Multi-object Tracking by Kalman Filter and Hungarian Matching Algorithm

Hy Truong Son

June 2017

1 Introduction

This paper aims to solve the problem of multi-object tracking in Computer Vision by Kalman filter and Hungarian matching algorithm on bipartite graph. We experimented our method with a synthetic dataset that simulates robotic soccer. The program is implemented in Java programming language with Graphical User Interface.

2 Object Detection Algorithm

2.1 Color Detection

Define an image of n rows and m columns as a function $\mathcal{I} : [n] \times [m] \rightarrow \Re^3$ that maps each pixel (i, j) where $1 \leq i \leq n$ and $1 \leq j \leq m$ to a triple of red, green and blue.

A classification of colors can be defined as a function $f : \Re^3 \rightarrow \{1, .., \mathcal{C}\}$ that maps from the space of red, green and blue into the set of $\mathcal{C}$ pre-defined colors.

Given an input image $\mathcal{I}$ and a color classification f, the color detection algorithm that produces an output image $\hat{\mathcal{I}} : [n] \times [m] \rightarrow \{1, .., \mathcal{C}\}$ can be described by the following pseudo-code:

function Color Detection (Image $\mathcal{I}$, Color classification f)
    for $i = 1 \rightarrow n$:
        for $j = 1 \rightarrow m$:
            $\hat{\mathcal{I}}(i, j) \leftarrow f(\mathcal{I}(i, j))$
        end for
    end for
    return $\hat{\mathcal{I}}$
end function

2.2 Breadth-First Search

Suppose that each object has the same color in the set of the detected colors $\{1, .., \mathcal{C}\}$. We apply Breadth-First Search algorithm to find all the connected components in the image such that each connected component has the same color. Two pixel (x_1, y_1) and (x_2, y_2) are adjacent if and only if $|x_1 - x_2| \leq 1$ and $|y_1 - y_2| \leq 1$. The object detection algorithm by BFS can be described by the following pseudo-code:

function Object Detection (Image $\hat{\mathcal{I}}$)
    Initialize a marking image that checks if a pixel is visited by BFS
    $M : [n] \times [m] \rightarrow \{T, F\}$
    $M(x, y) \leftarrow F, \forall (x, y) \in [n] \times [m]$
    Initilize the number of objects
    $\mathcal{O} \leftarrow 0$
    Brute-force each pixel in the image
    for $x = 1 \rightarrow n$:
        for $y = 1 \rightarrow m$:
            if $M(x, y) = F$:
                Find the new object
                $\mathcal{O} \leftarrow \mathcal{O} + 1$
                BFS $\big(\hat{\mathcal{I}}, (x, y)\big)$
            end if
        end for
    end for
end function

function BFS (Image $\hat{\mathcal{I}}$, Pixel (x_0, y_0))
    Initialize a queue of pixels
    $\mathcal{Q} \leftarrow \emptyset$
    Add the first pixel into the queue
    $\mathcal{Q} \leftarrow \{(x_0, y_0)\}$ and $M(x_0, y_0) \leftarrow T$
    The color of this object
    $c \leftarrow \mathcal{I}(x_0, y_0)$
    Search for all pixels in the connected component
    while $\mathcal{Q} \neq \emptyset$:
        Pop a pixel (i, j) from $\mathcal{Q}$
        for $\Delta_x = -1 \rightarrow 1$:
            for $\Delta_y = -1 \rightarrow 1$:
                $x \leftarrow i + \Delta_x$
                $y \leftarrow j + \Delta_y$
                if $1 \leq x \leq n$ and $1 \leq y \leq m$ and $\mathcal{I}(x, y) = c$ and $M(x, y) = F$:
                    $M(x, y) \leftarrow T$
                    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(x, y)\}$
                end if
            end for

19. end for
20. end while
21. *All pixels that has been in the queue belongs to this object*
end function

