Commit 1: finish lab5_ortools.cc

Commit 2: finish lab5_myalgo.cc

Commit 3: finish all tasks and report 並修改變數名稱(跟 spec 一樣)

*原本多限制了一條：一個 link 只能被用一次，後來刪除了(Commit 3)，變成可以給不同 SD pair 使用

Task 1: Complete the ILP model

Y-Z binding 當某 SD pair 使用了邊 (u, v)時，該邊必須被標記為佔用

$$Y_{f,u,v} \le Z_{u,v} \quad , \quad \forall f \in F \, , \quad (u,v) \in E$$

Single transmitter per node 要求所有從 u 出發且被分配使用的邊之和<=1

$$\sum_{v \in u} z_{u,v} \le 1 \quad , \quad \forall u \in U \, , \, \forall (u,v) \in E$$

Single receiver per node 要求所有從 u 接收且被分配使用的邊之和<=1

$$\sum_{v \in u} z_{v,u} \le 1 \quad , \quad \forall u \in U \, , \, \forall (u,v) \in E$$

Task 2: OR-Tools Program

讀取 U，E，F、讀取物理無向邊並轉換成有向邊

```
16  struct Edge {
17    int u;
18    int v;
19    double cap;
20    int pair_id;
21  };
```

```
32    for (int test = 0; test < t; test++) {
33      int U, E, F;
34      std::cin >> U >> E >> F;
35
36      // 儲存物理邊原始順序（每筆 undirected 邊）
37      // 並同時建立兩條 directed 物理邊 (pair_id = -1)
38      std::vector<std::tuple<int,int,double>> input_edges;
39      std::vector<Edge> edges;
40      for (int i = 0; i < E; i++) {
41        int u, v;
42        double cap;
43        std::cin >> u >> v >> cap;
44        input_edges.push_back({u, v, cap});
45        // 依讀入順序產生兩條有向邊（注意：兩條邊的順序固定：第一為 u->v, 第二為 v->u）
46        edges.push_back({u, v, cap, -1});
47        edges.push_back({v, u, cap, -1});
48      }
```

讀取 SD pair 並建立虛擬邊

```
    // 讀取 SD pair 並同時建立專屬虛擬邊：
    // 虛擬邊1：從虛擬源 S (編號 U) -> SD pair 的 source，容量設為 1e9，pair_id = i
    // 虛擬邊2：從 SD pair 的 destination -> 虛擬匯 D (編號 U+1)，容量設為 1e9，pair_id = i
    std::vector<std::pair<int,int>> sd_pairs;
    for (int i = 0; i < F; i++) {
      int src, dst;
      std::cin >> src >> dst;
      sd_pairs.push_back({src, dst});
      edges.push_back({U, src, 1e9, i});
      edges.push_back({dst, U + 1, 1e9, i});
    }
```

更新圖節點數與總邊數

```
    int total_nodes = U + 2;
    int N_edges = edges.size();
```

建立求解器與設定 Big-M

```
    MPSolver solver("lab5_ortools", MPSolver::CBC_MIXED_INTEGER_PROGRAMMING);
    double M_big = 1e9;
```

建立決策變數 (XYZ 的定義跟 spec 一樣)

```
  // Y[f][e] : SD pair f 是否採用邊 e (二元變數)
  std::vector<std::vector<const MPVariable*>> Y(F, std::vector<const MPVariable*>(N_edges, nullptr));
  for (int f = 0; f < F; f++) {
    for (int e = 0; e < N_edges; e++) {
      Y[f][e] = solver.MakeIntVar(0, 1, "Y_" + std::to_string(f) + "_" + std::to_string(e));
    }
  }
```

```
  // Z[e] : 邊 e 是否被使用 (只要有任一 SD pair 採用則為 1)
  std::vector<const MPVariable*> Z(N_edges, nullptr);
  for (int e = 0; e < N_edges; e++) {
    Z[e] = solver.MakeIntVar(0, 1, "Z_" + std::to_string(e));
  }
```

```
  // X[f] : SD pair f 的傳輸速率 (連續變數)
  // z[f] : SD pair f 是否啟用 (0 表示完全不傳輸)
  std::vector<const MPVariable*> X(F, nullptr);
  std::vector<const MPVariable*> z(F, nullptr);
  for (int f = 0; f < F; f++) {
    X[f] = solver.MakeNumVar(0.0, solver.infinity(), "X_" + std::to_string(f));
    z[f] = solver.MakeIntVar(0, 1, "z_" + std::to_string(f));
  }
```

