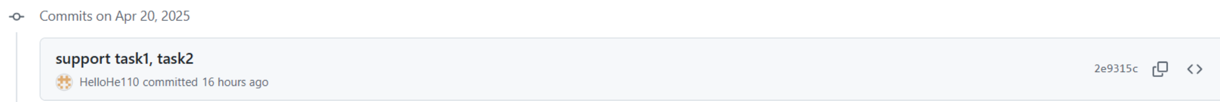
Report 111550022 施采緹

<commit history>



Here is the first commit – for supporting task1 and task2. Task1 is about to generate network.graph. And here's the thing I do:

a. Read network.pos (provided by TA)
b. Precompute angular grid for array pattern where the beam resolution is set to [0.5 : 0.5 : 90]
c. For every (GS, SAT) pair, I computed the (horizontal/vertical/total) distance of the pair; after at, I got the elevation angle by atan2d(horizontal, vertical), and since the elevation angle could only be in range of 0 to 90, I convert the $180 > \theta_1 > 90$ into θ_2 by $\theta_2 = 180 - \theta_1$ just like what I did in lab2! Furthermore, I select the closest tx beam direction and the AoD resolution index for the following computation about the gain table and the final (strongest) gain I picked from gain table.
d. (Still applied for every (GS, SAT) pair) I computed the Friss path loss and find out the Power of rx. Here is a thing that needs to be considered: if the link (Prx_dBm) is too weak $P_{rx(dBm)} < rx_{thresh(dBm)}$, we could say that this link (path) doesn't exist in the network graph. On the other hand, if the link is ok, I continue to compute date rate and store the Shannon_capacity's value in the links (array).
e. Finally, after traversing through all the (GS, SAT) pair, I wrote the {GS, SAT, DataRate} into the file $network.graph$ for the further usage in task2.

The previous description is about how task1 is implemented (the compute_datarate.m a.k.a bf.m) will be renamed in the following commit. And there are also some correlated files to support this MATLAB file, such as, lab2_bf_referece.m and ewa_functions.

Now, let's move on to the task2. The following are the process I do:

a. I read the $network.graph$ file which is located at $BasicExample/src/network.graph$ due to the program will be run under the cmake_or-tools.
b. After that, I construct the adjacency list by data-structure $std::vector < std::vector < std::pair < int, double >>>$ of size |GS|. Therefore, for each station v, we can get a list of (satellite_id, time) connected to this station. You can note that the time is simply compute by $1000.0/rate$ (sec) which converts the data rate into a transmission time of seconds for sending a 1000kb unit.
c. Before modeling, the code loops over all v form 0 to v-1 and ensures adj[v] is nonempty; if any station lacks valid links, it reports an infeasibility error and exits. This prevents creating an impossible constraint that would force a station with no edges to choose a satellite.

d. Now, it's time to initialize and setup the solver. A CBC-based MIP solver is instantiated via

```
auto solver = MPSolver::CreateSolver("CBC_MIXED_INTEGER_PROGRAMMING");
```

where MPSolver is OR-Tools' primary interface for LP/MIP problems. After that, I set up variables. For each ground station v, and for each outgoing link index k, the code creates a Boolean decision variable

```
x[v].push_back(solver->MakeBoolVar("x_v_s"));
```

indicating whether station v connects to satellite s. These variables are stored in a 2D vector x, and the corresponding satellite IDs in sat_of for later lookup.

e. Also, I set up the max-time variable, a continuous variable T is declared with bounds [0, ∞] to represent the maximum collection time across all satellites.

```
const MPVariable* T = solver->MakeNumVar(0.0, MPSolver::infinity(), "T");
```

f. Moreover, I setup constraints (1) the assignment constraint: for each station v, a row constraint enforces $\sum_k x_{v,k} = 1$ ensuring each ground station picks exactly 1 sat. (2) load-balancing constraints: To tie T to each satellite's load, the code builds, for each satellite s, a constraint

$$T - \sum_{(v,k):\ sat\_of[v][k]=s} t_{v,k} \cdot x_{v,k} \geq 0$$

by using an map to accumulate coefficients. This set of constraints guarantees that T is at least as large as every satellite's total assigned transmission time.

g. For the objective and solve, the objective function is set to minimize T via

```
solver->MutableObjective()->SetCoefficient(T, 1);
solver->MutableObjective()->SetMinimization();
```

and calling solver->Solve() runs CBC's branch-and-cut algorithm; the code checks for MPSOLVER::OPTIMAL to confirm a valid solution was found.

h. Finally, I wrote the result into the network.ortools.out!

