Integration of Graph Iterators into LEDA

Marco Nissen

MPI–I–1999–005 September 1999

Author's Address

Marco Nissen
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
marco@mpi-sb.mpg.de
http://www.mpi-sb.mpg.de/~marco/

Abstract

This paper explains some implementation details of graph iterators and data accessors in LEDA. It shows how to create new iterators for new graph implementations such that old algorithms can be re–used with new graph implementations as long as they are based on graph iterators and data accessors.

Keywords

LEDA, Graph Iterators, Generic Programming, Data Accessor, Algorithm Engineering, Design Patterns

Contents

1 Introduction

Those who are familiar with the STL[1] might have noticed that in preliminary versions of LEDA (< 3.6) there were no iterator objects in the `graph` class. Actually, it was straightforward to implement STL-like iterators for collections like a LEDA-`list` or -`dictionary`. This mechanism failed for the more complex graph classes because a graph is not a linear structure[2].

Before continuing, I will sketch briefly the main concept of STL. The idea is to decouple algorithms from data structures. The inventors of STL made the observation that algorithms and data structures can be decoupled by generalizing the idea of pointers: for simple c-arrays it is straightforward to use pointers to traverse the structure and to change data. For more complex containers such as lists or vectors, they created a pointer-like object that is able to traverse the structure and permits to change associated data.

For example, the generic quick sort algorithm gets two iterators and an ordering function. The iterators of random access kind (they enable direct access through indexing) indicate the range in the interval of the collection which has to be sorted. Another nice feature is the automatic selection of the appropriate algorithm for each kind of underlying data structure (i.e. iterator for a collection). For example, you do not have to specify that you want to perform a quick sort, but it suffices to use random access iterators such that the compiler deterministically selects[3] the quick sort algorithm.

In this paper I will distinguish structural access (iterating over a structure) and data access (requesting or change data associated to the elements of the structure).

[1]The standard template library [MS96] is a generic collection of simple data structures and algorithms.

[2]Acyclic graphs may be seen as linearly after sorted topologically, but I consider the general case, here.

[3]This is done via type tags: `RandomAccessIterator`, `ForwardIterator` ...— an iterator can be of one of the given types and the compiler will select at compile-time the most appropriate algorithm, i.e. the most efficient one.

The key idea behind the graph iterator implementation lies in the distinction between structural access and data access. The former is done by using graph iterators, and the latter by using data accessors. The package provides several types of iterators:

- node iterators (`NodeIt`),
- edge iterator (`EdgeIt`),
- and adjacency iterators (`AdjIt`, `OutAdjIt`, `...`) .

A node iterator traverses the list of nodes in a linear fashion and an edge iterator does the same for the list of edges.

Adjacency iterators are different: their list of traversable edges depend on which node is considered as "fixed" and of what kind the iterator actually is.

For example, an "adjacency iterator for outgoing edges" for a fixed node n traverses all outgoing edges of that node. All iterators return a handle (for LEDA node iterators, this is exactly `node`) that identifies the current state of the iterator.

The package also provides several types of data accessors (see also [KW97] or [N99]):

- for LEDA containers (`node_array_da`, `node_map_da`),
- for parameterized LEDA graphs (`node_member_da`)
- constant accessors (`constant_da`)
- pointer-to-member-variable accessors (`MemberAccessor`)
- pointer-to-member-method accessors (`MethodAccessor`)
- complex calculation accessors
- ...

Only a fraction of the implementation has been integrated into the LEDA manual tree because the other classes seemed to be of special interest. However, the implementation is contained in the headerfile `graphiterator.h` that is shipped with the LEDA-installation.

The idea is simply the following: for a given data accessor `col` that associates colours with nodes you can write `get(col,it)` to get the colour associated with the current handle of iterator `it` [4].

[4] Actually, it is possible to decide on the kind of data accessor which part of the given iterator is used as a key for storing the associated attribute. For example, adjacency iterators have two possible keys: the fixed node and the current edge.

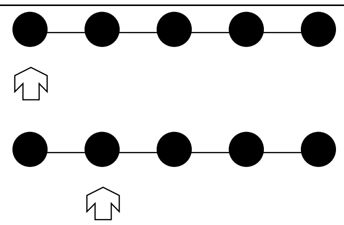
There are several graph algorithms that were implemented solely with graph iterators and data accessors. The benefit was that they were completely decoupled from implementation details, and could be used in different future environments, where the underlying graph implementation of data implementation might be completely different to LEDA's implementation.

To help here, there are implementation-independent versions of iterators which can be adjusted to fit into any possible environment, e.g. like LEDA.

Chapters 2 and 3 describe the implementation of the data structures that are the key tool for decoupling algorithms from underlying graph implementations. Chapter 4 shows how iterators may be adapted to other graph implementations. Chapter 5 shows how an algorithm can be implemented with the help of graph iterators and data accessors. Chapter 6 explains how everything was integrated into LEDA. Chapter 7 gives some remarks on the paper. Finally, chapter 8 explains some mechanism and idioms of `C++`.

2 Iterators

Figure 2.1 The black circle represents objects, the arrow an iterator. Here are two situations of an iterator. In the first situation it points to the first element, in the second to the second element.



