muSSP: Efficient Min-cost Flow Algorithm for Multi-object Tracking

Teja Dhondu
International Institute of Information Technology, Hyderabad
Gachibowli, Telangana
teja.dhondu@students.iiit.ac.in

Swetanjal Murati Dutta
International Institute of Information Technology, Hyderabad
Gachibowli, Telangana
swetanjal.dutta@research.iiit.ac.in

Nishanth Sharma
International Institute of Information Technology, Hyderabad
Gachibowli, Telangana
saketh.venkat@students.iiit.ac.in

Abstract

This is a report for Project Evaluation 2 of the Computer Vision Course taught by Dr. Avinash Sharma during Spring Semester of 2020 at IIIT-H. The goal of this evaluation is to assess the work done till 10th of March for the project component of the course. The deliverable for the evaluation includes this report, a presentation (to be presented in front of Dr. Avinash and our assigned TA Mr. Kunal Garg) and implementation code.

1. Introduction

Our project is to implement and extend the paper "muSSP: Efficient Min-cost Flow Algorithm for Multi-object Tracking" by Wang et al. from the proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada. Min-cost flow has been a widely used paradigm for solving data association problems in MOT. However, most existing methods of solving min-cost flow problems in MOT are either direct adoption or slight modifications of generic min-cost flow algorithms, yielding sub-optimal computation efficiency and holding the applications back from larger scale of problems. This paper develops an algorithm namely Minimum update Successive Shortest Path, to solve the problem of Multi Object Tracking(MOT) very efficiently by exploiting the special structures and properties of the graphs formulated in MOT.

This paper is a direct extension of the paper "Globally-Optimal Greedy Algorithms for Tracking a Variable Number of Objects" by Pirsiavash et al. which introduces the idea of solving MOT as a problem of solving for Successive Shortest Paths(SSP) in a graph which we are about to explain in the subsequent section.

For this evaluation, we have implemented only the Successive Shortest Path Algorithm. For the remaining half of the semester we update our code for muSSP which is nothing but a few additions from the currently implemented SSP algorithm based on the properties presented in this paper.

The muSSP algorithm improves the efficiency by 5 to 337 folds relative to the best competing algorithm and averagely 109 to 4089 folds to each of the three peer methods.

The report follows with a section on model where we discuss how the problem of MOT can be designed as a graph problem, following which we describe the SSP Algorithm we implemented. We end by discussing about the different networks we have tried to determine the edge costs. The conclusion explains what we implement for the remaining half of the semester to make this project a success.

2. Model

The problem of associating detections from all frames (data association) can be formulated as a unit capacity min-cost flow problem on a DAG as in Fig.1. In this paper, the term "object" represents a physical object existing over time (e.g. a person), and "detection" indicates a detected snapshot of an object at some time point. An object corresponds to a series (trajectory) of detections.

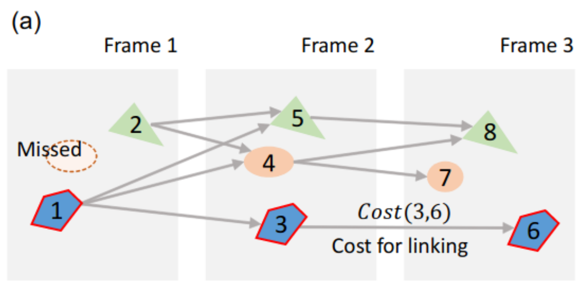


Figure 1. Objects detected in three frames. The first frame has two detections and misses one.Lines between detections are possible ways of linking them. Each line is associated with a cost. Detections 1, 3 and 6 should be linked together as a single trajectory

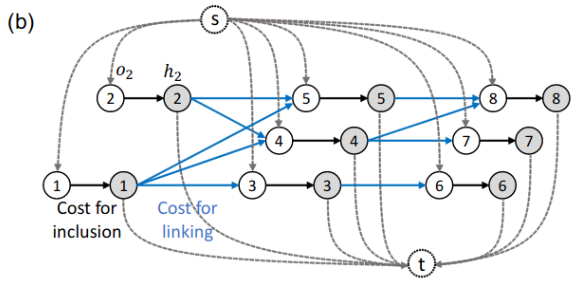


Figure 2. Typical min-cost flow model for MOT problem. Detection i is represented by a pair of nodes: a pre-node o_i and a post-node h_i. The source node s is linked to all pre-nodes and all post-nodes are linked to the sink node t. These edges are shown in dashed lines. Edges between detections are shown in blue.

