The SegmentNetworkProducerModule - a draft for discussion

Preliminary draft!
Do not circulate!

Jakob Lettenbichler

June 12, 2015

Contents

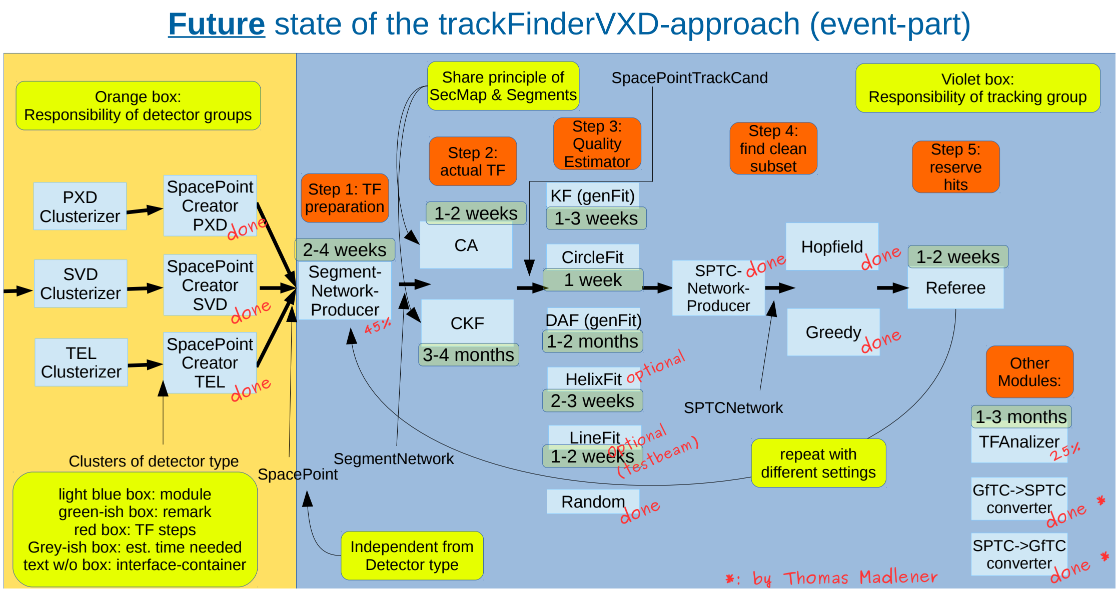

Figure 1.1

1 Introduction

1.1 Overall picture

Figure 1.1 shows the current state (as of May 2015) of the redesign-process.

Next proposed steps are:

- done: ~~SPTCNetworkProducer~~
- done: ~~QualityEstimatorRandom~~
- done: ~~TrackSetEvaluatorGreedy~~
- done: ~~TrackSetEvaluatorHopfield~~
- FilterCalculator/SecMapTrainerBase (- 50% done)
  - add newFilters, remaster as much as possible without breaking the interface to the `FilterCalculatorModule` which will still be used for the old `VXDTF`
  - fork, newModule: `SecMapTrainerBase`

  - change to support for new `SectorMap`
  - estimation: 2-4 weeks

- ExportSecMap/RawSecMapMerger (- 0% done)
  - fork newModule: `RawSecMapMerger`
  - implement new `SectorMap`-design (2-, 3- & 4-hit maps)
  - merge raw files
  - take care of RAM-consumption (depending on Thomas Madleners future plans and design of `SectorMap`)
  - estimation: 2-4 weeks
- SectorMapTuner (runOnly - 0% done)
  - load static `SectorMap` from root file
  - apply tuning parameters
  - make the map accessible on `StoreArray`
  - estimation: 1-2 weeks
- SegmentNetworkProducer (- 45% done) $\leftarrow$ *I am here*
  - access `SectorMap` on `StoreArray`
  - add virtual interaction point and find sectors for given `SpacePoints` $\rightarrow$ if found: `TrackNodes` $\leftrightarrow$ `SpacePoints`
  - find compatible sectors
  - apply segFinder-Filters $\rightarrow$ segments connecting `TrackNodes`
  - apply nbFinder-Filters $\rightarrow$ friends connecting segments
  - store `SegmentNetwork` to `StoreArray`
  - estimation: 2-4 weeks
- TrackFinderVXDCelloMat (- 0% done)
  - encapsulate old algorithm
  - load `SegmentNetwork`
  - apply cellular automaton on network (interface to be designed)
  - collect TCs as `SpacePointTrackCands`
  - calculate seed and store SPTCs on `StoreArray`
  - estimation: 1-2 weeks

- QualityEstimatorCircleFit (- 0% done)
  - Approach A: convert interface from VXDTFTrackCanditates to `SpacePointTrackCands`
  - estimation approach A: 1 week
  - Approach B: use existing implementation from the CDC-guys
  - estimation approach B: 1-3 weeks
- SpacePointReferee (- 0% done)
  - best x% (parameter) of TCs reserve the SP/Clusters (parameter) for further iterations
  - not completely clear how to store relevant info (not thought about that yet in detail)
  - estimation 1-2 weeks
- TrackFinderVXDComboKalFit (- 0% done) $\leftarrow$ *Outdated?* $\rightarrow$ *to be decided!*
  - load `SegmentNetwork` from `StoreArray`
  - for each allowed treeSeed, extrapolate to each sensor allowed by sectorCombi
  - collect x (parameter, $1 \leq x \leq 5$) best fits and store for further extrapolations
  - final trees of `SpacePointTrackCands`: store best y (parameter, $1 \leq y \leq 5$) TCs of single trees
  - on-the-fly (`genFit::Track`)seed-parameter estimation
  - estimation 3-4 months
- QualityEstimatorKalmanFilter (- 0% done) $\leftarrow$ *Outdated?* $\rightarrow$ *to be decided!*
  - take TC and apply seed needed for fitting
  - bad-ass-way: convert `SpacePointTrackCands` to `genFit::Track` before, apply old interface
  - estimation bad-ass: 1 week
  - efficient-way: start with that module after finishing `TrackFinderVXDComboKalFit`, use as much as possible from that module
  - estimation efficient: 1 week
  - correct way: make `genfit` compatible (how?), new interface to be written
  - estimation correct: 3+ weeks
- QualityEstimatorDAF (- 0% done)
  - take tree of TCs and determine the best one using DAF

  - mostly piping into `genfit`
  - open question is how to do the interface (synergies with `TrackFinderVXDComboKalFit` and `QualityEstimatorKalmanFilter` apparent)
  - estimation: 1-2 months

- TrackFinderVXDAnalizer, after finishing TrackSetEvaluatorModules mixed all in between (- 25% done)
  - fork, newModule: `TrackFinderVXDAnalizer`
  - analysis module compatible with new design
  - collect data from modules and store specific info to root files (eg efficiency vs pT, acceptance rate of dist3D for good combis...)
  - correct implementation:
    * heavily depending on observers, design only fixed yet for seg- and nbFinder (== **`SegmentNetworkProducerModule`**). completely unclear for everything else
    * problem of correctly linking data in an oo-way for not to lose info too early
  - actual implementation: on-the fly, minimal effort, only when needed
  - estimation: difficult, since many unsolved questions, 4-11 weeks

Figure 1.2 shows the time consumption for different cases for different parts of the old VXDTFmodule. It indicates that the segFinder-part of the code may be a bottleneck especially for high occupancy cases. It is therefore important to consider this for the **`SegmentNetworkProducerModule`** .