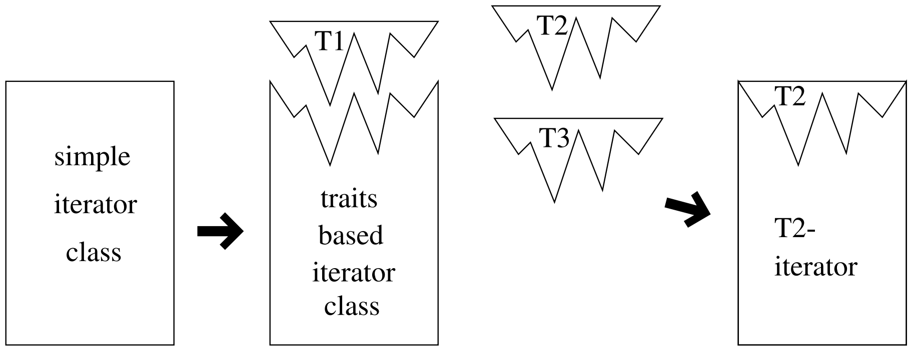
An iterator (see figure 2.1) is simply an object that knows how to traverse a structure in a certain manner. Actually, it is possible to write for each kind of implementation an appropriate iterator class and to instantiate iterator objects from it. Figure 2.1 shows a sample iterator implementation, but unfortunately, it is *only* suitable for graphs of type `graph` and nodes of type `node`. Clearly, if you write a base class `graph` and inherit all variations from it, you can re–use this iterator, but it remains the problem that the interface might differ. However, it is also possible to write interface adapter classes, but in this situation we decided to do something which is intentionally the same but differs in the way of implementation.

The implementation strategy was to use the traits technique (see the appendix section 8.2). Here, we have a set of basic iterator skeletons, a set of traits classes for different underlying graph implementations and get a set of iterator classes (see figure 2.2).

```
class node_iterator {
   node pointer;
   graph _G;
public:
   node_iterator(graph G, node n)
      : pointer(n), _G(G) { }
   node_iterator operator++() {
      pointer=_G.succ_node(pointer);
      return *this;
   }
   bool valid() {
      return pointer ≠ nil; }
   node get_node() {
      return pointer; }
};
```

C++-code 2.1: sample iterator implementation

Figure 2.2 Using the traits technique to re–use basic iterator capabilities for different underlying data structures. Each traits class has own type definitions and basic operations on them.



Now we will see how adjacency iterators for outgoing nodes are implemented in C++ using the traits class mechanism. See figure 2.2 for the big picture of iterator implementation. Firstly, a part of the traits-class[1] already specialized for the use with LEDA (see figure 2.2).

There are several type definitions, which provide uniform access to the node, edge or graph type. Then there is a list of static methods, which perform basic

[1]The prefix `T_` in `T_Node`, `T_Edge` and `T_Graph` means "this is a traits type". In the following only constant references to the underlying graph structure are exchanged, but not the complete graph (i.e. in the code, we use `T_Graph` instead of `const T_Graph&`).

graph operations. For example, `first_node()` gives the first node in the set of (ordered) nodes of the given graph G. If nodes are not pointers, there must be an alternative implementation for testing if the current node is valid or not. This is done with the functions `node_null()` and `is_node_null(v)`, where the latter tests if v is invalid. The first incident edge in the list of incident edges will be returned by function `first_incident_edge(G,v)`.

Another important function for adjacency iterators is the one which returns a "current adjacency iterator". This function inspects the current incident edge, and if there is any, it tries to follow its direction to the opposite node of the fixed node and returns a new adjacency iterator of the same type but with this opposite node as fixed node.

```
class OutNodeEdgeTrait {
public:
   typedef edge T_Edge;
   typedef node T_Node;
   typedef graph T_Graph;
   static T_Node node_null() {
      return nil ; }
   static bool is_node_null(T_Node v) {
      return v ≡ nil ; }
   static void forward(T_Graph& G, T_Node&, T_Edge& e) {
      e=G.adj_succ(e); }
   static T_Node first_node(T_Graph& G) {
      return G.first_node(); }
   static T_Edge first_incident_edge(T_Graph& G, T_Node v) {
      if (is_node_null(v)) return edge_null();
      return G.first_adj_edge(v);
   }
   static void curr_adj(T_Node& n, T_Edge& e, T_Graph& G) {
      if (is_edge_null(e)) return ;
      n=target(e);
      e=G.first_adj_edge(n); }
   ...
   };
```

C++-code 2.2: traits class for adjacency iterator for outgoing edges

The traits class in this concept will be the only connection to the underlying graph implementation, i.e. for each implementation you have to write new traits classes instead of writing new iterators.

```
template¡class traits¿ class AdjIt {
public:
   typedef typename traits::T_Edge edgetype;
   typedef typename traits::T_Node nodetype;
   typedef typename traits::T_Graph graphtype;
private:
   nodetype _n;
   edgetype _e;
   graphtype& _G;
   typedef AdjIt<traits> self;
public:
   AdjIt(graphtype& G) : _G(G) ,_n(traits::first_node(G)) {
      traits::assign(_e,traits::first_incident_edge(G,_n)); }
   void update(nodetype n) {
      traits::assign(_n,n);
      traits::assign(_e,traits::first_incident_edge(_G,_n)); }
   bool has_node() const { return !traits::is_node_null(_n); }
   bool valid() const { return !traits::is_edge_null(_e); }
   edgetype get_edge() const { return _e; }
   nodetype get_node() const { return _n; }
   self curr_adj() const {
      self temp(*this);
      traits::curr_adj(temp._n, temp._e, temp._G);
      return temp; }
   self& operator ++( ) {
      traits::forward(_G,_n,_e);
      return *this; }
   ...
};
```

C++-code 2.3: adjacency iterator class which takes a traits class as template parameter

Actually, `C++` provides no method to assure the correct use of types and functions in traits classes that are used as template parameter classes (sometimes called constraint parameterization). The only restriction lies in the use of correct names, and this will only be checked at compile time if the compiler tries to access the name in the traits class. If a static function or type definition is never used, it cannot be guaranteed that all required functions exist. This is an essential problem in libraries, because the traits classes have to be bullet-proof.

In practice, this means that traits-based iterators only work correctly as long as they use traits classes that are correctly implemented according to the traits specification of the iterator class.

