[boost] [graph] A few additions to propose to BGL

3 messages

Mikael Persson <mikael.s.persson@gmail.com> Sun, May 19, 2013 at 5:20 PM
To: boost@lists.boost.org

Hi all,

I have developed a number of small additions to the BGL, nothing major, just things that were useful to me and more practical as an extension to BGL instead of as a separate library, and if you're interested, I'm willing to contribute them back. These are non-intrusive additions, no modifications of existing BGL code is required (although some can be discussed, see below).

The proposed additions are:

1) Additional property-maps to fill the void between bundled properties and boost::property_map<> mechanisms. Incl. descriptor-to-bundle maps, bundle-to-property maps, composite maps (i.e., chaining property maps), etc.
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_more_property_maps.hpp

2) A few more property-tags that are common for graph algorithms.
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_more_property_tags.hpp

3) A few more graph-concepts:

a) Tree concept: Facilities for tree inspection.

b) MutableTree concept: Facilities for tree construction / destruction.

c) MutablePropertyTree concept: Same as MutableTree concept but with properties (like MutablePropertyGraph concept).

d) NonCompactGraph concept: For graphs / trees that may contain invalid vertices or edges (i.e., "holes").

https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/tree_concepts.hpp

4) Boost Graph Library Tree adaptors. This is for using the boost::adjacency_list to store a tree (and comply to tree concepts).
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_tree_adaptor.hpp

5) Pooled Adjacency-list. This is a wrapper around the boost::adjacency_list class which uses a method similar to Boost.Pool to store vertices in a contiguous storage (boost::vecS) while keeping vertex descriptors and iterators valid after removals.
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/pooled_adjacency_list.hpp

Here are the some subjective issues to discuss:

w.r.t.-3c) In my MutablePropertyTree concept, I have included removal functions that extract the properties of the elements to be removed. This can usually be implemented easily and provides symmetry w.r.t. to element-adding functions (if you can provide properties when creating elements, you should be able to retrieve them when removing elements). Without this, you can either copy the properties before removing the elements (needless copy and destruct), or you can std::move them out before removing the elements (leaving zombie elements in the graph in the meanwhile). The preferred option is to have the removal function move the properties into a destination (e.g. back-inserter). I would argue that the existing MutablePropertyGraph

concept should include overloads for recording properties during removals, this shouldn't break any existing user-side code and be easily added to existing graph data structures.

w.r.t.-3d) I have two issues with the NonCompactGraph concept:

1) Invalid elements mostly (or only) show up from iterators. So, one practical option is to have the iterators skip invalid elements. This effectively makes a "non-compact" graph look "compact". This would make this concept useless. However, iterators that must skip over invalid elements cannot provide random-access. There are ways to get the best of both worlds, such as introducing some "always-valid" iterators, which might or might not be identical to the base iterators (zero-overhead) depending on whether the graph is compact or not (a tag or meta-function might be needed to identify this).

2) A common operation with non-compact containers is to "re-pack" it. Generally, the advantage of leaving holes in a container is to avoid invalidating iterators. But, at certain points, one can decide to compact the container again. In the non-compact implementations that I have, I have included such functions, but never really needed them in practice (when removals <= insertions). I wonder if such functions should be included in the NonCompactGraph concept.

This is mainly what I propose to contribute back to the BGL (or any part of it). I have, however, lots of other code that is peripheral or closely tied to the BGL. Some is a bit more experimental at this stage, while other things are too peripheral or too specific. You are welcome to browse my library to see if anything else grabs your attention. See below for a short list of a few specific items that might be of interest.

Cheers, (and pardon the lengthy email)
Mikael.

P.S.:
Other interesting items that I have developed that might be worth proposing:

6) Linked-tree. A classic linked-tree data structure. It has similar signature as boost::adjacency_list and models the relevant BGL graph concepts and the proposed tree concepts:
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/linked_tree.hpp

7) Contiguous layout trees. These are fixed-arity (D-ary) trees that use a breadth-first layout (similar to heaps) in contiguous memory. They model the relevant BGL concepts and the proposed tree concepts.
a) D-ary BF-tree uses one contiguous storage with vertices laid out in breadth-first order. It works great, it's my work-horse tree-implementation and the memory-locality it provides has been a major performance factor in my code.
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_bf_tree.hpp
b) D-ary COB-tree (Cache-Oblivious B-tree) uses a tree of moderately-sized contiguous blocks with breadth-first layout within them. This thing is nice in theory, but in practice it has been difficult to guarantee correctness and to deliver good performance (it only out-performs the BF-tree when the size of the tree is in the gigabytes range).
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_cob_tree.hpp

