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

Solving min-cost algorithm efficiently is extremely important for multi-object tracking problems. The paper muSSP, solves this problem with better computational efficiency when compared to the other existing solutions. muSSP algorithm uses the special properties of the graph to clip unnecessary edges and tries to use minimum number of update steps to compute min-cost paths. We run the algorithm on MOT challenge datasets.

1. Introduction

Our project is to implement and extend the paper "muSSP: Efficient Min-cost Flow Algorithm for Multi-object Tracking" by Wang etal[2]. 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.

2. Problem

The problem of object tracking can be formulated as unit-capacity min-cost flow on a DAG. We construct a G(V, E, C), with node set V, arc set E and real valued arc cost function C. The graph has a source s and sink t. For each object detection i, in a frame we introduce two nodes, pre-node $\mathrm{o_i}$ and post-node $\mathrm{h_i}$. We then introduce three edges, ($\mathrm{o_i}$, $\mathrm{h_i}$), (s, $\mathrm{o_i}$), ($\mathrm{h_i}$, t). The edges costs are respectively the confidence of the object detection, and the probability that object is detected for the first time and last time respectively. We then define edges between two consecutive frames and join edges for all pairs of detection, i, j. We introduce edge ($\mathrm{h_i}$, $\mathrm{o_j}$), with the cost of this edge equals to cost of how similar the objects are in the two detection. The capacity of each arc is unity. Any directed path between node u and v is denoted as $\pi_{\mathrm{G}}(\mathrm{u}, \mathrm{v})$. We interpret each st-path as object trajectory candidate. We define the problem as min cost flow algorithm. We define $\mathrm{C_{flow}(f)} = \sum_{(\mathrm{u},\mathrm{v})\epsilon\mathrm{E}} \mathrm{C(u,v)f_{uv}}$. Here we define $\mathrm{f_{uv}}$ is flow between nodes u, v and edge(u, v). $\mathrm{f_{uv}}\epsilon\{0,1\}, \forall(\mathrm{u},\mathrm{v})\epsilon\mathrm{E}$. We can define the problem as integer linear programming problem. We define f^* as the optimal f with least flow cost.

$$\begin{aligned} \underset{f}{\text{minimize}} \quad & \sum_{(\mathrm{u},\mathrm{v})\epsilon\mathrm{E}} \mathrm{C(u,v)f_{uv}} \\ \text{subject to} \quad & \mathrm{f_{uv}}\epsilon\{0,1\}, \forall(\mathrm{u},\mathrm{v})\epsilon\mathrm{E} \\ \text{and} \quad & \sum_{\mathrm{v}:(\mathrm{v},\mathrm{u})\epsilon\mathrm{E}} \mathrm{f_{uv}} = \sum_{\mathrm{u}:(\mathrm{u},\mathrm{v})\epsilon\mathrm{E}} \mathrm{f_{uv}}, \forall \mathrm{u}\epsilon\mathrm{V} - \{\mathrm{s},\mathrm{t}\} \end{aligned}$$

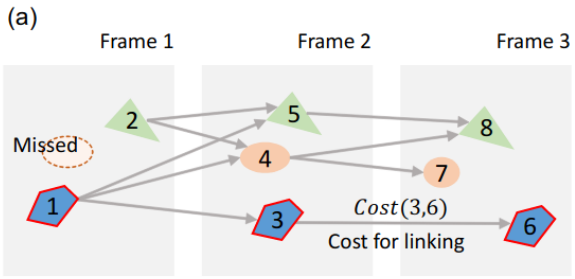

Figure 1: Objects detected in three frames. The first frame has two detections and misses one.Lines between detections are possible ways of linking them. Each line is associated with a cost. Detections 1, 3 and 6 should be linked together as a single trajectory

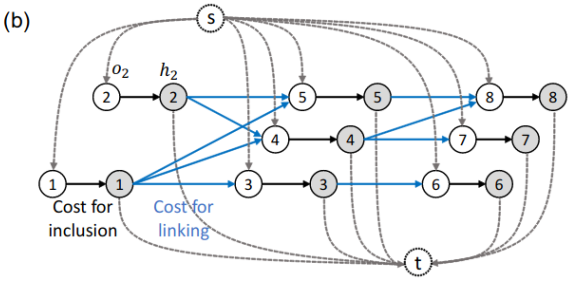

Figure 2: Typical min-cost flow model for MOT problem. Detection i is represented by a pair of nodes: a pre-node o_i and a post-node h_i. The source node s is linked to all pre-nodes and all post-nodes are linked to the sink node t. These edges are shown in dashed lines. Edges between detections are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

