The Design and Analysis of Algorithms HW 2

B12508026 戴偉璿

December 12, 2025

1. This problem can be solved by the classic Hierholzer's algorithm. Start up from any vertex in the graph and keep traversing unused edges until returning to the starting vertex (i.e. the result would be a circle). If there are still unused edges, start from any vertex on the current cycle that has unused edges and repeat the above process, obtaining another cycle. Iterating the process until there is no unused edge. Finally, we can combine all the cycles obtained above into one Eulerian cycle.

   This graph is guaranteed to has even degree on each vertex since it has an Eulerian cycle, so we can always enter a vertex through an unused edge and leave it through another unused edge (except for the case when we return to the starting vertex to end a cycle). Thus, we are guaranteed to traverse all edges once and only once. The time complexity is $O(E)$ since each edge is traversed once.

   The answer is discussed with ChatGPT and refrerred to `https://algorithms.discrete.ma.tum.de/graph-algorithms/hierholzer/index_en.html`.

2. The bottleneck of the Kruskal's algorithm is to sort the edges according to their weights. Consider the weights are integers within a limited range from 1 to W, thus, we can employ a more efficient sorting algorithm (e.g. bucket sort) to break the $O(E \log E)$ time complexity. However, W may be large, leveraging the radix sort is a more practical choice.

   First, we sort the edges according to their weights using the radix sort. The time complexity is $O(E \cdot \log W)$. After sorting, we can apply the normal Kruskal's algorithm to find the minimum spanning tree. The time complexity is $O(E \cdot \alpha(V))$, where α is the inverse Ackermann function. Therefore, the overall time complexity is $O(E \cdot \log W + E \cdot \alpha(V))$.

   Besides, if W is relatively small, we can also use the counting sort to sort the edges in $O(E + W)$ time. In this case, the overall time complexity would be $O(E + W + E \cdot \alpha(V))$.

   This answer is thought by myself and checked the correctness with ChatGPT.

3. To solve this problem, we can execute Single Source Shortest Path (e.g. Dijkstra's) algorithm for v_1, v_2, v_3, respectively, and store the shortest distance from each of these

vertices to all other vertices. After that, enumerating all vertices in the graph as the "intermediate" vertex s.

First, by applying Dijkstra's algorithm to v_1, v_2, v_3, we obtain the shortest-path distance and store them in $dist_1[\cdot]$, $dist_2[\cdot]$, $dist_3[\cdot]$. Besides, store the path in three predecessor arrays $prev_1[\cdot]$, $prev_2[\cdot]$, $prev_3[\cdot]$. This step takes $O(|V|\log|V| + |E|)$ (with Fibonacci heap). Second, we enumerate all vertices s in the graph and calculate $dist_1[s] + dist_2[s] + dist_3[s]$. The minimum among these values is the answer. This step takes $O(|V|)$ time. Afterall, reconstruct the subpaths from v_1, v_2, v_3 to s using the predecessor arrays. This step takes $O(|V|)$ time. The overall time complexity is $O(|V|\log|V| + |E|)$.

This answer is thought by myself and checked the correctness with ChatGPT.

4. (a) D$\rightarrow \Pi$ By definition, if $\pi_{ij} = k$, then $d_{ij} = d_{ik} + d_{kj} = d_{ik} + w(k, j)$ where $w(k, j)$ is the weight of edge (k, j). For all i, j pairs, we have the following cases:

   - If $i = j$ or $d_{ij} = \text{inf}$, then there is no vertex before j on the shortest path from i to j. Thus, we set $\pi_{ij} = \text{NIL}$.
   - Otherwise, scan all vertices adjacent to j to find a vertex m such that $d_{ij} = d_{im} + w(m, j)$. Set $\pi_{ij} = m$.

   Following is the pseudocode:

```
for i in 1..n:
    for j in 1..n:
        if i == j or d[i][j] == +inf:
            pi[i][j] = NIL
        else:
            pi[i][j] = NIL
            for k in 1..n:
                if (k,j) in E and d[i][j] == d[i][k] + w[k][j]:
                    pi[i][j] = k
                    break
```

   The time complexity is $O(V^3)$ since we have three nested loops.

   (b) $\Pi \rightarrow D$ For all i, j pairs, we have the following cases:

   - If $i \neq j$ and $\pi_{ij} = \text{NIL}$, then there is no path from i to j. Thus, we set $d_{ij} = \text{inf}$.
   - Otherwise, we can find d_{ij} by traversing the predecessor chain from j to i and summing up the weights of the edges along the path.

