FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Matěj Hrbáček

Construction of time-space trajectories from multimodal data

Department of Software Engineering

Supervisor of the master thesis:	prof. RNDr. Tomáš Skopal, Ph.D.
Study programme:	Computer Science
Study branch:	Software and Data Engineering

Prague 2023

I want to thank my supervisor, prof. RNDr. Tomáš Skopal, Ph.D., for all the patient guidance, help, comments and valuable advice. I also want to thank my colleagues and friends for all the support and help with thesis-related stuff, such as video recording. Finally, I would like to thank my girlfriend and family for their support and encouragement during my studies.

Title: Construction of time-space trajectories from multimodal data

Author: Matěj Hrbáček

Department: Department of Software Engineering

Supervisor: prof. RNDr. Tomáš Skopal, Ph.D., Department of Software Engineering

Abstract: With the growth of public camera recordings and video streams in recent years, there is an increasing need for automatic processing with limited human input. An important part of the process is detecting moving objects in the video and grouping individual detections across video frames into trajectories. This thesis presents a set of algorithms for creating trajectories from object detections while using a configurable analytic model. Presented algorithms are based on the clustering of detections, later even simple trajectories, into complex trajectories by their features, such as a timestamp (frame), bounding rectangle in the video frame and optionally, image crop defined by the bounding rectangle. To present the usage of the generated trajectories, we then introduce methods for further analysis and data extraction. The first method improves the input detections by adding missing detection due to the detector error. The second one is creating a simple semantic description of trajectories to enable further research, such as action analysis or trajectory searching.

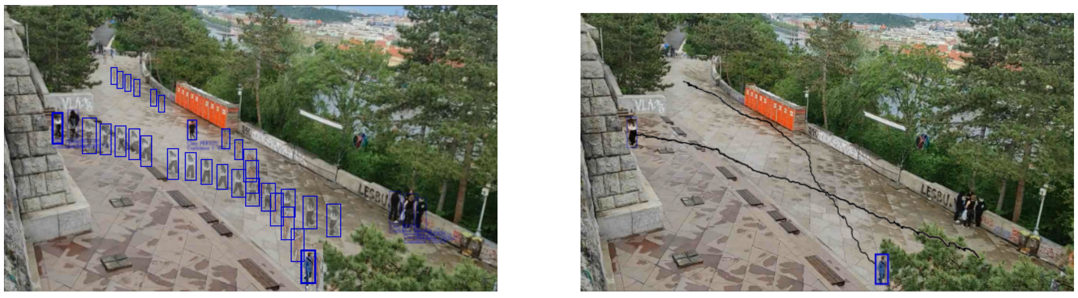
Keywords: video analytics · trajectory generation · trajectory description · object tracking · detection improving

Contents

Introduction

In recent years, there has been considerable growth in the number of security cameras and video records/streams, while it is desirable to process such data automatically. If detections of objects are available, it is possible to track their movement into trajectories, which is a more semantic (behavioural) feature than a simple detection of an object's appearance in a single video frame. Practical applications of trajectories can be found in various fields including activity analysis [21][11][17], traffic monitoring [36][10], autonomous driving [7][20][24], or medicine [25].

In this thesis, we are going to introduce our pipeline, called *Traged*, on generating trajectories of humans in outdoor videos for large public spaces such as squares or promenades. Object tracking is a topic that has been addressed by many authors and an overview of some methods can be found in chapter 1. The input for our algorithm consists of a video, generated detections of humans in the given video and a configuration file. Each detection contains information such as timestamps, bounding rectangles in the video frames, and image crops defined by the rectangle. Our goal is to generate trajectories from the input. As a trajectory, we would like to have a set of detections representing the trajectory of a moving person. From this set, we can extract other information, such as the motion curve of the object (see Figure 1). In the end, we will implement the described algorithm as an extension of the system Videolytics[32] and evaluate the results.


(a) Trajectory as a set of detections

(b) Motion curve

Figure 1: Illustration of generated trajectories

The motivation for generating trajectories is to describe objects throughout time and space. Trajectories can be used for various analyses, including tracking the movement of people in public spaces for security or marketing purposes. Also, obtaining trajectories opens up a whole new world of possibilities for improving existing systems as well as adding new features to detections.

We encountered a problem with generated detections by the detector. Commonly, objects are not detected by detectors in every single frame of the video, even though the object is visible in the video. It can be seen as glitching bounding boxes in video playthroughs. Thus, one planned use of generated trajectories will be to generate missing detections by interpolating their position based on given trajectories. Even though it is not essential, it certainly gives a solid foundation for future detection experiments. Trajectories can also be used for the behaviour/action analysis of objects. We start from the assumption that similar actions share similar patterns. These patterns can be described, and the description can be used for looking up similar actions in different videos.

Thesis structure

The thesis is organized into the following chapters. In the beginning, in chapter 1, we have created an overview of related works and systems. Chapter 2 is an overall description of our *Traged* pipeline with idea description and input and output data. Later than in chapters 3,4,5 are in detail described parts of *Traged*. The implementation of our approach is described in chapter 6. Finally, the evaluation of implementation is described in chapter 7.

Thesis contributions

The thesis presents a complex flow, processing detections of objects from a video into complete trajectories, with added missing detections and semantically described and prepared for further analysis. The algorithms of the thesis can be divided into four parts:

- Overview of identified features, usable to detections clustering
  - Overview and proposal of features that can be used to cluster detections of an object into trajectory.
- The algorithm clustering detections into complex trajectories
  - We propose a complex algorithm that clusters detections into a trajectory of one object.
- Generating of missing detections for objects
  - Algorithm designed to interpolate generated trajectory and generate missing/undetected detections.
- Semantic description of generated trajectories
  - Creating a foundation for semantic description and analysis of trajectories.

1. Related work

In this chapter, we would like to overview related methods to our approach in section 1.1 and briefly introduce related systems where our approach could be implemented in section 1.2.

1.1 Related methods

Since the early beginnings of automatic video analysis, a solution to object tracking and trajectory creation in videos has been needed due to its valuable potential usage. Many research projects have been carried out on the topic at the time of writing, using many different methods. The most common approach to generating trajectories is computer vision algorithms. The movement of tracked objects can be represented using Kalman filter [37], network flows [13], or deep neural networks, to name a few.

1.1.1 Neural networks, transformers

Sun et al. [34] proposed a Deep Affinity Network (DAN) network. Like other deep network solutions for multiple object tracking, DAN works in two essential steps. First, it detects objects within individual frames and harnesses their features with multiple levels of abstraction using a convolutional network. Next, the extracted metadata is used to group detections within pairs of frames. Exhaustive pairing permutations of these features are trying to find the most feasible affinities between these detections. Another noteworthy feature of DAN is the ability to account for new and disappearing objects thanks to its ability to look forward and backwards in time, recognizing ”unaffiliated” objects.

Shuai et al. [31] introduced a region-based Siamese Multi-Object Tracking network called SiamMOT, inspired by the Siamese-based single-object tracking approach. Their approach takes advantage of region-based features and template matching to estimate the motion of an object between two frames.

Sun et al. [33] tried to solve the task using transformer architecture. The inspiration came from the query-key mechanism previously used for single object tracking. To extend it to multiple object tracking, they had to consider newly created objects without features. To do that, they used two sets of keys with object queries. One takes care of newly introduced objects, while the other bears information about objects in the previous frame. Therefore, they generate two sets of bounding boxes and use IoU matching to generate the final ordered object set.

TrackFormer by Meinhardt et al. [26] is a multi-object tracking approach based on encoder-decoder transformer architecture. It uses queries for the track-by-attention paradigm. These queries are used to detect new and existing objects throughout the track. Static queries are used for new detections, whereas track queries are used to hold attention to existing objects. These track queries consist of identity features of objects in previous frames while also adapting to a potential change in the next. Another notable feature of this approach is the attention span during which track queries are kept in memory and, therefore, can be used for re-introducing an object, should it reappear in the video.

1.1.2 Methods in OpenCV

This section provides an overview of related methods implemented in the OpenCV[1] library.

Real-time tracking via online boosting.

Grabner et al. [12] proposed a method capable of online training, which is beneficial when an object changes appearance, including the change of illumination. The method is based on an online version of the AdaBoost algorithm. The algorithm uses only grey value information and utilises fast computable features such as Haar-like wavelets to stabilize tracking results.

Visual tracking with online multiple instance learning.

Babenko et al. [2] addressed the problem with traditional supervised learning methods - slight inaccuracies in the tracker that can lead to incorrectly labelled training examples. They presented a Multiple Instance Learning algorithm, resulting in a more robust tracker with fewer parameter tweaks.

Forward-backward error: Automatic detection of tracking failures.

Kalal et al. [15] proposed an object tracker called Median Flow. They define the Forward-Backward error of feature point trajectories, meaning that the tracking is performed forward and backwards in time. As follows, the discrepancies between the two trajectories are measured. This approach makes it possible to detect tracking failures and select reliable trajectories.

Visual object tracking using adaptive correlation filters.

Bolme et al. [3] tried to solve the problem with correlation filters. They presented a new type of correlation filter. Firstly, adaptive correlation filters are used to model the target's appearance, and then the tracking is performed via convolution. A tracker based on this filter is invariant to changes in the appearance of objects, such as lighting, scale, pose, and non-rigid deformations.

Tracking-learning-detection.

Kalal et al. [16] proposed a framework that consists of 3 notable parts: a detector, a tracker and a learner. The detector looks for detections, the tracker looks for trajectories, and the learner is used for computing errors, which are then used to minimize similar errors in the future. Their method uses P-N learning with two "experts" - The p-expert estimates missed detections, and N-expert estimates false alarms. The system works in real-time.

[1] `https://opencv.org/`

Discriminative Correlation Filter with Channel and Spatial Reliability.

Lukezič et al. [23] based their tracker on the discriminative correlation filter method. They called it CSR-DCF, the Discriminative Correlation Filter with Channel and Spatial Reliability. The filter support is adjusted to the portion of the object that is acceptable for tracking using the spatial reliability map. This enhances the tracking of non-rectangular objects and enables a larger search region. Reliability ratings serve as feature weighting coefficients in localization and indicate the channel-wise quality of the trained filters.

1.2 Related systems

This section provides a short overview of systems focusing on video analysis, especially object tracking, where we could use our approach or where similar functionality is implemented. The larger overview of other systems or older systems can be found in articles[32][28][29]. We focused on newer and still active systems or tools.

1.2.1 Commercial systems

Almost all relevant systems with active support and working pages are commercial. Thus, we are just providing brief information and links to official pages. Unfortunately, the non-commercial systems from articles are often abandoned or inactive.

Senstar

Senstar company provides commercial software, a video analytical tool[2], that processes IP video into business intelligence. The provided software collects video records and analyzes them with machine learning algorithms. The system is used for license plates recognition, face recognition, crowd detection and people tracking[3],

Google Cloud Video Intelligence API

Google Cloud Video Intelligence API[4] is a video analysis tool that uses machine learning to recognize and analyze objects, places and actions in stored and streaming videos. The API is split on `Vertex AI for AutoML`[5] and `Video Intelligence API`[6], where both provides object tracking.

[2]https://senstar.com/products/video-analytics/

[3]https://senstar.com/products/video-analytics/indoor-people-tracking/

[4]https://cloud.google.com/video-intelligence

[5]https://cloud.google.com/vertex-ai/docs/training-overview

[6]https://cloud.google.com/video-intelligence/docs/annotate-video-command-line

Cyberlink - people tracker

CyberLink's People Tracker[7] is a surveillance video analytics system using cutting-edge AI technology to track people's movements. It can be used for security reasons as well as for people searching.

Visio ai

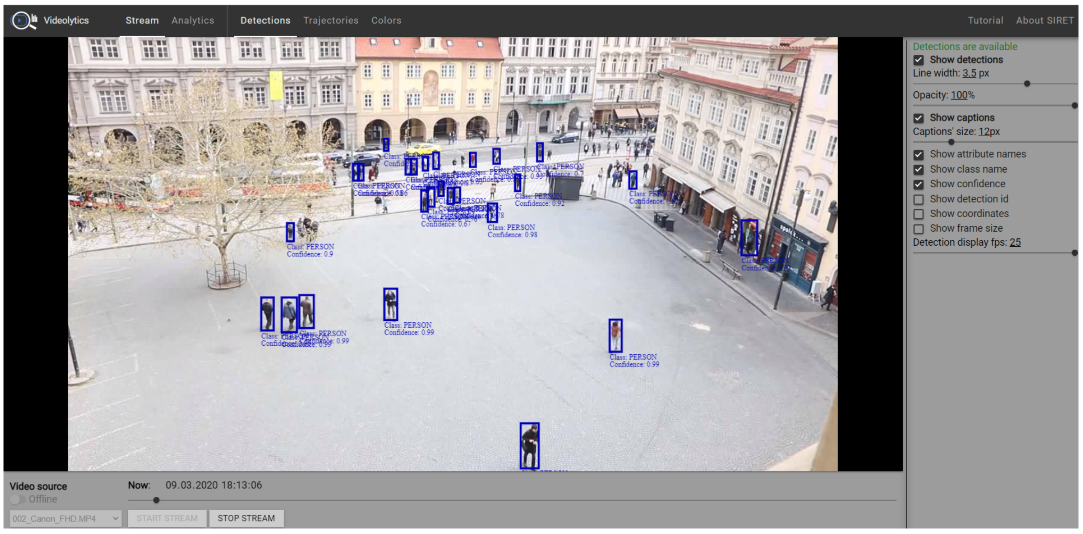
Visio AI[8] is a computer vision platform for building real-time computer vision and deep learning applications. Visio AI platform provides tools for ergonomic risk analysis, object counting, automatic number plate recognition, and object tracking.

1.2.2 Videolytics

The system Videolytics, founded by Skopal et al. [32], is a non-commercial web-based system [35] for advanced video analysis, working on data extraction and visual analysis of extracted data.

The whole pipeline of Videolytics is based on independent modules, where the only point of communication between the modules is a database. This means that each module extracts data from the video while using data from the database, and newly generated results are stored in the database. The Videolytics's architecture is present in Dobranský and Skopal[9].

Videolytics, by its architecture, combines the "best" of deep learning and hand-designed analytical models to use the advantages of both methods. Currently, Videolytics provides modules for detection extraction, detection and trajectory visualization and visual data searching.



Detections are available
☑ Show detections
Line width: 3.5 px
Opacity: 100%
☑ Show captions
Captions' size: 12px
☑ Show attribute names
☑ Show class name
☑ Show confidence
☐ Show detection id
☐ Show coordinates
☐ Show frame size
Detection display fps: 25

Video source
Offline
002_Canon_FHD.MP4
Now: 09.03.2020 18:13:06

Figure 1.1: Videolytics web application

[7]https://www.cyberlink.com/faceme/solution/people-tracker/overview

[8]https://viso.ai/

Traged extension

Based on the system parameters, we have decided to implement the approach designed in the thesis as a new module of Videolytics. Since all communication is handled through the database, the potentially implemented module will be independent and transferable simply by changing the database connector. And we could fully use the implemented modules of Videolytics.

2. Traged

In the current situation, almost all problems related to video analysis and object tracking are solved by deep learning [34], [31], [33], [26]. However, this approach needs a lot of annotated data, which we do not have and would like to avoid needing large sets. Also, during the research on detection generators, we encountered that the generated detections are not ideal, and objects are not detected correctly in all frames. Thus, we have taken this fact into account.

To satisfy those restrictions, we have tried a different approach and designed an algorithm that generates trajectories by grouping (or clustering) detections. We have decided to create a fully configurable analytical model and test different grouping methods. The designed algorithm, as a data input, accepts raw video, detections of objects, and a configuration with information on how to perform clustering. We called this approach *Traged* (TRAjectories GEnerated from Detections).

2.1 Definitions

Before we start with the algorithm itself, we will unify our terminology and define terms that will stay with us throughout the thesis and are crucial to understand.

Definition 1. *Detection* represents an object of interest in a single frame.

Each detection contains information such as

- timestamp - Absolute time of frame capture with this detection.[1]
- bounding box - Rectangle in the video frame, bounding the object.
  $\longrightarrow$ width of the object detection
  $\longrightarrow$ height of the object detection
  $\longrightarrow$ center_x of the object detection
  $\longrightarrow$ center_y of the object detection
- image crop - Image cropped from frame defined by the bounding box.

Definition 2. *Detections* represent all detections of one video.

Definition 3. *DFrame* represents a set of detections detected in a single frame of video.

$$DFrame(x) = \{d | d \in \mathit{Detections} \land \mathit{d.timestamp} = x\}$$

Definition 4. A *trajectory* is a set of detections representing the movement of one object.

$$Trajectory = \{d | d \in \mathit{Detections}\}$$

$$\forall d1, d2 \in Trajectory, \mathit{d1.timestamp} = \mathit{d2.timestamp} \implies d1 = d2$$

On each trajectory, we can define properties such as:

[1]Can be video time, but has to be in the same way for all detections.

- start_time - The start_time of the trajectory is the earliest timestamp of trajectory detections.
  $T.start\_time \Leftrightarrow \forall d \in T; start\_time <= d.timestamp$
- end_time - The end time of the trajectory is the latest timestamp of trajectory detection.
  $T.end\_time \Leftrightarrow \forall d \in T; end\_time >= d.timestamp$

and operations such as:

- detection is addable - The detection is to trajectory addable if in trajectory is no detection with the same timestamp.
  d is addable $T \Leftrightarrow \forall o \in T; d.timestamp \neq o.timestamp$
- trajectories are mergeable - Trajectories T1 and T2 are mergeable if detections of trajectories have no common timestamp.

$$T1\, is\; mergeable\; with\, T2$$
$$\Leftrightarrow$$
$$\forall d1 \in T1, \forall d2 \in T1; d1.timestamp \neq d2.timestamp$$

Definition 5. We define $DetectionsToTrajectories$ as a set operation that transforms a set of detections into multiple trajectories and sets of unused detections.

In our case, the DetectionsToTrajectories operation is implemented as an algorithm for clustering detections into Trajectories and is introduced in subsection 3.1.1.

Definition 6. We will define $TrajectoriesToTrajectories$ as a set operation that transforms a set of trajectories into a set where the produced trajectories are the same or in conjunction. In our case, the TrajectoriesToTrajectories operation is implemented as an algorithm for clustering trajectories into trajectories and is introduced in subsection 3.1.2.

2.2 Traged - pipeline

The pipeline aims to transform the detections into trajectories and use them to generate new data, such as trajectory description or find undetected detections, as shown in the picture 2.1. Since we use a configurable analytic model, a user-defined configuration file drives the whole pipeline. Thus, we will use plenty of variables to explain algorithms, where all variables and configurable properties are meant to be passed to algorithms through the configuration file.

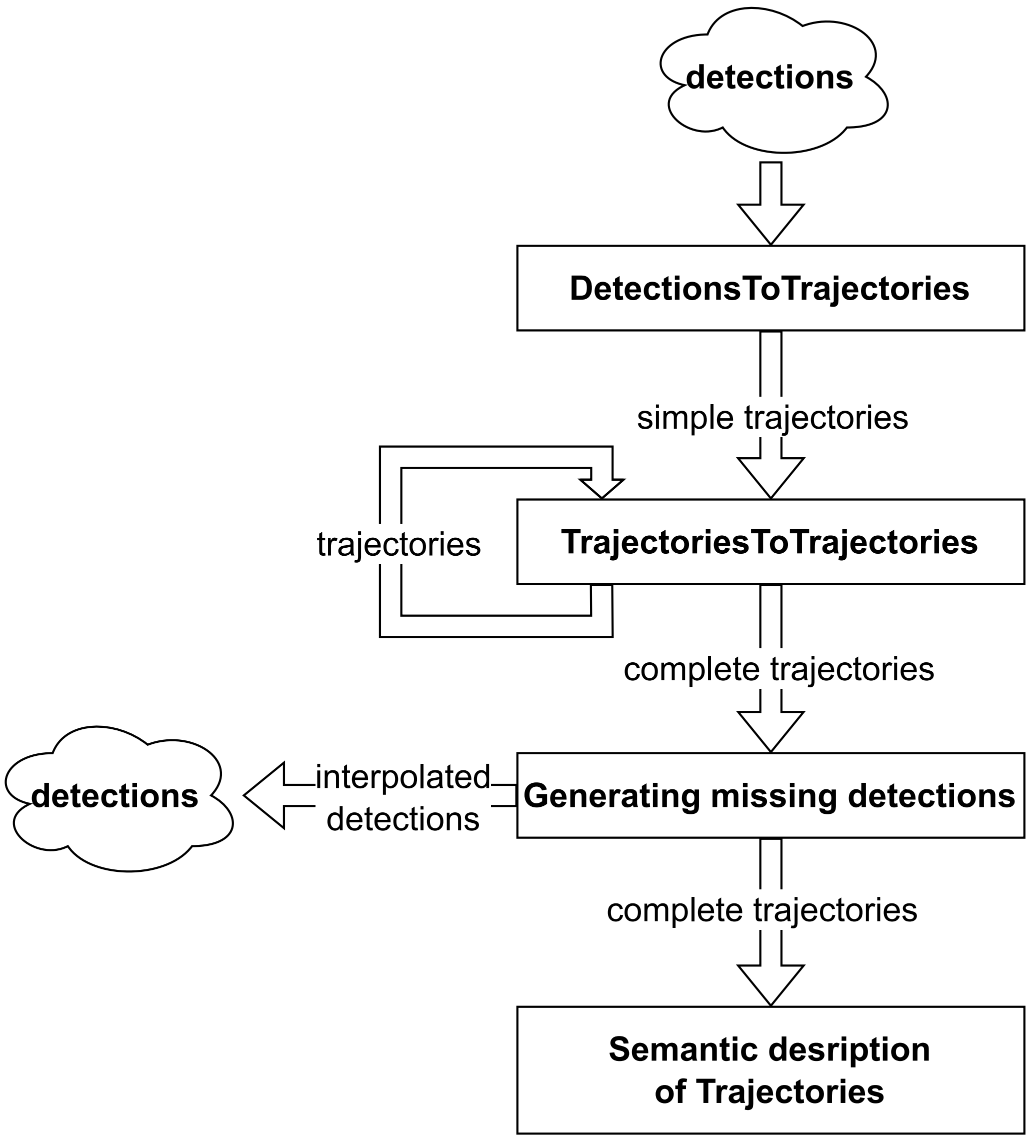


Figure 2.1: *Traged* - designed pipeline

2.2.1 Generating trajectories

Initially, we have a set of detections, which we want to cluster into trajectories based on the detection features. However, for detections, we can, for example, compare only their position or histograms. With the operation DetectionsToTrajectories, the algorithm clusters detections into trajectories. For trajectories, we can extract more features, such as the speed or direction of trajectories and use them for clustering with the TrajectoriesToTrajectories.

As shown in the picture 2.2, the second phase of clustering, which represents the operation TrajectoriesToTrajectories and clusters trajectories into longer trajectories, can be repeated multiple times to aggregate the full trajectories of the objects. Details on this clustering are described in chapter 3.

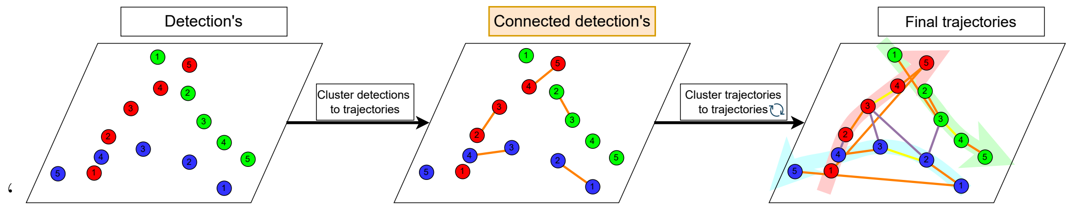

Detection's
Cluster detections to trajectories
Connected detection's
Cluster trajectories to trajectories
Final trajectories

Figure 2.2: *Traged* - clustering trajectories

Complexity reduction The problem with this approach is time complexity. In general, we could take all detections for one video and let described algorithm compare detections with each other and build the trajectories, but this would be highly impractical. Instead, we have decided to support only a restricted amount of detections and final conjunction run on generated trajectories, where we can easily reduce the number of comparisons. This approach also corresponds with the database-oriented source and is great for processing large streams. Detail descriptions with many optimizations can be found in chapter 3.

2.2.2 Usage of trajectories

Once we have generated trajectories, we will use them to improve existing data and create a semantic representation of trajectories for work.

Data improvement As mentioned, the detections we are using as input do not have to be ideal and usually are missing in some parts of the video. Thus, we would like to use generated trajectories and extract from the original video missing detections. This part of the pipeline is described in section 5.1.

Semantic representation The other use of trajectories is to create a semantic representation. This representation should describe on various low-level features of trajectories such as the type of movement - (standing, walking or running) or directional - (turning right, straight, turning left) and so on. These data are prepared for our colleges and will be used to describe activities and search for activities with a common pattern. More about representation generation can be found in section 5.2.

