Binary tree

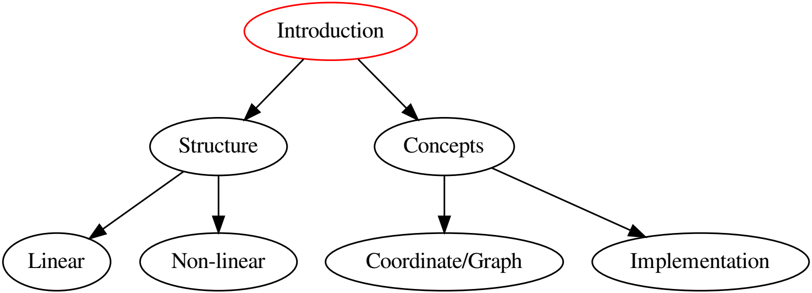
Why grow your own?

Jeremy Murphy

ResMed

May 8, 2019

Outline


Introduction
Structure
Concepts
Linear
Non-linear
Coordinate/Graph
Implementation

How did this start?

How did this start?

- Can we have a binary tree in Boost.Graph?

How did this start?

- Can we have a binary tree in Boost.Graph?
- No, because it won't be faster.

How did this start?

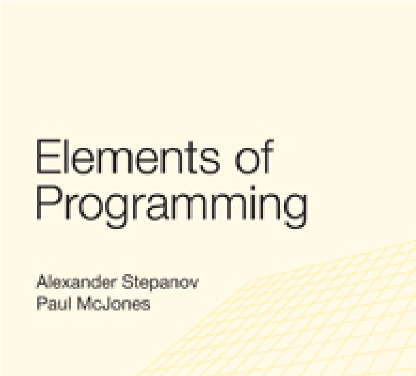
- Can we have a binary tree in Boost.Graph?
- No, because it won't be faster.
- Whoops, I was wrong.

How did this start?

- Can we have a binary tree in Boost.Graph?
- No, because it won't be faster.
- Whoops, I was wrong.
- Ordinal vs cardinal tree.

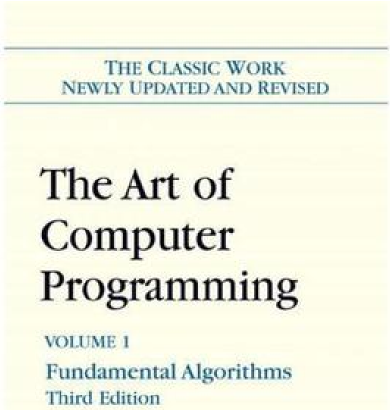
How did this start?

- Can we have a binary tree in Boost.Graph?
- No, because it won't be faster.
- Whoops, I was wrong.
- Ordinal vs cardinal tree.



How did this start?

- Can we have a binary tree in Boost.Graph?
- No, because it won't be faster.
- Whoops, I was wrong.
- Ordinal vs cardinal tree.



Goal

Boost.Graph has two classes, `adjacency_list`, which is mutable and `compressed_sparse_row_graph` (CSRG), which is efficient.

Goal

Boost.Graph has two classes, `adjacency_list`, which is mutable and `compressed_sparse_row_graph` (CSRG), which is efficient.
Create a binary tree that is:

- *mutable*,
- trivially also a forest,
- faster than `adjacency_list`,
- at least competitive with `compressed_sparse_row_graph`
- easily accessible to everyone.

Goal

Boost.Graph has two classes, `adjacency_list`, which is mutable and `compressed_sparse_row_graph` (CSRG), which is efficient.
Create a binary tree that is:

- *mutable*,
- trivially also a forest,
- faster than `adjacency_list`,
- at least competitive with `compressed_sparse_row_graph`
- easily accessible to everyone.
- Benefit of BGL: existing graph theory algorithms.

Binary tree in C++

Audience poll: is there a binary tree in the standard library?

Binary tree in C++

Audience poll: ~~is there a binary tree in the standard library?~~
How *many* binary trees are in the standard library?

Binary tree in C++

Audience poll: ~~is there a binary tree in the standard library?~~
How *many* binary trees are in the standard library?

1. `set` and `map`: *usually* a Red-black tree
2. Heap operations: `make_heap`, `push_heap`, `pop_heap`, etc.

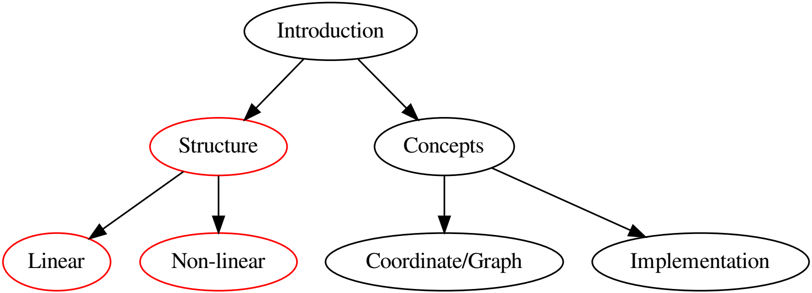
Binary tree in C++

Audience poll: ~~is there a binary tree in the standard library?~~
How *many* binary trees are in the standard library?

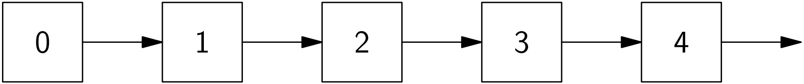
1. `set` and `map`: *usually* a Red-black tree
2. Heap operations: `make_heap, push_heap, pop_heap`, etc.

But these are very specific binary trees.
Many invariants to maintain but in return additional features provided.

Outline

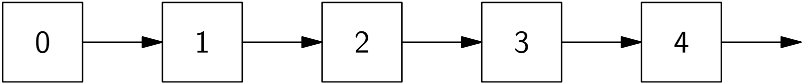

Introduction
Structure
Concepts
Linear
Non-linear
Coordinate/Graph
Implementation

Data structure: non-linearity is interesting in its own right



`forward_list<"void">` – numbers are indices, not data.

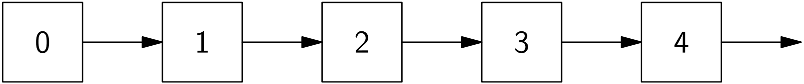
Data structure: non-linearity is interesting in its own right



`forward_list<"void">` – numbers are indices, not data.

Is it useful?