2.1 Safe Iterators

It is sometimes hard to say what happens, if a node or edge is deleted while some iterator referring to that item is currently in use. To help with this situation, we need to keep a list of all iterators whenever such an update–event occurs (deletion of a node or edge). Then, all the incidental iterators have to be informed to become 'invalid' or to move along the associated sequence.

The list of all iterators will be managed by a structure called 'safe graph' which contains also a reference to a regular graph. Actually, this structure could have been inherited from the LEDA-graph base class, but at implementation time, we used the traits mechanism to build a graph independent structure. A re–implementation of this class could be very useful, because the graph class now uses a very powerful event concept.

These are the collaborating classes:

- `SafeIteratorBase:` base class of all iterators
- `SafeGraph:` contains a reference to a graph and all iterators that are currently in use
- `SafeNodeIt`, `SafeEdgeIt`, `SafeInAdjIt`, `SafeOutAdjIt`, `SafeAdjIt` : iterators that are enriched by update-methods which are called in case of modifications in the graph

The key idea behind the safe iterator implementation lies in the cooperation of type of iterator, type of reaction and event–type.

- **type of iterator:** a safe iterator can be a node iterator, an edge iterator or an adjacency iterator
- **type of reaction:** a safe iterator has a state variable (initialized at construction time) that describes how to react on update events. There the following reactions:
  - the iterator becomes invalid
  - the iterator moves forward in the associated sequence
  - the iterator moves backward in the associated sequence

  Other scenarios maybe also possible
- **event–type:** deletion of a node or an edge. This again depends on the type of iterator:

  Events that lead to the reaction as described above:
  - deletion of the node of a node iterator
  - deletion of the edge of an edge iterator
  - deletion of the source or target node of an edge iterator

  Other more complicated events:
  - deletion of the fixed node of an adjacency iterator: make the iterator invalid
  - deletion of the adjacent node of an adjacency iterator: move along the sequence (forward or backward)

3 Data Accessors

A data accessor is simply a generalized interface for accessing associated attributes for given objects. They are implemented as instances of a data accessor class for which there are two functions:

- '**get**': returns the associated value for a given object
- '**set**': sets the associated value for a given object to the given value

More specifically, we have two function templates:

- `T get(DA da, Iter it)`, where da is the current data accessor and it the iterator
- `void set(DA da, Iter it, T value)`, where da is the current data accessor, it the iterator and $value$ the new value

Several data accessors use a so–called object accessor to access the item that the iterator refers to. An object accessor is simply a data accessor which only returns an associated value, namely for each iterator a certain member value. For example, a node object accessor returns the fixed node for an adjacency iterator, an edge object accessor the current edge of an adjacency iterator.

More specifically, an object accessor of type `OA` is required to come with a function `item get_object(OA oa, Iter it);` where `item` is the item that is appropriate e.g. for the handler object **??** of a handler accessor (see figure 3.1).

Since we are using iterators for graphs, we are only interested in item types like node or edge and therefore we require for iterators, which refer to nodes to have the type definition `Iter::nodetype` and the member function `Iter::nodetype Iter::get_node() const;` . Consequently, iterators that refer to edges need to have the type definition `Iter::edgetype` and member function `Iter::edgetype Iter::get_edge() const;` .

Here, adjacency iterators must provide both possibilities because they always refer to a fixed node and an incident edge.

All implemented graph iterators[1] provide the necessary member functions, so we can develop two object accessors, a node and an edge accessor. See figure 3.1 for an example that works with member template capable compilers.

```
class NodeAccessor {
template<class Iter>
  typename Iter::nodetype get_object(Iter it) {
    return it.get_node(); }
};

class EdgeAccessor {
template<class Iter>
typename Iter::edgetype get_object(Iter it) {
  return it.get_edge(); }
};
```

C++-code 3.1: two examples of object accessors

At the time of writing the code, our compilers was not able to handle member templates. Therefore, we had to write a workaround to use object accessors (see figure 3.2 for the workaround).

```
struct NodeAccessor { } nodeacc;
// object accessor of type NodeAccessor
struct EdgeAccessor { } edgeacc;
// object accessor of type EdgeAccessor

template<class Iter>
typename Iter::nodetype get_object(NodeAccessor, Iter it) {
  return it.get_node(); }

template<class Iter>
typename Iter::edgetype get_object(EdgeAccessor, Iter it) {
  return it.get_edge(); }
```

C++-code 3.2: workaround for object accessors

[1] See the graphiterator part in LEDA.

A handler accessor is a data accessor that hides internal use of maps, arrays, or other containers (see figure 3.3). We call such objects handler objects.

```
template<class T,class Handler, class ObjectAccessor>
class HandlerAccessor {
public:
   typedef T value_type;
private:
   Handler& i_handler;
   ObjectAccessor i_sa;
public:
   HandlerAccessor( Handler& handler, ObjectAccessor sa) :
      i_handler(handler), i_sa(sa) { }

template<class Iter> T get(Iter it) {
      return i_handler[get_object(i_sa,it)]; }

template<class Iter> void set(Iter it, T value) {
      i_handler[get_object(i_sa,it)]=value; }
};
```

C++-code 3.3: a member template version of a handler accessor

In the current implementation there is a workaround version (see figure 3.4).

```
template<class T,class Handler,class ObjectAccessor>
class HandlerAccessor {
public:
   typedef T value_type;
private:
   Handler& i_handler;
   ObjectAccessor i_sa;
public:
   HandlerAccessor(Handler& handler, ObjectAccessor sa) :
      i_handler(handler), i_sa(sa) { }
   ObjectAccessor internal_sa() {
      return i_sa; }
   Handler& internal_map() {
      return i_handler; }
};

template<class Iter,class Handler, class ObjectAccessor,class T>
T get(HandlerAccessor< T,Handler, ObjectAccessor> da, Iter it) {
   return da.internal_map()[get_object(da.internal_sa(),it)];
   }

template<class Iter,class Handler,class ObjectAccessor,class T>
void set(HandlerAccessor< T,Handler, ObjectAccessor> da,
Iter it, T value) {
   da.internal_map()[ get_object(da.internal_sa(),it)]=value;
   }
```

C++-code 3.4: a handler accessor that hides the internal use of handler objects

Another point of interest was introducing bottom and top variables for data accessors. For example, the Dijkstra shortest path algorithm assigns each node other than the starting node a beginning value of theoretically infinity, but practically the maximum value of the datatype of the distances. The integer version of the algorithm therefore uses `MAX_INT` as maximum value.

