High Performance Graph Segmentation for ATLAS GNN Track Reconstruction

Levi Condren, Xiangyang Ju, Ryan Liu, Alina Lazar,
Tuan Minh Pham, Daniel Murnane, Alexis Vallier
& Daniel Whiteson
On behalf of the
ATLAS Computing Activity

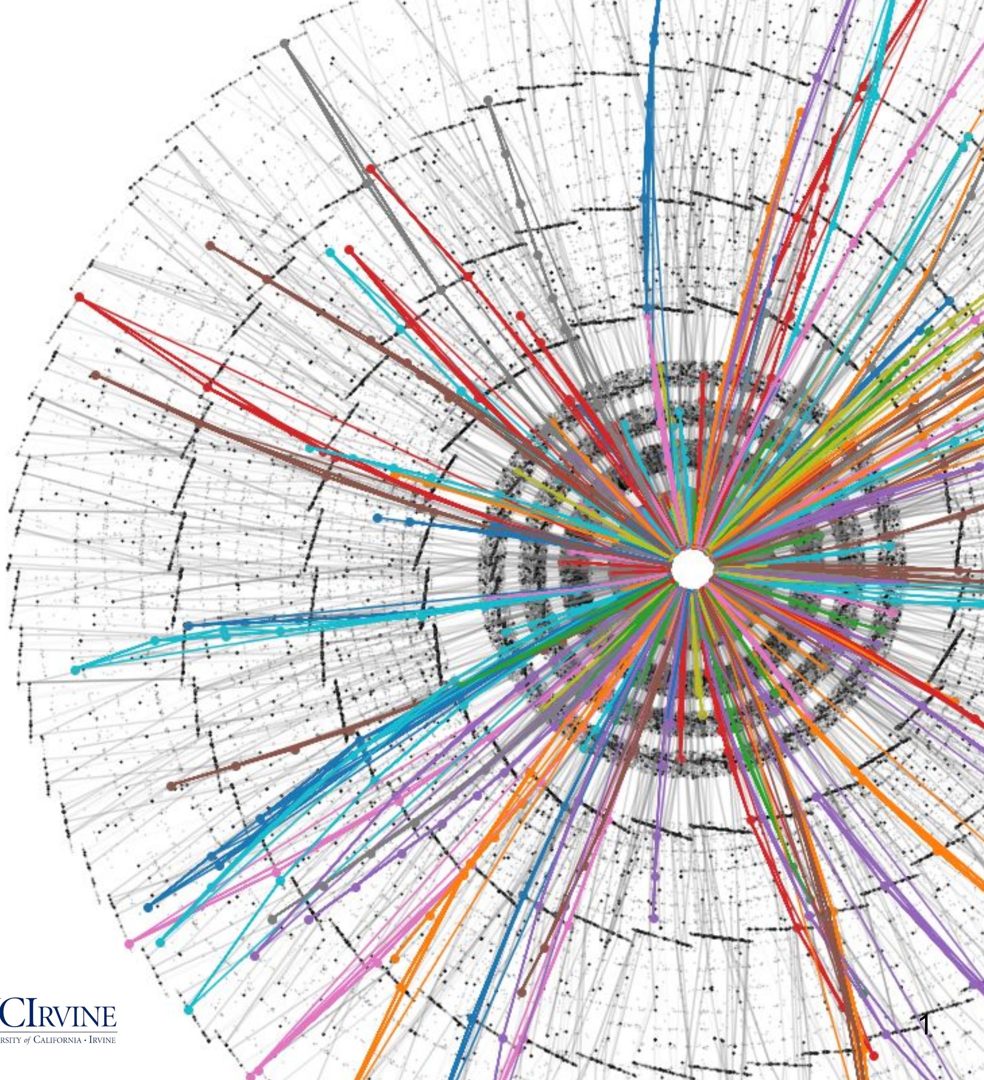
ATLAS EXPERIMENT

UNIVERSITY OF COPENHAGEN

Berkeley UNIVERSITY OF CALIFORNIA

BERKELEY LAB

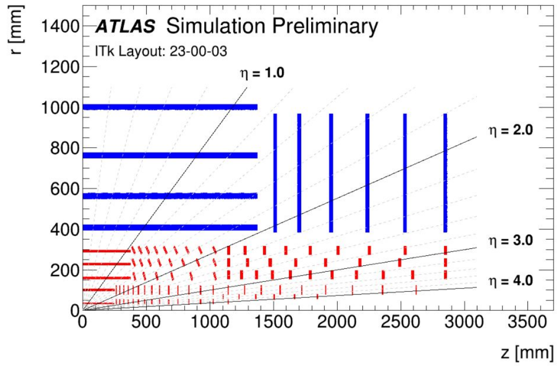
UCIRVINE UNIVERSITY of CALIFORNIA · IRVINE



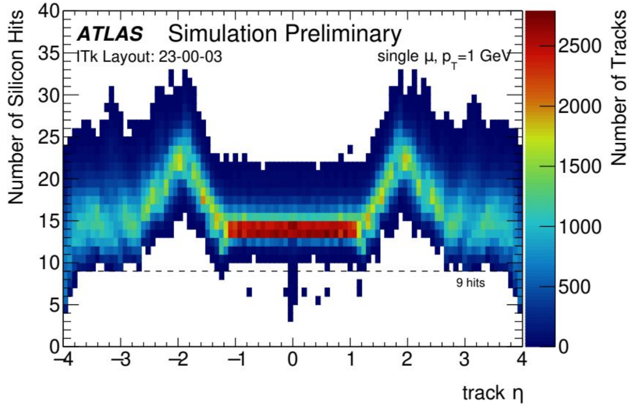
Track Reconstruction & the GNN4ITk Pipeline

Tracking in ATLAS HL-LHC Inner Tracker (ITk)

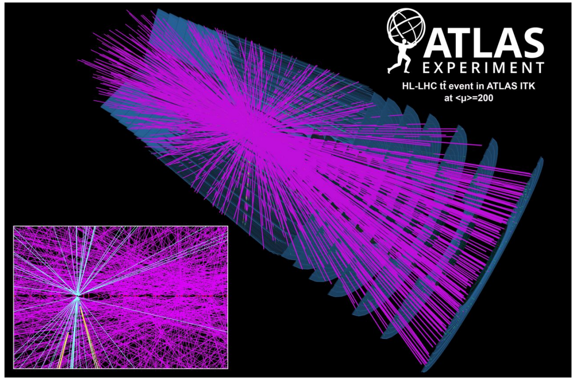
[ATL-PHYS-PUB-2021-024]



[ATL-PHYS-PUB-2021-024]

