Commit 1 → Calculate Data Rate (bf.m)、greedy.cc/out

Commit 2 → OR-tools.cc/out

Commit 3 → 加上 report

Task 1：Calculate Data Rate

先讀取 graph 並設置參數

```
function generate_network_graph(filename)
    % 讀取 network.pos
    fileID = fopen(filename, 'r');

    % 讀取第一行 (tx power, noise, frequency, bandwidth, rx power threshold)
    header = fscanf(fileID, '%f %f %f %f %f', [5 1])';
    P_tx_dBm = header(1);
    N0_dBm = header(2);
    freq = header(3);
    bandwidth = header(4);
    P_rx_thr_dBm = header(5);

    % 讀取第二行 (#ground station, #satellite)
    numStations = fscanf(fileID, '%d %d', [2 1])';
    numGroundStations = numStations(1);
    numSatellites = numStations(2);
```

將 station、satellite 座標存起來

```

    % 讀取 ground station 座標
    gs_data = zeros(numGroundStations, 4);
    for i = 1:numGroundStations
        gs_data(i, :) = fscanf(fileID, '%d %f %f %f', [4 1])';
    end

    % 讀取 satellite 座標
    sat_data = zeros(numSatellites, 4);
    for i = 1:numSatellites
        sat_data(i, :) = fscanf(fileID, '%d %f %f %f', [4 1])';
    end

    fclose(fileID);
```

計算每個可能存在的 link 的 datarate：

```
% 計算每個 Ground Station 到 Satellite 的數據傳輸率
links = [];
for i = 1:numGroundStations
    for j = 1:numSatellites
```

計算仰角並找到 optimal beam (跟 lab2 的參數、算法一樣，用 sin 去算)

```
% 計算仰角 θ
dis_xy = sqrt((gs_data(i, 2) - sat_data(j, 2))^2 + (gs_data(i, 3) - sat_data(j, 3))^2);
theta_degree = abs(atan2(dis_xy, sat_data(j, 4) - gs_data(i, 4)) * (180 / pi));
if theta_degree > 90
    theta_degree = 180 - theta_degree;
end

% 波束方向對準
tx_beam_direction = 0:5:90;
[~, index1] = min(abs(tx_beam_direction - theta_degree));
op_beam = tx_beam_direction(index1);
```

計算 Tx gain、pathloss、Rx power，若 Rx power > threshold，則 link 存在並且計算 data rate

```
% 計算 beamforming gain
d = 0.5;
tx_antenna_number = 16;
phi_degree = 0.5 : 0.5 : 180;
phi_rad = phi_degree * pi / 180;
psi = 2 * pi * d * sin(phi_rad);
a1 = uniform(d, op_beam, tx_antenna_number);
A1 = dtft(a1, -psi);
gain_table = abs(A1).^2;
rx_sector_index = round(theta_degree / 0.5);
Tx_gain = gain_table(rx_sector_index);
```

```
% 計算 Path Loss (Friis 公式)
path_loss_dB = friis_equation(freq, Tx_gain, 1, distance);

% 計算接收功率
Rx_power_dBm = P_tx_dBm + path_loss_dB;

% 若 Rx Power >= 門檻，則計算 Data Rate
if Rx_power_dBm >= P_rx_thr_dBm
    Rx_power_W = 10^(Rx_power_dBm / 10) / 1000;
    N0_W = 10^(N0_dBm / 10) / 1000;
    SNR = Rx_power_W / N0_W;
    data_rate_kbps = bandwidth * log2(1 + SNR)/1000   ;
    links = [links; gs_data(i,1), sat_data(j,1), data_rate_kbps];
end
```

輸出 network.graph！

```
% 輸出 network.graph
fileID = fopen('network.graph', 'w');
fprintf(fileID, '%d %d %d\n\n', numGroundStations, numSatellites, size(links, 1));
for k = 1:size(links, 1)
    fprintf(fileID, '%d %d %f\n', links(k, 1), links(k, 2), links(k, 3));
end
fclose(fileID);
disp('network.graph 已成功生成！');
```