目標函數：目標在於最大化所有 SD pair 的總傳輸速率(X[f]的總和)

```
    // ------------------- 目標函數 -------------------
    MPObjective* objective = solver.MutableObjective();
    for (int f = 0; f < F; f++) {
      objective->SetCoefficient(X[f], 1);
    }
    objective->SetMaximization();
```

設立 constraint：

Single path → 包含在 Single transmitter / receiver per node(也代表路不能分岔，一定是一進一出)

Flow conservation：

對於中間節點，流入必須等於流出，對於虛擬點，分別引入 zf 以激活 SD pair 的流

路徑結構是藉由二元變數 Yf,e來決定的（確定了「哪條路徑」被選中），而 SD pair 的流量由變數Xf控制，並通過 Link Capacity 限制將Xf與選中的路徑之瓶頸鏈路綁定。這種設計使得流量守恆約束專注於路徑連續性，而最終 Xf 的取值則受到後續約束控制。

```
106   for (int f = 0; f < F; f++) {
107     for (int v = 0; v < total_nodes; v++) {
108       MPConstraint* c = solver.MakeRowConstraint(0.0, 0.0, "flow_" + std::to_string(f) + "_" + std::
109       for (int e = 0; e < N_edges; e++) {
110         if (edges[e].u == v)
111           c->SetCoefficient(Y[f][e], 1);
112         if (edges[e].v == v)
113           c->SetCoefficient(Y[f][e], -1);
114       }
115       if (v == U) {          // S
116         c->SetCoefficient(z[f], -1);
117       } else if (v == U + 1) { // D
118         c->SetCoefficient(z[f], 1);
119       }
120     }
121   }
```

對於每個 SD pair f與每條邊 e，如果邊 e是虛擬邊且它的 pair_id 與 f 不相等，則強制 Yf,e=0，避免其它 SD pair 選用了這條屬於別人的虛擬邊。

```
123   // ------------------- 虛擬邊歸屬約束 -------------------
124   // 若邊 e 為虛擬邊 (edges[e].pair_id != -1) 且其 pair_id ≠ f，則要求 Y[f][e] = 0
125   for (int f = 0; f < F; f++) {
126     for (int e = 0; e < N_edges; e++) {
127       if (edges[e].pair_id != -1 && edges[e].pair_id != f) {
128         MPConstraint* c = solver.MakeRowConstraint(0, 0, "virtual_tag_" + std::to_string(f) + "_" + s
129         c->SetCoefficient(Y[f][e], 1);
130       }
131     }
132   }
```

Y – Z Binding：若某 SD pair f 選用了邊 e（Yf,e=1），則全局變數 Ze

必須被設置為 1，表示該邊被使用。

```
// ------------------- Y-Z binding -------------------
// 若 SD pair f 採用邊 e (Y[f][e] = 1)，則必須讓 Z[e] = 1
for (int f = 0; f < F; f++) {
  for (int e = 0; e < N_edges; e++) {
    MPConstraint* c = solver.MakeRowConstraint(-solver.infinity(), 0, "bind_" + std::to_string(f)
    c->SetCoefficient(Y[f][e], 1);
    c->SetCoefficient(Z[e], -1);
  }
}
```

Variable binding：控制 SD pair f 的分配流量 Xf 不超過經過物理鏈路 e 的容量

```
// ------------------- Link Capacity 限制 -------------------
// 對物理邊 (edges[e].pair_id == -1 且 cap < 1e8) 及每個 SD pair f，設 X[f] + (M_big - cap(e)) * Y[f]
for (int f = 0; f < F; f++) {
  for (int e = 0; e < N_edges; e++) {
    if (edges[e].pair_id == -1 && edges[e].cap < 1e8) {
      MPConstraint* c = solver.MakeRowConstraint(-solver.infinity(), M_big, "cap_" + std::to_string
      c->SetCoefficient(X[f], 1);
      c->SetCoefficient(Y[f][e], (M_big - edges[e].cap));
    }
  }
}
```

