Solutions to the Final Exam

Solution 1:

(a) The twice around tour is shown and the short-cutting are shown in Fig. 1(a) and (b).

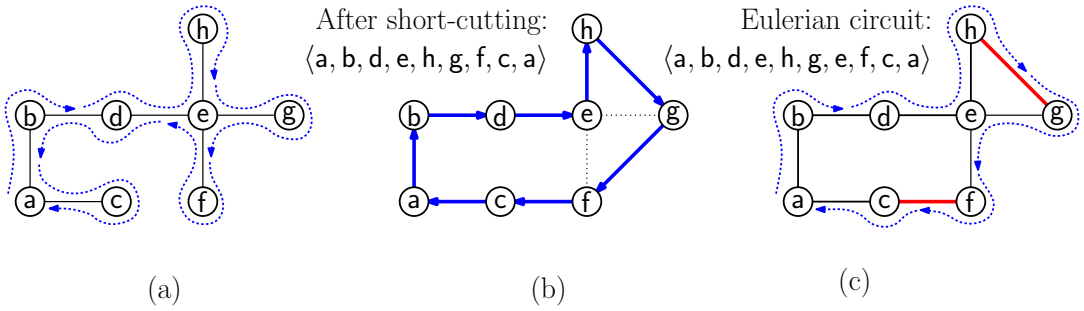

Figure 1: TSP heuristics.

(b) The vertices of odd degree are $\{\mathsf{c}, \mathsf{f}, \mathsf{g}, \mathsf{h}\}$. The minimum-weight matching on these vertices consists of (g, h) (weight 2) and (c, f) (weight 1). It is shown with red edges in Fig. 1(c).

(c) An Eulerian circuit is shown in Fig. 1(c). After short-cutting, the final tour is the same as in Fig. 1(b).

Solution 2:

(a) $2m/n$: The sum of degrees is twice the number of edges, thus the average degree is $2m/n$.

(b) A *topological ordering* is a linear order on the vertices of the DAG that respects edge directions. If the order relation is given by "$\prec$", then $(u, v) \in E$ implies that $u \prec v$. It can be computed in $O(n + m)$ time through DFS.

(c) $O(nm)$: Bellman-Ford runs in at most n phases, each of which performs relax on all edges, for a running time of $O(nm)$. (This assumes that the graph has no negative cost cycles. If so, Bellman-Ford will produce erroneous results and/or may not terminate.)

(d) True: Floyd-Warshall works even for graphs with negative cost edges, but no negative cost cycle.

(e) $\max(n, m)$, which assuming $m \leq n$ is n: In the worst case, every letter of the strings differ. We can convert one to the other by replacing each of the m letters of X to match the corresponding letter of Y, and then apply $n - m$ insertions of the letters of Y. The total is $m + (n - m) = n$.

(f) f is a maximum flow: This follows from the Max-Flow/Min-Cut Theorem.

(g) (i) and (iii): If any NP-complete problem is solvable in polynomial time, then all problems in NP (including NP-complete) are solvable in polynomial time. Because reductions can change the size of the problem instance the actual polynomial need not be the same. For example, if the reduction turns an input of size n into one of size n^2, then running an $O(n^9)$ function on this input would take time $O((n^9)^2) = O(n^{18})$ time.

Solution 3: We do this via reduction to circulations with vertex demands and lower and upper flow capacities on each edge.

We generate a network $G = (V, E)$ as follows. We create a source vertex s and a sink vertex t. We create two sets of vertices, one for the drone stations, denoted $\{d_1, \ldots, d_m\}$, and one for the customers, denoted $\{c_1, \ldots, c_m\}$. We create the following edges (see Fig. 2(a)):

- For $i \in [1, m]$, and $j \in [1, n]$, edge (d_i, c_j) of capacity range $[0, 2]$ if the drone station and customer are within 10 miles.
- For $i \in [1, m]$, edge (s, d_i) of capacity range $[3, 5]$, reflecting the fact that each station must deliver between 3 and 5 deliveries
- For $j \in [1, n]$, edge (c_j, t) of capacity range $[\max(1, o_j - 2), o_j]$.
- Edge (t, s) of capacity range $[20, 50]$. (This edge is needed because all the vertex demands are zero, implying that every vertex, including s and t must satisfy flow conservation. Note unlike edge capacities, vertex demands are fixed values, not intervals.)