1.2 Motivation

The **`SegmentNetworkProducerModule`** is a central cornerstone of the architecture of the VXD TrackFinder of the Belle2-experiment. Its task is to provide a versatile container which can (and will) be used by completely different track finding algorithms like a Cellular Automaton (CA), a Deterministic Annealing Filter (DAF) or a Combinatorial Kalman Filter (CKF). The creation of the container should keep the overhead low and has to effectively reduce the combinatorial problem omnipresent for local and semi-local tracking algorithms. The structure of the **`SegmentNetworkProducerModule`** is to apply filters of different complexity levels starting with so-called "one-hit-filters", which will be discussed briefly in one of the following paragraphs. The overall design of the **`SegmentNetworkProducerModule`** allows to apply a wide range of filters, which are

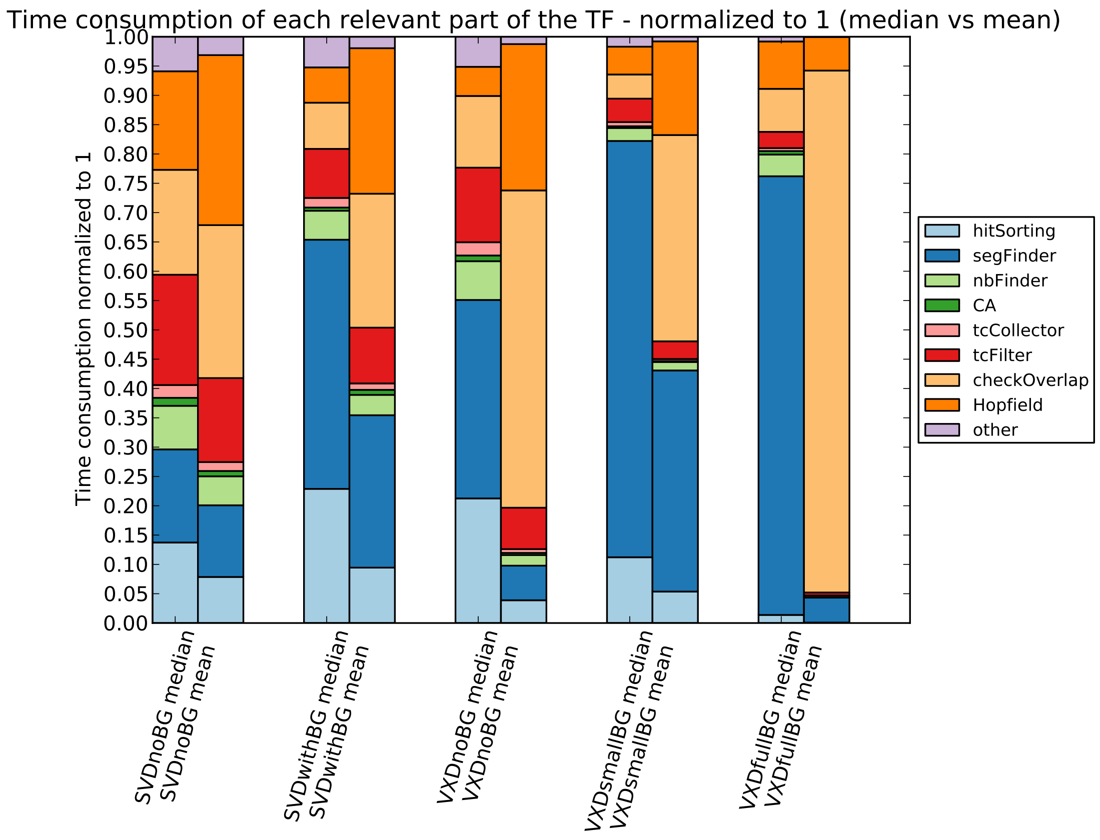

Figure 1.2

all accessed using unified interfaces. The access to the filters used is provided by the sectorMap and allows the utilization of purely geometrical filters like Distance3D or more sophisticated ones like fully trained neural networks. The output will be a network which links hits or a chain of hits called segments. These networks contain a fair number of directed graphs linking segments which can be used by the tracking algorithms mentioned above.

One hit filters are filters which can be applied on single hits and are therefore very good for reducing combinatorics since the total combinatorial problem is dominated by the possible number of combination of useful hits. Therefore these filters are the earliest possible point where filtering makes sense (this is of course a simplified view since the digitizers, clusterizers and cluster-combiners already filter their input too). In our case these filters are applied by finding the correct sector for a given hit. This step scales linearly with the number of hits and is therefore no bottleneck for the combinatorial problem. The basic idea is to have a previously trained sectorMap which stores the geometrical compatibility of neighbouring sectors, which are a subdivision of sensors of the tracking detector. This network of sectors therefore dictates which hits can be combined since only neighbouring sectors are allowed to be tested for compatible hits later-on. This results in 3 different cases possible for a hit after this filter stage:

1. There was no compatible sector found for given hit: hit is neglected
2. There was a compatible sector, but that sector has no hits in its compatible neighbours: hit is neglected
3. There was a compatible sector and at least one of its neighboring sectors got hits too: hit is accepted.

Using an approach where no sectorMap is included, this would mean that one has to combine each hit with any other hit "near" of him (e.g neighboring layers, the same layers if overlapping parts exist or even next-to-neighboring layers if one has to consider broken sensors). Even if the tests for a pair of hits are fast (like Distance3D), the combinatorial issue grows with the second power of the number of hits and therefore can be a bottleneck for reconstruction.

1.2.1 Why one should use multi-hit-segments

In the last few years the CA as a main driver behind the segment-idea evolved in manifold directions. One of the major developments was to introduce segments containing more than the classical "two-hit-segments" introduced by Kisel (Cite here). This is an important step to allow more fine-grained filtering of bad combinations of segments. The reason for that is that two-hit-segments in the CA-concept form neighborhoods with other two-hit-segments, which are then filtered by filters using the 3 hits of two neighboring segments (two neighboring segments share one hit). This means that the most complex filters one can apply can not use more information provided as of those 3 hits. A typical example for such a filter is Angle3D. Filters using 4 or more hits can be applied on 3 connected segments, but the Information can not be stored when staying at the two-hit-segment level since two pairs of connected segments - which share the middle segment - can both be valid neighbours but the overall combination could be invalid when applying a 4-hit-filter like zigzag. This information of the "neighbour of my neighbour" can not preserved this way. Of course one could start collecting all valid paths through that network and then apply these more sophisticated filters later-on, but in that case paths would be collected which would not survive that step. Depending on the occupancy of the events and therefore the combinatorial issue, this would mean a lot of needless work. Therefore using 3-hit-segments would allow to map the results of 4-hit-filters and longer segments could accept even better filters (starting with about 4 hits (3 hits are sufficient, but allow only one degree of freedom), the Kalman filter would be one of these filters.

For cases when there are no full hits (consisting of a 2D-pixel or two 1D-strips combined) available, this problem is even more severe, since then even comparably simple filters run into that problem.

1.2.2 Why we didn't

Although the internal design of the `SegmentNetworkProducerModule` supports multi-hit-segments. They are not used in the end. Several reasons contributed to this decision:

- The more hits used per segment, the more layers needed for that step. It is clear that one can not build 10-hit-segments when having only 3 layers and therefore simply not enough hits for such segments. Belle2 can provide on-line-information of the SVD, which has 4 layers of double-sided strip-sensors.
- The longer the segments, the more difficult it gets to deal with non-standard cases. Of course having 4 layers and expecting to have 4 hit-track-candidates strongly support using 4-hit-segments, since that would provide completely drafted paths which are easily collected as trackCandidates without further steps. But having the possibility of missing hits, broken sensors or simply two hits in one layer due to the overlapping parts means that one can have tracks with 3-8 hits in our 4 layer-system. Requesting longer segments would automatically filter those shorter stubs. Although one could work with bypasses, this would increase again the amount of work and would minimize the boosting effect of multi-hit-segments.
- Next to these theoretical issues mentioned above, the hardest support for our decision can be seen in Figure 1.2, which clearly states that the process of finding pairs of hits ($\rightarrow$ segments) is for high occupancy cases dominating and therefore the time gain of higher order filters comparatively small. Therefore one should invest the effort needed for implementing multi-hit-cells into improved pre-selection via one- and two-hit-filters.

2 Associated Classes

2.1 DirectedNodeNetwork

The heart of the `SegmentNetworkProducerModule` is a container which stores the network in an easy-to-use and efficient way. The elements of the network are called `DirectedNode` which are linked to other `DirectedNodes`. These nodes are templated and have only some very loose restrictions to the objects stored in them:

1. The stored class of template `EntryType` has to have an `== operator` defined
2. All nodes of the network store the same `EntryType`

Which effectively means that practically any class can be used to be stored as nodes in the `DirectedNodeNetwork`. The prerequisites for the `DirectedNodeNetwork` itself are the following:

1. All nodes store the same `EntryType` and links can not carry any extra information (except the direction: inner and outer).
2. `DirectedNodes` can not be deactivated afterwards and links not deleted (changes afterwards are not needed for the `SegmentNetworkProducerModule` and would only complicate things).
3. Can be walked through in a directed way (each `DirectedNode` knows its inner and outer partners) or in an undirected way.
4. Inner and outer end-points of the networks are always automatically updated when filling the network.
5. There are three different ways to add nodes to the network:
   a) Add a pair of nodes, where one is the outer and one is the inner node. Links are established in both directions.
   b) Add another inner node to the last outer node passed.
   c) Add another outer node to the last inner node passed.
6. there can not be any nodes in the network which have no links at all.