Each data accessor class was derived from a base class which defines for each value type appropriate bottom, top and null values.

A data accessor that refers to a boolean type attribute may have `true` and `false` as top and bottom value. A data accessor that refers to an integer type attribute may have `MAX_INT` as top and 0 as bottom value (see figure 3.5).

```
template<class DA> class TopBot_Integer {
public:
   typedef typename DA::value_type value_type;
   value_type value_null, value_max;
private:
   DA& i_da;
   TopBot_Integer(DA& da) :
      i_da(da), value_null(0), value_max(MAX_INT) { }
   DA& internal_da() {
      return i_da; }
};

template<class DA> typename DA::value_type
get(TopBot_Integer< DA> da, Iter it) {
   return get(da.internal_da(),it);
   }
```

C++-code 3.5: base class for defining top, bottom and null values for the integer datatype

In LEDA, there are many different data accessor, all of which are in some way derived from the list of basic data accessors (see table 3.1). For example, `node_map_da`, `edge_array_` are handler accessors, `constant_da` is a constant data accessor, `node_member_da` is a member data accessor and `node_attribute_da` is similar to a member data accessor.

Data Accessor da has a given type `T` and functions `get` and `set`.

da.`get`(it) returns the associated value for the iterator it which is of type `T`

da.`set`(it, val) sets the associated value to val

Object Accessor oa can be used as a parameter for a function `get_object`.

`get_object`(oa, it) returns node or edge of iterator it

Handler Accessor ha has an object accessor oa and a handler object $handler$

ha.`get`(it) returns $handler[get\_object(oa, it)]$ and

ha.`set`(it, val) sets $handler[get\_object(oa, it)]$ to val

Member Accessor ma has an object accessor oa and a member-value-pointer ptr

ma.`get`(it) returns $get\_object(oa, it). * ptr$

ma.`set`(it, val) sets $get\_object(oa, it). * ptr$ to val

Method Accessor mta has an object accessor oa and a member-function-pointer ptr

mta.`get`(it) returns $get\_object(oa, it).(*ptr)()$

Constant Accessor ca has a constant value c

ca.`get`(it) returns c

Calc Accessor cca has two data accessor s and t and a calculator object $calc$ and there is a global function `calculate`

cca.`get`(it) returns the result of `calculate`$(calc, result, s, t)$

Table 3.1: overview of data accessors.

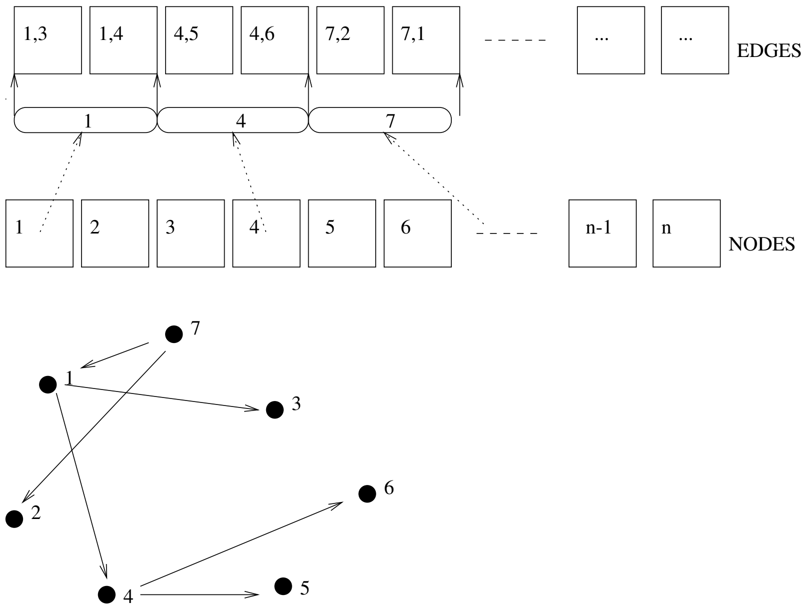
4 Adaptation

It is possible to re–use graph algorithms designed for iterators for other graph implementations as long as the traits interfaces are adapted.

Suppose we know at a certain stage of program execution that a data structure has a certain stable feature, e.g. the graph remains constant until the execution ends. Then we might convert the LEDA graph structure into a faster constant version on which we run the same algorithms. With the generic algorithms presented here and the possibility of adapting traits classes the effort is very little.