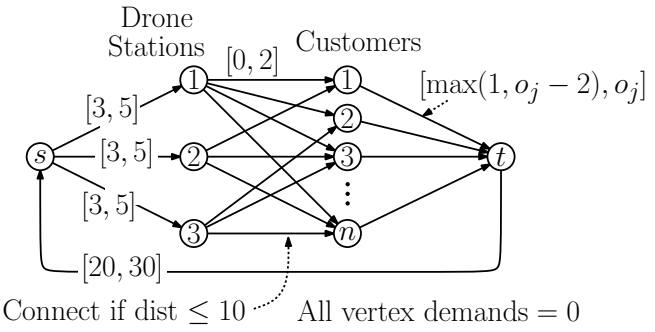

Figure 2: Solving the drone delivery problem by network flow.

All the demands are set to 0, meaning that all vertices must satisfy flow conservation. We then invoke the circulation algorithm to generate an integer-valued circulation f. If no circulation exists, we declare that there is no valid delivery schedule. Otherwise, for each edge (d_i, c_j), we ship $f(d_i, c_j)$ deliveries from station d_i to customer c_j. The following claim establishes the correctness of this reduction.

The correctness follows from the correctness of the original homework problem plus the following observations. The lower-bound constraint of 3 on the edges from s to each drone station enforce the condition that each drone station sends at least three shipments every day. The capacity interval $[20, 30]$ on the edge (t, s) enforces the condition that the total number of packages sent must be between 20 and 30. Note that due to flow conservation (since all vertex demands are zero), the flow along the edge (t, s) equals the flow out of s, which equals the total number of deliveries.

Solution 4: The size k^* of the largest 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 Clique oracle. If it returns "yes", we increase the size parameter and if it returns "no", we decrease it.

Once we know k^*, we compute the vertices of the clique as follows. (Note that we cannot hope to compute the clique in polynomial time without the aid of the oracle, since clique is NP-complete.) Let $k \leftarrow k^*$ denote the maximum size of any clique in G. For any vertex u, define $G \setminus \{u\}$ to be the graph resulting by removing u and its incident edges from G. Observe:

- if u is a member of *every* clique of size k, then $G \setminus \{u\}$ does not have a clique of size k
- if there exists a clique of size k that does *not* include u, then $G \setminus \{u\}$ still has a clique of size k.

This suggests the following strategy. For each vertex u, remove it and check whether $G \setminus \{u\}$ still has a clique of size k. If so, we remove u from further consideration. We set $G \leftarrow G \setminus \{u\}$, and recursively check whether G has a clique of size k. When the algorithm terminates, all the vertices that remain must be in every clique of size k, implying that exactly k vertices remain, and these vertices form a clique. The code is presented below, and an example is shown in Fig. 3.

Computing the Clique

```
get-Clique(G, k) {                                  // find a clique of size k in G
    if (clique(G, k) = "no") Error - k is wrong!
    for each (u in V) {
        G' = G with vertex u and its incident edges removed
        if (clique(G', k) == "yes")                 // u is not needed
            G = G'                                  // remove u
    }
    return the vertices remaining in G              // final clique
}
```

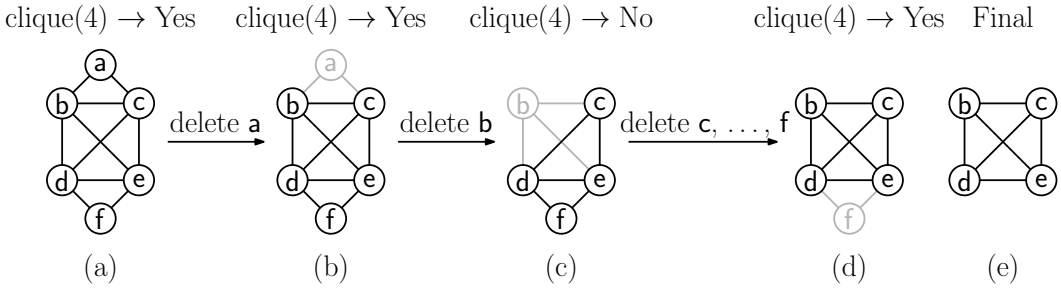

Figure 3: Finding a clique via an oracle.

