Data Types and Data Structures for Static Graphs

(LEDA Graph Types and Algorithms)

Stefan Näher
Universität Trier

Projects and Systems

- LEDA: a Platform for Combinatorial and Geometric Computing.
  `www.mpi-sb.mpg.de/LEDA`
- CGAL: a Library of Algorithms in Computational Geometry.
  `www.cgal.org`
- AGD: a Library for Automatic Graph Drawing.
  `www.mpi-sb.mpg.de/AGD`

Overview

Lecture 1: LEDA Graph Types and Algorithms

Lecture 2: Static Graphs

Lecture 3: Further Topics (Visualization)

Programming Exercises

Goal

Efficient Algorithm $\longrightarrow$ Efficient Program

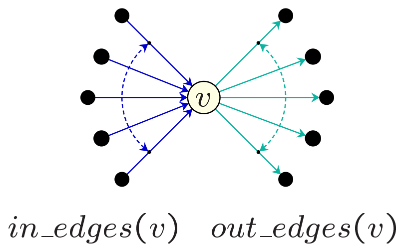
– parameterized data types (C++ classes and templates)
– precise specifications
– ease of use
– state of the art data structures

– program checkers
– library of algorithms
– extensibility
– visualisation and animation

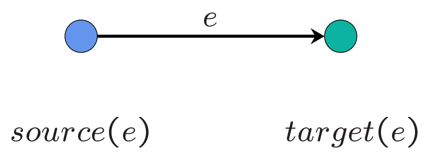
Graphs in LEDA

(data type `graph`)

node



edge



Operations

Access	int	G.outdeg(node v);
	node	G.target(edge e);
	int	G.number_of_nodes();
	list<edge>	G.out_edges(node v);
	⋮	
Update	node	G.new_node();
	void	G.del_node(node v);
	edge	G.new_edge(node v, node w);
	void	G.del_edge(edge e);
	⋮	
Iteration	forall_nodes(v, G)	$\{\ldots\}$
	forall_edges(e, G)	$\{\ldots\}$
	forall_out_edges(e, v)	$\{\ldots\}$
	forall_in_edges(e, v)	$\{\ldots\}$

A Simple Graph Problem

Acyclic Test

Input: a directed graph $G(V,E)$
Ouput: **true** if G is acyclic and **false** otherwise

Observation:
Every acyclic graph contains a node with in-degree zero.

Algorithm:
Remove iteratively nodes with in-degree zero.

Efficiency:
– maintain a candidate list
– delete edges

```
ZERO ← { v ∈ V | indeg(v) = 0 }
while ZERO ≠ ∅
do choose arbitrary v from ZERO;
   delete v from ZERO;
   forall w with (v,w) ∈ E
   do delete (v,w) from G;
      if indeg(w) == 0
      then add w to ZERO; fi
   od
od
return |E| == 0;
```

```
#include <LEDA/graph.h>
using namespace leda;

bool ACYCLIC(const graph& g0)
{ graph G = g0;    // we work on a local copy
  list<node> ZERO;
  node u;
  forall_nodes(u,G)
    if (G.indeg(u) == 0) ZERO.append(u);
  while (!ZERO.empty())
  { node v = ZERO.pop();
    forall_out_edges(e,v)
    { node w = G.target(e);
      G.del_edge(e);
      if (G.indeg(w) == 0) ZERO.append(w);
     }
   }
  return G.number_of_edges() == 0;
}
```

A Test Program with GraphWin

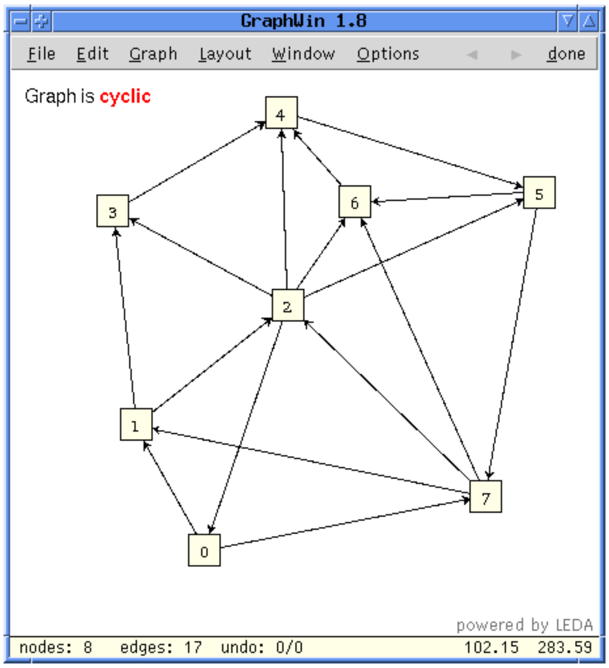
/LEDA/ADFOCS/exercise-1/acyclic.c

GraphWin: Editor and Visualization Tool for LEDA Graphs

Edit and Run

```
   GraphWin gw;
   gw.display();
   graph& G = gw.get_graph();

   while (gw.edit()) {
     "Run Algorithm"
     "Display Result"
   }
```

```
int main() {
   GraphWin gw;
   gw.display();
   graph& G = gw.get_graph();

   while (gw.edit()) {
     if (ACYCLIC(G))
        gw.message("Graph is acyclic.");
     else
        gw.message("Graph is cyclic.");
   }

  return 0;
}
```