4.1 fgraph

Figure 4.1 alternative graph structure



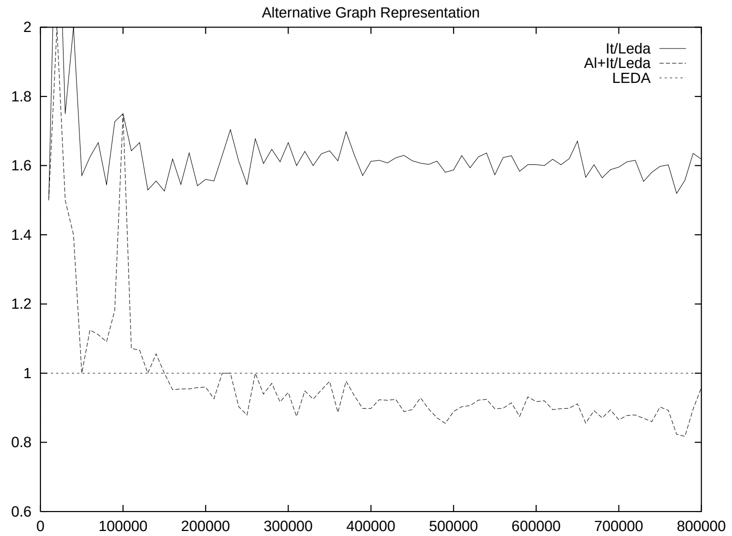
We will discuss an alternative implementation of graphs. `fgraph` (see figure 4.1) stores nodes and edges in a different way than LEDA's `graph` and is optimized in space requirements and access to the list of outgoing edges for each node. Each node knows the interval in the array of edges with incident edges and the edges are grouped according to the source node.

See figure 4.1 for a traits class for adjacency iterators for outgoing edges and the special graph implementation `fgraph` (it is incomplete in the sense that functions with void functionality are skipped).

After converting a LEDA graph to this fast graph structure, shortest path runs much faster (see figure 4.2). Figure 4.2 shows a runtime comparison of different versions of Dijkstra (LEDA and iterator version and the alternative version with fast graphs) on randomly generated graphs with 1.000 nodes and edges from 10.000 to 800.000). The graphs are generated by choosing arbitrarily three nodes and connecting them with edges until the number of edges is reached; no parallel edges are generated.

In practice, it might happen that a very large and inefficient graph structure is (implicitly) given - then conversion can be too time consuming (for instance, if the algorithm runs faster than the conversion would be). If possible, adapting the traits classes is a better solution and may result in better performance compared to conversion.

Figure 4.2 Runtime comparison: different lines describe different implementations of the algorithm. "It/Leda" means LEDA with graph iterators which are described in the LEDA manual. "Alt+It/Leda" means LEDA with adapted iterators that run on the alternative representation of the graph. Both lines are scaled by the runtime of a standard "Leda" version of the algorithm.



4.2 complete graphs

Sometimes implicit definition can be more efficient than explicit storage of graphs. This can be done by adapting the traits classes, as before.

A complete graph needs a lot of memory (space $\mathcal{O}(n+m) = \mathcal{O}(n^2)$, n number of nodes, m number of edges). It is possible to reduce the space bounds for complete graphs to linear size, because we only need the nodes as materialized entities.

See figure 4.4 for an appropriate traits-class[1].

```
struct fNodeEdgeTrait {
  typedef fedge* T_Edge;
  typedef fnode* T_Node;
  typedef fgraph T_Graph;
  static T_Node node_null() {return nil }
  static T_Edge edge_null() {return nil }
  static bool is_node_null(T_Node v) {return v=nil }
  static bool is_edge_null(T_Edge e) {return e=nil }
  static void forward(T_Graph& G, T_Node&, T_Edge& e) {
    e=G.succ_adj(e); }
  static T_Edge first_incident_edge(T_Graph& G, T_Node v) {
    return G.first_adj_edge(v); }
  static void assign(T_Node& n1, T_Node n2) { n1=n2; }
  static void assign(T_Edge& e1, T_Edge e2) { e1=e2; }
  static bool is_equal(T_Edge e1, T_Edge e2) {
    return e1 ≡ e2; }
  static bool is_equal(T_Node n1, T_Node n2) {
    return n1 ≡ n2; }
  static void curr_adj(T_Node& n, T_Edge& e, T_Graph& G) {
    if (is_edge_null(e))return ;
    n=e → target();
    e=G.first_adj_edge(n); }
  static T_Node first_node(T_Graph& G) {
    return G.first_node(); }
};
```

`C++`-code 4.1: traits class for adjacency iterators for outgoing edges and a special graph implementation `fgraph`

[1]This traits class is used in an example program that computes shortest paths for complete LEDA graphs, where we used the implicit implementation.

If we cannot assume anything about the constellation of data, we have to store it explicitly into a container (a list, an array or indexed table in a database; space $\mathcal{O}(n)$ for nodes or $\mathcal{O}(m) = \mathcal{O}(n^2)$ for edges).

Sometimes, values can be computed at run–time from other values. If, for example, edge lengths can be computed from node locations, we can improve the overall space bounds to $\mathcal{O}(n)$, since edge lengths will take constant size.

There is a special data accessor (see figure 3.1) that is able to calculate values out of some values associated to the source and target nodes of an edge: `CalcAccessor`. Unfortunately, usage is not easy, but not as difficult as rewriting a specialized version of all possible algorithms for on-line calculations of attributes.

To compute the length of an edge the program calculates the absolute value of the difference of the indices associated to the nodes (the formula to compute the length is only exemplary).

concrete implementation: we will need the following (ai is an adjacency iterator):

1. two appropriate object accessors
2. ai refers to node v and edge e:

   we need two data accessors that return the location of the source and the target node of e
3. a function that calculates the formula
4. the edge length calculation data accessor

(1.) Since we have designed new traits-classes, we also must define appropriate object accessors. See figure 4.2 for the implementation of object accessors, which follows what we said about object accessors in chapter 3.

(2.) Then we need two data accessors that return the location of the source and the target node: `SourceAcc` and `TargetAcc`. Let ai be an adjacency iterator with fixed node v and incident edge e. `SourceAcc` accesses v and returns the index of that node. `TargetAcc` accesses e and returns the index of the target node. This will be used for the exemplary formula. See figure 4.3 for the implementation.