8) I wrote an AD* algorithm, which I modeled after the BGL's existing A* search implementation. It's rather old code that I haven't done much with for a while, but I'm willing to resurrect it if there is interest for it. The fact that it is a dynamic / anytime / on-line graph algorithm makes it a bit odd compared to the other existing algorithms in BGL (all static / batch / off-line algorithms, AFAIK).
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/adstar_search.hpp

9) I wrote a multi-graph template to overlay an adjacency-list (i.e., general "graph") and a tree (i.e., complying to the proposed tree concepts). The point here is to have one set of vertices and two sets of edges, one forming a general graph and one forming a tree. The motivation is to: keep the graph and tree strongly synchronized by sharing the same storage; and, benefit from cache-friendly tree layouts (e.g., BF-tree) when using the general graph. My reservations with this are that the code is (1) not be quite polished yet, (2) a bit too advanced / specific for the general nature of the BGL, and (3) a bit convoluted to use (user-side code is messy).
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/adj_list_tree_overlay.hpp

10) If you find anything else you like, we can discuss it. I mostly write path-planning code and lots of numerical stuff. I have a nice sampling-based motion-planning framework in place that builds upon the BGL, and extended (and more mathematically sound) "Topology" concepts, as well as space-partitioning trees and stuff like that:
https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/graph_alg
https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/path_planning
https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/topologies

--
Sven Mikael Persson

Jeremiah Willcock <jewillco@osl.iu.edu> Tue, May 21, 2013 at 1:17 PM
Reply-To: boost@lists.boost.org
To: boost@lists.boost.org

On Sun, 19 May 2013, Mikael Persson wrote:

> Hi all,
>
> I have developed a number of small additions to the BGL, nothing major,
> just things that were useful to me and more practical as an extension to
> BGL instead of as a separate library, and if you're interested, I'm willing
> to contribute them back. These are non-intrusive additions, no
> modifications of existing BGL code is required (although some can be
> discussed, see below).

This seems really nice. One issue is that your changes will need to be under the Boost license for them to be included. There are more comments interspersed through the rest of the email.

> The proposed additions are:
>
> 1) Additional property-maps to fill the void between bundled properties and
> boost::property_map<> mechanisms. Incl. descriptor-to-bundle maps,
> bundle-to-property maps, composite maps (i.e., chaining property maps), etc.
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_more_property_maps.hpp

What is subobject_put_get_helper for? I can't tell what it is doing. Could you please briefly explain the others? How does self_property_map differ from typed_identity_property_map; I see that yours returns a reference rather than a value, but is that important?

> 2) A few more property-tags that are common for graph algorithms.
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_more_property_tags.hpp
>
> 3) A few more graph-concepts:
>
> a) Tree concept: Facilities for tree inspection.
>
> b) MutableTree concept: Facilities for tree construction / destruction.
>
> c) MutablePropertyTree concept: Same as MutableTree concept but with
> properties (like MutablePropertyGraph concept).

These look very useful, and seem intuitively "right" to me. What do you think of Boost.PropertyTree? It has a different domain than your concepts, but is also trying to represent trees. There also seem to be (limited) tree concepts in Boost.Graph; see <boost/graph/tree_traits.hpp> and <boost/graph/graph_as_tree.hpp>. They do not appear to be heavily used, though, and yours look to be cleaner and more capable.

> d) NonCompactGraph concept: For graphs / trees that may contain invalid

> vertices or edges (i.e., "holes").

The usual technique for this is to use a filtered_graph, which keeps the original graph's vertex and edge indexes (with not all values valid) and filters out the removed vertices in traversals. Could you please compare your approach to that?

> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/tree_concepts.hpp
>
> 4) Boost Graph Library Tree adaptors. This is for using the
> boost::adjacency_list to store a tree (and comply to tree concepts).
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_tree_adaptor.hpp
>
> 5) Pooled Adjacency-list. This is a wrapper around the
> boost::adjacency_list class which uses a method similar to Boost.Pool to
> store vertices in a contiguous storage (boost::vecS) while keeping vertex
> descriptors and iterators valid after removals.
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/pooled_adjacency_list.hpp

Those two items would both be very useful.

