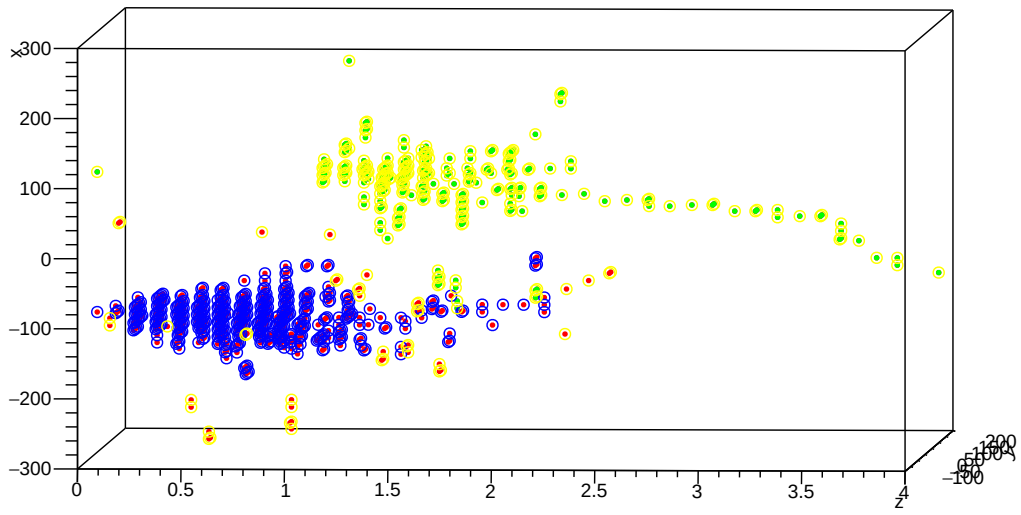
AMSTER
Algorithm with Minimum Spanning Tree for Energy Reconstruction

Julien Cheynis*, Gérald Grenier†
**Master's student at the University of Lyon, France*
†IP2I Lyon, France

July 8, 2021



Outline

Context

Reasons behind AMSTER:

- Bottom-up (Pandora, Arbor) vs top-down (AMSTER) approaches
- Generic programming for the code reusability particularly useful for the lengthy projects of particle physics

Parameters

Parameters used for the example:

- 2 particles (vertices 0 and 1) of 4 GeV each separated by 20 cm
- 8 hits (vertices 2, 3, 4, 5, 6, 7, 8 and 9) of 1 GeV each