(3.) The correct calculation function must be selected. To ensure this at compile-time, we use the technique from above: create a type with void functionality and use it for overloading a function. An object of that type will be stored in the data accessor class and assures that the compiler will use the following function:

```
struct MathNodeAcc { };
struct MathEdgeAcc { };

template<class Iter>
int get_object(MathNodeAcc, Iter it) {
   return it.get_node(); }
template<class Iter>
int get_object(MathEdgeAcc, Iter it) {
   return it.get_edge(); }

MathNodeAcc mathgraph_node;
MathEdgeAcc mathgraph_edge;
```

C++-code 4.2: object accessors for complete graphs

```
struct transfer_s {
   int operator [ ] ( int k) const {
      return k; }
};
struct transfer_t {
   int operator [ ] (pair<int , int > P) const {
      return P.second; }
};
transfer_s ts;
transfer_t tt;
typedef HandlerAccessor<int , transfer_s, MathNodeAcc>
      SourceAcc;
typedef HandlerAccessor<int , transfer_t, MathEdgeAcc>
      TargetAcc;
```

C++-code 4.3: implementation of two data accessors for complete graphs

```
class MathNodeEdgeTrait {
   typedef pair<int ,int > T_Edge;
   typedef int T_Node;
   typedef int T_Graph;
public:
   static T_Node node_null() {return -1; }
   static bool is_node_null const T_Node& v) {return v ≡ -1; }
   static T_Edge edge_null() {
      return make_pair(-1,-1); }
   static bool is_edge_null const T_Edge& v) {
      return v.second ≡ -1; }
   static T_Edge new_edge(T_Graph& G, T_Node v1, T_Node v2) {
      return make_pair(-1,-1); }
   static void del_edge(T_Graph& G, T_Edge e) { }
   static void forward(T_Graph& G, T_Node&, T_Edge& e) {
      if (e.first ≡ -1) return ;
      if (e.second< 0 || G-1 ≡ e.second) e.second=-1;
      else ++e.second; }
   static void backward(T_Graph& G, T_Node&, T_Edge& e) {
      if (e.first ≡ -1)return ;
      if (e.second< 0) e.second=-1;
      else −−e.second; }
   static T_Edge first_incident_edge(T_Graph& G, T_Node v) {
      return make_pair(v,0); }
   static void assign(T_Node& n1, const T_Node& n2) { n1=n2; }
   static void assign(T_Edge& e1, const T_Edge& e2) { e1=e2; }
   static bool is_equal const T_Edge& e1, const T_Edge& e2) {
      return ( e1 ≡ e2 || (e1.second ≡ e2.second && e1.second ≡ -1)); }
   static void curr_adj(T_Node& n, T_Edge& e, T_Graph& G) {
      if (is_edge_null(e))return ;
      n=e.second;
      e=make_pair(n,0); }
   static T_Node first_node(T_Graph& G) {return 0; }
   static T_Node last_node(T_Graph& G) {return G-1; }
   static T_Edge first_edge(T_Graph& G) {
      return make_pair(0,0); }
   static T_Edge last_edge(T_Graph& G) {
      return make_pair(G-1,G-1); }
   static T_Edge last_incident_edge(T_Graph& G, T_Node v) {
      return make_pair(v,G-1); }
   static bool is_equal(T_Node n1, T_Node n2) {
      return n1 ≡ n2; }
};
```

C++-code 4.4: traits class for complete graphs

```
struct LengthCalc { } lengthcalc;
void calculate(LengthCalc, int & result,int s,int t) {
   result=abs(s-t); }
```

(4.) The last step is to create a data accessor which computes the length of an edge at run–time.

```
SourceAcc SA(ts,mathgraph_node);
TargetAcc TA(tt,mathgraph_edge);
typedef CalcAccessor<int , SourceAcc, TargetAcc,LengthCalc> Length;
Length length_da(SA,TA,lengthcalc);
```

With this complex data accessor we reduced the space bounds from $\mathcal{O}(n+m)$ to $\mathcal{O}(n)$. In complete graphs this means $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The other attributes needed for Dijkstra, distance and maybe the predecessor entry, are stored in $\mathcal{O}(n)$ space.

Summarizing, Dijkstra can be implemented for complete graphs with time bounds $\mathcal{O}(m \cdot T(n,m) + n \cdot T'(n))$ where $T(n,m)$ is the complexity of a single length calculation and $T'(n)$ the (possibly amortized) complexity of a single queue update (for the priority queues in LEDA, $T'(n) = \mathcal{O}(\log n)$). The space requirements are $\mathcal{O}(n)$, if we can represent the length of edges as describes above.

5 Algorithm Classes

Algorithm classes are implemented in C++ in this way: algorithms as objects and the ability to "advance" an algorithm will be implemented by writing a class, which knows a certain state that can be advanced by applying a method with name "next()" to an object of this class.

```
class Algorithm {
   int state, endstate;
public:
   Algorithm(int max) :
   endstate(max), state(0) { }
   void next() {
      state++;
      }
   bool finished() {
      return state ≥ endstate;
      }
};
```

With this class `Algorithm` we can easily instantiate an algorithm object:

```
Algorithm alg(5);
while(!alg.finished())
   alg.next();
```

This small piece of code creates an algorithm object and invokes "next()" until it has reached an end state.

An advantage of this design is that we can write basic algorithms, which can be used in a standardized way and if needed, inspection of internal states and variables can be provided without writing complex code. Additionally, it makes it possible to write persistent algorithms, if the member variables are persistent.

Actually, those algorithms are quite more flexible than ordinary written algorithm functions:

```
template<class Alg>
class OutputAlg {
    Alg alg;
public:
    OutputAlg(int m) : alg(m) {
        cout << "max state: ";
        cout << m << endl;
        }
    void next() {
        cout << "old state: ";
        cout << alg.state;
        alg.next();
        cout << " new state: ";
        cout << alg.state << endl;
        }
    bool finished() {
        return alg.finished(); }
};
```

This wrapper algorithm can be used like this:

```
OutputAlg< Algorithm> alg(5);
while(!alg.finished())
    alg.next();
```

In addition to the algorithm mentioned earlier this wrapper writes the internal states to the standard output.