3. Clustering into trajectories

In this chapter, we will describe the designed algorithms for clustering and provide pseudo-codes. We will also explain all our core decisions during the algorithm design and present a variant we suggested to solve/minimize discovered problems. To present the algorithm properly, We will describe it in a bottom-up approach. First of all, in section 3.1, we will present algorithms designed to generate trajectories from a limited number of detections, subsection 3.1.1, and later, in subsection 3.1.2, algorithms merging trajectories into larger ones. Once we know sub-algorithms, how to create basic trajectories and merge them, in section 3.2, we will describe the algorithm connecting all sub-algorithms together and, from detections, generate complete trajectories in subsection 3.2.1. Finally, in subsection 3.2.2 will be described the main loop that processes detections for the whole video stored in the database into trajectories.

3.1 Clustering detections into trajectories

In this section, we will describe how the clustering itself is done. As mentioned, we can distinguish two cases: clustering detections into trajectories $\longrightarrow$ *DetectionsToTrajectories* and clustering trajectories into longer trajectories$\longrightarrow$ *TrajectoriesToTrajectories*. In the original proposal, we planned to create a universal algorithm. Still, as shown in this chapter, we can achieve better results and performance thanks to clustering detections and trajectories in different ways.

Classification Both variants use user-defined rules specifying which features of detections or trajectories should be compared and how to measure the connectivity based on the compared features. The techniques for extracting features and measuring connectivity are defined in chapter 4. For simplicity, in this chapter, we use the term *classifier*, which represents a block of code based on user-defined rules, taking two detections or trajectories as input and returning the *connectivity* $\in \langle 0, 1\rangle$ representing how well are input data connectable.

$$\forall d1, d2 \in detections; classifier.compare(d1, d2) \longrightarrow \langle 0, 1\rangle$$
$$\wedge classifier.compare(d1, d2) = classifier.compare(d2, d1)$$
$$\forall t1, t2 \in trajectories; classifier.compare(t1, t2) \longrightarrow \langle 0, 1\rangle$$
$$\wedge classifier.compare(t1, t2) = classifier.compare(t2, t1)$$

3.1.1 Transforming detections into trajectories

The main purpose of this algorithm, as shown in the picture 3.1, is to group detections into trajectories. These trajectories do not have to be long but have to be correct. As the input, the algorithm receives detections and user-defined rules used to construct *classifier*.

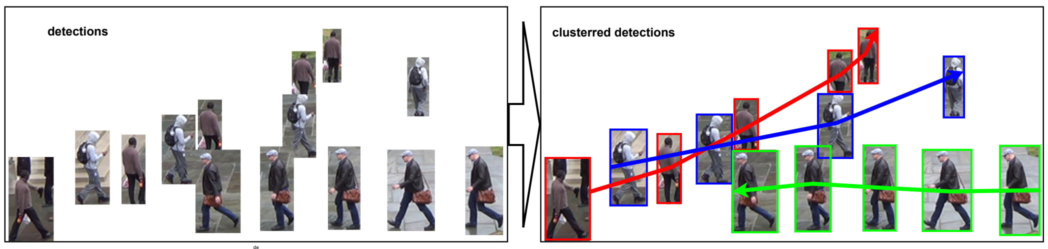


Figure 3.1: Clustering detections into trajectories example

The algorithm

The core of the algorithm is quite simple. First, the algorithm by $classifier$ compares all detection pairs, without repetitions, where detections belong to different frames $\longrightarrow$ *compared*. Then the algorithm sorts all comparisons by their connectivity in descending order $\longrightarrow$ *sorted* and starts building the trajectories. The algorithm from *sorted* take out the most connectable pair and processes it according to the state of the detections:

- Neither of the detections is in an existing trajectory.
  $\longrightarrow$ Create a new trajectory with these detections.
- Only one detection is in an existing trajectory.
  $\longrightarrow$ If the detection is addable to that trajectory, add it.
- Both detections are in existing trajectories.
  $\longrightarrow$ If trajectories are mergeable, merge them.

In other cases, we can forget this pair and continue with the next one.

In this way the algorithm processes pairs until all of them are processed[1]. For better visualisation, we can see the algorithm in pseudocode 1.

[1]We can use some of the optimization strategies, such as process pairs until each detection is in trajectory, define a threshold for connectivity or use only a limited number of top connectable pairs, such as $n * \#detections$.

Algorithm 1: DetectionsToTrajectories

```
Input: detections, classifier
Output: trajectories
$trajectories = \{\}$
$compared = []$
for d1,d2 in combinations(detections) do
    if $d1.timestamp \neq d1.timestamp$ then
        $connectability = classifier.compare(d1, d2)$
        $compared.add(connectability, d1, d2)$
$sorted$ = Sort in descending order compared by connectability.
while sorted is not empty do
    $detectionA, detectionB = sorted.pop()$
    if Neither detectinA or detectionB belong to trajectory then
        trajectory =Create new trajectory with detectionA and detectionB
        Add trajectory into all trajectories
    else
        if Both detectionA and detectionB belongs to trajectories then
            trajectoryA = From trajectories load trajectory with detectionA
            trajectoryB = From trajectories load trajectory with detectionB
            if trajectoryA and trajectoryB are mergeable then
                Remove trajectoryB from trajectories
                Add all detections from trajectoryB into trajectory trajectoryA
        else
            #Only one detection is in some trajectory
            detection = detection without trajectory
            trajectory = Get trajectory for detection with trajectory
            if detection is addable into trajectory then
                Add detection into trajectory
```

Complexity reduction

The problem with this algorithm is that the time complexity is growing quadratically because we are comparing all detections with each other, and the comparison of two detections is computationally complex. To avoid quadratic complexity, we propose two easy solutions that reduce the number of compared detections and one optimization based on the position of detections.

Sliding window

The first solution is to use a sliding window for generated comparisons. This means comparing each detection only with detections from N previous frames and N following frames, where N is a reasonably small number. This approach highly corresponds with the algorithm 1, and the only change is done while pairs comparison.

Fixed length blocks

Another solution is to split detections into blocks of fixed length, and on each block of detections, run an algorithm 1. By this approach, we generate a lot of short trajectories, which we can later merge with TrajectoriesToTrajectories, or we can use real-life knowledge to merge trajectories by common detection.

First, we organize detections into lists of DFrames. Since each detection has to be at most in one trajectory, we can split the list of DFrames into blocks with one[2] DFrame overlap, as shown in the pictures 3.2, process each block with the

[2]During tests, we have tried to use more than one DFrame as overlap, but the results did not improve, only time complexity gets bigger.

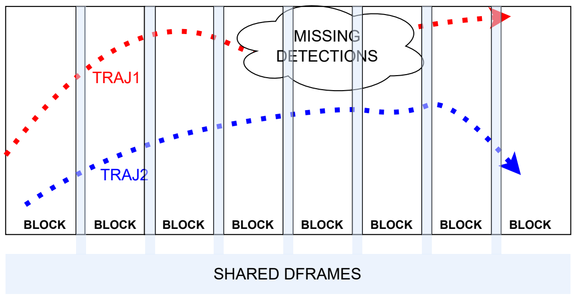
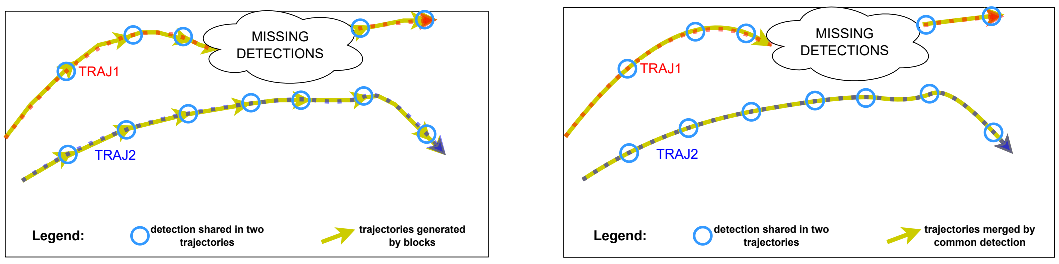


Figure 3.2: Detections splitted into blocks of DFrames with overlap


MISSING DETECTIONS
TRAJ1
TRAJ2
Legend:
detection shared in two trajectories
trajectories generated by blocks
trajectories merged by common detection

(a) generated trajectories by block

(b) merged by common detection

Figure 3.3: Trajectories processed and merged by common detection

algorithm 1 and after those merge trajectories with common detections, as shown in the pictures 3.3. For illustration, this approach is written in pseudocode 2.

Algorithm 2: Processing of detections blocks

```
Input: detections,classifier
Output: trajectories
$DFrames = \{\}$
for $detection$ in $detections$ do
    $DFrames[detection.timestamp].add(detection)$
$blocks\_with\_overlap$ = Split $DFrames$ into blocks of defined length with one DFrame overlap.
for $block$ in $blocks\_with\_overlap$ do
    $block\_detections$ = detections from $block$
    $trajectories+ = DetectionsToTrajectories(block\_detections, classifier)$
#Merge trajectories with a common detection.
for $detection$ in $detections$ do
    $sharers$ = Get trajectories with $detection$
    if $size(sharers) > 1$ then
        Merge trajectories from $sharers$ in one.
```

Position optimization

To optimize the number of comparisons even more, we can split the video image into a grid, sort all detections into cells, and compare only detections with detections from the nearest cells $\longrightarrow$ *surrounding*, as shown in a picture 3.4. The disadvantage of this approach is that the number of grid cells has to be specified in correlation with the *sliding window* or *fixed length blocks* technique. With a large interval - large block of DFrames, we have to enlarge the size of grid cells. Otherwise, there is a possibility that the object detections are out of the surrounding.

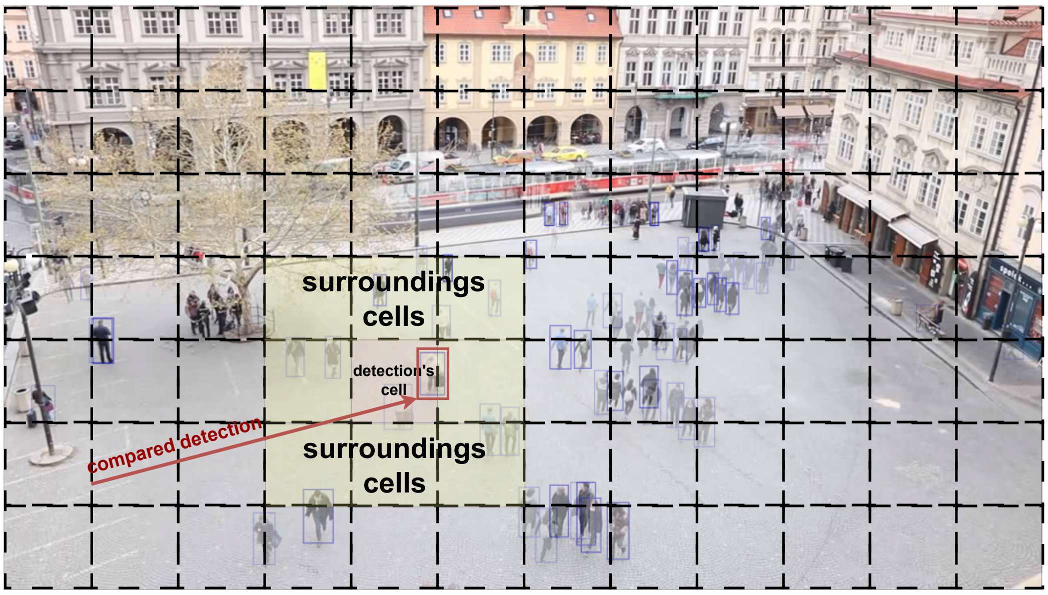


Figure 3.4: Example of detection comparison with detections in grid

Observation and used methods

During the implementation and testing of the proposed algorithms, we have implemented all of these optimizations. We achieved the best results with a combination of *fixed length blocks* and *position optimization*. More about our implementation can be found in chapter 6.

Unmatched detections

According to configuration, in some cases, the algorithm ends up with detections that are not connectable enough and do not end up in any trajectory. So a question arises what shall we do with the leftover detections? The best solution is to do nothing, and detections do not insert in any trajectories by any strategy at the current moment. Later in section 5.1, we provide a method to assign the unmatched detections into trajectories with a different approach. If there is a need to insert all detections to trajectories at any cost, the best way is not to specify a threshold for connections in configuration. In that case, usually, all detections are in trajectories and in the worst-case scenario, the leftover detections form trajectories together.

3.1.2 Transforming trajectories into trajectories

The TrajectoriesToTrajectories algorithm is designed to merge trajectories and, in some aspects, is quite similar to the DetectionsToTrajectories algorithm. As the input, the algorithm receives trajectories and user-defined rules used to construct $classifier$.

The algorithm

First, the algorithm by $classifier$ compares all trajectories with each other and filters out trajectory pairs that are not connectable by provided configuration

$\longrightarrow$ *compared*. To reduce the number of comparisons, we can also directly skip pairs where trajectories are not mergeable because of having detections at least in the same frame. Then the algorithm sorts all comparisons by their connectivity in descending order $\longrightarrow$ *sorted* and starts building the trajectories. The algorithm takes out the most connectable pair and processes it according to the state of the trajectories:

- Trajectories are mergeable.
  $\longrightarrow$ Merge trajectories in one.

- Trajectories are not mergeable.
  $\longrightarrow$ Forget this pair and continue with the next one.

Once the algorithm merges two trajectories, it has to take it into account, and once one of those merged trajectories appears in the pair again, it has to use the merged one instead. In this way algorithm processes pairs until all of them are processed[3].

Algorithm 3: TrajectoriesToTrajectories

```
Input: trajectories,classifier
Output: trajectories
merged = {}
compared = []
for t1, t2 in combinations(trajectories) do
    if t1 is mergeable with t2 then
        connectability = classifier.compare(t1, t2)
        compared.add(connectability, t1, t2)
sorted = Sort descending compared by connectability.
while sorted is not empty do
    trajectoryA, trajectoryB = sorted.pop()
    if trajectoryA is mergeable with trajectoryB then
        trajectoryNew = merge trajectoryA with trajectoryB
        merged.removeIfExist(trajectoryA)
        merged.removeIfExist(trajectoryB)
        merged.add(trajectoryNew)
        #Since now, in pairs, where appear trajectoryA or trajectoryB
        # algorithm use trajectoryNew
trajectories = merged
```

Complexity reduction

The algorithm has the problem of time complexity growing quadratically, mainly because of the number of trajectory comparisons. To reduce quadratic complexity, we propose a solution that reduces the number of compared detections and one optimization based on the position and location in time of trajectories.

Blocks of fixed length

We can use a similar approach for trajectories as for detections, splitting all the trajectories into blocks of restricted length $\longrightarrow$ *block_size*, but we have to do it with the account to time location. To illustrate the problem, we have prepared a few input trajectories, as we see in the picture 3.5.

[3] In this case, we did not use any technique to reduce the number of processed pairs, and we focused on complexity reduction in subsection 3.1.2

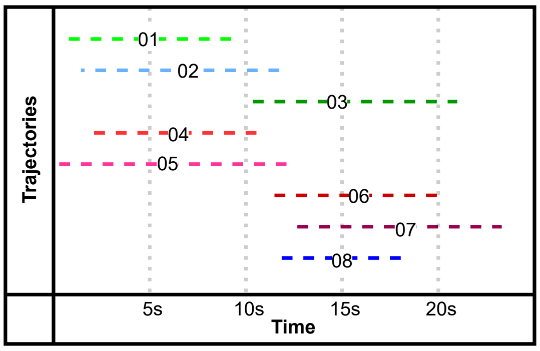
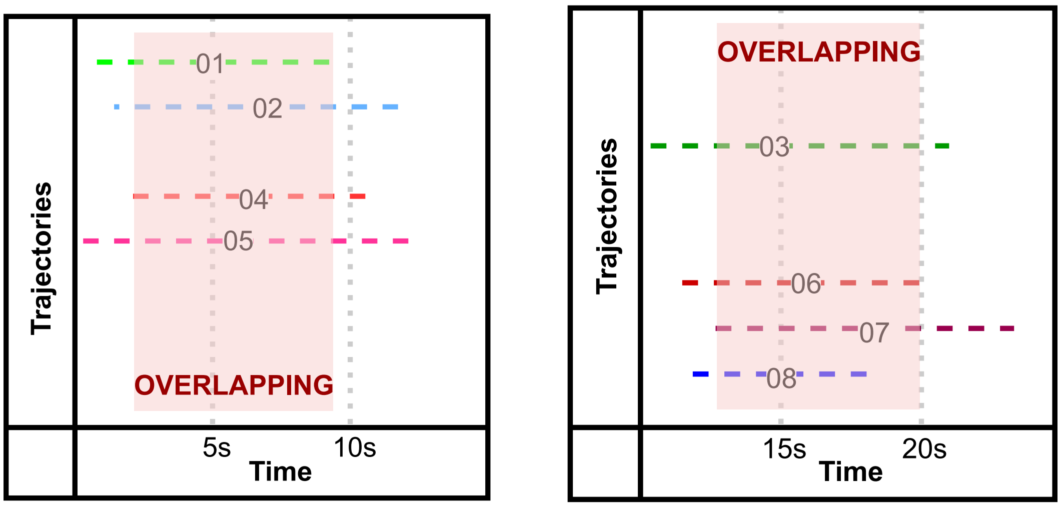


Figure 3.5: Example input trajectories

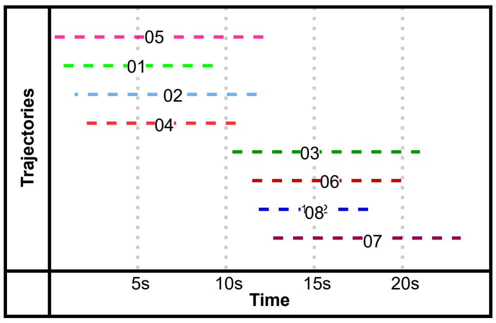
Now we divide trajectories into two blocks by their *start_time*, as seen in the image 3.6. This is a failure because for multiple continuous trajectories, blocks contain trajectories with detections in the same frames. Then the algorithm cannot merge any trajectories because they all have detections in the same frames $\longrightarrow$ *overlapping*.



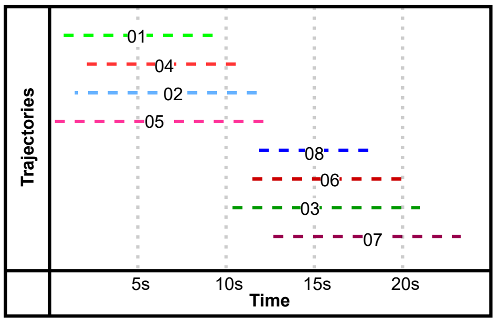
(a) first block (b) second block

Figure 3.6: Trajectories divided into blocks by trajectories time

To avoid this failure, where all trajectories in one block have detections in one frame and are unmergeable, we have designed the following procedure. The algorithm creates two sorted lists, where one is sorted by the trajectory's *start_time* $\longrightarrow$ *start_list* and the second list is sorted by its *end_time* $\longrightarrow$ *end_list*, as shown in picture 3.7.



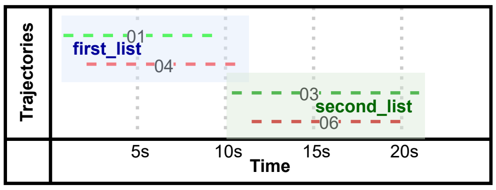
(a) sorted by start_time



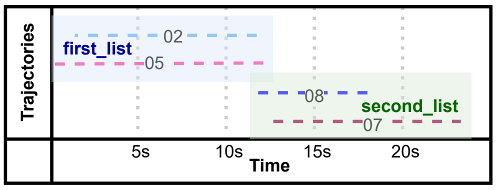
(b) sorted by end_time

Figure 3.7: Trajectories divided into blocks by trajectories starts

After that, we can create half $\longrightarrow$ $first\_list$ of the block by taking the first $\frac{block\_size}{2}$ trajectories out from end_list and remove these trajectories from $start\_list$. To create a second half$\longrightarrow$ $second\_list$ of the block, we will take the lowest end_time of trajectories from $first\_list$ and from $start\_list$ take out $\frac{block\_size}{2}$ the nearest later trajectories. The final block is created by the union of $first\_list$ and $second\_list$.



(a) first block



(b) second block

Figure 3.8: Blocks created by new method

Extension In some cases, the end_time of the latest trajectory from $first\_list$ is higher than the latest start of $second\_list$. These trajectories can be directly removed because they cannot be merged and can be inserted back into $start\_list$ and end_list.

Algorithm 4: Trajectories blocks

```
Input: trajectories, block_size,classifier
Output: new_trajectories
$new\_trajectories = []$
$blocks = []$
$start\_list$ = Sort trajectories by their $start\_time$.
$end\_list$ = Sort trajectories by their $end\_time$.
while $start\_list$ is not empty do
    $first\_list$ = From $end\_list$ take out top $\frac{block\_size}{2}$ $trajectories$
    $start\_list = start\_list - first\_list$
    $lowest\_endtime$ = From $first\_list$ get the lowest $end\_time$ of all trajectories.
    $second\_list$ = From $start\_list$ get top $\frac{block\_size}{2}$ trajectories, where
      $start\_time >= lowest\_endtime$.
    $end\_list = end\_list - second\_list$
    if use Extension then
        $latest\_starttime$ = From $second\_list$ get the latest start time of all
        $unmergable$ = From $first\_list$ remove trajectories, where $end\_time > latest\_starttime$
        $end\_list = end\_list + unmergable$
        $start\_list = start\_list + unmergable$
    $new\_trajectories+ = TrajectoriesToTrajectories([start\_list + second\_list], classifier)$
```

Position optimization

To optimize the number of comparisons, we can split all trajectories into a grid and compare only trajectories with trajectories from the nearest cells, as we did for the detections. However, the trajectories are a bit more complicated because they usually start at one cell and end at another. Thus the alghorithm is altered.

The algorithm splits a frame into a grid and, for each cell, creates two lists of trajectories.

- *start_list* - List of all trajectories starting in this cell.
- *end_list* - List of all trajectories ending in this cell.

Once we have all trajectories prepared in the grid's lists, we can finally generate comparisons. For each trajectory $\longrightarrow$ *compared*, the algorithm selects a **starting cell** and takes trajectories ending there and in the nearest cells, as well as ending before *start_time* of *compared*. For selected trajectories then generates a comparison with *compared*.

The detail, that the algorithm is finding trajectories endings in the surrounding is crucial, because for the algorithm is more efficient to select from already generated and ended trajectories $\longrightarrow$ searching in the history than in the future. In the opposite direction, the correct trajectory, can be still not generated.

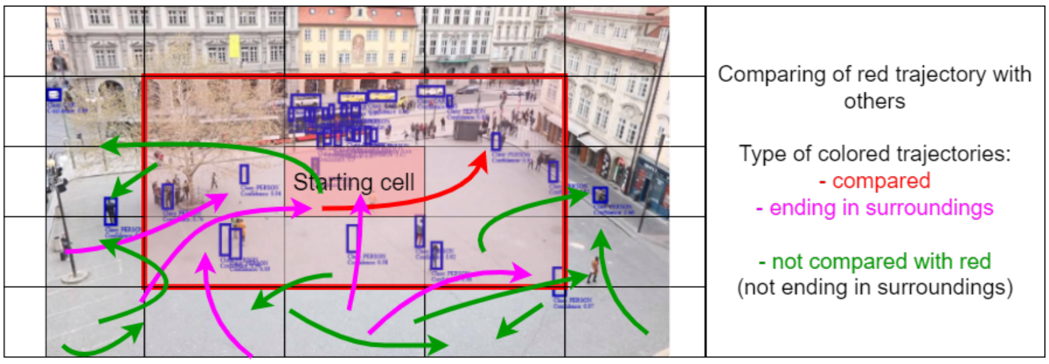

Starting cell
Comparing of red trajectory with others
Type of colored trajectories:
- compared
- ending in surroundings
- not compared with red
(not ending in surroundings)

Figure 3.9: Grid comparison for one trajectory

Observation and used methods

During the implementation and testing of the proposed algorithms, we implemented all of these optimizations, but in the end, we are using only the *position optimization*. More about our implementation and code can be found in chapter 6.

3.2 The algorithm flow

In this section, we present the architecture of the whole clustering flow. The processing of all detections from video into complete trajectories.

3.2.1 Clustering tree

When we have already defined basic algorithms, it is time to put them together and create a flow that generates complete trajectories. To generate complete trajectories, using algorithms defined in section 3.1 exist a lot of possible approaches. Each approach has different advantages and disadvantages, and we want to cover as many approaches as possible. Thus, we first designed a few possible approaches and, based on them, designed an algorithm capable of covering them all.

Possible approaches

The approaches are based on the quality of detections and the quality of the *classifiers*. To illustrate at least some of them, we came up with the following approaches:

Gradual clustering Gradual clustering is the approach where small trajectories are created in short intervals, and the interval is doubled in each round of clustering. The advantage of this approach is that compared trajectories/detections are close in time and space. The disadvantage of this approach is a need for multiple clustering layers, as shown in the picture 3.10, which worsens the time complexity.