7. Adding (and linking) the same pair of entries twice results in a warning.
8. The user has to take care of the lifetime of objects to be linked in the network since it only stores a reference of the objects passed.

Of the 3 ways to add Nodes, the latter two are there only to speed up the interface, since adding pairs of nodes all the time would lead to the situation that nodes will be re-added several times.

The code for the `DirectedNodeNetwork` consists of two Classes. One is the `DirectedNode`, which handles the storing and access of the carried object and keeps track of its inner and outer neighbours. The other one is the `DirectedNodeNetwork` itself, which adds, stores and connects all the nodes and keeps track of the inner and outer ends of the network. The code for that is mostly done (tests should be written for some of the most important member functions) and covers all the features needed. The comments in the code describe in detail what is happening there.

2.1.1 DirectedNode

The following code snippet describes the `DirectedNode`.

Its full implementation can be found in `r18777` in:
`../tracking/trackFindingVXD/segmentNetwork/include/DirectedNodeNetwork.h`.

```
#include <framework/logging/Logger.h>
#include <vector>

/** the node-class.
* carries an instance of a class which shall be woven into a network. */
template<class EntryType>
class DirectedNode {

/** only the network can create DirectedNodes and link them: */
  friend class DirectedNodeNetwork;

protected:
/** ************************ DATA MEMBERS  */

/** entry can be of any type, DirectedNode is just the carrier */
  EntryType& m_entry;

/** carries all links to inner nodes */
  std::vector<DirectedNode<EntryType>*> m_innerNodes;

/** carries all links to outer nodes */
  std::vector<DirectedNode<EntryType>*> m_outerNodes;

/** is the index position of this node in the network */
```

```
  unsigned int m_index;

/** ************************* CONSTRUCTORS  */

/** protected constructor. Accepts an unreplaceable entry */
  DirectedNode(EntryType& entry, unsigned int index) :
  m_entry(entry),
  m_index(index) {}

/** ************************* INTERNAL MEMBER FUNCTIONS */

/** adds new links to the inward direction */
  void addInnerNode(DirectedNode<EntryType>* newNode)
  { m_innerNodes.push_back(newNode); }

/** adds new links to the outward direction */
  void addOuterNode(DirectedNode<EntryType>* newNode)
  { m_outerNodes.push_back(newNode); }

/** returns the index position of this node in the network */
  unsigned int getIndex() const { return m_index; }

public:
/** ************************* OPERATORS */

/** == -operator - compares if two nodes are identical */
  inline bool operator == (const DirectedNode& b) const
  { return (this->getEntry() == b.getEntry()); }

/** != -operator - compares if two nodes are not identical */
  inline bool operator != (const DirectedNode& b) const
  { return !(this->getEntry() == b.getEntry()); }

/** == -operator.
* true if entry passed is identical with the one
* linked in this node */
  inline bool operator == (const EntryType& b) const
  { return (this->getEntry() == b); }

/** == -operator.
* true if entry passed is not identical with the one
* linked in this node */
  inline bool operator != (const EntryType& b) const
  { return !(this->getEntry() == b); }

/** ************************* PUBLIC MEMBER FUNCTIONS */

/** returns links to all inner nodes attached to this one */
```

```
  std::vector<DirectedNode<EntryType>*>& getInnerNodes()
  { return m_innerNodes; }

/** returns links to all outer nodes attached to this one */
  std::vector<DirectedNode<EntryType>*>& getOuterNodes()
  { return m_outerNodes; }

/** allows access to stored entry */
  EntryType& getEntry() { return m_entry; }

/** allows access to stored entry */
  EntryType* getEntry() { return &m_entry; }

/** const access to stored entry for external operator overload */
    const EntryType& getConstEntry() const { return m_entry; }

/** returns Pointer to this node */
  DirectedNode<EntryType>* getPtr() { return this; }
};
```

As one can see, the `DirectedNode` is strongly coupled with the `DirectedNodeNetwork`, since the constructor is protected and setter functions too. Only the getter-functions - allowing access to the object carried - are public and free to use for the user. The operators are overloaded to allow directly comparing the objects carried. If an object has already got a `DirectedNode`, that object will not be added again.