This is as efficient as rewriting the "`Algorithm`"-class with an output mechanism, but provides more flexibility.

6 Integration into LEDA

As the use of generic code which is based on templatized code is a very difficult task, all classes were specialized for the LEDA graph class. They were integrated to the LEDA software tree since version 3.6.

To see how easy it is to use iterators in own algorithms, one may look at the LEDA manual pages (graphiterator section).

It is now easy to use iterators. There are wrapper classes for all possibilities of data storage in LEDA that simplify the access in form of data accessors. For example, for a given `node_array<bool>` we can create an instance of `node_array_da<bool>` and initialize it with the given node array. The result will be a data accessor.

The piece of code may look like this:

```
graph G;
node_array<bool > _mark(G,false);
OutAdjIt ai(G);
node_array_da<bool > mark(_mark);
```

With these data structures we can now write a simple breadth first search algorithm. At first we need a simple queue that can be templatized with the element type `OutAdjIt`. Iterators can be updated using method `update` and `curr_adj` computes the successor iterator in direction of the current incident edge, i.e. the new fixed node would be the current adjacent node. See figure 6.1 for the algorithm implementation.

A nice observation is that algorithm classes are quite similar to iterator objects although they have not the same interface. It is easy (as shown in the package) to write a wrapper class that modifies the interface. For example, you can use an iterator that traverses the strongly connected components and can give you the component number of the current component.

```
queue< OutAdjIt> Q;
OutAdjIt temp(ai);
Q.clear();
Q.append(ai);
set(mark,ai,true);
while(!Q.empty()) {
  ai.update(Q.pop());
  while (ai.valid()) {
    temp=ai.curr_adj();
    if (get(mark,temp) ≠ true) {
      set(mark,temp,true);
      Q.append(temp);
      }
    ++ai;
    }
  }
```

C++-code 6.1: breadth first search algorithm

7 Final Remarks

Time and space costs may be described in $\mathcal{O}$-notation. If we compare two alternative approaches with the same complexity, we may be interested in constant factors. The overhead induced by using iterators, data accessors, and algorithm classes is a constant factor and is — if implemented in C++ using traits classes — quite small. Additionally, algorithms that are time-critical (for example in real-time applications) maybe inspected even before the algorithm has been terminated and this might reduce the run–time.

For example, pipelining can be used without changing the algorithm or introducing overhead. Algorithms are controllable like processes without expensive and difficult interprocess solutions. Thus, resources can be managed optimally. If an underlying data structure is exchanged by a more efficient one (in a specific situation), then time and space bounds may be improved.

In fact, designing algorithms for iterators and data accessors is more complicated than for more convenient data structures, e.g. the LEDA graph. Fortunately, this increased overhead will be rewarded in future — if code has to be re–used in some other situation, because the existing algorithm code can be re–used with only adapting the iterator and accessor classes.

8 Appendix

The appendix contains some explanations of important `C++`-concepts. Section 8.1 explains what `C++` means by templates. STL's efficiency is due to the heavy use of the traits technique which is shortly reviewed in section 8.2.

8.1 Templates

A **template** defines a family of types or functions. A template declaration may look like this, where `TList` is a list of template parameters, separated by a comma and each of which is e.g. a class type, introduced with the class-keyword. `declaration` defines a function, a class, or a static data member.

```
template< TList > declaration
```

This is an example of a template class `Test`, which can be instantiated with a given type.

```
template< class T >
class Test {
   T _t;
   Test(T t) : _t(t) { }
};
```

For example, we may instantiate the template class with `int` as the template parameter, and we call the constructor of that class with a given integer value 5 to construct an object k of type `Test<int>`:

```
Test<int > k(5);
```

A second possibility is to use template functions, i.e. `declaration` will be a function declaration. A declaration is correct if the function declaration is correct and the list of arguments in the parameter list of the function determines in a unique way the list of template arguments. Therefore, we do not have to explicitly provide a template parameter when calling the function `print<int>(int)`. So the compiler knows implicitly the list of template arguments and is able to instantiate the function.

```
template< class T >
void print(T t) {
   /* do something */
   }
```

It can be used like this:

```
int k(5);
double l(6.4);
print(k);
print(l);
```

Member templates are simply templates that are members. For example, the following template class `Pair` contains a template as a class member:

```
template<class A,class B>
class Pair {
public:
   A _a;
   B _b;
   Pair(A a, B b) : _a(a) , _b(b) {}
template<class T1,class T2>
   Pair(Pair< T1,T2> p) :
   _a(p._a), _b(p._b) { }
};
```

Now we can use this class to construct a pair from a second pair of different types, i.e. we construct an object of class `Pair<long,double>` from an object of class `Pair<int,float>` (conversion from `C` to `A` and `D` to `B` is assumed to be supported).

