Combinatorial Optimization
Lab No. 10
HW4: Minimum Cost Flow

Industrial Informatics Department
Czech Technical University in Prague
https://industrialinformatics.fel.cvut.cz/

April 20, 2026

Abstract

This lab is devoted to the minimum-cost flow problem. We will study the cycle-canceling algorithm that solves the minimum-cost flow problem. Next, we use this algorithm in a homework that is concerned with tracking objects in videos.

1 Cycle Canceling: Algorithm for Minimum Cost Flow Problem

The cycle canceling algorithm for solving Minimum Cost Flow Problem is conceptually very similar to Ford-Fulkerson's algorithm for Maximum Flow problem. Ford-Fulkerson starts from a feasible flow, that is further improved via consequent iterations. Similarly, the cycle canceling algorithm gradually reduces the cost of the flow while keeping balances satisfied in each vertex. Such an improvement in each iteration is characterized by a cost-negative cycle in a suitably defined graph – so-called *residual graph*. After a cost-negative cycle is found, the flow is augmented along the cycle which results into a new flow that still satisfies balances but is cheaper. The algorithm terminates when no additional cost-negative cycle can be found.

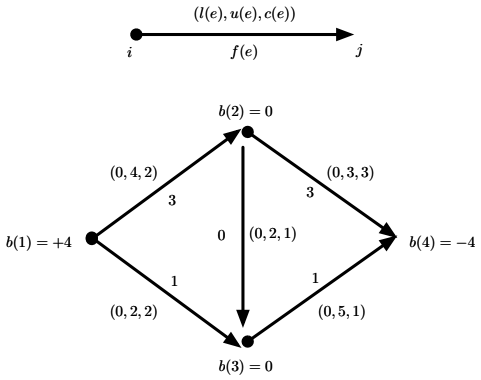
There are four key ingredients when implementing the cycle canceling algorithm:

1. Residual graph
2. Main loop
3. Detecting a negative cycle
4. Finding the initial balanced flow

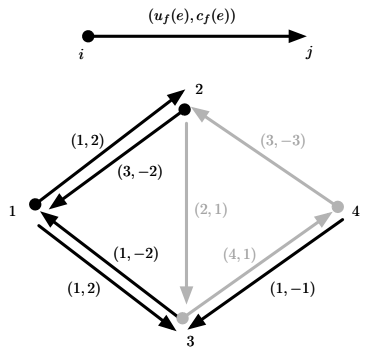
In the following sections, we will describe all of them.

1.1 Residual graph

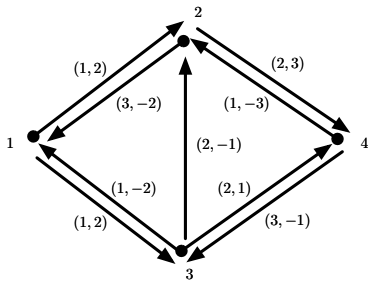
The whole algorithm operates on the structure called *residual graph*. Essentially, it is the same graph as we are solving minimum-cost flow for, but with minor differences that help us to find improving cycles faster.



(a) Original graph G with a feasible balanced flow f.



(b) Residual graph G_f with highlighted cost-negative cycle $C = (2, 3, 4)$ with $\delta = 2$.



(c) New residual graph $G_{f'}$ after augmenting flow f by δ units along the cycle C.

Figure 1: Example of the original graph with a feasible balanced flow and one iteration of cycle canceling algorithm on the residual graph.