只有被啟用的 SD pair（zf=1）才可以有正流量 X

當 zf=0 時，約束變為 Xf≤0（由於 Xf 非負，因此 Xf 必為 0）。

當 zf=1 時，約束變成 Xf≤M，對於足夠大的 M 不構成限制。

```
// ------------------- Rate 與啟用綁定 -------------------
// 若 SD pair 未啟用 (z[f] == 0) 則其傳輸速率必須為 0： X[f] - M_big * z[f] ≤ 0
for (int f = 0; f < F; f++) {
  MPConstraint* c = solver.MakeRowConstraint(-solver.infinity(), 0, "rate_bind_" + std::to_string(
  c->SetCoefficient(X[f], 1);
  c->SetCoefficient(z[f], -M_big);
}
```

節點 v 的所有出向物理邊，其全局變數 Ze 的和必須不超過 1；同理所有入向物理邊也不超過 1。

```
// ------------------- 單一發射／接收限制 -------------------
// 對每個物理節點 v ∈ [0, U-1]，僅考慮物理邊，限制出向及入向各 ≤ 1
for (int v = 0; v < U; v++) {
  MPConstraint* c_out = solver.MakeRowConstraint(0, 1, "transmitter_" + std::to_string(v));
  MPConstraint* c_in  = solver.MakeRowConstraint(0, 1, "receiver_" + std::to_string(v));
  for (int e = 0; e < N_edges; e++) {
    if (edges[e].pair_id == -1) {
      if (edges[e].u == v)
        c_out->SetCoefficient(Z[e], 1);
      if (edges[e].v == v)
        c_in->SetCoefficient(Z[e], 1);
    }
  }
}
```

求解：

```cpp
// ------------------ 求解模型 ------------------
MPSolver::ResultStatus result_status = solver.Solve();
if (result_status != MPSolver::OPTIMAL) {
  std::cout << "No optimal solution found.\n";
  return 1;
}
```

輸出使用的物理有向連結：檢查物理邊->判斷使用狀態->輸出排序

```cpp
  std::vector<std::pair<int,int>> used_links;
  for (int e = 0; e < 2 * E; e++) {  // 物理邊儲存在 indices [0, 2*E-1]
    bool used = false;
    for (int f = 0; f < F; f++) {
      if (Y[f][e]->solution_value() > 0.5) {
        used = true;
        break;
      }
    }
    if (used)
      used_links.push_back({edges[e].u, edges[e].v});
  }
  // 按照邊的起點由小到大、起點相同再比終點由小到大排序
  std::sort(used_links.begin(), used_links.end(), [](const std::pair<int,int>& a, const std::pair<i
    return (a.first == b.first) ? (a.second < b.second) : (a.first < b.first);
  });

  std::cout << used_links.size() << "\n";
  for (auto &p : used_links) {
    std::cout << p.first << " " << p.second << "\n";
  }
```

重建並輸出每個 SD pair 的路徑：起點設定(對於每個 SD pair f，從虛擬源 S（編號 U）開始) → 依序尋找路徑(滿足 $Y[f][e]$ 的解值大於 0.5) → 停止條件(直至到達虛擬點 D) → 輸出格式

輸出總 Trough put 與平均 Trough put

```cpp
    for (int f = 0; f < F; f++) {
      int cur = U;  // 從 S 開始
      std::vector<int> full_path;
      full_path.push_back(cur);
      bool path_found = true;
      while (cur != U + 1) {
        bool found_edge = false;
        for (int e = 0; e < N_edges; e++) {
          if (edges[e].u == cur && Y[f][e]->solution_value() > 0.5) {
            cur = edges[e].v;
            full_path.push_back(cur);
            found_edge = true;
            break;
          }
        }
        if (!found_edge) {
          path_found = false;
          break;
        }
      }
```

```
        double rf = X[f]->solution_value();
        test_throughput += rf;

        if (!path_found || rf < 1e-9) {
          std::cout << "0 0\n";
        } else {
          // 去除虛擬端點 S 與 D
          std::vector<int> actual_path;
          for (size_t i = 1; i + 1 < full_path.size(); i++) {
            actual_path.push_back(full_path[i]);
          }
          std::ostringstream oss;
          oss << std::fixed << std::setprecision(0) << rf << " " << actual_path.size();
          for (size_t i = 0; i < actual_path.size(); i++) {
            oss << " " << actual_path[i];
          }
          std::cout << oss.str() << "\n";
        }
      }

      std::cout << std::fixed << std::setprecision(0) << test_throughput << "\n";
      overall_throughput += test_throughput;
    }
```

