Solutions to Practice Problems 11

Solution 1: In all cases, the certificates are straightforward, but the verification procedure is sometimes subtle.

(a) The certificate consists of k sequences of vertices, each corresponding to one of the cycles. To verify, we check that:

- every vertex of G appears in exactly one of the k sequences, and
- each is a valid cycle by verify for every consecutive pair of vertices in these sequences (wrapping around at the end), there is an edge between them in G.

If all these tests pass, the verification process accepts, and otherwise it rejects. This is clearly correct, and it is easy to see that it involves a linear number of accesses to G.

(b) The certificate consists of the set of k edges, each corresponding to an element in the cycle cover. A naive verification would involve checking every cycle in the digraph, but this generally would require exponential time. To verify efficiently, observe that every cycle of the graph passes through a subset of edges if and only if removing these edges results in an acyclic graph.

To verify, we remove these edges from the digraph, and test whether the resulting digraph is acyclic. (Recall that this can be done by running DFS and checking that there is no back-edge in the DFS tree.) If so, the verification process accepts, and otherwise it rejects. The edges can be removed in time linear in the size of G (e.g., zeroing out entries in G's adjacency matrix and DFS runs in polynomial time.)

To see that this is correct, consider any cycle cover E' of size k. Clearly, when the edges of E' are removed from G, there can be no cycle remaining, since any remaining cycle is not covered by E' (see Fig. 1(a)). Conversely, if the removal of any set of k edges from G breaks all the cycles, then every cycle that is in G must go through at least one of these edges.

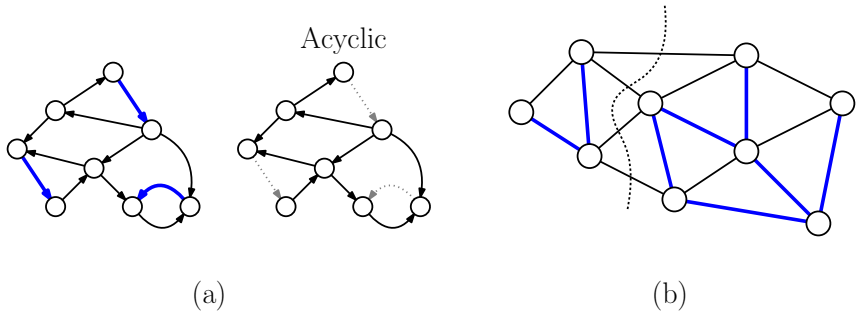

Figure 1: Verification procedures.

(c) The certificate consists of a subset E' of edges (u, v). A naive verification would involve checking every cut in the graph, but this generally would require exponential time. To verify efficiently, we observe that a subset of edges includes all cuts if and only if these edges span the entire graph.

To verify, we first check that each element of the certificate is indeed an edge of G, and we compute their sum of weights. If this sum is more than z, we reject immediately. Otherwise, we build the subgraph $G(E')$ consisting of only the edges of E'. We then check that the resulting subgraph spans G (that is, it is connected and contains all the vertices of G). If so, the verification accepts and otherwise it rejects. We can easily compute the total weight in linear time, and we can check that $G(E')$ spans G in linear time by running DFS in this subgraph.

To see that this is correct, suppose that G has a cut cover E' of weight at most z. The certificate consisting of these edges must pass the above checks. In particular, if $G(E')$ does not span G, then we could partition G's vertex set as $V = X \cup Y$, such that no edge connects X and Y, implying that this cut is not covered (see Fig. 1(b)). On the other hand, there is a subset of edges E' of total weight at most z that spans G, these edges form a cut cover because for any partition $V = X \cup Y$, there must be at least one edge of E' that connects a vertex in x to a vertex in y, implying that every cut is covered.

(If it is not connected, then there is a cut involving the connected components that is not covered by this edge set.)

By the way, this problem is not NP-complete. This is an equivalent way to formulate the minimum-spanning tree decision problem! Observe that any spanning subgraph covers every cut, and hence the minimum spanning tree, the lowest weight spanning subgraph, is the cut cover of minimum cost.

Solution 2: We will do parts (a) and (b) together. The size k^* of the smallest vertex cover is a value between 1 and $n = |V|$. We can determine this value by performing binary search with the oracle, which involves $O(\log n)$ calls to the VC oracle. If it returns "no", we increase the size parameter and if it return "yes", we decrease it.