There is no single residual graph for the given instance of the problem; actually, the residual graph is always associated with the specific feasible flow f. Hence, at each iteration, you have a new residual graph (i.e., we say *residual graph with respect to flow* f). It is a valued directed graph depending on the current flow f defined as follows:

$$
\begin{aligned}
G_f &= (V, E_f, u_f, c_f) \\
E_f &= E \cup \{(j,i) \,|\, \forall\, (i,j) \in E\} \\
u_f(i,j) &= u(i,j) - f(i,j) \quad \forall\, (i,j) \in E \\
u_f(j,i) &= f(i,j) - l(i,j) \quad \forall\, (i,j) \in E \\
c_f(i,j) &= c(i,j) \quad \forall\, (i,j) \in E \\
c_f(j,i) &= -c(i,j) \quad \forall\, (i,j) \in E
\end{aligned}
$$

See an example of the original graph and initial flow in Figure 1a and the corresponding residual graph in Figure 1b. The edges with zero residual capacity $u_f(e)$ are not displayed, since they cannot be on any cycle (and they can be even omitted from the residual graph). Therefore, for any feasible flow f in $G = (V, E)$, G_f has at most $2 \cdot |E|$ edges valued with numbers:

- $u_f(i,j)$ – improving capacity of the edge (i,j)
- $c_f(i,j)$ – cost of the edge (i,j)

Notice that edges $e \in E_f$ with the same orientation as in original graph have capacity $u(e) - f(e)$ and cost $c(e)$, wheres edges in the opposite direction have capacity $f(e) - l(e)$ and cost $-c(e)$.

These costs correspond to potential improvements (degradation) for decreasing (increasing) the flow along the edge e[1].

1.2 Main loop

The main loop of the cycle canceling algorithm is essentially the same as in Ford-Fulkerson algorithm. The main difference is, that instead of looking for an improving path (non-oriented path in the original graph) P, we are looking for a cost-negative cycle C in the residual graph G_f.

We denote the capacity of the cycle C by δ and it is given as $\delta = \min_{e \in C} u_f(e)$. If $\delta > 0$, we can augment the flow by δ units along the edges on the cycle C, and thus, the cost of the flow in the original graph decreases. See the resulting flow f' after the augmentation in Figure 1c.

Algorithm 1 Cycle canceling algorithm for Minimum-Cost Flow Problem

Require: initial feasible balanced flow f in (G, l, u, c, b)
Ensure: minimum-cost flow f in (G, l, u, c, b)

```
function CYCLE-CANCELING(G, l, u, c, b, f)
    G_f ← build residual graph w.r.t. f
    while True do
        δ, C ← BELLMAN-FORD(G_f)          ▷ remember not to use edges with u_f = 0
        if δ > 0 then
            for all e = (i, j) ∈ C do
                if e is a forward arc in G then          ▷ i.e., (i, j) ∈ E
                    f(i, j) ← f(i, j) + δ
                else
                    f(j, i) ← f(j, i) − δ
                end if
                u_f(i, j) ← u_f(i, j) − δ          ▷ update residual graph G_f
                u_f(j, i) ← u_f(j, i) + δ
            end for
        else
            break
        end if
    end while
    return f
end function
```

A cost-negative cycle can be found by e.g., Bellman-Ford algorithm for the shortest paths. When no cost-negative cycles can be found, the algorithm terminates.

1.3 Detecting a negative cycle

Finding a negative cycle in the residual graph G_f can be done easily by Bellman-Ford algorithm:

1. Take the current G_f, remove all edges with zero residual capacity (i.e., edges $e \in E_f$ with $u_f(e) = 0$).
2. Introduce a dummy vertex s and connect it with every other vertex in G_f by edge with cost $c_f(e) = 0$.
3. Compute the shortest path tree (with respect to costs c_f) from the dummy vertex s to all other vertices.
4. Recover a negative cycle or state that it does not exist.

See the lectures on Shortest Path Problem for more details.

[1] Compare this to the labeling procedure in Ford-Fulkerson algorithm.

1.4 Finding the initial feasible balanced flow

Similarly as in Ford-Fulkerson for Maximum Flow problem, we need to start the cycle canceling algorithm with some feasible flow. Luckily, this problem can be solved easily by Ford-Fulkerson algorithm we have developed in the last homework assignment:

1. Introduce dummy vertices s, t to the original graph G.
2. Connect s with vertices $v \in V$, if $b(v) > 0$, set $l(s, v) = 0$ and $u(s, v) = b(v)$.
3. Connect vertices $v \in V$ with t, if $b(v) < 0$, set $l(v, t) = 0$ and $u(v, t) = -b(v)$.
4. Find maximum flow f in this graph by Ford-Fulkerson algorithm we developed in the last homework assignment.
5. If f saturates all outgoing edges from s, then disconnect s and t from G. Now, the flow f is a feasible balanced flow for the original minimum-cost flow problem.

