Solutions to Practice Problems 13

Solution 1: We want to show that the alternating heuristic for bottleneck TSP is a factor-2 approximation. Let B^* be the cost of the optimum bottleneck TSP tour and let B^A be the cost of the alternating TSP tour. Let L denote the maximum distance between any two consecutive points, that is $L = \max_{1<i\leq n} |x_i - x_{i-1}|$. Observe that *any* TSP tour must make at least one hop of length at least L, therefore $B^* \geq L$. On the other hand, the alternating heuristic never makes a hop of length greater than $2L$, since this is the maximum distance between two points on the alternating path. Therefore $B^A \leq 2L$. Combining these, we have $B^A \leq 2L \leq 2B^*$, which implies that the alternating heuristic is at most twice as bad as the optimal bottleneck TSP solution.

Note: I believe that the alternating heuristic is in fact optimal. Here is a sketch of a proof, which looks very similar to the proofs of greedy algorithms we have studied. Suppose towards a contradiction that the optimal tour is not the same as the alternating tour. Find the first instance where the alternating pattern is violated, which implies that the hop taken from one path completely jumps over the hop taken from the other path. By a bit of "path surgery" you can locally rearrange these paths so they are alternating at this location in such a way that the maximum hop length does not increase. (The details would need to be spelled out.) By repeating the process, the path can be converted to one that is alternating, and its cost has not increased. Therefore, the alternating solution is optimal.

Solution 2:

(a) The counterexample takes a bit of effort. Our approach is to simulate the worst-case example for the greedy set cover. (Recall Figure 3 from Lecture 7.)

We start with two rows of vertices, each containing 7 vertices. (Any sufficiently large number would work.) We create two vertices a and b that will be connected to all these vertices. These constitute a dominating set of size 2, which is the optimal solution.

Our objective will be to trick the greedy algorithm into a selecting a larger dominating set by creating vertices other than a and b that have higher degree. To do this, we add two new vertices, c and d. Vertex d will be joined to the last 4 vertices in each of the rows. Vertex c will be joined to preceding 2 vertices in each of the rows. Since we want a and b to form a dominating set, we also add edges (a, d) and (b, c). (See Fig. 1.)

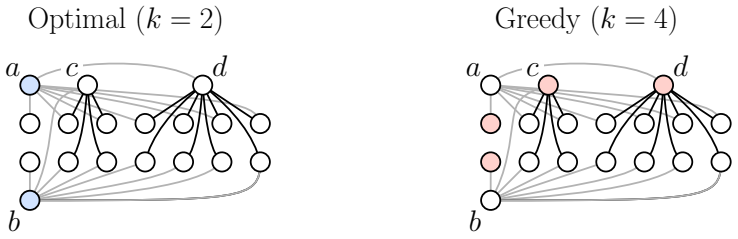

Figure 1: Counterexample for the greedy dominating set algorithm.

Now, let's see what the greedy algorithm does with this example:

- Initially, the vertex with the highest degrees is d, with a degree of 9. (Vertices a and b each have degree 8.) The greedy heuristic first selects d.
- After this, a is covered, so it is not eligible to be selected and the degree of vertex b decreases to 4. The next vertex to be selected is c, which has degree 5.
- After this, vertex b is covered, so it is not eligible. There are just two vertices remaining (the first vertices in the two rows), and there is no edge between them. So both must be selected to be in the dominating set.
- The final result is a dominating set of size 4, which is twice the optimal size.

Observe that each time we add a vertex to our dominating set, we cover roughly half of the remaining vertices. This example can be generalized to show that there the worst case for greedy is roughly $\log_2 n$.

(b) The dominating-set problem can be reduced to the set-cover problem. Recall that in set cover we are given a universe $X = \{x_1, \ldots, x_m\}$ and a collection of sets $S = \{s_1, \ldots, s_n\}$ and are asked to find a minimum-sized collection of S whose union covers all the elements of X.

