N-ary tree improvements for clustering

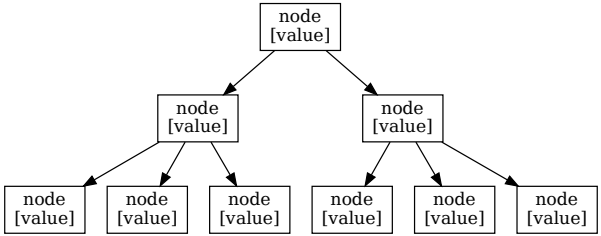
Brett Viren

December 23, 2024

Topics

- Reminder of *n-ary* tree, nodes, values and facades.
- New tree methods in support of clustering:
  - sorting
  - separate
  - remove
  - merge

N-ary tree nodes and values

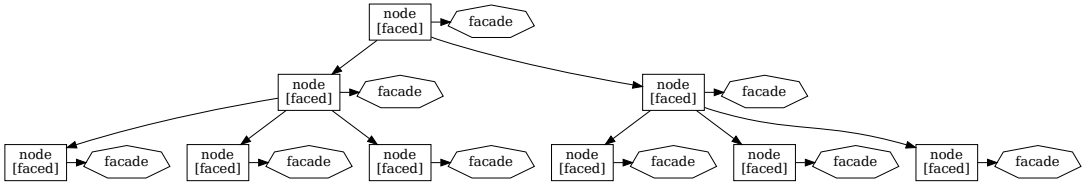


- The `Node` type handles all aspects of basic *n-ary tree* data structure.
  - Construction, mutation, depth-first search and children iteration.
- A **parent node owns its children nodes** via `std::list<unique_ptr<Node>>`.
  - User code accesses **lent pointers**: `Node*` or special iterators.
- A `Node` is templated by a `Value` type and holds an instance… by value.

As described, the tree of nodes is **homogeneous** of type.

Tree with facades

Users may need a tree of nodes that is **heterogeneous** of type.



- All edges represent owning pointers (`unique_ptr<>`).

Facade support

User may DIY

Give your `Value` type some polymorphic pointer or `std::variant`, etc.

General purpose support from `NaryTree`

`Faced`	Bolt on polymorphic pointer to `Value`.
`Facade`	General facade base class.
`FacadeParent`	Map facade class type hierarchy to tree hierarchy.

The n-ary tree facades for clustering

```
// Bolt support to hold a facade on to a Points Value type.
class Points : public Faced<Points> { ... };

// Define a concrete facade leaf type.
class Blob : public Facade<Points> { ... };

// Define concrete facade parent of concrete facade child.
class Cluster : public FacadeParent<Blob, Points> { ... };
class Grouping : public FacadeParent<Cluster, Points> { ... };

// Depth first descent
for (auto cluster : grouping->children()) { // iterate on L0's children
    cluster->....;
    for (auto blob : cluster->children()) { // iterate on L1's children
        blob->...;                          // L2 leaves.
    }
}
```

Facade navigation

With `NaryTree` facades, you can always find anything from anything.

```
assert(node->value.node() == node);
assert(facade->node()->value.facade() == facade);
```

Plus the usual node–node navigation via *n-ary* tree methods are provided by `Node`.

Sorting

The children of a `NaryTree::Node` are **well ordered**.

- Thus the tree itself is well ordered.
- Children are held in order of their insertion (by default).
  - May not be a universally-desired order.

Children can be re-ordered (sorted), but this necessarily **mutates** the tree.