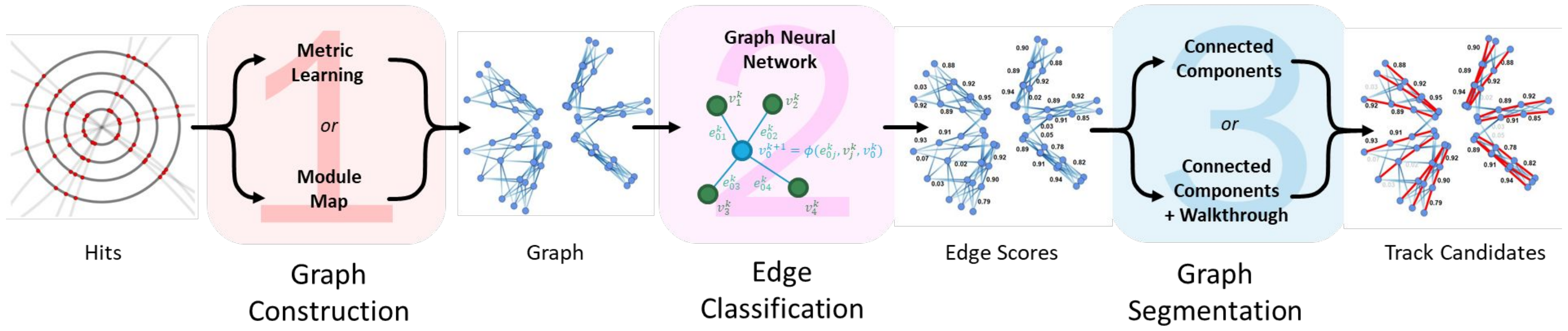


- Track finding requires associating each hit to a track candidate
- Number of hits per $pp \to t\bar{t}$ event: 311,000 +/- 35,000
- Number of particles per $pp \to t\bar{t}$ event: 16,000 +/- 1,700
- Innermost pixel layer 25x100 μm^2, all other pixel layers 50x50 μm^2
- Strip layers are at millimeter resolutions
- We focus on Athena simulation in the following slides



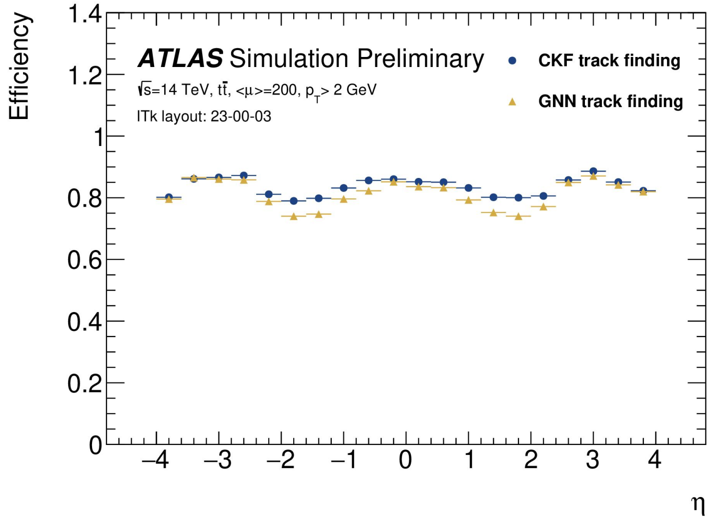
UNSG-2021-57

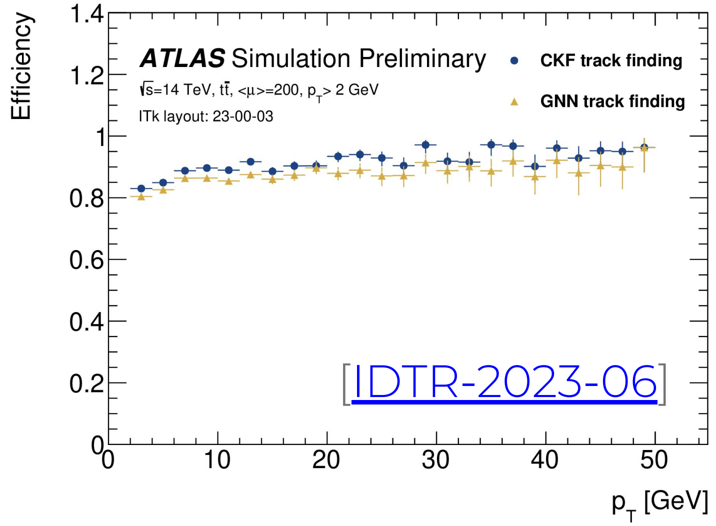
GNN4ITk Pipeline



- Pipeline receives clusters = collections of energy deposits on silicon. These are associated with 3D spacepoints, to be used as nodes for stage 1 onwards
- Out of stage 3 we obtain a set of track candidates, each is an unordered set of spacepoints
- For processing in Athena track fitting chain, we associate these back to the original clusters, and order in increasing distance from beamspot origin

Track Reconstruction Performance





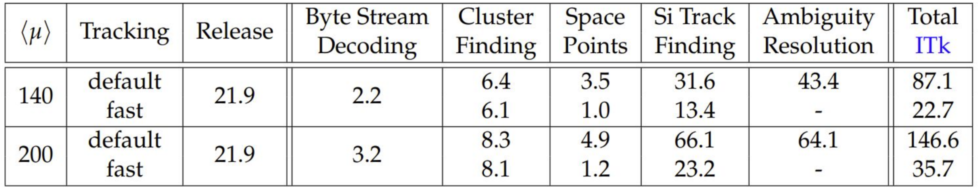
[IDTR-2023-06]

- Tracking efficiency compared with current combinatorial kalman filter (CKF) technique
- Behaviour across **η** and p_T similar to CKF - good sanity check!

HL-LHC Offline & Online Track Reconstruction Needs

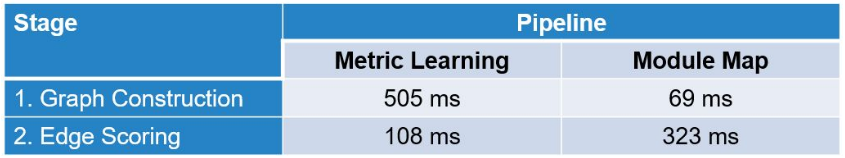
	LHC Run 3	**HL-LHC**
L0 trigger accept	100 kHz	**1 MHz**
Event Filter accept	1 kHz	**10 kHz**
Event size	1.5 MB	**4.6 MB**

- Event filter (high level trigger) contains tracking
- CPU-based Fast Tracking: 23.2 HS06s/event (around 1 second per single-core CPU), small drop in track efficiency: 1-2% on average, 5% for pT in [1,1.5]GeV
- GPU-based GNN4ITk pipeline: First two steps run in 400-600 ms/event. But final step has previously taken around **42 seconds** to run