> Here are the some subjective issues to discuss:
>
> w.r.t.-3c) In my MutablePropertyTree concept, I have included removal
> functions that extract the properties of the elements to be removed. This
> can usually be implemented easily and provides symmetry w.r.t. to
> element-adding functions (if you can provide properties when creating
> elements, you should be able to retrieve them when removing elements).
> Without this, you can either copy the properties before removing
> the elements (needless copy and destruct), or you can std::move them out
> before removing the elements (leaving zombie elements in the graph in the
> meanwhile). The preferred option is to have the removal function move the
> properties into a destination (e.g. back-inserter). I would argue that the
> existing MutablePropertyGraph concept should include overloads for
> recording properties during removals, this shouldn't break any existing
> user-side code and be easily added to existing graph data structures.
>
> w.r.t.-3d) I have two issues with the NonCompactGraph concept:
>
> 1) Invalid elements mostly (or only) show up from iterators. So, one
> practical option is to have the iterators skip invalid elements. This
> effectively makes a "non-compact" graph look "compact". This would make
> this concept useless. However, iterators that must skip over invalid
> elements cannot provide random-access. There are ways to get the best of
> both worlds, such as introducing some "always-valid" iterators, which might
> or might not be identical to the base iterators (zero-overhead) depending
> on whether the graph is compact or not (a tag or meta-function might be
> needed to identify this).

In which situations do you need random access iteration? Are there places that you would use the non-compact iterators that are not immediately followed by filtering the vertices manually?

> 2) A common operation with non-compact containers is to "re-pack" it.
> Generally, the advantage of leaving holes in a container is to avoid
> invalidating iterators. But, at certain points, one can decide to compact
> the container again. In the non-compact implementations that I have, I have
> included such functions, but never really needed them in practice (when
> removals <= insertions). I wonder if such functions should be included in
> the NonCompactGraph concept.

It is fine to have functions in your implementation that are not part of the concept; it just means that code that

uses them would rely on that particular type rather than the general concept. You could also have a refining concept that adds the extra functionality.

> This is mainly what I propose to contribute back to the BGL (or any part of it). I have, however, lots of other code that is peripheral or closely tied to the BGL. Some is a bit more experimental at this stage, while other things are too peripheral or too specific. You are welcome to browse my library to see if anything else grabs your attention. See below for a short list of a few specific items that might be of interest.
>
> Cheers, (and pardon the lengthy email)
> Mikael.
>
> P.S.:
> Other interesting items that I have developed that might be worth proposing:
>
> 6) Linked-tree. A classic linked-tree data structure. It has similar signature as boost::adjacency_list and models the relevant BGL graph concepts and the proposed tree concepts:
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/linked_tree.hpp
>
> 7) Contiguous layout trees. These are fixed-arity (D-ary) trees that use a breadth-first layout (similar to heaps) in contiguous memory. They model the relevant BGL concepts and the proposed tree concepts.
> a) D-ary BF-tree uses one contiguous storage with vertices laid out in breadth-first order. It works great, it's my work-horse tree-implementation and the memory-locality it provides has been a major performance factor in my code.
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_bf_tree.hpp

Can this tree use external storage? The current d-ary heap in BGL doesn't, and that causes problems for some users.

> b) D-ary COB-tree (Cache-Oblivious B-tree) uses a tree of moderately-sized contiguous blocks with breadth-first layout within them. This thing is nice in theory, but in practice it has been difficult to guarantee correctness and to deliver good performance (it only out-performs the BF-tree when the size of the tree is in the gigabytes range).
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_cob_tree.hpp

Would this (and the one above) be useful for people who need BFS and DFS trees that allow going from parents to children, not just the other way?

> 8) I wrote an AD* algorithm, which I modeled after the BGL's existing A* search implementation. It's rather old code that I haven't done much with for a while, but I'm willing to resurrect it if there is interest for it.
> The fact that it is a dynamic / anytime / on-line graph algorithm makes it a bit odd compared to the other existing algorithms in BGL (all static / batch / off-line algorithms, AFAIK).
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/adstar_search.hpp

There is an incremental connected components function, but having more dynamic algorithms would be good in general.