Given the graph $G = (V, E)$, we set $X = V$, and for each vertex u, we define the set s_u to consist of u together with all of u's neighbors in G. To see why this is correct, observe that if G has a dominating set V' of size k, then every vertex of V is either in or is adjacent to a vertex of V', which implies that the sets associated with the vertices of this dominating set define a set cover for S of size k. Conversely, if S has a set cover of size k, then the associated vertices of G must have the property that ever vertex of G is either in this set or is adjacent to a vertex of this set. Therefore, the associated vertices defines a dominating set of size k within G'.

As shown in class, the greedy set cover algorithm is guaranteed to produce a collection whose size is larger than the optimum by a factor if at most $\ln |X| = \ln |V| = \ln n$. Therefore, it will succeed in computing a dominating set that satisfies this size constraint.

Solution 3:

(a) We claim that the optimum tour just walks around the bounding rectangle for the points. To see that this is optimal, observe that there are $2n$ points and hence $2n$ edges, and each pair of points is separated by unit distance, so any TSP tour has total length at least $2n$. This is exactly the same as the perimiter of the bounding rectangle, so this is optimal.

(b) Let's start with a_1 and travel in clockwise order around the MST. Before short-cutting, the twice-around tour has $\langle a_1, b_1, a_1, a_2, b_2, a_2, b_3, \ldots, a_n, b_n, a_n, a_1\rangle$ (see Fig. 2(a)). After short-cutting, we have $\langle a_1, b_1, a_2, b_2, b_3, \ldots, a_n, b_n, a_1\rangle$. There are $n-1$ segments, each of the form $\langle a_i, b_i, a_{i+1}\rangle$, which has an L_1 length of $1+2=3$. At the end of the tour, we have the sequence $\langle a_n, b_n, a_1\rangle$, which has an L_1 length of $1+2+(n-1)$. Thus, the overall L_1 length of this tour is $\text{TA}(P(n)) = 3(n-1) + (1+2+(n-1)) = 4n-1$. Therefore, the performance ratio is

$$\lim_{n\to\infty} \frac{\text{TA}(P(n))}{\text{TSP}(P(n))} = \lim_{n\to\infty} \frac{4n-1}{2n} = 2.$$

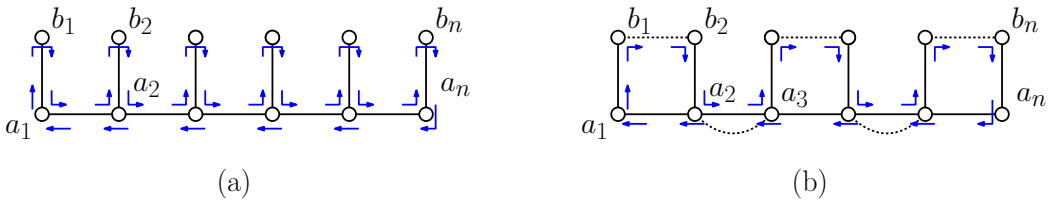

Figure 2: Approximation ratios for the metric TSP problem.

(c) The odd-degree vertices consist of the teeth of the comb (all of degree 1) and the vertices $a_2, \ldots, a_{n-1}$ (all of degree 3). Assuming that n is even, the minimum-weight perfect matching among these points connects (b_{2k-1}, b_{2k}) for $1 \leq k \leq n/2$ and (a_{2k}, a_{2k+1}), for $1 \leq k \leq n/2-1$. In total, there are $n-1$ edges of length 1, for a total weight of $n-1$.

The Eulerian circuit has total weight equal to the weight of the MST, which is $2n-1$ plus the $n-1$ from the matching, which is $3n-2$. The resulting Eulerian circuit consists of two pieces. First, a path that zig-zags from left to right: $\langle a_1, b_1, b_2, a_2, a_3, b_3, b_4, a_4, \ldots, b_n, a_n \rangle$. The second is a straight-line path from a_n back to a_1, going through all the a-points. Short-cutting does not affect the length of either of these paths, so the L_1 length of the Eulerian circuit after short-cutting is still $3n-2$. Therefore, the performance ratio is

$$\lim_{n\to\infty} \frac{\mathrm{TA}(P(n))}{\mathrm{TSP}(P(n))} = \lim_{n\to\infty} \frac{3n-2}{2n} = \frac{3}{2}.$$

Solution 4: This problem is known as the *recitilinear minimum Steiner tree problem* (RMST).