Step 1: Data input and filling of the graph

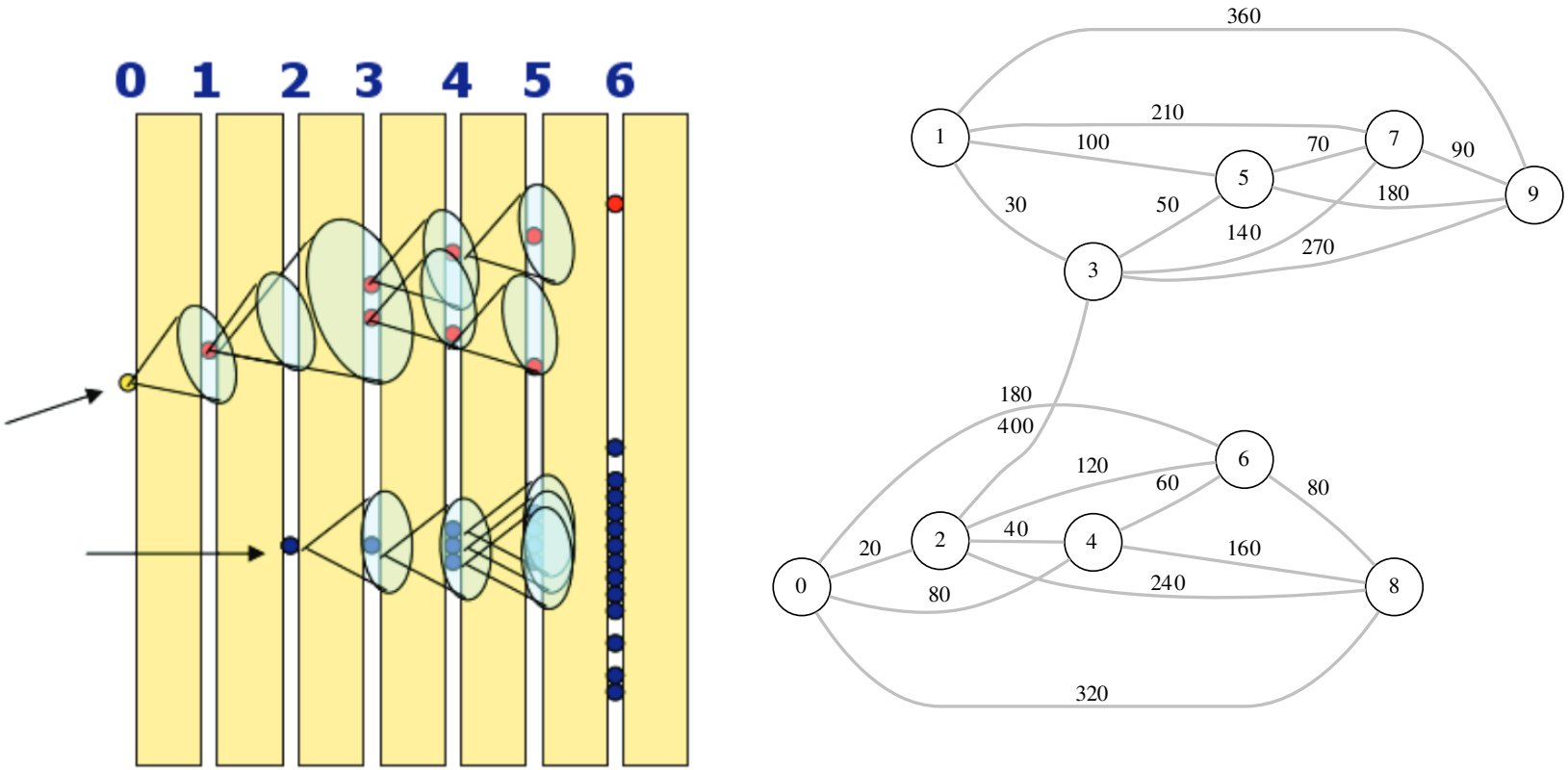


Figure: Step 1: Data input (Pandora vs AMSTER).

Step 2: Minimum spanning tree and edges ordering

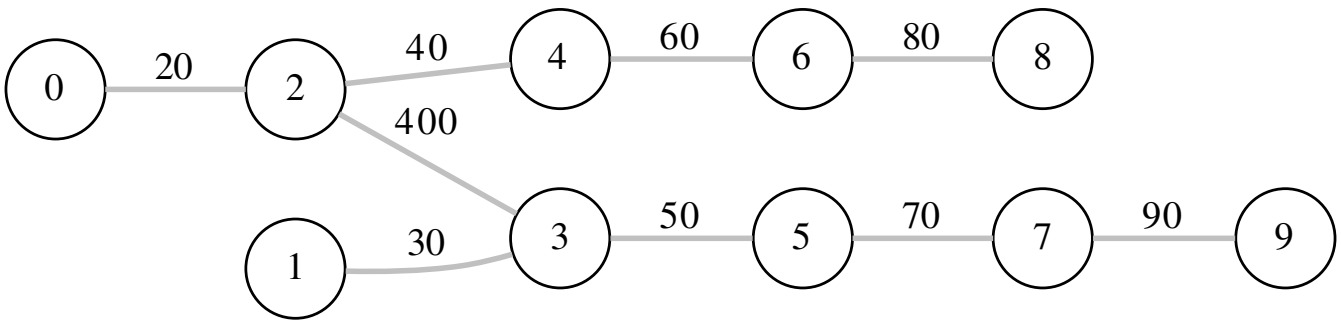


Figure: Step 2: Minimum spanning tree.

Step 3: Clustering

```
clustering(graph, list of ordered edges, deltaE)
{
   int i = 0;
   while (energy of cluster 0 > pion energy+deltaE)
   {
      if (energy of cluster 0 > pion energy)
         remove edge i
      if (energy of component 0 <= pion energy-deltaE)
         add edge i
      ++i;
   }
   return graph;
}
```

Listing 1: Clustering algorithm.