> 9) I wrote a multi-graph template to overlay an adjacency-list (i.e., general "graph") and a tree (i.e., complying to the proposed tree concepts). The point here is to have one set of vertices and two sets of edges, one forming a general graph and one forming a tree. The motivation is to: keep the graph and tree strongly synchronized by sharing the same

> storage; and, benefit from cache-friendly tree layouts (e.g., BF-tree) when
> using the general graph. My reservations with this are that the code is (1)
> not be quite polished yet, (2) a bit too advanced / specific for the
> general nature of the BGL, and (3) a bit convoluted to use (user-side code
> is messy).
> https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/adj_list_tree_overlay.hpp
>
> 10) If you find anything else you like, we can discuss it. I mostly write
> path-planning code and lots of numerical stuff. I have a nice
> sampling-based motion-planning framework in place that builds upon the BGL,
> and extended (and more mathematically sound) "Topology" concepts, as well
> as space-partitioning trees and stuff like that:
> https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/graph_alg
> https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/path_planning
> https://github.com/mikael-s-persson/ReaK/tree/master/src/ReaK/ctrl/topologies

I'll take a look at these last two items and think about them more.

-- Jeremiah Willcock

Mikael Persson <mikael.s.persson@gmail.com> Tue, May 21, 2013 at 7:43 PM
To: boost@lists.boost.org

Hi Jeremiah, thanks for the reply,
here are the answers to some of your questions:

> This seems really nice. One issue is that your changes will need to be under the Boost license for them to be included.

Sure, that was implied when I said "I'm willing to contribute them back". The code in my public repository is GPL-licensed, but I'm the sole author and I will re-license whatever parts would be incorporated into Boost. No problem.

> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_more_property_maps.hpp
>
> What is subobject_put_get_helper for? I can't tell what it is doing. Could you please briefly explain the others? How does self_property_map differ from typed_identity_property_map; I see that yours returns a reference rather than a value, but is that important?

Currently, Boost uses "put_get_helper" as a CRTP-style helper class to give the "put-get" free-function capabilities to property map classes that are
then only required to have an operator[]. The subobject_put_get_helper class does the same but for the cases where the mapped value is a subobject (e.g., data-member) of the key value. This is the main thing. This idea of the mapped value being a subobject of the key value is a pretty critical thing and allowing it is probably the major "change" introduced by some of these property-maps.

There are two kinds of property-maps in my header.

The first kind are those that are useful when creating custom graph classes with BGL-like property-maps associated to them:

whole_bundle_property_map:
This map delivers the vertex_bundled / edge_bundled associated with a given vertex or edge descriptor. This exists in the BGL as implementation details of graphs like adjacency_list (I believe it is "vec_adj_list_vertex_all_

properties_map").

propgraph_property_map:
This is a property-map that relies on some get-function on the graph. BGL's adjacency_list implements calls like "get(prop_tag, graph, key)" as a combination of "get(prop_tag, graph)" to get the property-map and them "prop_map[key]" to get the value. This property-map does the reverse, in implements "prop_map[key]" in terms of "get(prop_tag, graph, key)", which is easier to implement when writing a new graph class (if you've looked at the BGL's adjacency_list implementation, you know what I mean).

bundle_member_property_map:
Similarly to the others, this is what you can use to implement calls like "get(&MyBundle::SomeMember, graph)" on any graph that implements the operator[]. The motivation for this is the same as the other two.

In other words, I have a more straight-forward approach, my graph types implement an operator[] to get the vertex-/edge-bundle, and possibly some specific "get(prop_tag, graph, key)" overloads. The existing BGL classes have a much more convoluted implementation, probably as a consequence of starting out using only boost::property<> and property-maps. I simply found it easier to do things the other way around in my implementations, which led to these property-maps as building-blocks to turn a simple graph class implementation into one that fully supports BGL-style property-maps. That's all this is. The BGL is not very welcoming from a user to write/use his own graph classes, one of the main hurdle is wrapping one's head around the convoluted property-map-related meta-programming it uses (and implicitly requires).

