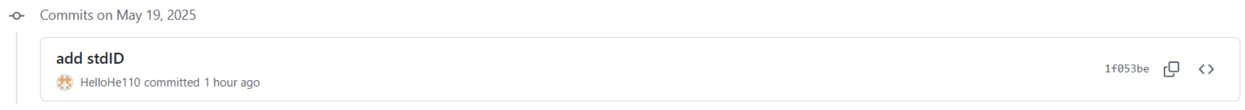
Report

<commit history>

1st commit:



Commits on May 19, 2025

add stdID

HelloHe110 committed 1 hour ago

1f053be

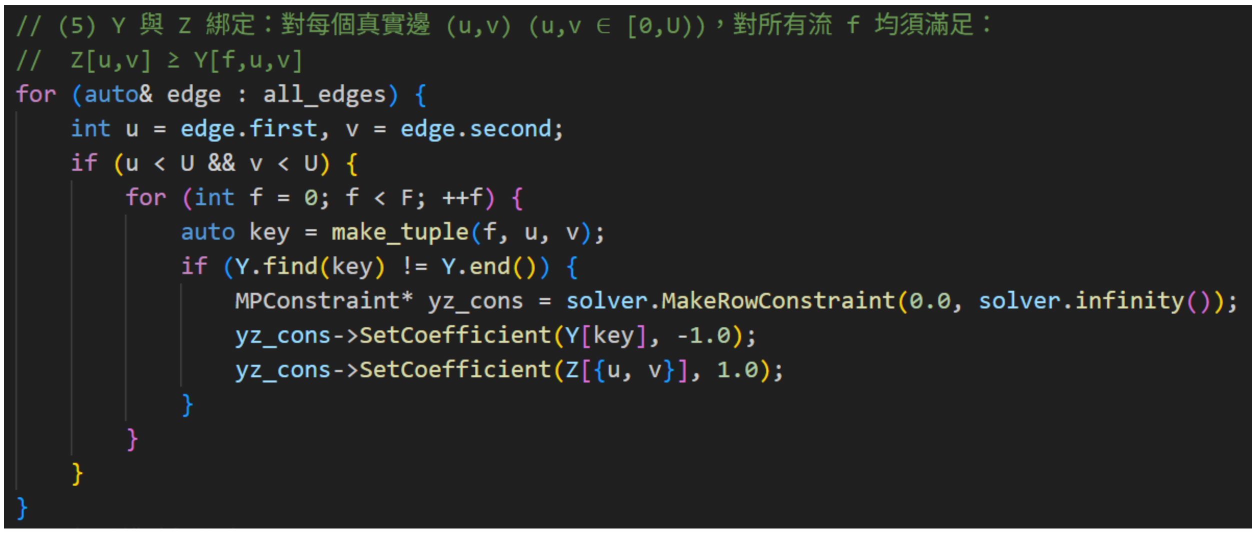
The first commit is about to upload the student ID file.

2nd commit:

The second commit is about the support task1, task2 and task3.

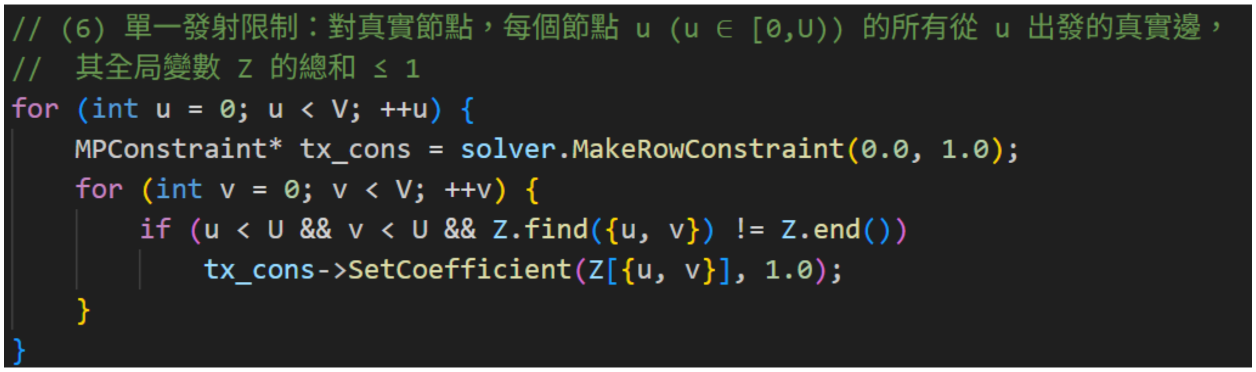
Task1: ILP model.

$$Y - Z\ binding: Z_{u,v} \geq Y_{f,u,v}, \quad \forall f \in F, (u,v) \in E$$



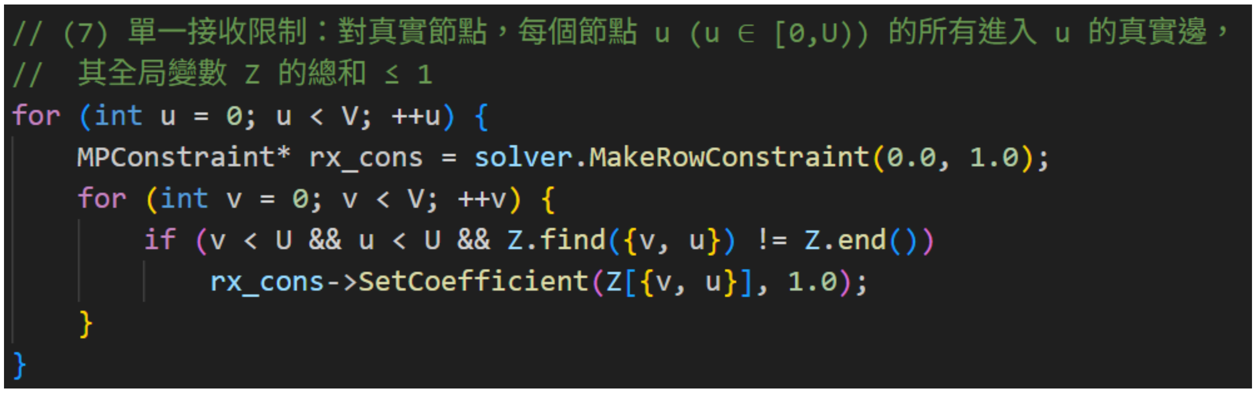
```
// (5) Y 與 Z 綁定：對每個真實邊 (u,v) (u,v ∈ [0,U))，對所有流 f 均須滿足：
//  Z[u,v] ≥ Y[f,u,v]
for (auto& edge : all_edges) {
    int u = edge.first, v = edge.second;
    if (u < U && v < U) {
        for (int f = 0; f < F; ++f) {
            auto key = make_tuple(f, u, v);
            if (Y.find(key) != Y.end()) {
                MPConstraint* yz_cons = solver.MakeRowConstraint(0.0, solver.infinity());
                yz_cons->SetCoefficient(Y[key], -1.0);
                yz_cons->SetCoefficient(Z[{u, v}], 1.0);
            }
        }
    }
}
```

$$Single\ transmitter: \sum_{v \in U} Z_{u,v} \leq 1, \quad \forall u \in U, \forall (u,v) \in E$$