2.1.2 DirectedNodeNetwork

The `DirectedNodeNetwork`-class is more complex since it has to handle the cases if nodes were already added. The requirements for this class is listed in 2.1, which are met with the design listed down below. Although the main design stands, there is still room for speed-optimization in that class. This is then a matter for forthcoming refactoring- and optimization steps (to mention one: switch from classic to `unique_ptr`). Its full implementation can be found in `r18777` in:
`../tracking/trackFindingVXD/segmentNetwork/include/DirectedNodeNetwork.h`. Here some of the less relevant functions are replaced by small descriptions which explain their intended behavior. The missing code is then marked with `[...]`.

```
#include <path/to/DirectedNode.h>
#include <framework/logging/Logger.h>
#include <vector>
#include <algorithm>    // std::find

using namespace std, Belle2; // just for the example here

/** A network container where the nodes can carry any entryType. */
```

```
template<class EntryType>
class DirectedNodeNetwork {
protected:
/** ************************* DATA MEMBERS  */

/** carries all nodes */
  vector<DirectedNode<EntryType>* > m_nodes;

/** temporal storage for last outer node added, used for speed-up */
  DirectedNode<EntryType>* m_lastOuterNode;

/** temporal storage for last inner node added, used for speed-up */
  DirectedNode<EntryType>* m_lastInnerNode;

/** keeps track of current outerEnds (nodes without outerNodes).
* entries are the indices of the nodes which are an outermost node */
  vector<unsigned int> m_outerEnds;

/** keeps track of current innerEnds (nodes without innerNodes).
* entries are the indices of the nodes which are an innermost node */
  vector<unsigned int> m_innerEnds;

/** *********************** INTERNAL MEMBER FUNCTIONS  */

/** internal function for adding Nodes */
  DirectedNode<EntryType>& addNode(EntryType& entry)
  {
    unsigned int index = m_nodes.size();
    m_nodes.push_back(new DirectedNode<EntryType>(entry, index) );
    return *m_nodes[index];
  }

/** checks if given entry is already in given vector */
  template<class T>
  auto isInVector(T& entry, vector<T>& aVector) {[...]}

/** checks if given entry is already in the network. */
  template<class T>
  auto isInNetwork(T& entry) {[...]}


/** links nodes to each other.
* returns true if everything went well,
* returns false, if not  */
  static bool linkNodes(
    DirectedNode<EntryType>& outerNode,
    DirectedNode<EntryType>& innerNode)
  {
    auto outerPos = find(
        outerNode.getInnerNodes().begin(),
        outerNode.getInnerNodes().end(),
        &innerNode);
    auto innerPos = find(
        innerNode.getOuterNodes().begin(),
        innerNode.getOuterNodes().end(),
        &outerNode);

    if (outerPos == outerNode.getInnerNodes().end() or
      innerPos == innerNode.getOuterNodes().end()
      { return false; }

    outerNode.addInnerNode(&innerNode);
    innerNode.addOuterNode(&outerNode);
    return true;
  }

/** adds newNode as a new end to the endVector
* and replaces old one if necessary */
  bool updateNetworkEnd(
    DirectedNode<EntryType>& oldNode,
    DirectedNode<EntryType>& newNode,
    vector<unsigned int> endVector)
  {
    auto iter = isInVector(oldNode.getIndex(), endVector);
    if (iter != endVector.end()) {
      *iter = newNode.getIndex();
      return true;
    }
    endVector.push_back(newNode.getIndex());
    return false;
  }

/** to a given existing node, the newEntry will be added to the network
* if it was not there yet and
* they will be linked (returning true) if they weren't linked yet.
* returns false if linking didn't work */
  bool addToExistingNode(
    DirectedNode<EntryType>* existingNode,
    EntryType& newEntry,
    bool newIsInner)
  {
    // check if entry is already in network.
    auto nodeIter = isInNetwork(newEntry);

    DirectedNode<EntryType>* outerNode = NULL;
    DirectedNode<EntryType>* innerNode = NULL;

    // assign nodePointers, create new nodes if needed:
    if (newIsInner) {
      outerNode = existingNode;
      if (nodeIter == m_nodes.rend()) { // new node was not in network yet
        innerNode = &addNode(newEntry);

        auto iterPos = find(m_innerEnds.begin(),
                            m_innerEnds.end(),
                            outerNode->getIndex());
        if (iterPos != m_innerEnds.end()) { *iterPos = innerNode->getIndex(); }
        else { m_innerEnds.push_back(innerNode->getIndex()); }
      } else { // new node was not new after all:
      innerNode = *nodeIter;
      }
    } else { // outerNode is new, not innerNode:
      innerNode = existingNode;

      if (nodeIter == m_nodes.rend()) { // new node was not in network yet
        outerNode = &addNode(newEntry);

        auto iterPos = find(m_outerEnds.begin(),
                            m_outerEnds.end(),
                            innerNode->getIndex());
        if (iterPos != m_outerEnds.end()) { *iterPos = outerNode->getIndex(); }
        else { m_outerEnds.push_back(outerNode->getIndex()); }
      } else { // outerNode is new, not innerNode:
      outerNode = *nodeIter;
      }
    }

    m_lastInnerNode = innerNode;
    m_lastOuterNode = outerNode;
    return linkNodes(*outerNode, *innerNode);
  }

public:

/** ************************ CONSTRUCTOR/DESTRUCTOR */

/** standard constructor for ROOT IO */
  DirectedNodeNetwork() :
    m_lastOuterNode(NULL),
    m_lastInnerNode(NULL) {}

/** destructor taking care of cleaning up the pointer-mess  */
  ~DirectedNodeNetwork()
  {
    for (DirectedNode<EntryType>* nodePointer : m_nodes) {
      delete nodePointer;
    }
    m_nodes.clear();
  }

/** ************************ PUBLIC MEMBER FUNCTIONS  */

/** to the last outerNode added, another innerNode will be attached */
  void addInnerToLastOuterNode(EntryType& innerEntry)
  { return addToExistingNode(m_lastOuterNode, innerEntry, true); }

/** to the last innerNode added, another outerNode will be attached */
  void addOuterToLastInnerNode(EntryType& outerEntry)
  { return addToExistingNode(m_lastInnerNode, outerEntry, false); }

/** takes two entries and weaves them into the network */
  void linkTheseEntries(EntryType& outerEntry, EntryType& innerEntry)
  {
    // check if entries are already in network.
    auto outerNodeIter = isInVector(outerEntry, m_nodes);
    auto innerNodeIter = isInVector(outerEntry, m_nodes);

    /* case 1: none of the entries are added yet:
    * create nodes for both and link with each other,
    * where outerEntry will be carried by outer node
    * and inner entry by inner node
    * outerNode will be added to outerEnds,
    * and innerNode will be added to innerEnds. */
    if ( outerNodeIter == m_nodes.rend() and
      innerNodeIter == m_nodes.rend()) {
      DirectedNode<EntryType>& newOuterNode = addNode(outerEntry);
      DirectedNode<EntryType>& newInnerNode = addNode(innerEntry);

      linkNodes(newOuterNode, newInnerNode);
      m_lastInnerNode = &newInnerNode;
      m_lastOuterNode = &newOuterNode;

      m_outerEnds.push_back(newOuterNode.getIndex());
      m_innerEnds.push_back(newInnerNode.getIndex());
      return;
    }

    /* case 2: the outerEntry was already in the network,
    * but not innerEntry:
    * add new node(innerEntry) to network
    * add innerNode to existing outerNode
    * add innerNode to innerEnds,
    * if outerNode was in innerEnds before,
    * replace old one with new innerNode. */
    if ( outerNodeIter != m_nodes.rend() and
         innerNodeIter == m_nodes.rend()) {
      DirectedNode<EntryType>& outerNode = **outerNodeIter;
      DirectedNode<EntryType>& newInnerNode = addNode(innerEntry);

      linkNodes(outerNode, newInnerNode);
      m_lastInnerNode = &newInnerNode;
      m_lastOuterNode = &newOuterNode;

      updateNetworkEnd(outerNode, newInnerNode, m_innerEnds);

      return;
    }

    /* case 3: outerNode was not there yet, but innerEntry was:
    * add new node(outerEntry) to network
    * add outerNode to existing innerNode
    * add outerNode to outerEnds, if innerNode was in outerEnds before,
    * replace old one with new outerNode */
    if ( outerNodeIter == m_nodes.rend() and
         innerNodeIter != m_nodes.rend()) {
      DirectedNode<EntryType>& newOuterNode = addNode(outerEntry);
      DirectedNode<EntryType>& innerNode = *innerNodeIter;

      linkNodes(newOuterNode, innerNode);
      m_lastInnerNode = &newInnerNode;
      m_lastOuterNode = &newOuterNode;

      updateNetworkEnd(innerNode, newOuterNode, m_outerEnds);

      return;
    }

    /** case 4: both are already in the network ... */
    DirectedNode<EntryType>& outerNode = **outerNodeIter;
    bool wasInnermostNode =
      (outerNode.getInnerNodes().empty()) ? true : false;
    DirectedNode<EntryType>& innerNode = **innerNodeIter;
    bool wasOutermostNode =
      (innerNode.getOuterNodes().empty()) ? true : false;

    bool wasSuccessful = linkNodes(outerNode, innerNode);

    /* case 4A: ... but were not linked to each other yet:
    * add innerNode to existing outerNode
    * add outerNode to existing innerNode
```

Preliminary draft -- do not circulate!

```
    * if innerNode was in outerEnds before,
    *  replace old one with new outerNode
    * if outerNode was in innerEnds before,
    *  replace old one with new innerNode */
    if (wasSuccessful) {
      updateNetworkEnd(outerNode, newInnerNode, m_innerEnds);
      updateNetworkEnd(innerNode, newOuterNode, m_outerEnds);

      return;
    }

    /** case 4B: both are already there and already linked: */
    B2ERROR("outerEntry␣and␣innerEntry␣were␣already␣in␣the␣network␣"<<
      "␣and␣were␣already␣connected!")
  }

/** returns all nodes which have no outer nodes (but inner ones)
* and therefore are outer ends of the network */
  vector<DirectedNode<EntryType>*> getOuterEnds() {[...]}

/** returns all nodes which have no inner nodes (but outer ones)
* and therefore are inner ends of the network */
  vector<DirectedNode<EntryType>*> getInnerEnds() {[...]}

/** returns pointer to the node with given entry. Returns NULL if not found */
  DirectedNode<EntryType>* getNode(EntryType& toBeFound) {[...]}

/** returns pointer to the node with given index. Returns NULL if not found */
  DirectedNode<EntryType>* getNodeWithIndex(unsigned int index) {[...]}

/** returns pointer to the last outer node added. */
  DirectedNode<EntryType>* getLastOuterNode() { return m_lastOuterNode; }

/** returns pointer to the last inner node added. */
  DirectedNode<EntryType>* getLastInnerNode() { return m_lastInnerNode; }

/** returns all nodes of the network */
  vector<DirectedNode<EntryType> >& getNodes() { return m_nodes; }

/** returns number of nodes to be found in the network */
  unsigned int size() const { return m_nodes.size(); }
};
```

The design of the `SegmentNetworkProducerModule` creates networks in a directed way when using pairs of nodes-to-be for combining. That approach leads to some unnecessary temporay recreations of `DirectedNode`s when nodes become linked to several other nodes. This is a possible downside which should be worked on, if upcoming studies show a relevant fraction of double- and multi-creations of the same nodes. Another issue might be the fact that the actual network is searched pretty often when adding new nodes. This should also be addressed if that design turns out to be a bottle-neck in the reconstruction-process. The draft of the upper `DirectedNodeNetwork` is now designed in a way that it can (and will) be used on various locations in the draft of the **`SegmentNetworkProducerModule`** . A simplified code of the **`SegmentNetworkProducerModule`** can be found in chapter 3.

2.2 ActiveSector

Stores a pointer to `sector` in `SectorMap` and will be the interface to everything which is `SectorMap`-related.