$\langle\mu\rangle$	Tracking	Release	Byte Stream Decoding	Cluster Finding	Space Points	Si Track Finding	Ambiguity Resolution	Total ITk
140	default	21.9	2.2	6.4	3.5	31.6	43.4	87.1
	fast			6.1	1.0	13.4	-	22.7
200	default	21.9	3.2	8.3	4.9	66.1	64.1	146.6
	fast			8.1	1.2	23.2	-	35.7

CPU-based Fast tracking vs Default tracking timing (HS06 x s) [ATLAS-TDR-029-ADD-1]



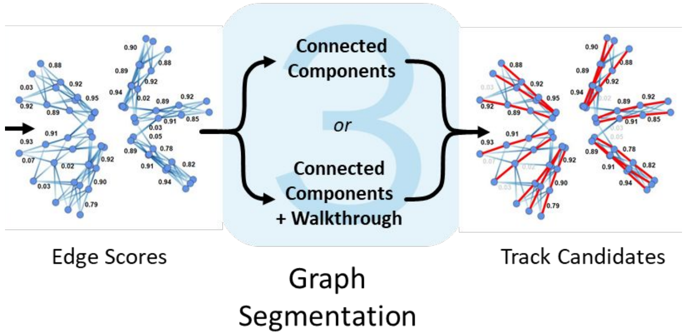
Stage	Pipeline	
	Metric Learning	Module Map
1. Graph Construction	505 ms	69 ms
2. Edge Scoring	108 ms	323 ms

GNN4ITk pipeline execution times of first two stages, per event [ATL-PHYS-PUB-2024-018]

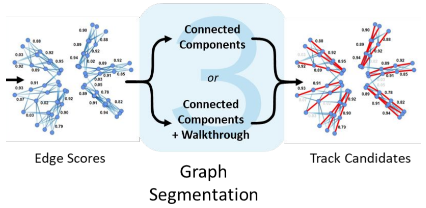
Graph Segmentation

Stage 3: Graph Segmentation

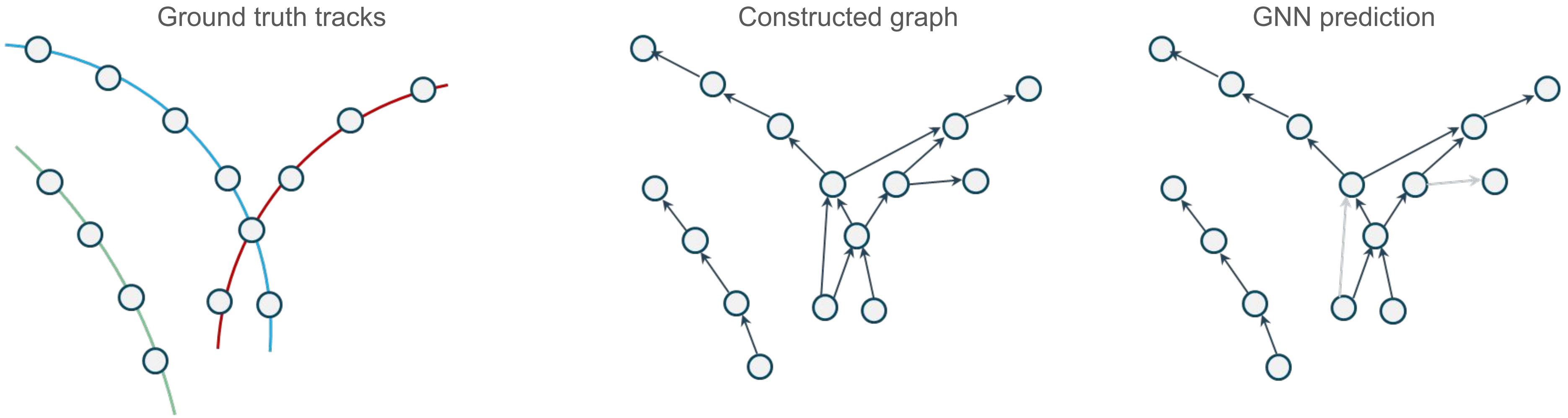
- The task:
  - Given a collection of hits in detector and directed edges (a hypothesis that the two hits were created successively by the same particle), each with a score…
  - Produce a set of track candidates (sets of nodes)
- Some nodes may belong to background tracks, some may be noise, some may be shared between target tracks



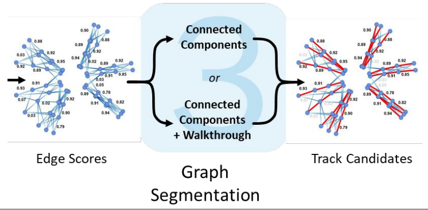
Stage 3: Graph Segmentation



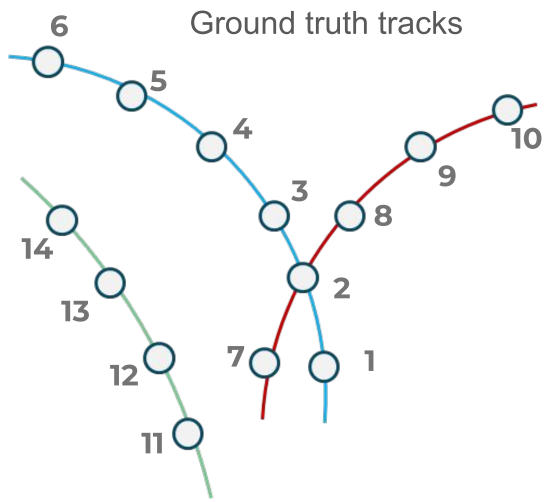
- There are many ways to solve this problem
- In the case where we do not allow shared nodes, this is a classic *graph partitioning* problem



Stage 3: Graph Segmentation



- There are many ways to solve this problem
- In the case where we do not allow shared nodes, this is a classic *graph partitioning* problem



Would like to produce:

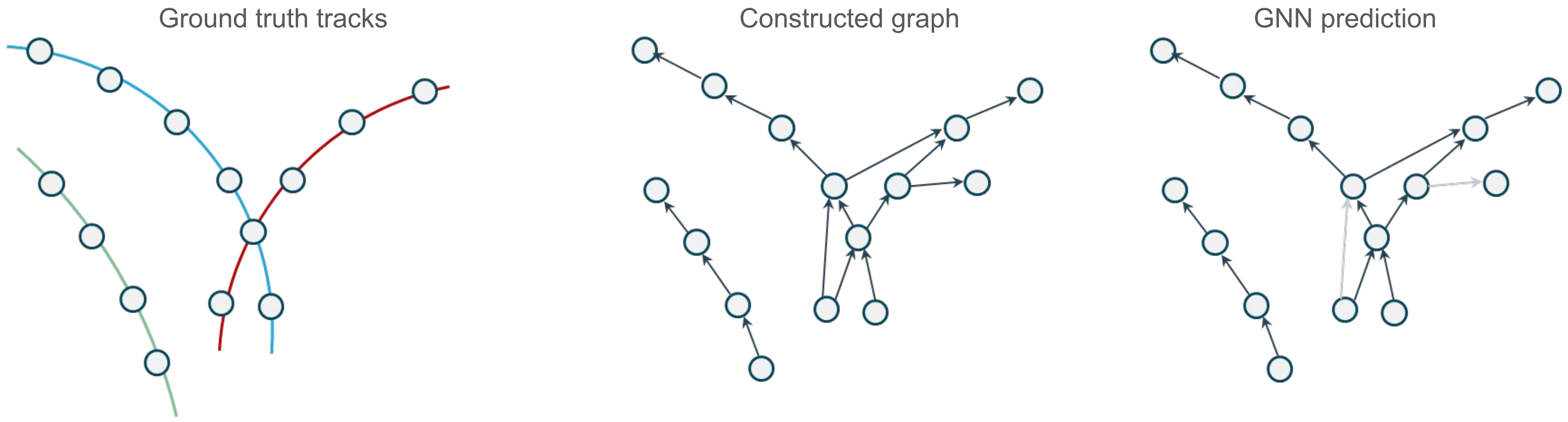
[1, 2, 3, 4, 5, 6]

[7, 2, 8, 9, 10]

[11, 12, 13, 14]

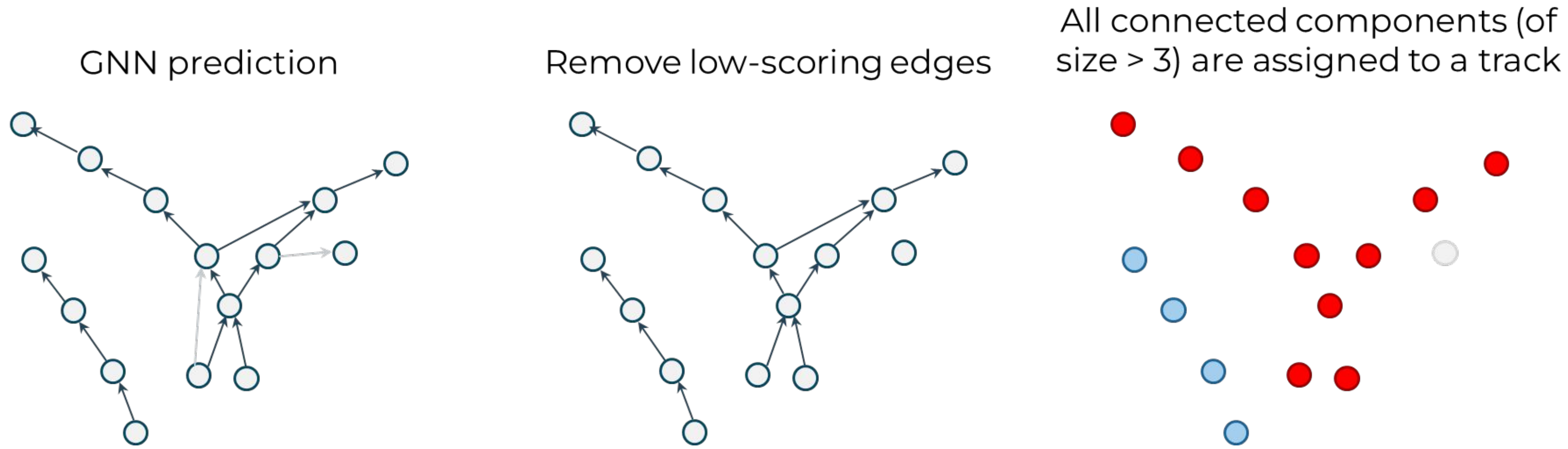
The Easiest Solution: Connected Components

Recall that we have the GNN predicted edge scores:



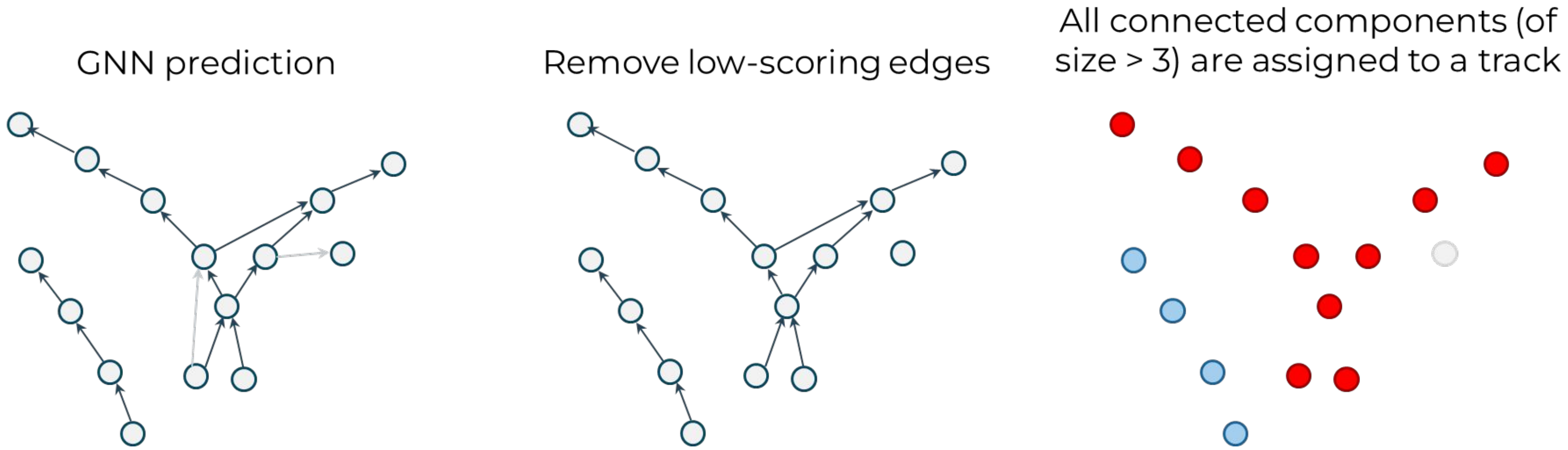
The Easiest Solution: Connected Components

- The simplest idea is (weakly) Connected Components
- All nodes belonging to the same component after removing low-scoring edges, are assigned to a track
- That is, the nodes must be reachable via an undirected path (hence “weakly”)


GNN prediction
Remove low-scoring edges
All connected components (of size > 3) are assigned to a track