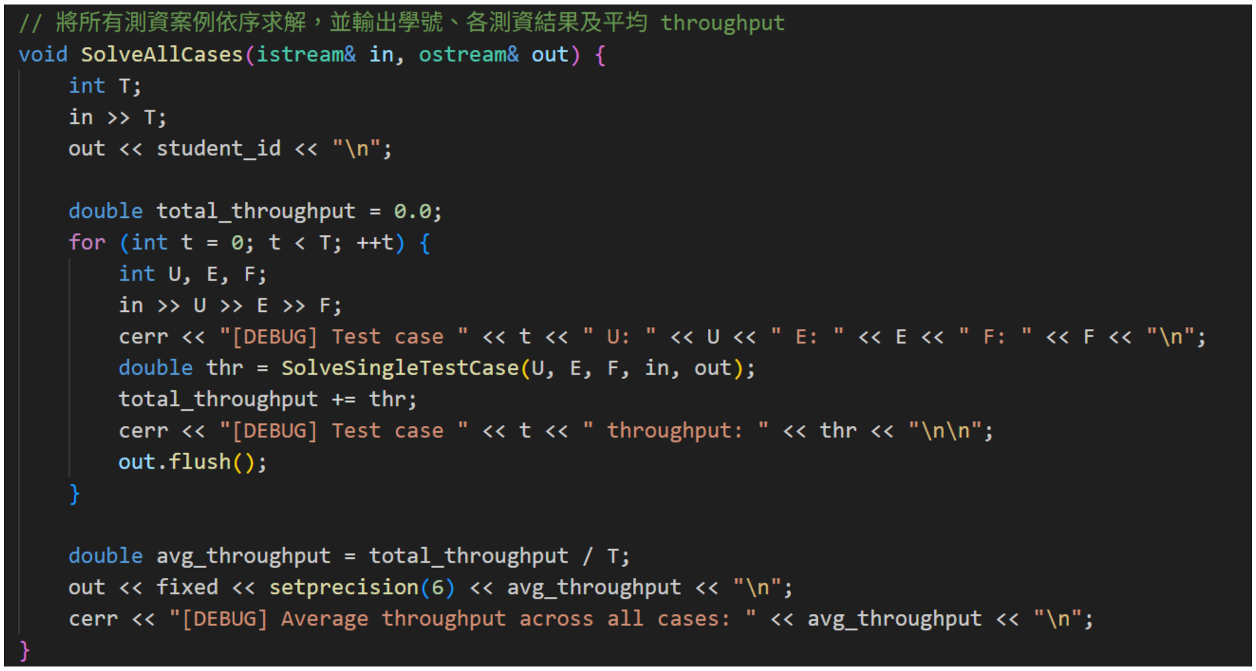
```
// (6) 單一發射限制：對真實節點，每個節點 u (u ∈ [0,U)) 的所有從 u 出發的真實邊，
//  其全局變數 Z 的總和 ≤ 1
for (int u = 0; u < V; ++u) {
    MPConstraint* tx_cons = solver.MakeRowConstraint(0.0, 1.0);
    for (int v = 0; v < V; ++v) {
        if (u < U && v < U && Z.find({u, v}) != Z.end())
            tx_cons->SetCoefficient(Z[{u, v}], 1.0);
    }
}
```

$$\mathit{single\ reciever}\colon \sum_{v \in U} Z_{v,u} \leq 1,\ \ \forall u \in U, \forall (v,u) \in E$$



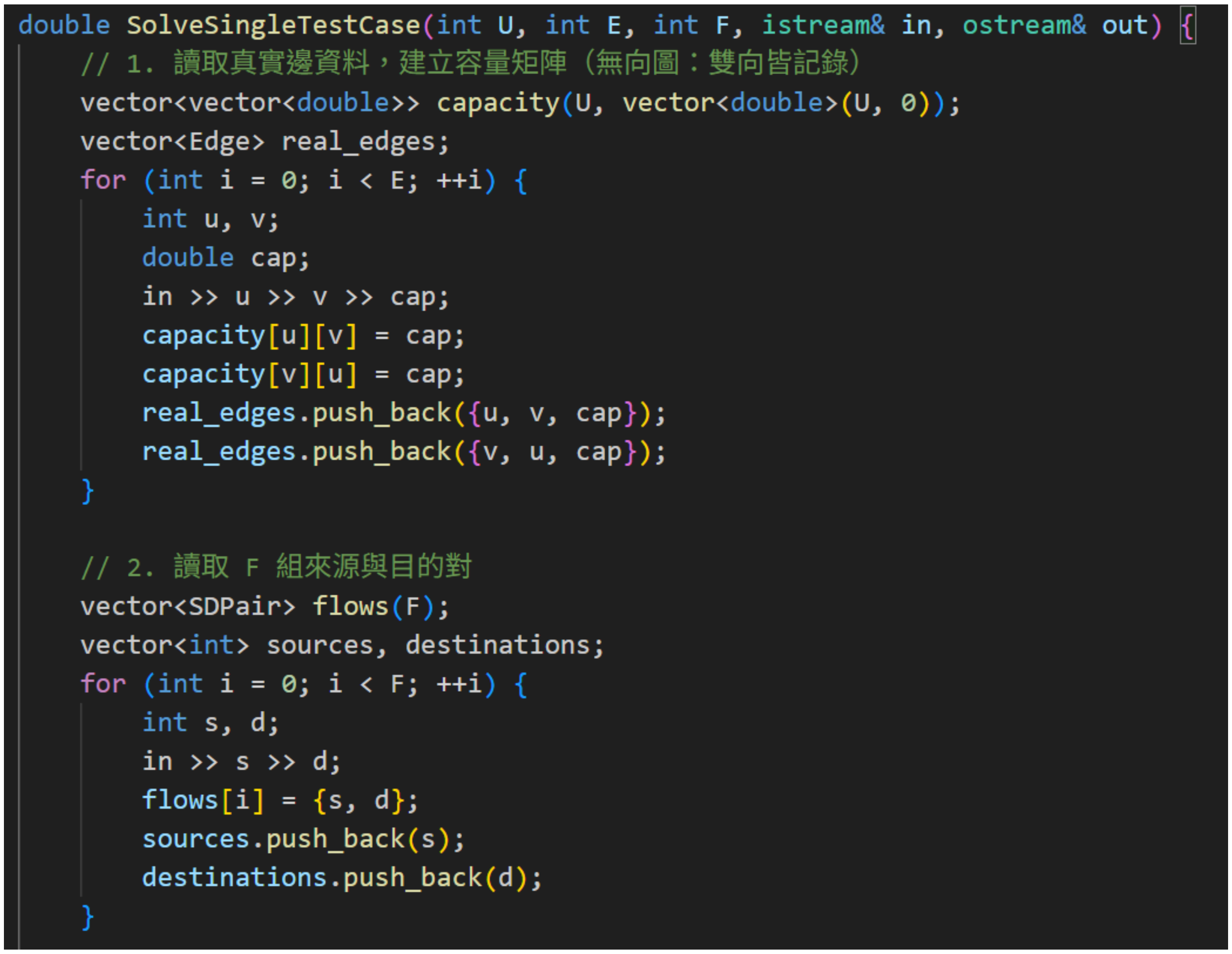
```
// (7) 單一接收限制：對真實節點，每個節點 u (u ∈ [0,U)) 的所有進入 u 的真實邊，
//  其全局變數 Z 的總和 ≤ 1
for (int u = 0; u < V; ++u) {
    MPConstraint* rx_cons = solver.MakeRowConstraint(0.0, 1.0);
    for (int v = 0; v < V; ++v) {
        if (v < U && u < U && Z.find({v, u}) != Z.end())
            rx_cons->SetCoefficient(Z[{v, u}], 1.0);
    }
}
```

Task2: OR-tools program



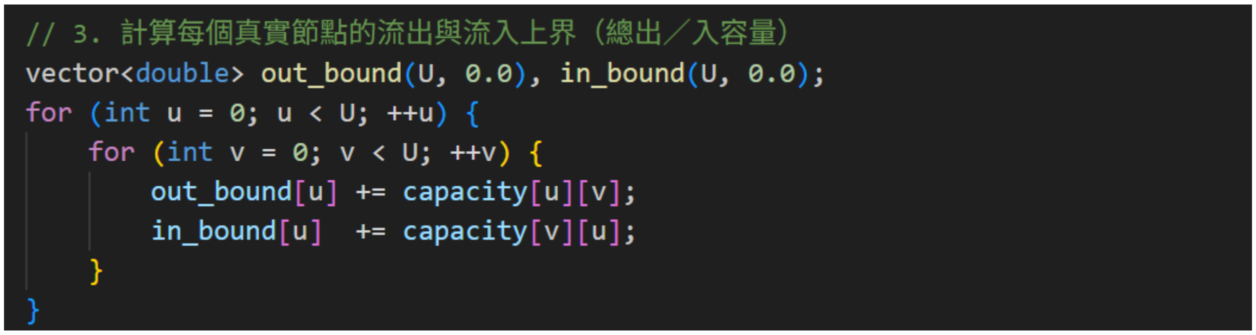
```
// 將所有測資案例依序求解，並輸出學號、各測資結果及平均 throughput
void SolveAllCases(istream& in, ostream& out) {
    int T;
    in >> T;
    out << student_id << "\n";

    double total_throughput = 0.0;
    for (int t = 0; t < T; ++t) {
        int U, E, F;
        in >> U >> E >> F;
        cerr << "[DEBUG] Test case " << t << " U: " << U << " E: " << E << " F: " << F << "\n";
        double thr = SolveSingleTestCase(U, E, F, in, out);
        total_throughput += thr;
        cerr << "[DEBUG] Test case " << t << " throughput: " << thr << "\n\n";
        out.flush();
    }

    double avg_throughput = total_throughput / T;
    out << fixed << setprecision(6) << avg_throughput << "\n";
    cerr << "[DEBUG] Average throughput across all cases: " << avg_throughput << "\n";
}
```

This part is to read in the first line to know how many test cases are there, and to solve each testcase one by one by function $SolveSingleTestCase$.