Its full implementation can be found in `r18777` in:
`../tracking/trackFindingVXD/segmentNetwork/include/ActiveSector.h`.

```
#include <tracking/dataobjects/FullSecID.h>
#include <vector>

/** The ActiveSector Class.
* associated with static sector to be able to access filter cutoffs.
* Contains a vector with associated hits
* Allows to determine related inner sectors. */
template<class StaticSectorType, class HitType>
class ActiveSector {
protected:
/** ************************* DATA MEMBERS  */

/** Pointer to real sector after design of SectorMap */
  const StaticSectorType* m_staticSector;

/** stores indices of all associated Hits */
  std::vector<HitType*> m_hits;

public:
/** ************************* CONSTRUCTORS  */

/** Default constructor for root compatibility */
  ActiveSector(): m_staticSector(NULL) {}

/** Constructor.
  * staticSector: pointer to static sector associated with this one. */
  ActiveSector(const StaticSectorType* staticSector):
      m_staticSector(staticSector) {}

/** ************************* OPERATORS  */

/** overloaded '=='-operator for sorting algorithms */
  bool operator==(const ActiveSector& b) const
```

```
    { return (getFullSecID() == b.getFullSecID()); }

/** overloaded '<'-operator for sorting algorithms */
  bool operator<(const ActiveSector& b) const
    { return (getFullSecID() < b.getFullSecID()); }

/** overloaded '>'-operator for sorting algorithms */
  bool operator>(const ActiveSector& b) const
    { return (getFullSecID() > b.getFullSecID()); }

/** ************************ PUBLIC MEMBER FUNCIONS  */

/** returns all pointers to attached Hits */
  inline const std::vector<HitType*>& getHits() const
    { return m_hits; }

/** returns all IDs for inner sectors stored in the static SectorMap*/
  inline const std::vector<FullSecID::BaseType>& getInnerSecIDs() const
    { return m_staticSector->getInnerSecIDs(); }

/** returns pointer to associated static Sector in StoreArray */
  inline const StaticSectorType* getAttachedStaticSector() const
    { return m_staticSector; }

/** returns VxdID of sensor carrying current sector */
  inline FullSecID::BaseType getFullSecID() const
    { return m_staticSector->getFullSecID(); }

/** for pair of Hits, apply filters of staticSector,
* returns true if accepted */
  bool acceptTwoHitCombination(
      FullSecID secID,
      HitType& outerHit,
      HitType& innerHit)
  { return m_staticSector->accept(secID, outerHit, innerHit); }

/** for triple of Hits, apply filters of staticSector,
* returns true if accepted */
  bool acceptThreeHitCombination(
      FullSecID centerSecID,
      FullSecID innerSecID,
      HitType& outerHit,
      HitType& centerHit,
      HitType& innerHit)
  { return m_staticSector->accept(
        centerSecID,
        innerSecID,
        outerHit,
        centerHit,
        innerHit); }

/** adds new Hit to vector of Hits */
  inline void addHit(HitType* newNode)
    { m_hits.push_back(newNode); }
};
```

The class is very simple and just allows interfacing with the static sector. All functions addressing the static sector are currently dummy functions, to be able to sketch the member functions needed to be implemented in the static sector. Its task is to allow access to the static sector, which does not change during runtime when a SpacePoint can be applied to it. The ActiveSector does only live for the current event while the StaticSector is meant for storing static info like the filters for sector-combinations and their cuts. The ActiveSector on the other hand collects the hits found for it and allows accessing them too.

2.3 TrackNode

The issue about the necessity of `TrackNodes` has been discussed at several meetings (one important mail is the one with header: *Re: SectorMap* from 2015-04-14 sent by *jakob.lettenbichler@oeaw.ac.at*). Now it becomes clear again that no final decision could be made so far. As a reminder, the following section restates what has been noted at the mail mentioned above.

> *[...]*
> *As a consequence, there is no 1-1 correspondence between SpacePoints and VXDTFHits:*
>
> - *there are cases when SpacePoints are created, but no VXDTFHit: e.g. the SpacePoint does not lie in a valid sector (e.g. low momentum pass), or when former passes have reserved a certain cluster or cluster-combination.*
> - *there are cases when VXDTFHits are created, but no SpacePoints: e.g. virtual interaction point, for each valid cluster-combi, nSpacePoints == 1, but nVXDTFHits <= nPasses. Relations between storeArray-objects (which are an approach proposed by Martin Heck last autumn in Pisa) can not completely cover tasks formerly fulfilled by the VXDTFHit, since that interface is not designed for efficiently switching on and off of relations.*
>
> *[...]*

The design presented in this document reduces the need of a special `TrackNode`-class since old essential things like the connection to other hits are now stored by the

DirectedNodeNetwork and the definition of overlapping TCs is now handled in a different way and independently implemented in the SPTCNetworkProducerModule. Still the issue needs to be resolved since a SpacePoint can not know which active-sector it is attached to. The other main issue is the treatment of the virtual interaction point (VIP). This means that there are two options with their individual advantages and downsides:

1. Do not introduce an extra class for the TrackNodes, but use SpacePoints instead.

   This has the advantage, that the code can be kept shorter and no extra implementations are needed. The downside on the oder hand is that the hit for the IP does not fit to the other SpacePoints in various aspects:

   - It is not associated to a detector (SpacePoint::m_sensorType = ?, maybe Const::IR?).
   - It does not lie on a sensor (SpacePoint::m_vxdID = ?, maybe 0?).
   - It has therefore got no local coordinates (SpacePoint::m_normalizedLocal = ?, maybe {0, 0}?).
   - There is no constructor allowing to set all variables needed for a virtual interaction point (e.g. SpacePoint::m_position and SpacePoint::m_clustersAssigned) when there is no XYZCluster given. When adding such a constructor, its only purpose would be to construct a SpacePoint to be used as virtual IP, which seems a bit ill-designed.
   - How to store it? Since it is not of the same detector type as the others, a StoreArray just for the hit of the virtual IP would have to be used so the modules can access it like the other SpacePoints.

   The second (and more severe) downside is the storing of the sector-link to a SpacePoint. While this is no problem for building SpacePointNetworks, since ActiveSectorNetworks are used as a starting point, the issue becomes apparent when building SegmentNetworks: the SpacePoint have no Active- or StaticSector-Information and therefore have lost their information. This could be solved using relations between StaticSectors and SpacePoints, but that could not be tested so far.

2. Introduce an extra class TrackNode.

   Advantage is that many of these issues mentioned above can be encapsulated in the TrackNode-class, but it would mean that another layer of abstraction has to be introduced just for virtual IPs (and the Sector-information) again. Storing would be solved by keeping the info directly in the output of the SegmentNetworkProducerModule
   .

The VIP and the sector-issue have lead to a simplistic solution for the TrackNodes, which an be found in `r18777` in:
`../tracking/trackFindingVXD/segmentNetwork/include/TrackNode.h`.

```
#include <path/to//SpacePoint.h>
#include <path/to//StaticSector.h>
#include <path/to//ActiveSector.h>

/** Store combination of sector and spacePoint,
* since SpacePoint can not carry sectorConnection */
struct TrackNode {
/** pointer to sector */
  ActiveSector<StaticSector, TrackNode>* sector;

/** pointer to spacePoint */
  SpacePoint* spacePoint;

/** overloaded '=='-operator */
  bool operator==(const TrackNode& b) const
  {
    // simple case: no null-ptrs interfering:
    if (spacePoint != NULL and
        b.spacePoint != NULL and
        sector != NULL and
        b.sector != NULL) {
      // compares objects:
      return (*spacePoint == *(b.spacePoint)) and
             (*sector == *(b.sector));
    }

    /// case: at least one of the 2 nodes has no null-ptrs:
    // means that this Node has no null-Ptrs -> the other one has:
    if (spacePoint != NULL and sector != NULL) return false;

    // means that the other Node has no null-Ptrs -> this one has:
    if (b.spacePoint != NULL and b.sector != NULL) return false;

    // case: both nodes have got at least one null-ptr:
    bool spacePointsAreEqual = false;
    if (spacePoint != NULL and b.spacePoint != NULL) {
      spacePointsAreEqual = (*spacePoint == *(b.spacePoint));
    } else {
      spacePointsAreEqual = (spacePoint == b.spacePoint);
    }
    bool sectorsAreEqual = false;
    if (sector != NULL and b.sector != NULL) {
      sectorsAreEqual = (*sector == *(b.sector));
    } else {
      sectorsAreEqual = (sector == b.sector);
    }
    return (spacePointsAreEqual == true and sectorsAreEqual == true);
  }

/** overloaded '!='-operator */
  bool operator!=(const TrackNode& b) const { return !(*this == b); }

/** constructor  */
  TrackNode() : sector(NULL), spacePoint(NULL) {}
};
```

That simplistic approach listed above is used in the design of the `SegmentNetworkProducerModule` to keep the sector-information at hand.

