde{K}$ for which $D(W, V, C, \tilde{K})$ accepts.

This algorithm is efficient since summing the values $V[i]$ takes linear time in the total input length, and performing a binary search on the given search range is *linear* in the input size since the input is represented in binary. Also, the algorithm is correct because the optimal K^* must be in the range from 0 to $\sum_{i=1}^{n} V[i]$, and binary search will find the largest value in that range for which D accepts.

Note that looping *incrementally* from 0 up to the first value $\tilde{K}$ that D rejects and then returning $\tilde{K} - 1$ (or alternatively, looping incrementally down from the total value $\sum_i V[i]$) will take time *exponential* in the input size, since the input is represented in binary. Additionally, greedy approaches are not guaranteed to produce the optimal knapsack value and thus cannot be used here.

(b) Describe an *efficient* algorithm that, given a KNAPSACK instance (W, V, C, K) and the correct value of K^* (defined above) as input, uses D to find a set of items whose total weight is at most C, and whose total value is K^*.

Solution: We iterate through each of the n items and use D to determine whether to include the ith item.

For each item i, we temporarily put it in the knapsack and use D to test whether it is possible to use the remaining items (i.e., those with larger indices) to obtain an optimal solution. If so, we keep the ith item in the knapsack and continue with a reduced knapsack capacity $C \leftarrow C - W[i]$ and target value $K^* \leftarrow K^* - V[i]$, otherwise we leave it out and continue with unchanged capacity and target value.

```
function KnapSearch($((W, V, C, K), K^*)$)
    $S \leftarrow \emptyset$
    for $i \in \{1, 2, \ldots, n\}$ do
        if $D(W[(i+1)\ldots n], V[(i+1)\ldots n], C - W[i], K^* - V[i])$ accepts then
            $S \leftarrow S \cup \{i\}$
            $C \leftarrow C - W[i]$
            $K^* \leftarrow K^* - V[i]$
    return $S$
```

By construction, we place an item in the knapsack S if and only if it is possible to obtain the optimal value K^* with the remaining items (because D accepts). Because we are initially guaranteed that there exists a knapsack with optimal value K^*, and in the end there are no more items to consider, the final set S must be an optimal knapsack. The algorithm is efficient because it performs a linear number of iterations (in the input length, which is at least n), each of which consists of efficient operations like arithmetic and calling D.

4. Recall the language

$$\textsc{Max-Cut} = \{\langle G, k\rangle : G \text{ is an undirected graph with a cut of size at least } k\}.$$

Suppose that there exists a "black box" D that decides Max-Cut. Describe (with proof) an efficient algorithm that, given an undirected graph G, uses D to output a cut in G of maximum size.

Solution: The main ideas are as follows:

- First we use D to determine the maximum cut size k in G, by trying all possible values.
- Then, we conditionally remove each edge from G, maintaining the invariant that *G has a cut of size k*. More specifically, for each edge we temporarily remove it, and ask D whether the resulting graph has a cut of size $\geq k$; if so we remove the edge permanently, otherwise we keep it.
- At this point G has exactly k edges, because otherwise there would be some non-crossing edge of a maximum cut that ought to have been removed (since its removal would maintain the invariant). Because all k remaining edges are crossing edges of some particular maximum cut, the graph is now *bipartite*, since every edge crosses that cut.
- Using breadth-first search (or a similar greedy strategy) we can partition the vertices of the bipartite graph into two "sides," so that all k edges cross between them. This partition is therefore a maximum-size cut for the original graph.

We now proceed in more detail. Consider the following algorithm, where $G = (V, E)$ is the (undirected) input graph:

```
1: initialize $k \leftarrow 0$
2: while $D$ accepts $(G, k)$ do
3:     $k \leftarrow k + 1$
4: $k \leftarrow k - 1$
5: initialize $E' \leftarrow E$
6: for each $e \in E$ do
7:     if $D$ accepts $((V, E' \setminus \{e\}), k)$ then
8:         $E' \leftarrow E' \setminus \{e\}$
9: initialize $S \leftarrow \emptyset$, $T \leftarrow \emptyset$
10: while there exists $v \in V \setminus (S \cup T)$ do
11:     let $S \leftarrow S \cup \{v\}$
12:     run a breadth-first search on $(V, E')$ starting at vertex $v$
13:     for each vertex $u$ found during the BFS search do
14:         if the distance from $v$ to $u$ is even then
15:             $S \leftarrow S \cup \{u\}$
16:         else
17:             $T \leftarrow T \cup \{u\}$
```

```
18: return $(S, T)$
```