Once we know k^*, computing the vertices of the vertex cover is complicated by the fact that G may have multiple vertex covers. A natural idea is to remove vertices one by one and check whether there is still a vertex cover of size k^*. If there is still one, then the vertex we tested was not needed, and so we remove it. Unfortunately, this might fail if the graph has multiple vertex covers. (Suppose, for example, that G is a complete bipartite graph such as $K_{3,3}$. There are two disjoint vertex covers of size $k^* = 3$. After the removal of any vertex, a vertex cover of size 3 remains. If we are not careful, we might delete all the vertices without putting any in the vertex cover!)

The trick is handle the removal differently. For any vertex u, define $G \setminus \{u\}$ to be the graph resulting by removing u and its incident edges from G. A key observation is that whenever we remove a vertex u, there are two possibilities:

- If u is a member of *any* vertex cover of size k, then $G \setminus \{u\}$ has a vertex cover of size $k - 1$ (namely, the original vertex cover minus u)
- If u is not a member of *every* vertex cover of size k, then $G \setminus \{u\}$ still has a vertex cover of size k (namely, any original vertex cover that did not include u)

This suggests the following strategy. For each vertex u, remove it and check whether $G \setminus \{u\}$ has a vertex cover of size $k-1$. If so, we add u to our vertex cover, we set $G \leftarrow G \setminus \{u\}$, and recursively check whether G has a vertex cover of size $k-1$. The code is presented below and an example is shown in Fig. 2.

Computing the Vertex Cover

```
    get-VC(G, k) {                                    // find VC of size k in G
        if (VC(G, k) = "no") Error - k is wrong!
        V' = empty                                    // V' holds the VC
        for each (u in V) and while (k > 0)
            G' = G with vertex u and its incident edges removed
            if (VC(G', k-1) == "yes")                 // u is in some VC
                V' = V' + {u}                         // add u to the VC
                G = G'                                // remove u
                k = k-1
        return V'                                     // final vertex cover
}
```

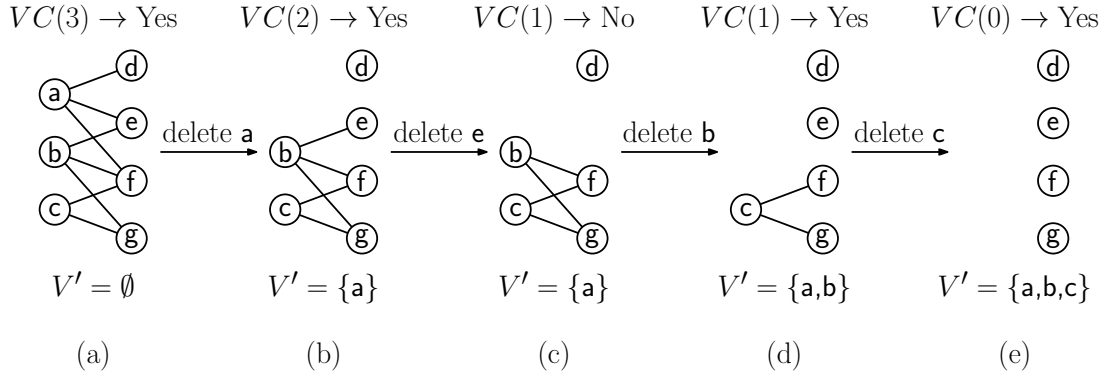

Figure 2: Finding an optimal vertex cover. The original graph has an optimal vertex cover $V' = \{a, b, c\}$ of size 3. Whenever we remove one of these vertices and invoke the oracle on $k-1$, the test succeeds.

Claim: If G has a vertex cover of size k, then `get-VC`(G, k) returns such a cover.

Proof: Observe that if G has a vertex cover of size k, then whenever we process a vertex u that is in any vertex cover of size k, the VC oracle will return "yes", and we will add u to V' and recursively find a vertex cover of size $k-1$ in $G \setminus \{u\}$. Since G has a vertex cover of size k, this test must succeed at least k times, after which $k = 0$ and the algorithm terminates. When the last vertex is removed, that is, $k = 1$, the oracle tells us that $G \setminus \{u\}$ has a vertex cover of size $k - 1 = 0$, which means that all the edges of G have been covered, implying that V' is indeed a vertex cover.