2.4 Segment

Stores a pointer to the `TrackNodes` used and has extra data members: `bool isAlive, unsigned int state, bool isSeed`.

Its full implementation can be found in `r18777` in:
`../tracking/trackFindingVXD/segmentNetwork/include/Segment.h`.

```
#include <tracking/dataobjects/FullSecID.h>
#include <framework/logging/Logger.h>
#include <vector>

/** The Segment class.
* Represents segments of TCs needed for TrackFinderVXD-Modules */
template<class HitType>
class Segment {
protected:
/** ************************ DATA MEMBERS  */

/** pointer to hit forming the outer end of the Segment. */
  HitType* m_outerHit;

/** pointer to hit forming the inner end of the Segment. */
  HitType* m_innerHit;

/** iD of sector carrying outer hit */
  FullSecID::BaseType m_outerSector;

/** iD of sector carrying inner hit */
  FullSecID::BaseType m_innerSector;

/** state of Segment during CA process, begins with 0 */
  unsigned int m_state;

```

```
/** activation state.
* Living Cells (active) are allowed to evolve in the CA,
* dead ones (inactive) are not allowed */
  bool m_activated;

/** sets flag whether or not Segment is allowed to increase state
* during update step within CA */
  bool m_stateUpgrade;

/** sets flag whether or not Segment is allowed to be the seed
* of a new track candidate or not */
  bool m_seed;

public:
/** ************************ CONSTRUCTORS  */

/** Default constructor for the ROOT IO. */
  Segment():
    m_outerHit(NULL),
    m_innerHit(NULL),
    m_outerSector(FullSecID()),
    m_innerSector(FullSecID()),
    m_state(0),
    m_activated(true),
    m_stateUpgrade(false),
    m_seed(true) {}

/** Constructor.
* outerSector: secID of outer Sector associated with this segment.
* innerSector: secID of inner Sector associated with this segment.
* outerNode: pointer to outer Hit associated with this segment.
* innerNode: pointer to inner Hit associated with this segment. */
  Segment(
    FullSecID::BaseType outerSector,
    FullSecID::BaseType innerSector,
    HitType* outerNode,
    HitType* innerNode):
      m_outerHit(outerNode),
      m_innerHit(innerNode),
      m_outerSector(outerSector),
      m_innerSector(innerSector),
      m_state(0),
      m_activated(true),
      m_stateUpgrade(false),
      m_seed(true) {}

/** ************************ PUBLIC MEMBER FUNCIONS  */
// getters:
```

```

/** CA-feature: returns state of Segment */
  inline int getState() const
    { return m_state; }

/** CA-feature: returns activationState */
  inline bool isActivated() const
    { return m_activated; }

/** CA-feature: returns info whether stateIncrease is allowed or not */
  inline bool isUpgradeAllowed() const
    { return m_stateUpgrade; }

/** returns whether Segment is allowed to be a seed for TCs */
  inline bool isSeed() const
    { return m_seed; }

/** returns inner hit of current Segment */
  inline const HitType* getInnerHit() const
    { return m_innerHit; }

/** returns outer hit of current Segment */
  inline const HitType* getOuterHit() const
    { return m_outerHit; }

/** returns inner secID of current Segment */
  inline FullSecID::BaseType getInnerSecID() const
    { return m_innerSector; }

/** returns outer secID of current Segment */
  inline FullSecID::BaseType* getOuterSecID() const
    { return m_outerSector; }

// setters:

/** CA-feature: increases state during CA update step */
  inline void increaseState()
    { m_state++; }

/** CA-feature:
* if true, Segment is allowed to increase state during update step,
* if false, not allowed */
  inline void setStateUpgrade(bool up)
    { m_stateUpgrade = up; }

/** if true, Segment is allowed to be the seed for a new TC,
* if false, not allowed */
  inline void setSeed(bool isSeed)
```

```
    { m_seed = isSeed; }

/** if true, Segment is active = takes part during current CA iteration
* if false: Segment is inactive = does not take part, it is 'dead' */
  inline void setActivationState(bool activationState)
    { m_activated = activationState; }
};
```

The `Segment`-class is pretty simple too and mainly consists of some getters and setters. This design is meant for two-hit-segments only (see 1.2 and its subsections for more details), but a multi-hit-variant would still be compatible with the `DirectedNodeNetwork`-design and an implementation would be straight forward.

3 `SegmentNetworkProducerModule`

3.1 Overview

Input: What is needed to run the `SegmentNetworkProducerModule` :

- `SpacePoints`
- `SectorMap`

Internal classes used: Which classes are relevant here:

- `DirectedNodeNetwork` - defined and discussed in chapter 2.1
- `ActiveSector` - defined and discussed in chapter 2.2
- `TrackNode` - defined and discussed in chapter 2.3
- `Segment` - defined and discussed in chapter 2.4

Main steps: The `SegmentNetworkProducerModule` has the following main steps which will be described in more detail in chapter 3.2:

- `matchSpacePointToSectors(...)` - for each `SpacePoint` given, find according sector and store them in a fast and intermediate way.
- `buildActiveSectorNetwork(...)` - build a `DirectedNodeNetwork<ActiveSector>`, where all `ActiveSectors` are stored which have `TrackNodes` and compatible inner- or outer neighbours.
- `segFinder/buildTrackNodeNetwork(...)` - use `TrackNodes` stored in `ActiveSectors` to build `TrackNodes` which will stored and linked in a `DirectedNodeNetwork<TrackNode>`.
- `nbFinder/buildSegmentNetwork(...)` - use connected `TrackNodes` to form segments which will stored and linked in a `DirectedNodeNetwork<Segment>`.
- `storeToStoreObjPtr(...)` - fill output format.

Output: of the `SegmentNetworkProducerModule` :

- A StoreArray-container `StoreObjPtr<DirectedNodeNetworkContainer>`, which contains the networks mentioned above

The code of the `SegmentNetworkProducerModule` is written in in pseudo-code, which describes the things which have to be done for each step. Comments in the code shall help explaining some of the nontrivial parts.

3.2 Pseudo-implementation

Each event the `SegmentNetworkProducerModule` executes the following functions:

1. `vector< RawSectorData > collectedData = matchSpacePointToSectors();` 3.2.1
2. `void buildActiveSectorNetwork(collectedData);` 3.2.2
3. `void buildTrackNodeNetwork();` 3.2.3
4. `void buildSegmentNetwork();` 3.2.4

Which store their results in a `StoreObjPtr<DirectedNodeNetworkContainer>`, which carries 3 networks, the `DirectedNodeNetwork<ActiveSector>`, `DirectedNodeNetwork<TrackNode>` and the `DirectedNodeNetwork<Segment>`. In the code descriptions of the following sections, that StoreObjPtr is called `m_network`.