The Easiest Solution: Connected Components

- This is an extremely fast algorithm with the Scipy implementation, requiring **only a few milliseconds for O(1000) components,** on a single thread
- Each node and edge visited only once, giving **good scaling performance: O(N+E)**
- Iterative Depth-first Search (DFS) means **predictable and reliable memory behaviour** (as opposed to recursive DFS)

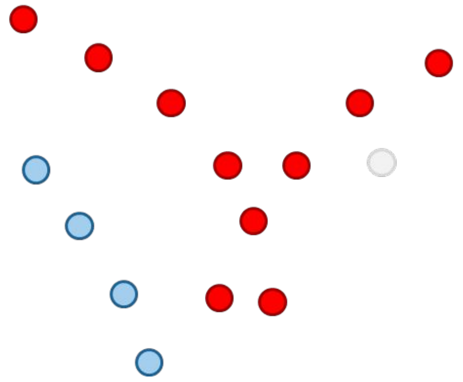

GNN prediction
Remove low-scoring edges
All connected components (of size > 3) are assigned to a track

The Easiest Solution: Connected Components

- This is an extremely fast algorithm with the Scipy implementation, requiring **only a few milliseconds for O(1000) components,** on a single thread
- Each node and edge visited only once, giving **good scaling performance: O(N+E)**
- Iterative Depth-first Search (DFS) means **predictable and reliable memory behaviour** (as opposed to recursive DFS)

- However note that (as in example), it can *merge* tracks into a single candidate if the score threshold is too low
- We can raise the threshold, but then we are just as likely to *split* a track in multiple candidates
- Both lead to low track finding efficiency

All connected components (of size > 3) are assigned to a track



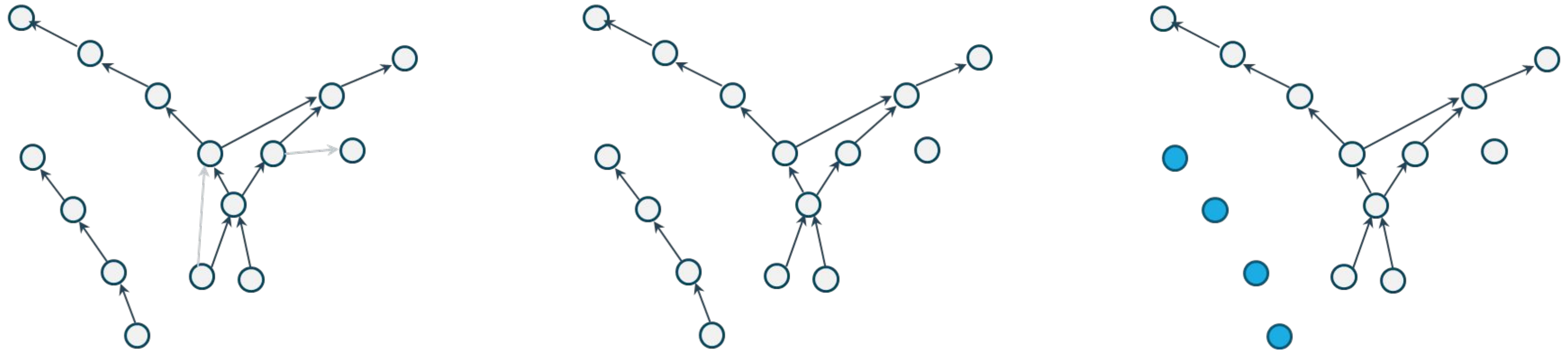
Walking through the Hit Graph

Walkthrough Algorithm

GNN prediction

1. Remove low-scoring edges

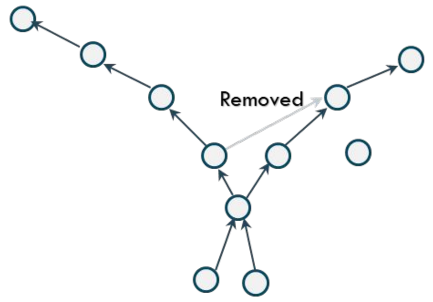
2. Identify "chains" or "simple paths" – nodes with one in, one out edge. Output them first



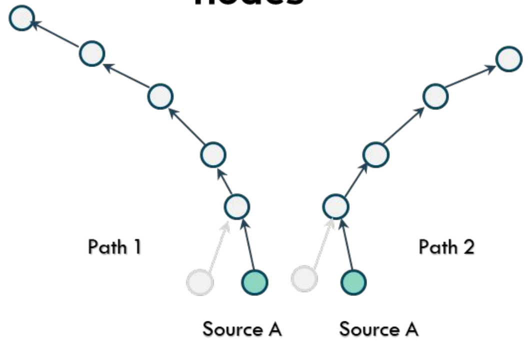
- To avoid merging tracks, we want out candidates to be "chain-like", with maximum one incoming and one outgoing edge
- The walkthrough algorithm is inspired by the way traditional track finding is performed
- It performs well, but is much slower than CC

Walkthrough Algorithm

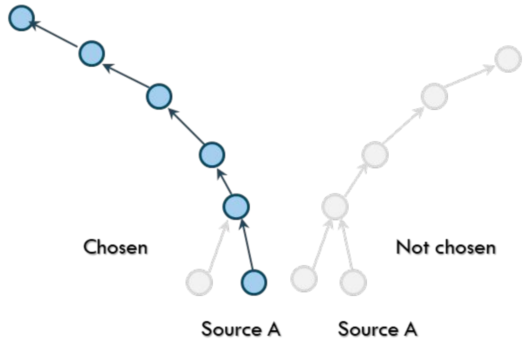
3. On the remaining non-chains, select all edges that are either the highest scoring out edge **or** are above a high threshold



4. For each source node (no in edges), build all paths leading away, that **do not contain used nodes**



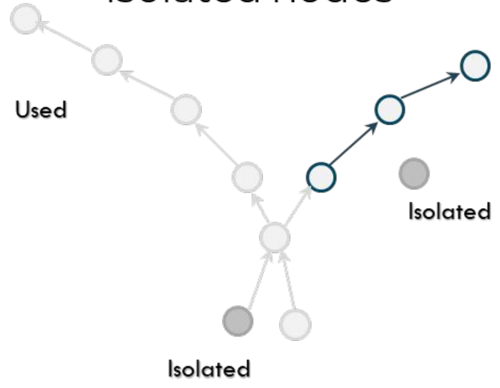
5. For each source, choose the longest path. Add it to the "used nodes" pile



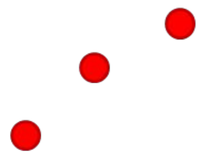
- To avoid merging tracks, we want out candidates to be "chain-like", with maximum one incoming and one outgoing edge
- The walkthrough algorithm is inspired by the way traditional track finding is performed
- It performs well, but is much slower than CC