Data structure: non-linearity is interesting in its own right

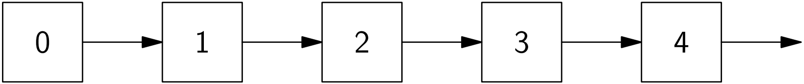


`forward_list<"void">` – numbers are indices, not data.

Is it useful?

What about reachability? `bool reachable(x, y)`

Data structure: non-linearity is interesting in its own right



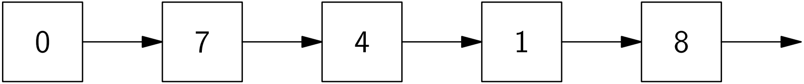
`forward_list<"void">` – numbers are indices, not data.

Is it useful?

What about reachability? `bool reachable(x, y)`

For a simply growing tail, we can use < to answer the question.

Data structure: non-linearity is interesting in its own right



`forward_list<"void">` – numbers are indices, not data.

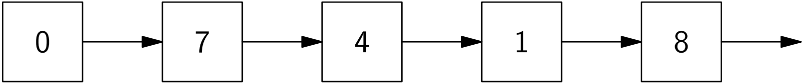
Is it useful?

What about iterator reachability? `bool reachable(x, y)`

For a simply growing tail, we can use $<$ to answer the question.

For randomly inserted elements, we need to remember where things are.

Data structure: non-linearity is interesting in its own right



`forward_list<"void">` – numbers are indices, not data.

Is it useful?

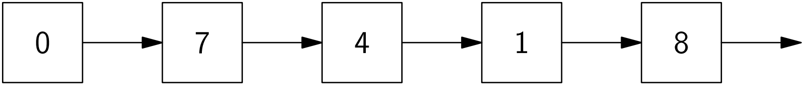
What about iterator reachability? `bool reachable(x, y)`

For a simply growing tail, we can use $<$ to answer the question.

For randomly inserted elements, we need to remember where things are.

Audience quiz: What functions are there on the metadata of a standard linear data structure?

Data structure: non-linearity is interesting in its own right



`forward_list<"void">` – numbers are indices, not data.

Is it useful?

What about iterator reachability? `bool reachable(x, y)`

For a simply growing tail, we can use $<$ to answer the question.

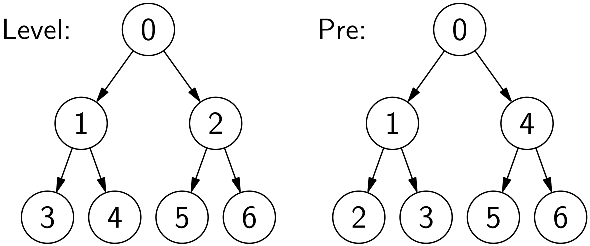
For randomly inserted elements, we need to remember where things are.

Audience quiz: What functions are there on the metadata of a standard linear data structure?

- `begin`/`end`
- `size` (and `empty`)
- `resize`
- `capacity` (but it's not salient)

Data structure: non-linearity is interesting in its own right

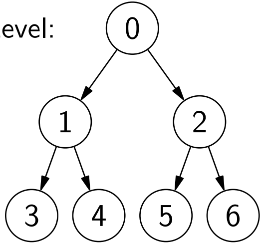
Reachability in a systematically constructed binary tree?



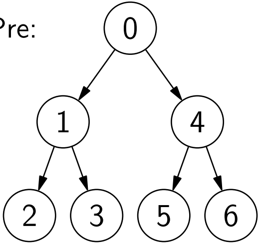
Data structure: non-linearity is interesting in its own right

Reachability in a systematically constructed binary tree?

Level:



Pre:

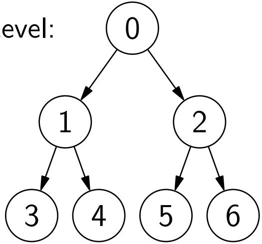


So what about functions and algorithms on non-linear structures?

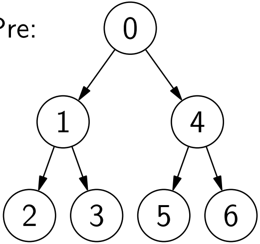
Data structure: non-linearity is interesting in its own right

Reachability in a systematically constructed binary tree?

Level:



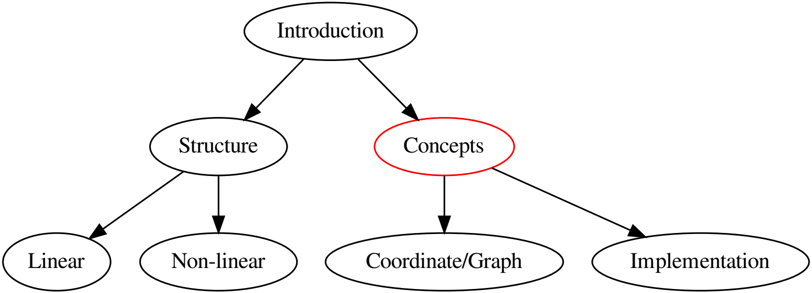
Pre:



So what about functions and algorithms on non-linear structures?

- `weight`
- `height`
- `reachable`
- `isomorphic`
- connected components
- common subgraph
- dominator tree
- planar

Outline



Introduction
Structure
Concepts
Linear
Non-linear
Coordinate/Graph
Implementation

What is Boost.Graph?

What are the key features of the STL on which the design is based?

What is Boost.Graph?

What are the key features of the STL on which the design is based?

Algorithm/Data-Structure Interoperability One algorithm implementation can work on various data structures: iterators are the key ingredient for decoupling.

Extension through Functions Objects Specific operations in an algorithm are customizable, e.g. the BinaryOp here:

```
std::accumulate(I first, I last, T init,
                BinaryOp op=std::plus<>);
```

Element Type Parameterization Containers parameterized on type T. Most common understanding of 'generic' but probably least interesting.

What is Boost.Graph?

Generic programming library of graph theory data structures and algorithms. Same principles as STL with these differences:

Algorithm/Data-Structure Interoperability
: Three different kinds of iterator for traversal of: vertices, edges, adjacent neighbours.

Extension through Visitors
: Key event points during an algorithm can be acted on. In depth-first search for example: start vertex, discover vertex, tree edge, etc.

Vertex and Edge Property Multi-Parameterization
: Property maps for the parts of the graph that interest us. Could be vertices, edges or any subset of both.

What is Boost.Graph?