(a) Consider the following point set $P = \{(0,1), (1,0), (1,2)\}$ (see Fig. 3(a)). $\mathrm{MST}(P)$ consists of two edges, each of length 2, for a total weight of 4 (see Fig. 3(b)). In contrast, the minimum connector places a vertex at $(1,1)$, and has three line segments going to each of the three points, for a total weight of 3 (see Fig. 3(c)).

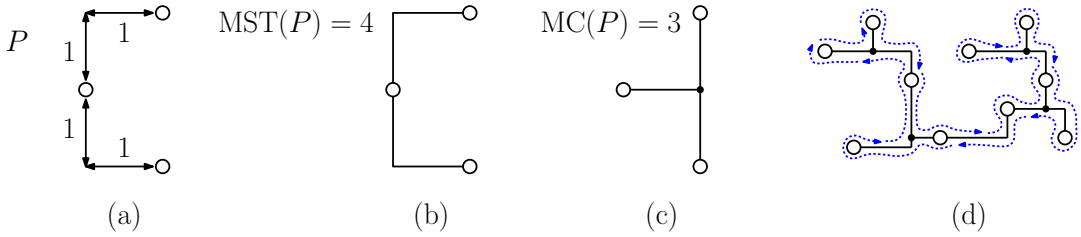

Figure 3: Minimum connector.

(b) In class, we showed that in any metric space $\mathrm{MST}(P) \leq \mathrm{TSP}(P)$ (since the TSP minus one edge is a spanning tree). We will use a similar argument to show that $\mathrm{TSP}(P) \leq 2 \cdot \mathrm{MC}(P)$. First, observe that the minimum-connector must be a tree, since we could eliminate any cycle while maintaining connectivity and decreasing the total weight. Since it is connected, we can apply the twice-around tour and then apply short-cutting to obtain a TSP tour whose cost is at most twice that of $\mathrm{MC}(P)$. Therefore, we have

$$\mathrm{MST}(P) \leq \mathrm{TSP}(P) \leq 2 \cdot \mathrm{MC}(P),$$

implying that

$$\frac{\text{MST}(P)}{\text{MC}(P)} \leq 2.$$

Solution 5: Let's first derive an exact algorithm. Recall that in the exact algorithm presented in class, after phase i, for $0 \leq i \leq n$, the list L contains all the possible sums that can be made from the elements $\{x_1, \ldots, x_i\}$. Rather than maintain a single list L, we will maintain $m+1$ lists L_j, for $0 \leq j \leq m$. 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 hazardous elements in the sum. We modify the algorithm as follows.

- If x_i is non-hazardous, then we update each list L_j following the standard process. That is, $L_j \leftarrow L_j \cup (L_j + x_i)$, for $0 \leq j \leq m$.
- If it is hazardous, then when adding x_i, we need to promote from L_{j-1} to L_j. That is, $L_j \leftarrow L_j \cup (L_{j-1} + x_i)$, for $1 \leq j \leq m$.

The final result is the max among all the lists. The correctness follows as with the standard algorithm. We can convert this into an approximation algorithm by applying the compression process. The running time is larger by a factor of m which is at most 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 Subset Sum with Hazards

```
approx-hss(x[1..n], t, eps) {              // approx subset sum with hazards
    delta = eps/n                          // approx factor per stage
    for (j = 0 to m) L[j] = <0>            // basis case - no elements in sum
    for (i = 1 to n) {                     // consider item x[i]
        if (x[i] is not hazardous) {
            for (j = 0 to m)
                L[j] = merge(L[j], L[j] + x[i])     // standard update rule
        } else {                                    // x[i] is hazardous
            for (j = 1 to m)
                L[j] = merge(L[j], L[j-1] + x[i])   // promote when adding x[i]
        }
        L = compress(L, delta, t)          // ...compress similar values and items > t
    }
    return the largest element in L[0], ..., L[m]
}
```