Task 2：OR-Tools Program

開啟輸入檔與輸出檔並讀取資料 (地面站數量、衛星數量、link 數量)

逐筆讀入 links 資料，並存入 links。

time_map 記錄地面站與衛星之間的資料傳輸時間（傳輸 1000 單位資料所需時間 = 1000 / rate）。

```
struct Link {
  int ground_station;
  int satellite;
  double data_rate;
};
```

```
std::ifstream fin("BasicExample/src/network.graph");
std::ofstream fout("BasicExample/src/network.ortools.out");

int num_gs, num_sat, num_links;
fin >> num_gs >> num_sat >> num_links;

std::vector<Link> links(num_links);
std::unordered_map<std::string, double> time_map;

for (int i = 0; i < num_links; ++i) {
  int gs, sat;
  double rate;
  fin >> gs >> sat >> rate;
  links[i] = {gs, sat, rate};
  time_map[std::to_string(gs) + "_" + std::to_string(sat)] = 1000.0 / rate;
}
```

建立一個求解器 solver。

宣告變數 x[g_s]，代表地面站 g 是否選擇連到衛星 s，是 0 / 1 的整數變數。

```
MPSolver solver("BipartiteAssignment", MPSolver::CBC_MIXED_INTEGER_PROGRAMMING);
std::map<std::string, MPVariable*> x;

for (const auto& link : links) {
  std::string key = std::to_string(link.ground_station) + "_" + std::to_string(link.satellite);
  x[key] = solver.MakeIntVar(0, 1, "x_" + key);
}
```

Constraint 1：對於每個地面站，計算所有可能連線的變數和，強制設為 1

→ 表示只能選一條

```
// 每個 ground station 只能選一個 satellite
for (int g = 0; g < num_gs; ++g) {
  LinearExpr sum;
  for (const auto& link : links) {
    if (link.ground_station == g) {
      sum += x[std::to_string(g) + "_" + std::to_string(link.satellite)];
    }
  }
  solver.MakeRowConstraint(sum == 1);
}
```

Constraint 2：計算每一顆衛星的總資料接收時間 total_time，限制這些 total_time 都必須小於等於 max_time

```
// 每個 satellite 的 data collection time (以一個變數代表最大值)
MPVariable* max_time = solver.MakeNumVar(0.0, MPSolver::infinity(), "max_time");

for (int s = 0; s < num_sat; ++s) {
  LinearExpr total_time;
  for (const auto& link : links) {
    if (link.satellite == s) {
      std::string key = std::to_string(link.ground_station) + "_" + std::to_string(s);
      total_time += time_map[key] * LinearExpr(x[key]);
    }
  }
  solver.MakeRowConstraint(total_time <= max_time);
}
```

設定目標為 最小化 max_time (最忙的衛星需要的時間)。

```
// 目標：minimize max_time
MPObjective* const objective = solver.MutableObjective();
objective->SetMinimization();
objective->SetCoefficient(max_time, 1);
```

呼叫 Solve() 開始求解問題。

```
// Solve!
const MPSolver::ResultStatus result_status = solver.Solve();

if (result_status != MPSolver::OPTIMAL) {
  std::cerr << "No optimal solution found!" << std::endl;
  return 1;
}
```

輸出最小的最大時間。

根據變數的解（solution_value > 0.5）來確定哪些地面站選擇哪些衛星。

計算每顆衛星實際的總負載時間。

```
// Output
fout << max_time->solution_value() << std::endl;

std::vector<int> gs_to_sat(num_gs, -1);
std::vector<double> sat_time(num_sat, 0.0);

for (const auto& link : links) {
  std::string key = std::to_string(link.ground_station) + "_" + std::to_string(link.satellite);
  if (x[key]->solution_value() > 0.5) {
    gs_to_sat[link.ground_station] = link.satellite;
    sat_time[link.satellite] += time_map[key];
  }
}

for (int g = 0; g < num_gs; ++g)
  fout << g << " " << gs_to_sat[g] << std::endl;

for (int s = 0; s < num_sat; ++s)
  fout << s << " " << sat_time[s] << std::endl;
```