We claim that this algorithm returns a cut of maximum size. In lines 1-4, the algorithm finds the largest integer k such that $(G, k) \in$ MAX-CUT, which is the size k of a cut of maximum size in G. In lines 5-8, the algorithm conditionally removes each edge from (V, E') while maintaining the invariant that the maximum cut size in (V, E') is k. Hence, by the end of the algorithm, there is a cut of size k in (V, E') but there would not be such a cut if any single edge were removed from E'. It follows that all the edges in E' are crossing edges in some (maximum) cut of size k in (V, E'), so (V, E') is bipartite. Lines 10-17 find such a cut, i.e., a partition of vertices where E' is the set of crossing edges. Since we need to make E' the set of crossing edges, we must put the endpoints u, v of any edge $(u, v) \in E'$ on opposite sides of the cut. This is why the algorithm puts all the nodes which are an even distance away are in S, while all the nodes an odd distance away are in T. Because the graph is bipartite, assigning the nodes in this way ensures that every edge in E' crosses the constructed cut (S, T).

We claim that this algorithm is efficient. Indeed, the while loop on line 2 goes through at most $|E| + 1$ iterations, since G clearly can't have a cut of size $\geq |E| + 1$, and the for loop on line 6 goes through $|E|$ iterations. All the BFS runs cost a total of $|V| + |E'|$ steps. Each call to the black box costs unit time. Thus, the algorithm overall is efficient in terms of the input size.

3 Approximation

5. Suppose algorithm $\mathcal{A}$ is a 2-approximation algorithm for a minimization problem. Then, for (all / some / no) inputs x we have $|\mathcal{A}(x)| = 2|\texttt{OPT}|$, where `OPT` is an optimal solution for input x.

Solution: Some. An α-approximation algorithm must produce a solution within a factor α of the optimal for each input. It may produce better solutions for some inputs. On the other hand, it may produce solutions that are exactly $\alpha|OPT|$ for some inputs.

6. An *independent set* of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices for which there is no edge between any pair of vertices in S. The *maximum independent set* (MIS) problem is: given a graph, find an independent set of maximum size.

   Consider the following algorithm:

   1. Let $S = \emptyset$ and let $G' = G$.
   2. While G' still has at least one vertex:
      a) Choose an arbitrary vertex v of G'.
      b) Let $S = S \cup \{v\}$.
      c) Remove v and all its neighbors (including all their incident edges) from G'.
      (A neighbor of v is any vertex that is connected to v by an edge.)
   3. Output S.

(a) Let $U = V \setminus S$ denote the set of all vertices removed in step 2c, **not including** the vertices selected for S; and let Δ be the maximum degree of all vertices in G. Prove that $|U| \leq |S| \cdot \Delta$.

> **Solution:** Since the maximum degree of any vertex in the graph is Δ, then at most Δ vertices are added to U for each element added to S (and these are the only vertices added to U). Therefore, $|U| \leq |S| \cdot \Delta$.

(b) Using the result from the previous part, conclude that the algorithm obtains a $1/(\Delta+1)$-approximation for MIS.

> **Solution:** Let T^* be the maximum independent set. Then,
>
> $$\begin{aligned} |T^*| &\leq |V| && \text{Every independent set is a subset of } V. \\ &= |U| + |S| && U \text{ and } S \text{ partition } V. \\ &\leq |S| \cdot (\Delta + 1) && \text{By part a.} \end{aligned}$$
>
> Additionally, $|S| \leq |T^*|$ since T^* is a maximum independent set. Combining these two results,
>
> $$\frac{1}{\Delta+1}|T^*| \leq |S| \leq |T^*|.$$
>
> Moreover, the S output by the algorithm is an independent set by construction (because it removes from the graph all nodes that are incident to vertices in S). Therefore, the algorithm outputs a $1/(\Delta + 1)$-approximation to MIS.

7. Consider the following situation: You have been invited to a k-round game show where, in round i, you are presented with some positive integer n_i, and must decide to either add or subtract n_i from a running counter C (which is initialized to 0) before the next round. Once you commit to adding or subtracting a number, you cannot change your decision in a later round. Your goal is to keep the difference Δ between the minimum and maximum values of the counter C (over all rounds) as small as you can.
Denote the *minimum* possible difference (for a given sequence of numbers) by OPT. Show that there is an efficient algorithm that obtains a difference $\Delta \leq 2\mathsf{OPT}$, given each of the k integers as input.

> **Solution:** Here is one algorithm that works:

```
function OFFLINE($n_1, n_2, \ldots, n_k$)
    $c_{\max} \leftarrow 0$ // Tracks max value of $C$ over all rounds
    $c_{\min} \leftarrow 0$ // Tracks min value of $C$ over all rounds
    $B \leftarrow 2 \cdot \max\{n_1, \ldots, n_k\}$
    for $i = 1, \ldots, k$ do
        if adding $n_i$ would cause $c_{\max} - c_{\min} > B$ then
            subtract $n_i$ from the counter
        else
            add $n_i$ to the counter
        update $c_{\max}$ and $c_{\min}$ appropriately
    return the sequence of add/subtract decisions
```

Runtime: Computing the maximal n_i can be done by a linear pass, while the rest of the algorithm requires only one more linear pass, so the algorithm is efficient.