Generic programming library of graph theory data structures and algorithms. Same principles as STL with these differences:

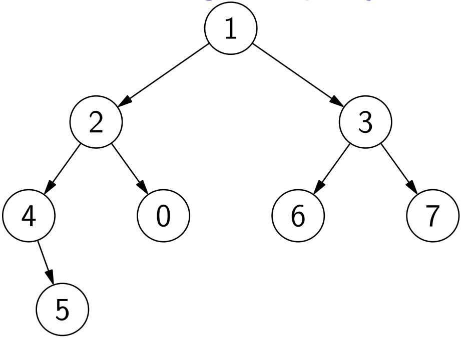
Algorithm/Data-Structure Interoperability
: Three different kinds of iterator for traversal of: vertices, edges, adjacent neighbours.

Extension through Visitors
: Key event points during an algorithm can be acted on. In depth-first search for example: start vertex, discover vertex, tree edge, etc.

Vertex and Edge Property Multi-Parameterization
: Property maps for the parts of the graph that interest us. Could be vertices, edges or any subset of both.

In Elements of Programming, all algorithms are defined on Coordinates, which are the equivalent of an Iterator.

Vertex and Edge Property Multi-Parameterization



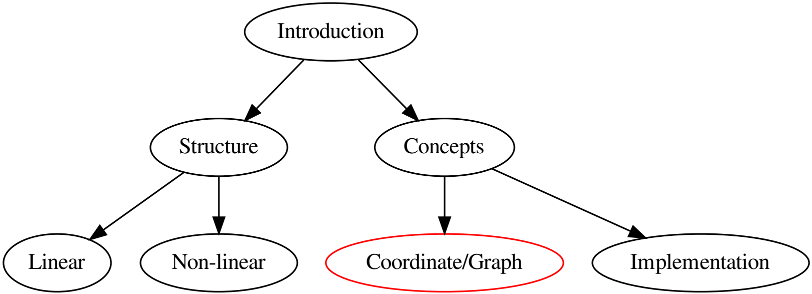
vertex	value
0	“foo”
5	“bar”
6	“baz”
7	“xyzzy”

edge	value
1-3	1.09
1-2	2.55
2-4	4.32
2-0	1.23
3-6	9.99
3-7	0.08

Summary

- Non-linear data structures such as graphs have interesting structure without data.
- Thus, a graph does not need to be a container.

Outline


Introduction
Structure
Concepts
Linear
Non-linear
Coordinate/Graph
Implementation

Binary tree concepts: Elements of Programming

EoP defines a *BifurcateCoordinate* concept that fits with the definition given in Knuth and is a recursive definition of a binary tree.

```
bool empty(C)
bool has_left_successor(C)
bool has_right_successor(C)
C left_successor(C)
C right_successor(C)
```

```
bool has_predecessor(C)
C predecessor(C)
```

Binary tree concepts: Elements of Programming

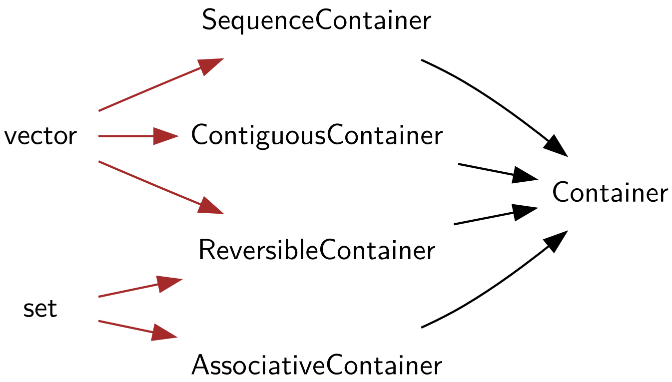
EoP defines a *BifurcateCoordinate* concept that fits with the definition given in Knuth and is a recursive definition of a binary tree.

```
bool empty(Vertex, Graph)
bool has_left_successor(Vertex, Graph)
bool has_right_successor(Vertex, Graph)
Vertex left_successor(Vertex, Graph)
Vertex right_successor(Vertex, Graph)
```

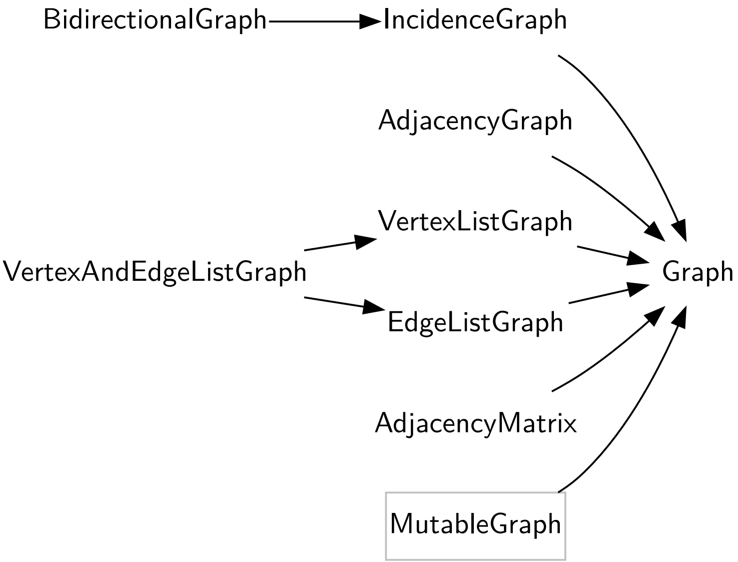
```
bool has_predecessor(Vertex, Graph)
Vertex predecessor(Vertex, Graph)
```

But in Boost.Graph we need to place the recursive structure in a class for algorithms to operate on.