3. Algorithm

3.1. SSP 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

Algorithm 1: muSSP Algorithm

Input: Graph G(V, E, C)
Output: Min-cost flow f
$G \leftarrow \text{ClipDummyEdge}(G)$;
$f \leftarrow 0$
$G_r^{(0)} \leftarrow G$
$T_{SP}^{(0)}, \pi^{(0)}, d^{(0)} \leftarrow \text{DAG} - \text{SP}(G_r^{(0)})$
while $\sum_{i=0}^{k} Cost(\pi^{(k)}, G_r^{(0)}) < 0$ **do**
  $f \leftarrow \text{AugmentFlow}(f, \pi^{(k)}, G_r^{(0)})$
  $U^{(k+1)} \leftarrow \text{IdentifyNode4Update}(\pi^{(k)}, T_{SP}^{(k)})$
  $U^{(k+1)} \leftarrow U^{(k+1)} \bigcup U^{(k)}$
  $\Pi = \text{FindMultiPath}(T_{SP}^{(k)}, U^{(k+1)})$
  if $\neg empty(\Pi)$ **then**
    $T_{SP}^{(k+1)} \leftarrow T_{SP}^{(k)}$
    $G_r^{(k+1)} \leftarrow G_r^{(k)}$
    $d^{(k+1)} \leftarrow d^{(k)}$
    $k \leftarrow k+1$
  else
    $G_r^{(k)} \leftarrow \text{ConvertEdgeCost}(G_r^{(k)}, d^{(k)})$
    $G_r^{(k+1)} \leftarrow \text{ResidualGraph}(G_r^{(k)}, \pi^{(k)}, \Pi)$
    $k \leftarrow k+1$
    $G_r^{(k)} \leftarrow \text{ClipPermanentEdge}(G_r^{(k)})$
    $G_{SP}^{(k+1)}, d^{(k)} \leftarrow$
     $\text{DjikstraWithBatchProc}(G_r^{(k)}, U^{(k)}, T_{SP}^{(k-1)})$
    $U^{(k)} \leftarrow \phi$
  end
  $\pi^{(k)} \leftarrow \text{ExtractPath}(T_{SP}^{(k)}, \Pi)$
end

$C_{u,v}$ 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.

3.2. Independent Flipping Lemma

Definition 3.1 (Shortest Path Tree)**.** A single-source shortest path tree (SP) is a tree T (V' , E' , C) embedded in a graph G(V, E, C), rooted at node s, such that π_T(v) ϵ Π^*_G(s, v) for all v reachable from s in G.

Definition 3.2 (Descendants)**.** Given a tree T(V', E', C) rooted at v_0, we define the branch node and the set of descendants nodes for a node v ϵ V': branch(v) = u ϵ V' such that (v_0, u) $\epsilon \pi_T(v)$; descendants(v) = $\{u \epsilon V' | \exists v_x \epsilon V', (v, v_x) \epsilon \pi_T(u)\}$

Definition 3.3 (Independent nodes)**.** In an SP, two nodes u and v are independent, if and only if branch(u) $\neq$ branch(v).