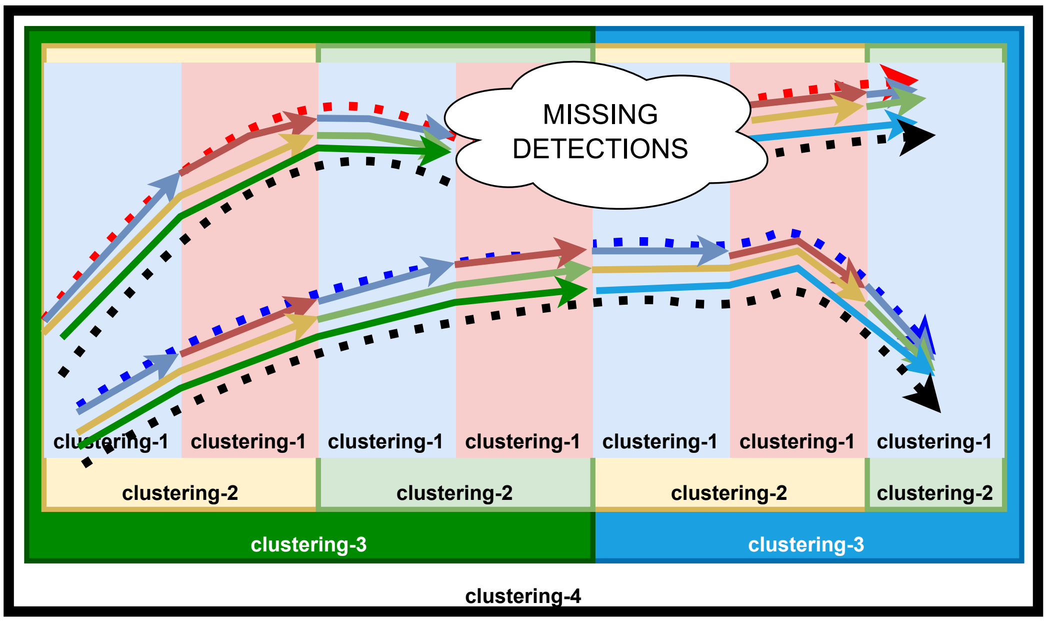


Figure 3.10: All possible sub-trajectories created by *Gradual clustering*

Bearing strategy The second approach is based on bearing trajectory, where the algorithm creates trajectories across the large part of the video and small trajectories locally. Later the algorithm just merges the large trajectories with the local ones. The expected advantage of this approach is the reduction of used layers, and the disadvantage is the need for accurate *classifiers*, working precisely on detections in a large interval, as we can see in the illustration picture 3.11.

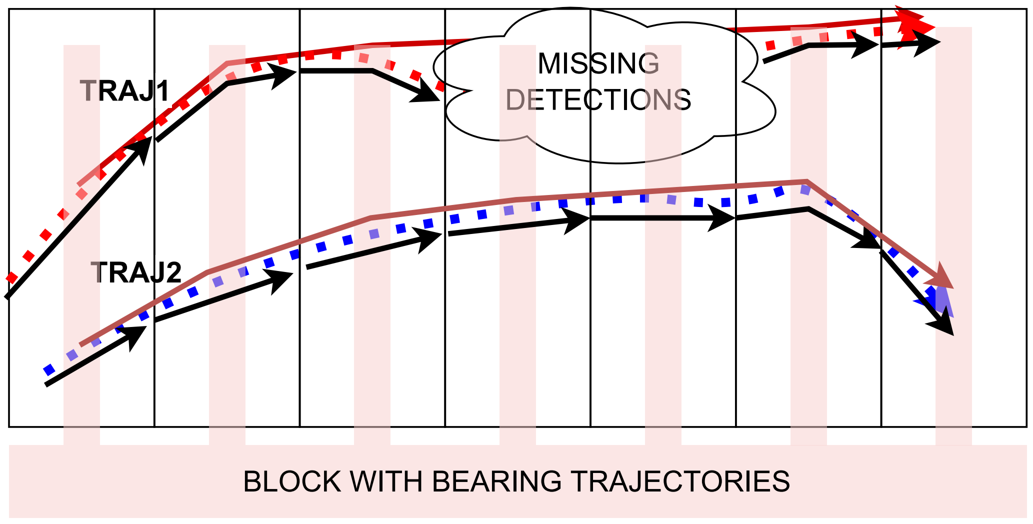


Figure 3.11: Sub-trajectories created by *Bearing strategy*

Tree structure

To support both approaches and even more, we are defining clustering as a tree structure. The tree structure is user-defined in the configuration, and each node contains parameters for evaluation(for $classifier$). In our structure, are only two types of nodes:

- $DetectionsToTrajectorie \longrightarrow leaves$
  $\longrightarrow$ input detections
  $\longrightarrow$ output trajectories

- $TrajectoriesToTrajectories$—
  $\longrightarrow$ input trajectories
  $\longrightarrow$ output trajectories

The processing of the tree is done by evaluation from root, represented as a function $TreeEvaluation(detections, tree)$, as shown in the pseudocode 5.

First, each node evaluated all children and generated trajectories inserted into *children_trajectories*. In the end, evaluates itself, whereas input uses *children_trajectories*. The leaf directly evaluates itself with provided detections as input.

Algorithm 5: TreeEvaluation(detections,tree)

```
Input: detections, tree
Output: trajectories
if $tree$ is $Leaf$ then
    $classifier = tree.classifier$
    $trajectories$ = With $classifier$ run $DetectionsToTrajectorie$ over $detections$
else
    $children\_trajectories = []$
    for $child$ in $tree.children$ do
        $children\_trajectories += TreeEvaluation(detections, child)$
    $classifier = tree.classifier$
    $trajectories$ = With $classifier$ run $TrajectorieToTrajectorie$ over $children\_trajectories$
return $trajectores$
```

The node implementation In our implementation, we have implemented a few cases of nodes. The key ones, as described in this chapter and the developer-oriented ones.

- Key nodes:
  - DetectionsToTrajectorie as defined in subsection 3.1.2
  - TrajectoriesToTrajectories as defined in subsection 3.1.1
  - TrajectoryGetter - node responsible for merging of trajectories, with shared detections as mentioned in subsection 3.1.1.
- The developer nodes:
  - TrajectoryLoader - Loading already generated trajectories from the database, to avoid re-running of the whole sub-tree.
  - TrajectoryTransformer - The place where can developers directly test a new methods and develop new ways how to cluster detections and trajectories.

3.2.2 The database stream

In the end, we have to consider variants, where the detections are loaded from the database, and final trajectories are stored in the database as well. This is usually used in cases of large videos or streams where we cannot have all detections for a single video in local memory. Thus, we have to load only a restricted amount of consecutive DFrames and process them into partial trajectories.

Load

The algorithm loads the first *block_size* of DFrames from the database. Where the *block_size* is a user-defined value defining the maximal number of DFrames that can be loaded from the database at once. Once DFrames are loaded, the program can transform their detections with $TreeEvaluation(detections, tree)$, where the tree is loaded from configuration, into trajectories. These generated trajectories are not finished yet, because they overlap with the following DFrames. Thus, program stores these trajectories in local memory $\longrightarrow$ *dose*, marks them with the order number and repeats the whole process.

Merge and store

Once there are at least two[4] *dose* of trajectories in local memory, generated by previous step, the program can merge those two *dose* trajectories with the algorithm TrajectoriesToTrajectories. After this step, we can distinguish two types of newly generated trajectories:

- overlapping trajectories - Trajectories that are partially constructed from trajectories from the latest *dose*.

[4]In fact, the algorithm can be easily altered to wait and process more than two generations.

- complete trajectories - Trajectories not constructed from the latest *dose*.

After this, the algorithm can store all the complete trajectories in a database, and the overlapping trajectories keep in the latest *dose*.

Algorithm 6: processing a video stream

```
Input: conf
$latestDose = []$
while exist following block of DFrames do
    $latestDFrames$ = Load DFrames from the database.
    $detections$ =Get detections from $latestDFrames$
    $dose = TreeEvaluation(detections, conf.tree)$
    if latestDose is not empty then
        $allTrajectories = latestDose + dose$
        $newTrajectories = TrajectoriesToTrajectories(allTrajectories, conf.classifier)$
        $overlapping$ = trajectories from $newTrajectories$ at least partially from $dose$
        $complete = newTrajectories \setminus overlapping$
        Store $complete$ into the database.
        $latestDose = overlapping$
    else
        $latestDose = dose$
Store $latestDose$ into the database.
```

4. Features extraction and connectivity

In this chapter, we will focus on how to evaluate if two detections or trajectories belong to the same object. To achieve that, we will, for both detections and trajectories, define procedures of feature extraction, methods to measure the feature-connectivity based on extracted features, and methods to measure the connectivity based on the feature-connectivity of multiple features, as can be seen in the illustration picture 4.1. The output of this chapter is used for sorting pairs of detections or trajectories and can be seen in the previous chapter in section 3.1.

First of all, in section 4.1, we will define basic terminology, introduce methods for extracting features of detections and trajectories, and determine if they belong to the same object. For simplicity, the unit[1] responsible for extracting and comparing single features, we will call *comparator*.

In section 4.2, we describe the algorithm to evaluate all features and introduce methods for reducing the vector of features-connectivity generated by multiple *comparators* into a single connectivity value. We will call the unit responsible for this $classifier$.

4.1 Features extraction & feature-connectivity

The purpose of measuring the detections and trajectories feature-connectivity is to define if or how well two detections or two trajectories are addable/mergeable, ergo, if they are two detections or two trajectories of the same object. Be aware that the feature-connectivity is not the same as similarity, even though, in some cases, they are measured similarly. This section will introduce features we identified as usable for this purpose and used methods.

4.1.1 Basic terminology and expectations

First, we will define the terminology used in this chapter and describe the generic procedure of extracting features.

Definition 7. $Detectionfeature$ represents the detection's feature/property, such as the object's size, histogram or position in the video. Feature extraction for detection is a projection from a single detection into a vector of real values. $df(d) \longrightarrow \mathbb{R}^n; d \in detections$

Definition 8. $Trajectoryfeature$ represents trajectory feature/property, such as the direction or speed of an object. Feature extraction for a trajectory is a projection from a single trajectory (set of detections) into a vector of real values. $tf(t) \longrightarrow \mathbb{R}^n; t \in trajectories$

Definition 9. $feature$-*connectivity*, or shortly f*connectivity*, is a single value representing the possibility that two detections or trajectories represent the same

[1]In our code, is this unit represented by a   class.

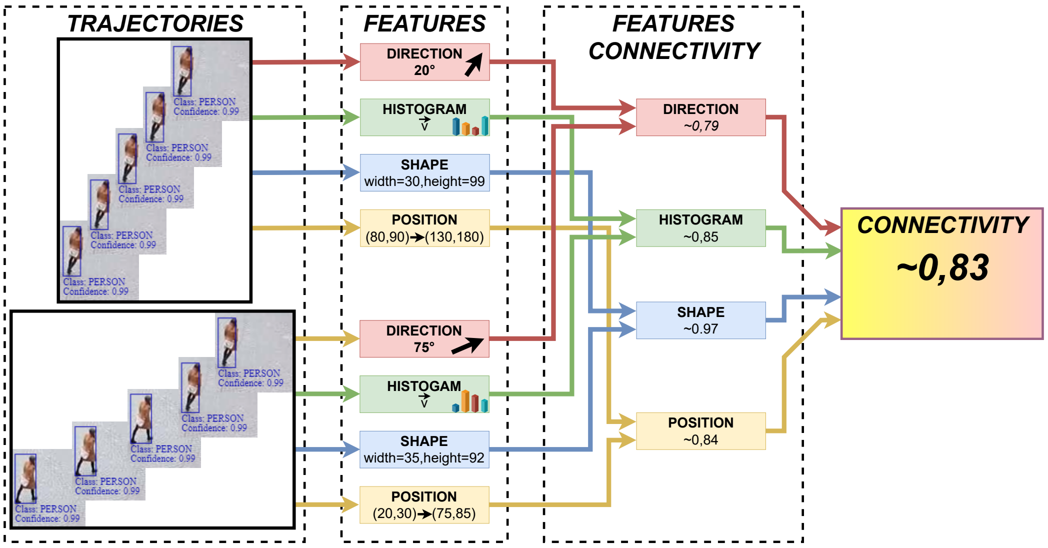


Figure 4.1: Illustrated flow of measuring pair connectivity

object based on one feature. The evaluation of feature-connectivity is then transformation of two detection/trajectory features into a single value in the range $\langle 0, 1\rangle$. For detections:
$dfc(df(d1), df(d2)) \longrightarrow \langle 0, 1\rangle; d1, d2 \in detections$
For trajectories:
$tfc(tf(t1), tf(t2)) \longrightarrow \langle 0, 1\rangle; t1, t2 \in trajectories$
Where the boundaries identify:

- $fconnectivity = 1$ - Highly probable[2] that, based on feature, belongs to the same object.
- $fconnectivity = 0$ - detections/trajectories do not belong to the same object.

Mapping When transforming features into feature-connectivity, we often need to map the feature-connectivity value from one range to another. To describe it, we will use the notation

$$fconnectivity = map(original\_range, new\_range, value)$$

where $value$ is from $original\_range$, that we want to transform into new_range.

Structure

The structure of the following subsections describes a single *comparator*, which means identifying features, determining how to extract features from data and using the extracted feature to measure feature-connectivity. From a programmer's point of view, we can look at the following sections as on a function, accepting pair of detections or trajectories and returning a feature-connectivity value.

To help with orientation, we can divide *comparators* on multiple levels:

[2]Even though the feature-connectivity is 1, it does not mean that the detections/trajectories belong to the same object. For example, by the histogram, there can exist detections of identical cars, which doesn't mean, there are not two of them.

Input Not all the features can be measured on detections. Thus some of the features are defined and compared only for trajectories, but all the compared features for detections can also be used for trajectories. A typical example is speed or direction.

Output feature-connectivity The feature-connectivity can be divided into a few groups by returned values.

- Binary decision - objects are connectable or not, nothing between.
- The list of fixed values - detections or trajectories can be only in limited states.
- The whole range $\langle 0, 1 \rangle$

Parameters

During the evaluation of the feature-connectivity, the algorithm uses user-defined parameters provided in the configuration. The parameters usually contain the following values depending on the context, such as:

- *tolerance* - The difference between compared features that can be ignored
- $max\_difference$ - The maximal difference between compared features to be connectable (return value > 0)
- *min_connectivity* - The minimal value of feature-connectivity that can be returned[3]
- *treshold* - The minimally acceptable feature-connectivity. Once feature-connectivity is generated below, the *comparator* returns 0

But in general, we are not restricting parameters for *comparators*, and the user creating the configuration can provide whatever they need.

4.1.2 Shape

We can identify the shape for pairs of detections, especially height and width. In our scope, the width and height are included in the information for each detection. The idea of this *comparator* is that the objects are not changing their real size during the movement, even though the object is changing its detected size, for example, because of moving toward the camera.

Width and height for trajectories

The *comparator* uses the direct detected width and height for detections, whereas for trajectories, we have to decide how to obtain width and height. We focused on two possibilities:

- Compare the nearest detections of trajectories in time.

[3]In some cases, where we are not sure about used feature-connectivity, it is handy to be able to set a non-zero value as a minimal feature-connectivity to avoid a pair exclusion.

- Compare the nearest detections of trajectories in space.

Where each method has a different advantage.

Time distance Getting the nearest detections in time can be easily achieved because there can be only two cases. In cases where trajectories do not overlap the *comparator* can use the end of the earlier trajectory and the start of the later trajectory. In cases where trajectories overlap the *comparator* can use two detections from overlapped detections.

Space distance For space, the *comparator* has to find the nearest tuple of detections, which can be time-consuming. As an advantage, the comparison is more precise because the objects are in the nearest possible position relative to the camera.

Average dimensions Average width and height usage is a bad idea. Imagine a scenario where the object is walking from the camera. Thus, in the beginning, the object detection is n times bigger than at the end of the trajectory. The average shape of the trajectory would be the same as the centre of the trajectory.

Width and height into feature-connectivity

Our approach to comparing the shapes is once the *comparator* has two dimensions to compare, the *comparator* will compare dimensions and multiply the ratios. $compared\_width = min(width\_1, width\_2)/max(width\_1, width\_2) \in \langle 0, 1\rangle$ $compared\_heights = min(height\_1, height\_2)/max(height\_1, height\_2) \in \langle 0, 1\rangle$ $compared\_ration = compared\_width * compared\_heights \in \langle 0, 1\rangle$

The *compared_ration* we can directly use as feature-connectivity, or we can define a minimal $minimal\_ration \longrightarrow$ the minimal possible value to use and then $fconnectivity = map([minimal\_ration, 1], \langle 0, 1\rangle, compared\_ration)$.

4.1.3 Direction

For trajectories, we can determine the direction of the tracked object, compare the directions of two trajectories, and check if they are aiming in the same direction.

Direction measurement

We must remember that the object of trajectory is freely moving and changing direction. Thus, we cannot simply measure the trajectory as a whole. Our approach determines for each trajectory multiple directions. Mainly measures the direction of a whole trajectory $\longrightarrow$ *main_direction* and of the starting$\longrightarrow$ *start_direction* and ending $\longrightarrow$ *end_direction* detections as we can see in the picture 4.2 and as proposed in the pseudocode 7.

Algorithm 7: directions of trajectory

```
Input: trajectory,subtraj_size
Output: main_vector,start_vector,end_vector
$sorted\_detections$ =sort detections from trajectory by timestamp
$first\_detection = sorted\_detections[0]]$
$last\_detection = sorted\_detections[-1]]$
$main\_vector = [$
$last\_detection.center\_x - first\_detection.center\_x,$
$last\_detection.center\_y - first\_detection.center\_y$
$]$
if $sorted\_detections >= subtraj\_size$ then
    $start\_detection = sorted\_detections[subtraj\_size]]$
    $end\_detection = sorted\_detections[-subtraj\_size]]$
    $start\_vector = [$
    $start\_detection.center\_x - first\_detection.center\_x,$
    $start\_detection.center\_y - first\_detection.center\_y$
    $]$
    $end\_vector = [$
    $last\_detection.center\_x - end\_detection.center\_x,$
    $last\_detection.center\_y - end\_detection.center\_y$
    $]$
else
    $start\_vector = main\_vector$
    $end\_vector = main\_vector$
return $main\_vector, start\_vector, end\_vector$
```

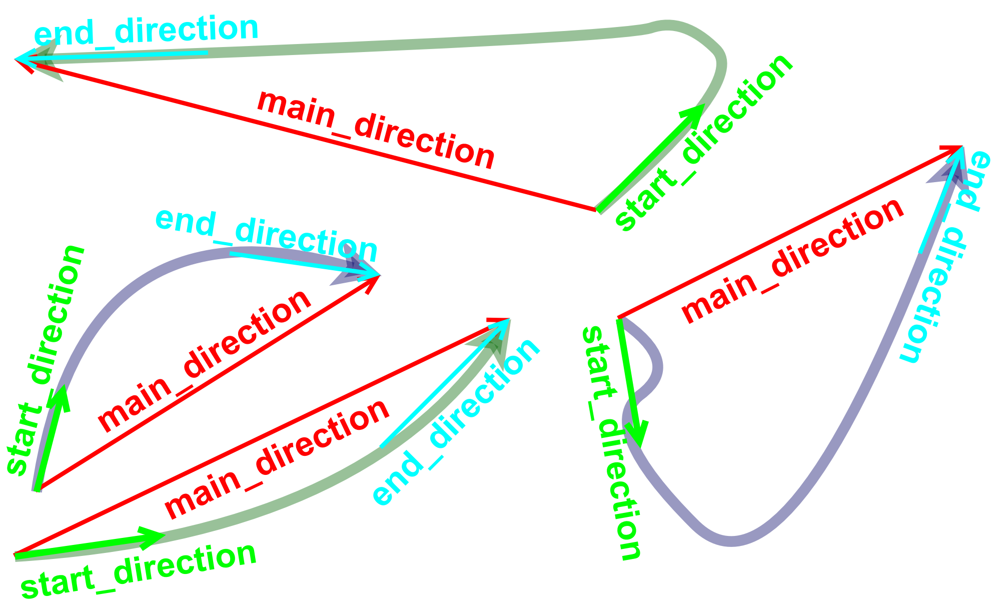


Figure 4.2: Example of trajectories with directions

Directions to feature-connectivity

Once we have identified directions for trajectories, we can compare two trajectories. Where can we use to compare from early starting trajectory $[main\_direction, end\_direction]$ with $[main\_direction, start\_direction]$ for later trajectory. As a result, we take the smallest difference.

To transform the difference into a feature-connectivity, algorithm use the user-defined boundaries:

- $difference \in [0, tolerance] \longrightarrow fconnectivity = 1$

- $difference \in [tolerance, max\_difference] \longrightarrow fconnectivity = map([tolerance, max\_difference], [1, min\_connectivity], difference)$.
- $difference \in [max\_difference, inf] \longrightarrow fconnectivity = min\_connectivity$

In this *comparator*, we used $min\_connectivity$ value to avoid assigning 0 because, in observations, we noticed that some object trajectories movement were meaningless, for example, a man waiting on a tram walking in a spiral. Still, in general, this *comparator* is quite handy for cases where multiple trajectories are crossing over.

4.1.4 Category

Some of the detectors provide for each detection classification into categories such as $[PERSON, CAR, TRUCK, TRAM, BICYCLE, ...]$. These categories can be used to measure feature-connectivity as well.

Category extraction

This *comparator* is designed for datasets with a category for each detection. Thus, for detection, we know the category and for trajectories, we use the most occurred category in trajectory.

Categories into feature-connectivity

The *comparator* extracts a category for both detections/trajectories $\longrightarrow category\_A, category\_B$. Then the approach to generate feature-connectivity differs in detector-provided categories.

Simple categories In cases where the detector classifies detections only on basic categories with no risk of category mismatch such as $[PERSON, VEHICLE, TRAIN, ...]$, the feature-connectivity is split into two cases:

- $category\_A \neq category\_B \longrightarrow fconnectivity = 0$
- $category\_A = category\_B \longrightarrow fconnectivity = 1$

Structured categories In cases where the detector classifies detections into somehow structured categories, where is a risk of category mismatch, such as $[PERSON, CAR, TRUCK, TRACTOR, TRAIN, TRAM, ...]$,
the feature-connectivity has to be adapted to a classification error. For example, on tested data, it was common for small trucks to receive a category CAR or $TRUCK$. Thus, we identified a set of classes that are often mistaken $\longrightarrow mistaken\_categories$ and tuples from $mistaken\_categories$ has feature-connectivity lowered by $penalization\ in (0, 1\rangle$.

- $category\_A = category\_B \longrightarrow fconnectivity = 0$
- $category\_A, category\_B \in mistaken \longrightarrow fconnectivity = 1 - penalization$
- $category\_A \neq category\_B \longrightarrow fconnectivity = 0$

4.1.5 Timestamps

We can define a binary *comparator* for trajectories that verifies that trajectories are *mergeable* $\longrightarrow$ have no detection with the same timestamps.

Timestamps into feature-connectivity

Each detection has a timestamp. Thus, to obtain timestamps of trajectory, the *comparator* simply extracts a set of its timestamps. To evaluate feature-connectivity, the *comparator* then, by definition 4, verifies that trajectories have an empty intersection.

- trajectories have an empty intersection $\longrightarrow fconnectivity = 1$
- *otherwise* $\longrightarrow fconnectivity = 0$

4.1.6 Speed

For trajectories, we can measure the speed of the tracked object and later compare the speed of two trajectories and check if the speed of both objects was similar or, as described below, use the measured speed for distance comparison.

Speed measurement

There are multiple ways how to measure the speed of trajectory, but we have to avoid distortion caused by object randomness. First of all, we have to keep in mind that the object of trajectory is freely moving. Thus, we cannot measure the trajectory as a whole, but, to eliminate the distortion, we have to divide the trajectory into parts. On the other hand, we do not need the speed to be absolutely precise. Thus we can divide a trajectory into blocks and measure the speed by blocks. The measuring:

For each block of detections, the *comparator* chooses the first $\longrightarrow first\_d$ and the last$\longrightarrow last_d$ detection and, using them, computes the speed of the block.
$time\_difference = last\_d.timestamp - first\_d.timestamp$
To measure the distance compactor, use the *Euclidean distance.*
$distance\_pixels = \sqrt{(last\_d.x - first\_d.x)^2 + (last\_d.y - first\_d.y)^2}$
$speed\_pxs = distance\_pixels/time\_difference$

The disadvantage of this approach is that the object's speed depends on the position relative to the camera. Thus, instead of distance in pixels, we have used distance relative to the object size, where distance in pixels is divided by a bigger dimension of the average dimension of the block.

The average width of measured block $\longrightarrow avg\_width$ and average height of measured block $\longrightarrow avg\_height$.
$distance\_object\_size = distance\_pixels/max(avg\_width, avg\_height))$
$speed\_object\_size = distance\_object\_size/time\_difference$

Speed to feature-connectivity

Once the *comparator* measures the speed of each block, there is a need to choose the type of speed to compare, such as an average speed, median speed, maximal speed and so on. Our *comparator* selects the fastest one because, from observations, most of the objects during their movement were moving at a constant speed, their maximal speed that reaches or slows down to stand.

The feature-connectivity is determined as the ratio of the fastest speeds of two trajectories $\longrightarrow fastest\_speed\_A, fastest\_speed\_B$
$compared\_ration = \frac{min(fastest\_speed\_A, fastest\_speed\_B)}{max(fastest\_speed\_A, fastest\_speed\_B)}$
The $compared\_ration$ then *comparator* directly use as $fconnectivity$, or use value $map([min\_speed\_ratio, 1], [min\_connectivity, 1], compared\_ration)$.