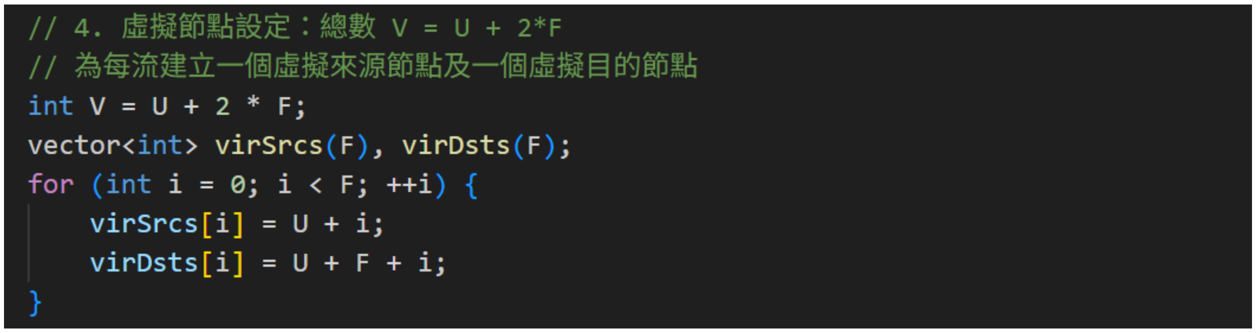
```cpp
double SolveSingleTestCase(int U, int E, int F, istream& in, ostream& out) {
    // 1. 讀取真實邊資料，建立容量矩陣（無向圖：雙向皆記錄）
    vector<vector<double>> capacity(U, vector<double>(U, 0));
    vector<Edge> real_edges;
    for (int i = 0; i < E; ++i) {
        int u, v;
        double cap;
        in >> u >> v >> cap;
        capacity[u][v] = cap;
        capacity[v][u] = cap;
        real_edges.push_back({u, v, cap});
        real_edges.push_back({v, u, cap});
    }

    // 2. 讀取 F 組來源與目的對
    vector<SDPair> flows(F);
    vector<int> sources, destinations;
    for (int i = 0; i < F; ++i) {
        int s, d;
        in >> s >> d;
        flows[i] = {s, d};
        sources.push_back(s);
        destinations.push_back(d);
    }
```

In $SolveSingleTestcase$, the first thing is to read in all the edges and flows (SD pair).



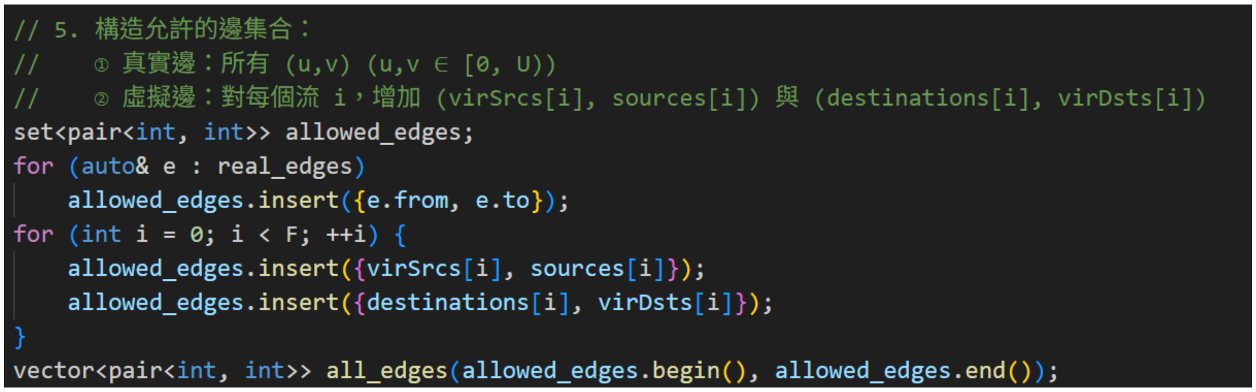
```cpp
// 3. 計算每個真實節點的流出與流入上界（總出／入容量）
vector<double> out_bound(U, 0.0), in_bound(U, 0.0);
for (int u = 0; u < U; ++u) {
    for (int v = 0; v < U; ++v) {
        out_bound[u] += capacity[u][v];
        in_bound[u]  += capacity[v][u];
    }
}
```

After that, due to the fact that I need to build virtual node and virtual edge, I calculated the upper-bound of (incoming / outgoing) capacity of each real node.



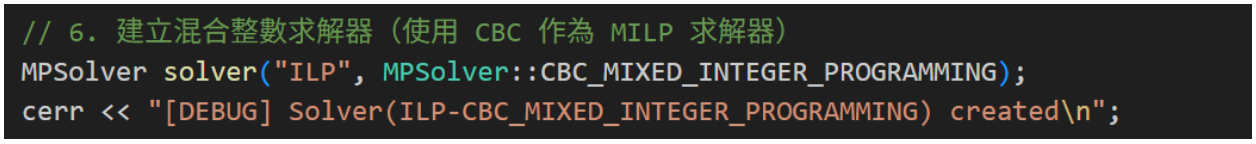
```
// 4. 虛擬節點設定：總數 V = U + 2*F
// 為每流建立一個虛擬來源節點及一個虛擬目的節點
int V = U + 2 * F;
vector<int> virSrcs(F), virDsts(F);
for (int i = 0; i < F; ++i) {
    virSrcs[i] = U + i;
    virDsts[i] = U + F + i;
}
```

Now, I create the virtual node by assigning these virtual nodes with $virSrc's\ idx \in [U, U + F)$ and $virDst's\ idx \in [U + F, U + 2F)$



```
// 5. 構造允許的邊集合：
//    ① 真實邊：所有 (u,v) (u,v ∈ [0, U))
//    ② 虛擬邊：對每個流 i，增加 (virSrcs[i], sources[i]) 與 (destinations[i], virDsts[i])
set<pair<int, int>> allowed_edges;
for (auto& e : real_edges)
    allowed_edges.insert({e.from, e.to});
for (int i = 0; i < F; ++i) {
    allowed_edges.insert({virSrcs[i], sources[i]});
    allowed_edges.insert({destinations[i], virDsts[i]});
}
vector<pair<int, int>> all_edges(allowed_edges.begin(), allowed_edges.end());
```

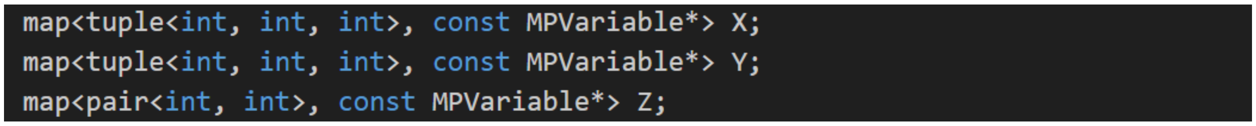
After creating the virtual nodes, I create both virtual edges and the actual edges (directed). In order to prevent duplication of edge, I applied data structures – set at first and then convert it into vector.



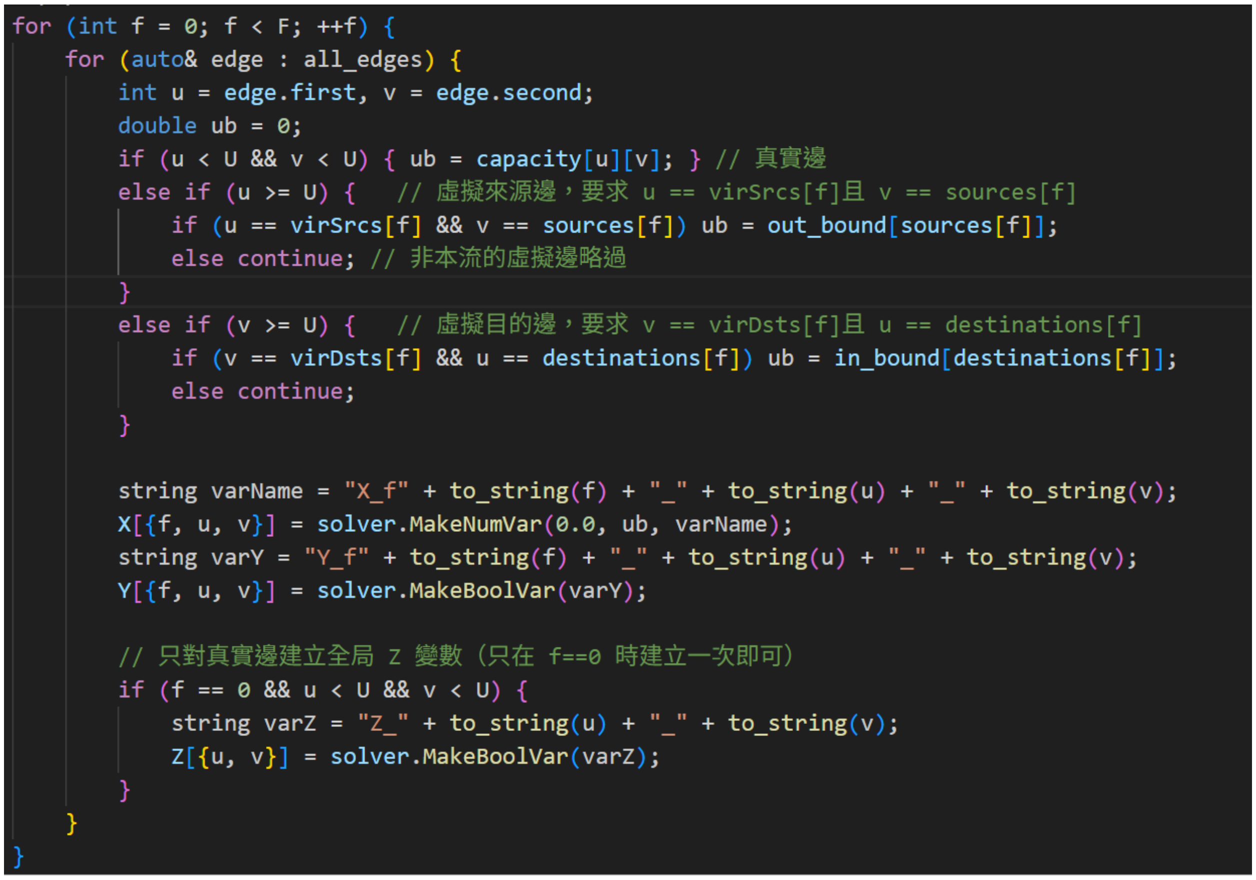
```
// 6. 建立混合整數求解器 (使用 CBC 作為 MILP 求解器)
MPSolver solver("ILP", MPSolver::CBC_MIXED_INTEGER_PROGRAMMING);
cerr << "[DEBUG] Solver(ILP-CBC_MIXED_INTEGER_PROGRAMMING) created\n";
```