3.2.1 Section - `matchSpacePointToSectors(...)`:

returns: `vector<RawSectorData> collectedData;`

```
vector< RawSectorData > collectedData;
// collects trackNodes in there:
vector<TrackNode* >& trackNodes = m_network->accessTrackNodes();

// match all SpacePoints with the sectors:
for (SpacePoint& aSP : storeArray) {
  StaticSector* sectorFound = findSectorForSpacePoint(aSP);

  if (sectorFound == NULL) {
    B2WARNING("SpacePoint␣discarded!"); continue; }

  // sector for SpacePoint exists:
  FullSecID foundSecID = sectorFound->getFullSecID();

  trackNodes.push_back(new TrackNode(&aSP);

  vector<RawSectorData>::iterator iter =
      find_if(
        collectedData.begin(),
        collectedData.end(),
        [&](const RawSectorData & entry) -> bool
          { return entry.secID == foundSecID; }
      );

  // if secID not in collectedData:
  if (iter == collectedData.end()) {
    collectedData.push_back(
      { foundSecID , false, NULL, sectorFound, {trackNode}});
  } else {
    iter->hits.push_back(trackNode);
  }
} // loop over SpacePoints in StoreArray

// store IP-coordinates
if (m_PARAMAddVirtualIP == true) {
  m_network->setVirtualInteractionPoint();
  TrackNode* vIP = m_network->getVirtualInteractionPoint();
  StaticSector* sectorFound = findSectorForSpacePoint(*vIP->spacePoint);
  collectedData.push_back(
    {FullSecID(), false, NULL, sectorFound, {vIP}});
}
return move(collectedData);
```

Description: The function `findSectorForSpacePoint(SpacePoint aSP)` is discussed in 3.2.5. After executing the function `matchSpacePointToSectors(...)`, we have got all secIDs, which are relevant for this `SectorMap` and event and for each of them, their `SpacePoints` are collected and wrapped into `TrackNodes`. `SpacePoints` where no valid sector was found (relevant e.g. for low momentum passes which do not use all layers), are neglected during this step.

3.2.2 Section - `buildActiveSectorNetwork(...)`:

→ Building our first network.

returns: void (but creates `DirectedNodeNetwork<ActiveSector>`

```
DirectedNodeNetwork<ActiveSector>& activeSectorNetwork = m_network->accessActiveSectorNetw
// collects ActiveSectors in there:
vector<ActiveSector*>& activeSectors = m_network->accessActiveSectors();

for (RawSectorData& outerSectorData : collectedData) {
    ActiveSector* outerSector = new ActiveSector
```

```
      (outerSectorData.staticSector);

    // find innerSectors of outerSector and add them to the network:
    const std::vector<FullSecID>& innerSecIDs = outerSector->getInnerSecIDs();

    // skip double-adding of nodes into the network after first iteration
    bool isFirstIteration = true;
    for (const FullSecID innerSecID : innerSecIDs) {
      vector<RawSectorData>::iterator pos =
        std::find_if(
          collectedData.begin(),
          collectedData.end(),
          [&](const RawSectorData & entry) -> bool
            { return (entry.wasCreated == false) and
                     (entry.secID == innerSecID); }
        );

      // current inner sector has no SpacePoints in this event:
      if (pos == collectedData.end()) { continue; }

      // take care of inner sector first:
      ActiveSector* innerSector = NULL;
      if (pos->wasCreated) { // was already there
        innerSector = pos->sector;
      } else {
        innerSector = new ActiveSector(pos->staticSector);
        pos->wasCreated = true;
        pos->sector = innerSector;
        for (auto* hit : pos->hits) { hit->sector = innerSector; }
        // add all SpacePoints of this sector to ActiveSector:
        innerSector->addHits(pos->hits);
        activeSectors.push_back(innerSector);
      }

      // when accepting combination the first time, take care of outer sector:
      if (isFirstIteration) {
        outerSectorData.wasCreated = true;
        outerSectorData.sector = outerSector;
        for (auto* hit : outerSectorData.hits) { hit->sector = outerSector; }
        // add all SpacePoints of this sector to ActiveSector:
        outerSector->addHits(outerSectorData.hits);
        activeSectors.push_back(outerSector);

        activeSectorNetwork.linkTheseEntries(*outerSector, *innerSector);
        isFirstIteration = false;
        continue;
      }
      activeSectorNetwork.addInnerToLastOuterNode(*innerSector);
    } // inner sector loop

    // discard outerSector if no valid innerSector could be found
    if (isFirstIteration == false) { delete outerSector; }
  } // outer sector loop
```

Description: now we have got a network with all `ActiveSectors` which actually had any compatible `ActiveSectors`. All `ActiveSectors` which had `TrackNodess` but no compatible other `ActiveSectors` with `TrackNodess` this event, are stored. This consequently means: `TrackNodess` without any valid partners to be combined later on are neglected during this step. $\hat{=}$ 1-hit-filter. One last comment: The `ActiveSector` is actually a `ActiveSector<StaticSector, TrackNode>` which was omitted in the pseudo-code above for the sake of readability.

3.2.3 Section - `segFinder/buildTrackNodeNetwork(...)`:

→ Building our second network:

returns: `void` (but creates `DirectedNodeNetwork<TrackNode>`

There are two obvious ways to build this one : In a directed manner where `sectors = activeSectorNetwork.getOuterEnds()` or in an undirected one, where `sectors = activeSectorNetwork.getNodes()` will be used. Going through the `DirectedNodeNetwork` in an undirected way is easier because of the fact that one does not have to care whether or not all `DirectedNodes` have really been passed as intended, since all have been visited anyway. The downside is that there could be cases that the same `TrackNode` will be added more than once. This is captured by the design of the `DirectedNodeNetwork` and therefore will not lead to unintended behavior, but the issue of the possible overhead remains and has to be studied in more detail, before deciding if refactoring that part is needed. Since that implementation is a bit tricky, the undirected approach has been used for now, since the code is pretty simple for that one.

```
DirectedNodeNetwork<ActiveSector>& activeSectorNetwork =
        m_network->accessActiveSectorNetwork();
// collects TrackNodes in there:
DirectedNodeNetwork<TrackNode>& hitNetwork =
        m_network->accessHitNetwork();

// loop over outer sectors to get ->outerHits and inner sectors
for (auto* outerSector : activeSectorNetwork.getNodes()) {
  if (outerSector->getInnerNodes().empty()) continue;

  vector<TrackNode*>& outerHits = outerSector->getEntry().getHits();
  if (outerHits.empty()) continue;

  // loop over inner sectors to get ->innerHits and do compatibility-check
  for (auto* innerSector : outerSector->getInnerNodes()) {
    vector<TrackNode*>& innerHits = innerSector->getEntry().getHits();
    if (innerHits.empty()) continue;

    for (TrackNode* outerHit : outerHits) {
      // skip double-adding of nodes into the network after first iteration
      bool isFirstIteration = true;
      for (TrackNode* innerHit : innerHits) {
        // applying filters provided by the sectorMap:
        bool accepted =
          outerSector->getEntry().acceptTwoHitCombination(
            innerSector->getEntry().getFullSecID(),
            *outerHit,
            *innerHit);

        // skip combinations which weren't accepted:
        if (accepted == false) continue;

        // store combination of hits in network:
        if (isFirstIteration) {
          hitNetwork.linkTheseEntries(*outerHit, *innerHit);
          isFirstIteration = false;
          continue;
        }
        hitNetwork.addInnerToLastOuterNode(*innerHit);
      } // inner hit loop
    } // outer hit loop
  } // inner sector loop
} // outer sector loop
```

Description: Now we got a network, where each `TrackNode` carries a single `TrackNode` each and the links between the `DirectedNodes` are implicitly the segments - but not the class `Segment` yet. Therefore the segments are there, but not visible (no extra class for them), since their only job at the moment is to link `TrackNodes`. This is a reason why the old name `segFinder` (although the same filters - translated into the new design - are used) is maybe a bit misleading. To bypass this, here is a proposal for the new one: `buildTrackNodeNetwork`

3.2.4 Section - `nbFinder/buildSegmentNetwork(...)`:

→ Building our third network. Again, the design does not fully suppress the inadvertently recreated `Segments`. One of the steps to counteract that downside is that segments are created at the latest possible moment, when a compatible 3-hit-chain of hits are found, up to two `Segments` are created. For the sake of readability, the `Segment<TrackNode>` is like the `ActiveSector` reduced to `Segment` without the template arguments.