Speed observation

During the testing of this $comparators$, we observed that this *comparator* is useful for moving objects and surrounding where objects are moving most of the time but quite useless for not moving objects. Thus, we are not using this $comparator$. However, based on this *comparator*, we based the following approach in subsection 4.1.7 using a distance of two objects.

4.1.7 Position

Each detection has a determined space position in a video, and we can use it to measure the distance.

Distance

For every two positions, we can measure the distance in pixels with
`Euclidean distance` $\longrightarrow d(p,q) = \sqrt{(p-q)^2} = \sqrt{[\sum_{i=1}^{n}(p_i - q_i)^2]}$

To obtain a distance for two detections the algorithm takes their positions and calculates distance, but for trajectories, the algorithm has to use a different approach. In simple cases, where trajectories are consecutive, we can measure the distance between the end of the first trajectory and the start of the second trajectory. But as mentioned before, the algorithm must be capable of covering cases where trajectories overlap in time.

To measure the distance of overlapping trajectories, we can use some of the existing algorithms, but the problem with those algorithms is time complexity. In our case, we need only the distance of two points of trajectories because of the feature-connectivity of space overlapping and complex situations we are measuring in subsection 4.1.8. Thus, we can measure, for example, the distance from the start of the second trajectory to the end of the first one or any other significant points in the overlapping parts.

Distance to feature-connectivity

Once we have a distance, the next step is to transform the distance into a feature-connectivity.

The basic approach is to create a mapping of the distance into $\langle 0, 1\rangle$. For example:

- $distance \in [0, min\_distance] \longrightarrow fconnectivity = 1$

- $distance \in [min\_distance, max\_distance] \longrightarrow$
  $fconnectivity = map([min\_distance, max\_distance], [1, 0], distance)$

- $distance \in [max\_distance, inf] \longrightarrow fconnectivity = 0$

The disadvantage of this approach is that we have to define distance limits, and it does not consider time. Thus, in case of missing detections, the feature-connectivity can be distorted. To take the time into account, we have extended this approach in the following method in subsection 4.1.7.

Distance and speed to feature-connectivity

To measure the feature-connectivity, taking into account the time, we altered the previous method, and instead of measuring the distance, we measured the required speed to move the object from one position in a second using the shortest possible way $\longrightarrow$ *required_speed*, where the speed measurement can be seen in the previous *comparator* in subsection 4.1.6.

To transform the needed speed into feature-connectivity for trajectories, we can measure the speed of both compared trajectories and use the faster one as a limit $trajectory\_speed = max(trajectory1\_speed, trajectory2\_speed)$.

Then transform the needed speed into $\langle 0, 1 \rangle$ in a similar way as for distance. By user-defined values, we will set boundaries such as
$tolerated = trajectory\_speed * tolerance$, where *tolerance* defines a tolerance ratio, and $max\_limit = trajectory\_speed * max\_ratio$.

- $required\_speed \in [0, tolerated] \longrightarrow fconnectivity = 1$

- $required\_speed \in [tolerated, max\_limit]$
  $\longrightarrow fconnectivity = map([tolerated, max\_limit], [1, 0], distance)$

- $required\_speed \in [max\_limit, inf] \longrightarrow fconnectivity = 0$

Detections Since the detections have nothing like speed, we must determine a value representing a possible speed to use a similar approach for detection feature-connectivity. In our implementation, we measured the speed for a few trajectories and used it as an expected speed.

4.1.8 Trajectory position

Once we have generated larger and more complex trajectories, we can more exhaustively compare their time and space positions. This *comparator* is quite complex and uses generated features like direction and speed. Thus, it is better to use it on higher clustering levels after reducing the number of trajectories.

The result of this *comparator* is binary:

- trajectories are connectable by space conditions $\longrightarrow fconnectivity = 1$

- otherwise $\longrightarrow fconnectivity = 0$

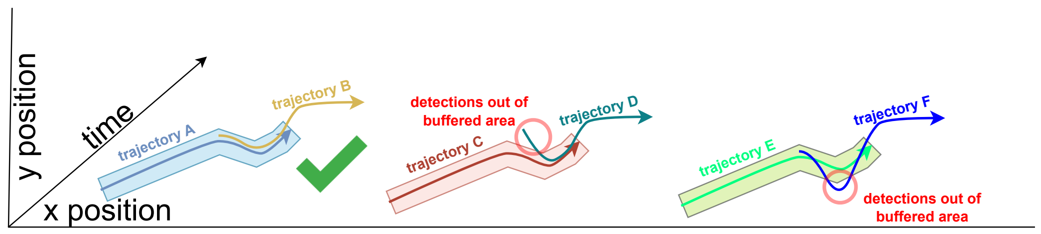


Figure 4.3: Trajectories compared by overlapped space

Trajectory curve

As the trajectory is defined - a set of detections, it represents a 3D curve with axis $time, position\_x, position\_y$, where each detection represents a single point. We can use it and, based on position, measure the trajectory feature-connectivity.

Time overlapped trajectories

In cases where trajectories are overlapped in time, the *comparator* verifies if the trajectories are also space overlapped. This means that all detections of one trajectory, in shared time, are present in the buffered area of the other trajectory, as shown in the picture 4.3.

To implement this measurement, we use a first trajectory to create a $tolerated\_area$ - area where the overlapped detections have to belong. Then the *comparator* extracts the overlapped detections $\longrightarrow$ *overlapped* from the second trajectory. To mark these two trajectories as connectable, each of the *overlapped* detections has to be in $tolerated\_area$. There are several approaches to evaluating if *overlapped* detections are in the area. In our case, we have used an existing library for space arithmetic, and around $tolerated\_area$ created a buffered area$\longrightarrow buffered_area$. Then used, a library method that verifies that the $overlapped\_detections$ belongs to $buffered\_area$.

- all *overlapped* detections belongs to $buffered\_area \longrightarrow fconnectivity = 1$
- otherwise $\longrightarrow fconnectivity = 0$

Time consecutive trajectories

In cases where trajectories are not overlapped in time, one trajectory ends before the second starts. The *comparator* verifies if the key point of one trajectory belongs to the expected area of the second one.

To achieve this measurement, we differ $first\_trajectory$ - earlier trajectory and $second\_trajectory$ - later trajectory. The *comparator* then for $first\_trajectory$ creates space area $\longrightarrow end\_cone$, where expects the beginning of the $second\_trajectory$, its first detection$\longrightarrow start_detection$. *Comparator* then verifies if $start\_detection$ is spatially present in end_cone. The same approach is applied in the opposite direction, as shown in the picture 4.4. If at least one of the conditions is fulfilled, trajectories are connectable. Otherwise, not.

To evaluate this *comparator* , we have once again used an existing library for space arithmetic.

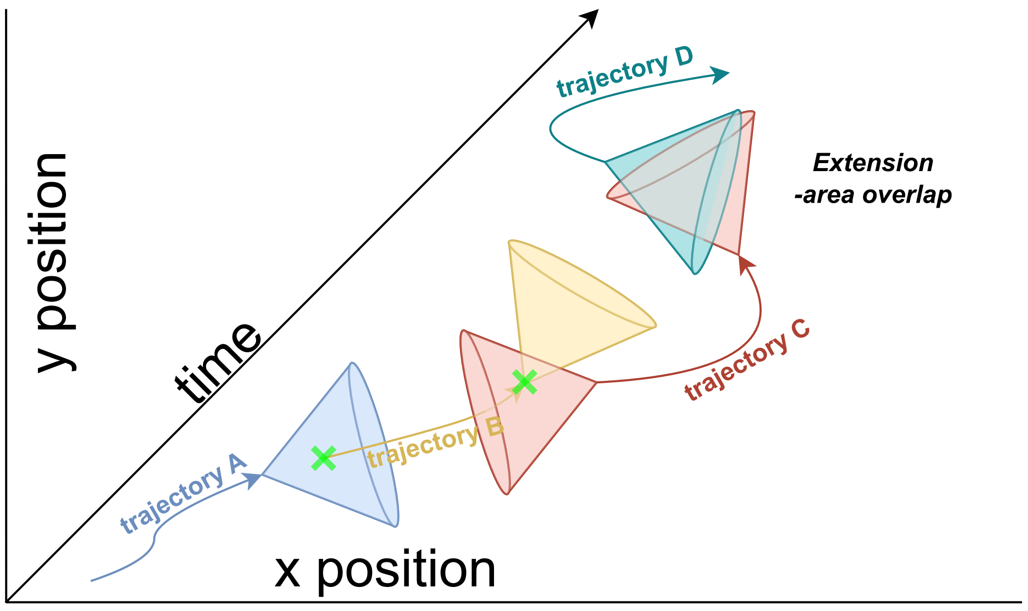


Figure 4.4: Trajectories compared by consecutive areas

- $start\_detection \in end\_cone \vee end\_detection \in start\_cone$
  $\longrightarrow fconnectivity = 1$
- otherwise $\longrightarrow fconnectivity = 0$

Extension for missing detections Thanks to missing detections, this method can exclude trajectories with sharp turns. Thus, in cases where trajectories are marked as unconnectable, to avoid problems with missing detections, we can measure the intersection of the end cones and, as feature-connectivity, return the intersection or some minimal value.

4.1.9 Image crop

The cropped image is one of the most useful features because it gives us visual information about detections or trajectories. A human being could probably create trajectories just based on the cropped images. We use cropped images to compare the colour similarity of two detections or trajectories.

Histogram

The best approach we came up with is to extract a histogram for cropped images. There are many ways to extract histograms, precisely analysed in other works[1][6][30][19]. Thus, we will rather focus on methods for getting histograms for trajectory, where are multiple images and we have to select which compare.

There are many possible ways to choose, which detections of trajectory to use, and all of them are valid, but bring different advantages and disadvantages. Thus we will describe a chain of decision that we use to define our approach.

All detections Theoretically, we could compare all detections, but it would be too slow. Thus, we have to reduce to use of only a subset of detections. One of the approaches could be to construct a single histogram from all detections of a trajectory, but this approach has a problem because it reduces the uniqueness of the object too much and creates histograms similar for many objects.

The nearest detections The next approach is similar, as we used for other features, it is to pick from trajectories the nearest detections in time or space. The great advantage is the similarity of the method and the fact that we are comparing only two detections. The disadvantage is that in cases where trajectories are on the point of light change. For example, when someone walks out of the shade or behind another object. Thus, our approach to reducing the impact of this is to, together with the nearest detections, always use a second pair of detections, for example, from the middle of the trajectory or a random one.

Extension of histograms

With generated cropped images, there is a problem in that images contain objects and the surroundings. Thus, the histogram is distorted and we could end up in a situation where the histogram represents the surroundings more than the object. To minimize this problem, there are several solutions with different complexity.

- Instead of the full image crop, use only the central part.
  $\longrightarrow$ for example cut off 15% from each side of the image crop
- weighted histogram, where the most valuable information is in the centre.
- Use a method to obtain an object silhouette and use only the object.[4]

Histogram to feature-connectivity

To compare histograms, it pairs of images, we used a method implementing $Pearson\ correlation\ coefficient$. For a pair of detections, the *comparator* directly compares the detection's histograms $\longrightarrow$ *compared_similarity*. For pairs of trajectories, the *comparator* compares the defined pairs of histograms. In our case, compares histograms of the nearest detections $\longrightarrow$ *nearest_similarity* and of the middle detections$\longrightarrow$ *middle_similarity*. Then as *compared_similarity* is used $max(nearest\_similarity, middle\_similarity)$ or alternatively

$$max(nearest\_similarity, avg(nearest\_similarity, middle\_similarity))$$

to take into account position. Where as feature-connectivity is then used, a result of a comparison $\longrightarrow$ *compared_similarity* or the mapping
$fconnectivity = map(\langle minimal\_similarity, 1\rangle, \langle 0, 1\rangle, compared\_similarity)$.

[4]This approach makes sense when the pre-computed silhouettes are in detection information.

4.1.10 Comparators in context

Be aware that the used *comparators* have to be compatible with used optimizations. For example, in cases where the number of comparisons is reduced with position optimization, as described in subsection 3.1.1, it is necessary to consider it while using position features.

Cached features

During this section, we have introduced many features, some of which need a not trivial time to be computed. To reduce time complexity, all the computed features should be cached. Thus, each feature is computed at most once.

4.2 Connectivity based on feature-connectivity

Once we have feature-connectivity for all features we are interested in, we need to classify the vector of feature-connectivity into a single number, as can be seen in the picture 4.5.

Definition 10. *Classification* is mapping the feature-connectivities of multiple features into a single value. $classification(\langle 0,1\rangle^n) \longrightarrow \langle 0,1\rangle$ where

$$v \in \langle 0,1\rangle^n classification(v) = \left\{ \begin{array}{ll} 0, & \exists x \in v; x = 0 \\ c \in \langle 0,1\rangle, & otherwise \end{array} \right\}$$

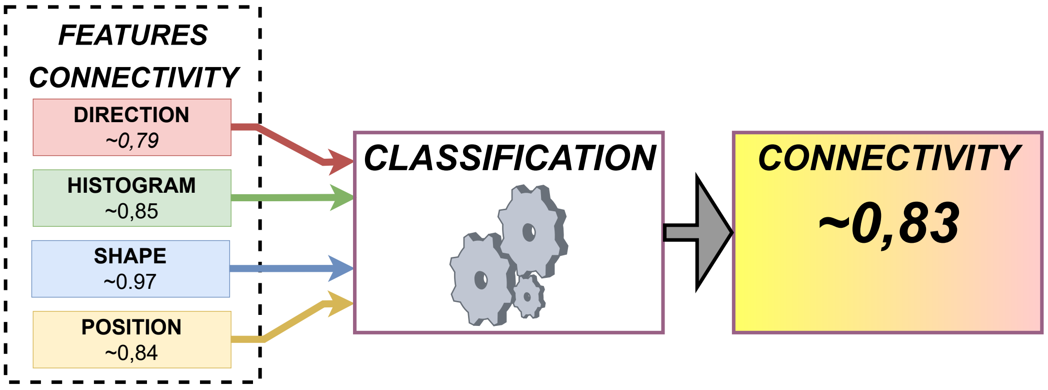


Figure 4.5: Transformation of feature-connectivity into connectivity

4.2.1 Classification methods

There exist a lot of approaches on how to implement a classifier. During our work, we have implemented and tested a few of them, and here we would like to describe their connection with features. For all the methods below, apply the rules. Thus, If any feature-connectivity equals zero, the connectivity is zero.

The average value

One of the basic approaches is to classify the vectors using an average value.

$$Classification(v) = \frac{\sum_{i=1}^{n}(v_i)}{n}$$

The minimal value

To classify the vector, we can use the minimal value. Which gives us minimal feature-connectivity in vector.
$Classification(v) = min(v)$

The maximal value

To classify the vector, we can use the maximal value. Which gives us maximal feature-connectivity in vector.
$Classification(v) = max(v)$
This approach is quite useless while using binary *comparators* because the *classifier* would end up every time with the *connectivity* $= 1$.

The product value

One of the approaches is to use the product of the vector.
$Classification(v) = \prod_{i=1}^{n}(v_i)$
This approach is great when we are sure about the feature-connectivity methods, but in some cases, the low feature-connectivity of one feature eliminates the potentially good result.

The weighted arithmetic mean value

To classify the vector, we can use the weighted arithmetic mean value.
$Classification(v) = \sqrt{[\sum_{i=1}^{n}(v_i)]}$
The problem with this approach is how to determine the weights.

Observation

In our implementation, we have tested all the methods above, and the average value is the most practical for our use.

4.2.2 Connectivity evaluation

Once we have defined methods for the whole chain, we can propose an algorithm to generate a feature-connectivity for a pair of detections or trajectories. The algorithm is universal for detections and trajectories. For simplicity, we will describe the algorithm for detections, but the algorithm also works for trajectories.

Evaluation of classifier

The algorithm as an input accepts a pair of detections to generate connectivity for $\longrightarrow$ *pair*, list of *comparators* $\longrightarrow$ *comparators_list*, and one of the classification methods $\longrightarrow$ *connectivity_method*. In a straight form algorithm, for a *pair* with each *comparator* from *comparators_list* generates feature-connectivity

$\longrightarrow$ *features_connectivity_vector*, and then with *connectivity_method* transform generated *features_connectivity_vector* in connectivity.

Optimisation To optimise evaluation, we use the fact that none of the feature-connectivity can equal zero. Thus, once the algorithm, for the *pair*, receives *feature-connectivity* $= 0$, the rest of the features is irrelevant and the *connectivity* $= 0$.

User-side optimisation The user defining the configuration can sort the *comparators* in a way where the most eliminative *comparators* are on the front positions and prevent evaluation of the following *comparators*. For example, the *class comparator* is quite useful as a first *comparator* because it directly eliminates interclass comparing.

Algorithm 8: *classifier.compare(args)*

```
Input: pair,comparators_list,connectivity_method
Output: connectivity
features_connectivity_vector = []
for comparator in comparators_list do
    feature-connectivity = With comparator evaluate connectivity of pair.
    if feature-connectivity == 0 then
        return 0
    features_connectivity_vector+ = feature-connectivity
total_connectivity =With connectivity_method evaluate connectivity of list_connectivity.
return total_connectivity
```

5. Interdetections and Trajectory descriptors

In this chapter, we will describe the usage of generated trajectories. The first usage is generating of missing detections by interpolating trajectories. The second one is to generate a semantic description of trajectories.

5.1 Interpolated detections

As described in the introduction, the detections we use for trajectory generating are incomplete. It is common that object in video is not detected in each frame and has missing detections. Thus, we have decided to use the generated trajectories and, based on them, generate the missing detections. To generate a missing detection, we will use interpolation. Thus, the detections generated by this method will be called interdetections. Each interdetection contains the same information as detection, plus the information about the trajectory it was based on. Interdetections can be used to retrain the original model or to smooth the trajectories. The disadvantage of this approach is, that the quality of generated interdetections is directly tied up with the quality of trajectories.

5.1.1 Interdetection generating

As input, the algorithm will use the original video, all detected detections and the generated trajectories for the video. The output of this process is interdetections, representing missing detections of objects in the video.

As we know, the object tracked by trajectory must always exist and cannot disappear. Thus, we know that the object is in the video but not detected. The reasons, which we can see in the picture 5.1, why the object is not detected can be the following:

- Object is, for a moment, out of the camera.
- Object is behind another object.
- Error of the detector.

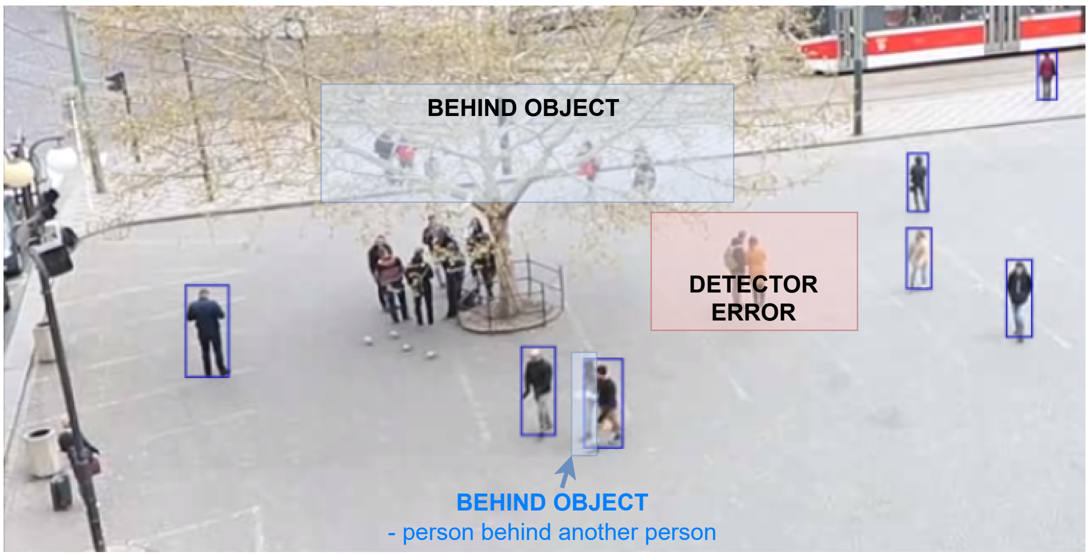


Figure 5.1: Undetected objects

In this scope, we will not target finding out the reason why the object is not detected and we will focus only on generating detections of the object in each frame $\longrightarrow$ *interdetections*.

5.1.2 Interpolating trajectories

The algorithm to generate interdetections starts with extracting timestamps for each frame $\longrightarrow$ *all_timestamps* from the video. Then we can determine, for each trajectory, timestamps of missing detections between trajectory *start_time* and *end_time*.

For each trajectory T, we can distinguish two lists of time stamps

- $present\_timestamps = \{d.timestamp | d \in T\}$
- $missing\_timestamps = \{t | t \in all\_timestamps \setminus present\_timestamps$ $\wedge T.strat\_time < t \wedge t < T.end\_time\}$

And create a mapping for each attribute that algorithm needs to interpolate:

- $trajectory\_center\_x = \{(t.timestamp, t.center\_x) | t \in T\}$
- $trajectory\_center\_y = \{(t.timestamp, t.center\_y) | t \in T\}$
- $trajectory\_width = \{(t.timestamp, t.width) | t \in T\}$
- $trajectory\_height = \{(t.timestamp, t.height) | t \in T\}$

Now the algorithm, thanks to this data, generates for each attribute an interpolation function and over *missing_timestamp* interpolates all attributes. For each foursome of interpolated attributes, we create interdetection, and once we have all trajectories processed, the algorithm can playthrough the video and generate image crops of interdetections as shown in the algorithm 9.

Algorithm 9: Interdetections from trajectories

```
Input: video, trajectories, detections
Output: interdetections
$interdetections = []$
$all\_timestamps$ = Get all timestamps from the video.
for $T$ in $trajectories$ do
    $present\_timestamps = \{t.timestamp | t \in T\}$
    $missing\_timestamps = \{t.timestamp | t \in all\_timestamps \setminus present\_timestamps$
        $\wedge\ T.strat\_time < t \wedge t < T.end\_time\}$

    $interpolation\_center\_x$ = Create interpolation of $\{(t.timestamp, t.center\_x) | t \in T\}$

    $interpolation\_center\_y$ = Create interpolation of $\{(t.timestamp, t.center\_y) | t \in T\}$

    $interpolation\_width$ = Create interpolation of $\{(t.timestamp, t.width) | t \in T\}$

    $interpolation\_height$ = Create interpolation of $\{(t.timestamp, t.height) | t \in T\}$

    for $f$ in $missing\_timestamps$ do
        $new\_interdetection$ =Create a new interdetection where:
            $center\_x = interpolation\_center\_x(f)$
            $center\_y = interpolation\_center\_yt(f)$
            $width = interpolation\_width(f)$
            $height = interpolation\_height(f)$
            $trajectory = T$
            $timestamp = f$
            $image\_crop = UNDEFINED$
        Add $new\_interdetection$ into $interdetections$

for $frame$ in $video$ do
    for $interdetection$ in $interdetections$;
    where $interdetection.timestamp == frame.timestap$ do
        $interdetection.image\_crop$ = From frame crop($center\_x, center\_y, width, height$) image
```

5.1.3 Interdetections to detections

As mentioned during the clustering of detections into trajectories, in the subsection 3.1.1, sometimes there are some leftover detections that are un-addable into any trajectory with the clustering algorithm and we mentioned that the best approach is not to assign them anywhere yet. The reason why they are not connectable can be various, but thanks to the interdetections, we can fix it and try to assign them into trajectories.

Matching detections with interdetections

To assign unmatched detections into trajectories, the algorithm will use all unmatched detections $\longrightarrow$ *unmatched* and generated
interdetections$\longrightarrow$ *all_interdetections* for a single video.

To optimise the time complexity, the algorithm focuses only on frames with unmatched detections, $frames = \{u.timestamp | u \in unmatched\}$. Thus, the algorithm will sort *all_interdetections* into
$interdetections = \{r | r \in all\_interdetections; r.timestamp \in frames\}$

Then the algorithm will, in each frame, compare the areas of *unmatched* with *interdetections* and if the best overlaying *interdetection* meets the conditions, for example, overlay between *unmatched* and *interdetection* is >= 90% of area $\longrightarrow$ *overlay_boundary*, the algorithm marks them as identical. Then we can delete newly created interdetection and the original detection insert into a trajectory.

Algorithm 10: Matching unmatched detections

```
Input: $all\_interdetections$, unmatched, $overlay\_boundary$
$frames = \{u.timestamp|u \in unmatched\}$
$interdetections = \{r|r \in all\_interdetections; r.timestamp \in frames\}$
for $f$ in $frames$ do
    $current\_interdetections = \{r|r \in interdetections; r.timestamp = f\}$
    $current\_unmatched = \{u|u \in unmatched; u.timestamp = f\}$
    for $unmatched$ in $current\_unmatched$ do
        $interdetection$ = From $current\_interdetections$ get with best overlay with $unmatcheded$.
        if $interdetection$ overlay unmatch $>= overlay\_boundary$ then
            Remove $interdetection$ from $current\_interdetections$
            Insert $unmatched$ into $interdetection.Trajectory$
            Delete $interdetection$.
```

5.1.4 Future work on this field

The interdetections give us a new space to research and develop. Thus, we have come up with a few ideas on how to interdetections extend and use in future work.