We assert that if G has a clique of size k, then `get-clique`(G, k) returns such a clique. Observe that if G has a clique of size k, then whenever we process a vertex u that is not in all remaining cliques of size k, the clique oracle returns "yes", and we remove u from G. The vertices that are essential for forming a clique cannot be deleted. Eventually, the only vertices that remain in G will belong to all cliques of size k, implying that G consists of a single clique, that is, a completely connected subgraph on k vertices. Clearly these vertices form a clique in G and will be returned.

Finding the optimum clique size takes $O(\log n)$ calls to the VC oracle, and `get-clique` 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.

Grading Notes: Beware that there are minor variants of this algorithm, which *fail*. Although the differences may seem minor, getting the proper combination of ingredients is not easy, and I was fairly rigid in requiring correctness.

One incorrect approach removes a vertex u and asks the oracle whether the graph has a clique of size $k-1$. If so, it adds u to a set of vertices forming the clique and removes u permanently. This fails because the oracle *always* answer "yes", irrespective of whether u is in the clique. (Observe that if u is in the clique, then the graph has a clique of size $k-1$ without u. But, if u is not in the clique, then any $k-1$ of the original clique form a valid clique of size $k-1$, so the oracle still answers "yes".)

Another variant removes a vertex and asks the oracle whether a clique of size k still exists. If the oracle returns "yes", it reasons that this vertex is not needed for the clique, and adds it back to the graph without taking any other actions. The problem is that if the graph has multiple cliques, then no matter which vertex is removed, a clique of size k always remains, so the oracle always returns "yes", and no vertices are ever added to the clique.

Solution 5: We first show that HPP is in NP. The certificate consists of two sequences of vertices, each representing one of the two paths. We verify these sequences by checking that (1) both sequences contain the same number of vertices, (2) both start at the same vertex, and both end at the same vertex, (3) both sequences are valid paths, meaning that there is an edge between each consecutive pair of vertices, and (4) other than the start and end, every vertex appears in exactly one path. If so, the verification process accepts, and otherwise it rejects. Clearly, all of these tests can be performed in polynomial time.

Grading Notes: It was important to spell out exactly what the certificate is. It needs to be interpretable as two sequences (not sets) of vertices.

To show that HPP is NP-hard, we reduce directed Hamiltonian path (DHP) to HPP. First, observe that the reduction does not know the start or end vertices for the Hamiltonian path, since this would involve solving an NP-complete problem. Given the graph G for DHP, where $G = (V, E)$, we first create an artificial source vertex s and an artificial sink vertex t. For each $u \in V$, we add the edges (s, u) and (u, t). Letting $n = |V|$, we generate a sequence of n vertices, which we join together in a linear chain, and connect s to the front of the chain and t to the back. (Actually, any digraph having a Hamiltonian path will do. The most common answer was to just make two copies of G.) This chain will carry the second path, which also has $n+2$ vertices. Let G' denote the resulting graph (see Fig. 4). Output the pair G'. Clearly, this can be done in polynomial time. Correctness is established in the following claim.

Claim: G has a Hamiltonian path if and only if G' has a Hamiltonian path pair.

Proof: ($\Rightarrow$) If G has a Hamiltonian path π, to form one of the paths, we prepend s to the path and append t. The resulting path has $n+2$ vertices, where $n = |V|$. The second path goes from s to t along the chain of n vertices in the chain. Therefore, G' has a Hamiltonian path pair.