Task 3: Design & Implement MyAlgo

定義資料結構

```
// 讀入無向邊的結構
struct InputEdge {
    int u, v;
    double cap;
};
```

```
// 物理 directed 邊的結構
struct Edge {
    int u, v;       // directed edge: from u to v
    double cap;     // 邊容量（原始滿額值，用於 getCandidatePath 初步計算）
    int origIndex;  // 表示該邊所屬的原始 undirected 邊編號（用於更新剩餘容量及後續輸出）
    // 不再維護 used 狀態，允許同一條邊被多個 SD pair 共用
};
```

```
// SD pair 結構
struct SDPair {
    int src, dest;
};
```

```
// SD pair 結果
struct SDResult {
    double flow;          // 分配到的流量
    vector<int> path;     // 路徑上依序包含的節點（物理節點），若 flow==0 則 path 為空
};
```

```
// 用於 BFS 時記錄前驅資訊
struct NodeInfo {
    int parent;   // 前驅節點
    int edgeIdx;  // 從 parent 到當前節點所用的邊的索引（在 directed 邊陣列中的 index）
};
```

使用 BFS 在物理圖中尋找從 source 到 dest 的一條路徑：

建立 visited 陣列，初始化所有節點為「未訪問」。

將 source 節點加入 BFS queue，並標記為訪問。

每次取出 queue 的最前端節點 u，遍歷 u 的所有出向邊。

若 BFS 結束仍未找到通路，返回 false

```cpp
// BFS 函數：在物理網路中從 source 到 dest 搜尋一條有向路徑
// 注意：此 BFS 不檢查 transmitter/receiver 限制（會放到候選檢查中處理）
// U 為物理節點個數（節點編號在 [0, U-1]）
bool bfsPath(int U, int source, int dest,
             const vector<vector<int>> &adj,
             const vector<Edge> &pEdges,
             vector<NodeInfo> &pred) {
    int n = U;
    vector<bool> visited(n, false);
    queue<int> q;
    q.push(source);
    visited[source] = true;
    pred.assign(n, {-1, -1});

    while (!q.empty()){
        int u = q.front();
        q.pop();
        for (int edgeIdx : adj[u]) {
            int v = pEdges[edgeIdx].v;
            if (!visited[v]) {
                visited[v] = true;
                pred[v] = {u, edgeIdx};
                if (v == dest)
                    return true;
                q.push(v);
            }
        }
    }
    return false;
}
```

找出 BFS 搜索的路徑，並計算該路徑上的瓶頸流量（最小容量）：若 bfsPath() 找到路徑，則開始回溯 pred 陣列，重建完整路徑

1. 從 dest 回溯到 source，沿途記錄節點與邊索引。
2. 反轉節點與邊序列，確保符合 BFS 方向（由 source 到 dest）。
3. 依據重建路徑，計算瓶頸流量（沿候選路徑取最小邊容量）。

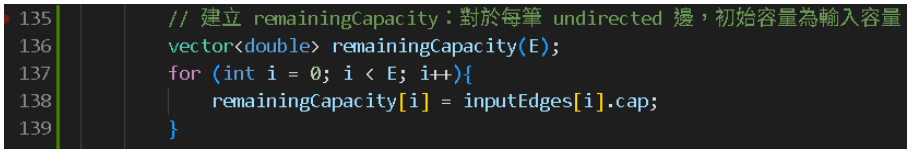
```cpp
// 取得從 source 到 dest 的候選路徑，返回候選路徑上 directed 邊的索引序列與經過節點序列，
// 並計算該路徑的初步"潛在流量"，其為所有邊的原始容量最小值（之後會根據剩餘容量重新評估）。
bool getCandidatePath(int U, int source, int dest, const vector<vector<int>> &adj, cons
    vector<NodeInfo> pred;
    bool found = bfsPath(U, source, dest, adj, pEdges, pred);
    if (!found)
        return false;

    // 重建路徑（從 dest 回到 source）
    pathNodes.clear();
    pathEdges.clear();
    int cur = dest;
    while(cur != source) {
        pathNodes.push_back(cur);
        int eIdx = pred[cur].edgeIdx;
        pathEdges.push_back(eIdx);
        cur = pred[cur].parent;
    }
    pathNodes.push_back(source);
    reverse(pathNodes.begin(), pathNodes.end());
    reverse(pathEdges.begin(), pathEdges.end());

    // 初步計算：沿候選路徑上所有 directed 邊的原始容量最小值
    potentialFlow = numeric_limits<double>::max();
    for (int idx : pathEdges)
        potentialFlow = min(potentialFlow, pEdges[idx].cap);

    return true;
}
```