Exercise 1

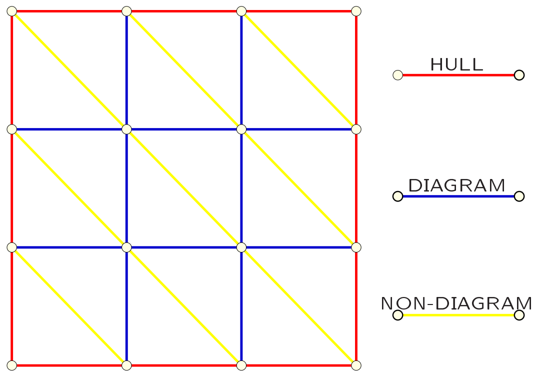
- learn how to use GraphWin
- do not change the graph
- find a circle (if cyclic)
- compute a topological ordering
- make graph acyclic by deleting edges

Parameterized Graphs: GRAPH<vtype,etype>

GRAPH<point,color> DT; // Delaunay Triangulation

$DT[v]$: position of v

$DT[e]$: kind of e

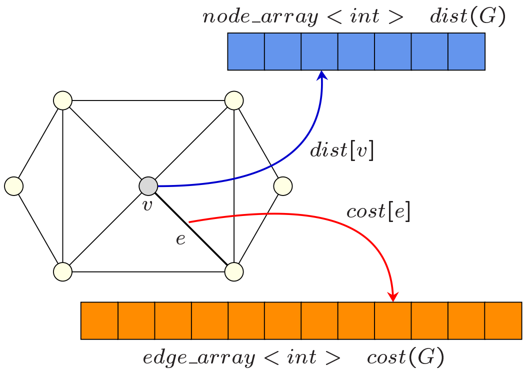


GRAPH<circle,point> VD; // Voronoi Diagram

$DT[v]$: voronoi circle of v

$DT[e]$: voronoi site of e

Node/Edge Arrays



Algorithms

```
COMPONENTS(const graph& G, node_array<int>& compnum);

SHORTEST_PATHS(const graph& G, node s, const edge_array<int>& cost,
                                           node_array<int>& dist)
⋮
```

Dijkstra's Algorithm

Input: graph $G(V, E)$, $cost : E \longrightarrow \mathbb{R}$, $s \in V$

Output: $dist : V \longrightarrow \mathbb{R}$

for all $v \in V$ **do** $dist(v) \leftarrow \infty$ **od**;
$dist(s) \leftarrow 0$;
$Q \leftarrow \{s\}$;
while $Q \neq \emptyset$
do delete u from Q with $dist(u)$ minimal
    for all $(u, v) \in E$
    do if $dist(u) + cost(u, v) < dist(v)$
        then $dist(v) = dist(u) + cost(u, v)$;
            $Q \leftarrow Q \cup \{v\}$;
        fi;
    od;
od;

Dijkstra Program

```
void DIJKSTRA(graph& G, node s, const edge_array<int>& cost
                                node_array<int>& dist
{ node_pq<int> Q(G);
  node v;
  forall_nodes(v, G) dist[v] = MAXINT;
  dist[s] = 0;
  Q.insert(s, dist[s]);
  while ( ! Q.empty() )
  { node u = Q.del_min();
    forall_out_edges(e, u)
    { v = G.target(e);
      int c = dist[u] + cost[e];
      if (c < dist[v])
      { if (dist[v] == MAXINT)
            Q.insert(v, c);
        else
            Q.decrease_p(v, c);
        dist[v] = c;
       }
     }
   }
 }
```

Kruskal's Algorithm

Input: graph $G(V, E)$, $cost : E \longrightarrow \mathbb{R}$

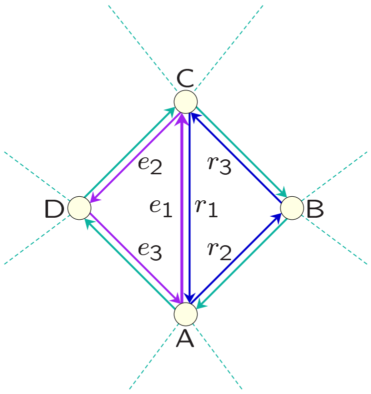
Output: MST of G

1. $MST \leftarrow \emptyset$
2. sort E according to $cost$
3. create a node partition $P = \{\ \{v\} \mid v \in V\ \}$
4. **for all** $(v, w) \in E$
5. **do if** v and w are not in the same block of P
6.   **then** unite blocks of v and w in P
7. $MST \leftarrow MST \cup \{(v, w)\}$
8.   **fi**
9. **od**