Correctness: It is clear to see that $\mathsf{OPT} \geq \max_{1 \leq i \leq k} n_k$ because the largest n_i will set the counter to be $C + \max_{1 \leq i \leq k} n_k$ or $C - \max_{1 \leq i \leq k} n_k$. Then the difference between these two values of the counter C is $\max_{1 \leq i \leq k} n_k$, as desired.

The algorithm maintains the invariant that $c_{\max} - c_{\min} \leq 2\mathsf{OPT}$. At the end of the first iteration, it is the case that $c_{\max} - c_{\min} \leq 2 \max_{1 \leq i \leq k} n_k \leq 2\mathsf{OPT}$ because C is initialized to 0 and the maximum number which could be added is $\max_{1 \leq i \leq k} n_k$.

By construction of the algorithm, in *any* iteration, adding n_i will not cause the difference to be greater than $2\mathsf{OPT}$. What remains to be shown is that subtracting will not cause the difference to be greater than $2\mathsf{OPT}$. If n_i is subtracted from the counter, by construction of the algorithm, it is the case that adding n_i would cause $c_{\max} - c_{\min} = (C + n_i) - c_{\min} > B = 2 \cdot \max_{1 \leq i \leq k} n_k$. Therefore, $C - c_{\min} > 2 \cdot \max_{1 \leq i \leq k} n_k - n_i \geq \max_{1 \leq i \leq k} n_k \geq n_i$ and $C - \max_{1 \leq i \leq k} n_k > c_{\min}$, so the difference does not change in this case.

Thus, the algorithm generates a sequence of add/subtract decisions attaining a difference which is at most $2 \max_{1 \leq i \leq k} n_k \leq 2\mathsf{OPT}$.

8. In lecture we discuss an approximation algorithm for TSP in a metric space (i.e., one that satisfies the triangle inequality). However, an approximation algorithm for TSP in the general case is not known to exist. Prove that an approximation algorithm for generalized TSP implies $\mathsf{P} = \mathsf{NP}$. To do this, show that for any $\alpha > 1$, if we have an efficient α-approximation algorithm A for the general TSP, then we can solve the NP-$\mathsf{Complete}$ problem HAM-CYCLE efficiently.

Hint: Given an input instance $G = (V, E)$ for HAM-CYCLE, make a new weighted graph $f(G) = (V', E')$ such that $f(G)$ is a complete graph with $V' = V$. For an edge $e \in E'$, assign weights such that $w(e) = 1$ if $e \in E$ and $w(e) = K$ if $e \notin E$, for some K.

Solution: Suppose that we are given an undirected graph $G = (V, E)$ that we would like to know if $\langle G \rangle \in$ HAM-CYCLE. Then, we make a new weighted graph $f(G) = (V', E')$

such that $f(G)$ is a complete graph with $V' = V$. For an edge $e \in E'$, we assign weights such that $w(e) = 1$ if $e \in E$ and $w(e) = 2\alpha n$ where $n = |V|$ if $e \notin E$. Clearly, f is a polynomial time computable function.

Suppose G has a Hamiltonian cycle. Then, $f(G)$ has a tour T such that $w(T) = n$. If we run A on $f(G)$, we must get a tour T' with $w(T') \le \alpha n$ since A is α-approximation algorithm.

Now, suppose G does not have a Hamiltonian cycle. Then, any tour T in $f(G)$ must contain some edge that is not originally in G, which means $w(T) \ge 2\alpha n$ since we must include a newly added edge which has weight $2\alpha n$.

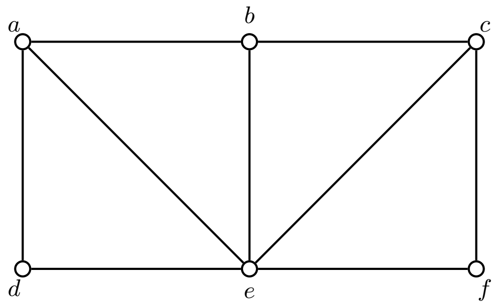
Note that the range of the solutions we get for the case when $\langle G\rangle \in$ Ham-Cycle and the case when $\langle G\rangle \notin$ Ham-Cycle are disjoint. It can be summarized in the following table. Let T^* denote a tour with optimal weight in $f(G)$ and T' the output of algorithm A on $f(G)$.