And then, I solve it by calling solver(ILP-CBC_MIXED_INTEGER_PROGRAMMING)

Now, it's time to create variables for the solver based on the spec TA provided:

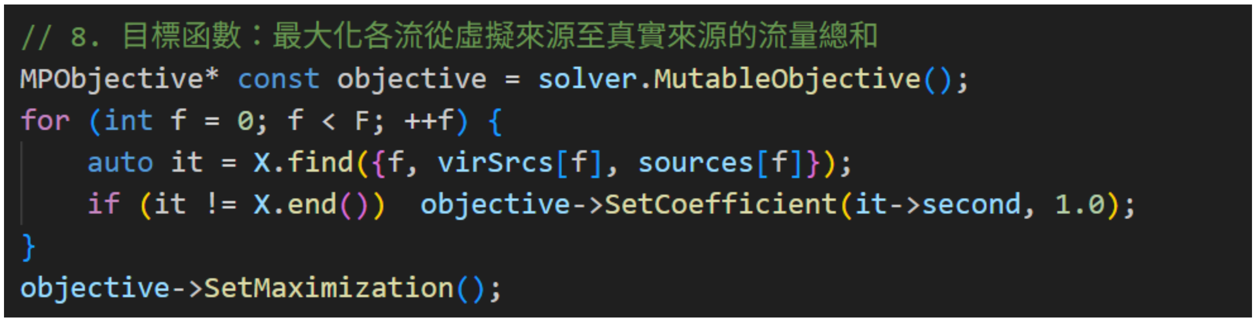


```
map<tuple<int, int, int>, const MPVariable*> X;
map<tuple<int, int, int>, const MPVariable*> Y;
map<pair<int, int>, const MPVariable*> Z;
```



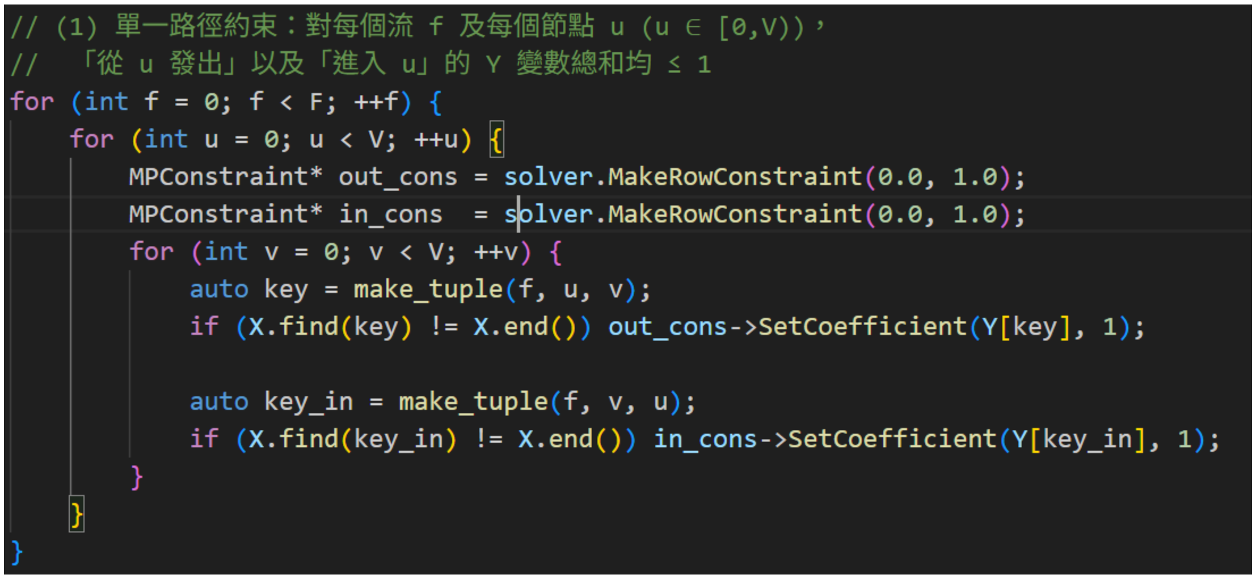
```cpp
for (int f = 0; f < F; ++f) {
    for (auto& edge : all_edges) {
        int u = edge.first, v = edge.second;
        double ub = 0;
        if (u < U && v < U) { ub = capacity[u][v]; } // 真實邊
        else if (u >= U) {   // 虛擬來源邊，要求 u == virSrcs[f]且 v == sources[f]
            if (u == virSrcs[f] && v == sources[f]) ub = out_bound[sources[f]];
            else continue; // 非本流的虛擬邊略過
        }
        else if (v >= U) {   // 虛擬目的邊，要求 v == virDsts[f]且 u == destinations[f]
            if (v == virDsts[f] && u == destinations[f]) ub = in_bound[destinations[f]];
            else continue;
        }

        string varName = "X_f" + to_string(f) + "_" + to_string(u) + "_" + to_string(v);
        X[{f, u, v}] = solver.MakeNumVar(0.0, ub, varName);
        string varY = "Y_f" + to_string(f) + "_" + to_string(u) + "_" + to_string(v);
        Y[{f, u, v}] = solver.MakeBoolVar(varY);

        // 只對真實邊建立全局 Z 變數（只在 f==0 時建立一次即可）
        if (f == 0 && u < U && v < U) {
            string varZ = "Z_" + to_string(u) + "_" + to_string(v);
            Z[{u, v}] = solver.MakeBoolVar(varZ);
        }
    }
}
```

This section defines three types of variables for network flow optimization. $X[f, u, v]$ is a continuous variable representing flow f along edge (u, v), with upper bounds determined by edge type. $Y[f, u, v]$ is a binary variable indicating whether flow f uses edge (u, v). $Z[u, v]$ is a binary variable created only for real edges when $f\ ==\ 0$, ensuring unique transmission or reception at nodes. The algorithm iterates through edges to assign appropriate variables and constraints.