Kruskal Program

```
list<edge> KRUSKAL(graph& G, const edge_array<int>& cost)
{  list<edge> MST;
   G.sort_edges(cost);
   edge e;
   node_partition P(G);
   forall_edges(e, G)
   { node v = G.source(e);
     node w = G.target(e);
     if ( ! P.same_block(v, w) )
     { P.union_blocks(v, w);
       MST.append(e);
      }
    }
   return MST;
 }
```

Planar Maps (plane graphs)

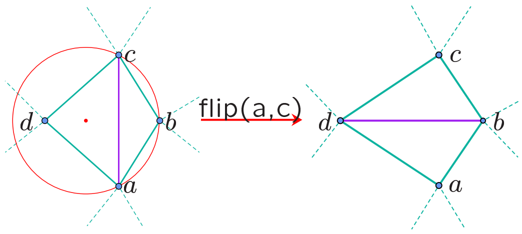



```
GRAPH<point,color> G;

edge e1 = ...
edge e2 = G.face_cycle_succ(e1);
edge e3 = G.face_cycle_succ(e2);

edge r1 = G.reversal(e1);
edge r2 = G.face_cycle_succ(r1);
edge r3 = G.face_cycle_succ(r2);

point A = G[source(e2)];
point B = G[target(e2)];
point C = G[source(r2)];
point D = G[target(r2)];
```

Delaunay Flipping

Triangulation $\longrightarrow$ Delaunay Triangulation

Local Transformation



Precondition:

d lies in the interior of circle (a, b, c)
$side\_of\_circle(a, b, c, d) > 0$

```
void DELAUNAY_FLIPPING(GRAPH<POINT,int>& G) {
  list<edge> S = G.all_edges();

  while ( !S.empty() )
  { edge e = S.pop();
    edge r = G.rev_edge(e);

    edge e1 = G.face_cycle_succ(r);  // edges of quadriliteral
    edge e2 = G.face_cycle_succ(e1); // with diagonal e
    edge e3 = G.face_cycle_succ(e);
    edge e4 = G.face_cycle_succ(e3);

    POINT a = G[G.source(e1)];       // corners of quadriliteral
    POINT b = G[G.target(e1)];
    POINT c = G[G.source(e3)];
    POINT d = G[G.target(e3)];

    if (side_of_circle(a,b,c,d) > 0)
    { S.push(e1); S.push(e2); S.push(e3); S.push(e4);
      G.move_edge(e,e2,source(e4));  // flip diagonal
      G.move_edge(r,e4,source(e2));
     }
   }
}
```

graph data structure

Fully dynamic (doubly linked) lists of objects:
V, E
out_edges(v)
in_edges(v)

nodes	edges
1. V_succ	1. E_succ
2. V_pred	2. E_pred
3. out_first	3. out_succ
4. out_last	4. out_pred
5. in_first	5. in_succ
6. in_last	6. in_pred
7. index	7. source
8. indegree	8. target
9. outdegree	9. index
10. owner	10. reversal
11. user-inf	11. user-inf

Total Space: $11(n+m)$ words $= 44(n+m)$ bytes.

A Template for Static Graphs

```
st_graph<class graph_category, int node_slots, int edge_slots>
```

`graph_category`
class describing the type of graph to be used
(basic_graph, in_edges_graph, fixed_degree_graph, . . .)

`node_slots`
number of node slots for user data (node_arrays)

`edge_slots`
number of edge slots for user data (edge_arrays)

Algorithm Templates

```
template <class graph_t>
void DIJKSTRA_T(const graph_t& G,
                graph_t::node s,
                const edge_array<int,graph_t>& cost,
                node_array<int,graph_t>& dist)
{
  node_pq<int,graph_t>  PQ(G);
  dist.init(G,MAXINT)

  dist[s] = 0;
  PQ.insert(s,0);

  while (!PQ.empty())
  { node u = PQ.del_min();
    int du = dist[u];
    edge e;
    forall_out_edges(e,u)
    { node v = G.target(u,e);
      int c = du + cost[e];
      if (c < dist[v])
      { if (dist[v] == MAXINT)
           PQ.insert(v,c);
        else
           PQ.decrease_p(v,c);
        dist[v] = c;
       }
    }
  }
}
```

Application Programs

```
#include <LEDA/st_graph.h>
#include <LEDA/dimacs.h>

int main()
{
  typedef st_graph<flow_graph_category,3,2> flow_graph;
  typedef flow_graph::node node;
  typedef flow_graph::edge edge;

  flow_graph G;
  node s,t;
  edge_array<int,flow_graph>  cap(G);
  edge_array<int,flow_graph>  flow(G);

  read_dimacs_mf(cin,G,cap,s,t);

  MAXFLOW_T(G,s,t,cap,flow);

  return 0;
}
```