Finding the optimum vertex cover size takes $O(\log n)$ calls to the VC oracle, and `get-VC` makes $O(n)$ oracle calls, for a total of $O(n + \log n) = O(n)$ oracle calls. The running time of each iteration (which involves just deleting a vertex from G) is clearly polynomial in the size of G.

By the way, there is an alternative solution that does not involve vertex removal, but requires more oracle calls. We start with G and repeatedly attempt to add edges to G, as long as the addition of the edge does not alter the size of the vertex cover. The key observation is that when the algorithm terminates, the graph will have a particular structure. It will consist of k^* vertices, denoted V', that are connected to all the vertices of the graph (that is, they have degree $n-1$). The remaining $n-k^*$ vertices are connected to just vertices of V' (that is, they have degree k^*). It is easy to see that V' is a vertex cover of size k^* in the final graph, and further, it is not possible to add an edge to this graph without increasing the size of the vertex cover. We can identify the vertices of V' by computing the degrees of all the vertices. We output V' as the vertices of the vertex cover. This algorithm makes $O(n^2)$ calls to the oracle.

Solution 3:

(a) We first apply the function to determine whether G is Hamiltonian. If not, we immediately output a message indicating this. Otherwise, consider any edge e. We check whether $G' = G \setminus \{e\}$ (that is, the graph G with edge e removed) is Hamiltonian, by running $\mathsf{HamCycle}(G')$. If so, then we know that G' has at least one Hamiltonian cycle that does not use edge e, and so we may safely delete e. Otherwise, we keep e in the graph (every Hamiltonian cycle of G passes through e). When this algorithm terminates, we assert that G consists of a set of edges that forms a single Hamiltonian cycle. This is true, since any additional edges would have been checked and removed.

```
PrintHamCycle(G=(V,E))
    if (!HamCycle(G))
        output "G is not Hamiltonian"
        return
    for each (e in E)
        if (HamCycle(G \ {e})) G = G - {e}    // if e is unneeded, delete it
    // At this point the edges of G form a single Hamiltonian cycle
    Traverse the cycle (e.g. by DFS) and output the vertices
```

Clearly, the number of calls to $\mathsf{HamCycle}$ is $O(m)$, where $m = |E|$.

(b) Create three *color*— vertices $\{c_1, c_2, c_3\}$ that are all adjacent to each other. These must be given three distinct colors. We may assume that the colors are labeled so that c_i has color i.

For each of the remaining vertices $v \in V$, join v to each pair $\{c_1, c_2\}$, then $\{c_2, c_3\}$, then $\{c_1, c_3\}$ and ask in each case whether the graph is still 3-colorable. If so, we conclude that it is possible to 3-color G where vertex v is assigned the color corresponding to the color vertex to which it was not adjacent. Keep these two edges that force v's color. (This is important.) Repeat until all vertices have been attached to two of the new vertices. The final coloring is determined by the color vertex to which each vertex is not connected.

```
3Color(G=(V,E))
    if (!3Col(G))
        output "G is not 3-colorable"
        return
    Create vertices c1, c2, c3 and edges (c1,c2), (c2,c3), (c3,c1)
```

```
for each (v in V)
    for (i = 1 to 3)
        add edges joining v to cj, for j != i
        if (3Col(G))
            assign v color i
            break
        else
            remove the edges just added to v
output color assignment
```

On termination, every one of the original vertices of the graph will be connected to two of the special vertices, and its color will be determined by the vertex that it is not connected to. Notice that we continually check that the current graph G is 3-colorable, so assuming the original graph is 3-colorable, this procedure will succeed.

The number of calls to the 3-colorability oracle is at most $3n$ (and in fact, $2n$ suffices, since we know that at least one pair must work). Thus, the total number of calls is $O(n)$, where $n = |V|$.

Solution 4: The AKS problem is equivalent to the Hamiltonian cycle problem for undirected graphs. First, form a graph whose vertices are the knights $V = \{v_1, \ldots, v_n\}$ and the edges are the pairs $\{v_i, v_j\}$ of knights who dislike each other. Let $G = (V, E)$ be the associated graph. Let $\overline{G}$ be the complement graph, in which two knights are connected by an edge if and only if they do not dislike each other. We observe that a valid seating arrangement corresponds to a Hamiltonian cycle in $\overline{G}$, since all consecutive pairs of knights do not dislike each other.