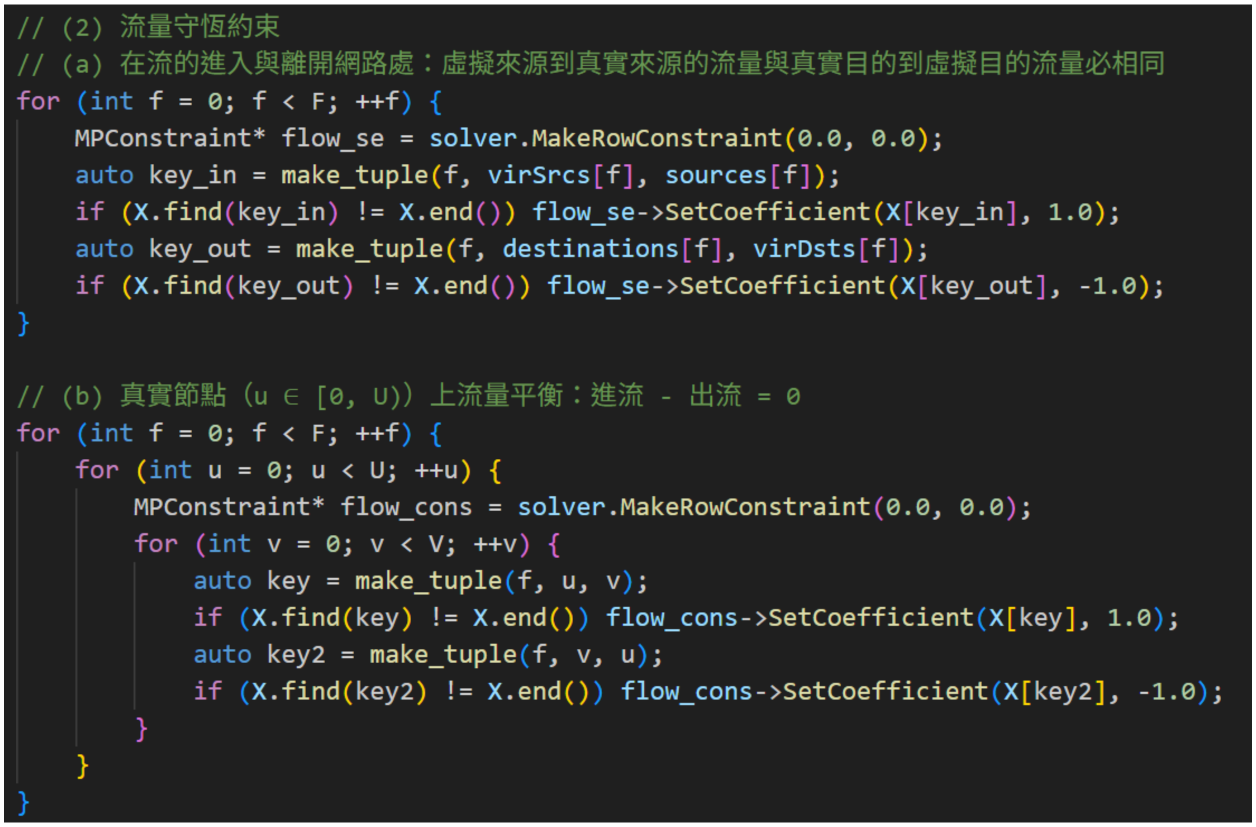
```cpp
// 8. 目標函數：最大化各流從虛擬來源至真實來源的流量總和
MPObjective* const objective = solver.MutableObjective();
for (int f = 0; f < F; ++f) {
    auto it = X.find({f, virSrcs[f], sources[f]});
    if (it != X.end())  objective->SetCoefficient(it->second, 1.0);
}
objective->SetMaximization();
```

This section defines the objective function, which seeks to maximize the total flow from virtual sources to real sources. It assigns a coefficient of 1.0 to corresponding $X[f, virSrcs[f], sources[f]]$ variables, ensuring their contribution to the optimization goal. Finally, the function is set to maximize this sum.



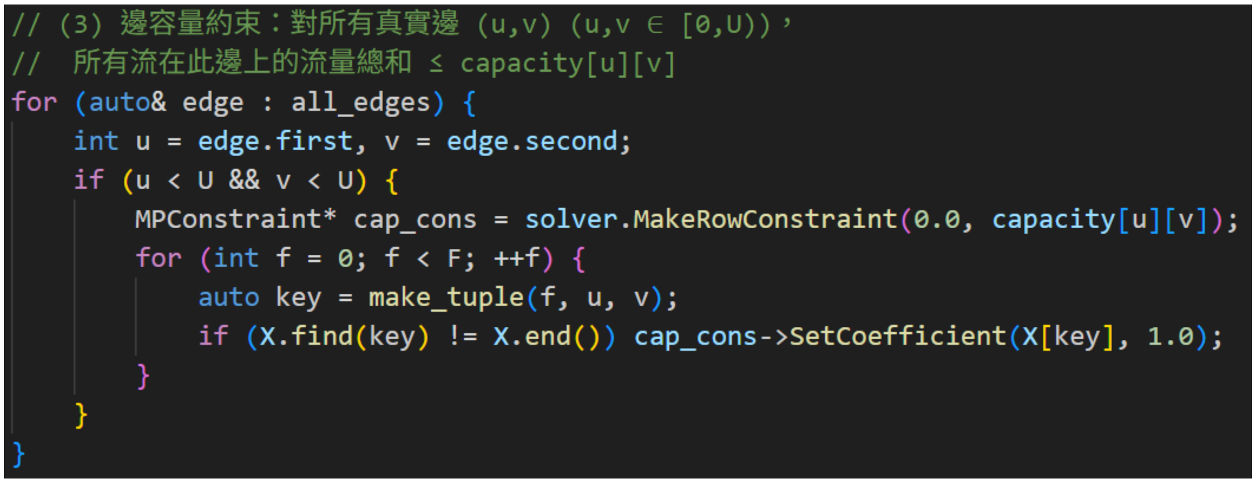
```
// (1) 單一路徑約束：對每個流 f 及每個節點 u (u ∈ [0,V))，
//   「從 u 發出」以及「進入 u」的 Y 變數總和均 ≤ 1
for (int f = 0; f < F; ++f) {
    for (int u = 0; u < V; ++u) {
        MPConstraint* out_cons = solver.MakeRowConstraint(0.0, 1.0);
        MPConstraint* in_cons  = solver.MakeRowConstraint(0.0, 1.0);
        for (int v = 0; v < V; ++v) {
            auto key = make_tuple(f, u, v);
            if (X.find(key) != X.end()) out_cons->SetCoefficient(Y[key], 1);

            auto key_in = make_tuple(f, v, u);
            if (X.find(key_in) != X.end()) in_cons->SetCoefficient(Y[key_in], 1);
        }
    }
}
```

This section enforces a single-path constraint for each flow f at every node u. It ensures that the total selection of edges entering or leaving u does not exceed 1, preventing multiple paths from forming. Constraints are applied by summing the corresponding $Y[f, u, v]$ variables and limiting them within 0 to 1.



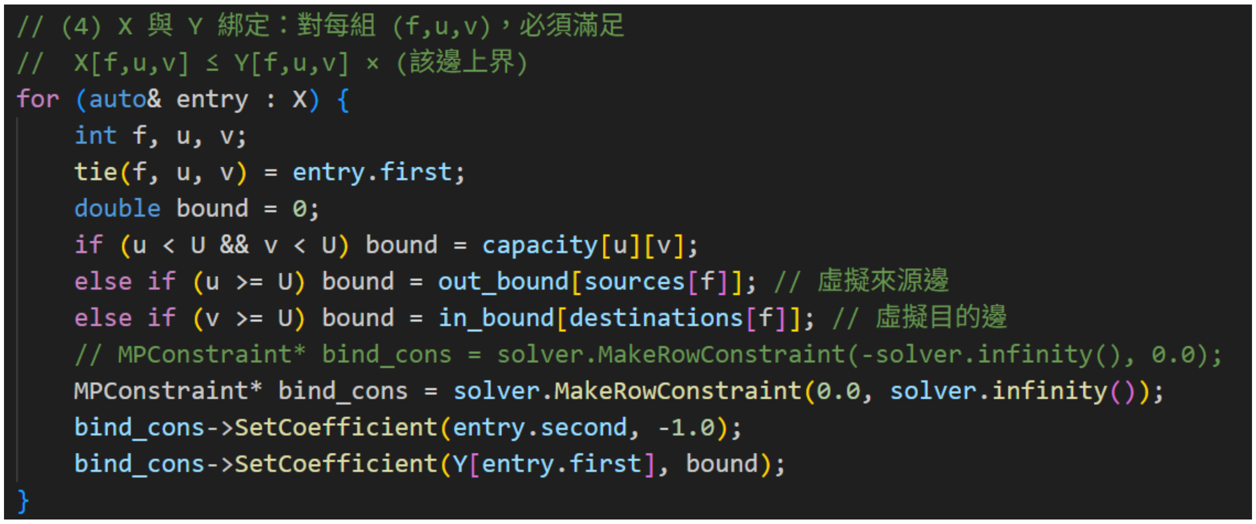
```
// (2) 流量守恆約束
// (a) 在流的進入與離開網路處：虛擬來源到真實來源的流量與真實目的到虛擬目的流量必相同
for (int f = 0; f < F; ++f) {
    MPConstraint* flow_se = solver.MakeRowConstraint(0.0, 0.0);
    auto key_in = make_tuple(f, virSrcs[f], sources[f]);
    if (X.find(key_in) != X.end()) flow_se->SetCoefficient(X[key_in], 1.0);
    auto key_out = make_tuple(f, destinations[f], virDsts[f]);
    if (X.find(key_out) != X.end()) flow_se->SetCoefficient(X[key_out], -1.0);
}

// (b) 真實節點 (u ∈ [0, U)) 上流量平衡：進流 - 出流 = 0
for (int f = 0; f < F; ++f) {
    for (int u = 0; u < U; ++u) {
        MPConstraint* flow_cons = solver.MakeRowConstraint(0.0, 0.0);
        for (int v = 0; v < V; ++v) {
            auto key = make_tuple(f, u, v);
            if (X.find(key) != X.end()) flow_cons->SetCoefficient(X[key], 1.0);
            auto key2 = make_tuple(f, v, u);
            if (X.find(key2) != X.end()) flow_cons->SetCoefficient(X[key2], -1.0);
        }
    }
}
```

This section enforces flow conservation constraints. First, it ensures that the total flow from virtual sources to real sources equals the flow from real destinations to virtual destinations. Second, for real nodes, it maintains balance by setting incoming flow minus outgoing flow to zero, preserving consistency in the network.