3 Kalman Filter

3.1 Random Accelerations Model [1]

Consider N objects moving in a 2D plane. Let $x_t^{(i)}$ and $y_t^{(i)}$ be the horizontal and vertical locations of object $i \in \{1, .., N\}$ at time t, and $\Delta x_t^{(i)}$ and $\Delta y_t^{(i)}$ be the corresponding velocity. We can represent this as a state vector $\boldsymbol{x}_t \in \Re^{4N}$ as follows:

$$\boldsymbol{x}_t^T = \begin{bmatrix} x_t^{(1)} & y_t^{(1)} & \cdots & x_t^{(N)} & y_t^{(N)} & \Delta x_t^{(1)} & \Delta y_t^{(1)} & \cdots & \Delta x_t^{(N)} & \Delta y_t^{(N)} \end{bmatrix}$$

Let us assume that the object is moving at constant velocity, but is *perturbed* by random Gaussian noise. Thus we can model the system dynamics as follows:

$$\boldsymbol{x}_t = A_t \boldsymbol{x}_{t-1} + \boldsymbol{\epsilon}_t$$

$$\begin{bmatrix} x_t^{(1)} \\ y_t^{(1)} \\ \vdots \\ \Delta x_t^{(1)} \\ \Delta y_t^{(1)} \\ \vdots \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_{t-1}^{(1)} \\ y_{t-1}^{(1)} \\ \vdots \\ \Delta x_{t-1}^{(1)} \\ \Delta y_{t-1}^{(1)} \\ \vdots \end{bmatrix} + \boldsymbol{\epsilon}_t$$

where I is the identity matrix of size $2N \times 2N$ and

$$A_{11} = I \quad A_{12} = \Delta \cdot I \quad A_{21} = I \quad A_{22} = I$$

and $\boldsymbol{\epsilon}_t \sim \mathcal{N}(\mathbf{0}, R)$ is the system noise, and Δ is the sampling period.

Suppose that we can observe the location of the object but not its velocity. Let $\boldsymbol{z}_t \in \Re^{2N}$ represent our observations, which we assume is subject to Gaussian noise. We can model this as follows:

$$\boldsymbol{z}_t = C_t \boldsymbol{x}_t + \boldsymbol{\delta}_t$$

$$\begin{bmatrix} \hat{x}_t^{(1)} \\ \hat{y}_t^{(1)} \\ \vdots \end{bmatrix} = \begin{bmatrix} C_1 & C_2 \end{bmatrix} \begin{bmatrix} x_t^{(1)} \\ y_t^{(1)} \\ \vdots \\ \Delta x_t^{(1)} \\ \Delta y_t^{(1)} \\ \vdots \end{bmatrix} + \boldsymbol{\delta}_t$$

where

$$C_1 = I \qquad C_2 = \mathbf{0}^{2N \times 2N}$$

and $\boldsymbol{\delta}_t \sim \mathcal{N}(\mathbf{0}, Q)$ is the measurement noise.

Remark that:

$$A_t \in \Re^{4N \times 4N} \quad R \in \Re^{4N \times 4N} \quad C_t \in \Re^{2N \times 4N} \quad Q \in \Re^{2N \times 2N}$$

3.2 Kalman Filter Algorithm

Suppose that there is no control u_t. The Kalman Filter for multi-object tracking can be described as the following pseudo-code:

function Kalman Filter

Initialization

$\mu_0 \leftarrow \boldsymbol{x}_0$

$\Sigma_0 \leftarrow I$

Filtering

for $t = 1 \to \infty$:

  Prediction

  $\bar{\mu}_t \leftarrow A_t \mu_{t-1}$

  $\bar{\Sigma}_t \leftarrow A_t \Sigma_{t-1} A_t^T + R_t$

  Get a new measurement $\boldsymbol{z}_t$

  Update

  $K_t \leftarrow \bar{\Sigma}_t C_t^T (C_t \bar{\Sigma}_t C_t^T + Q_t)^{-1}$

  $\mu_t \leftarrow \bar{\mu}_t + K_t(\boldsymbol{z}_t - C_t \bar{\mu}_t)$

  $\Sigma_t \leftarrow (I - K_t C_t) \bar{\Sigma}_t$

  $\boldsymbol{x}_t \leftarrow \mu_t$

end for

end function

4 Hungarian Matching Algorithm

4.1 Matching Objects to Measurements

One problem with the object detection algorithm is that it can only return to us the set of rectangles that cover objects in an image. That means vector measurement $\boldsymbol{z}_t$ can appear in an unexpected permutation most of the time. In addition, the object detection algorithm can be very noisy. It can detect two or more objects overlapping as a single one. It can mis-detect some objects also due to environmental conditions such as light, background, etc. The simple heuristics that matches each current object position to the nearest measurement by Euclidean distance cannot work. The reason is that one measurement can be matched by multiple objects. Remark that objects are moving without any controls, the only thing that we can observe is the input image.

