Please enter your name and uID below.

Name: Qianlang Chen
uID: u1172983

Submission notes

- Due at 11:59 pm on **Wednesday, December 2**.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Minimum Flow) a. My proposed algorithm runs as follows:

Take input of a graph G = (V, E), a source vertex s, and a target vertex t.
Run Breadth-First Search to find and record the shortest path (in terms of the number of edges involved) from s to each vertex in V.
Run Breadth-First Search to find and record the shortest path to from each vertex in V to t.
For each edge uv in E: Set flow f(uv) = 0.
While there is an edge uv with f(uv) = 0:
    Set f(uv) = 1.
    For each edge ab in the shortest path from s to u: set f(ab) = f(ab) + 1.
    For each edge ab in the shortest path from v to t: set f(ab) = f(ab) + 1.
Find the edge in E with the highest demand and call that demand d.
For each edge uv in E: set f(uv) = d * f(uv).
Return f.

b. As follows:

Take input of a graph G = (V, E), a source vertex s, and a target vertex t.
Run AlgorithmFromPartA with input (G, s, t).
For each edge uv in E: set capacity c(uv) = f(uv) – demand of uv.
Run the efficient MaxFlow with input (G, c, s, t) and call its flow output f'.
For each edge uv in E: set f(uv) = f(uv) – f'(uv).
Return f.

2. (APSP) a. My proposed algorithm runs as follows (assuming that the edges are stored with a data structure that allows fast lookups, such as a hash-set):

Take input of a graph G = (V, E) and a vertex v in V.
Let E' = E (make a copy).
Traverse E to find the set of edges H that have v as the heads.
Traverse E to find the set of edges T that have v as the tails.
For each edge av in H:
    For each edge vb in T:
        Remove av and vb from E'.
        Create edge ab with weight the sum of the weights of av and vb.
        If there already exists ab in E': keep the one with the lower weight.
        Otherwise: add ab to E'.
Return G' = (V \ {v}, E').

b. As follows:

Take input of graphs G = (V, E), G' = (V \ {v}, E'), the vertex v, and the shortest distance dist(a, b) between each pair of vertices (a, b) in G'.
Traverse E to find the set of edges H that have v as the heads.
For each vertex s in (V \ {v}):
    Let dist(s, v) = positive infinity.
    For every edge av in H:
        Let dist(s, v) = min[dist(s, v), weight of av + dist(s, a)].
Traverse E to find the set of edges T that have v as the tails.
For each vertex t in (V \ {v}):
    Let dist(v, t) = positive infinity.
    For every edge vb in T:
        Let dist(v, t) = min[dist(v, t), weight of vb + dist(b, t)].

c. Let's call this algorithm APSP, which runs as follows (assuming that the input dist is already initially filled with the weights of all edges uv and zeros for all the same-vertex pairs uu):

Take input of a graph G = (V, E) and the shortest distances dist.
If |V| = 1: return.
Let v = any vertex in V.
Let G' = AlgorithmFromPartA with input (G, v).
Run APSP with input (G', dist).
Run AlgorithmFromPartB with input (G, G', v, dist).

3. (BoxDepth) a. Part 1, reduction: given an instance of the BoxDepth problem (R = {r_1, …, r_n}, k), where R contains the rectangles and k is as in the description, we can create an instance of the Clique problem (G = (V, E), p) as follows:

For each rectangle r_i in R, create the vertex v_i in V.

For each rectangle r_i in R, create the edge v_i v_j in E if r_i and r_j share a common point, where 1 <= j < i. We can check whether two rectangles r and s share a common point by evaluating this: "r.x <= s.x + s.width && s.x <= r.x + r.width && r.y <= s.y + s.height && s.y <= r.y + r.height".

Set p = k.

The Clique instance constructed by this strategy has n vertices, where n is the number of rectangles there are, and m edges, where m is the number of pairs of rectangles that share a common point. The most time-consuming part of the reduction is creating the edges, which runs in O(n^2) by traversing all pairs of rectangles. Therefore, the entire reduction runs in O(n^2).

Part 2, show that BoxDepth => Clique: suppose that there are at least k rectangles in R sharing a common point. Call the first k rectangles that share the same point s_1, …, s_k. This means that s_i shares a common point with s_j for all 1 <= i, j <= k. By the way we constructed the Clique instance, there must exist an edge u_i u_j for all 1 <= i, j <= k, and we've defined p = k. This implies that G has a subgraph involving the vertices u_1, …, u_p that's complete, meaning that G has a complete subgraph with at least p vertices.