The second kind are those that deal directly with bundles:

self_property_map:
This is an "identity" map, but it doesn't do a copy. That's all. It differs from existing "typed_identity_property_map" exactly because of that. But it's a critical difference when coupled with the next two property-maps.

composite_property_map:
This is can be used to chain two property-maps, i.e., the value-type of the inner-map is the key-type of the outer-map. That's pretty self-explanatory I think.

data_member_property_map:
This delivers a data-member of a key-type object. For example, consider this simple function:

```
template <typename Graph, typename PositionMap>
void add_random_vertex(Graph& g, PositionMap position) {
  typedef typename boost::graph_traits<Graph>::vertex_bundled VertexProp;

  VertexProp vp;                // create a vertex-bundle to be inserted into the graph.
  put(position, vp, rand() );   // set its position value.
  add_vertex(vp, g);            // add it to the graph.
};
```

Here, the position map would be of type data_member_property_map, it's key-type is the vertex_bundled type and it's value-type is some position type. The point here is that the vertex-property (given to the add_vertex function) can be set with a generic property-map. This was essential to me, and I'm sure would be useful to others too. AFAIK, the BGL's doesn't have such capability. Note that coupled with the composite-property-map, you can, for example, create a vertex-descriptor to bundle-member property-map out of a whole_bundle_property_map and a data_member_property_map, which is also very handy.

So much for a "brief" explanation, sorry.

> These look very useful, and seem intuitively "right" to me. What do you think of Boost.PropertyTree? It has a different domain than your concepts, but is also trying to represent trees. There also seem to be (limited) tree concepts in Boost.Graph; see <boost/graph/tree_traits.hpp> and <boost/graph/graph_as_tree.hpp>. They do not appear to be heavily used, though, and yours look to be cleaner and more capable.

Frankly, I don't understand the point of Boost.PropertyTree, I think it's just too foreign for me to say much about it, I think the domain is just very different, and I'm not a big fan, in general, of the whole "making a tree-like container", I don't see the point.

I actually didn't know about the BGL's tree_traits / graph_as_tree. It's minimalistic, to say the least. The only thing it has is the "traverse_tree" function, which is just confusing (N.B.: it's implemented with actual recursive calls, that's not acceptable), and the root / children / parent functions (which are kind of trivial). It's very incomplete, and doesn't really address the main thing, which is constructing and destructing the tree structure. When constructing a tree (using a BGL graph), you always create a vertex and then create an edge to its parent, and when destructing a tree, you always clear an edge, remove a vertex, and possibly all its children (which can be tricky). These are the annoying / error-prone operations that also break genericity (e.g., you cannot change a graph-class for a tree-class without having to re-code these parts, and similarly if the assumptions about the graph-class change).

Another possible set of useful functions for trees would be "re-wiring" function, such as re-connecting a child to a new parent. Currently, with my tree concepts, you can remove an entire branch, but you cannot "move it" to another parent. This obviously doesn't make sense for all tree types, but could constitute another refined concept (e.g., "ReWireableTree").

> > d) NonCompactGraph concept: For graphs / trees that may contain invalid
> > vertices or edges (i.e., "holes").
>
> The usual technique for this is to use a filtered_graph, which keeps the original graph's vertex and edge indexes (with not all values valid) and filters out the removed vertices in traversals. Could you please compare your approach to that?

The filtered_graph is used when the user has some external mechanism for "invalidating" vertices, and wants to create a filtered view of the vertices. The NonCompactGraph concept would apply to graphs that have an internal mechanism that invalidates vertices, and yet conserve an unfiltered view of itself. In other words, the concept encodes the predicates that would/could be used when building a filtered view. I guess that's the best way I can put it. I thought this concept made sense when I first wrote it, but that conviction has been growing weaker ever since.

> In which situations do you need random access iteration? Are there places that you would use the non-compact iterators that are not immediately followed by filtering the vertices manually?

That is exactly why my convictions have grown weak. I have never used random-access on the vertex iterators, and can't think of a reason to. Every traversal I have in my code is followed by a manual test for validity... well, I forgot it in some places, which caused me a lot of grief. So, I'm mostly on your side on this. I certainly know that people use random-access when using the grid_graph class, which could also potentially be made to have "holes" in it. But there are no such classes (neither in BGL nor in my proposed additions).

I'd be more than willing to remove that NonCompactGraph concept, and amend some of my implementations to inherently be "filtered" to skip invalid vertices and edges during traversals.