Walkthrough Algorithm

4. Repeat path building, avoiding used nodes and isolated nodes



5. Choose the longest path, add to pile of used nodes



- Repeat steps 4 and 5 until all nodes have been used, or have no neighbours in the graph
- Clearly, as this is sequential, some tracks will arbitrarily be built that split later tracks

Walkthrough Sub-algorithms

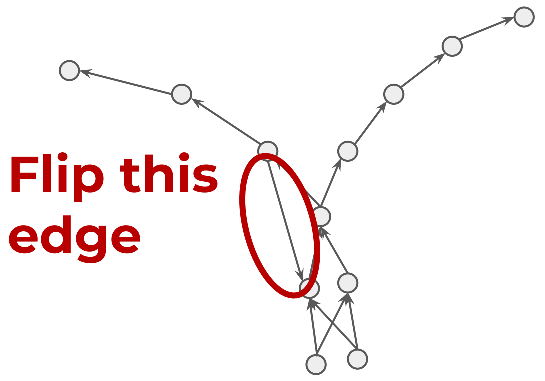
Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles*
2. *Filter Graph*
3. *Extract Chains*
4. *Topological Sort*
5. *Build Paths*

Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles:* Ensure the directed graph is acyclic by directing edges in increasing *R*, scaling as ~ *O(E)*
2. *Filter Graph*
3. *Extract Chains*
4. *Topological Sort*
5. *Build Paths*

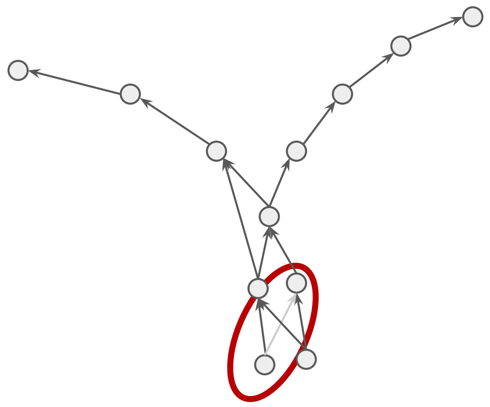


Graph with number of nodes *N*, number of edges *E*

Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles*
2. *Filter Graph:* Remove low-scoring edges ~ *O(E)*
3. *Extract Chains*
4. *Topological Sort*
5. *Build Paths*

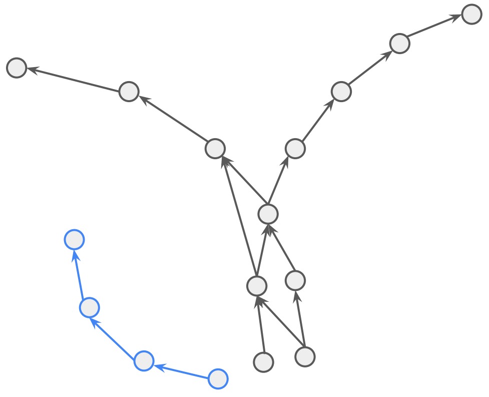


Remove this edge

Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

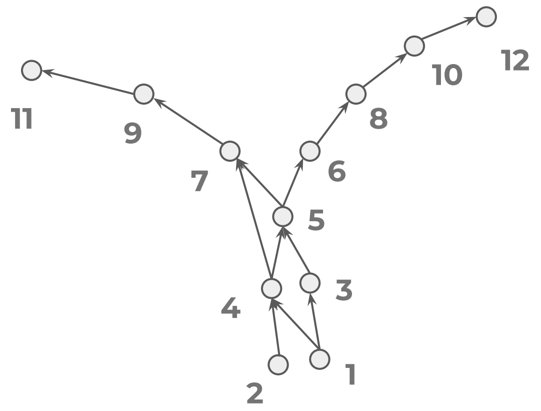
1. *Remove Cycles*
2. *Filter Graph*
3. *Extract Chains:* Apply connected components, and any chain-like components are instantly submitted as candidates ~ $O(N+E)$
4. *Topological Sort*
5. *Build Paths*



Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles*
2. *Filter Graph*
3. *Extract Chains*
4. *Topological Sort:* Order node indices such that earlier nodes can visit subsequent nodes if a path exists between the two ~ *O(N+E)*
5. *Build Paths*



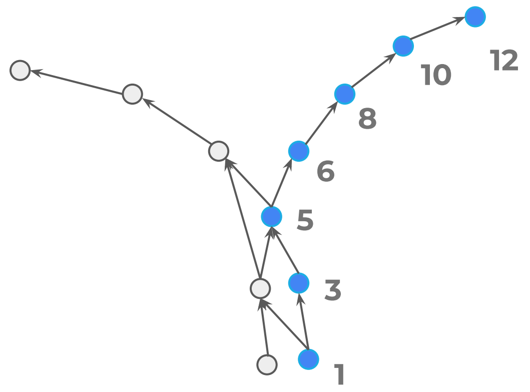
Implemented as a Breadth-first Search (visit each edge only once): Kahn's Algorithm*

* A. B. Kahn. 1962. Topological sorting of large networks. Commun. ACM 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles*
2. *Filter Graph*
3. *Extract Chains*
4. *Topological Sort*
5. *Build Paths:* For each starting node, choose longest path - this is a DAG *source-to-sink* algorithm. Currently no heuristic to choose between equally-long paths ~ best case *O(N)*, worst case *O(N!)*

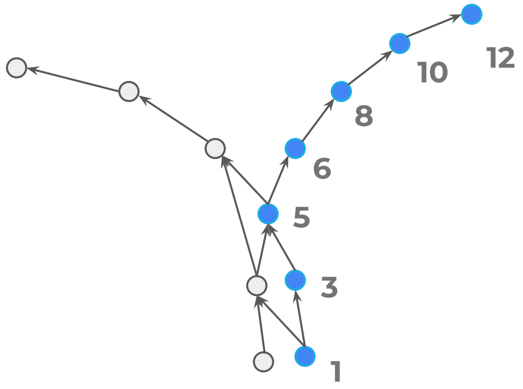


Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

5. *Build Paths:* To trade off between best case *O(N)* and worst case *O(N!)*, we do a sort of *beam search:*
   a. The highest scoring neighbour is always considered for the longest path (regardless of edge score)
   b. Any neighbours with edge score > 0.6 are considered for the longest path

Reduces combinatorics significantly, while avoiding splitting tracks (every node will have at least one outgoing edge)

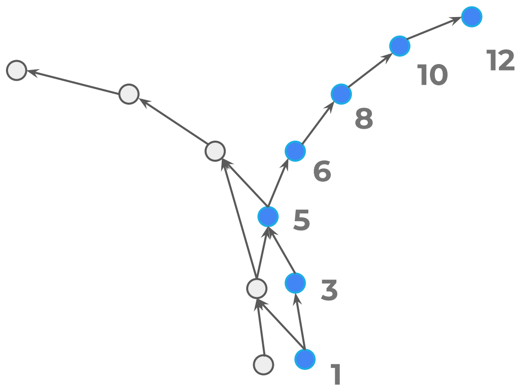


Walkthrough Sub-algorithms

Precisely, the walkthrough requires the following sub-algorithms:

1. *Remove Cycles*
2. *Filter Graph*
3. *Extract Chains*
4. *Topological Sort*
5. *Build Paths*

To produce public physics results, most of these steps used NetworkX implementation. This took *O(minutes)* per event - we need something faster!