Part 3, show that Clique => BoxDepth: suppose that G has complete subgraph with at least p vertices. Call the first p vertices from the same subgraph u_1, …, u_p. The completeness implies that there exists an edge u_i u_j for all 1 <= i, j <= p. By the way we constructed the Clique instance, the rectangle s_i must've shared a common point with s_j for all 1 <= i, j <= p, and we've defined k = p. This means that the rectangles s_1, …, s_k all share a common point, so there are at least k rectangles in R that share a common point.

b. For each pair of rectangles (r, s) in R, count how many rectangles contain the point (r.x, s.y). We can check whether a rectangle t contains a point p by evaluating this: "t.x <= p.x && t.x + t.width >= p.x && t.y <= p.y && t.y + t.height >= p.y". Record the maximum count ever reached in that process. Output "yes" if that maximum count is at least k, or "no" otherwise. This algorithm has running time of O(n^3): O(n^2) for going over all pairs of rectangles, and O(n) for going over all rectangles to count the containments.

c. In this case, we used an NP-hard problem as a subroutine to solve a P problem, which is the opposite to what's needed to attempt showing that P = NP. In other words, using an "inefficient" subroutine for some "easy" problem that can be solved efficiently doesn't make or show the easy problem being hard, nor does it show the hard problem being easy.

4. (MultiCut) a. A certificate of a yes-instance of the MultiCut problem is a set of edges to remove (called R). To verify it, we need to do the following:

Check if |R| <= k.
Remove edges in R from E.
For each pair of vertices (s, t) in {(s_1, t_1), …, (s_c, t_c)}:
Run Depth-First Search starting from s to check whether t can be reached.

It's easy to see that the above verification algorithm has running time O(c * (V + E)) = O(V^3) because c <= |V| and |E| <= |V|^2. Since we can verify a yes-answer in polynomial time, MultiCut is in NP.

b. Part 1, reduction: given an instance of the VertexCover problem (G = (V = {v_1, …, v_n}, E), k), where G is the graph and k is the integer constant as in the problem description, we can create an instance of the MultiCut problem (H = (W, F), P = {(s_1, t_1), …, (s_c, t_c)}, q) as follows:

Create n+1 vertices in W numbered 0, …, n (where n = |V|).
Create n edges that connect every vertex i > 0 to vertex 0.
For each edge u_i u_j in V, create the vertex-pair (i, j) in P.
Set q = k.

The MultiCut instance constructed by this strategy has n+1 vertices (where n = |V|), n edges, and m vertex-pairs (where m = |E|). The vertex- and edge- creation parts should both take O(n) running time, while the vertex-pair-creation part takes O(m). Therefore, the entire reduction runs in O(n + m).

Part 2, show that VertexCover => MultiCut: suppose that there exists a vertex cover of size d (d <= k) in G. Call the vertices involved in the vertex cover v_a_1, …, v_a_d, which correspond to the vertices numbered a_1, …, a_d in W. Now, remove the edges (0, a_1), …, (0, a_d) from F. Since every vertex-pair (i, j) in P corresponds to the edge u_i u_j in E and by definition of a vertex cover, removing the edges (0, a_1), …, (0, a_d) effectively removes either (or both) of (0, i) and/or (0, j) for any (i, j) in P. By the way we originally constructed the edges in H, the only path between any pair of non-zero vertices i and j must go through vertex 0 and include precisely the edges (0, i) and (0, j), but now at least one of (0, i) and (0, j) is removed if (i, j) is a pair in P, meaning that any vertex-pair in P has been disconnected. We only removed d edges to achieve this, and we know that d <= k and have defined q = k. Therefore, there is a way to remove at most q edges from F to disconnect all vertex-pairs in P.

Part 3, show that MultiCut => VertexCover: suppose that there's a way to remove d (d <= q) edges from F to disconnect all vertex-pairs in P. Call the removed edges (0, a_1), …, (0, a_d), where a_i, …, a_d correspond to the vertices v_a_1, …, v_a_d in V. Since removing these edges disconnects all vertex-pairs in P, for any pair (i, j) in P, at least one of (0, i) and (0, j) must've been removed because any path from i to j must go through vertex 0 since i, j > 0 by construction. This means that if we took the vertices v_a_1, …, v_a_d in V, they'd form a vertex cover because then all edges in E would have at least one endpoint included in the cover, as the edges in E corresponded to the vertex-pairs in P. Therefore, since d <= q and we've defined k = q, there exists a vertex cover in G that involves at most k vertices.