Commits on Apr 22, 2025

support task3 and update task2, file_name

HelloHe110 committed 2 days ago

017d9dc

Ok, and here's my second commit for supporting task3 – the greedy approach and modify the output order in task2, and also the file name. And the following are what I've done in task3:

a. Same as task2, I read the $network.graph$ file which is located at $BasicExample/src/network.graph$ due to the program will be run under the cmake_or-tools.

b. After getting the network graph, it's time to select the "best" link per station. best rate will hold the highest data rate seen so far for station v. And best sat will stores which statellite gave that highest rate. I looped over all the link to update the value of those 2 vectors. Finally, each station v has chosen its single statellite best_sat[v] with maximum link rate.

```
+   // For each ground station: track best satellite (max rate)
+   std::vector<double> best_rate(V, 0.0);
+   std::vector<int> best_sat(V, -1);
+
+   for (int i = 0; i < L; ++i) {
+     int v, s;
+     double rate;
+     infile >> v >> s >> rate;
+     if (rate > best_rate[v]) {
+       best_rate[v] = rate;
+       best_sat[v]  = s;
+     }
+   }
```

c. Now, I can compute satellite loads and maximum time. The sat_time (satellite-load counter) is initialized to 0. And in per-station, I check that station v did find any valid satellite (best_sta[v] >= 0). After that, I compute its transmission time t = 1000/rate and accumulate t into sat_time[best_sat[v]]. We can finally get the global maximum by iterating through sat_time[s] and track the largest into T, which is the worst-case (bottleneck) satellite collection time under the greedy assignment.

```
+   // Compute per-satellite collection times and global max
+   std::vector<double> sat_time(S, 0.0);
+   double T = 0.0;
+   for (int v = 0; v < V; ++v) {
+     if (best_sat[v] < 0) {
+       std::cerr << "No valid link for station " << v << std::endl;
+       return 1;
+     }
+     double t = 1000.0 / best_rate[v]; // time for one data unit
+     sat_time[best_sat[v]] += t;
+   }
+   for (int s = 0; s < S; ++s) {
+     if (sat_time[s] > T) T = sat_time[s];
+   }
```

d. Finally, we can write the result into network.greedy.out.

Furthermore, I rename compute_datarate.m to bf.m and add a sorting to make the output of task2 be in order! Moreover, I add the studentID.txt as required.

After all, here is the third commit -> the report is added!

<Comparison>

Station	Greedy → Satellite	OR-Tools → Satellite
0	1	0
1	2	0
2	2	1
3	3	1
4	3	2
5	3	2
6	3	3
7	4	4
8	5	5
9	6	6
10	4	5
11	5	3
12	6	6
13	8	7
14	8	8
15	8	8
16	8	7
17	8	9
18	8	9
19	5	4

- **Maximum collection time T**
  - **Greedy**: 0.155097 second
  - **OR-Tools**: 0.0537272 second
- **Per-Satellite Loads:**
  - Greedy creates idle satellites (satellite with index = 0, 7, 9 spends 0s) while others pile up, e.g. satellites with index 8 has 0.155097 seconds.
  - OR-Tools spreads the work by letting all active satellites finish within $\sim 0.0517\ to\ 0.0537$ seconds.
- **Execution Time:**
  - **Greedy:** the asymptotic cost is about $O(L)$ over links, and the measured runtime is about $< 1ms$. (Greedy reads each of the L links once and picks the beat rate per station, so it runs in microseconds on this problem size)
  - **OR-Tools:** the asymptotic cost is about ILP via CBC, and the measured runtime is approximately $0.5\ to\ 3\ sec$ (OR-Tools builds a mixed-integer program and solves it with CBC; we might typically see runtimes on the order of seconds for a 20×10 instance.

- **Advantages & Disadvantages:**

Criterion	**Greedy**	**OR-Tools ILP**
Solution Quality	Suboptimal worst-case time	Optimal min-max load
Speed	Near instantaneous	Seconds (can be tuned)
Implementation	Simple loops, no external libs	Requires OR-Tools setup & modeling
Scalability	Scales to very large V, S	Limited by ILP solver on large data
Determinism	Fully deterministic	Deterministic but solver-tunable

Therefore, we can implement Greedy when we need a rapid, “good-enough” assignment in real-time or on very large networks; on the other hand, we can apply OR-Tools when worst-case performance matters and can afford an offline solve!

The overall comparison toward greedy and or-tools method is that the or-tools solver achieves a much lower maximum collection time (≈0.0537 s) than the Greedy heuristic (≈0.1551 s), at the cost of a heavier solve routine. Greedy is trivial to implement and near-instantaneous but can leave some satellites idle while others become bottlenecks. OR-Tools balances the load perfectly, minimizing the worst-case satellite time, but requires a full ILP solve which—even on modest hardware—can take seconds rather than microseconds.