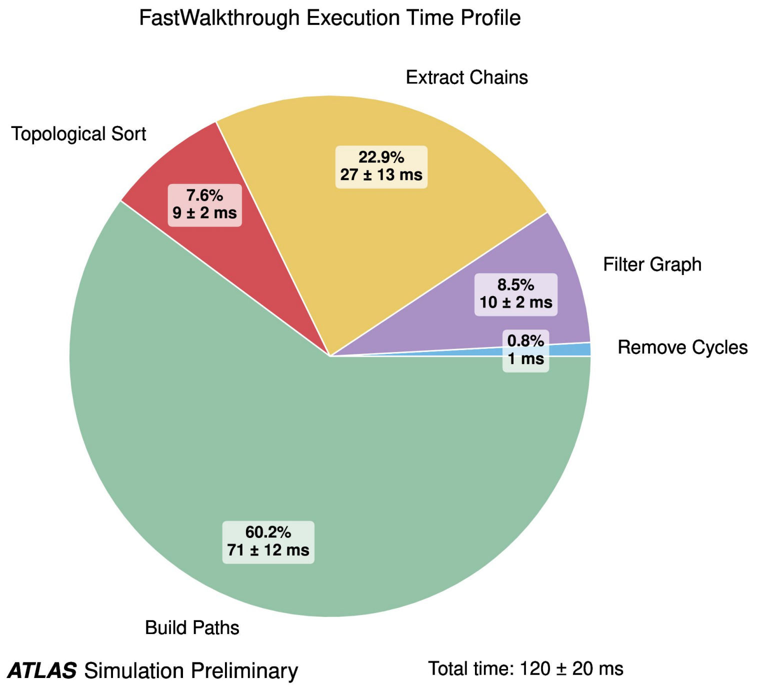
Optimisations to Segmentation

FastWalkthrough

A variety of improvements can be made:

1. ***Remove Cycles***: Unchanged
2. ***Filter Graph***: Use Pytorch Geometric (PyG) graph representation
3. ***Extract Chains***: Use Scipy CC and PyG scatter_max across components
4. ***Topological Sort***: Pure python+numba implementation
5. ***Build Paths***: Pre-process graphs to get max+min edges (with scatter operations), then pure python+numba implementation

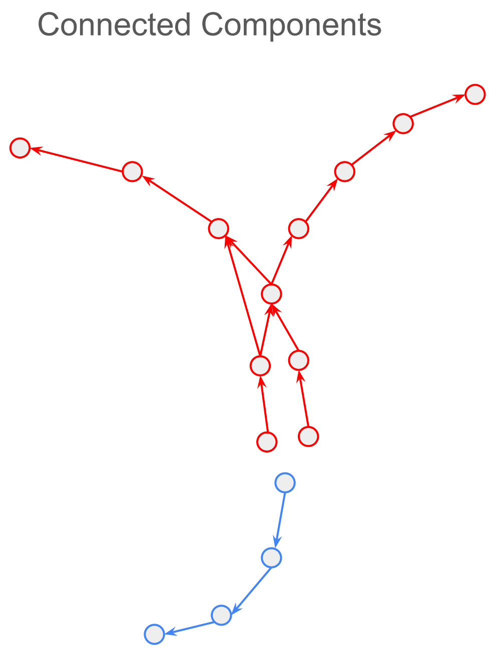
(Timing is per-event)



[ATL-PHYS-PUB-2024-018]

Junction Removal

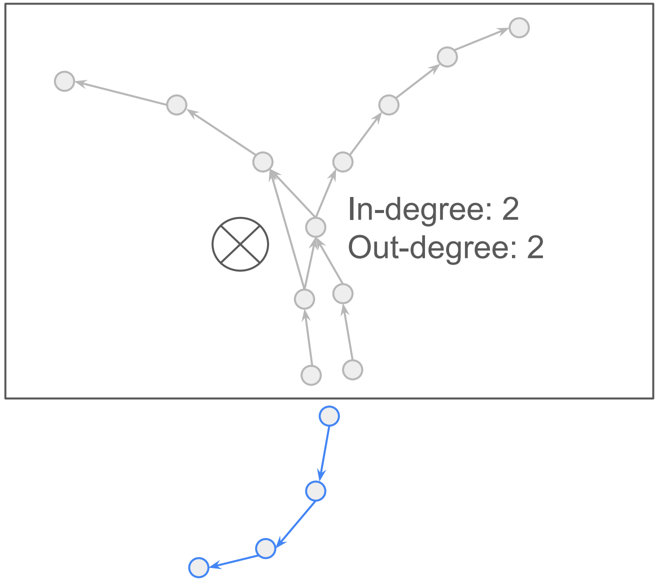
- The preprocessing of junctions in FastWalkthrough can be extended to any heuristics
- For example, first remove chain-like components as usual



Junction Removal

- The preprocessing of junctions in FastWalkthrough can be extended to any heuristics
- For example, first remove chain-like components as usual

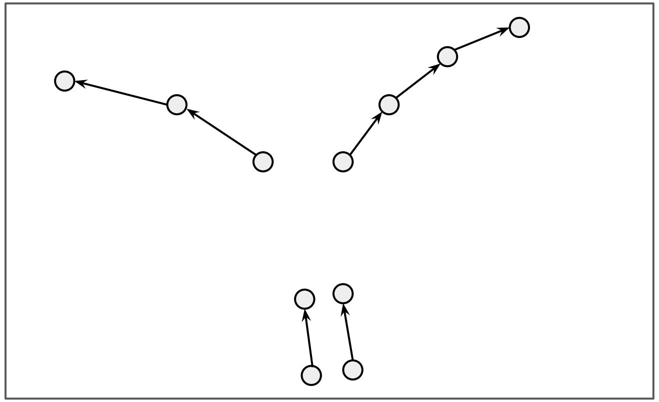
Chain-like: all hits must have in-degree and out-degree less or equal to 1



Junction Removal

- The preprocessing of junctions in FastWalkthrough can be extended to any heuristics
- For example, first remove chain-like components as usual
- Then remove all “junctions” (nodes with more than one outgoing or incoming edge)