Task 3：Greedy Program

跟 task 2 一樣讀取資料 (地面站數量、衛星數量、link 數量)

逐筆讀入 links 資料，並存入 links。

。。。

為每一個地面站，選擇「資料率最高」的衛星作為配對對象。

(遍歷所有連線、為每個地面站記錄目前「最佳資料率」的衛星、最終建立 assignment[地面站] = 衛星 的映射表。)

```
// 執行 Greedy 分配演算法
map<int, int> greedy_assignment(const vector<Link>& links) {
    map<int, int> assignment;
    map<int, double> max_data_rate;

    // 每個 Ground Station 選擇數據率最高的 Satellite
    for (const auto& link : links) {
        if (max_data_rate.find(link.ground_station) == max_data_rate.end() ||
            link.data_rate > max_data_rate[link.ground_station]) {
            max_data_rate[link.ground_station] = link.data_rate;
            assignment[link.ground_station] = link.satellite;
        }
    }

    return assignment;
}
```

計算每顆衛星實際需花多少時間來接收所有地面站傳來的資料。

```cpp
// 計算 Satellite 的數據收集時間
map<int, double> calculate_collection_time(const vector<Link>& links, const map<int, int>& assignment) {
    map<int, double> collection_time;

    for (const auto& link : links) {
        if (assignment.at(link.ground_station) == link.satellite) {
            collection_time[link.satellite] += 1000.0 / link.data_rate; // 假設 1000kb 為 1 個數據單元
        }
    }

    return collection_time;
}
```

跟 task 2 一樣輸出 output (最大傳輸時間、分配結果、每個 Satellite 的數據收集時間)

Comparison

Greedy **ortools**

network.greedy.out

```
0.155097
0 1
1 2
2 2
3 3
4 3
5 3
6 3
7 4
8 5
9 6
10 4
11 5
12 6
13 8
14 8
15 8
16 8
17 8
18 8
19 5
1 0.0259293
2 0.0518876
3 0.103519
4 0.0520076
5 0.0780136
6 0.0519061
8 0.155097
```

network.ortools.out

```
0.0537272
0 0
1 0
2 1
3 1
4 2
5 2
6 3
7 6
8 5
9 5
10 3
11 4
12 6
13 7
14 8
15 8
16 7
17 9
18 9
19 4
0 0.0535995
1 0.0537272
2 0.0533184
3 0.0530636
4 0.0528552
5 0.0528196
6 0.0535035
7 0.0526304
8 0.0516969
9 0.0524409
```

Greedy：地面站分配較不平均，例如有 6 個地面站（13~18）都分配到衛星 8，導致過載、某些衛星（如 0、7、9）完全沒有被使用。

Or-tools：分配更平均，幾乎每顆衛星都分配到 2～3 個地面站。每顆衛星的收集時間都接近（約 0.052～0.053），顯示負載平衡做得很好。

Maximum_transmission_time：

Or-tools < Greedy

execution time：

Or-tools > Greedy

Greedy 只是單純選擇最大資料率，不需要解決最佳化問題。

Greedy

優點：

實作簡單，邏輯直觀

計算速度非常快，適合即時處理或大規模資料初步分配

缺點：

分配容易不平均，某些衛星可能過載

無法保證最佳解，僅為局部最佳

OR-Tools

優點：

分配結果平均，有效減少單一衛星的最大負載

適合需要效能與公平性的應用場景

缺點：

計算複雜，執行時間較長

需要額外學習與設定 OR-Tools 工具與最佳化模型