Step 3: Clustering

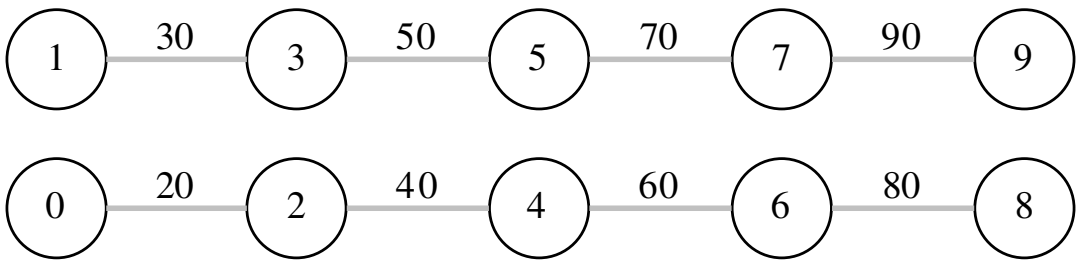


Figure: Step 3: Clustering.

Generic programming

```
1  #include <iostream>
2  template <typename T>
3  T sumvalarray(T array[], T initvalue, int size)
4  {
5     T sum = initvalue;
6     for (int i=0; i<size; ++i)
7       sum += array[i];
8     return sum;
9  }
10 int main()
11 {
12    int array1[] = {1,1,1};
13    sumvalarray<int>(array1, 0, 3);          /* = 3   */
14    double array2[] = {1.1,1.1,1.1};
15    sumvalarray<double>(array2, 0., 3);      /* = 3.3 */
16    std::string array3[] = {"a","b","c"};
17    sumvalarray<std::string>(array3, "", 3);/* = abc */
18    return 0;
19 }
```

Listing 2: Example of a function template.

Introduction to the Boost Graph Library

Graph terminology

Graph

A graph G consists of a vertex set V, an edge set E and we write $G = (V, E)$. For example $V = \{a, b, c\}$ and $E = \{(a, b), (a, c), (b, c)\}$.

Undirected

If the edges of a graph have no direction, the graph is *undirected*.

Adjacent, source, target

If a graph contains an edge (a, b), vertex a is *adjacent* to vertex b.
If the graph is directed, a is the *source* vertex and b is the *target* vertex.
If the graph is undirected, edge (a, b) is *incident* on the vertices a and b.

Introduction to the Boost Graph Library

Minimum spanning tree

Minimum spanning tree

For an undirected graph $G = (V, E)$, a spanning tree T of G is an acyclic subset of the edges $T \subseteq E$ that connects all the vertices in G. If the total weight $w(T)$ is minimized, it is called a minimum spanning tree and the number of edges is $|E| = |V| - 1$.

The total weight is the sum of the weight of the edges in T:

$$w(T) = \sum_{(a,b) \in T} w(a, b) \tag{1}$$

Introduction to the Boost Graph Library

Relevant concepts

`boost::adjacency_list` class

Template parameters: `EdgeList`, `VertexList`, `(Un)Directed`, `VertexProperties`, `EdgeProperties`.

`boost::connected_components` function

If a path exists from vertex *a* to *b*, then we say that vertex *b* is reachable from vertex *a*. A *connected component* is a group of vertices in an undirected graph that are reachable from one another.

`boost::graph_traits< Graph<E,V> >::vertex/edge_iterator`

Iterators are used to traverse the vertex set and the edge set of a graph.

Introduction to the Boost Graph Library

Minimum spanning tree algorithms

`boost::kruskal_minimum_spanning_tree` algorithm

Kruskal's algorithm starts with each vertex in a tree by itself and with no edges in the set T. The algorithm then examines each edge in the graph in order of increasing edge weight. If an edge connects two vertices in different trees, the algorithm merges the two trees into a single tree and adds the edge to T. Once all the edges have been examined, the tree T spans the graph and the tree is a minimum spanning tree.

`boost::prim_minimum_spanning_tree` algorithm

Insted of one edge at a time as in Kruskal's, Prim's algorithm grows the minimum spanning tree one vertex at a time. The basic idea of Prim's algorithm is to add vertices to the minimum spanning tree based on which of the remaining vertices shares an edge having minimum weight with any of the vertices already in the tree.

Time taken

Time complexity of the BGL functions

BGL functions	Time complexity
`boost::add_edge()`	$O(log(\|E\|/\|V\|)$
`boost::add_vertex()`	$O(\|V\|)$
`boost::remove_edge()`	$O(\|E\|)$
`boost::connected_components()`	$O(\|E\| + \|V\|)$
`boost::kruskal_minimum_spanning_tree()`	$O(\|E\|log\|E\|)$
`boost::prim_minimum_spanning_tree()`	$O(\|E\|log\|V\|)$

Table: Time complexity of the different Boost functions.