   Following is the pseudocode:

```
for i in 1..n:
    for j in 1..n:
        if i == j:
            d[i][j] = 0
        else if pi[i][j] == NIL:
            d[i][j] = +inf
        else:
```

```
            dist = 0
            k = j
            while k != i:
                p = pi[i][k]
                dist += w[p][k]
                k = p
            d[i][j] = dist
```

The time complexity is $O(V^3)$ since we have three nested loops in the worst case.

This answer is thought by myself and checked the correctness with ChatGPT.

5. Since any shortest path from u to v has all of its intermediate vertices in T, the path must take the form $u \to t_1 \to t_2 \to \cdots \to t_k \to v$, where $t_1, t_2, \ldots, t_k \in T$. If we mark t_k as t, the the path would be $u \rightsquigarrow t$ and $t \to v$. Thus, we can enumerate all vertices in T as the last intermediate vertex t before reaching v (i.e. $d(u,v) = \min_{t \in T}(w(u,v), d(u,t) + d(t,v))$).

   For each t. We can compute the shortest $d(t,v)$ and $d(t,u)$ (due to the graph is directed, we should construct a reverse graph to obtain it) with the SSSP algorithm (e.g. Dijkstra's). Note thay $|E| = \Omega(|V|^2)$, this step takes $O(|V|^2)$. We have $|T|$ intermediate vertices, so the overall time complexity is $O(|T| \cdot |V|^2)$. After this, the APSP distance can be calculated by $d(u,v) = \min_{t \in T}(w(u,v), d(u,t) + d(t,v))$. Given that $|T| = o(|V|)$, consequently, the overall time complexity is $o(|V|^3)$, which is better than the Floyd-Warshall algorithm.

   This answer is thought by myself and checked the correctness with ChatGPT.

6. We can convert this problem into a maximal flow problem. To seek the maximum number of vertex-disjoint paths from s to t, we can set the capacity of each vertex in the graph to 1. After that, we can apply the classic maximal flow algorithm (e.g. Edmonds-Karp) to find the maximum flow from s to t.

   However, in a classic maximal flow problem, the capacity is assigned to edges instead of vertices. To tackle this, we can split each vertex $v \in V \backslash \{s,t\}$ into two vertices v_{in} and v_{out}. All incoming edges of v will be connected to v_{in}, and all outgoing edges of v will be connected from v_{out}. Finally, we add an edge from v_{in} to v_{out} with a capacity of 1 to enforce the vertex capacity constraint. After transforming the graph in this way, we can run the Edmonds-Karp algorithm to find the maximum flow from s_{out} to t_{in}.

   Since each vertex has a capacity of 1, the maximum flow value will be equal to the maximum number of vertex-disjoint paths from s to t. The time complexity of the Edmonds-Karp algorithm is $O(VE^2)$. Suggested by ChatGPT, we can also use Dinic's algorithm to achieve a better time complexity of $O(\min(V^{2/3}, E^{1/2}) \cdot E)$.

   This answer is discussed with ChatGPT and refrerred to `https://www.geeksforgeeks.org/dsa/dinics-algorithm-maximum-flow/`.

7. Set $S_i = \{v \in V; d(s,v) \leq i\}$. The distance from s to t is at least k, thus, $(S_i, V \backslash S_i)$ is a s-t cut $\forall i \in [0, k-1]$.

   Consider each edge $(u,v) \in E$, it will contribute to at most one of the cuts, specifically, the cut $(S_i, V \backslash S_i)$ where $i = d(s,u)$. Since $d(s,v) \leq d(s,u) + 1$, thus, $u \in S_i$ and $v \notin S_i$. If $d(s,v) = d(s,u)$, then v is also in S_i and this edge does not contribute to any cut. Therefore, the total capacity of all these cuts is $\sum_{i=0}^{k-1} |E(S_i, V \backslash S_i)| \leq |E|$

   The result above represents that the sum of all k cuts' capacities is at most $|E|$. By the pigeonhole principle, at least one of these cuts must have a capacity of at most $\frac{|E|}{k}$. Thus, there exists an s-t cut with capacity no greater than $\frac{|E|}{k}$.

   This answer is discussed with ChatGPT.

8. The time complexity of Ford-Fulkerson is $O(|E||f^*|)$ where $|f^*|$ is the maximum flow value. The maximum flow is $|U|$, therefore the time complexity can also represented as $O(|E||U|)$.

   However, the capacity is recorded in binary form, the exact input of $O(|E||U|)$ has just $O(\log |U|)$ input. Which means, if $|U|$ is large (e.g. 2^n), the input size is only $O(n)$, but the time complexity is $O(|E|2^n)$, which is not polynomial in terms of the input size.

   For algorithm that won the best paper award in FOCS 2022, the time complexity is $O(|E|^{1+o(1)} \log |U|)$, which is polynomial in terms of the input size.

   This answer is discussed with ChatGPT.