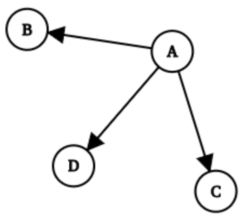
ADA HW #3 - Hand-Written

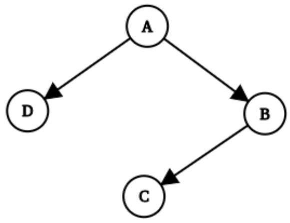
Student Name: 林楷恩
Student ID: b07902075

Problem D - Basic Problems in Graphs

1. • Same Order:

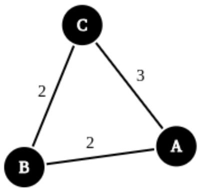


• Different Order:

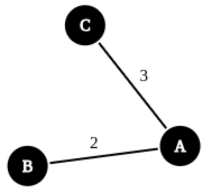


2. **Disproof by Counterexample:**

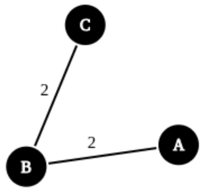
• Original Graph:



• Shortest Path Tree:



• Minimum Spanning Tree:



(source vertex is A)

3. (a) We can model this problem as **Single Source Shortest Path** problem, by spliting all vertex v into v_s (represents "start" visiting city v) and v_e (represents "end" visiting city v) with an edge weighted $c(v)$ connected them. Formally speaking, given V cities, capital v^*, E roads, and funtion $r(u, v)$ and $c(v)$, the graph G' and weight function w is constructed as follow:

$$V' = V_s \cup V_e$$
$$(v_s,\ v_e) \in E',\ w(v_s, v_e) = c(v),\ \forall\ v \in V$$
$$(u_e,\ v_s) \in E',\ w(u_e, v_s) = r(u, v),\ \forall\ (u, v) \in E$$
The source vertex is v_s^*

Since the roads would not form any cycle, G' would be a **Directed Acyclic Graph**. To solve single source shortest path problem on DAG, we first perform topological sorting on it, and then relax the edges in the topologically sorted order. The answer is the shortest path to all v_e in G'.

(b) i. **Time Complexity Analysis:**
- contruct G': $\Theta(V' + E')$
- topological sort: $\Theta(V' + E')$
- initializing: $\Theta(V')$
- relax each edge: $\Theta(V' + E')$

Given $|V'| = 2V = \Theta(V)$ and $|E'| = E + V = \Theta(V + E)$, the overall time complexity is:

$$\Theta(V + E) + \Theta(V) + \Theta(V + E) = \Theta(V + E)\ \#$$

ii. **Correctness Proof:** I will show that the equality between original problem and single source shortest path problem in G':

- because for every v_s, it has only one out-edge points to v_e, if we enter v_s, we must pass (v_s, v_e) and pay $w(v_s, v_e) = c(v)$.
- if we are at u_e, we must have passed u_s and pay $c(u)$. Then, we can go to v_s if we can reach v from u in the original graph, which means in the next step we must pass (v_s, v_e) and pay $c(v)$ to "visit" city v.

Thus, the 2 problem is exactly equivalent, so the algorithm is correct.

Problem E - Mailing Fees

(1) This problem can be modeled as a **Single Source Shortest Path** problem, where node 0 is the source vertex. For the "sending back" part, because the graph is undirected, so the shortest path from source to a vertex is equivalent to the shortest path from the vertex to source. Hence, I can run *Dijkstra's Algorithm* first to find the shortest distance to each factory and make them double, then they are the cost of a "round trip", which is the answer of the problem.

Mailing Fees I: Dijkstra

```
function Dijkstra(G, w):
    src ← 0
    // initializing
    for i from 1 to N − 1:
        dist[i] ← ∞
    dist[src] ← 0
    // initialize priority queue of 2−tuple,
    // which is ordered by the first element (distance)
    // with the minimum value on the top
    pq ← ∅
    INSERT(pq, (0, src))

    while pq is not empty:
        d, v ← EXTRACT_MIN(pq)
        // test if this element should be ignored
        if d ≠ dist[v]:
            continue
        // relax each edge
        for u in G.adj[v]:
            if d + w(v, u) < dist[u]:
                dist[u] ← d + w(v, u)
                INSERT(pq, (dist[u], u))

    // return the array that contains the shortest path to all vertices
    return dist

function solve(G, w):
    dist ← Dijkstra(G, w)
    for i from 0 to N − 1:
        dist[i] ← dist[i] ∗ 2
    return dist

ANSWER ← solve(G, w)
```

(2) Now the edge becomes directed, so a path from source to a vertex does not equal to a path from the vertex to source. To find the shortest path from each vertex to the source, I run *Dijkstra's Algorithm* on G^T, i.e. the graph that reverses the direction of every edge, because in G^T, a path from the source to a vertex is equivalent to a path from the vertex to the source in G.