Time taken

Comparison of Kruskal's and Prim's algorithms

$\pi^-_{60\pm0.1GeV}$ 1000 events	Kruskal	Prim	Prim + optimization
$\lvert\overline{V}\rvert$	855	855	855
$\lvert\overline{E}\rvert$	378286	378286	116122
$\overline{t_{\texttt{input()}}}$ (s)	0.338102	0.335376	0.133898
$\overline{t_{\texttt{algomst()}}}$ (s)	0.965631	0.292179	0.088498
$\overline{t_{\texttt{fillmst()}}}$ (s)	0.002006	0.005928	0.003320
$\overline{t_{\texttt{sortedges()}}}$ (s)	0.001900	0.002026	0.001980
$\overline{t_{\texttt{clustering()}}}$ (s)	0.019416	0.025762	0.025177
$\overline{t_{event}}$ (s)	1.327058	0.661272	0.252875
$\overline{t_{total}}$ (s)	1327.058	661.2721	252.8747

Table: Comparison of Kruskal's and Prim's algorithms.

Time taken

Evolution of the time taken with the number of edges

π^- 1000 events	5 GeV	15 GeV	30 GeV	60 GeV
$\lvert\overline{V}\rvert$	98	271	492	855
$\lvert\overline{E}\rvert$	1928	13429	42015	116122
$\overline{t_{\texttt{input()}}}$ (s)	0.002404	0.015705	0.048482	0.133898
$\overline{t_{\texttt{prim()}}}$ (s)	0.001476	0.009635	0.030849	0.088498
$\overline{t_{\texttt{fillmst()}}}$ (s)	0.000238	0.000742	0.001591	0.003320
$\overline{t_{\texttt{sortedges()}}}$ (s)	0.000221	0.000616	0.001132	0.001980
$\overline{t_{\texttt{clustering()}}}$ (s)	0.001350	0.004942	0.012279	0.025177
$\overline{t_{event}}$ (s)	0.005689	0.031640	0.094334	0.252875
$\overline{t_{total}}$ (s)	5.688773	31.64040	94.33401	252.8747

Table: Evolution of the time taken with the number of edges.