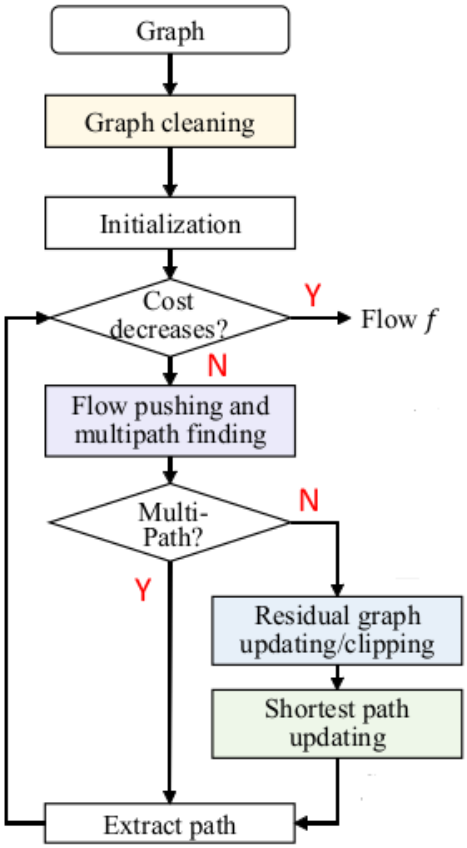

Figure 3: Flow Chart of muSSP Algorithm

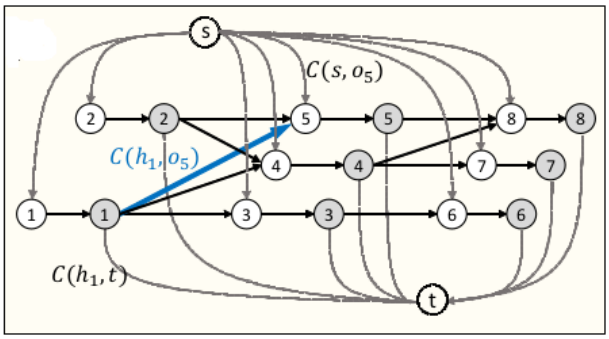

Figure 4: Graph cleaning module for dummy edge removal. The blue arc between h 1 and o 5 are thicker as it has larger arc cost. If $C(h_1, o_5) > C(s, o_5) + C(h_1, t), arc(h_1, o_5)$ will be removed.

Lemma 3.1 (Independent flipping). Given a residual graph $G_r(V_r, E_r, C_r)$ and its SP denoted as T rooted at s, if we flip all the edges in path $\pi_T(t)$ and get a new graph G'_r, for node $v \epsilon V_r$ that is independent with node t in T, its least cost-path $\pi_T(v)$ is still valid in the new graph. $\pi_T(v) \epsilon \Pi^*_{G'_r}(s, v)$

The lemma states that when we flip the shortest path π = (s, v_0), (v_1, v_2), ..., (v_x, t), only the nodes whose branch nodes is v_0 need update. Reducing the size of descendants directly helps to improve the efficiency.

3.3. Graph cleaning with dummy edge

Definition 3.4 (Dummy Edge)**.** An edge (u, v) ϵ E is defined as a dummy edge if C(u, v) $>$ C(s, v) + C(u, t).

Lemma 3.2 (Dummy edge). No dummy edges will appear in any optimal solution.

Theorem 3.3. Given a graph G(V, E, C) for multi object tracking, removing all dummy edges will not influence the optimality of the solution.

The lemma states that no dummy edges will appear in the solution and hence can be discarded.

3.4. Residual Graph with permanent edge clipping

Definition 3.5 (Permanent Edge)**.** Given a residual graph $G_r(V_r, E_r, C_r)$, we define the set of permanent edges as $\{(u, v) \epsilon E_r |$ v = s OR u=t$\}$

Lemma 3.4. Any s-t path with permanent edge is not a simple path.

Lemma 3.5. Given a residual graph, we can always find a shortest s-t path which is also a simple path, unless t cannot be reached from s.

Theorem 3.6. Given a residual graph $G_r(V_r, E_r, C_r)$, removing all its permanent edges does not influence the optimality of the final solution.

The lemma states that permanent edge clipping will not affect the optimality of final solution.

3.5. Multi-path finding

Lemma 3.7. Given a residual graph $G_r(V_r, E_r, C_r)$, its SP rooted at s, and the shortest distance function d from s to each node, we have d(t) = min(d(v)+$d_t(v)$), $v \epsilon V_r - \{t\}$.

Theorem 3.8. Given a residual graph $G_r(V_r, E_r, C_r)$, its SP denoted as $T(V'_r, E - r', C'_r)$ rooted at s, and a sorted list of $\{$d(v)+$d_t(v)\}$ with $v \epsilon V_r$ with ascending order, if k nodes $\{v_1, v_2, \ldots, v_k\}$ that occupy the top k locations of the list are mutually independent, the k paths $\{\pi_T(v_1), \pi_T(v_2), \ldots, \pi_T(v_k)\}$ can be simultaneously initiated as k shortest paths.