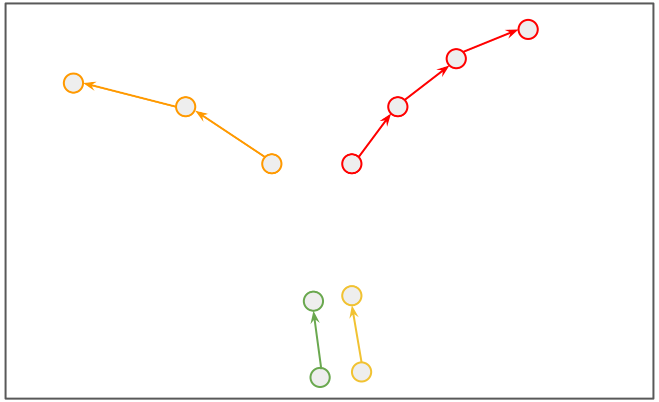
Remove junctions (in or out degree >= 2)



Junction Removal

- The preprocessing of junctions in FastWalkthrough can be extended to any heuristics
- For example, first remove chain-like components as usual
- Then remove all "junctions" (nodes with more than one outgoing or incoming edge)
- Then re-run connected components

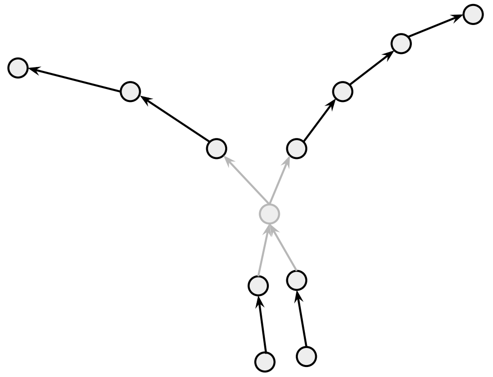
Connected Components



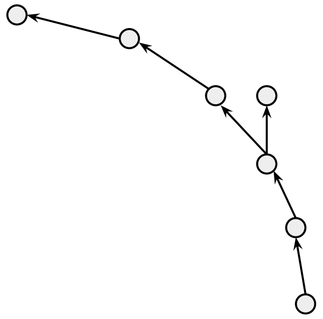
Junction Removal

- One could apply many kinds of choices
- E.g. only remove “X-junctions”, but allow “Y-junctions”

Junction removed
(in-degree=out-degree=2)

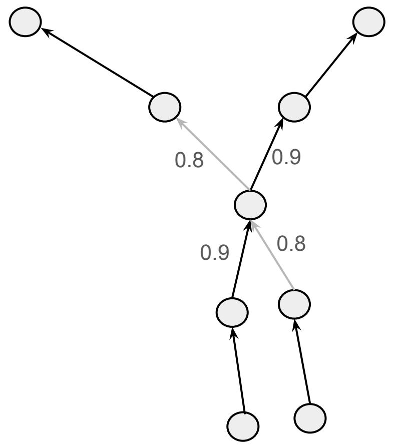


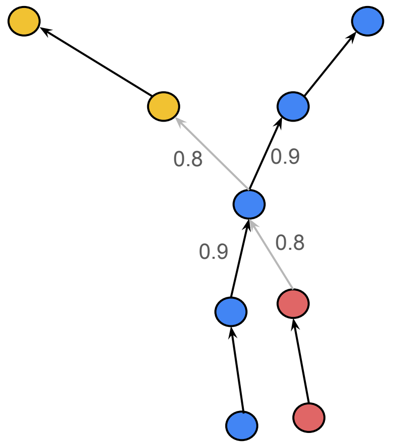
Junction NOT removed
(in-degree ≠ out-degree)



Junction Removal

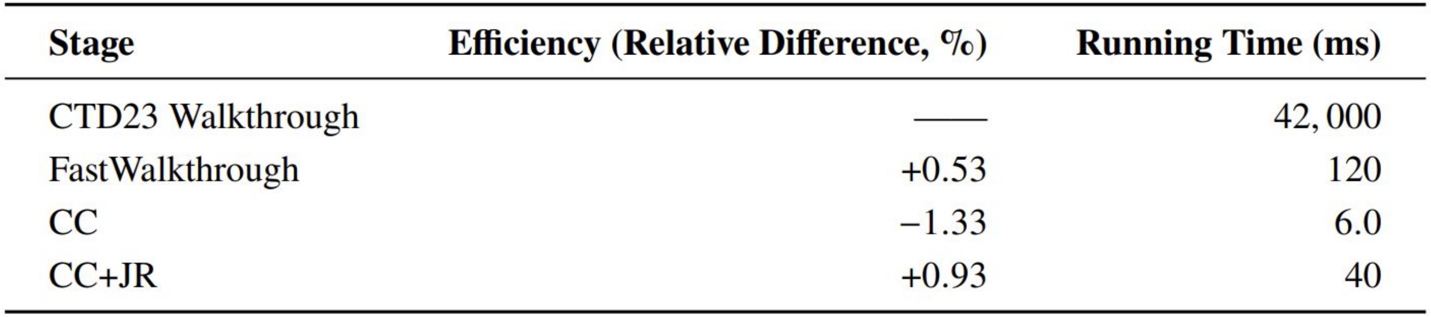
- The best physics performance actually came simply from choosing the highest-scoring incoming and outgoing edges
- Then apply connected components, as usual
- This is *Connected Components + Junction Removal Version 3 (CC+JR v3)*





Comparison of Segmentation Approaches

We are able to reduce the running time of this stage (compared with that used to produce public physics results “CTD23”) by 3 orders of magnitude, and *increase* the physics performance!



Stage	Efficiency (Relative Difference, %)	Running Time (ms)
CTD23 Walkthrough	——	42, 000
FastWalkthrough	+0.53	120
CC	–1.33	6.0
CC+JR	+0.93	40

Per-event execution times and relative difference in integrated physics efficiencies of the various graph segmentation techniques available in Stage 3 (graph segmentation). Differences are calculated relative to the baseline CTD23 Walkthrough as (eff – CTD)/CTD. The score cut on CC set to 0.01, with the minimum and additive thresholds of walkthroughs set to 0.1 and 0.6 respectively. The running times are evaluated on a single CPU core (AMD EPYC 7763). CTD23 Walkthrough is the same as that used in IDTR-2023-06.

[ATL-PHYS-PUB-2024-018]

Next Steps

- Small tweaks required to run on GPU (Pytorch, PyG support GPU out-of-the-box, numba requires some massaging) - expect to reduce existing CC+JR version closer to 10ms
- Integration with production systems - Athena and ACTS
- C++ implementation needed for integration, and expected to be somewhat faster than PyG+numba
- Current FastWalkthrough and CC+JR still involve duplication (e.g. running across number of edges *E* with CC *~O(E)* then JR *~O(E)*)
- Fusing these steps will reduce again by some factor
- Expect to obtain an algorithm of >1KHz on single GPU with native CUDA