See more details in the lectures on flows.

2 Homework: Object Tracking in Videos

Object tracking in videos [1] is a problem where multiple objects have to be tracked across all frames of a video. Tracking is not concerned with just detecting the positions of the objects, but also their identity. An interesting example is a player tracking in sport games. Player tracking systems are commonly used to collect various statistics about the games and players. In football, for example, these statistics include percentage of ball possession by each team, average speed of the players, traveled distance, etc. To compute these statistics, the systems need to track the individual players on the field. To achieve this, a video is being taken of the game and in each frame of this video, the position of all the players on the field are detected using Machine Learning algorithms[2]. These positions, however, do not contain any information about which players they represent. This is done in the next step called *data association*, where the detected positions in a video frame are *associated* with the individual players. Association is done using the information from the current and previous frame.

Consider Fig. 2 representing k-th frame containing 22 detected and associated positions, i.e., already identified with the individual players.

[2] In the homework we won't concern ourselves with detecting objects in the frames as this is outside the course scope.

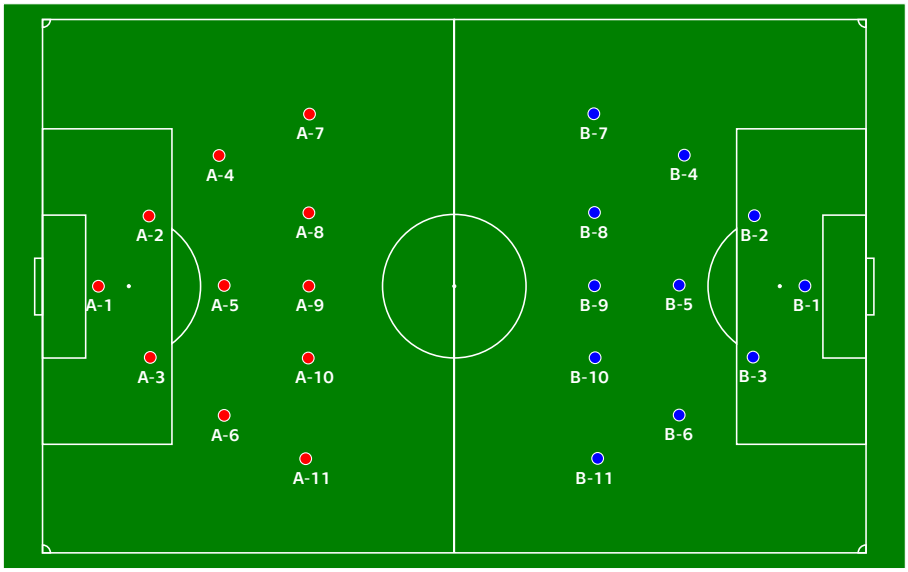


Figure 2: k-th frame with detected and associated player positions.

In the next frame $k+1$, the players run a little bit and end up in new positions. Fig. 3 shows the newly detected positions which are not yet associated with the individual players.