Cases	$\langle G\rangle \in$ Ham-Cycle	$\langle G\rangle \notin$ Ham-Cycle
$w(T^*)$	n	$\ge 2\alpha n$
Range of $w(T')$	$n \le w(T') \le \alpha n$	$2\alpha n \le w(T')$

Since $\alpha n < 2\alpha n$, running A on $f(G)$ can actually tell us whether there is a Hamiltonian cycle in G or not. We are assuming that A is efficient, so this would imply that Ham-Cycle $\in \mathsf{P}$. Therefore, the α-approximation search problem for TSP is NP-Hard.

9. In this problem, we will be working with the "double cover" vertex cover approximation algorithm from lecture.

   (a) Run the approximation algorithm on the following graph. What is the approximation ratio?



Solution:
The optimal solution (one of them) is $\{a, c, e\}$, which has a size of 3.
We run the approximation algorithm:

i. $C = \{\}$

ii. Remove (a, b), $C = C \cup \{a, b\}$

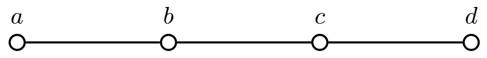
iii. Remove (c, e), $C = C \cup \{c, d\}$

iv. No edges are left, stop.

The approximation algorithm finds a solution $\{a, b, c, e\}$ with size 4. The approximation ratio is $\frac{4}{3} = 1.\overline{3}$

(b) Notice that the ordering of the edges picked in the double cover approximation algorithm is not specified. Prove or disprove that for a given graph G, the approximation ratio is the same no matter the ordering.

Solution: Here is a very simple counterexample to the statement:



Notice that the minimum vertex cover size is 2 (e.g., $C^* = \{b, c\}$); one vertex is clearly not sufficient. When using the double-cover algorithm, if we first pick the edge (b, c), then b, c are added to the cover and we are done. But if we first pick either (a, b) or (c, d), then we have not covered the other one. So that other edge will be picked next, and the output cover will have size 4.

More detailed explanation: The approximation ratio is *not* the same no matter the ordering. We can run the approximation algorithm like this:

i. $C = \{\}$

ii. Remove (a, b), $C = C \cup \{a, b\}$

iii. Remove (c, d), $C = C \cup \{c, d\}$

iv. No edges are left, stop.

This creates a solution $\{a, b, c, d\}$ with a size of 4 and an approximation ratio of 2. However, we can also run the approximation algorithm like this:

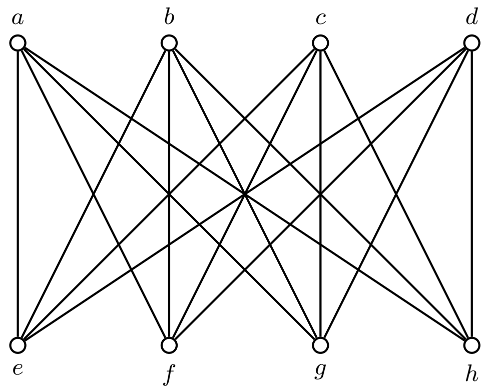
i. $C = \{\}$

ii. Remove (b, c), $C = C \cup \{b, c\}$

iii. No edges are left, stop.

This creates a solution $\{b, c\}$ with a size of 2 and an approximation ratio of 1.
Therefore, the ordering of the edges picked in the double cover approximation algorithm is significant in determining the approximation ratio.

(c) Run the approximation algorithm on the following graph. What is the approximation ratio?



Solution:
The optimal solution (one of them) is $\{a, b, c, d\}$, which has a size of 4.
We run the approximation algorithm:

i. $C = \{\}$
ii. Remove (a, e), $C = C \cup \{a, e\}$
iii. Remove (b, f), $C = C \cup \{b, f\}$
iv. Remove (b, f), $C = C \cup \{c, g\}$
v. Remove (b, f), $C = C \cup \{d, h\}$
vi. No edges are left, stop.

The approximation algorithm finds a solution $\{a, b, c, d, e, f, g, h\}$ with size 8. The approximation ratio is $\frac{8}{4} = 2$

(d) Show that for any complete bipartite graph, the double cover approximation algorithm obtains the same ratio that you found in (c), regardless of its internal choices.

Solution: Let $K_{n,m}$ be the complete bipartite graph on biparts of size m and n, with $n \geq m$. Let A be the size m bipartite and B be the size n bipartite. Note that all such $K_{n,m}$ have a minimum vertex cover of size m.
Note that whenever we remove an edge, we remove one vertex from A and one vertex from B. Let k be the number of edges that our algorithm removes on $K_{n,m}$. It cannot be the case that $k < m$ since we will have some vertex left in A. Also, it cannot be the case that $k > m$, since after we remove m edges, we would have removed all vertices A. So, $k = m$ and thus our output is $2m$, which is twice more than the optimal solution m.