Container Concepts

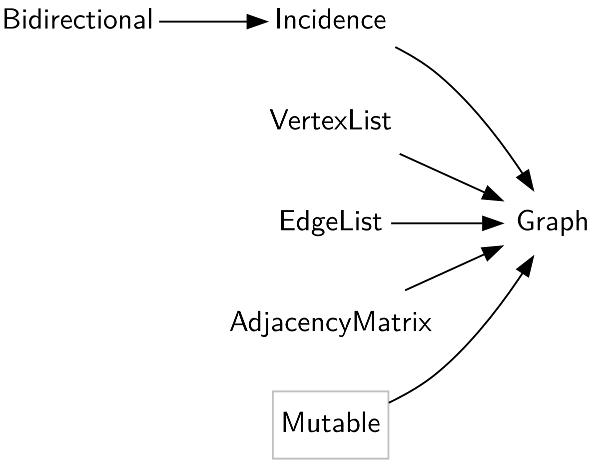


Graph Concepts

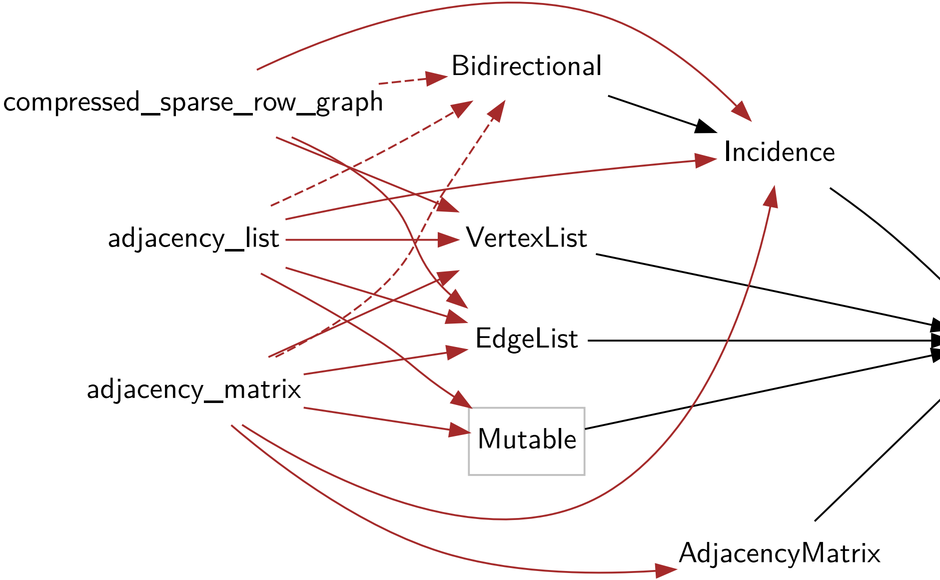


Graph Concepts Simplified

Let's drop *AdjacencyGraph* and *VertexAndEdgeListGraph*, and drop the *Graph* suffix.

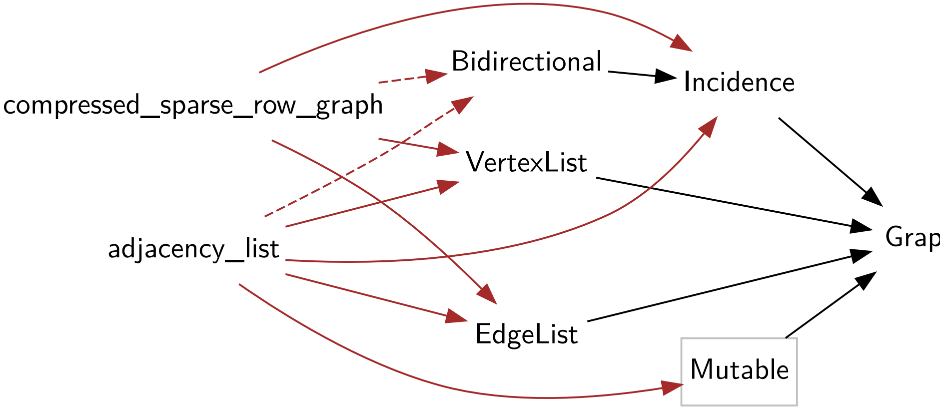


Graph Concepts and Structures Detailed



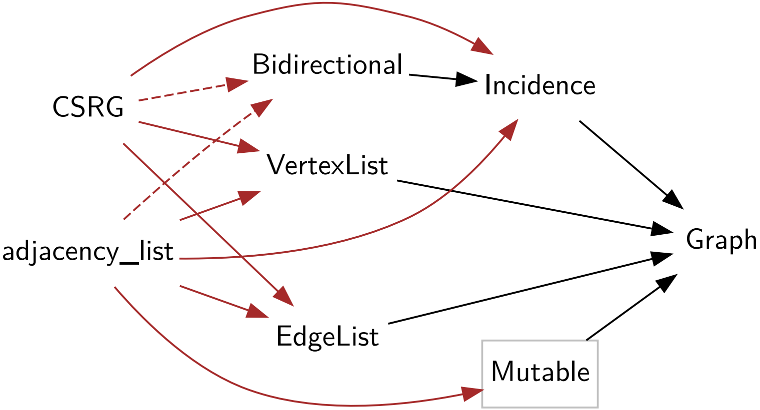
Graph Concepts and Structures Detailed

Let's drop *AdjacencyMatrix* concept because we're not going to model it.

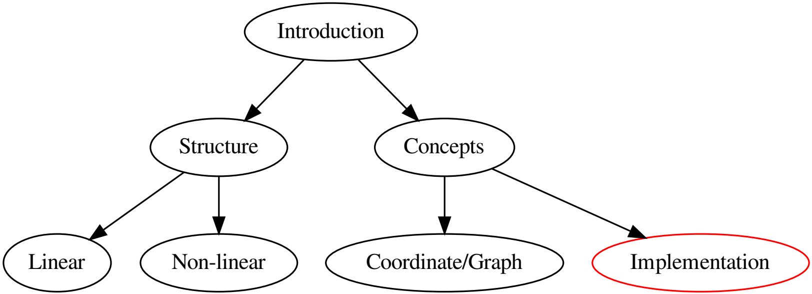


Graph Concepts and Structures Detailed

This is the subset of Boost.Graph structures and concepts we are really interested in.