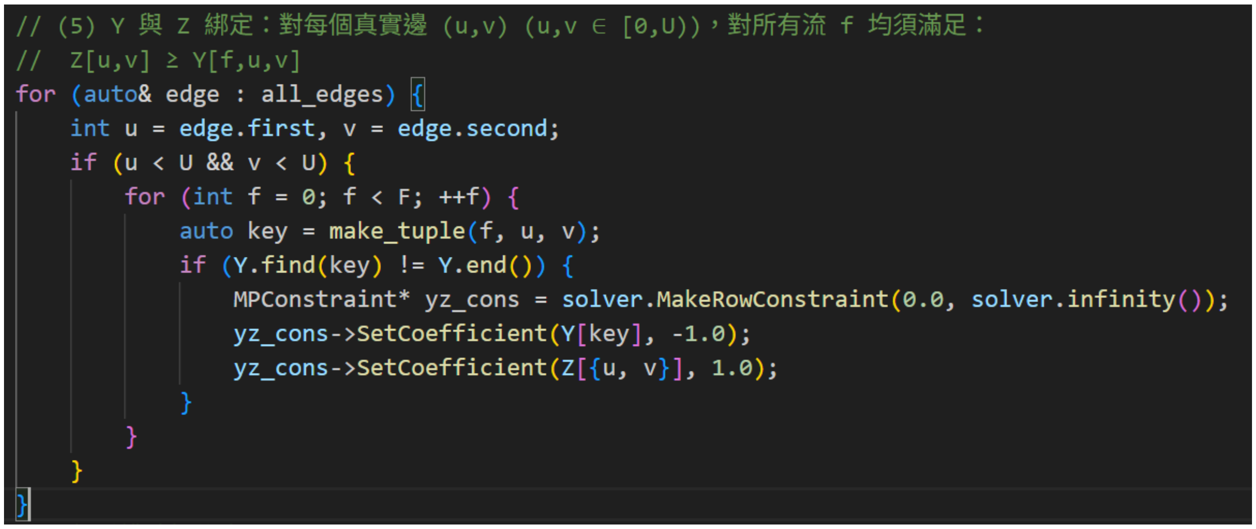
```
// (3) 邊容量約束：對所有真實邊 (u,v) (u,v ∈ [0,U))，
//  所有流在此邊上的流量總和 ≤ capacity[u][v]
for (auto& edge : all_edges) {
    int u = edge.first, v = edge.second;
    if (u < U && v < U) {
        MPConstraint* cap_cons = solver.MakeRowConstraint(0.0, capacity[u][v]);
        for (int f = 0; f < F; ++f) {
            auto key = make_tuple(f, u, v);
            if (X.find(key) != X.end()) cap_cons->SetCoefficient(X[key], 1.0);
        }
    }
}
```

This section enforces edge capacity constraints by ensuring the total flow across each real edge (u, v) does not exceed its predefined capacity. Constraints are established for all real edges, summing up the $X[f, u, v]$ variables across all flows and limiting them within $[0, capacity[u][v]]$. This guarantees that no edge is overloaded beyond its designed capacity.



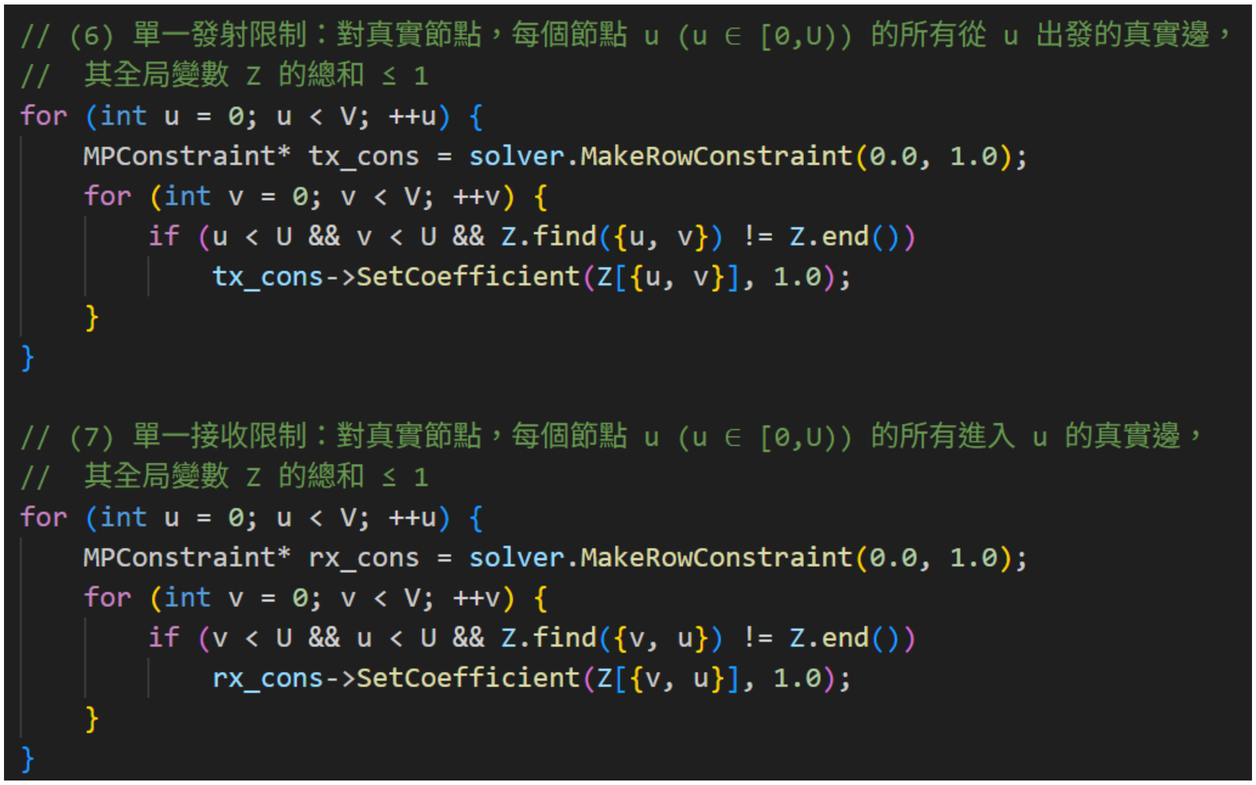
```
// (4) X 與 Y 綁定：對每組 (f,u,v)，必須滿足
//  X[f,u,v] ≤ Y[f,u,v] × (該邊上界)
for (auto& entry : X) {
    int f, u, v;
    tie(f, u, v) = entry.first;
    double bound = 0;
    if (u < U && v < U) bound = capacity[u][v];
    else if (u >= U) bound = out_bound[sources[f]]; // 虛擬來源邊
    else if (v >= U) bound = in_bound[destinations[f]]; // 虛擬目的邊
    // MPConstraint* bind_cons = solver.MakeRowConstraint(-solver.infinity(), 0.0);
    MPConstraint* bind_cons = solver.MakeRowConstraint(0.0, solver.infinity());
    bind_cons->SetCoefficient(entry.second, -1.0);
    bind_cons->SetCoefficient(Y[entry.first], bound);
}
```

This section ensures that the flow variable $X[f, u, v]$ is constrained by the selection variable $Y[f, u, v]$. It sets $X[f, u, v]$ to be at most $Y[f, u, v]$ multiplied by the upper bound of edge (u, v), ensuring consistency in flow assignments. Constraints are created dynamically based on edge type, enforcing logical flow limitations.



```cpp
// (5) Y 與 Z 綁定：對每個真實邊 (u,v) (u,v ∈ [0,U))，對所有流 f 均須滿足：
//  Z[u,v] ≥ Y[f,u,v]
for (auto& edge : all_edges) {
    int u = edge.first, v = edge.second;
    if (u < U && v < U) {
        for (int f = 0; f < F; ++f) {
            auto key = make_tuple(f, u, v);
            if (Y.find(key) != Y.end()) {
                MPConstraint* yz_cons = solver.MakeRowConstraint(0.0, solver.infinity());
                yz_cons->SetCoefficient(Y[key], -1.0);
                yz_cons->SetCoefficient(Z[{u, v}], 1.0);
            }
        }
    }
}
```