returns: void (but creates `DirectedNodeNetwork<Segment>`

```
DirectedNodeNetwork<TrackNode>& hitNetwork =
        m_network->accessHitNetwork();
DirectedNodeNetwork<Segment>& segmentNetwork =
        m_network->accessSegmentNetwork();
// collects Segments in there:
vector<Segment*>& segments = m_network->accessSegments();

for (auto* outerHit : hitNetwork.getNodes()) {
  vector<DirectedNode<TrackNode>*>& centerHits = outerHit->getInnerNodes();
  if (centerHits.empty()) continue; // go to next outerHit

  for (auto* centerHit : centerHits) {
    vector<DirectedNode<TrackNode>*>& innerHits = centerHit->getInnerNodes();
    if (innerHits.empty()) continue; // go to next centerHit

    // skip double-adding of nodes into the network after first iteration
    bool isFirstIteration = true;
    for (auto* innerHit : innerHits) {

      // applying filters provided by the sectorMap:
      bool accepted = outerHit->getEntry().sector->acceptThreeHitCombination(
                        centerHit->getEntry().sector->getFullSecID(),
                        innerHit->getEntry().sector->getFullSecID(),
                        outerHit->getEntry(),
                        centerHit->getEntry(),
                        innerHit->getEntry());

      // skip combinations which weren't accepted:
      if (accepted == false) continue;

      // create innerSegment first (order of storage in 'segments' is irrelevant):
      Segment* innerSegment = new Segment(
        outerHit->getEntry().sector->getFullSecID(),
        centerHit->getEntry().sector->getFullSecID(),
        &outerHit->getEntry(),
        &centerHit->getEntry());
      DirectedNode<Segment>* tempSegment = segmentNetwork.getNode(*innerSegment);
      if (tempSegment == NULL) {
        segments.push_back(innerSegment);
      } else {
        delete innerSegment;
        innerSegment = &tempSegment->getEntry();
      }

      // store combination of hits in network:

      if (isFirstIteration) {
        // create outerSector:
        Segment* outerSegment = new Segment(
          outerHit->getEntry().sector->getFullSecID(),
          centerHit->getEntry().sector->getFullSecID(),
          &outerHit->getEntry(),
          &centerHit->getEntry());
        DirectedNode<Segment>* tempSegment = segmentNetwork.getNode(*innerSegment);
        if (tempSegment == NULL) {
          segments.push_back(outerSegment);
        } else {
          delete outerSegment;
          outerSegment = &tempSegment->getEntry();
        }

        segmentNetwork.linkTheseEntries(*outerSegment, *innerSegment);
        isFirstIteration = false;
        continue;
      }
      segmentNetwork.addInnerToLastOuterNode(*innerSegment);
    } // innerHit-loop
  } // centerHit-loop
} // outerHit-loop
```

Description: Now we got a network, where `Segments` form the nodes and the links between them are implicitly that what we called *neighbours* so far. If we had more time, one could use the last section as a blueprint for easily writing a recursive network-algorithm, which simply continues sticking node-pairs of the input network into a single node of the output network (e.g. the next step would take neighbouring segments and would combine them to `3-hit-segments`, which would then form the nodes of the new `DirectedNodeNetwork`). For 4-hit filters the current implementations already have some useful filters, but for 5- and more-hit-filters, this approach would simply say alwaysTrue, but would still deliver automatically the longest chains. In that case one only has to search for multi-hit-segments which are inner-and outer end at the same time and collect them before starting the next iteration - et voilà, another tracking algorithm done. Since there is simply not enough time for this anyway, we have to neglect that one, but it would have been a nice (and cheaply to implement) idea...

3.2.5 Additional Classes and Helper-functions:

As mentioned in the comments of the code above, there are some additional classes and helper-functions included in the module, which are just mentioned briefly and in pseudo code here to cover their intended behavior:

`StaticSector* findSectorForSpacePoint(SpacePoint& aSP):` This function shall retrieve the correct sector for given SpacePoint and return a pointer to it (or NULL if no sector found).

```
for (StaticSector& aSector : m_secMap)
  if aSP->getVxdID() != aSector.getVxdID() {continue;}
  if (aSector.getFullSecID(aSP->getNormalizedLocalU(),
      aSP->getNormalizedLocalV()) != isValid)
        { continue;}
  return &aSector;
return NULL;
```

`struct RawSectorData:` - a simple struct to bundle raw data for a single sector before creating the actual ActiveSectors.

```
struct RawSectorData {
/** secID of rawSector */
  FullSecID secID;

/** needed for creating ActiveSectorNetwork:
* if yes, the sector was already added to the network */
  bool wasCreated;

/** stores a sector if one is found, NULL else */
  ActiveSector<StaticSector, TrackNode>* sector;

/** stores a static sector */
  StaticSector* staticSector;

/** collects the hits found on this sector */
  std::vector<Belle2::TrackNode*> hits;
};
```

3.3 Use cases:

After executing the **`SegmentNetworkProducerModule`** , the output lies in the DataStore and can be used for various purposes. Three of the most relevant ones will be listed now, which demonstrates how the results can be processed by several tracking algorithms.

CA: Take the `segmentNetwork`, loop over all nodes (which are the segments) and ask their connected inner nodes (which are the neighbours) if they are compatible. If yes, allow them to increase their state in the upgrade step. This is in fact a simplified sketch of the CA-algorithm, which will not fully be described here again. After several rounds, each segment has got its final state.

The second step in the module then will be to call a `findSeeds(segmentNetwork)`-function. This function marks all segments as seeds, whose state is "high enough". All these seeds are then allowed to be seed for a `tc-collector` (e.g the path finder-algorithm from Rudi) to start collecting. The resulting bunch of `TrackCand`s are then stored as `SpacePointTrackCand`s in a storeArray. Whether or not there shall be a simple QI-calculator to be included, we have to decide later on. Alternatively all of them are set onto a value of 0.5, but slightly smeared to suppress issues with the Hopfield Neural Network.

CKF: Since it would take to long to fully implement an fully-grown CKF, this is a draft for a slower - but from the result point-of-view equivalent - approach.

Short reminder: the CKF starts from a seed and goes in the direction defined by the seed and collects the $x(\leq 1)$ best next inner hits for extending the TC. Each of these valid inner hits are then added to a copy of the current TC and then independently followed on until end of road or a χ^2/QI-threshold has been surpassed. In the end, for each seed, one keeps the $y(\leq 1)$ best TCs for further studies.

Since `genFit` is not really designed to allow using their algorithms in such a freely manner, as would be needed for a real CKF, the implementation would take a while. Therefore a sufficiently good approximation to the intended behavior has to be sketched, to allow a feasible estimation whether or not implementing a real CKF could be recommended:

Start again with a `findSeeds(segmentNetwork)`-function. This one marks all segments as seeds, which are outerEnds of the network. That approach would be the strictest definition. Alternatively one could mark all segments as seeds which have a hit at layer 5 or higher, but in the end the following steps are the same, no matter how the seeds were chosen. Again a `tc-collector`-algorithm will be called for each seed and the resulting paths then converted to `genFit::Tracks` to be able to use the normal KF-interface used in the old design. This would then for each seed give a bunch of `TrackCandidates`, where the y best ones are kept and exported to the `storeArray<SpacePointTrackCand>`. Compared to a real CKF this approach is slower, since all paths collected from a seed have to be followed by the KF separately, which leads to several extrapolation- and update-steps to be done several times. But compared to a classical standard KF which would only add the best next hit to its path starting from a seed, more possible combinations can be scanned. This should lead to a higher efficiency, even if the execution time is relatively slow compared to a real (and optimized) CKF.

Thomas: to get the training data he needs for his neural networks, he can use the output of the `SegmentNetworkProducerModule` too. He then simply takes the `segmentNetwork` and executes another custom `findSeeds(segmentNetwork)`-function: This one simply marks all segments as seeds, which are not `innerEnd` of the network. Then he can simply loop over the network and collect each seed and its inner neighbours $\rightarrow$ this produces 3-hit-combinations (with the seed being the outer end of them) he needs for his training.

3.4 Wrap up:

The `DirectedNodeNetwork` seems to fulfill all requests asked for by the `SegmentNetworkProducerModule` design described in the chapter 3.2. Its performance - especially compared to the old design - is yet unclear but will tested thoroughly. Another point which is not fully solved yet is the interface to the `sectorMap`, which still has to be finalized. One has to consider the extra amount of work for the translation of the neighbourFinder-filters of the old code into the new design. Here a problem could be that the segments are not yet existing when testing for 3-hit-acceptance. The reason why they are not existing yet, is discussed and described in 3.2.3 and in 3.2.4. Even though the design should fulfill the expectations, the implementation and testing with realistic imput still will need a while, especially when considering the time needed for converting the neighbourFinder-filters too. Therefore the estimation for the time needed for implementing all what is needed for the `SegmentNetworkProducerModule` is still at least 2 weeks (without neighbourFinder-filters) and additional 2-3 weeks for neighbourFinder-filters conversion and testing of the whole thing.