Based on theorem, we can efficiently check whether we can instantiate multiple paths from current SP. This is achieved by function FindMultiPath given the information of the branches to be updated (IdentifyNode4Update).

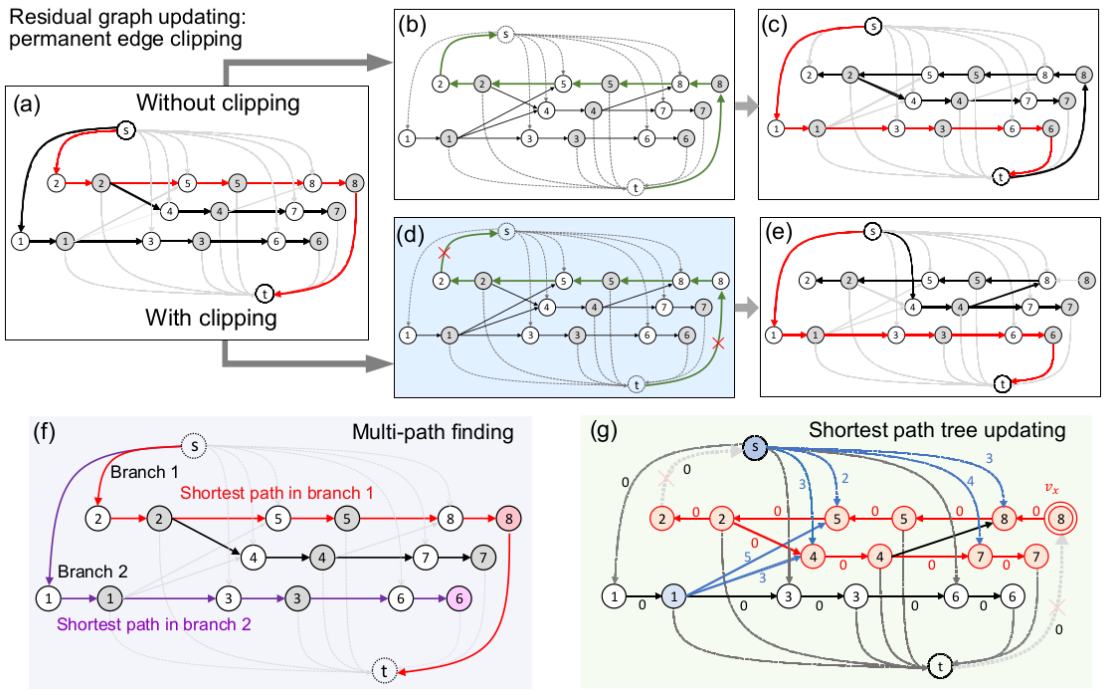

Figure 5: (a) A single source shortest path tree (SP) shown with bold edges. The red path is the shortest s-t path. (b) The shortest path in (a) is flipped (shown in green). (c) SP based on the updated residual graph in (b). The red path is the new shortest path. The path through t makes the branch containing the shortest s-t path large and we have more nodes to update. (d) Updated residual graph with permanent edges clipped (red crosses). (e) With clipping, the branch containing the shortest path is much smaller than (c). (f) Two branches in the SP, each containing a shortest s-t path (red and purple). Only the red one connects to t in SP as it has a lower cost. We can push flows on both paths without updating residual graph. Post-nodes h_8 and h_6 are independent while h_8 and h_7 are not. (g) After removing permanent edges in the residual graph (b), we need to update the distances from s to the red nodes, because the shortest paths from s to these nodes no longer exist due to the flipping operations. These nodes form a 0-tree. Note that edges in SP of (a) all have zero costs. We initialize the distances from s to them using external edges (in blue) connecting to them. For example, pre-node o_5 is linked by arc (h_1, o_5) and (s,o_5), so the initial distance is 2. Based on the initial distance, only a subset of red nodes need to be pushed to the heap in Dijkstra's algorithm. Once a node is popped from the heap, we can batch update its descendants. For example, if o_5 is firstly popped, all its descendants will be assigned the same distance as o_5.