We denote the graph built in min-cost flow formulation of MOT as $G(V, E, C)$ with node set V, arc set E, and real-valued arc cost function C. The graph has one source s and one sink t. For each detection i, we create a pre-node o_i and a post-node h_i, and three arcs, (o_i,h_i), (s,o_i) and (h_i,t). Any possible spatio temporal transition of an object is corresponding to some pair of detections, i and j (i before j in time), and an arc(h_i,o_j)is created for it. The capacity of any arc is 1. Any directed path between node u and node v on graph G is denoted as $\pi_G(u, v)$. Under such construction, we can interpret each $s - t$ path $\pi_G(s, t)$ as an object trajectory candidate, linking a sequence of detections.

The problem is then turned into selecting a set of object trajectories from all the candidates. In the min-cost flow formulation, this is done by sending flow from s to t, and the $s - t$ paths eventually with flow inside are selected. Due to unit capacities and the total uni-modularity of the problem, it can be formulated as an Integer Programming problem as shown in Fig.2.

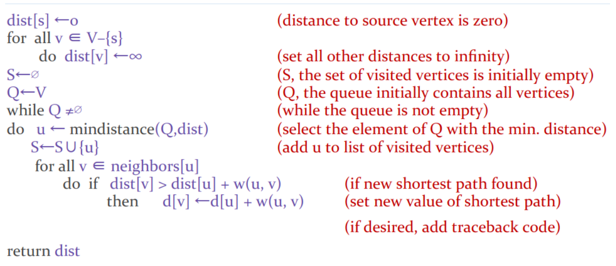
The selection of $s-t$ paths is guided by the design of arc cost C. MOT looks for object trajectories with stronger evi-

$$\min_f \sum_{(u,v)\in E} C(u,v) f_{uv}$$

$$\text{s.t.} \quad f_{uv} \in \{0,1\}, \text{ for all } (u,v) \in E$$

$$\text{and} \quad \sum_{v:(v,u)\in E} f_{vu} = \sum_{v:(u,v)\in E} f_{uv}, \text{ for all } u \in V \setminus \{s,t\}$$

Figure 3. Integer Programming Formulation of the MOT problem presented in this paper



```
dist[s] ←o                                   (distance to source vertex is zero)
for  all v ∈ V–{s}
     do  dist[v] ←∞                          (set all other distances to infinity)
S←∅                                          (S, the set of visited vertices is initially empty)
Q←V                                          (Q, the queue initially contains all vertices)
while Q ≠∅                                   (while the queue is not empty)
do   u ← mindistance(Q,dist)                 (select the element of Q with the min. distance)
    S←S∪{u}                                  (add u to list of visited vertices)
     for all v ∈ neighbors[u]
          do  if   dist[v] > dist[u] + w(u, v)          (if new shortest path found)
                   then     d[v] ←d[u] + w(u, v)        (set new value of shortest path)

                                                        (if desired, add traceback code)

return dist
```

Figure 4. Dijkstra's Algorithm to compute the Shortest Path in a Graph used in our code

dences of detection, smaller change of a single object across time, and stronger evidences of initial and terminate points. Accordingly, the arc cost $C(o_i, h_i)$ reflects the reward of including the detection i. $C(h_i, o_j)$ encodes the similarity between detection i and j. $C(s, o_i)$ and $C(h_i, t)$ represent respectively how likely the detection i is the initial point or the terminate point of a trajectory.

3. Algorithm

The SSP algorithm works as follows: it first computes the shortest path between source and target (i.e., path with the lowest negative cost). It then iterates between reversing the edges of the previously found shortest path to form the residual graph G_r, and computing the shortest path in this new residual graph. This process is iterated until no trajectory with negative cost can be found. Finally, trajectories are extracted by backtracking connected, inverted edges starting at the target node.

In the first iteration the shortest path from the source to the target is efficiently retrieved using dynamic programming as the input graph is directed and acyclic. In the following iterations, Dijkstra's algorithm can be leveraged after converting the graph such that all costs are positive. This conversion is achieved by simply replacing the current cost $C_{u,v}$ between nodes (u, v) linked by a directed edge, with $C'_{u,v} = C_{u,v} + d(u) - d(v)$, where $d(u)$ and $d(v)$ are the distance on the shortest path from the source to nodes u and v. Importantly, for u and v on the shortest path we have $C'_{u,v} = 0$ after conversion. Note that for graphs with positive costs and unit flow capacity, the SSP algorithm is similar to the k-shortest paths algorithm.

```
Algorithm: DAG-SPT (G, s)
Input: Graph G=(V,E) with edge weights and designated source vertex s.

     // Initialize priorities and create empty SPT.
     Set priority[i] = infinity for each vertex ;
     // Sort vertices in topological order and place in list.
     vertexList = topological sort of vertices in G;
     // Place source in shortest path tree.
     priority[s] = 0
     Add s to SPT;

     // Process remaining vertices.
     while vertexList.notEmpty()
         // Extract next vertex in topological order.
         v = extract next vertex in vertexList;
         // Explore edges from v.
         for each neighbor u of v
             w = weight of edge (v, u);
             // If there's a better way to get to u (via v), then update.
             if priority[u] > priority[v] + w
                 priority[u] = priority[v] + w
                 predecessor[u] = v
             endif
         endfor
     endwhile

     Build SPT;
     return SPT
```