Outline


Introduction
Structure
Concepts
Linear
Non-linear
Coordinate/Graph
Implementation

Implementation: Primary class interface

```
template <bool Predecessor, typename Vertex = std::size_t>
class binary_tree;

template <typename Vertex>
class binary_tree<false, Vertex>
  : public detail::binary_tree_base<Vertex,
    detail::binary_tree_forward_node<binary_tree<false, Vertex> > >

template <typename Vertex>
class binary_tree<true, Vertex>
  : public detail::binary_tree_base<Vertex,
    detail::binary_tree_bidirectional_node<binary_tree<true, Vertex> > >
```

Implementation: Tree node classes

```
template <typename BinaryTree>
struct binary_tree_forward_node {
  using vertex_descriptor = vertex_descriptor_t<BinaryTree>;

  binary_tree_forward_node() {
    fill(successors, graph_traits<BinaryTree>::null_vertex());
  }

  array<vertex_descriptor, 2> successors;
};

template <typename BinaryTree>
struct binary_tree_bidirectional_node
  : binary_tree_forward_node<BinaryTree>
{
  using vertex_descriptor = vertex_descriptor_t<BinaryTree>;

  binary_tree_bidirectional_node(
    vertex_descriptor predecessor = graph_traits<BinaryTree>::null_vertex())
    : predecessor(predecessor)
  {}

  vertex_descriptor predecessor;
};
```

Implementation: Base class data and construction

```
template <typename Vertex, typename Node>
class binary_tree_base
{
protected:
  std::vector<Node> nodes;
  std::vector<Vertex> free_list;
public:
  typedef Vertex vertex_descriptor;
  typedef std::pair<vertex_descriptor, vertex_descriptor> edge_descriptor;
  typedef disallow_parallel_edge_tag edge_parallel_category;
  typedef std::size_t degree_size_type;
  typedef std::size_t vertices_size_type;

  BOOST_STATIC_CONSTEXPR
  vertex_descriptor null_vertex() {
    return vertex_descriptor(-1);
  }

public:
  binary_tree_base(Vertex n) : nodes(n), free_list{{n}} {}
```

Implementation: ForwardBinaryTree concept

```
BOOST_concept(ForwardBinaryTree,(G))
    : Graph<G>
{
  BOOST_CONCEPT_USAGE(ForwardBinaryTree) {
    t = has_left_successor(u, g);
    t = has_right_successor(u, g);
    v = left_successor(u, g);
    v = right_successor(u, g);
    t = empty(u, g);
    const_constraints(g);
  }
  void const_constraints(G const &g) {
    t = has_left_successor(u, g);
    t = has_right_successor(u, g);
    v = left_successor(u, g);
    v = right_successor(u, g);
    t = empty(u, g);
  }
  bool t;
  G g;
  typename graph_traits<G>::vertex_descriptor u, v;
};
```

Implementation: BidirectionalBinaryTree concept

```
BOOST_concept(BidirectionalBinaryTree,(G))
  : ForwardBinaryTree<G>
{
  BOOST_CONCEPT_USAGE(BidirectionalBinaryTree) {
    t = has_predecessor(u, g);
    t = predecessor(u, g);
    u = root(u, g);
    const_constraints(g);
  }

  void const_constraints(G const &g) {
    t = has_predecessor(u, g);
    t = predecessor(u, g);
    u = root(u, g);
  }

  bool t;
  G g;
  typename graph_traits<G>::vertex_descriptor u;
};
```

Implementation: Mutable BinaryTree concepts

```
BOOST_concept(MutableForwardBinaryTree,(G))
  : ForwardBinaryTree<G>
{
  BOOST_CONCEPT_USAGE(MutableForwardBinaryTree) {
    e = add_left_edge(u, v, g);
    e = add_right_edge(u, v, g);
    // TODO: remove_left_edge, remove_right_edge
  }
  G g;
  typename graph_traits<G>::vertex_descriptor u, v;
  typename graph_traits<G>::edge_descriptor e;
};
```

Implementation: Mutable BinaryTree concepts

```
BOOST_concept(MutableForwardBinaryTree,(G))
  : ForwardBinaryTree<G>
{
  BOOST_CONCEPT_USAGE(MutableForwardBinaryTree) {
    e = add_left_edge(u, v, g);
    e = add_right_edge(u, v, g);
    // TODO: remove_left_edge, remove_right_edge
  }
  G g;
  typename graph_traits<G>::vertex_descriptor u, v;
  typename graph_traits<G>::edge_descriptor e;
};
```

Don't need a *MutableBidirectional* because removing a predecessor is removing someone else's left/right successor.

Implementation: IncidenceGraph concept (1/3)

```
class out_edge_iterator
  : public boost::iterator_adaptor<out_edge_iterator,
                                   vertex_descriptor const *,
                                   edge_descriptor,
                                   forward_traversal_tag,
                                   edge_descriptor>
{
  vertex_descriptor const *last;
  vertex_descriptor source;

public:
  out_edge_iterator(Vertex const *first, Vertex const *last, Vertex source)
    : out_edge_iterator::iterator_adaptor_(first), last(last),
      source(source)
  {
    BOOST_ASSERT(source != null_vertex());
    post_increment();
  }

private:
  edge_descriptor dereference() const
  {
    return edge_descriptor(source, *this->base_reference());
  }
```

Implementation: IncidenceGraph concept (2/3)

```
  void post_increment()
  {
    while (this->base_reference() != last
           && *this->base_reference() == null_vertex()) {
      this->base_reference()++;
    }
  }

  void increment()
  {
    this->base_reference()++;
    post_increment();
  }

  friend class boost::iterator_core_access;
};

friend
vertex_descriptor source(edge_descriptor e, binary_tree_base const &) {
  return e.first;
}

friend
vertex_descriptor target(edge_descriptor e, binary_tree_base const &) {
  return e.second;
}
```

Implementation: IncidenceGraph concept (3/3)

```
friend
std::pair<out_edge_iterator, out_edge_iterator>
out_edges(vertex_descriptor u, binary_tree_base const &g)
{
  auto const &successors = g.nodes[u].successors;

  return std::make_pair(out_edge_iterator(boost::begin(successors),
                                          boost::end(successors), u),
                        out_edge_iterator(boost::end(successors),
                                          boost::end(successors), u));
}

friend
degree_size_type
out_degree(vertex_descriptor v, binary_tree_base const &g)
{
  return 2 - count(g.nodes[v].successors, null_vertex());
}
```

Implementation: BidirectionalGraph concept (1/2)

```
private:
  struct make_in_edge_descriptor {
    make_in_edge_descriptor(vertex_descriptor target) : target(target) {}

    edge_descriptor operator()(vertex_descriptor source) const {
      return edge_descriptor(source, target);
    }
    vertex_descriptor target;
  };

public:
  typedef transform_iterator<make_in_edge_descriptor, vertex_descriptor const *,
          edge_descriptor> in_edge_iterator;

  friend
  std::pair<in_edge_iterator, in_edge_iterator>
  in_edges(vertex_descriptor u, binary_tree const &g) {
    auto const p = has_predecessor(u, g);
    return std::make_pair(in_edge_iterator(&g.nodes[u].predecessor,
                                           make_in_edge_descriptor(u)),
                          in_edge_iterator(&g.nodes[u].predecessor + p,
                                           make_in_edge_descriptor(u)));
  }
```

Implementation: BidirectionalGraph concept (2/2)

```
friend
degree_size_type
in_degree(vertex_descriptor u, binary_tree const &g)
{
  return has_predecessor(u, g);
}

friend
degree_size_type
degree(vertex_descriptor u, binary_tree const &g)
{
  return in_degree(u, g) + out_degree(u, g);
}
```

Implementation: VertexListGraph concept (1/3)

How to iterate over vertices in a sparse array?