Visibility of interdetections Firstly, we would like to extend interdetections with flag information why the object wasn't detected in the first place.

Trajectory exterpolation Some of the trajectories end up in the middle of nowhere. It can be caused by a lack of detection and the trajectory is incomplete, or the object left the view, for example, entering the building. For the first cause, we would like to create an algorithm for trajectory extrapolation.

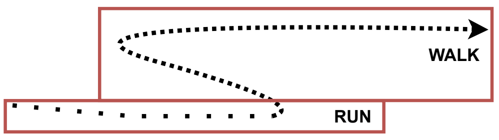
5.2 Trajectory descriptors

To make generated trajectories searchable, we have decided to generate, for each trajectory, a simple semantic description presenting its movement. However, the description analysis and the searching in trajectories are not a topic of this work. This section presents only a foundation for future descriptions.

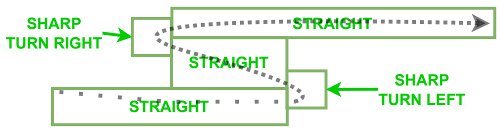
5.2.1 Describing steps

First, we must realize that we cannot describe a whole trajectory by a single value. Thus, the algorithm will split the trajectory into sub-trajectories of the consecutive detections, generate its attributes, and classify them as a type of movement.

Once we have classified each sub-trajectory, we can merge them into longer sub-trajectories and create by them the description. As a result, we would like to receive for each trajectory a sequence of its actions, for example, as we can see in the picture 5.2, $speed\_classification = [run, walk]; direction\_classification = [straight, left, straight, right, straight]$.



(a) speed description

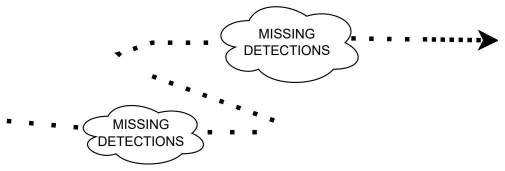


(b) direction description

Figure 5.2: Trajectory described by speed and direction

Sub-trajectories

There are a lot of methods for splitting a complete trajectory into sub-trajectories, and those methods influence the result. We have designed three methods to use and described their pros and cons. One of the key features of generated trajectories is that the algorithm can use the detections or uses detections and interdetections, which change the generated results. An example of input trajectory with interdetections and without interdetections can be seen in figure 5.3.



(a) trajectory

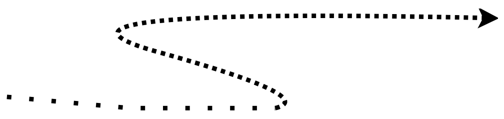
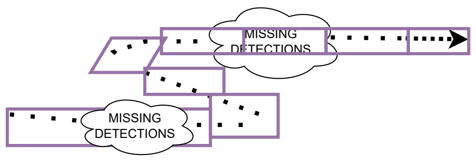

(b) trajectory with interdetections

Figure 5.3: Example of trajectory

Sub-trajectories of fixed time The first idea is to split trajectories into sub-trajectories of fixed duration. This method is easy to implement, but missing detections cause the problem because holes can occur in the trajectory without detections. Therefore, it is better to use detections and interdetections, as we can see in the picture 5.4.



(a) trajectory

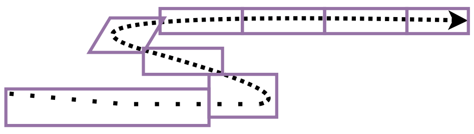
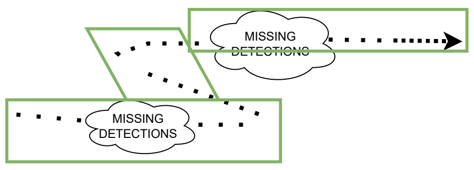
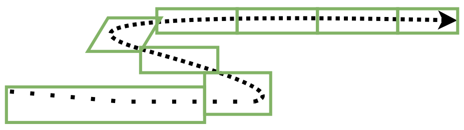

(b) trajectory with interdetections

Figure 5.4: Sub-trajectories of fixed time

Sub-trajectories of fixed length The next idea is to split trajectories into sub-trajectories of a fixed number of detections. This method is also easy to implement and is resistant to missing detections, as seen in the picture 5.5, but can distort the real trajectory. Therefore, it is better to use detections and interdetections.



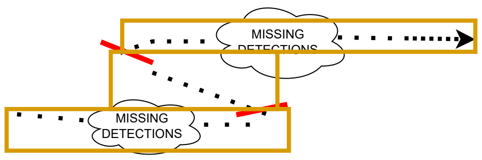
(a) trajectory



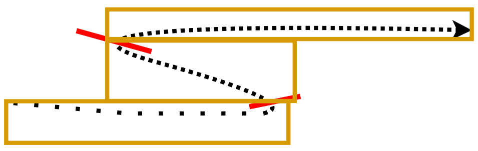
(b) trajectory with interdetections

Figure 5.5: Sub-trajectories of fixed length

Sub-trajectories based on simplified curve The last designed approach is based on trajectory key points. The algorithm reduces the number of detections with some compressing algorithm, such as the *Ramer–Douglas–Peucker* algorithm, and uses compressed points such as boundaries for sub-trajectories. This approach works equally well for both variants, as seen in the picture 5.6, but the disadvantage is that sub-trajectories created this way do not consider the speed information, which can be by this approach distorted.



(a) trajectory



(b) trajectory with interdetections

Figure 5.6: Sub-trajectories by key points

Bisection extension In some cases, for the methods above can be useful using bisecting of the intervals where each sub-trajectory is measured as a whole and as bisected. If the bisected parts are not similar, the sub-trajectory is split and measured again until the bisected parts are similar. The disadvantage of this approach is the need to define similarity and only one key measurement. Thus, we rather created small sub-trajectories and later merged them only on one of their classifications, as will be described in subsection 5.2.1.

Measured attributes

For trajectory, there can be a lot of measured attributes, but to meaningfully describe the trajectory, we are now reduced only to speed and angle of sub-trajectory. With attributes such as angle and speed, we can later classify if the object was slowly walking in a straight line or was zigzag running.

Speed attribute Measuring sub-trajectories' speed is done the same way as already described in the section 4.1.7. During the design of the speed attribute, we found that it is not easy to determine the unit. The basic approaches, such as using pixels, are impractical because the object's speed would depend on the camera's distance to objects and the camera's angle. Thus, we are measuring and storing the object's speed due to object size.

Angle attribute To determine the angle of the sub-trajectory, there is a need to set a basic axis and determine the angle of the sub-trajectory relative to this axis. Once the axis is set, in our case, it is vector $v(0, 1)$. We can count the angle in a clockwise direction. The measured angle determines the angle of the sub-trajectory in the frame as well as gives us information on the direction within the trajectory.

Sub-trajectories classification

Once we have the speed and angle for each sub-trajectory, we can compare sub-trajectories of trajectory with each other and classify the type of movement.

The boundaries, presented in the following classification, are determined by observing test data but are present only as an illustration.

Speed classification To classify sub-trajectory $\longrightarrow sub\_traj$ into a category based on speed, we use simple categories with observed boundaries. For example, for humans $\longrightarrow speed\_type = [stand, walk, run]$.

Classification, for sub-trajectory, is then assigned based on speed $\longrightarrow speed\_classification(sub\_traj) \longrightarrow speed\_type$

- $sub\_traj.speed \in \langle 0, min\_walk\_limit)$
  $\longrightarrow speed\_classification(sub\_traj) = (standing)$
- $sub\_traj.speed \in \langle min\_walk\_limit, max\_walk\_limit\rangle$
  $\longrightarrow speed\_classification(sub\_traj) = (walk)$
- $sub\_traj.speed \in (max\_walk\_limit, inf\rangle$
  $\longrightarrow speed\_classification(sub\_traj) = (run)$

For speed-related attributes, we can as well classify the acceleration, but to achieve this, we have to compare each sub-trajectory speed with the previous one.

Direction classification To classify the type of direction of the sub-trajectory, we decided to compare each sub-trajectory $\longrightarrow subtraj$ with the previous $\longrightarrow previous\_subtraj$ and the next one $\longrightarrow next\_subtraj$. Then the algorithm counts the

$$total\_difference = (next\_subtraj.angle - previous\_subtraj.angle)$$

and, based on the angle algorithm, classifies the type of the direction for $subtraj$.

$direction\_classification(subtraj) \longrightarrow direction\_type$ where

$$direction\_type =$$

$$[sharp\ left\ turn, left\ turn, straight, right\ turn, sharp\ right\ turn, standing]$$

In cases where the sub-trajectory is not moving - speed is classified as standing, the algorithm marks it as standing,$direction\_classification(subtraj) = (stand)$, otherwise:

- $total\_difference \in (-180°, -90°\rangle$
  $\longrightarrow direction\_classification(subtraj) = (sharp\ left\ turn)$

- $total\_difference \in (-90°, -15°)$
  $\longrightarrow direction\_classification(subtraj) = (left\ turn)$
- $total\_difference \in \langle -15°, 15°$
  $\rangle \longrightarrow direction\_classification(subtraj) = (straight)$
- $total\_difference \in (15°, 90°\rangle$
  $\longrightarrow direction\_classification(subtraj) = (right\ turn)$
- $total\_difference \in (90°, 180°\rangle$
  $\longrightarrow direction\_classification(subtraj) = (sharp\ right\ turn)$

The presented angles are only for illustration.

Merging over classification

As mentioned before, once sub-trajectories are classified, the algorithm merges them into longer sub-trajectories based on each classification. For example, the man whose trajectory is processed was running slalom - still running, but zigzag. Thus, we cannot, on the level we are targeting, mark the whole trajectory as "slalom", but we can mark the whole trajectory on speed classification as *run* and on direction classification as sequence $[straight, left\ turn, straight,$ $right\ turn, straight, left\ turn, ...]$. Determining the specific meaning of the trajectory is out of our scope, but this foundation of semantic description could be used for it.

Merging by classification Merging by classification is then implemented by grouping the consecutive sub-trajectories with the same classification. Grouping is run over each classifier and union the sub-trajectories with the same classification. As a result, we obtain a mapping of classification on multiple sub-trajectories.

5.2.2 Description structure

While creating descriptions of the trajectories, information becomes more and more abstract. To create an effective way of storing trajectory descriptions and keeping the link to all core information, we have designed a structure that keeps all the information accessible through a tree structure.

On the bottom level are the detections of trajectory. The middle section is represented by sub-trajectories with measured attributes and classified information at the top layer, as illustrated in the picture 5.7. The structure is designed to enable adding other layers with more abstract information in the future.

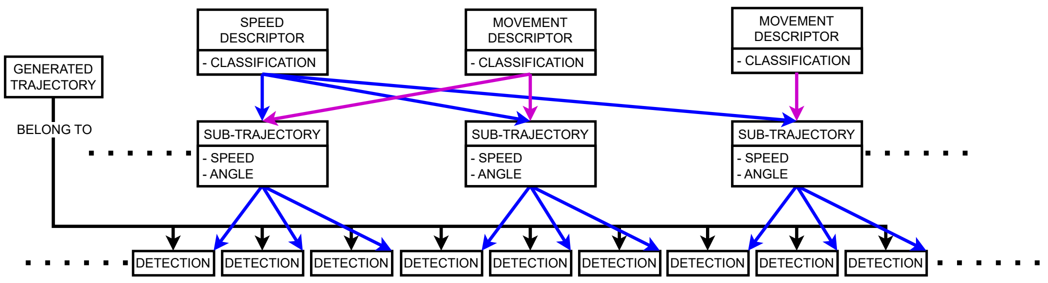


Figure 5.7: Tree structure of a semantic description

5.2.3 Generating descriptions

In the previous subsections, we have prepared all the needed components, and now the algorithm can create descriptions of the trajectories. The algorithm accepts, as input, trajectories, a method for splitting trajectories into sub-trajectories $\longrightarrow$ *subtraj_generator*, and a list of attributes $\longrightarrow$ *descriptor_list* and classifications $\longrightarrow$ *classification_list* to generate. For each trajectory, the algorithm first splits the trajectory into sub-trajectories.

For each sub-trajectory generates its attributes and once all sub-trajectories are enriched with attributes, the algorithm classifies sub-trajectories.

In the end, the algorithm clusters consecutive sub-trajectories with the same classification and stores the information in the proposed structure, as seen in pseudocode 11.

Algorithm 11: Alghorithm describing trajectories

```
Input: trajectories,subtraj_generator,descriptor_list,classification_list
for trajectory in trajectories do
    sub_trajectories = Split trajectory into sub-trajectories with subtraj_generator.
    for sub_trajectory in sub_trajectories do
        # In described approach attribute_list = [speed, angle]
        for attribute in attribute_list do
            sub_trajectory[attribute.name] = attribute(sub_trajectory)
    Store sub-trajectories into the database.
    for classification in classification_list do
        for sub_trajectory in sub_trajectories do
            sub_trajectory.[classification.name] = classification(sub_trajectory)
        merged = Merge consecutive sub-trajectories with the same classification.
        Store merged blocks into the database.
```

6. Implementation into Videolytics

To prove our concept and create an evaluation of trajectories generated by proposed algorithms, we have implemented a *Traged* pipeline into a system called Videolytics introduced in section 1.2.2.

6.1 The provided data

In the Videolytics are created detections by previous modules and we use them for trajectories generating, as shown in the picture 6.1. The prepared detections are in line with our requirements on detections properties, defined in definition 1. Thus, we can directly use them without any further preparation.

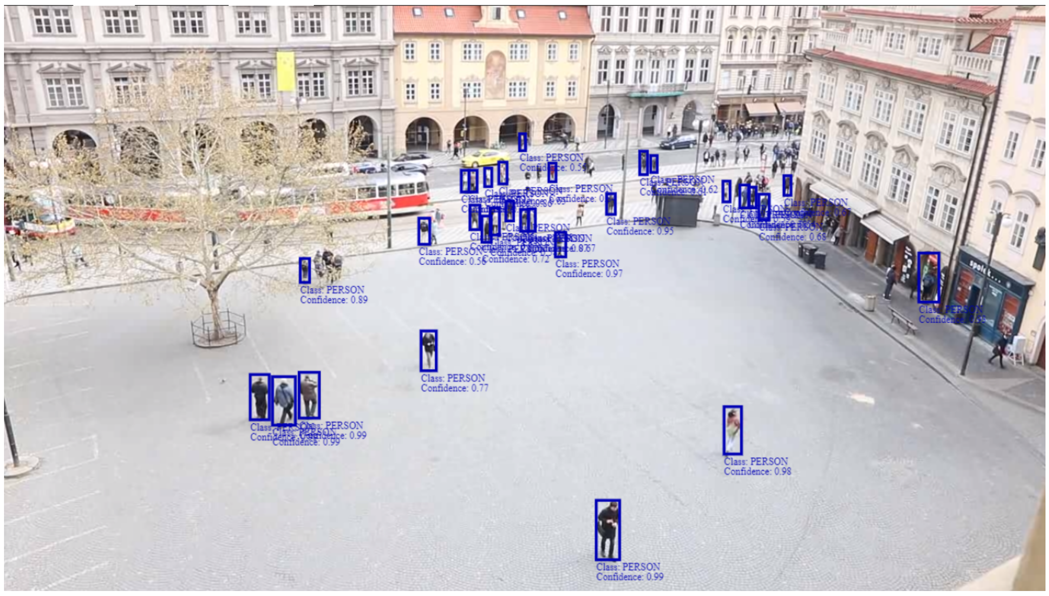


Figure 6.1: Detections in the Videolytics video

6.1.1 The missing scenarios and scenography

During the work on the *Traged*, we noticed that the Videolytics dataset is missing interesting scenarios and camera views for us. Thus, we have recorded and added a set of videos[1] containing the needed scenarios and scenography for future projects. Each video was shot from 3 points and contained the needed action multiple times.

- The fight scene
- The robbery scene
- The meeting of pairs or groups
- The stalkers

[1]Not all of them were added into videolytics because of the memory limits. The server is currently migrating to a new server.

6.2 The database

Since the main common point for all modules in Videolytics is a database, we looked at it and considered its form in our work to avoid duplicating or remaking already implemented parts. As we can see in the picture 6.2 of database hierarchy, the database in the current state already contains prepared tables for trajectory storing because there is already implemented visualization in web application[2]. The tables in videolytics, that we used:

- *camera* - information about the video (camera)
- *detection* - detection record
- *frame* - record for each frame
- *traj* - record for each trajectory
- *traj_model* - a record for each configuration, used to generate trajectories
- *traj_detection* - mapping detections to the trajectory, representing that detection is a member of the trajectory

After analysis, we have decided that the provided structures fulfill all our requirements and we can use them without any changes. The only need was to extend a database with a new structure for interdetections, annotated data and trajectory descriptions, as shown in the picture 6.3. The newly generated tables in videolytics:

- *annotated_trajectory* - a record for each annotated trajectory
- *annotated_detection* - mapping detections to the annotated trajectory, representing that detection is a member of the annotated trajectory
- *interdetection_generation* - record, keeping information about interdetections origin - trajectory model it was based on and for multiple runs, the generation distinguishing each run
- *interdetection* - interpolated detection record
- *descriptor_generation* - record, keeping information about descriptors origin - trajectory model it was based on and for multiple runs, the generation distinguishing each run
- *descriptor* - a set of sub-trajectories with the same classification over one feature
- *block_descriptor* - mapping of sub-trajectories to the descriptor, representing that sub-trajectory is a member of the descriptor
- *block* - sub-trajectory of trajectory with measured features

[2]Web application can be seen on http://videolytics.ms.mff.cuni.cz/stream.html

- *block_detection* - mapping detections to the sub-trajectory, representing that detection is a member of the sub-trajectory

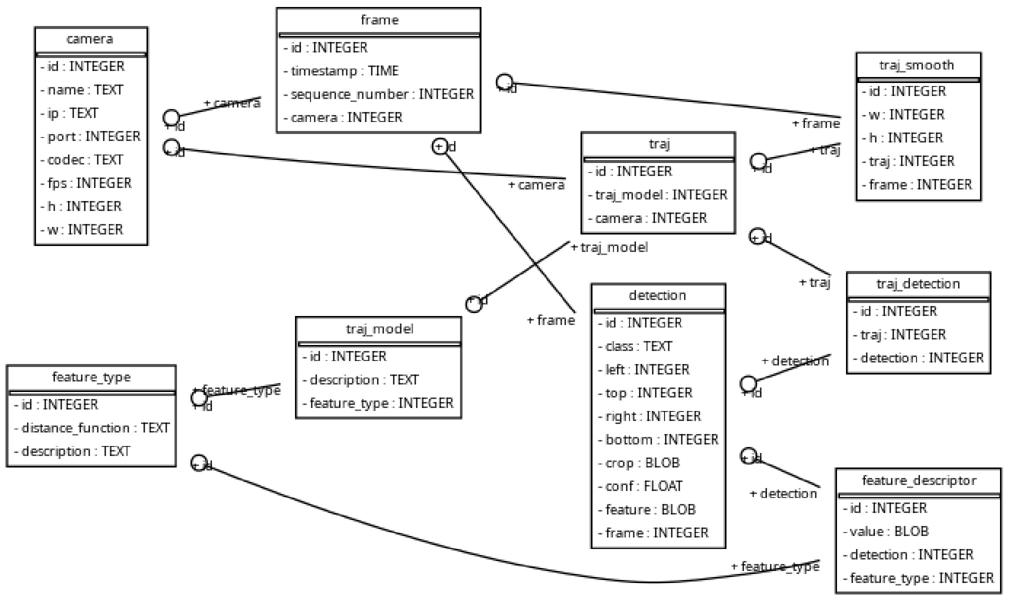


Figure 6.2: Data structures in Videolytics database

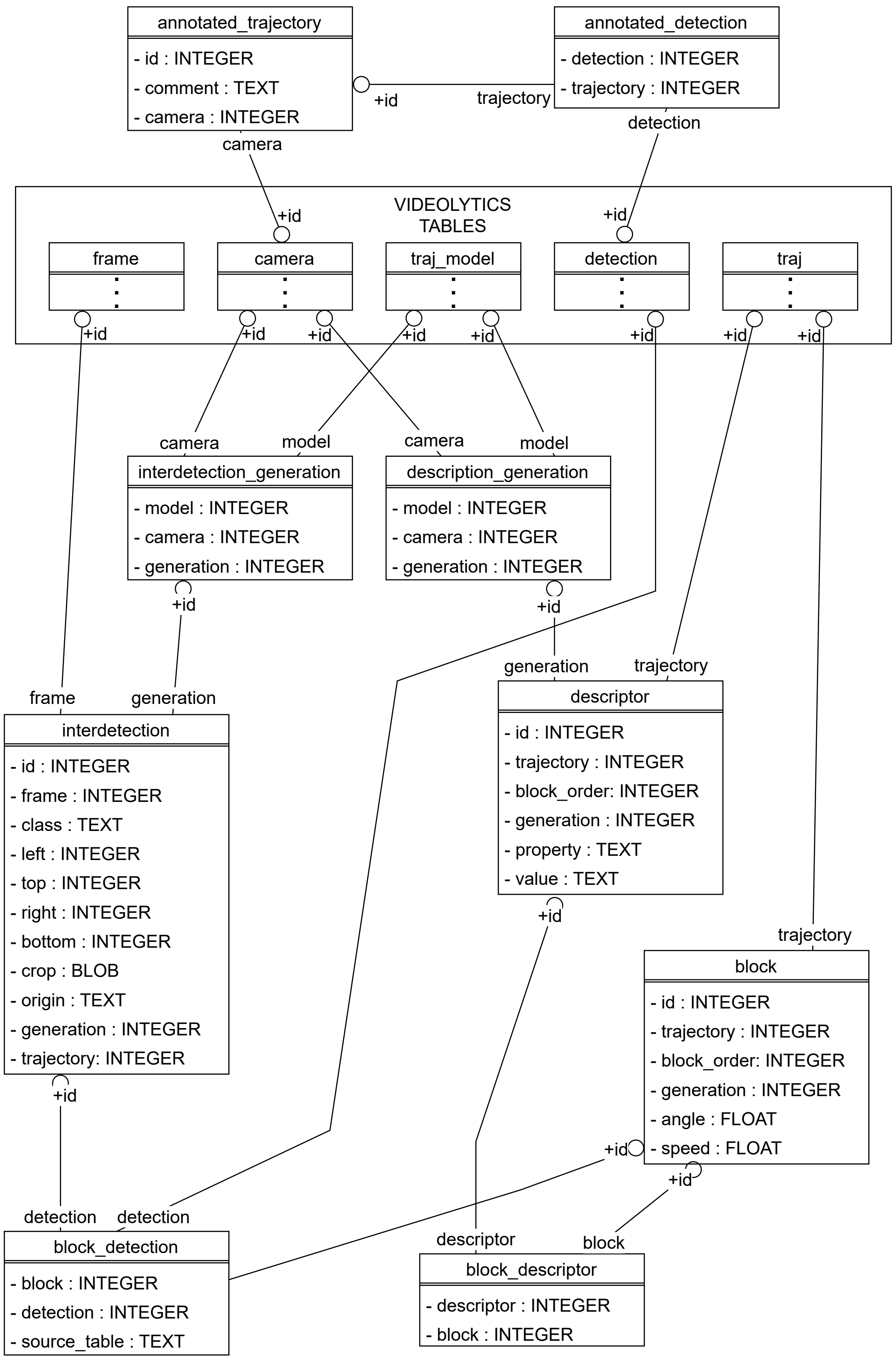


Figure 6.3: Extended structures in Videolytics database

6.3 Configurations

As mentioned in the algorithm description, the algorithm process uses a user-defined configuration, determining methods to compare a pair of trajectories or detections, strategies and how to build a clustering tree. Our configuration is written in JSON format and gives the developer freedom of form. Thus, developers can define in a structure any properties they need. The main blocks in the structure of configuration, as can be seen in the diagram picture 6.4 are:

- metadata - The information about the configuration itself, such as comments on used strategy and so on.
- loadFrame - The information on how to load data from the database and how to merge.
  - loadAndMerge - The description on how to merge trajectories, information such as overlap, and strategies to use.
- ”generator”- Representation of tree nodes as presented in section 3.2.1.
  - *comparator* - List of comparators to get feature-connectivities.
  - *classifier* - The method how to merge feature-connectivities into connectivity.
  - nodes - The descendant generators for the clustering.