Main：採用 greedy 策略分配 SD pair 的路徑：

讀取無向邊，並對每條邊產生雙向的邊

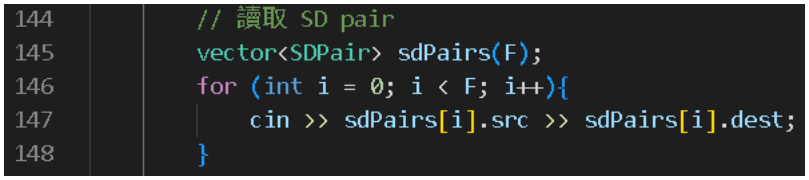
```
for (int test = 0; test < t; test++){
    int U, E, F;
    cin >> U >> E >> F;

    // 讀取物理無向邊（輸入順序）
    vector<InputEdge> inputEdges(E);
    for (int i = 0; i < E; i++){
        cin >> inputEdges[i].u >> inputEdges[i].v >> inputEdges[i].cap;
    }

    // 產生 directed 物理邊（2*E 條），記錄每筆邊所屬的原始 undirected 邊編號
    vector<Edge> pEdges;
    for (int i = 0; i < E; i++){
        // u -> v
        pEdges.push_back({inputEdges[i].u, inputEdges[i].v, inputEdges[i].cap, i});
        // v -> u
        pEdges.push_back({inputEdges[i].v, inputEdges[i].u, inputEdges[i].cap, i});
    }
    int numPhysicalEdges = pEdges.size(); // = 2*E
```

追蹤 每條物理無向邊的剩餘容量，確保不會超過 cap。



```
// 建立 remainingCapacity：對於每筆 undirected 邊，初始容量為輸入容量
vector<double> remainingCapacity(E);
for (int i = 0; i < E; i++){
    remainingCapacity[i] = inputEdges[i].cap;
}
```

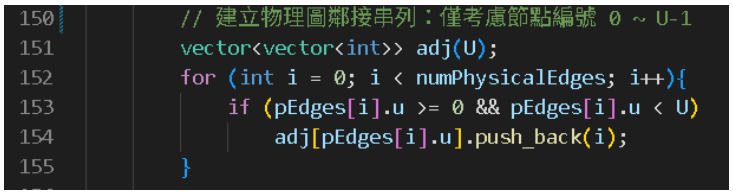
讀取 SD piars



```
// 讀取 SD pair
vector<SDPair> sdPairs(F);
for (int i = 0; i < F; i++){
    cin >> sdPairs[i].src >> sdPairs[i].dest;
}
```

建立物理網路的鄰接串列，以便 BFS 搜索：

逐一遍歷 numPhysicalEdges，確保該邊的起點 u 是有效物理節點（不包括虛擬節點），將該 directed 邊 (i) 加入其起點 u 的鄰接串列 → adj 陣列中存放了每個節點的所有出向邊。



```
// 建立物理圖鄰接串列：僅考慮節點編號 0 ~ U-1
vector<vector<int>> adj(U);
for (int i = 0; i < numPhysicalEdges; i++){
    if (pEdges[i].u >= 0 && pEdges[i].u < U)
        adj[pEdges[i].u].push_back(i);
}
```

Greedy：

初始化變數

```
168         // Greedy loop：當還有未分配 SD pair 存在時
169         while (true) {
170             double bestFlow = 0.0;
171             int bestIdx = -1;
172             vector<int> bestPathEdges, bestPathNodes;
```

遍歷所有 SD pair (若 assigned[i] == true，代表該 SD pair 已被分配，則跳過。)

設定來源與目的

使用 BFS 搜尋 SD pair 的候選路徑：回溯 BFS 結果，重建完整路徑、計算瓶頸流量(potFlow)

```
174             // 嘗試所有未分配的 SD pair
175             for (int i = 0; i < F; i++){
176                 if (assigned[i]) continue;
177                 int src = sdPairs[i].src, dest = sdPairs[i].dest;
178                 vector<int> candEdges, candNodes;
179                 double potFlow;
180                 if (getCandidatePath(U, src, dest, adj, pEdges, candEdges, candNodes, potFlow)) {
181                     // 重新計算候選路徑實際可用的流量，根據剩餘容量更新
```