> It is fine to have functions [repack] in your implementation that are not part of the concept; it just means that code that uses them would rely on that particular type rather than the general concept. You could also have a refining concept that adds the extra functionality.

I'll leave it as is, especially if I remove the NonCompactGraph concept. Although it wouldn't be hard to have a general function template like:

```
template <typename Graph>
void repack_graph(Graph&) { };
```

and overload it for any graph that does provide such a capability. In other words, have genericity by the "works for all, but actually does something only when needed" principle. No need for an additional concept.

> > 4) Boost Graph Library Tree adaptors.
> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/bgl_tree_adaptor.hpp
> >
> > 5) Pooled Adjacency-list.
> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/pooled_adjacency_list.hpp
>
> Those two items would both be very useful.

I agree. The tree adaptor is sort of an obvious and easy item to include if tree concepts are introduced (currently, it only works for adjacency_list, could be extended further, I was just too lazy to do all the meta-programming required to dispatch to correct overloads). The pooled adjacency list is very useful too. I was tempted by the option of using a Boost.Pool allocator for a std::list vertex storage, but that doesn't really work quite as well, due to the overhead of the next/prev pointers and the non-linear traversal pattern. So, I would favor this approach instead. Also note that my implementation relies on Boost.Variant to discriminate valid/invalid nodes and to store the "next-hole" pointers within the invalid vertices (i.e., as a "union"), but I'm open to suggestions if there is a better way to do it.

> > 6) Linked-tree.
> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/linked_tree.hpp
> >
> > 7) Contiguous layout trees. [..]
> > a) D-ary BF-tree [..]
> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_bf_tree.hpp
> > b) D-ary COB-tree (Cache-Oblivious B-tree) [..]
> > https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/graph_alg/d_ary_cob_tree.hpp
>
> Can this tree use external storage? The current d-ary heap in BGL doesn't, and that causes problems for some users.

I don't really understand what you mean by "external storage". These trees are for storage, nothing else. If you externalize the storage, there wouldn't be much left except an adaptor (like the tree-adaptor above). The linked-tree does the exact same kind of work as the adjacency_list, but restricted to a tree structure (each node has one parent (one in-edge) and any number of children (and out-edges)). The contiguous layout trees are similar except they have a maximum number of children (fixed-arity). They do not have any other "logic" beyond enforcing those invariants (one parent, some children for each node). So, the only thing they do is storage. In fact, I have mostly used the contiguous layout trees as the external storage for other things (to improve cache-performance).

About the d-ary heap, I've been using it a lot, it's a great little heap implementation, but it's quite annoying to use. But my main problem with it is the fact that it uses external storage for the indices. But that's a bit off-topic.

> Would this (and the one above) be useful for people who need BFS and DFS trees that allow going from parents to children, not just the other way?

All my trees are bi-directional. You can get to the parent node via the "in_edges(v, g)" function (or there could be an additional function for that in the tree concepts), just like a normal graph, except that there is a guarantee that there is always exactly one in-edge, unless it's the root vertex. The contiguous layout trees are bi-directional without any overhead because, given a vertex, you can calculate the memory location of the parent vertex or any child vertex with some simple arithmetic. The breadth-first layout is optimal for breadth-first traversal (which is just a linear memory traversal), it is also, AFAIK, as good as can be for best-first search/traversal, and it's also pretty good for depth-first traversal (of course, a depth-first layout would be better for that). But again, these are merely for storage purposes. For actually organizing the tree, you would code stuff on top of that, for example, I have a Dynamic Vantage-Point tree implementation that I use for metric-space partitioning (for fast nearest-neighbor queries), and the tree storage type is just one template parameter (and relies on the proposed tree concepts to use it):
https://github.com/mikael-s-persson/ReaK/blob/master/src/ReaK/ctrl/path_planning/dvp_tree_detail.hpp

This separation of "storage" versus "indexing" is very important in my opinion. There is the indexing logic by which the tree is created / organized / re-wired, and there is the way it is stored in memory. The indexing logic seems to be more the domain of the Boost.Intrusive library (red-black trees, AVL-trees, etc.., this is all "indexing logic", not storage). And my trees are purely in the domain of storage.

Cheers,
Mikael.

--
Sven Mikael Persson, M.Sc.(Tech.)
PhD Candidate and Vanier CGS Scholar,
Department of Mechanical Engineering,
McGill University,