Implementation: VertexListGraph concept (1/3)

How to iterate over vertices in a sparse array?

```
struct vertex_iterator
  : public iterator_facade <vertex_iterator, vertex_descriptor,
        multi_pass_input_iterator_tag, vertex_descriptor const &> {
  typedef iterator_facade<vertex_iterator, vertex_descriptor,
        multi_pass_input_iterator_tag, vertex_descriptor const &> super_t;
  typedef typename super_t::value_type value_type;
  typedef typename super_t::reference reference;

  vertex_descriptor last;
  std::stack<vertex_descriptor> traversal;
  binary_tree_base const *g;
public:
  vertex_iterator(binary_tree_base const &g) : g(&g) {}

  vertex_iterator(vertex_descriptor start, binary_tree_base const &g)
    : last(g.null_vertex()), g(&g)
  {
    traversal.push(start);
    while (has_left_successor(traversal.top(), g))
      traversal.push(left_successor(traversal.top(), g));
  }
```

Implementation: VertexListGraph concept (1/3)

How to iterate over vertices in a sparse array?

```
struct vertex_iterator
  : public iterator_facade <vertex_iterator, vertex_descriptor,
        multi_pass_input_iterator_tag, vertex_descriptor const &> {
  typedef iterator_facade<vertex_iterator, vertex_descriptor,
        multi_pass_input_iterator_tag, vertex_descriptor const &> super_t;
  typedef typename super_t::value_type value_type;
  typedef typename super_t::reference reference;

  vertex_descriptor last;
  std::stack<vertex_descriptor> traversal;
  binary_tree_base const *g;
public:
  vertex_iterator(binary_tree_base const &g) : g(&g) {}

  vertex_iterator(vertex_descriptor start, binary_tree_base const &g)
    : last(g.null_vertex()), g(&g)
  {
    traversal.push(start);
    while (has_left_successor(traversal.top(), g))
      traversal.push(left_successor(traversal.top(), g));
  }
```

In-order traversal. Should have used pre-order.

Implementation: VertexListGraph concept (2/3)

```
reference dereference() const {
  return traversal.top();
}

void increment() {
  if (has_right_successor(traversal.top(), *g)) {
    if (right_successor(traversal.top(), *g) != last) {
      traversal.push(right_successor(traversal.top(), *g));
      while (has_left_successor(traversal.top(), *g))
        traversal.push(left_successor(traversal.top(), *g));
      return;
    }
  }

  do {
    last = traversal.top();
    traversal.pop();
  } while (!traversal.empty()
         && (!has_right_successor(traversal.top(), *g)
             || right_successor(traversal.top(), *g) == last));
}
```

Implementation: VertexListGraph concept (1/3)

```
  bool equal(vertex_iterator const &other) const
  {
    BOOST_ASSERT(g == other.g);

    if (traversal.empty())
      return other.traversal.empty();

    if (other.traversal.empty())
      return false;

    return traversal.top() == other.traversal.top();
  }
};

friend
std::pair<vertex_iterator, vertex_iterator>
vertices(binary_tree_base const &g)
{
  if (num_vertices(g) == 0)
    return std::make_pair(vertex_iterator(g), vertex_iterator(g));
  auto start = default_starting_vertex(g);
  return std::make_pair(vertex_iterator(start, g), vertex_iterator(g));
}
```

Implementation: Summary

A binary tree can satisfy all of the Graph concepts. [1]

[1] *EdgeList* yet to be demonstrated.

Algorithms

1. create_tree & create_binary_tree
2. depth-first search (EoP)
3. isomorphism (EoP)

Algorithms

1. create_tree & create_binary_tree
2. depth-first search (EoP)
3. isomorphism (EoP)

With the exception of `create_tree`, these algorithms use the *BinaryTree* concept (not the general *Graph* concepts).

Implementation: Algorithms & Benchmarks

Google Benchmark.

Implementation: Algorithms & Benchmarks

Google Benchmark.
Run on (8 X 3700 MHz CPU s)
CPU Caches:

- L1 Data 32K (x4)
- L1 Instruction 32K (x4)
- L2 Unified 256K (x4)
- L3 Unified 6144K (x1)

Benchmarks

Google Benchmark.
Run on (8 X 3700[2] MHz CPU s)
CPU Caches:

- **L1 Data 32K** (x4)
- L1 Instruction 32K (x4)
- L2 Unified 256K (x4)
- L3 Unified 6144K (x1)

model name : Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz

[2] Turbo Boost

Benchmarks

Google Benchmark.
Run on (8 X 3700[2] MHz CPU s)
CPU Caches:

- **L1 Data 32K** (x4)
- L1 Instruction 32K (x4)
- L2 Unified 256K (x4)
- L3 Unified 6144K (x1)

model name : Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz
The results are noisy, but the signal is large.

[2] Turbo Boost

Algorithm: create_tree

```
template <typename Graph>
void create_tree(Graph &tree, vertex_descriptor_t<Graph> weight)
{
  BOOST_ASSERT(weight >= 0);
  if (weight == 0) return;
  if (weight == 1) {
    add_vertex(tree);
    return;
  }

  typedef vertex_descriptor_t<Graph> vertex_descriptor;
  vertex_descriptor parent = 0;
  for (vertex_descriptor child = 1; child != weight; child++) {
    add_edge(parent, child, tree);
    if (!(child & 1))
      parent++;
  }
}
```

Algorithm: create_tree

```
template <typename Graph>
void create_tree(Graph &tree, vertex_descriptor_t<Graph> weight)
{
  BOOST_ASSERT(weight >= 0);
  if (weight == 0) return;
  if (weight == 1) {
    add_vertex(tree);
    return;
  }

  typedef vertex_descriptor_t<Graph> vertex_descriptor;
  vertex_descriptor parent = 0;
  for (vertex_descriptor child = 1; child != weight; child++) {
    add_edge(parent, child, tree);
    if (!(child & 1))
      parent++;
  }
}
```

Uses *MutableGraph* concept.