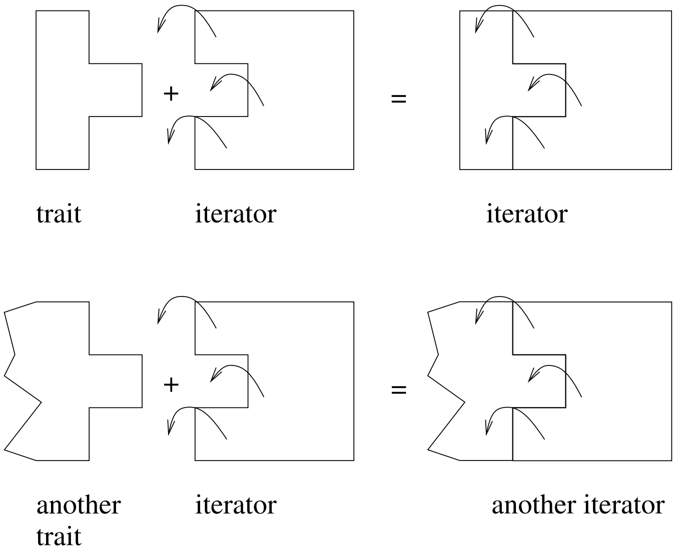
centerline

```
Pair<int ,float > a(10,10.5);
Pair<long,double > b(a);
```

Without member templates, we could only declare a constructor with fixed argument types such as `Pair(Pair<int,float> p)` or we could use the template arguments of class `Pair` itself such as `Pair(Pair<A,B> p)`.

Unfortunately, most of the common compilers (except egcs) are not able to process member templates, but since this feature is covered by the ANSI `C++` standard, it will be supported in the future.

Figure 8.1 Traits-Technique



8.2 Traits Classes

With **traits classes** we can make a class polymorphic without introducing large performance loss. Dynamic polymorphism using inheritance normally uses runtime method binding, which is not necessary here.

A **traits class** is a class that is suitable as an "input class" for another class and contains type definitions and static methods. The parameterized class defines "by usage" a convention for the names of these types and the signatures of the static functions. When designing a traits class, one has to adhere to this convention.

The traits-technique (see figure 8.1) makes it possible to divide the implementation of an iterator into two parts: the first is specific for the underlying graph representation and the other is specific for the iterator. For example, suppose "iterator" is a node iterator, "trait" is a traits class for LEDA types (`node`, `graph`) and "another trait" is a traits class for XYZ types (from a different library, say XYZ). Then we have a node iterator for LEDA types and a node iterator for XYZ types without rewriting the code for iterators.

Iterators are implemented using traits classes as template parameters. Iterators use only the types and static functions defined in the traits class while the traits class invokes the correct method of the underlying graph structure. If such a method is not defined by the graph structure, the traits class has to provide the desired functionality, itself. We refer to a class, which uses a traits class as a *master class*.

The following example will explain the structure of traits-classes:

First a master class is defined, which demonstrates the usage of a traits class. This class requires a static method "print":

```
template<class traits>
class TestTraits {
public:
   typedef typename traits::value_type value_type;
private:
   value_type _f;
public:
   TestTraits(value_type f) : _f(f) {
      traits::print(_f);
      }
};
```

And now follows an example of a traits-class:

```
class traits1 {
public:
   typedef int value_type;
   static void print(value_type f) {
      cout << f; }
};
```

By instantiating master class `TestTraits` with class `traits1`, we can give class `TestTraits` functionality:

```
TestTraits< traits1> K(5);
```

This prints "5" to the standard output[1].

To change class `TestTraits` functionality we can write a different class `traits2`:

```
class traits2 {
public:
   typedef int value_type;
   static void print(value_type f) {
      while(f > 0) cout << f--;
      }
};
```

[1]A short explanation: `TestTraits<traits1>` is a type produced by instantiating class `TestTraits<>` with `traits1`. `K(5)` will be an object of this type, initialized with 5, i.e. the constructor of that class will be invoked with 5.

This produces “54321” and is an example for exchanging the functionality in an elegant way without violating efficiency because the function `print` may be bound in both cases statically by the compiler.

Traits classes must always have a defined structure and functionality to avoid compile and runtime errors. To do this we discuss two ideas:

- *Syntactic Requirements:* today, there is no language support in `C++` for template requirements. Therefore, we must specify all requirements in the documentation of the code.

  E.g., let `A` and `B` be two master classes and `C` be traits classes fitting into the requirements of class `A`. Suppose that `B` has the requirements of `A` and an additional one that `C` does not meet. Then, `C` will only fit for `A` but not for `B`. The compiler will not know if `C` fits for `B` before using `C` as a template for `B`. The compiler reports an error-message if `B` uses something not present in `C`. It will not report that requirements are violated and might make debugging more difficult. It gets problematic, if the current implementation of `B` only use the requirements of `A`, because there is no such error-message although the requirements are not met by `C`. With a requirement documentation, we are able to check the list of requirements at programming time to avoid these situations.

- *Semantic Requirements:* the master class is correct if at least the traits classes are implemented correctly according to the requirement specification. Therefore, the documentation must describe the behavior, and the pre- and postconditions of the static methods. Additionally, the requirements for the type definitions must be listed.

Bibliography

[BOSW97] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler: *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language*.
Proceedings of the 12th ACM Symposium on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)

[GHJV95] Gamma,E. and Helm, R. and Johnson, R. and Vlissides, J.: Design patterns. *Addison-Wesley*, 1995.

[KW97] Kühl, Dietmar and Weihe, Karsten: "Data Access templates"
C++ Report, 9/7, 15 and 18-21, 1997.

[LEDA] LEDA homepage:
`http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MN97] Mehlhorn, Kurt and Näher, Stefan: "The LEDA Platform of Combinatorial and Geometric Computing"
to appear with *Cambridge University Press*, 1999.

[MS96] Musser, David R. and Saini, Atul: *STL tutorial and reference guide : C++ [plus plus] programming with the standard template library. Addison Wesley*, 1996.

[N99] Nissen, M.:
Design Pattern Data Accessor
Proceedings of the 4th European Conference on Pattern Languages of Programming (EuroPLoP '99)

[NW96] Nissen, M. and Weihe, Karsten: "Attribute Classes in Java and Language Extensions"
Konstanzer Schriften Schriften in Mathematik und Informatik, Universität Konstanz, 1996
Web: http://www.informatik.uni-konstanz.de/˜nissen/JavaPaper/

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph

MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values