重新計算可用流量

availableFlow：該候選路徑的 實際可分配瓶頸流量（受剩餘容量影響）。

遍歷候選路徑中的每條 directed 邊：找出 該 directed 邊所對應的 undirected 邊索引 (origIndex)、取候選路徑上的剩餘容量的最小值，確保不超過。

```
182                     double availableFlow = numeric_limits<double>::max();
183                     bool candidateFeasible = true;
184                     for (int edgeIdx : candEdges) {
185                         int undirected = pEdges[edgeIdx].origIndex;
186                         availableFlow = min(availableFlow, remainingCapacity[undirected]);
```

檢查 Transmitter/Receiver 限制

如果該 undirected 邊尚未被使用，則檢查：

該邊的起點 (u) 是否已被 另一條新分配的邊佔用 transmitter (usedOutgoing[u])？

該邊的終點 (v) 是否已被 另一條新分配的邊佔用 receiver (usedIncoming[v])？

若違反限制，則 標記該候選不可行並停止檢查

```
                    // 若此邊尚未被使用，則要求該物理節點的 transmitter/receiver尚未被佔用
                    if (! (remainingCapacity[undirected] < pEdges[edgeIdx].cap)) {
                        int u = pEdges[edgeIdx].u;
                        int v = pEdges[edgeIdx].v;
                        if (usedOutgoing[u] || usedIncoming[v]) {
                            candidateFeasible = false;
                            break;
                        }
                    }
                }
```

計算候選路徑可分配流量：

瓶頸容量必須符合 potFlow（候選路徑原始瓶頸流量） 和 availableFlow（剩餘容量）。

更新最佳候選：

若目前找到的流量比 bestFlow 更大，則更新該 SD pair 為最佳候選。

```
                    // 可分配的流量取兩者最小值
                    double candFlow = min(potFlow, availableFlow);

                    if (candFlow > bestFlow) {
                        bestFlow = candFlow;
                        bestIdx = i;
                        bestPathEdges = candEdges;
                        bestPathNodes = candNodes;
                    }
```

分配最佳候選 SD pair

若 bestIdx == -1 或 bestFlow ≈ 0，則結束分配迴圈。

記錄選中的 SD pair

累計本測試案例的 through put

標記該 SD pair 已分配

```
                if (bestIdx == -1 || bestFlow < 1e-9)
                    break; // 沒有找到可分配的 SD pair

                // 記錄該 SD pair 的分配結果
                sdResults[bestIdx].flow = bestFlow;
                sdResults[bestIdx].path = bestPathNodes;
                testThroughput += bestFlow;
                assigned[bestIdx] = true;
```

沿候選路徑所有 directed 邊，更新剩餘容量

若該邊是第一次使用，則更新該節點的 Transmitter/Receiver 狀態

記錄該 directed 邊已被使用，供最終輸出使用的物理連結

```
            // 更新候選路徑上所有邊的剩餘容量及節點 transmitter/receiver 狀態
            // 遍歷候選路徑中的每一個 directed 邊
            for (int edgeIdx : bestPathEdges) {
                int undirected = pEdges[edgeIdx].origIndex;
                // 扣除本次分配的流量
                remainingCapacity[undirected] -= bestFlow;
                // 若這筆 undirected 邊第一次被使用（即仍有剩餘且此前為未使用），更新節點狀態
                if (remainingCapacity[undirected] + bestFlow == pEdges[edgeIdx].cap) {
                    usedOutgoing[pEdges[edgeIdx].u] = true;
                    usedIncoming[pEdges[edgeIdx].v] = true;
                }
                edgeOutputUsed[edgeIdx] = true; // 此 directed 邊被使用過
            }
        }
```

最後輸出用到的 link 跟每個 path

計算平均 throughput 並輸出

Questions

1. Write down the 3 constraints you add in task 1 and briefly explain it
   Task 1 那裡有講了
2. Calculate the average throughput ratio between network.myalgo.out and network.ortools.out
   100.400000/188.000000=0.534...

3. Briefly explain the main idea of MyAlgo

在滿足所有的 constraint 下，遍歷所有未分配的 SD pair，嘗試尋找最佳候選路徑（使用 BFS 搜索）。在每個 pair 找出來的路徑中找流量最大的 (greedy)，選擇那條並更新圖，再繼續下一輪直到所有 SD pair 被分配或無可用路徑

4. Analyze the time complexity of MyAlgo
   BFS：O(U + 2E) → 遍歷所有 SD pair：O(F × (U + 2E))
   + 檢查候選路徑、分配 SD pair：O(U)
   $O(F \times (U+2E+U)) = O(F \times (U+2E)) \fallingdotseq O(F \times E)$