Competitive Programming Honours: Feb 2023 Batch

Tim van Dam, Koos van der Linden, Emir Demirović

February 22, 2023

1 Optimal Card Trading

Collectable card games such as *Magic: The Gathering*, *Pokémon* and *Yu-Gi-Oh!* are games where the aim is to collect all n unique trading cards. On their own, each card is not worth much. A *complete collection* of n unique cards, however, can be worth a lot of money. It is therefore preferable to trade cards in such a way that maximizes the number of complete collections you own. A single card can be traded for one other card, assuming you find someone willing to trade.

A group of m card collectors have banded together to increase the worth of their collection by trading with each other. The card collectors share which cards they are willing to trade, and then trade with each other aiming to maximize the total amount of complete collections among them all. Note that the traders are working together to optimise the total number of sets shared amongst one another and are (surprisingly!) not selfish.

Before trading, the card collectors do not own any complete collections yet.

Determine the maximum amount of complete collections that can be made given a list of cards each collector is willing to trade. Make sure that each card is traded for exactly one other card.

Example

This example illustrates a scenario where three traders trade five unique cards.

Trader	Cards Owned					Complete Collections
Trader 1	0	0	6	7	8	0
Trader 2	2	2	0	2	2	0
Trader 3	10	3	0	0	0	0

Table 1: Optimal Card Trading Example – Before Trading

Trader	**Cards Owned**					**Complete Collections**	**Gained**
Trader 1	5	4	4	4	4	4	**+4**
Trader 2	4	1	1	1	1	1	**+1**
Trader 3	3	0	1	4	5	0	

Table 2: Optimal Card Trading Example – After Trading

In the example, a total of **five** complete collections were created by trading. Note that since one card trades for exactly one other card, every trader still has the exact same total amount of cards. Note that there can be multiple ways of achieving an optimal solution.

File Format

The input (available as `[name].in` files in the `./normal` folder) uses the following format:

```
[m = number of traders] [n = number of unique cards]
1 2 3 ... (how many of each card is owned by trader i, m rows total)
```

The previous example has the following input:

```
3 5
0 0 6 7 8
2 2 0 2 2
10 3 0 0 0
```

The expected output for each input can be found in `[name].ans` files. A solution for each input can be found in `[name].sol` files (note that there may be multiple optimal solutions).

2 Fair Optimal Card Trading

Some card collectors have complained about not gaining any complete collections, despite helping other traders to create complete collections by trading with them. After careful consideration, the card collectors decided to change the objective of their trades to increase fairness. The new objective aims to *maximize* the *minimum* number of complete collections gained among all collectors.

Determine the maximum amount of complete collections that can be made with this new objective. Output both the total number of complete collections, and the minimum number of complete collections among all traders.

Example

This example has an identical input as previous example, but the trades are now much fairer.

Trader	Cards Owned					Complete Collections
Trader 1	0	0	6	7	8	0
Trader 2	2	2	0	2	2	0
Trader 3	10	3	0	0	0	0

Table 3: Fair Optimal Card Trading Example – Before Trading

Trader	Cards Owned					Complete Collections	Gained
Trader 1	9	2	3	5	2	2	**+2**
Trader 2	1	1	1	2	3	1	**+1**
Trader 3	2	2	2	2	5	2	**+2**

Table 4: Fair Optimal Card Trading Example – After Trading

A total of **five** complete collections were created by trading, just like last time. However, it is now much fairer!

File Format

The input format is the same as the input format used for the previous problem. All files for this problem are contained in the `./fair` folder. Input files are named `[name]-fair.in`, the expected output for each input can be found in `[name]-fair.ans`, and a solution for each input can be found in `[name]-fair.sol` (note that there may be multiple optimal solutions).

3 Cooperative Multi-Agent Chasing

In multi-agent chasing, the goal is to direct a set of agents towards a set of moving targets. The traditional setting has focussed on the adversarial case, where targets actively avoid chasers, but here we will focus on a *cooperative* version, where targets reveal their path to the agents to make the chase more efficient.

Cooperative multi-agent chasing makes sense, since there are many settings where the two sides would like to cooperate to achieve their goals. For example, consider a battery-exchange service for electric cars. A client with an electric car plans their journey, but would like to exchange their low-battery for a fully charged battery on the way without deviating from their planned path. In this case, the client may reveal their journey plans to the battery-exchange service, which could then agree to meet the client somewhere on their path. In this scenario, the service (agent) and client (target) are cooperative. While algorithms for the adversarial setting may be used in this setting, it is conceivable that sharing the trajectory of the client upfront may lead to a better solution for both parties.

The *cooperative multi-agent chasing problem* can be defined as a tuple $(G, A, T, S_A, S_T, W, P)$, where $G = (V, E)$ is a graph, V is the set of vertices, E is the set of directed edges, A and T are the sets of agents and targets with $|A| = |T|$, $S_A : A \to V$ and $S_T : T \to V$ are mappings of agents and targets to their starting vertices, $W : E \to \mathbf{R}$ is a mapping from each edge to a weight, and $P(t) = (v_{(i,0)}, v_{(i,1)}, ..., v_{(i,k)})$ is a sequence of vertices that represent the path of target t. The targets move according to their predefined path ($P(t)$) which is known to the agents. The edge weight represents the time required to travel along the edge, which captures the temporal aspect of the problem. An agent intercepts a target if both entities are at the same vertex at the same time (note that while traversing an edge, the agent/target are not considered to be placed on any node). A target may be intercepted at any point on their path.

A *solution* is a set of paths, one for each agent, such that each agent intercepts exactly one target. Targets do not wait for agents, but agents may choose to wait at a vertex, e.g., an agent may decide to wait on a vertex along the path of a target. Note that according to our definition, an agent cannot intercept two or more targets - this makes the problem much more difficult!

Collisions between agents are *not* considered, e.g., it is possible to have two agents at one node. This makes the problem simpler, and is motivated by the fact that since any agent may be assigned to any target, it is ”easy” to resolve conflicts should this be a problem in the real-world.

There are two metrics that we will use to evaluate solutions. The *makespan* is the time when the last target is intercepted. The *summation* objective is the sum of edge weights traversed by agents. Optimising the makespan leads to completing all chases as quickly as possible. Optimising summation optimises the total travelled distance (useful, for instance, to minimise fuel consumption). Waiting at a node may impact the makespan, but not the summation objective.

Given a cooperative multi-agent chasing problem, your task is to compute two solutions: one that minimises the makespan, and another that minimises the summation objective.

Hint: You could start with solving the summation objective, and then consider how your approach can be modified to optimise makespan. The core part of the algorithm is the same, but makespan optimisation may require some additional changes.

File Format

The .in files have the following structure:

[n = vertex count] [m = edge count] [k = agent/target count]

[u = edge start] [v = edge destination] [w = edge weight/traversal time] (* m, one line per edge)

[s_i = starting vertex of agent i] (* k, one line per agent)

[p_j_len = number of vertices on target j's path] | (* k, one path per target)

[p_j_l = l-th vertex on target j's path] [t_j_l = time at which target arrives here] (* p_j_len, one for each vertex on the target's path) |

All vertices are 0-indexed.

The .ans files contain one number, which is simply the objective value (optimal makespan/optimal sum of costs) The .sol files contain agent-target pairs, including where the agent meets the target (which vertex and which time step):

[agent] [target] [meet vertex] [meet time] (* k, one for each agent/target)