This section enforces the dependency between $Y[f, u, v]$ and $Z[u, v]$ for all real edges (u, v). It ensures that if any flow f selects edge (u, v), the global indicator $Z[u, v]$ must also be active. Constraints are set such that $Z[u, v]$ is always greater than or equal to any selected $Y[f, u, v]$, preserving logical consistency in edge utilization.



```cpp
// (6) 單一發射限制：對真實節點，每個節點 u (u ∈ [0,U)) 的所有從 u 出發的真實邊，
//  其全局變數 Z 的總和 ≤ 1
for (int u = 0; u < V; ++u) {
    MPConstraint* tx_cons = solver.MakeRowConstraint(0.0, 1.0);
    for (int v = 0; v < V; ++v) {
        if (u < U && v < U && Z.find({u, v}) != Z.end())
            tx_cons->SetCoefficient(Z[{u, v}], 1.0);
    }
}

// (7) 單一接收限制：對真實節點，每個節點 u (u ∈ [0,U)) 的所有進入 u 的真實邊，
//  其全局變數 Z 的總和 ≤ 1
for (int u = 0; u < V; ++u) {
    MPConstraint* rx_cons = solver.MakeRowConstraint(0.0, 1.0);
    for (int v = 0; v < V; ++v) {
        if (v < U && u < U && Z.find({v, u}) != Z.end())
            rx_cons->SetCoefficient(Z[{v, u}], 1.0);
    }
}
```

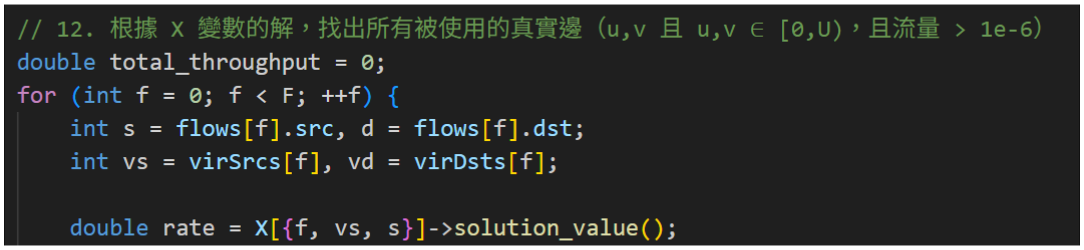
These constraints enforce single transmission and reception rules for real nodes. The total number of outgoing edges selected for transmission from any node u must not exceed 1, ensuring exclusive transmission. Similarly, the total number of incoming edges selected for reception at u is limited to 1, preventing multiple receptions at a node.

```
// 10. 求解模型
MPSolver::ResultStatus result_status = solver.Solve();
cerr << "[DEBUG] Solver status: " << result_status << "\n";
cerr << "[DEBUG] Objective value: " << solver.Objective().Value() << "\n";
if (result_status != MPSolver::OPTIMAL) {
    cerr << "[DEBUG] Solver did not find an optimal solution.\n";
    out << "0\n";
    return 0.0;
}
```

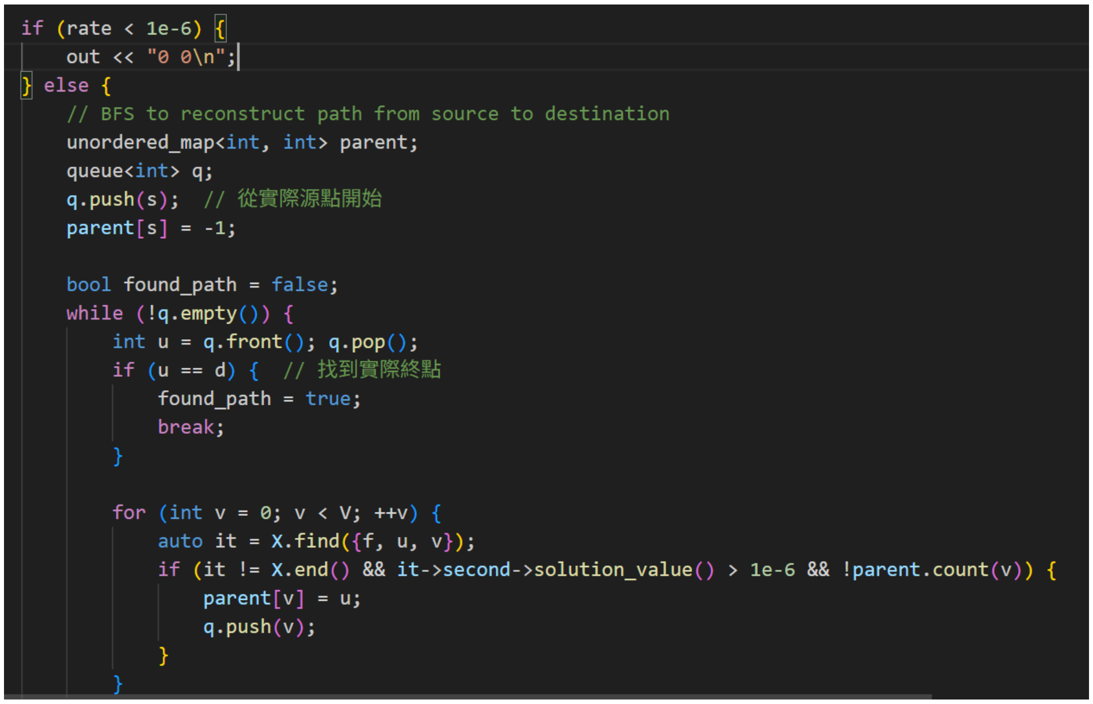
This section solves the optimization model and retrieves the solution status. It prints debug information about the solver's result and objective value. If an optimal solution is not found, it outputs 0 and terminates execution.

```
// 11. 記錄所有被使用的真實邊
map<pair<int, int>, bool> edge_used_map;
for (auto& [key, xvar] : X) {
    int f, u, v;
    tie(f, u, v) = key;
    if (u < U && v < U && xvar->solution_value() > 1e-6) {
        edge_used_map[{u, v}] = true;
    }
}
vector<pair<int, int>> used_edges;
for (auto& [uv, flag] : edge_used_map) {
    if (flag) {
        used_edges.push_back(uv);
    }
}
// out << "--------------------------------\n";
out << used_edges.size() << "\n";
for (const auto& [u, v] : used_edges) {
    out << u << " " << v << "\n";
}
```