By all these above reasons, we need the Hungarian matching algorithm on bipartite graph. The bipartite graph has two sides X and Y. Each vertex of side X represents for an object position. Each vertex of side Y represents for a measurement. The cost of matching a vertex of X to another vertex of Y is the Euclidean distance between the corresponding object position and the corresponding measurement.

In the case that we do not have enough measurement, $|X| > |Y|$, we add some more virtual vertices to Y and the Euclidean distances from X to these new vertices are set to be infinity. For simplicity, we only consider the case when $|X| = |Y|$.

4.2 Maximum Flow Minimum Cost

We can solve the Hungarian matching problem efficiently by Kuhn-Munkres algorithm. In this paper, we introduce another way of solving it by Maximum Flow Minimum Cost. First of all, we construct our flow graph by as following:

- Create a virtual source vertex s
- Create a virtual sink vertex t
- Connect s to all vertices in X with cost 0 and edge capacity 1
- Connect all vertices in Y to t with cost 0 and edge capacity 1
- Connect all vertices in X to all vertices in Y with the cost as the Euclidean distance and edge capacity 1

Let the new constructed graph be $G = (V, E, c, f, w)$ where V is the set of vertices that includes X, Y, s and t; E is the set of edges that we constructed as above; $c : E \to \Re$ is the edge capacity; $f : E \to \Re$ is the flow carrying on the edge; and $w : E \to \Re$ is the weight of the edge. The maximum flow minimum cost between s and t gives us the optimal solution of Hungarian matching problem.

We will apply the modified Ford-Fulkerson algorithm with Bellman-Ford algorithm (instead of Breadth First Search) to find minimum-cost augmenting paths from s to t.

For the Bellman-Ford algorithm, we will use Rounded Queue data structure (for speed-up the original version of Bellman-Ford) with PUSH, POP operators and IS-EMPTY query as following. We maintain a global boolean array **inqueue** of size $|V|$: **inqueue**$[v]$ = True if vertex v is currently in the queue, otherwise False. We maintain a global integer array **queue** of size $|V|$ indexed from 0 to $|V| - 1$. We maintain two integers: **rear**, **front** for the ending and starting positions of the **queue**. We implement the operators:

```
procedure PUSH(v ∈ V):
    if inqueue[v] = True:
        return
    end if
    queue[front] ← v
    front ← (front + 1) mod |V|
    inqueue[v] ← True
end procedure
```

```
function POP():
    v ← queue[rear]
    rear ← (rear + 1) mod |V|
    inqueue[v] ← False
    return v
end function
```

```
function IS-EMPTY():
    if rear = front:
        return True
    end if
    return False
end function
```