Two levels for children sorting interfaces are provided:

- node
- facade

Node and facade level sorting

Node

```
// Sort node children by their "whatever" value.
node->sort_children([](const owned_ptr& a, const owned_ptr& b) -> bool {
    return a->value.whatever() < b->value.whatever();
})
```

Facade

```
// Sort grouping facade children by their cluster size.
grp->sort_children([](const Cluster* ca, const Cluster* cb) -> bool {
    return ca->nchildren() < cb->nchildren();
});
```

Connected components (CC) array and separation

```
A=[a, b, c, d, e, f]  original array of some values
C=[0,-1, 1, 1, 2,-1]  connected components (CC) array of IDs
```

After "separation", A is reduced and the loss is grouped into S:

```
S = separate(A,C)

A = [b,f]

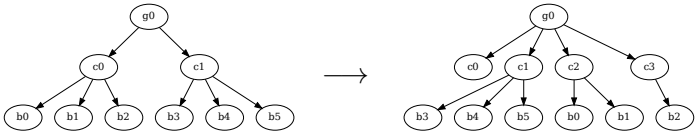
S = {
  0:[a]
  1:[c,d]
  2:[e]
}
```

The CC ID numbers may be arbitrary.

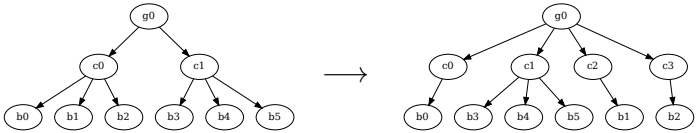
- but negative means to leave the value **in place** (in A).

Separation in tree context - group child's children to new sibling parents

separate c0's children {b0,b1,b2} with fully-specified CC array: [2 , 2 , 3]

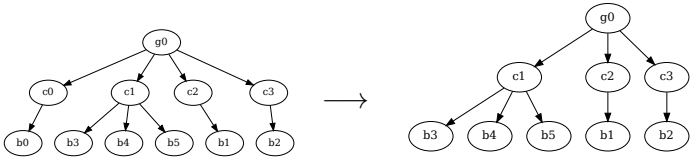


similar but retain c0's first child (b0) in place: [-1 , 2 , 3]



Remove a child from a parent

Remove c0 after separating some of its children.



Remember a `Node` **has** its `Value` and **owns** its children and facade.

- If you remove a child, you take ownership.
- If you drop the child, it and all that it holds is destroyed.

Node-level support for separate

```
assert(node->nchildren() == 10);
std::vector<int> cc = {0,0,-1,1,1,-1,2,2,-1, -1};

using nursery_type = std::list<std::unique_ptr<Node>>;
using nursery_group_type = std::map<int, nursery_type>;
nursery_group_type nurseries = node->separate(cc);

assert(nurseries.size() == 3);
assert(node->nchildren() == 4); // the -1's stay with original parent.
```

New parents **are not** created for node-level `separate()`.

- Caller accepts `vector` of `nurseries` that now own the separated children.
- If user drops `nurseries`, all separated children are lost (destroyed).

Facade-level support for separate - no remove

```
Grouping* grouping = root->value.facade<Grouping>();
std::vector<Cluster*> clusters = grouping->children();
const size_t orig = clusters.size();
Cluster* cluster = clusters[0];
std::vector<int> cc = /*...*/;

std::map<int, child_type*> splits =
    grouping->separate(cluster, cc);  // no remove by default

assert(groupin->nchildren() == orig + splits.size());
```

New parents **are** created as children of the root facade (`grouping` in eg).

- The separated child cluster is left in place.

Facade-level support for separate - with remove

```
Grouping* grouping = root->value.facade<Grouping>();
std::vector<Cluster*> clusters = grouping->children();
const size_t orig = clusters.size();
Cluster* cluster = clusters[0];
std::vector<int> cc = /*...*/;

std::map<int, child_type*> splits =
    grouping->separate(cluster, cc, true);  // remove

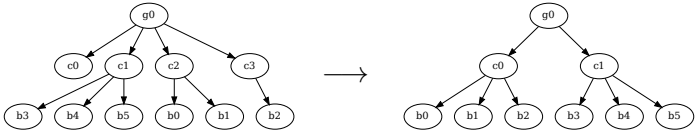
assert(cluster == nullptr);
assert(grouping->nchildren() == orig + splits.size() - 1);
```

Same as before but we pass `remove=true` causing `cluster` to be destroyed.