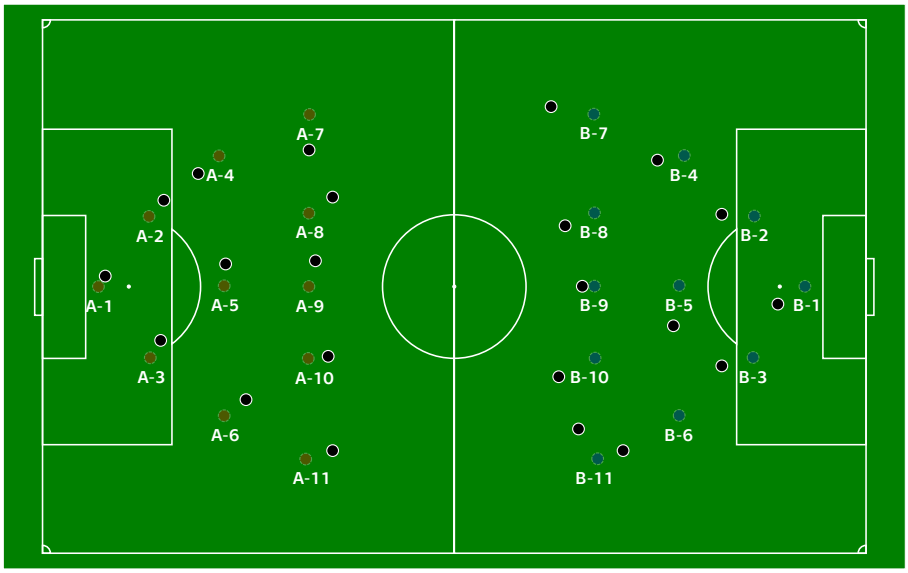


Figure 3: $(k+1)$-th frame with the newly detected positions (shown in black) not yet associated with the individual players. The player positions from frame k (shown in transparent colors) are also shown.

Using the information from frame k and frame $k+1$, we can associate the new positions with the individual players using a method from Section 2.1, which will result in a tagged frame illustrated in Fig. 4. This way, we can track the movement of the individual players during the whole game.

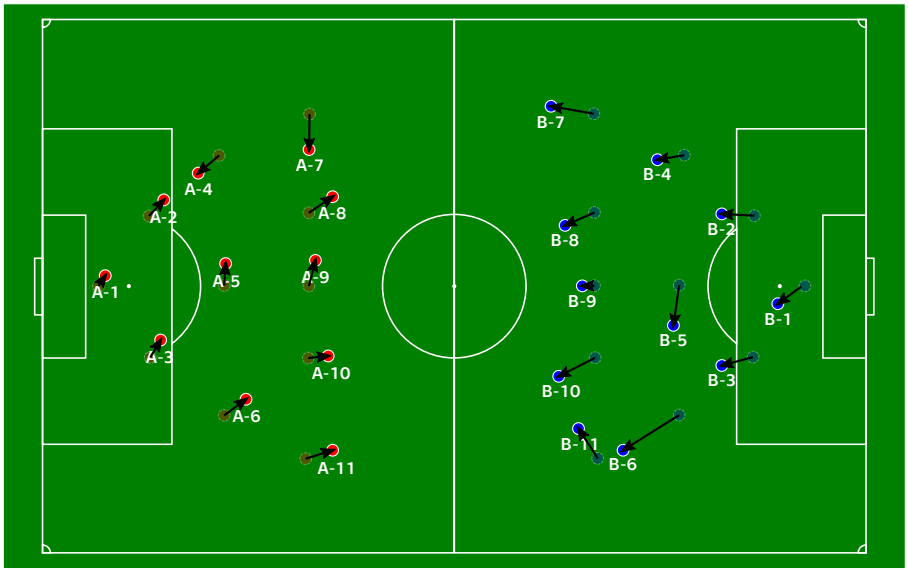


Figure 4: $(k+1)$ frame with detected and associated player positions. The movement of the individual players from frame k to frame $k+1$ is illustrated by arrows.

2.1 Data Association

Let n denote the number of objects, e.g., players. We are given a set of p frames $\{\mathbf{F}^{(1)}, \ldots, \mathbf{F}^{(p)}\}$, where each frame $\mathbf{F}^{(k)} \in \mathbb{N}^{n \times 2}$ is a matrix representing the (x, y) integer positions of the objects in the k-th frame. Each row $\mathbf{f}_i^{(k)}$ of the matrices represents one position. The goal of the data association problem is to find $p-1$ bijective functions $\{a^{(1)}, \ldots, a^{(p-1)}\}$, where each function $a^{(k)} : \{1, \ldots, n\} \to \{1, \ldots, n\}$ associates the positions in frame $\mathbf{F}^{(k)}$ with positions in frame $\mathbf{F}^{(k+1)}$.