Time taken

Evolution of the time taken with the number of edges

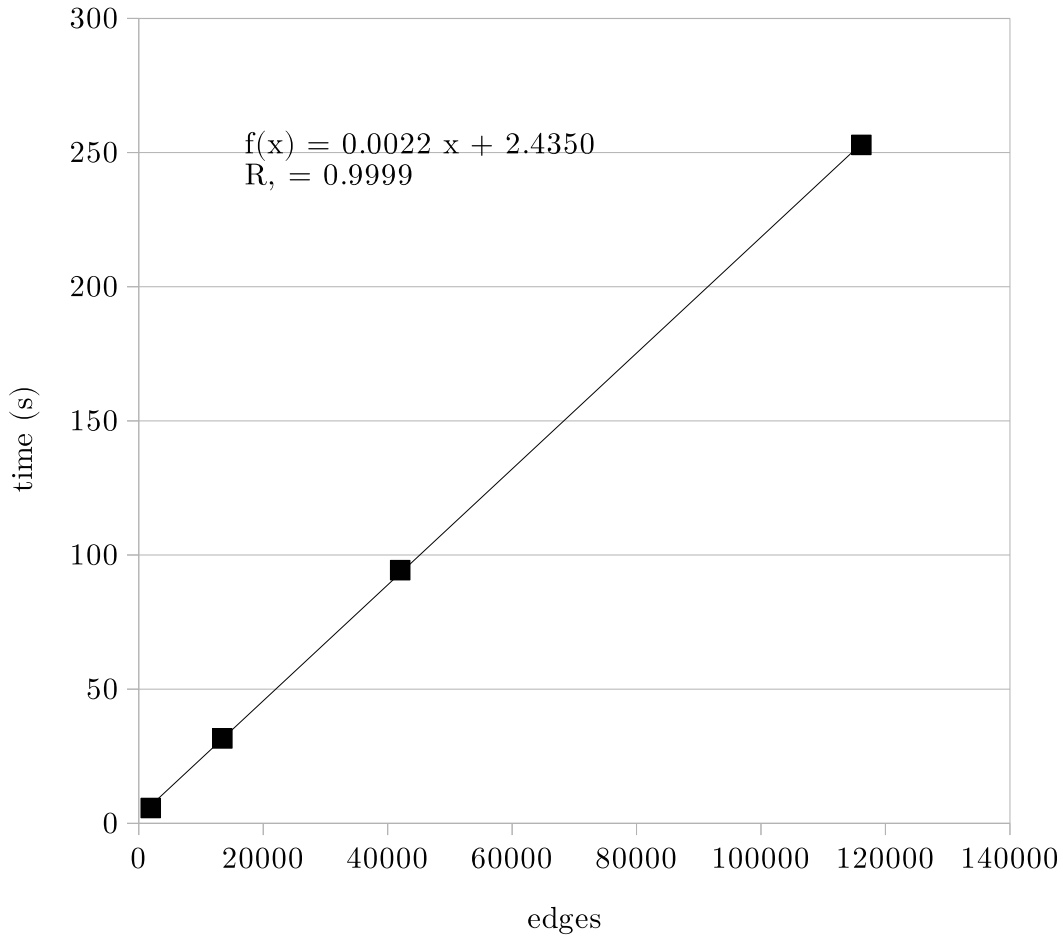


Figure: Evolution of the time taken with the number of edges.

Energy reconstruction
Formula used in SDHCAL

The energy in SDHCAL is given by

$$E_{reco} = \alpha(N_{hit}) \cdot N_{hit1} + \beta(N_{hit}) \cdot N_{hit2} + \gamma(N_{hit}) \cdot N_{hit3} \tag{2}$$

where

$$\alpha(N_{hit}) = \alpha_0 + \alpha_1 \cdot N_{hit} + \alpha_2 \cdot N_{hit}^2 \tag{3a}$$
$$\beta(N_{hit}) = \beta_0 + \beta_1 \cdot N_{hit} + \beta_2 \cdot N_{hit}^2 \tag{3b}$$
$$\gamma(N_{hit}) = \gamma_0 + \gamma_1 \cdot N_{hit} + \gamma_2 \cdot N_{hit}^2 \tag{3c}$$

with α_0, α_1, α_2, β_0, β_1, β_2, γ_0, γ_1, γ_2 determined with

$$\chi^2 = \frac{1}{N_{event}} \cdot \sum_{i=1}^{N_{event}} \frac{(E_{beam,i} - E_{reco,i})^2}{E_{beam,i}} \tag{4}$$

where i is the particle considered.

Energy reconstruction

Energy reconstructed in function of the energy and the calibration

1000 events π^- (GeV)	1.140	5.140	15.140	30.140	60.140
$\overline{E_{reco1}}$ (GeV)	1.374	5.183	15.568	30.779	58.196
$\overline{E_{reco2}}$ (GeV)	0.953	4.158	13.855	28.196	53.562
$\overline{E_{reco1}}/E_{init}$	1.205	1.008	1.028	1.021	0.968

Table: Energy reconstructed in function of the energy and the calibration.

Separation of two particles

Efficiency and purity of a cluster

We define the efficiency

$$\varepsilon = \frac{N_{hits,right}}{N_{hits,particle}} \tag{5}$$

which is the number of hits in the right cluster over the true number of hits of the particle.

We also define the purity

$$\rho = \frac{N_{hits,right}}{N_{hits,reco}} \tag{6}$$

which is the number of hits in the right cluster over the number of hits in the cluster of the reconstructed particle.