The time complexity of the above implementation of *Dijkstra's Algorithm* is analyze as follow:

(a) initializing dist[]: $O(N)$

(b) initializing priority queue(binary min-heap): $O(1)$

(c) In every iteration of the while loop: $O(V\,log\,E) + O(E\,log\,E) = O(E\,log\,E)$

- the min-element is extracted from `pq`: $O(log\,E)$
- relax all edges leaves v, upon successfully relaxing, insert the better weight and node id to `pq`: $O(\textit{out-degree}(v) * log\,E)$

  * Note that the upper bound of the size of `pq` is E if every time we relax an edge, we insert a new element to `pq`.

....The overall complexity is $O(E\,log\,E)$

We run *Dijkstra's Algorithm* twice, one on G, one on G^T (can be contructed in $\Theta(E)$). And we need to add the 2 result together, which takes $O(N)$ trivially. Therefore, the overall time complexity of my algorithm is

$$\Theta(E) + 2 \times O(E\,log\,E) + O(N) = O(E\,log\,E)$$

(3) Now the edges can have negative weights, so I turn to *Bellman-Ford's Algorithm*, while *Dijkstra* cannot handle this scenario. The implementation is as follow:

Mailing Fees II: Bellman-Ford

```
function BellmanFord(G, w):
    src ← 0
    // initializing
    for i from 1 to N − 1:
        dist[i] ← ∞
    dist[src] ← 0
    // N − 1 rounds
    for i from 1 to N − 1:
        for (u, v) in G.E:
            if dist[u] + w(u, v) < dist[v]:
                dist[v] = dist[u] + w(u, v)

    // detect negative cycle
    for (u, v) in G.E:
        if dist[u] + w(u, v) < dist[v]:
            return FALSE

    return dist

function solve(G, w):
    result ← BellmanFord(G, w)
    if result is FALSE:
        print("I am rich!")
        return −1
    else:
        return dist

ANSWER ← solve(G, w)
```

The time complexity is analyzed as follow:

(a) initializing: $O(N)$

(b) $|V| - 1$ iterations of relaxation of all edges: $O(N * E)$

(c) detect negative cycle: $O(E)$

Thus, the overall time complexity is $O(N * E)$

Correctness: The ability to find the shortest path and detect negative cycle of *Bellman-Ford's Algorithm* is proved in the class.

(4) Here is a comparison table:
* NOTE: Let M_G be the space complexity to store the graph with adjacency lists, $M_G = O(V + E)$

	subproblem (2)	subproblem (3)
Algorithm	Dijkstra	Bellman-Ford
Time Complexity	$O(E \log E)$	$O(VE)$
Space Complexity	$O(V + E) + M_G$	$O(V) + M_G$
Advantages	better time complexity	can handle negative weights, can detect negative cycle
Disadvantages	cannot handle negative weights	worse time complexity

(5)
- **Algorithm Description:** I reweight the edges such that a "trustful cycle" is equal to a "negative cycle" in the graph. Let graph be $\langle V, E\rangle$the original weight function be w, and $R(v)$ denotes the reliability value of factory v, then the new weight function w' is defined as follow:

$$\forall\ (u, v) \in E,\ w'(u, v) = K \times w(u, v) - R(v)$$

Then I run *Bellman-Ford's algorithm* to take use of its capability of detecting negative cycle. If the return value is FALSE, which means there exists a negative cycle, then we know there exists a "trustful cycle".