($\Leftarrow$) If G' has a Hamiltonian path pair, observe that it must start at s and end at t (because s has no incoming edges and t has no outgoing edges). One of the two paths must travel

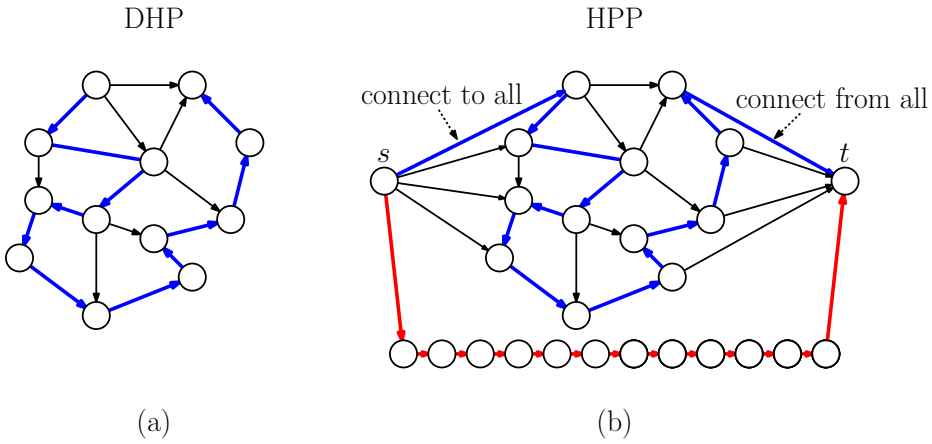

Figure 4: Reducing directed Hamiltonian path to Hamiltonian path pair.

through the chain of n vertices. The other path must visit all the vertices of G, implying that it is a Hamiltonian path in G.

By the way, there is another solution that does not involve creating the s and t vertices. Generate two copies of G, say G' and G'', and add a directed edge from each vertex $u' \in G'$ to its corresponding vertex $u'' \in G''$. This works since any Hamiltonian path in G can be mapped to two paths. One runs all the way through G' and then jumps to G' after its final vertex, and the other jumps from the starting vertex in G' into G'', and then follows the remainder of the path there. If G has n vertices, then both paths have $n+1$ vertices.

Grading Notes: I encountered many ambiguously written answers. Before requesting a regrade, please check that what you wrote (not what you thought) was clear and correct. Also, there were a few novel reductions that may have been correct, but I could not be sure. If you believe yours was correct, you can file a regrade request, but note that your grade may go down if, upon deeper reflection, I discover that the reduction is indeed not correct.

Solution 6:

(a) Here is an exact algorithm to BSS. Let $n = |A| + |B|$, and let $X = A \cup B = \{x_1, \ldots, x_n\}$. We construct a list L that contains all the possible sums that can be made from the elements $X = \{x_1, \ldots, x_i\}$. Rather than maintain a single list L, we will maintain $2n+1$ lists L_j, for $-n \le j \le n$. After the ith phase of the algorithm L_j stores all the possible sums of the elements $\{x_1, \ldots, x_i\}$, but with the additional condition that there are exactly j more items from set A than items from set B. (Alternatively, we could have stored lists $L_{i,j}$, where i indicates the number of elements of A and j indicates the number from B. But, since we only care in the relative difference of i and j, our method needs only a linear number of lists, rather than a quadratic number.) We modify the update step from the standard exact subset-sum algorithm as follows.

- If x_i is from A, we update each list L_j by copying the elements from L_j and we add x_i to the elements of L_{j-1}, reflecting the fact that we have one more item from A. That is, $L_j \leftarrow L_j \cup (L_{j-1} + x_i)$, for $-n+1 \le j \le n$.

- If x_i is from B, then we update each list L_j by copying the elements from L_j and we add x_i to the elements of L_{j+1}, reflecting the fact that we have one fewer item from A. That is, $L_j \leftarrow L_j \cup (L_{j+1} + x_i)$, for $-n \leq j \leq n-1$.