Probably the simplest algorithm for finding $a^{(k)}$ is to compute an euclidean distance between each tuple of positions in k-th and $(k+1)$-th frame and then greedily associate the positions that are closest together. However, such algorithm will not always give an expected result; think about what would happen in Fig. 3 if the position in frame 1 corresponding to player B-11 would be associated with the nearest position in frame $k+1$.

A better approach is to minimize the total distance between all the associated positions. Formally, let $\mathbf{D}^{(k)} \in \mathbb{R}^{n \times n}$ be the matrix of the euclidean distances between the positions in k-th and $(k+1)$-th frame, i.e.,

$$d_{i,j}^{(k)} = \text{euclid}\left(\mathbf{f}_i^{(k)}, \mathbf{f}_j^{(k+1)}\right) \tag{1}$$

Then we select function $a^{(k)}$ such that the total distance between the associated positions is minimized, i.e.,

$$\min_{a^{(k)}} \sum_{i=1}^{n} d_{i,a^{(k)}(i)}^{(k)} \tag{2}$$

This is the famous *assignment problem* which can be solved using minimum-cost flow algorithms such as cycle-canceling.

Note that we could also try to find the optimal associations at once by optimizing multi-layered assignment problem containing all the frames. However, such approach could be inefficient since the videos usually contain tens of thousands frames. For this reason, we will limit ourselves in this problem to finding locally optimal associations between two consecutive frames.

A homework assignment: Implement the cycle-canceling algorithm for solving the minimum-cost flow problem. Use the implemented algorithm for solving the data association problem in object tracking. Using Gurobi in this homework is **not allowed**.

Additional material: See https://old.datahub.io/dataset/magglingen2013 for a real-world dataset of recorded football matches.

2.2 Input/Output format

Your program will be called with two arguments: the first one is absolute path to input file and the second one is the absolute path to output file which has to be created by your program.

The input file has $p+1$ lines and has the following form

$$
\begin{array}{llllllll}
n & p & & & & & & \\
f_{1,1}^1 & f_{1,2}^1 & f_{2,1}^1 & f_{2,2}^1 & \cdots & f_{n,1}^1 & f_{n,2}^1 \\
f_{1,1}^2 & f_{1,2}^2 & f_{2,1}^2 & f_{2,2}^2 & \cdots & f_{n,1}^2 & f_{n,2}^2 \\
\vdots & & & & & & \\
f_{1,1}^p & f_{1,2}^p & f_{2,1}^p & f_{2,2}^p & \cdots & f_{n,1}^p & f_{n,2}^p
\end{array}
$$

One space is used as a separator between values on one line. All the values in the input file are non-negative integers.

The output file has $p-1$ lines and has the following form

$$
\begin{array}{llll}
a^{(1)}(1) & a^{(1)}(2) & \ldots & a^{(1)}(n) \\
a^{(2)}(1) & a^{(2)}(2) & \ldots & a^{(2)}(n) \\
\vdots & & & \\
a^{(p-1)}(1) & a^{(p-1)}(2) & \ldots & a^{(p-1)}(n)
\end{array}
$$

One space is used as a separator between values on one line. All the values in the output file are positive integers.

2.2.1 Example 1

Input:

```
3 5
53 65 70 62 35 79
60 9 94 90 98 27
33 50 79 69 91 6
6 87 46 16 67 74
44 54 58 68 89 78
```

Output:

```
1 3 2
1 2 3
1 3 2
2 1 3
```

References

[1] Y. Cai, N. de Freitas, and J. J. Little, "Robust visual tracking for multiple targets," in *Computer Vision – ECCV 2006* (A. Leonardis, H. Bischof, and A. Pinz, eds.), (Berlin, Heidelberg), pp. 107–118, Springer Berlin Heidelberg, 2006.