Algorithm: create_binary_tree

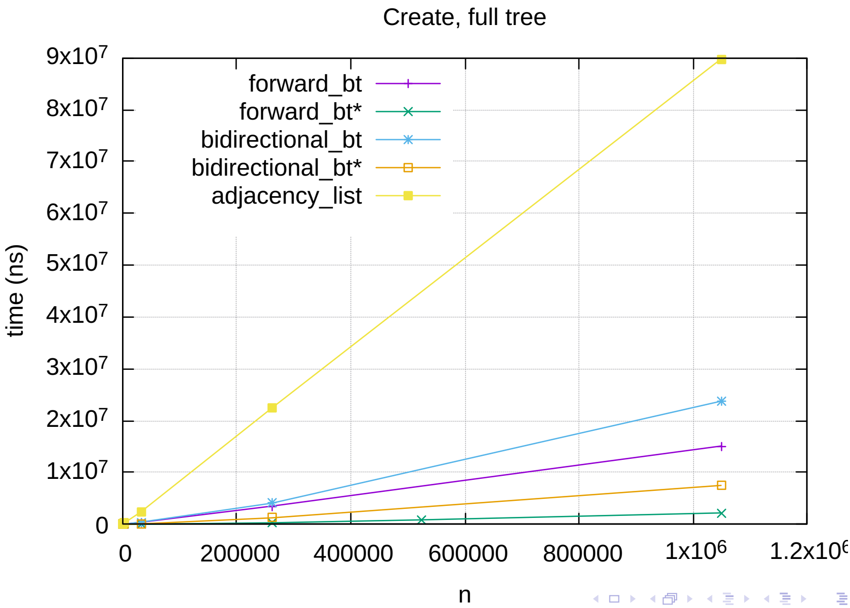
```
template <typename BinaryTree>
void create_binary_tree(BinaryTree &tree,
                        vertex_descriptor_t<BinaryTree> weight)
{
  BOOST_ASSERT(weight >= 0);

  tree = BinaryTree(weight);
  typedef vertex_descriptor_t<BinaryTree> vertex_descriptor;
  vertex_descriptor parent = 0;
  for (vertex_descriptor child = 1; child < weight; child++)
  {
    if (child % 2 == 1)
      add_left_edge(parent, child, tree);
    else
      add_right_edge(parent++, child, tree);
  }
}
```

Algorithm: create_binary_tree

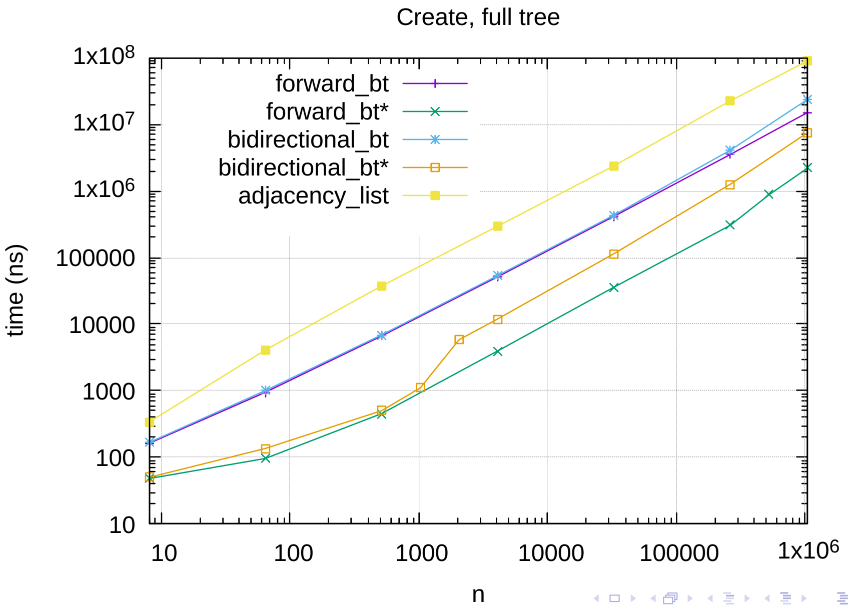
```
template <typename BinaryTree>
void create_binary_tree(BinaryTree &tree,
                        vertex_descriptor_t<BinaryTree> weight)
{
  BOOST_ASSERT(weight >= 0);

  tree = BinaryTree(weight);
  typedef vertex_descriptor_t<BinaryTree> vertex_descriptor;
  vertex_descriptor parent = 0;
  for (vertex_descriptor child = 1; child < weight; child++)
  {
    if (child % 2 == 1)
      add_left_edge(parent, child, tree);
    else
      add_right_edge(parent++, child, tree);
  }
}
```

Uses *MutableForwardBinaryTree* concept. This is the * algorithm on the following benchmark graph.

Benchmarks: Create tree (linear example)



Benchmarks: Create tree



Implementation: Depth-first search, Forward

```
template <typename BinaryTree, typename Visitor>
Visitor traverse_nonempty(vertex_descriptor_t<BinaryTree> u,
                          BinaryTree const &g, Visitor vis)
{
  vis(visit::pre, u);
  if (has_left_successor(u, g))
    vis = traverse_nonempty(left_successor(u, g), g, vis);
  vis(visit::in, u);
  if (has_right_successor(u, g))
    vis = traverse_nonempty(right_successor(u, g), g, vis);
  vis(visit::post, u);
  return vis;
}
```