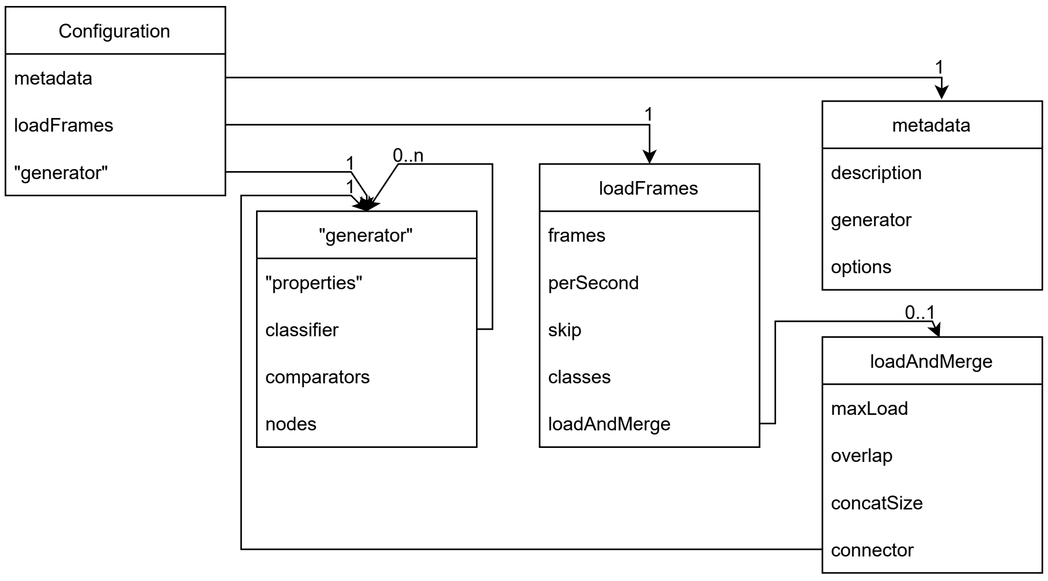


Figure 6.4: Configuration structure in JSON

6.4 The code base

To introduce the whole implementation, we will briefly provide the code structure, the class hierarchy, and the *Traged* algorithms mapping to implemented code. For the implementation details and the full code structure, please refer to the attached code repository/directory with all the code, user and technical documentation.

6.4.1 The structure

The main code is divided into three parts. Each of them is independently runnable and data are shared (loaded and stored) throughout the database to align with the videolytics approach. The additional data, such as the configuration file or video id, can be passed with command line arguments. The parts are ordered as presented and as they should be run:

- *Traged* - Clustering detections into trajectories
- *Interdetections* - Trajectory interpolation into interdetections
- *Descriptions* - Semantic description of trajectories

6.4.2 Traged

The first part is responsible for clustering detections into trajectories and basically implements chapters 3 and 4 The code is runnable through the file `traged.py`, which manages the database communication, runs the clustering operations for the sub-parts and corresponds to algorithms introduced in subsection 3.2.2. In the folder `Generators`, can be found elements of tree and the tree evaluation algorithm as described in subsection 3.2.1, where elements implement the clustering operations as proposed in section 3.1.

In the folder `Comparators` is implemented chapter 4. The descendants of class `IComparator` implement *comparator*'s as described in section 4.1 and are responsible for feature extraction and feature-connectivity evaluation. The descendants of class `IClassification` implement $classifier$'s as described in section 4.2.

6.4.3 Interdetections

The second part generates interdetections as proposed in section 5.1. The code is runnable through the file `interdetection_generator.py` and the code is in folder `Interdetections`.

6.4.4 Descriptions

The third part generates descriptions of trajectories as proposed in section 5.2. The code is runnable through the file `descriptor_generator.py` and the code is in folder `Descriptors`.

6.4.5 Technologies

To implement our module, we were not restricted to any language. Thus, we had to decide on a programming language and we decided on Python. Even though Python is not the fastest possible option, Python has a great advantage over the others and it is that Python has great library support and is easily writable. Thus, we were able to test a large number of different approaches and methods in a short time. Also, during the implementation, we used Python support for runtime class and object modification, which helped us with data caching a lot and dynamic detection to trajectory mapping.

6.5 Annotations and data search

During the work, we lacked annotated data for basic evaluation and any simple way to display the data from the dataset to research corner cases. Thus, we have created a simple application, shown in the picture 6.5, for basic detection and trajectory visualization with functions to create trajectory annotation and annotate some videos with a few trajectories.

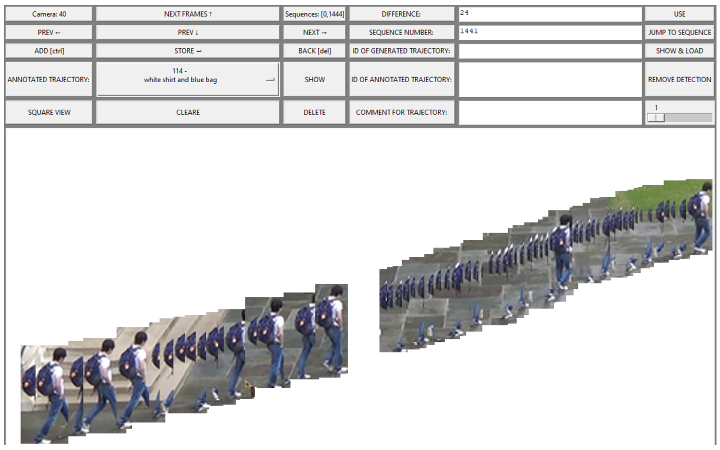

Camera: 40
Sequences: [0,1444]
DIFFERENCE: 24
SEQUENCE NUMBER: 1441
ID OF GENERATED TRAJECTORY:
ANNOTATED TRAJECTORY: 114 - white shirt and blue bag
ID OF ANNOTATED TRAJECTORY:
COMMENT FOR TRAJECTORY:

Figure 6.5: Simple app for data search and annotations

6.5.1 Annotation process

To describe the process and basic operation, that user $\longleftarrow$ *annotator* does to annotate the video trajectories, we write it as a simplified chain of steps.

1. *Annotator* runs the application with an id of the annotated video in the database.

2. If needed, *annotator* specifies the number of frames, and where to start $\longrightarrow$ $current\_DFrame$. Otherwise, $current\_DFrame$ is the first frame with detections.

3. *Aplications* display detection from current $current\_DFrame$.

4. *Annottator* can with a key or button browse detections from $current\_DFrame$ or jump on next/previous Dframe.

5. Once *annottator* chooses the object to track, confirm with a `Ctrl` key.

6. *Aplication* set next DFrame as $current\_DFrame$ and display the nearest detection to chosen one from current $current\_DFrame$.

7. *Annottator* can GOTO step 4, or enter the description of the trajectory and with `Enter` store the trajectory into Annotated.

8. Repeat step 3.

7. Evaluation

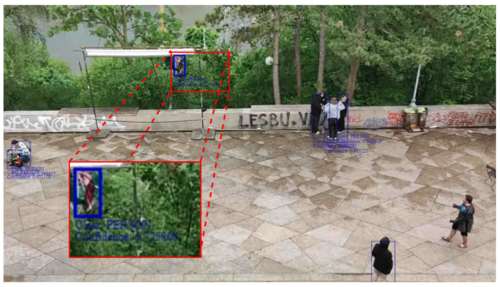
In this chapter, we would like to evaluate the proposed algorithms and methods overall as implemented in the Videolytics. First of all, in section 7.1 is a review of the videolytics dataset that we used for evaluation. Then in the following section 7.2 is described the process of configuration of the *Traged* and in section 7.3 are evaluated the generated trajectories and described observation. Finally, in section 7.4 are presented outputs of interdetections and semantic descriptors.

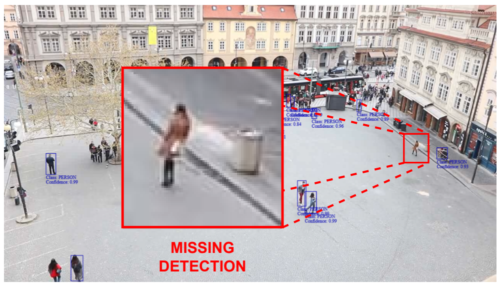
7.1 Dataset

To get to know the Videolytics and prepare the ground for our experiments, we will first investigate the dataset and its videos. The Videolytics dataset, in its current state, is primarily focused on the videos of squares and promenades. Thus, the videos contain plenty of human detection but lack detection of different objects, such as cars, which is not a problem since we are focused on trajectories for humans and will use only human detection. The problem was the lack of videos, but it was solved by creating new materials, as described in section 6.1.1, and using the videolytics pipeline to generate detections.

7.1.1 Detections

Detections in the dataset are present for all videos and are of satisfying quality but are not perfect. The two most common problems with provided detections, as shown in the picture, are longer missing detections, where the object is not detected for a few consecutive seconds and false positive detections, where in data-set are present detections even though there is no object at all. Both cases can be seen in the picture 7.1. As mentioned, we aimed to develop a robust system immune to missing detections. Thus we have not altered detections anyhow and straight-use generated detections.


(a) False detection



(b) Missing detection

Figure 7.1: Detection failures

7.1.2 The videos

The recording suitable for our needs can be distinguished in two cases:

Videolitics dataset						
id	name	view	SIM	problematic	ref_img	time
40	*duke*	close	∅	∅	7.2(a)	60s
43	001_*Cannon_FHD*	wide	∅	tree crown	7.2(b)	300s
48	002_*Cannon_FHD*	wide	∅	tree crown	7.2(b)	120s
469	*tram_1*	wide	470	tree shadow	7.3(b)	1079s
470	*tram_2*	wide	469	tree shadow	7.3(a)	1035
920	*rvacka_pravo*	wide	922	∅	7.4(b)	577s
921	*kradez_stred*	close	928	∅	7.4(a)	329s
922	*rvacka_stred*	close	920	∅	7.4(a)	437s
928	*kradez_pravo*	wide	921	∅	7.4(b)	344s

Table 7.1: Videolitics dataset summarization

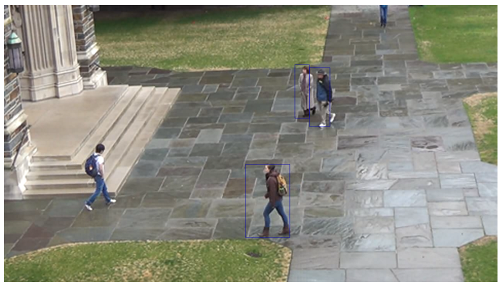
- close-view recording - recording, where objects all over the view are similarly sized, as shown in the picture 7.2(a)
- wide-view recording - recording of a large area, where objects are diametrically different sized, as shown in the picture 7.2(b)

To summarize the dataset recording, we have created a table 7.1.2 with all the needed information, where columns represent:

- id - The id of recording in the database
- name - The name of the recording in videolytics as presented on the web
- view - The type of view as described above
  - [close-view recording $\longrightarrow$ close,wide-view recording $\longrightarrow$ wide]
- problematic - Problematic parts of the video, if any
- simultaneous - SIM - Ids of videos that capture the same place simultaneously
- ref_img - The picture representing the recording view [1]
- time - Approximate video length in second

To make presented data in a table understandable, there is a text description of row with $id = 469$. The video with $id = 469$, can be accessed on videolytics web, under the name *tram*_1. The video is a wide-view, where objects are in the distance and as well relatively close to the camera, and the recording is a simultaneous with the video with $id = 470$. The video, for trajectory-generating purposes, contains a problematic area with a tree shadow, as can be seen in reference picture 7.3(b).

[1]Some recordings have the same view. Thus, we will not attach reference frames for each of them and will use the most similar ones.


(a) close-view recording

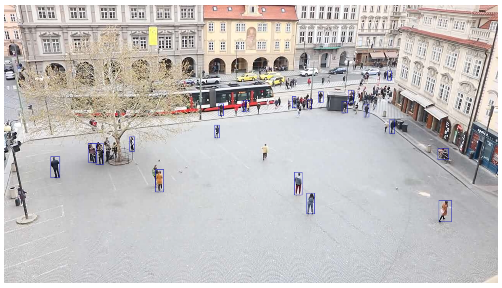
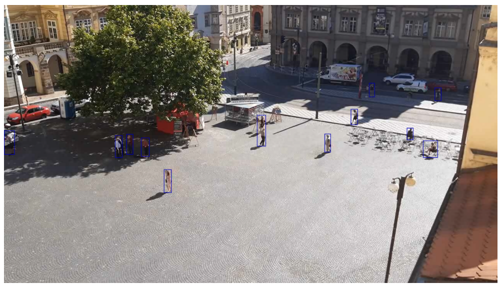

(b) wide-view recording

Figure 7.2: Videos of squares with a different view properties


(a) wide-view recording - right

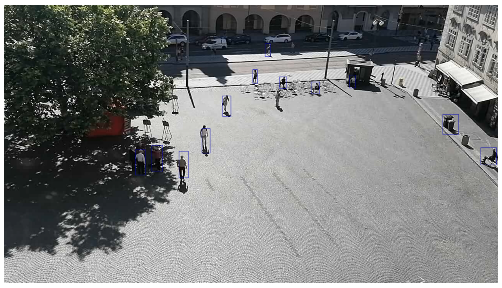
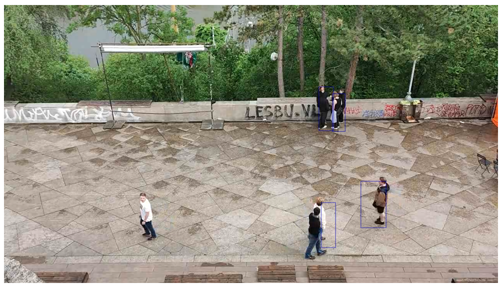

(b) wide-view recording - left

Figure 7.3: Videos of the square during a sunny day


(a) close recorded - fight

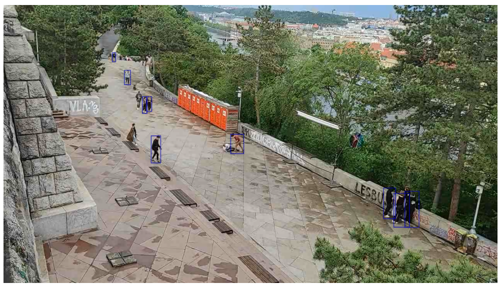

(b) wide-view recording - fight

Figure 7.4: Videos of the promenade, with staged fight scenes

7.2 Trajectories

In this section, we evaluate the flow for generating trajectories as well as the generated trajectories themselves. First of all, in section 7.2.1, we will describe how we designed the configuration for the *Traged*, the metric we used during the process and our observations. Later, in section 7.3, is presented a comparison of *Traged* against the `OpenCV tracker`.

7.2.1 Configuration design

As mentioned in chapter 3, our analytic approach runs based on the user-defined configuration specifying the tree of clustering operations. Thus, the key task during the algorithm evaluation was to define the configuration and find the optimal mapping $configurations \longleftrightarrow videos$ because, during the development, we had a question about how good and reusable configurations will be. If there is a need to create a configuration directly for each video, or if exists one universal configuration covering all of the videos, this question is for used data answered in this section.

Internal evaluation metric

First, we needed to evaluate the results of the designed configurations. The first evaluation method was observing, which means for each run with a different configuration, we observed the generated trajectories and used the observation to adjust the configuration. Observation is a slow process, and there is a great risk of missing some errors. Thus, we have prepared annotated[2] trajectories and based on observation, we identify the possible states of annotated and generated trajectories, prepared a few annotated trajectories in each video and proposed metrics for configuration evaluation.

The possible states of $annotated - generated$ trajectories:

- $One - Zero$[3] - Annotated trajectory does not have generated trajectory at all.
- $One - One$ - Annotated trajectory has exactly one generated trajectory. The goal.
- $One - Many$ - Annotated trajectory has many generated trajectories.
- $Many - One$ - Many annotated trajectories have one shared generated trajectory.
- $Many - Many$ - Many annotated trajectories have many generated trajectories.

Each state represents a different problem of the configuration, and each has a different severity and resolution. In the order of seriousness:

- $One - One$ - Our goal. Nothing to do.
- $One - Many$ - Generated trajectories are under-clustered. There is a need to add another layer of clustering or lower the feature-boundaries for clustering.
- $Many - One$ - Trajectories are over-clustered. There is a need to increase the feature-boundaries for clustering.
- $Many - Many$ - Trajectories are wrongly clustered. There is a need to adjust multiple feature-boundaries.

[2]We did not annotate all trajectories in each video, because it would be extremely time-consuming. Instead of it, we annotated an interesting situation, corner cases and so on.

[3]There is also a possibility, $Zero - One$, where exist generated trajectory, but there is no annotated trajectory, which we ignore because we are unable to annotate all the real trajectories.

- $One - Zero$ - There is no generated trajectory for annotated, which means, the algorithm probably did not process part of the video at all.

As a result of the clustering by a configuration, a developer can directly use the generated numbers of each category and, by proposed resolution strategies, adjust the values of configuration or can use a *merged* value.

$$merged = \frac{\#(One-One)*4+\#(One-Many)*3+\#(Many-One)*2+\#(Many-Many)*1}{\#annotated*4}$$

The configuration development

We started with a plan to create one universal configuration for all of the videos in the dataset, as presented in table 7.1.2. As a learning video, a video we used to define configuration parameters, we chose the close-view recorded video *duke* with $id = 40$. As the test video, a video where we test the configuration, we chose wide-view recording 001_Cannon_FHD with $id = 43$.

In the beginning, we had reached quite good results with a single configuration, and for some time, the results were getting better, but over time and many tests, we ended up in a situation where whatever we tried, the generated trajectories were not improving on a learning video. At that moment, we swapped the learning and the test video and did the same procedure. With this approach, we were able to receive "good" results on the learning video and "bad" results on the test video or a "bit better than bad" results on both of them.

Thus, we have decided that we cannot process all the videos with a single configuration and have to separate them into two or multiple categories. We then lowered the expected limits, and the learning and test videos were used for the videos with a same view. One configuration was created for close-view recording videos and one for wide-view recordings. Once we were satisfied with the results on the learning video, we generated trajectories on the test videos.

To present the results we have generated trajectories with the configuration - in the web are marked as model $model = 2198$ for close-view recordings and $model = 2259$ for wide-view recordings. The generated trajectories can be seen in the pictures 7.5, 7.6, 7.7, or directly in the videos on the web[4].

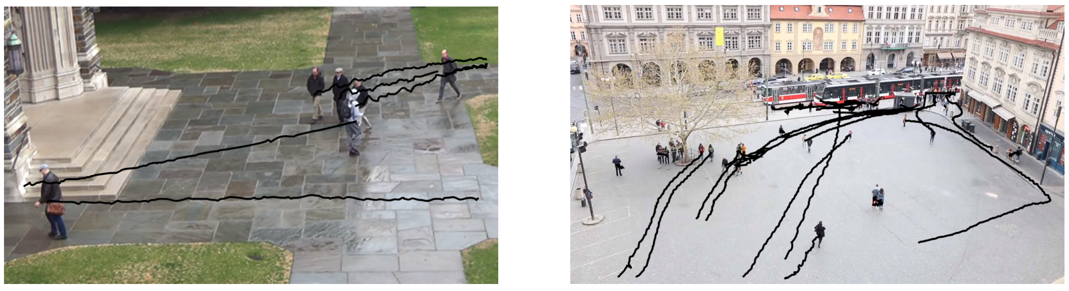
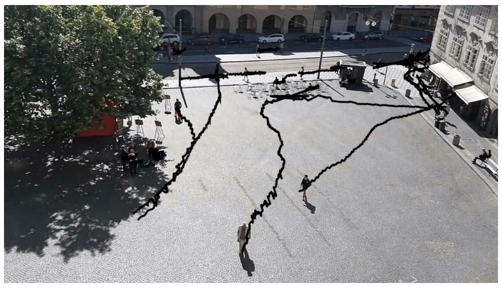

(a) close-view recording (b) wide-view recording

Figure 7.5: Trajectories in squares with a different view properties

[4]Videolytics web - http://videolytics.ms.mff.cuni.cz/stream.html


(a) wide-view recording

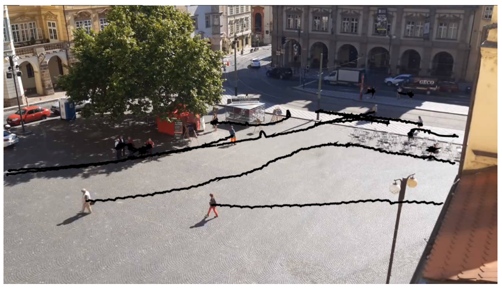
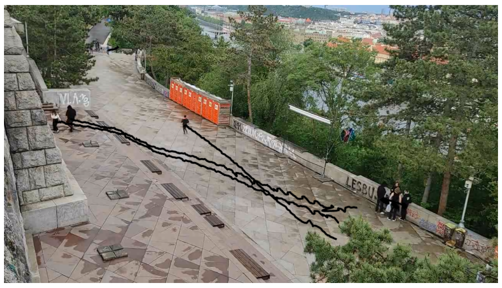

(b) wide-view recording

Figure 7.6: Trajectories in the square during a sunny day


(a) wide-view recording - fight

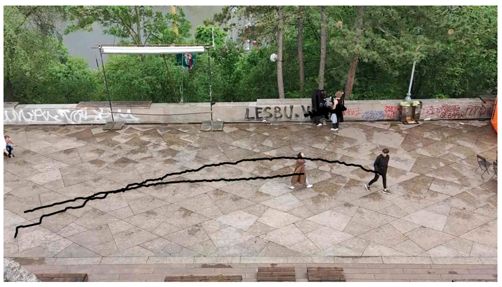

(b) close-view recording - fight

Figure 7.7: Trajectories in the promenade, with staged fight scenes

Results: As can be seen in the tables $\longrightarrow$ table 7.2.1 for close-view recordings and table 7.2.1 for wide-view recordings, with the most recent configurations of all evaluated videos, we did not reach for learning videos the optimal state, where each object has exactly one trajectory. After the data investigation, we discovered that, for example, for learning video with $id = 40$, the problematic parts, with trajectories in state $Many - Many$, are caused by objects overlaying over each other and missing detections caused by detector errors, as illustrated in the picture 7.8 presenting the generated output. In the picture, there is a couple, detected while walking next to each other. After some time, the woman is not detected at all, because she is walking behind the man. When woman walks out of the man, she is detected, but in the same time the detector stops detecting the man. Thus, the trajectory is from man passed to the woman.

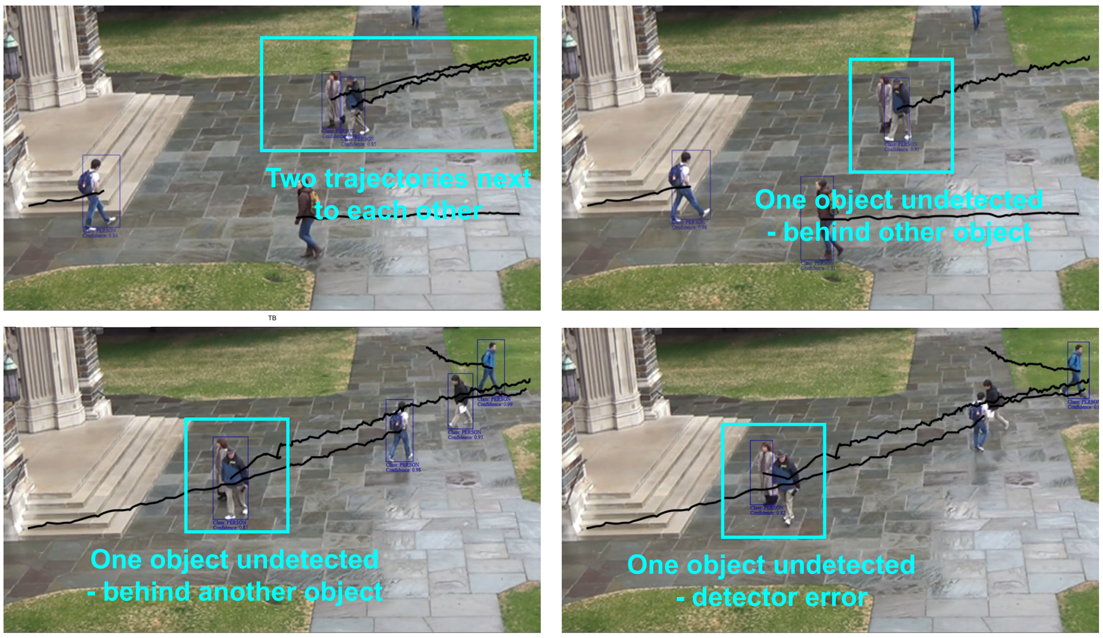

Two trajectories next to each other
One object undetected - behind other object
One object undetected - behind another object
One object undetected - detector error

Figure 7.8: Objects overlaying each other

Also, for the videos with trajectories generated based on the configuration for close-view recording, we have a lot of trajectories in the $Many - Many$ state, as can be seen in the picture 7.9. In the video there is a lot of objects standing next to each other, where trajectories are jumping from one to another due to the lack of detection.