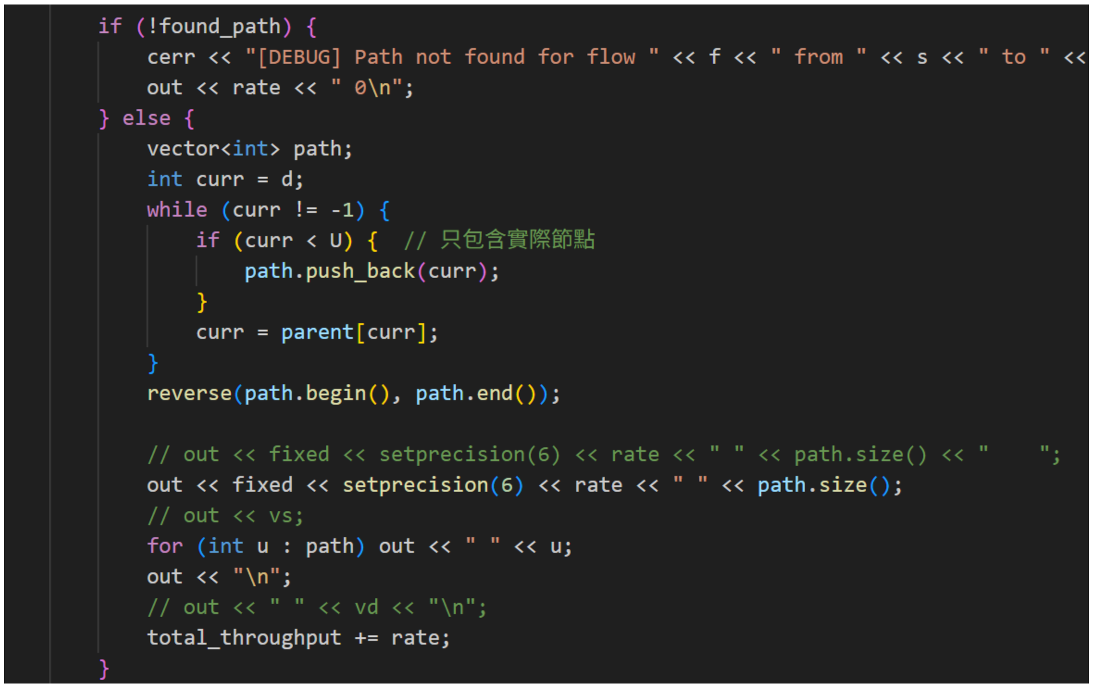
This section records all real edges that were utilized in the solution. It identifies edges (u, v) where the flow variable $X[f, u, v]$ has a positive solution value. The selected edges are stored in $used\_edges$ and their count is outputted. Finally, each used edge is printed for reference.



```cpp
// 12. 根據 X 變數的解，找出所有被使用的真實邊 (u,v 且 u,v ∈ [0,U)，且流量 > 1e-6)
double total_throughput = 0;
for (int f = 0; f < F; ++f) {
    int s = flows[f].src, d = flows[f].dst;
    int vs = virSrcs[f], vd = virDsts[f];

    double rate = X[{f, vs, s}]->solution_value();
```



```cpp
if (rate < 1e-6) {
    out << "0 0\n";
} else {
    // BFS to reconstruct path from source to destination
    unordered_map<int, int> parent;
    queue<int> q;
    q.push(s);  // 從實際源點開始
    parent[s] = -1;

    bool found_path = false;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        if (u == d) {  // 找到實際終點
            found_path = true;
            break;
        }

        for (int v = 0; v < V; ++v) {
            auto it = X.find({f, u, v});
            if (it != X.end() && it->second->solution_value() > 1e-6 && !parent.count(v)) {
                parent[v] = u;
                q.push(v);
            }
        }
```



```cpp
    if (!found_path) {
        cerr << "[DEBUG] Path not found for flow " << f << " from " << s << " to " <<
        out << rate << " 0\n";
    } else {
        vector<int> path;
        int curr = d;
        while (curr != -1) {
            if (curr < U) {  // 只包含實際節點
                path.push_back(curr);
            }
            curr = parent[curr];
        }
        reverse(path.begin(), path.end());

        // out << fixed << setprecision(6) << rate << " " << path.size() << "    ";
        out << fixed << setprecision(6) << rate << " " << path.size();
        // out << vs;
        for (int u : path) out << " " << u;
        out << "\n";
        // out << " " << vd << "\n";
        total_throughput += rate;
    }
```

This section computes the total throughput by processing each flow individually. For each flow, it retrieves the real source and destination along with corresponding virtual nodes, and then determines the flow rate from the virtual source to the real source using the solution value of the variable X. It prints debug information about the flow's rate and node details. If the flow rate is negligible (below $1e-6$), it outputs "0 0" to indicate no valid flow for that case. Otherwise, it uses a breadth-first search (BFS) starting from the real source to reconstruct the path to the destination by exploring subsequent nodes with significant flow values. The BFS continues until it finds the destination or exhausts all possible paths. If a valid path is not discovered, it logs an error and outputs the flow rate with a path size of zero. When a path is found, the code reconstructs the path by backtracking using parent pointers and then reverses it so that it correctly runs from the source to the destination. The complete valid path along with the flow rate and the number of nodes in the path is then output. Finally, the flow rate is accumulated into a total throughput, which is printed in a fixed precision format and logged for debugging purposes.

Now, let's move on to task3.

Due to the length of the report might be too long, I am only to describe lab5_myalgo.cpp in text without providing the screenshot of the code.

The code begins by defining a network flow problem where each instance comprises nodes, bidirectional edges (each represented twice with their capacities), and specified source-destination pairs for flows. The $Instance$ struct encapsulates the number of nodes (n), edges (m), flows (f), a vector of $Edge$ objects, an adjacency list mapping nodes to edge IDs, and a list of source-destination pairs. Within the $read\_input$ function, the program reads multiple instances, and for each edge, it creates two directed edges to represent bidirectional connectivity while updating the adjacency list accordingly. It also captures the source-destination pairs which are later used to determine the flow paths.

The $find\_path$ function applies a modified Dijkstra's algorithm to find a valid path from a given source to destination while respecting both capacity constraints and node usage restrictions. It initializes distance and predecessor arrays and uses a priority queue with a custom weight function that penalizes edges nearing full capacity, while skipping over edges that are fully used or connect to nodes already marked as transmitting or receiving. If a path is found, the function reconstructs it by backtracking and determines the bottleneck capacity along the route. In the $main$ function, each instance is processed by attempting to route each flow with $find\_path$; successful flows update edge usage, mark nodes to enforce single transmission and reception, and record both individual flow throughput and the sequence of nodes in the path. Finally, the code outputs the number of unique links used, detailed information for each flow, the total throughput per instance, and the average throughput across all instances.

3rd commit:

The third commit will be uploaded the report file!

Questions:

1. Write down the 3 constraints you add in task1 and briefly explain it.
   Y-Z Binding Constraint: For all flows $f \in F$ and every edge $(u, v) \in E(u, v)$, the constraint is:
   $$Y - Z\ binding: Z_{u,v} \geq Y_{f,u,v}, \ \ \forall f \in F, (u, v) \in E$$
   This ensures that if any flow ff uses edge (u, v) (i.e., $Y(f, u, v) = 1$), then the edge is marked as used overall by setting $Z(u, v) = 1.$
   Single Transmitter Constraint: For every node u in the set of real nodes U, we have
   $$single\ transmitter: \sum_{v \in U} Z_{u,v} \leq 1, \ \ \forall u \in U, \forall (u, v) \in E$$
   This guarantees that each node u can transmit via at most one outgoing edge.
   Single Receiver Constraint: Similarly, for every node $u \in U$, the constraint is
   $$single\ reciever: \sum_{v \in U} Z_{v,u} \leq 1, \ \ \forall u \in U, \forall (v, u) \in E$$
   This ensures that each node u can only be the receiver of one incoming edge, preventing multiple simultaneous receptions.

2. Calculate the average throughput ratio between network.myalgo.out and network.ortools.out
   $$ratio = \frac{77.600000}{111.200000} = 0.697841727$$
3. Briefly explain the main idea of myalgo
   The algorithm reads multiple test instances, each representing a network with nodes, bidirectional edges (each having a defined capacity), and a set of source-destination pairs for the flows. In each instance, the graph is built by duplicating each edge to enable bidirectional traversal, and an adjacency list is constructed to facilitate efficient access to incident edges. Additionally, each flow is processed individually using the provided source-destination pairs.
   For each flow, the algorithm applies a modified Dijkstra's algorithm (implemented in the $find\_path$ function) to determine a feasible path from the source to the

destination. This path search incorporates a cost function that penalizes edges increasingly used relative to their capacity, and it avoids edges if either the sending or receiving node has been marked as already active (to enforce the single transmitter/receiver constraint). Once a valid path is found, the algorithm updates the used capacity on each edge along the path, marks the corresponding nodes as transmitting or receiving, and records the flow's throughput. Finally, it outputs details about the used links, individual flow paths and rates, and computes both the total and average throughput across all instances.

4. Analyze the time complexity of myalgo

   <Read input> :

$$per\ instance: O(m+f)$$

$$\Rightarrow total\ over\ all\ instances: O\left(\sum_i m_i + f_i\right)$$

   <per-pair shortest-path search $find\_path$>:

$$Dijkstra\ style\ search\ with\ a\ binary\ heap\ (pq)$$

$$Each\ node\ may\ be\ extracted\ once\ (in\ the\ usual\ Dijkstra\ analysis) \rightarrow up\ to\ O(n)$$

$$Each\ arc\ is\ considered\ once\ in\ total \rightarrow up\ to\ O(m) \text{relaxation}\ steps$$

$$Each\ heap\ operation\ (extract\ or\ push)\ costs\ O(\log n)$$

$$final\ paht\ reconstruction\ is\ O(n)$$

$$\Rightarrow findpath = O\big((n+m)\log n\big)$$

   <main loop over f pairs>

   For each of the f sd-pairs:

   1. $call\ findpath \rightarrow O\big((n+m)\log n\big)$
   2. Walk the returned path ($length \le n$) updating used-capacity, flags and a set insert $O(n + \log m)$

   So per pair: $O\big((n+m)\log n + n + \log m\big) = O\big((n+m)\log n\big)$

   Over all f pairs in one instance: $O(f(n+m)\log n)$

   <output the result>

   - Printing up to $|used\ links| \le m$ edges $\rightarrow O(m)$
   - Printing f paths of length $\le n \rightarrow O(fn)$

$$\Rightarrow per\ instance\ time = O(f \cdot (n+m)\log n)$$

$$\Rightarrow total\ time = O\left(i = \sum_{i=1}^{T} f_i \cdot (n_i + m_i)\log n_i\right)$$