Separation of two particles

Typical event where the clustering can be improved

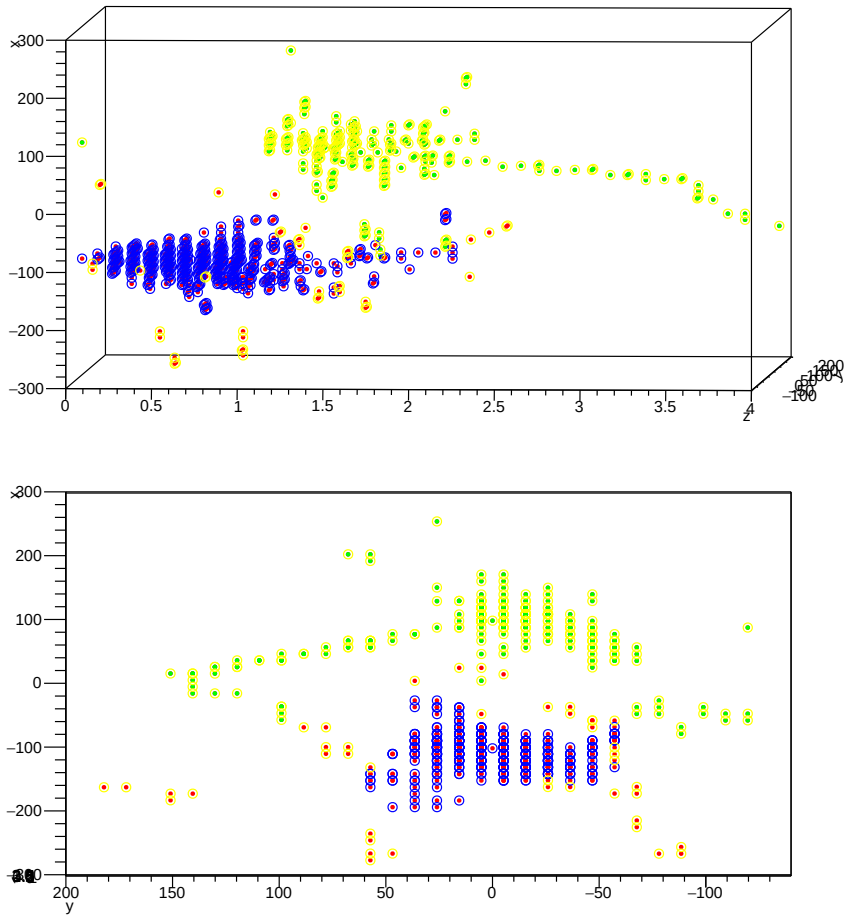


Figure: Event of a π^- (red) and a K^0 (green) in two clusters (blue and yellow). 3D view (top), (x, y) view (bottom). $d = 20$ cm, $\varepsilon = 0.91$, $\rho = 1$, $E^{\pi^-}_{reco} = 30.1844$ GeV, $E^{\pi^-}_{init} = 30.1396$ GeV.