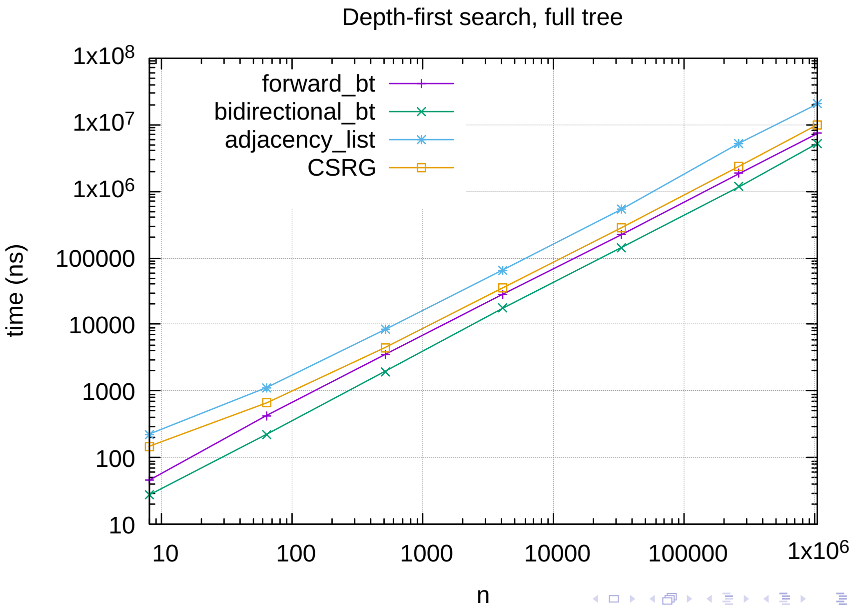
Implementation: Depth-first search, Bidirectional (1/2)

```
template <typename BinaryTree>
int traverse_step(visit &vis, vertex_descriptor_t<BinaryTree> &u,
                  BinaryTree const &g)
{
  switch (vis) {
  case visit::pre:
    if (has_left_successor(u, g)) {
                            u = left_successor(u, g);    return 1;
    } vis = visit::in;                                    return 0;
  case visit::in:
    if (has_right_successor(u, g)) {
      vis = visit::pre; u = right_successor(u, g);       return 1;
    } vis = visit::post;                                  return 0;
  case visit::post:
    if (is_left_successor(u, g)) {
      vis = visit::in;
    }                       u = predecessor(u, g);       return -1;
  }
}
```

Implementation: Depth-first search, Bidirectional (2/2)

```
template <typename BinaryTree, typename Visitor>
Visitor traverse(vertex_descriptor_t<BinaryTree> u,
                 BinaryTree const &g, Visitor vis)
{
  if (empty(u, g))
    return vis;
  auto root = u;
  visit v = visit::pre;
  vis(v, u);
  do {
    traverse_step(v, u, g);
    vis(v, u);
  } while (u != root || v != visit::post);
  return vis;
}
```

Benchmarks: Depth-first search



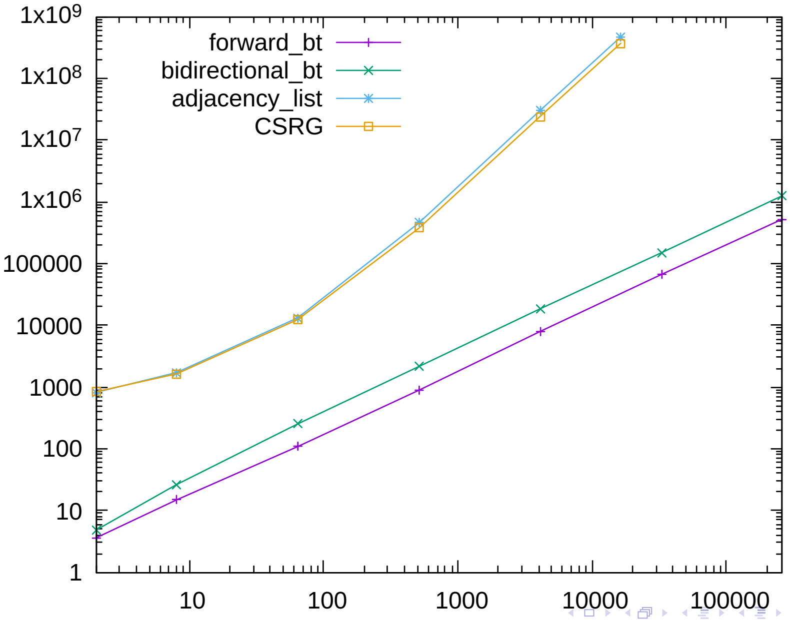
Implementation: Isomorphism, Forward

```
template <typename BinaryTree0, typename BinaryTree1>
bool bifurcate_isomorphic_nonempty(
  vertex_descriptor_t<BinaryTree0> u, BinaryTree0 const &g,
  vertex_descriptor_t<BinaryTree1> v, BinaryTree1 const &h)
{
  if (has_left_successor(u, g)) {
    if (has_left_successor(v, h)) {
      if (!bifurcate_isomorphic_nonempty(left_successor(u, g), g,
                                          left_successor(v, h), h))
        return false;
    } else
      return false;
  } else if (has_left_successor(u, g))
    return false;

  if (has_right_successor(u, g)) {
    if (has_right_successor(v, h)) {
      if (!bifurcate_isomorphic_nonempty(right_successor(u, g), g,
                                          right_successor(v, h), h))
        return false;
    } else
      return false;
  } else if (has_right_successor(u, g))
    return false;
  return true;
}
```

Implementation: Isomorphism, Bidirectional

```
template <typename BinaryTree0, typename BinaryTree1>
bool bifurcate_isomorphic(
                    vertex_descriptor_t<BinaryTree0> u, BinaryTree0 const &g,
                    vertex_descriptor_t<BinaryTree1> v, BinaryTree1 const &h)
{
  BOOST_CONCEPT_ASSERT((concepts::BidirectionalBinaryTreeConcept<BinaryTree0>));
  BOOST_CONCEPT_ASSERT((concepts::BidirectionalBinaryTreeConcept<BinaryTree1>));

  if (empty(u, g)) return empty(v, h);
  if (empty(v, h)) return false;
  auto root0 = u;
  visit visit0 = visit::pre;
  visit visit1 = visit::pre;
  while (true) {
    traverse_step(visit0, u, g);
    traverse_step(visit1, v, h);
    if (visit0 != visit1) return false;
    if (u == root0 && visit0 == visit::post) return true;
  }
}
```

Benchmarks: Isomorphism



References and further reading

Knuth, D.E. (1997)
The Art of Computer Programming. Volume 1
Addison-Wesley

Siek, J., Lumsdaine, A. & Lee, L.Q. (2002)
The Boost Graph Library: User Guide and Reference Manual
Addison-Wesley

Stepanov, A. & McJones, P. (2009)
Elements of Programming
Addison-Wesley

Navarro, G. (2016)
Compact Data Structures - A Practical Approach
Cambridge University Press

Thank you

Thank you

And my very patient and supportive wife.

What's next

- Complete it.
- Compact structure: stored in $2n$ bits.
- `https://github.com/boostorg/graph/pull/139`

The end

https://github.com/boostorg/graph/pull/139