- This is what is currently used in the clustering code.
- If later we want to merge-back, then best use `remove=false`.

Merge - undo a separate

Assuming a previous `separate()` was done with `remove=false`, we can "undo" it:



Of course, can perform any arbitrary merge, does not need to be associated with a previous `separate()`.

DIY merge

User can do a merge by themselves:

At **node-level**, assuming we separated as above

```
for (auto& [gid,nur] : nurseries) {
    root->adopt_children(nur);
}
```

At **facade-level**, assuming we separated with `remove=false` as above

```
// The "splits" may be from a prior separate().
for (auto& [groupid, facade] : splits) {
    cluster->take_children(*facade);
    grouping->destroy_child(facade); // optional
}
```

Algorithm may decide to destroy or keep the now empty child facades.

Merge support - `std::vector` of children

The `NaryTree::FacadeParent` base class provides explicit merge support.

```
vector<Cluster*> clusters = ...;
ChildFacade* target = nullptr;  // optional/default
bool keep = false;              // optional/default
int existingID = -1;            // optional/default
vector<int> cc = grouping->merge(clusters, target, keep, existingID);
```

- Last three arguments are optional with their defaults as shown.
- The `target` will collect the children of `clusters`, make new target if `nullptr`.
- The `cc` records the cluster parent of each of the `target`'s children:
  - `existingID` (-1 by default) for the original children of `target`
  - Count up by 1 for children given by each in `clusters`.
- Remove all, now empty, in `clusters` unless `keep` is `true`.

Merge support - `std::map` of children

Basically same but takes `std::map<int,Children*>` instead of `std::vector<Children*>`

```
std::map<int,Children*> splits = ...;
ChildFacade* target = nullptr;  // optional/default
bool keep = false;              // optional/default
int existingID = -1;            // optional/default
vector<int> cc = grouping->merge(splits, target, keep, existingID);
```

- The `cc` values come from two places:
  - `existingID` for children originally found in `target`.
  - keys of `splits` for children of clusters in this map.
- Caller be careful not to put the same number as a key and as `existingID`.

Round trip example: separate → merge → store → re-separate

```
Cluster* cluster = ...;          // some "primary" cluster
vector<int> cc = ...;            // some original CC array

// Inside some clustering function....
// Keep "cluster" in place in grouping:
std::map<int, Cluster*> splits = grouping->separate(cluster, cc);
// Remvoe the clusters in "splits" after merging into "cluster".
vector<int> cc2 = grouping->merge(splits, cluster);
// The "cc" and "cc2" should be same up to ordering.
cluster->put_pcarray(cc2, "isolated"); // store for later.

// Inside some later clustering function....
for (Cluster* cluster : grouping->children()) {
    if (! cluster->has_pcarray("isolated") ) { continue; }
    vector<int> cc2 = cluster->get_pcarray("isolated");
    std::map<int, Cluster*> splits2 = grouping->separate(cluster, cc);
}
// We have now come full circle: splits == splits2.
```

Identifying a “primary” cluster.

By default, we use CC arrays where `-1` has special meaning:

- In `separate()` it means “keep this child in original (pre-separation) parent”.
- From `merge()` it means “this child came from original (pre-merge) parent”.

Thus if you have a CC array and find a `-1` it means that child (blob) was part of an original aka “primary” parent (eg cluster).

Changes to clustering code for separate

Changes

- Remove/replace original `separate()` definition.
- Replace all usage in `clus/` with:

```
Cluster* cluster = ...;
// Separate with remove.
grouping->separate(cluster, groups, true);
// Remind caller that cluster is destroyed with remove
assert(cluster);
```

Non-changes

- No implementation of the “merge” pattern.
  - May be most relevant in `clustering_neutrino()`.
- No fixes for separate+remove logic error (example later).
  - Added `assert()` to make that error obvious.