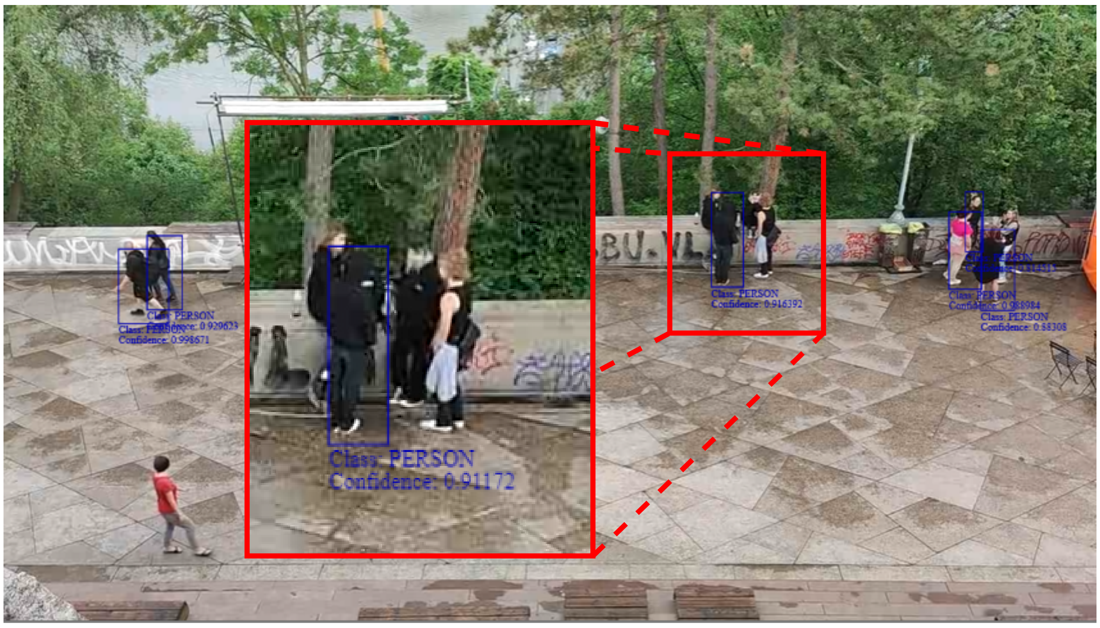


Figure 7.9: Standing people, undetected

Internal results for close recording						
id	One-One	One-Many	Many-One	Many-Many	One-Zero	merged
40	0.75	0.0	0.0	0.25	0.0	0.8125
921	0.31	0.19	0.0	0.5	0.0	0.58
922	0.3	0.5	0.0	0.2	0.0	0.725

Table 7.2: Results for the close-view recordings - model 2198

Internal results for wide recording						
id	One-One	One-Many	Many-One	Many-Many	One-Zero	merged
43	0.55	0.16	0.0	0.29	0.0	0.74
48	0.71	0.1	0.0	0.19	0.0	0.83
469	0.36	0.5	0.0	0.14	0.0	0.77
470	0.64	0.36	0.0	0.0	0.0	0.91
920	0.53	0.29	0.0	0.18	0.0	0.79
928	0.77	0.08	0.0	0.15	0.0	0.87

Table 7.3: Results for the wide-view recordings - model 2259

Next improvements: In the future, we would like to improve the existing configurations to maximize trajectories in state $One - One$, as well as create new configurations for different types of videos. To achieve that, we would like to try an automatic process to adjust and alter the parameters to improve the results. At this moment, it seems the best approach is to use evolution algorithms. Furthermore, we would like to prepare a process, deciding which configuration to use, base on video properties to automatize the whole process.

7.3 Evaluation of generated trajectories

In this section, we would like to evaluate the results of our approach and compare our generated trajectories to the trajectories generated with a different tool or approaches. First of all, in section 7.3.1, we will choose a method to compare results with. Later in section 7.3.2, we will propose the used metrics, and finally, in section 7.3.3, we will evaluate the measured data.

7.3.1 The reference approach

The first idea was to use any well-known benchmark for trajectory-oriented algorithms, such as MOTChallenge[27][18], but we have faced the problem of being too narrowly focused on specified camera views, and MOTChallenge videos are not working for us. Thus, we have decided to change the strategy and compare our results against a well-known tool also used in some benchmarks.

We have decided to compare the *Traged* with OpenCV[4]. OpenCV is one of the most used libraries in the computer vision field and for OpenCV, there are already works [8], [38], [14], [5] that use OpenCV as a reference method and benchmarks using OpenCV methods against each other as well as against different works. In our measurement we will use trajectories generated by code from the benchmark [8].

7.3.2 Metrics

We are using a set of different metrics from different sources to measure the quality of generated trajectories on multiple levels. Some of our metrics are based on metrics proposed in papers [22] and [38], and some of the used metrics are based on metrics described in the previous section 7.2.1.

Data and terminology

To unify the algorithms used for evaluation, we will first describe data and used terminology.

As metrics input, we use our annotated trajectories as a *ground truth* $\longrightarrow GT$ and *generated trajectories* $\longrightarrow GEN$. Since our approach is designed to overcome incompleteness in the input detections, our annotated data are incomplete because they contain only a subset of all detections belonging to a real trajectory. The annotated trajectories can not be complete because the detections are incomplete and miss detections in multiple frames, as well as we did not annotate all trajectories with all the possible detection because it was too much time-consuming and, in some cases, it was not even humanly possible. Thus, we have slightly altered the used metric.

Area coverage In some evaluation cases, the area coverage is measured, but since the *Traged* trajectories are based on the same detections as *GT*, we will not highlight measuring the area coverage.

Latency During evaluations of trajectories, the latency metrics are measured, but in our evaluation, we give the first detection to the OpenCV of each annotated trajectory to unify tracked objects. Thus, we are not going to measure the latency.

Performance evaluation of object tracking algorithms

The first set of metrics is based on the article[38]. The article describes the basic trajectory metrics and we use three of them. Other metrics from the article are covered by other proposed metrics, or their result are irrelevant to us[5]. For example, as mentioned before, the latency. Unlike the article, we have altered the threshold values and used a more strict value, $threshold = 30\%$ Also, we are evaluating trajectories only in frames with annotated detections.

- True Positive $\longrightarrow TP$: Percentage of *GT* trajectories that are correctly tracked $\longrightarrow$ the *GEN* trajectories has been correctly assigned more than 30% of detections and more than 30% space overlap.
- False Positive $\longrightarrow FP$: Percentage of *GEN* trajectories that do not correspond to *GT* trajectories $\longrightarrow$ *GEN* trajectories has been correctly assigned less than 30% of detections or less than 30% of space overlap.
- FN (False Negative): Percentage of *GT* trajectories not matched *GEN* trajectories $\longrightarrow$ the *GT* trajectories has assigned less than 30% of detections or less than 30% space overlap.

[5]Even though we are not presenting the result of the unused metrics, we have in most cases implemented them.

HOTA - Higher Order Tracking Accuracy

To evaluate trajectories with newer metrics, we have decided to implement $HOTA$ - Higher Order Tracking Accuracy metrics. As mentioned, we could not provide the same type of GT data described in the article. Thus, we must alter a described HOTA[22] algorithms and all its parts needed to evaluate our results. To highlight that the metric was altered, we use the prefix $P-$.

- $DetA$ - Detection Accuracy over whole dataset, counted simply as $\frac{TP}{TP+FP+FN}$ as described in [22] over the whole dataset
- $AssA$ - Association measures how well are detections assigned to a object. Association Accuracy is computed as the average of alignment between two trajectories over all pairs of matching predicted and ground-truth detections in the whole dataset.
- $P-HOTA$ - Higher Order Tracking Accuracy. A single value represents how well the tracker performs overall.

Our internal metrics

To compare the generated trajectories, while using a subset of our original metrics presented in section 7.2.1, we have focused on the two most crucial variants for us.

- Fragmentation $\longrightarrow FRAG$ - when multiple GEN trajectories partially correspond to one GT trajectory (state $One-Many$)
- Swap of ids $\longrightarrow SWAP$ - when one GEN trajectory corresponds to multiple GT trajectories(state $Many-One$, and $Many-Many$).
  - $SWAP-A$: GEN trajectory swap to a different GT trajectory after its end. Generated trajectory wrongly detects the end and is tracking another object.
  - $SWAP-B$: GEN trajectory swap to a different GT trajectory before its end. GEN trajectory mismatch a tracked object.

Fragmentation represents the problem where we are not able to create a continuous trajectory for an object and we are creating multiple short ones. To evaluate a $FRAG$, metric count a number of GT trajectories with assigned more than one $GEN \longrightarrow overAssigned$, where the assigned trajectory is longer than one common detection.

$$FRAG = \frac{overAssigned}{\#GT}$$

Swap represents the problem where trajectory mismatches the object. To evaluate $SWAP$, we must count a number of GEN trajectories with assigned more than one GT trajectory $\longrightarrow overTracked$.

$$SWAP = \frac{overTracked}{\#GEN}$$

The variant $SWAP - A$ measures the problem of the algorithm that doesn't know when to end the trajectory, and variant $SWAP - B$ measures the problem of approaches that do not keep tracking trajectory correctly on one object and swap to another. To evaluate $SWAP - A$, we must count a number of GEN trajectories that are covering multiple GT trajectories $\longrightarrow$ $overflowing$ and for $SWAP - B$, we must count a number of GEN trajectories swapping from one GT trajectory to another before end$\longrightarrow$ $skipping$.

$$SWAP - A = \frac{overflowing}{\#GEN}; SWAP - B = \frac{skipping}{\#GEN}$$

Illustration of fragmentation and swap of trajectories can be seen in the picture 7.10.

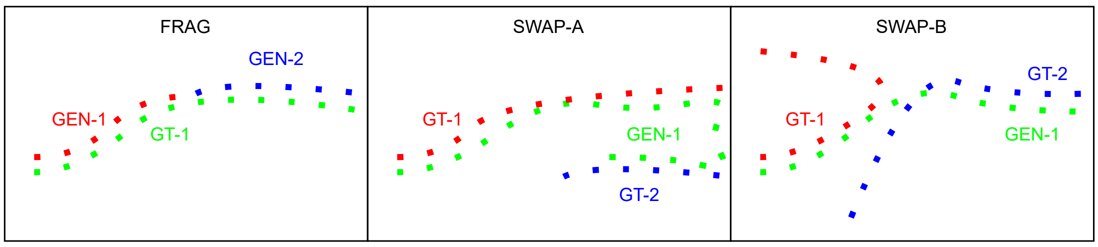


Figure 7.10: Fragmentation and swap of GT

7.3.3 Trajectory evaluation

Finally, in this subsection, we will compare the $Traged$ trajectories against the OpenCV trajectories created by the benchmark[8]. We will use the described metrics on generated trajectories and based on that, make a comparison. Firstly, since the benchmark provides multiple options to generate trajectories, we will choose the best ones and then compare the best OpenCV trajectories against $Traged$ trajectories.

Colored To make tables easily readable, we have marked the best result with a **green colour** and the worst one with a **red colour**.

OpenCv on videolytics

First, we must decide which method from the OpenCV should be used for the comparison. Thus, we ran the benchmark over the subset of the videolytics dataset and for each option generated trajectories. The results of video $duke, id =$ 40 can be seen in the table 7.4 and aggregated result for multiple videos can be seen in the table 7.5. Based on the result, we came up to the same conclusion as the author of the benchmark[8], that overall metrics, the best results provide the $CSRT$. Thus, we have generated trajectories with OpenCV-$CSRT$ for the whole dataset, as can be seen in the table 7.6.

KCF As can be seen in the tables, the KCF method received the best results over our $FRAG$ and $SWAP$ metric $\longrightarrow$ measuring that trajectory is in one piece and not jumping between objects, but in a same time the worst result over TP and FP $\longrightarrow$ how good were objects tracked over all. Thus, the explanation is simple. Since KCF trajectories are not relevant and hardly exist, they are not making problems with $FRAG$ and $SWAP$.

Method / Metric	`csrt`	`boosting`	`kcf`	`tld`	`mosse`	`medianflow`	`mil`
TP	**0.938**	0.75	0.25	0.375	**0.062**	0.562	0.75
FP	**0.25**	0.312	0.75	0.812	**0.938**	0.562	0.5
TDF	**0.125**	0.25	0.75	0.562	**0.938**	0.438	0.25
FRAG	**0.0**	0.312	**0.0**	0.5	0.562	**0.625**	0.188
SWAP	0.125	0.562	**0.0**	0.438	0.375	**0.625**	0.312
SWAP-B	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SWAP-A	0.125	0.562	**0.0**	0.438	0.375	**0.625**	0.312
P-AssA	**0.814**	0.63	0.216	0.343	**0.165**	0.464	0.681
P-DetA	0.589	0.507	**0.105**	0.273	0.114	0.407	**0.609**
P-HOTA	**0.692**	0.565	0.151	0.307	**0.137**	0.435	0.644

Table 7.4: Evaluation of all OpenCV methods on single video - duke

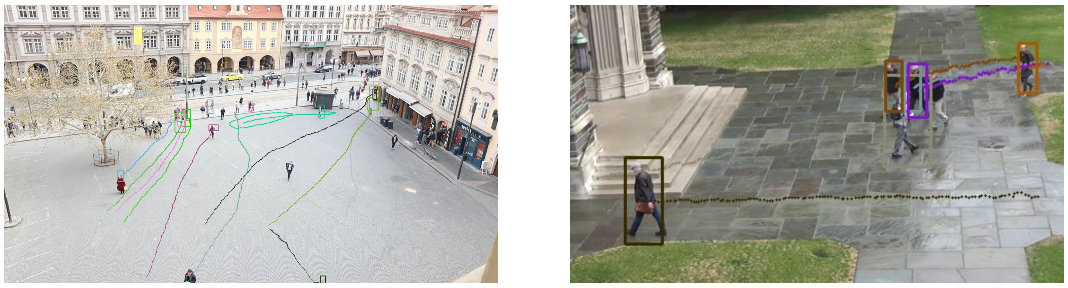
Method / Metric	csrt	boosting	kcf	tld	mosse	medianflow	mil
TP	**0.955**	0.716	**0.107**	0.328	0.348	0.345	0.771
FP	**0.116**	0.234	**0.893**	0.791	0.735	0.714	0.33
TDF	**0.098**	0.293	**0.893**	0.642	0.652	0.795	0.246
FRAG	0.048	0.202	**0.024**	0.284	0.303	**0.35**	0.237
SWAP	0.098	0.341	**0.024**	0.383	0.29	**0.424**	0.278
SWAP-B	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SWAP-A	0.098	0.341	**0.024**	0.383	0.29	**0.424**	0.278
P-AssA	**0.824**	0.672	**0.15**	0.288	0.342	0.305	0.622
P-DetA	**0.746**	0.46	**0.102**	0.188	0.207	0.228	0.536
P-HOTA	**0.782**	0.555	**0.119**	0.231	0.265	0.263	0.576

Table 7.5: Average results of evaluation of all OpenCV methods on multiple videos

ID / Metric	43	48	40	928	921	920	922	469	470
TP	1.0	0.714	0.938	1.0	0.875	0.824	1.0	0.857	0.929
FP	0.026	0.333	0.25	0.077	0.188	0.235	0.0	0.143	0.071
TDF	0.026	0.286	0.125	0.0	0.125	0.235	0.0	0.286	0.143
FRAG	0.0	0.0	0.0	0.077	0.188	0.176	0.0	0.0	0.143
SWAP	0.026	0.048	0.125	0.154	0.125	0.235	0.0	0.0	0.143
SWAP-B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SWAP-A	0.026	0.048	0.125	0.154	0.125	0.235	0.0	0.0	0.143
P-AssA	0.866	0.912	0.814	0.797	0.727	0.597	0.842	0.846	0.793
P-DetA	0.819	0.778	0.589	0.745	0.64	0.49	0.603	0.737	0.831
P-HOTA	0.842	0.842	0.692	0.771	0.682	0.541	0.713	0.79	0.811

Table 7.6: OpenCV - CSRT over the whole dataset

Vizualization To investigate the trajectories generated by the benchmark we have drawn them directly into videos. For illustration, the visualization can be seen in the pictures 7.11



(a) wide-view recording (b) close-view recording

Figure 7.11: Trajectories generated by OpenCV-CSRT

OpenCV - CSRT versus TRAGED

To create a comparison of *OpenCV-CSRT* and *Traged*, we have generated trajectories for each video in the dataset with both $\longrightarrow$ *OpenCV-CSRT* and *Traged*. For *Traged*, we used configuration for a close-view recording with $id = 2198$ and for a wide-view recording with $id = 2259$.

Results can be seen for a wide-view recording in the table 7.7, for a close-view recording in the table 7.8 and aggregated results for all recordings for both types of view in table 7.11. As can be seen in the tables, the results reached are not very different. To evaluate the differences, we will take a look at the metrics and explain what they are telling us about generated trajectories.

	001_*Cannon*		002_*Cannon*		*kradez_pravo*		*rvacka_pravo*		*tram_2*	
METHODS	2259	csrt	2259	csrt	2259	csrt	2259	csrt	2259	csrt
TP	**1.0**	**1.0**	**1.0**	**0.714**	**1.0**	**1.0**	**1.0**	**0.824**	**1.0**	**0.929**
FP	**0.189**	**0.026**	**0.16**	**0.333**	**0.188**	**0.077**	**0.4**	**0.235**	**0.263**	**0.071**
TDF	**0.0**	**0.026**	**0.0**	**0.286**	**0.0**	**0.0**	**0.0**	**0.235**	**0.0**	**0.143**
FRAG	**0.289**	**0.0**	**0.095**	**0.0**	**0.231**	**0.077**	**0.353**	**0.176**	**0.286**	**0.143**
SWAP	**0.075**	**0.026**	**0.04**	**0.048**	**0.062**	**0.154**	**0.12**	**0.235**	**0.0**	**0.143**
SWAP-B	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
SWAP-A	**0.075**	**0.026**	**0.04**	**0.048**	**0.062**	**0.154**	**0.12**	**0.235**	**0.0**	**0.143**
P-AssA	**0.846**	**0.866**	**0.962**	**0.912**	**0.929**	**0.797**	**0.854**	**0.597**	**0.948**	**0.793**
P-DetA	**0.994**	**0.819**	**0.999**	**0.778**	**1.0**	**0.745**	**1.0**	**0.49**	**1.0**	**0.831**
P-HOTA	**0.917**	**0.842**	**0.98**	**0.842**	**0.964**	**0.771**	**0.924**	**0.541**	**0.973**	**0.811**

Table 7.7: Evaluation *Traged* X OpenCV - wide-view

	duke		*kradez_stred*		*rvacka_stred*	
METHODS	2198	csrt	2198	csrt	2198	csrt
TP	**1.0**	**0.938**	**0.812**	**.875**	**1.0**	**1.0**
FP	**0.0**	**0.25**	**0.667**	**0.188**	**0.48**	**0.0**
TDF	**0.0**	**0.125**	**0.188**	**0.125**	**0.0**	**0.0**
FRAG	**0.125**	**0.0**	**0.5**	**0.188**	**0.4**	**0.0**
SWAP	**0.188**	**0.125**	**0.167**	**0.125**	**0.04**	**0.0**
SWAP-B	0.0	0.0	0.0	0.0	0.0	0.0
SWAP-A	**0.188**	**0.125**	**0.167**	**0.125**	**0.04**	**0.0**
P-AssA	**0.944**	**0.814**	**0.659**	**0.727**	**0.842**	**0.842**
P-DetA	**0.994**	**0.589**	**0.962**	**0.64**	**0.918**	**0.603**
P-HOTA	**0.968**	**0.692**	**0.796**	**0.682**	**0.879**	**0.713**

Table 7.8: Evaluation *Traged* X OpenCV - close-view

Metric \ Method	2198	csrt
TP	**0.938**	**0.938**
FP	**0.382**	**0.146**
TDF	**0.062**	**0.083**
FRAG	**0.342**	**0.062**
SWAP	**0.131**	**0.083**
SWAP-B	0.0	0.0
SWAP-A	**0.131**	**0.083**
P-AssA	**0.815**	**0.794**
P-DetA	**0.958**	**0.611**
P-HOTA	**0.881**	**0.696**

Table 7.9: close-view video

Metric \ Method	2259	csrt
TP	**1.0**	**0.893**
FP	**0.24**	**0.149**
TDF	**0.0**	**0.138**
FRAG	**0.251**	**0.079**
SWAP	**0.06**	**0.121**
SWAP-B	0.0	0.0
SWAP-A	**0.06**	**0.121**
P-AssA	**0.908**	**0.793**
P-DetA	**0.999**	**0.733**
P-HOTA	**0.952**	**0.762**

Table 7.10: wide-view video

Table 7.11: Aggregated comparison *Traged* X OpenCV-CSRT

TP,FP,TDF Metrics TP and TDF, tell us how good were objects tracked into trajectories - how good is GT covered by GEN and FP a occurences of GEN with no corresponding object $\longrightarrow$ exist generated trajectory for not existing object. As can be seen *Traged* is better at covering the object's trajectories, but it seems, that sometimes generates trajectory for not existing object. After the data investigation, we have found out, that this is caused by the incompleteness of annotated trajectories. Due to the complexity, we did not annotat all objects with trajectories in the videos and covered only a subset - the problematic one and trajectories, where human annotator was sure of the correctness[6]. For example, we have succesfully annotated all the trajectories for the video duke and as can be seen in the table 7.7, *Traged* had no problems with FP.

FRAG and SWAP The metrics - $FRAG$ representing GT split on multiple GEN and $SWAP$ representing GT merged into one by GEN as defined in section 7.3.2. First of all, on the measured data can be seen that, we had no occurrence of $SWAP-B$, which signalizes, that approaches were not mixing up different objects, while were present in the screen. On the other hand, *Traged* was in all videos worse on the $FRAG$ metric, which is telling us that the configuration should allow more merging by lowering some boundaries. After data investigation, we came to the conclusion that $FRAG$ and $SWAP-A$ is caused for *Traged* in edge situations, where objects pass each othere near the edge of the videos.

HOTA Finally, as can be seen the *Traged* reached better results on all $HOTA$ metrics, telling us that it seems that *Traged* has better detection accuracy and mainly that has better alignment between GT and GEN.

Disclaimer After exploring the results, it seems that the *Traged* is better and provides in some aspects a better results, but we have to keep in mind that *Traged* is highly specialized on the specific types of videos, where on the other hand, OpenCV-CSRT is universal. To show the *Traged* problem, we have generated variants with swapped configurations, as can be seen in the table 7.14 and is obvious that even if the camera views (close-view and wide-view) are quite similar, the results are drastically worse.

[6]In some cases, the objects were moving in groups too far from the camera and it was not humanly distinguishable.

002_*Cannon_FHD*

Metric \ Method	2198	2259	csrt
TP	**1.0**	**1.0**	**0.714**
FP	**0.697**	**0.16**	0.333
TDF	**0.0**	**0.0**	**0.286**
FRAG	**0.381**	0.095	**0.0**
SWAP	**0.061**	**0.04**	0.048
SWAP-B	0.0	0.0	0.0
SWAP-A	**0.061**	**0.04**	0.048
P-AssA	**0.868**	**0.962**	0.912
P-DetA	0.997	**0.999**	**0.778**
P-HOTA	0.93	**0.98**	**0.842**

Table 7.12: 48 - wide-view recording

duke

Metric \ Method	2259	2198	csrt
TP	**1.0**	**1.0**	**0.938**
FP	0.062	**0.0**	**0.25**
TDF	**0.0**	**0.0**	**0.125**
FRAG	**0.25**	0.125	**0.0**
SWAP	**0.312**	0.188	**0.125**
SWAP-B	0.0	0.0	0.0
SWAP-A	**0.312**	0.188	**0.125**
P-AssA	0.885	**0.944**	**0.814**
P-DetA	**1.0**	0.994	**0.589**
P-HOTA	0.941	**0.968**	**0.692**

Table 7.13: 40 - close-view recording

Table 7.14: Results with swapped configuration

MOTChallenge in future

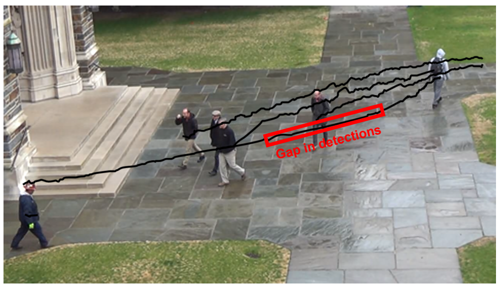
As mentioned in the introduction of this section, in the current state of the *Traged* and Videolytics we are not able to join the MOTChallenge and be able to be a balanced opponent on a given dataset. Still, we would like to, in the future, try it. Thus, the presented evaluation was run on the same data format as is used in MOTChallenge. Also, we have prepared modules to download our data in the MOTChallenge format and once the Videolytics transfer to a bigger server, we would like to have the option to use MOTChallenge dataset in Videoliytics.

7.4 Interdetections and semantic description

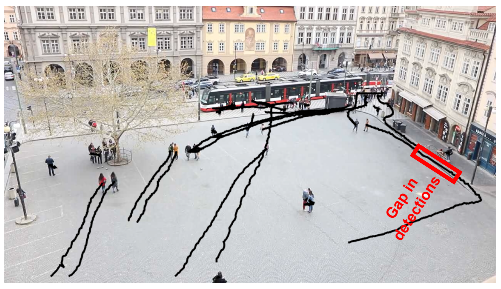
This section will present examples of generated interdetections and semantic descriptions for trajectories. As mentioned, the results are directly affected by the quality of the trajectories. Thus, we are not evaluating the quality of generated data by any metric and we will only observe the generated data.

7.4.1 Interdetection

To observe the generated interdetections in valuable situations, we have watched the videos and identified the trajectories with a large gaps of detections, as shown in the pictures 7.12. The screenshots are taken directly from the Videolytics web for videos *duke*, $model = 2198$ and 001_*Cannon_FHD*, $model = 2259$.