- **Correctness Proof:** I show that a "trustful cycle" in G is equivalent to a "negative cycle" in G':

$$\begin{aligned} &\text{Let } C \text{ be a "trustful cycle" in } G \\ \Longrightarrow &\sum_{v\in C} R(v) \ \div \sum_{(u,v)\in C} w(u,v) > K \\ \Longrightarrow &K \times \sum_{(u,v)\in C} w(u,v) - \sum_{v\in C} R(v) < 0 \\ \Longrightarrow &\sum_{(u,v)\in C} K \times w(u,v) - R(v) < 0 \\ \Longrightarrow &\sum_{(u,v)\in C} w'(u,v) < 0 \end{aligned}$$

- **Time Complexity Analysis:**
  (a) reweighting: $O(E)$
  (b) Bellman-Ford: $O(V * E) = O(N * E)$
  Thus, the overall complexity is $O(E) + O(N * E) = O(N * E)$ #

Problem F - Gaussian is Too Slow

I model this problem as a *Minimum Spanning Tree* problem. Construct a undirected graph $G = \langle V, E \rangle$ and the weight function w as follow:

$$\begin{aligned} V &= \{v_0, v_1, v_2, \dots, v_N\} \\ E &= \{(v_i, v_j) \mid 0 \leq i < j \leq N\} \\ w(v_i, v_j) &= \text{the cost of } x_{i+1} + x_{i+2} + \cdots + x_j, 0 \leq i < j \leq N \end{aligned}$$

After constructing the graph, run *Prim's algorithm* on it, then the total weight of result minimum spanning tree is the answer. I will prove the correctness and time complexity of the algorithm in the following.

- **Proof of Correctness:** (For simplicity, let $f_{i,j} = x_i + x_{i+1} + \cdots + x_j$ in the following)

  Claim: If there exist a simple path from v_i to v_j, WLOG assume $i < j$, then we can obtain $f_{i+1,j} = x_{i+1} + x_{i+2} + \cdots + x_j$ by adding or substracting the equations corresponded to the edges in the path.

  Proof of Claim: Let the number of edges in the simple path be k, prove by induction on k:

  - **Base Case**: $k = 1$, trivial, we only have one edge and one equation, and the euqation is just what we want.
  - **Inductive Step**: Assume the claim is correct when $k = m$, then I show that it is also correct when $k = m + 1$. If we have a simple from v_i to v_j that consists of $m + 1$ edges, then the path can be split into 2 parts, one from v_i to some other node v_x that has m edges and another from v_x to v_j has 1 edges. By induction hypothesis, we can obtain $f_{i,x}$ if $i < x$ or $f_{x,i}$ if $x < i$:
    - ∗ $i < x$:
      - · $j < x$: we have $f_{i+1,x}$ and $f_{j+1,x} \implies f_{i+1,x} - f_{j+1,x} = f_{i+1,j}$
      - · $j > x$: we have $f_{i+1,x}$ and $f_{x+1,j} \implies f_{i+1,x} + f_{x+1,j} = f_{i+1,j}$
    - ∗ $i > x$:
      - · $j < x$: in this case $j < x < i$, which contradicts $i < j$, so it is impossible
      - · $j > x$: we have $f_{x+1,i}$ and $f_{x+1,j} \implies f_{x+1,j} - f_{x+1,i} = f_{i+1,j}$

    By induction, the claim is true for all $k \geq 1$.

  By claim, if a set of edges $T \subseteq E$, such that for all $i \in \{1, \dots, N\}$, there exists a simple path from v_{i-1} to v_i (i.e. all nodes are connected), then we can obtain $\{x_1, x_2, \dots, x_N\}$ by adding or substracting the equations corresponded to the edges, which means each variable is solvable. Furtheremore, a minimum spanning tree of G makes G connected with the total weight of edges in tree minimized. Hence, an algorithm for *Minimum Spanning Tree* like *Prim's* outputs the optimal answer of the original problem.

- **Time Complexity Analysis:** Since in this case, $|E| = C_2^{N+1} = \frac{N(N+1)}{2} = \Theta(N^2) \gg |V| = N + 1 = \Theta(N)$, we implement the priority queue in *Prim's algorithm* by an array indexed by the id of nodes, such that the operations have the following complexity:

  - extract-min: $O(|V|)$
  - insert: $O(1)$
  - decrease-key: $O(1)$

  Then *Prim's algorithm* run in $|V| \times O(|V|) + O(|E|) \times O(1) = O(|E|) = O(N^2)$ time complexity.