Here is a formal argument:

- AKS $\in$ NP: The certificate consists of the sequence of knights around the table. In polynomial time, we check that every knight appears exactly once in the sequence and that no two consecutive pairs dislike each other. If so, we accept and otherwise we reject.
- HC $\leq_P$ AKS: Given an instance of a graph $G = (V, E)$ for HC, we generate an instance of AKS where the vertices are the knights and two knights v_i and v_j dislike each other if and only if $\{v_i, v_j\}$ is *not* an edge of G. Clearly, the graph associated with AKS is just the complement, $\overline{G}$. We assert that G has an Hamiltonian cycle if and only if $\overline{G}$ has a valid seating arrangement. An HC in G consists of a sequence where all consecutive pairs are connected by an edge. This is a valid seating arrangement in $\overline{G}$ since all consecutive pairs are *not* connected by an edge, and hence do not dislike each other.

Solution 5:

(a) To show that HDIS is in NP, given (G, k) the certificate consists a subset V' of k vertices of G, which will form the independent set. We verify by checking that every vertex in V' has degree at least k and no pair $u, v \in V'$ is adjacent. If so, we accept the input and otherwise we reject it. Clearly, if G has an HDIS of size k, then this verification procedure will correctly accept.

(b) We show that IS $\leq_P$ HDIS, where IS is the standard independent set problem. Consider an input (G, k) to the independent set problem. Since we do not know which vertices of G are in the independent set (or even whether an independent set exists), our approach will be to add a bunch of bogus vertices whose job it is to make the degree of *every* vertex at least as high as k, but to do so in such a way that we neither create nor destroy any existing independent sets.

First make a copy of G and add k new vertices to it. Add edges between all these new vertices. Also, for each original vertex in G, add add k edges joining it to each of the new vertices. Let G' denote the resulting graph. Output (G', k) (see Fig. 3(a)). Clearly, this can be done in polynomial time, and in fact in time $O(n + m + kn)$.

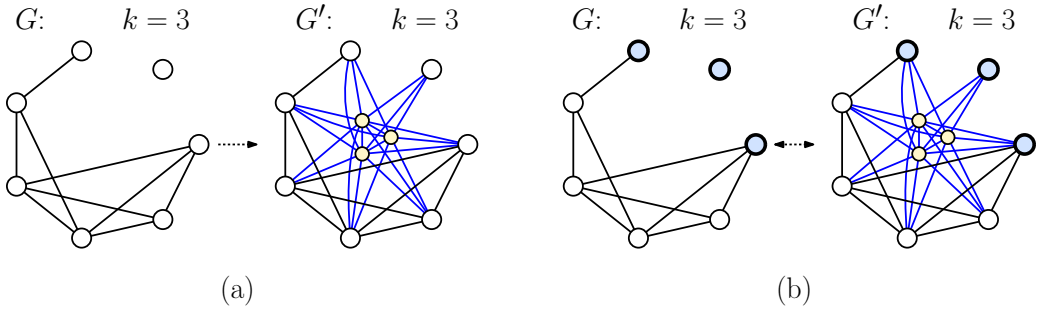

Figure 3: Reducing Independent Set to High-Degree Independent Set.

Notice that G' has $n' = n + k$ vertices, and each vertex of G' has degree at least k (because it is adjacent to all the newly added vertices in the clique induced by the new vertices). To establish correctness, we assert that G has an independent set of size k if and only if G' has an HDIS of size k.

- ($\Rightarrow$) If V' is an IS in G of size k, then the corresponding vertices of G' are not adjacent to each other, but they all have degree at least k (since every vertex in G does). Thus, V' is an HDIS in G' of size k (see Fig. 3(b)).
- ($\Leftarrow$) Suppose that V' is an HDIS in G' of size k. We may assume that $k \geq 2$, since otherwise the problem is trivial. We claim that there can be no vertices in G' from the clique of new vertices (excluding the trivial case where $k = 1$). The reason is that these vertices are adjacent to each other and every other vertex in G'. Therefore the vertices of V' come from the original vertices of G, implying that V' forms an independent set in G of size k.