We implement Bellman-Ford algorithm finding the min-cost augmenting path. We have an integer array d of size $|V|$ such that $d[v]$ stores the minimum **distance/cost** from vertex s to v. We maintain an integer array π of size $|V|$ such that $\pi[v]$ is the previous vertex of v in the minimum cost path.

```
function Bellman-Ford(G(V, E, c, f, w), s ∈ V, t ∈ V):
    Initialization for d and π
    d[v] ← ∞ (∀v ∈ V)
    π[v] ← NIL (∀v ∈ V)
    Initialization for the Rounded-Queue data structure
    rear ← 0
    front ← 0
    Initialization for the vertex s
    d[s] ← 0
    PUSH(s)
    Main algorithm
    while not IS-EMPTY():
        u ← POP()
        for each v ∈ V:
            if (u, v) is a forward edge:
            if c_f((u, v)) = c((u, v)) − f((u, v)) > 0 or equivalently (u, v) ∈ G_f:
```

```
            $\delta \leftarrow w\big((u,v)\big)$
            if $d[v] > d[u] + \delta$:
                $d[v] \leftarrow d[u] + \delta$
                $\pi[v] \leftarrow u$
                PUSH($v$)
            end if
        end if
    end if
    if $(u,v)$ is a **backward** edge:
        if $c_f\big((u,v)\big) = f\big((v,u)\big) > 0$ or equivalently $(u,v) \in G_f$:
            $\delta \leftarrow w\big((v,u)\big)$
            if $d[v] > d[u] - \delta$:
                $d[v] \leftarrow d[u] - \delta$
                $\pi[v] \leftarrow u$
                PUSH($v$)
            end if
        end if
    end if
  end for
end while
We construct the path (as a list of vertices) $P$ from $\pi$
if $\pi[t]$ = False:
    **return NIL**
end if
$P \leftarrow \emptyset$
$v \leftarrow t$
while True:
    $P \leftarrow \{v\} \cup P$
    if $v = s$:
        **break**
    end if
    $v \leftarrow \pi[v]$
end while
**return** $P$
end **function**
```

We implement Ford-Fulkerson to find the maximum-flow minimum-cut s-t:

```
**function** Ford-Fulkerson($G(V,E,c,f,w)$, $s \in V$, $t \in V$):
    Flow initialization
    $f(e) \leftarrow 0$ ($\forall e \in E$)
    Compute the residual graph $G_f$
    Finding flow as follows:
    while True:
        $P \leftarrow$ Bellman-Ford($G(V,E,c,f,w)$, $s$, $t$)
        if $P$ = **NIL**:
```

```
            break
        end if
        Δ ← min_{e∈P} c_f(e)
        for each e = (u, v) ∈ P:
            if e is a forward edge:
                f(e) ← f(e) + Δ
            else e is a backward edge:
                f((v, u)) ← f((v, u)) − Δ
            end if
        end for
    end while
end function
```

Time analysis:

- Bellman-Ford algorithm with Rounded-Queue data structure: $O(|V|.|E|)$. In practice, it will run much **faster** than the original version.
- Number of iterations in Ford-Fulkerson algorithm: $O(|V|.|E|)$.
- Total complexity: $O(|V|^2|E|^2)$.

5 Software

5.1 Synthetic Dataset

Each image frame contains the following components:

- NAO robots
- Red balls
- A fixed soccer field as the background

The users can:

- Choose the number of robots
- Choose the number of balls
- Choose the size of robots
- Choose the size of balls
- Choose the number of pixels robots and balls can move in a time step
- Choose the probability that robots and balls change their current directions (otherwise they continue their moving directions)
- Choose to speed up or slow down the experiment (with some time delay)

When you start the experiment, there will be two frames, one frame as the input images from the camera, another frame shows the calibration image with object detection results and Kalman filter results.

5.2 Software Components

Main components/classes supporting the algorithms:

- Ball detection: Algorithms/BallDetection.java
- Robot detection: Algorithms/RobotDetection.java
- Linear algebra (matrix multiplication, matrix inverse): Algorithms/Matrix.java
- Image (matrix) resize or normalization: Algorithms/Normalization.java
- Hungarian matching algorithm: Algorithms/Hungarian_ Matching.java
- Kalman Filter: Algorithms/KalmanFilter.java

Main components/classes supporting Graphical User Interface:

- Starting frame to choose parameters: GUI/StartingFrame.java
- About the author (me) frame: GUI/AboutFrame.java
- Showing the camera and the filter images: GUI/ShowFrame.java
- Agent that creates a synthetic image then tracks the objects: GUI/Agent.java

Other special components/classes:

- Coordinate in 2D: GUI/Coordinate.java
- Rectangle in 2D (contains the center): GUI/Rectangle.java

To run the program:

```
javac MainProgram.java
java MainProgram
```

References

[1] Kevin R. Murphy, “Machine Learning: A Probabilistic Perspective”, Chapter 18. State Space Models, *MIT Press*, 2012