Separation of two particles

Same event after improvements

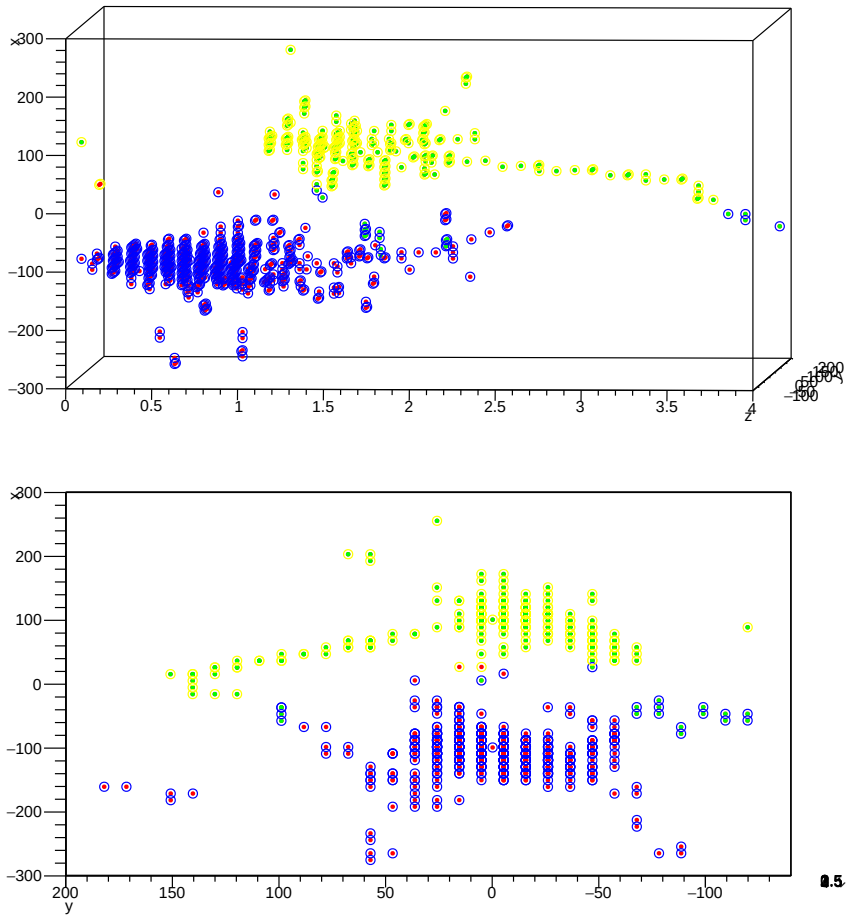


Figure: Event of a π^- (red) and a K^0 (green) in two clusters (blue and yellow). 3D view (top), (x, y) view (bottom). $d = 20$ cm, $\varepsilon = 0.99$, $\rho = 0.96$, $E^{\pi^-}_{reco} = 34.3208$ GeV, $E^{\pi^-}_{init} = 30.1396$ GeV.

Separation of two particles

One of the worst result

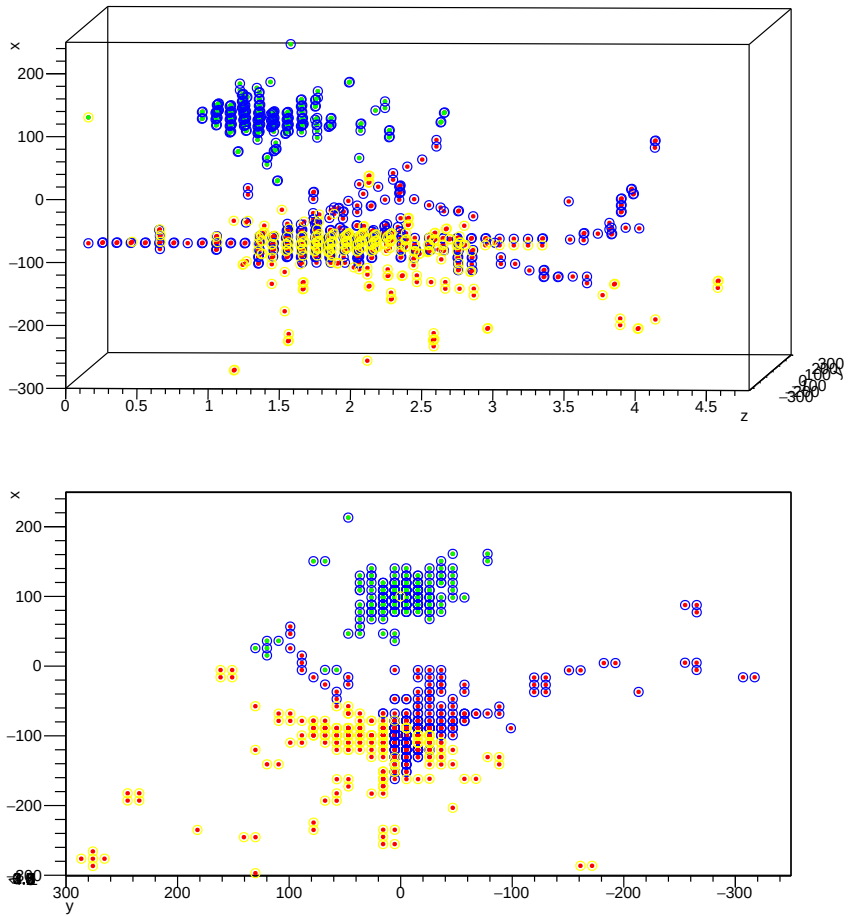


Figure: Event of a π^- (red) and a K^0 (green) in two clusters (blue and yellow). 3D view (top), (x, y) view (bottom). $d = 20$ cm, $\varepsilon = 0.52$, $\rho = 0.59$, $E^{\pi^-}_{reco} = 34.4176$ GeV, $E^{\pi^-}_{init} = 30.1396$ GeV, $E_{leaked} = 0.66$ GeV.