3.6. Batch updating and heap shrinking for shortest path tree

Definition 3.6 (0 tree)**.** If a tree has zero cost for all edges, we call it a 0-tree.

Lemma 3.9. Given a 0-tree $T_0(V_0, E_0, C_r)$ embedded in residual graph $G_r(V_r, E_r, C_r)$, if $v \in descendants(v_0)$ $d_u^*(v) \leq d_u^*(v_0) \forall u \epsilon V_r, \forall v_0 \epsilon V_0$.

Theorem 3.10. In Dijkstra's algorithm, if the distance from s to a node v in a 0-tree is permanently labeled as $d(v)$, $d(v)$ is also the permanent label for the nodes in $descendants(v)$ that haven't been permanently labeled.

Lemma 3.11. In a 0-tree, nodes with larger temporary distance labels than their parent belong to set P

From this Lemma, we can use a breadth-first search starting from root (v_x) of the 0-tree, pushing only nodes whose temporary distance labels are not larger than their parents in the 0-tree

4. 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[1]

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.

5. Results

We evaluate our results on MOT16 sequences available publicly on https://motchallenge.net. As promised by the authors of this algorithm, we got massive improvements in speed when compared with implementation of SSP algorithm.

We use detections from Yolo v3[1] to do multi object tracking in the selected sequences. Using detections from the datasets gave us really bad results indicating the importance of object recognition and accurate object detection in Multi Object Tracking.

The MOTA is the most used summary metric for MOT. It is defined as follows:

$$MOTA = 1 - \frac{\sum_t FN_t + FP_t + IDS_t}{\sum_t GT_t} \qquad (1)$$

where GT_t is the number of objects present in Ground Truth at time t, and FN_t is the number of false negatives (missed targets), FP_t the number of false positives (ghost trajectories), IDS_t the number of identity switches at time t.

A target is considered missed if the IoU with the ground truth is inferior to a given threshold. (Note that the MOTA can be negative.) In "Tracking the trackers", the authors conducted an experiment where people evaluated which tracker was the best. It turns out that people usually agree with these metrics, meaning that their evaluations were positively correlated with the MOTA.

Note that in the MOTA metric, the predominant factor of bad performance is usually due to False Negative: a tracker will usually help the detector to delete False Positive, but when the detector fails, current multiple object trackers are not able to recover from this failure.

Sequence Name	motA	IDF1
MOT16-02	14.3%	16.9%
MOT16-04	16.4%	22.9%
MOT16-05	23.1%	28.2%
MOT16-09	26.2%	29.6%
MOT16-10	18.9%	20.3%
Overall	17.4%	22.4%

Table 1: Results on the given detections in the corresponding datasets.

Sequence Name	motA	IDF1
MOT16-02	17.5%	17.2%
MOT16-04	17.8%	15.3%
MOT16-05	52.6%	51.1%
MOT16-09	58.3%	47.0%
MOT16-10	31.2%	21.9%
MOT16-11	53.8%	50.1%
MOT16-13	23.3%	23.4%
Overall	26.9%	25.6%

Table 2: Results on using Yolo v3 detections using a conf-thres of 0.4.

6. Conclusion

The best performing systems on MOT Challenge Ranklist has accuracy of barely 50%. In order to deploy these systems in real time applications, speed is of utmost importance. As far as speed is concerned, muSSP has indeed done a great job in improving the speed by many folds. The sub-field of Multi Object Tracking has scope for vast improvements in terms of speed as well as accuracy. This domain requires knowledge of Graph Algorithms as well as sophisticated deep learning techniques for identification of objects. Advancement in this field would in turn advance us towards solving many fundamental problems in Computer Vision. Hence, we inspire young Computer Vision researchers to come forward and research in this area which indeed has immense scope for amazing research.

References

[1] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018. 4, 5

[2] C. Wang, Y. Wang, Y. Wang, C.-T. Wu, and G. Yu. mussp: Efficient min-cost flow algorithm for multi-object tracking. In *Advances in Neural Information Processing Systems*, pages 423–432, 2019. 1