(Note that by alternating elements from A and B, we could do with just two lists, L_0 and L_1. Here we are exploiting the fact that both sets have the same numbers of elements.)

As with the standard subset-sum algorithm, after updating each list L_j, we apply the function `trim`(L_j, t) to eliminate duplicates and remove values larger than t. The final result is the max among all the elements of L_0, since this list reflects all sums having an equal number of elements of both types. The algorithm's correctness follows from the invariant at after the ith stage, for all j, the list L_j contains all the possible sums of elements that have j more elements from A than from B. We defer description of the algorithm, since it is the same as the approximation algorithm, but with the call to `compress` replaced with a call to `trim` (both from the subset-sum lecture).

Exact Balanced Subset Sum

```
exact-bss(x[1..n], t) {                    // balanced subset sum
    for (j = -n to n) L[j] = <0>           // basis case - no elements in sum
    for (i = 1 to n) {                     // consider item x[i]
        if (x[i] is from A) {              // from A
            for (j = -n+1 to n)
                L[j] = merge(L[j], L[j-1] + x[i])
        } else {                           // from B
            for (j = -n to n-1)
                L[j] = merge(L[j], L[j+1] + x[i])
        }
        L = trim(L, t)                     // remove duplicates and items > t
    }
    return the largest element in L[0]
}
```

The running time of the algorithm is $O(n^2 t)$. There are n elements to be processed, each of which is added to $2n + 1 = O(n)$ separate lists. Each list has at most t elements, and hence merging and trimming can be performed in $O(t)$ time, for a total of $O(n^2 t)$.

An alternative approach is to store one list, but where each element contains a pair consisting of the sum and the relative difference in the number of elements from A and B. Note, however, that the `trim` function needs to be carefully redefined so that it considers both the value and the count when considering that two elements are duplicates. (The same applies to `compress` in part (b).)

Grading Notes: I encountered many ideas that were clearly aimed at achieving balance, but that fell short. I was fairly harsh if I did believe that the method would work.

First, it is necessary to apply some explicit mechanism for tracking the number of elements from each list. For example, adding elements from A and B in an alternating manner may seem reasonable, but if all these sums are merged into a common list, then information needed for maintaining balance is lost. Another approach that does not work is computing pairs $a_i + b_j$ and adding them to a single list. The problem here is that if we sum these

entries, e.g., $(a_3 + b_4) + (a_3 + b_7) = 2a_3 + b_4 + b_7$, we have multiple copies of an individual item (in this case a_3), which is not allowed. What if we add pairs with equal indices only, that is, $a_i + b_i$? The problem with this is that we do not consider balanced sums involving elements with different indices, e.g., $(a_1 + a_6) + (b_3 + b_9)$.

(b) We can convert this into a PTAS by applying the compression process described in the subset-sum approximation. The algorithm is structurally the same as the one given in lecture, using the parameter $\delta = \varepsilon/n$ in the compression process. The novel element is the different interpretation of the lists. The running time is larger by a factor of $2n + 1 = O(n)$. Since the original algorithm ran in $O((n^2 \log t)/\varepsilon))$, this runs in time $O((n^3 \log t)/\varepsilon))$. The algorithm is presented in the following code block.

Approximate Balanced Subset Sum

```
approx-bss(x[1..n], t, eps) {              // approx balanced subset sum
    delta = eps/n                          // approx factor per stage
    for (j = -n to n) L[j] = <0>           // basis case - no elements in sum
    for (i = 1 to n) {                     // consider item x[i]
        if (x[i] is from A) {              // from A
            for (j = -n+1 to n)
                L[j] = merge(L[j], L[j-1] + x[i])
        } else {                           // from B
            for (j = -n to n-1)
                L[j] = merge(L[j], L[j+1] + x[i])
        }
        L = compress(L, delta, t)          // ...compress similar values and items > t
    }
    return the largest element in L[0]
}
```

Grading Notes: Many people repeated the same structure from (a), and if points were deducted there, I repeated the deduction in (b), albeit slightly reduced. Many people made reference to δ without defining it, and there was a small deduction for this.