(a) 40 - *duke*



(b) 43 - 001_*Cannon_FHD*

Figure 7.12: Trajectories with large detections gaps

Cases description

To illustrate the results and added data value of interdetections, we have generated two images, shown in the picture 7.13, with inserted:

- detections - all detections for a trajectory (left side)
- interdetections - all interdetections for a trajectory (right side)

As can be seen on the left side of the image 7.13, a large continuous gap in the trajectory detections is present, approximately 4 seconds $\approx$ 100 frames. This gap is caused by detector error[7] and the object is undetected in these frames.

On the right side of the image 7.13, are displayed all interdetections of the object. As can be seen, the interdetections are along the whole trajectory because the detector undetected the object, but it is in a single-frame amount. On the other hand, in the highlighted area are interdetections for each frame.

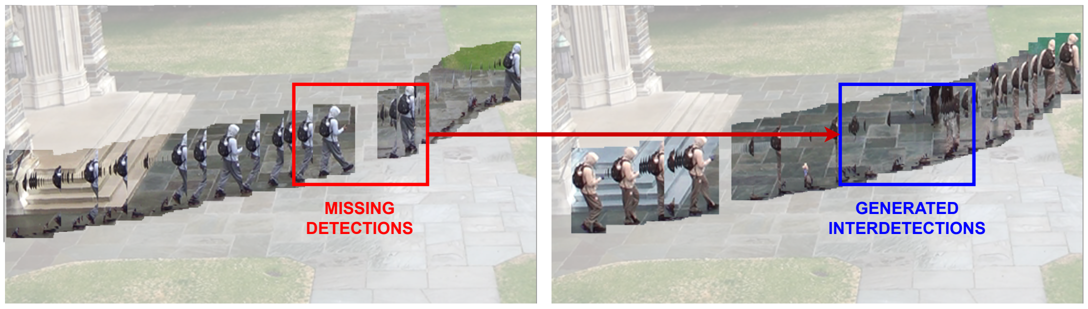


Figure 7.13: Generated interdetections for *duke*

The same results can be seen in different videos, with different camera positions, as shown in the picture 7.14.

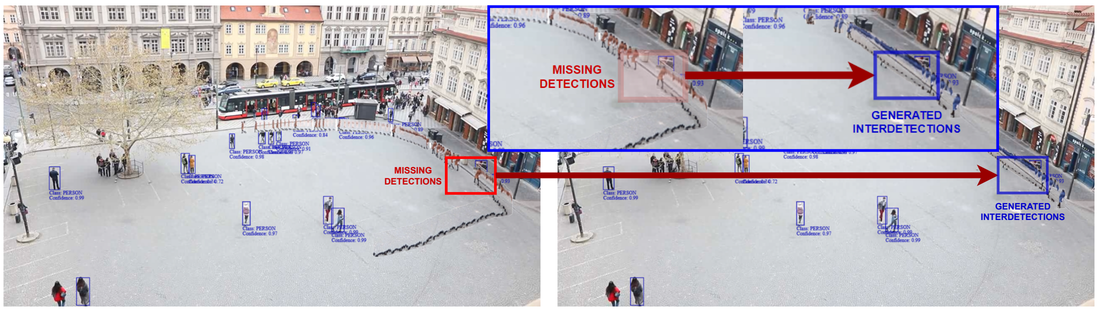


Figure 7.14: Generated interdetections for 001_*Cannon*_*FHD*

7.4.2 Semantic description

To evaluate the implemented algorithm for semantic description, we have generated descriptions for a few videos and then chose a few trajectories to demonstrate

[7]We have verified that the trajectory contains all possible detections.

the results. In the current state, descriptions are not used for searching yet. Thus, results are evaluated by observing and case studying.

Examples of descriptions

To present the generated descriptions of trajectories, we have chosen two trajectories with not-simple progress because most of the trajectories are the object's moving in straight lines.

In the presented images, trajectories are described only by their direction[8] within the trajectory. Also directions are merged $[left, sharpLeft] \longrightarrow left$, $[right, sharpRight] \longrightarrow right$ for simplicity.

The descriptors are implemented as designed in section 5.2, and individual parts can be seen in the pictures 7.15 and 7.17. Each trajectory is split into multiple sub-trajectories, where, in the presented illustration is, each sub-trajectory marked by its order number. Then, for each sub-trajectory, is measured speed and angle and, based on the context of the whole trajectory, the sub-trajectory is classified into categories of speed and direction. The direct output of the program is presented in the pictures 7.16 7.18.

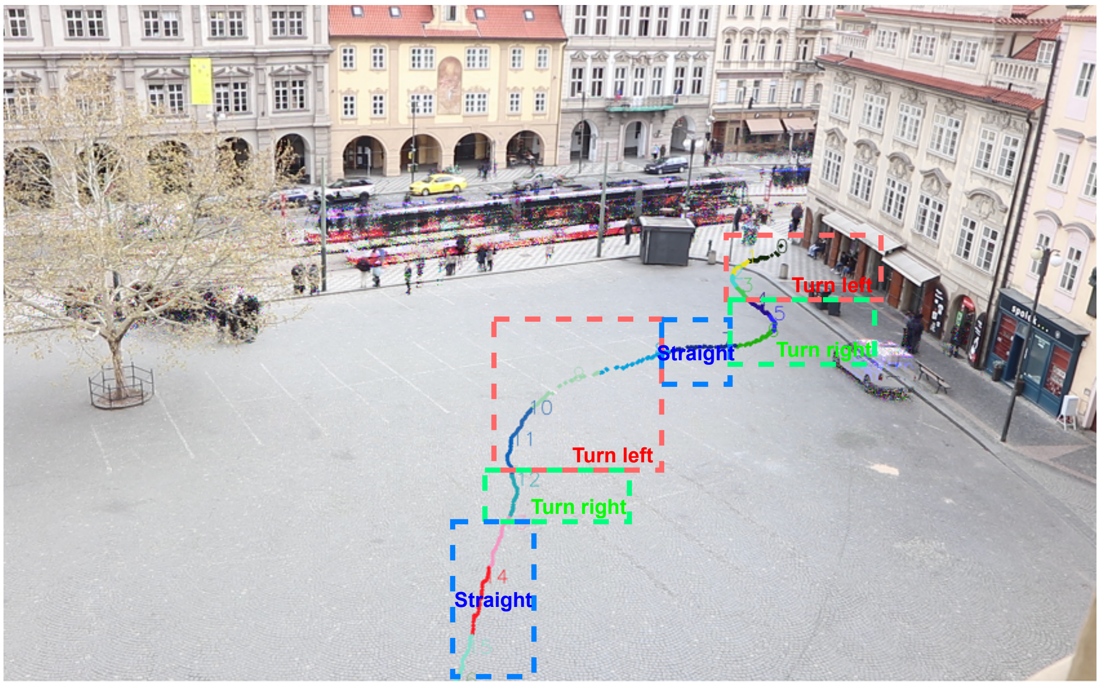
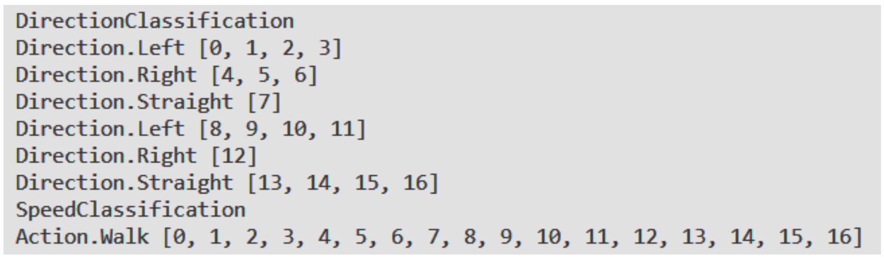


Figure 7.15: Semantic description of trajectory A

[8]We generated speed as well, but all of the presented objects were just walking.



```
DirectionClassification
Direction.Left [0, 1, 2, 3]
Direction.Right [4, 5, 6]
Direction.Straight [7]
Direction.Left [8, 9, 10, 11]
Direction.Right [12]
Direction.Straight [13, 14, 15, 16]
SpeedClassification
Action.Walk [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
```

Figure 7.16: Semantic description of trajectory A

In the picture 7.15, we can see an illustration of generated description for the trajectory. This description enriches the trajectory by its semantic value. It tells us that the object was walking $[left, right, straight, left, right, straight]$, which can be used in future steps for object behaviour analysis.

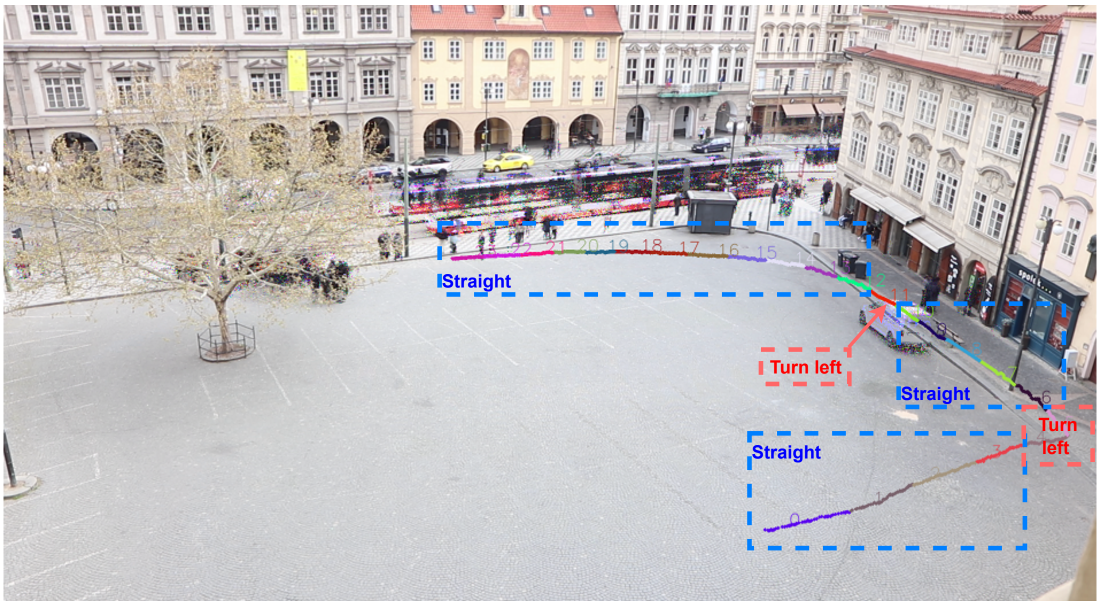
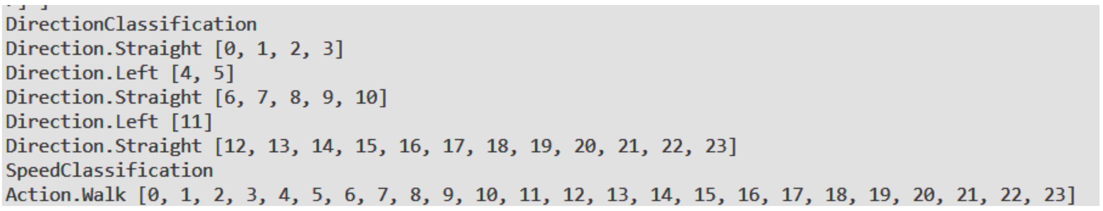


Figure 7.17: Semantic description of trajectory B

In the picture 7.17, we can see an illustration of generated description for another trajectory.



```
DirectionClassification
Direction.Straight [0, 1, 2, 3]
Direction.Left [4, 5]
Direction.Straight [6, 7, 8, 9, 10]
Direction.Left [11]
Direction.Straight [12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
SpeedClassification
Action.Walk [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]
```

Figure 7.18: Semantic description of trajectory B

As shown, the proposed description enriches the trajectories with abstract information about its movement as was planned.

7.5 Summarization

In this chapter, we have evaluated the proposed algorithms from previous chapters. First of all, we have presented the Videolytics dataset and described video recordings in it.

While using Videolytics data, we created a configuration for our proposed flow of algorithms $\longrightarrow$ *Traged* and, based on the configuration, generated trajectories for videos in Videolytics. To evaluate the trajectories' quality, we proposed a few simple metrics, and later, we used well-known metrics to create a comparison with OpenCV. We showed that with the correct configuration, on a restricted area, *Traged* is usable for trajectory generating and, at the very least, provides as good results as OpenCV.

In the end, we evaluated the trajectory usage by presenting the results of interdetections and trajectory semantic description of objects. We have presented generated interdetections, showing the quality improvement of detections and enriching the detections with new detections of objects. With the presented semantic description, we have laid the groundwork for further research on object behaviour analysis and trajectory search.

Based on the evaluation, the proposed pipeline works and is a good fit in the Videolytics system as a new module. As for the algorithms themselves, it means that they are working as expected.

Conclusion

In this thesis, we have designed a complex flow for video detection processing while using analytic methods.

As a first step, we have proposed algorithms for detection clustering into trajectories based on connectivity and user configuration in chapter 3. To measure the connectivity of detections/trajectories, we have proposed a set of algorithms, in chapter 4 extracting features and defining how possible it is that two detections/trajectories represent the same object.

To improve the detections, we have proposed an algorithm interpolating generated trajectories to fill errors in detections by interdetections in section 5.1. Finally, to enable trajectory search and future analysis work, we have designed an algorithm describing trajectories with semantic description in section 5.2.

To prove the validity of the proposed algorithms, we have implemented a new module, in chapter 6, with the implementation of all proposed algorithms. With implementation, we have generated trajectories, interdetections and semantic descriptions for Videolytics videos, created visual material for result observations and evaluated the approach, for trajectory generation, against the OpenCV in chapter 7. By measured results, we have concluded that the proposed approach can generate great trajectories on the restricted video domains based on the provided configuration.

In the thesis, we have fulfilled our goals and verified that the proposed analytical model can generate complex trajectories based only on the provided configuration file and that the trajectories are a valuable source of data and a great foundation for further work in videos analysis field.

7.6 Future work

We have reached multiple problematic areas with potential for future research or Videolytics improvement and further system extensions during the work. To present at least some of them and propose a solution, we would suggest to:

Configuration and challenges In the current state of Videolitycs, limited types of videos restrict the possible testing and configuration development. Thus in the future would be useful to extend the dataset by new video types or use a different source of detections for configuration development.

Once the set of configurations covers a large pallet of video types, comparing generated trajectories in any well-known challenge or benchmark will be desirable.

Auto configuration Currently, the user-programmer has to choose the configuration for the processed video. But from the observations, as mentioned in section 7.2, the configuration is selected based on the camera view. Thus, in future steps, this configuration selection could be automatized and selected directly by the module based on the camera and detections.

Self evolving configuration As described in section 7.2, in the current state are configuration files created by the user-programmer. Thus, in the future, work would be useful to train a model or prepare any automatic way to create a new configuration file for new types of videos.

Detection silhouettes In the current state of the videolytics, a bounding box containing a background is used for detections. During the connectivity measurement, we found that it would be useful to have cropped objects without a background. Thus, as a part of future work would be great to extend detections with image variant without background or update a current one and remove it.

Video to perspective With the capability to identify objects throughout the whole image, we can use this knowledge to identify the image distortion and transform a video into a perspective view. While using the perspective view, we can better understand the object movement and use the knowledge to determine the mutual position of multiple overlaying videos.

Object behaviour Based on the generated trajectories and their semantic description, there is a great opportunity to focus on object behaviour and analyse the most crucial types of movement. Furthermore, based on trajectories, there is potential to detect suspicious behavior such as stealing, stalking or street fights.

Bibliography

[1] Anas Al-Oraiqat and Natalya Kostyukova. A modified image comparison algorithm using histogram features. *International Journal of advanced studies in Computer Science and Engineering*, 7:8, 03 2018.

[2] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. Visual tracking with online multiple instance learning. In *2009 IEEE Conference on computer vision and Pattern Recognition*, pages 983–990. IEEE, 2009.

[3] David S Bolme, J Ross Beveridge, Bruce A Draper, and Yui Man Lui. Visual object tracking using adaptive correlation filters. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 2544–2550. IEEE, 2010.

[4] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Adnan Brdjanin, Amila Akagic, Džemil Džigal, and Nadja Dardagan. Single object trackers in opencv: A benchmark. 08 2020. doi: 10.1109/INISTA49547.2020.9194647.

[6] Rishav Chakravarti and Xiannong Meng. A study of color histogram based image retrieval. pages 1323 – 1328, 05 2009. doi: 10.1109/ITNG.2009.126.

[7] Hsu-kuang Chiu, Jie Li, Rareş Ambruş, and Jeannette Bohg. Probabilistic 3d multi-modal, multi-object tracking for autonomous driving. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14227–14233. IEEE, 2021.

[8] Nadja Dardagan, Adnan Brdjanin, Džemil Džigal, and Amila Akagic. Multiple object trackers in opencv: A benchmark, 10 2021.

[9] Marek Dobranský and Tomáš Skopal. On fusion of learned and designed features for video data analytics. *International Conference on Multimedia Modeling (MMM)*, pages 268–280, 01 2021. doi: 10.1007/978-3-030-67835-7_23.

[10] Mauro Fernandez-Sanjurjo, Brais Bosquet, Manuel Mucientes, and Victor M Brea. Real-time visual detection and tracking system for traffic monitoring. *Engineering Applications of Artificial Intelligence*, 85:410–420, 2019.

[11] R Ganapathyraja and SP Balamurugan. Suspicious loitering detection using a contour-based object tracking and image moment for intelligent video surveillance system. *JOURNAL OF ALGEBRAIC STATISTICS*, 13(2): 1294–1303, 2022.

[12] Helmut Grabner, Michael Grabner, and Horst Bischof. Real-time tracking via on-line boosting. In *Bmvc*, volume 1, page 6. Citeseer, 2006.

[13] Andrea Hornakova, Roberto Henschel, Bodo Rosenhahn, and Paul Swoboda. Lifted disjoint paths with application in multiple object tracking. In *International conference on machine learning*, pages 4364–4375. PMLR, 2020.

[14] Peter Janku, Karel Koplik, Tomáš Dulík, and Istvan Szabo. Comparison of tracking algorithms implemented in opencv. *MATEC Web of Conferences*, 76:04031, 01 2016. doi: 10.1051/matecconf/20167604031.

[15] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Forward-backward error: Automatic detection of tracking failures. In *2010 20th international conference on pattern recognition*, pages 2756–2759. IEEE, 2010.

[16] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1409–1422, 2011.

[17] UM Kamthe and CG Patil. Suspicious activity recognition in video surveillance system. In *2018 Fourth international conference on computing communication control and automation (ICCUBEA)*, pages 1–6. IEEE, 2018.

[18] L. Leal-Taixé, A. Milan, I. Reid, S. Roth, and K. Schindler. MOTChallenge 2015: Towards a benchmark for multi-target tracking. *arXiv:1504.01942 [cs]*, April 2015. URL `http://arxiv.org/abs/1504.01942`. arXiv: 1504.01942.

[19] S. Lee, John Xin, and Stephen Westland. Evaluation of image similarity by histogram intersection. *Color Research and Application*, 30:265 – 274, 08 2005. doi: 10.1002/col.20122.

[20] Peiliang Li, Tong Qin, et al. Stereo vision-based semantic 3d object and ego-motion tracking for autonomous driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 646–661, 2018.

[21] Wenhe Liu, Guoliang Kang, Po-Yao Huang, Xiaojun Chang, Yijun Qian, Junwei Liang, Liangke Gui, Jing Wen, and Peng Chen. Argus: Efficient activity detection system for extended video analysis. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision Workshops*, pages 126–133, 2020.

[22] Jonathon Luiten, Aljosa Osep, Patrick Dendorfer, Philip Torr, Andreas Geiger, Laura Leal-Taixé, and Bastian Leibe. Hota: A higher order metric for evaluating multi-object tracking. *International Journal of Computer Vision*, pages 1–31, 2020.

[23] Alan Lukezic, Tomas Vojir, Luka Cehovin Zajc, Jiri Matas, and Matej Kristan. Discriminative correlation filter with channel and spatial reliability. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6309–6318, 2017.

[24] Chenxu Luo, Xiaodong Yang, and Alan Yuille. Exploring simple 3d multi-object tracking for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10488–10497, 2021.

[25] H MEDDEBER and B YAGOUBI. Parallel object tracking in image sequences based on k-means and an improved gradient vector flow. *Journal of Engineering Science and Technology*, 16(4):3119–3135, 2021.

[26] Tim Meinhardt, Alexander Kirillov, Laura Leal-Taixe, and Christoph Feichtenhofer. Trackformer: Multi-object tracking with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8844–8854, 2022.

[27] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler. MOT16: A benchmark for multi-object tracking. *arXiv:1603.00831 [cs]*, March 2016. URL `http://arxiv.org/abs/1603.00831`. arXiv: 1603.00831.

[28] Utkarsha Mokashi, Aarush Dimri, Hardee Khambhla, and Pradnya Bhangale. Review: Video analytics technologies available for surveillance systems. In *2022 5th International Conference on Advances in Science and Technology (ICAST)*, pages 466–470, 2022. doi: 10.1109/ICAST55766.2022.10039583.

[29] Iyiola Olatunji and Chun Hung Cheng. *Video Analytics for Visual Surveillance and Applications: An Overview and Survey*, pages 475–515. 07 2019. ISBN 978-3-030-15627-5. doi: 10.1007/978-3-030-15628-2_15.

[30] G. Pass and R. Zabih. Histogram refinement for content-based image retrieval. In *Proceedings Third IEEE Workshop on Applications of Computer Vision. WACV'96*, pages 96–102, 1996. doi: 10.1109/ACV.1996.572008.

[31] Bing Shuai, Andrew Berneshawi, Xinyu Li, Davide Modolo, and Joseph Tighe. Siammot: Siamese multi-object tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12372–12382, 2021.

[32] Tomáš Skopal, Dominika Ďurišková, Petr Pechman, Marek Dobranský, and Vladislav Khachaturian. Videolytics- system for data analytics of video streams. *International Conference on Information and Knowledge Management (CIKM 2021)*, pages 4794–4798, 10 2021. doi: 10.1145/3459637.3481980.

[33] Peize Sun, Jinkun Cao, Yi Jiang, Rufeng Zhang, Enze Xie, Zehuan Yuan, Changhu Wang, and Ping Luo. Transtrack: Multiple object tracking with transformer. *arXiv preprint arXiv:2012.15460*, 2020.

[34] ShiJie Sun, Naveed Akhtar, HuanSheng Song, Ajmal Mian, and Mubarak Shah. Deep affinity network for multiple object tracking. *IEEE transactions on pattern analysis and machine intelligence*, 43(1):104–119, 2019.

[35] System for data analytics of video streams. Videolytics. `http://videolytics.ms.mff.cuni.cz/stream.html`. Accessed: 2023-04-5.

[36] Bin Tian, Qingming Yao, Yuan Gu, Kunfeng Wang, and Ye Li. Video processing techniques for traffic flow monitoring: A survey. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 1103–1108. IEEE, 2011.

[37] Greg Welch, Gary Bishop, et al. An introduction to the kalman filter. 1995.

[38] Fei Yin, Dimitrios Makris, and Sergio A Velastin. Performance evaluation of object tracking algorithms. In *IEEE International Workshop on Performance Evaluation of Tracking and Surveillance, Rio De Janeiro, Brazil*, volume 25. Citeseer, 2007.

List of Figures

List of Tables

List of Abbreviations

- GEN - Generated trajectory
- GT - Ground truth trajectory
- $fconnectivity$ - $feature$-$connectivity$
- API - Application Programming Interface
- $TRAGED$ - TRAjectories GEnerated from Detections
- JSON - JavaScript Object Notation

A. Attachments

To this thesis are attached source codes and examples of input data.

A.1 Traged

The main attachment to this thesis is folder `Traged.zip`, representing a module described in this thesis and implemented in the Videolytics system[1]. The module is also accessible on university GitLab `https://gitlab.mff.cuni.cz/hrbacem/construction-of-time-space-trajectories-from-multimodal-data`.
The content is almost identical, whereas GitLab provides extra video materials.

Content The content is described directly in `README.md` and in this thesis in the section 6.4. In the sections below are pointed out the subfolders and their purpose.

A.1.1 DOCUMENTATION

In the folder, `DOCUMENTATION` is the documentation for the whole module.

- `full_documentation.pdf`
- `user_documentation.pdf`
- `technical_documentation.pdf`
- `thesis.pdf`

and other files, for example, the `CookBook.md` with information on how to create docker for the Traged module.

A.1.2 SQL

In the folder, `SQL` are contained `*.sql` files to create newly designed tables in the thesis.

A.1.3 Evaluation

In the folder, `Evaluation` are codes for evaluation, measured data in `MEASURED\_DATA` and the benchmark used for the evaluation.

A.1.4 TESTS

In the folder, `TESTS` are unit tests for comparators.

[1]The module is also part of the Videolytics code base and can be accessed through the official project.

A.1.5 configuration_files

In the folder, `configuration_files` are JSON files with used configurations.

A.1.6 The rest of data

The rest of the folders are data, code and scripts for Traged, interdetections and semantics description as described in the thesis and the documentation. The code is commented, and if needed, the sub-folder contains `README.md` file with information.