Figure 5. Algorithm to compute the Shortest Path in a Directed Acyclic Graph using Dynamic Programming. This works for DAGs even in presence of negative edge costs. This algorithm is used as a first initialisation step to compute the shortest distances from source to each node after which we modify the edge costs to be positive so that Dijkstra's Algorithm presented above is applicable

4. Networks to determine Edge Weights

This is a theoretical paper and doesn't present any advice on how to practically determine edge weights for running this algorithm on a real video. Hence, we had to dig deep and find out how to go about constructing the MOT graph from a real video. This section presents what we did to determine edge costs.

Firstly, we had to get the detections from different frames in the videos. We used an implementation of YOLO V3 for this. The detections we got is in the form of a 4 tuple (x, y, w, h) where x, y are the x and y coordinates of the detection in the frame and w, h are the width and height of the bounding box.

Next, we used a Person Re-Identification Network based on Pose Estimation to get how a score of how similar two detections are. This will help us in determining how likely both the detections are detections of the same person.

We know any cost in Computer Vision is divided into two parts, the unary cost and the binary cost. For the problem of Multi Object Tracking, the unary cost is the similarity score we get from Person ReID. The binary cost is the Intersection over Union(IoU) between two detections in successive frames. This measures the smoothness score in a way. We had to modify these scores so that the problem remains that of minimizing cost and not of maximizing score which would have been the case had we directly used similarity scores.

Right now we are using the expression $log(1 - score)$ to handle this.

After computing the required scores, we had to build the graph in a format so that we could feed it to our SSP Algorithm. This involved creating a text file in a specific format that would be compatible with our C++ implementation.

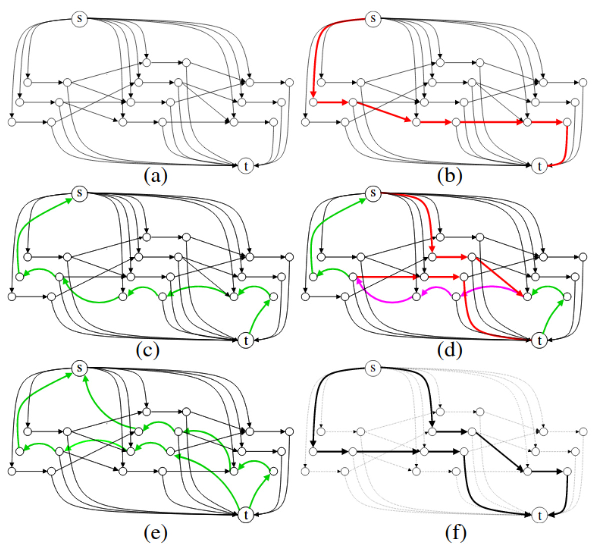


Figure 6. Illustration of successive shortest path algorithm. (a) The tracking problem modeled as a graph as described in Fig. 2. The algorithm should send a given amount of flow from source node s to the terminal t. (b) One unit of flow f_1 is passed through the shortest path (in red) from source to terminal. (c) The residual graph $G_r(f_1)$ produced by eliminating the shortest path and adding edges (in green) with unit capacity and negative cost with the opposite direction. (d) The shortest path found in the residual graph. In this example, this path uses previously added edges, pushing flow backwards and "editing" the previously instanced tracks. (e) Residual graph after passing two units of flow. At this point, no negative cost paths exist and so the algorithm terminates and returns the two tracks highlighted in (f). Note that the algorithm ultimately splits the track instanced in the first step in order to produce the final optimal set of tracks. In this example only one split happened in an iteration, but it is possible for a shortest path to use edges from two or more previously instanced tracks, but it is very rare in practice.

5. Conclusion

For the mid evaluations, what we implemented is the standard SSP algorithm. This will serve as our baseline to beat for our muSSP implementation which is actually the focus of our project. We did it this way because we feel the main structure of the muSSP algorithm is already implemented if we implement SSP. Besides, our understanding of the overall picture has become clear on implementing this. muSSP is a direct extension of SSP. We just need to add a few more observations which the muSSP paper presents which will be our focus for the remaining half of the semester. Besides, we plan to show results on MOT20 challenge dataset.