Time taken

Purity vs minimum purity

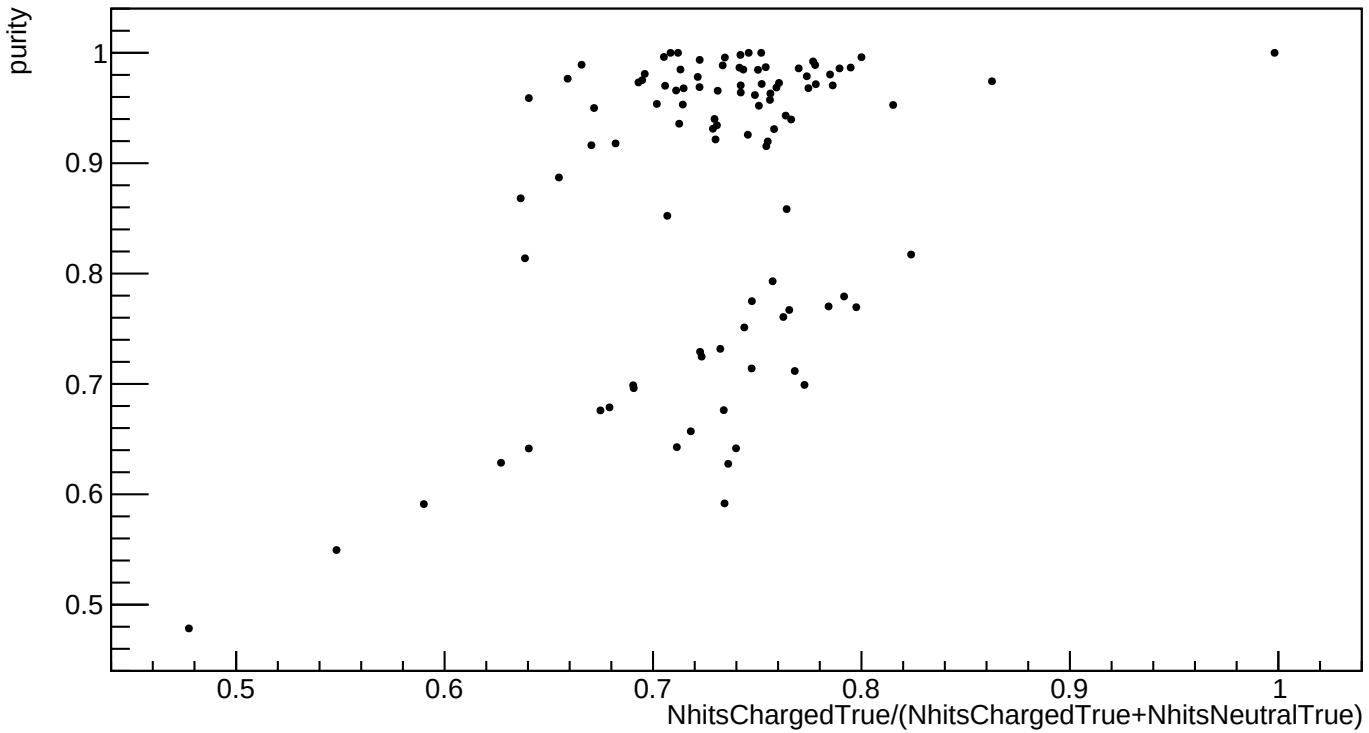


Figure: Purity vs minimum purity.

Outlooks

To improve the speed and quality of AMSTER we need to improve

- how to create even less the edges
- how to set the best ΔE

How can AMSTER evolve in the near future?

- With less edges, AMSTER would run faster and with information about the physics, the efficiency and the purity should improve.
- With the development of better TPC that will measure time at the picosecond scale (calorimeter hits with $t \pm 30$ ps), AMSTER could include the causality of special relativity when creating edges.

References


J. Siek, L.-Q. Lie, A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, (2002).
`https://www.boost.org/`


Stroustrup, B. *The C++ programming language*. Fourth Edition, Addison-Wesley, (2013).


F. Gaede, H. Vogt. *LCIO - Users manual*. Version 2-14-01, (2020).
`https://github.com/iLCSoft/LCIO/blob/master/doc/manual.pdf`


ILC collaboration. *ILC Technical Design Report*. Volume 1 and 4, (2013).
`https://linearcollider.org/technical-design-report/`


CALICE collaboration. *First results of the CALICE SDHCAL Technological Prototype*. CALICE Analysis note CAN-037, (2012).
`https://twiki.cern.ch/twiki/pub/CALICE/CaliceAnalysisNotes/CAN-037.pdf`


R. Ete. *Développement d'un algorithme de suivi de particules pour l'ILC : outils de surveillance de qualité de données en ligne*. Université de Lyon, (2017).
`https://hal.archives-ouvertes.fr/tel-01579761`