COIN-OR::LEMON

The Graph Library

Alpár Jüttner
alpar@cs.elte.hu

ELTE-TTK, Dept. of Operations Research

http://lemon.cs.elte.hu

http://lemon.cs.elte.hu/trac/lemon/wiki/SummerSchool2016

Overview

LEMON Graph Library

http://lemon.cs.elte.hu

- Open source C++ library for graph and network modeling and optimization.
- Applicable to both research projects *and* industrial applications.
  - Permissive (BSD style) open source licens, allowing commercial use.
  - Most efficient implementations (fastest on the market)
- Reliable, comprehensive implementations
  - $\approx$ 65000 lines of heavily optimized code
  - Support of different platform and compilers
    - Windows, Unix, Linux
    - GCC, Intel C++, Clang, Visual Studio, MinGW
- Contributors wanted!!!

Download and Install — Prerequisites

Dependencies (Linux)

- C++ compiler (GCC, Clang or Intel C++)
- IDE (`emacs`, `CodeBlocks`, `Sublime` etc.)
- CMAKE (`http://cmake.org`)
- Mercurial (`https://www.mercurial-scm.org/`)
  - for obtaining and contributing to the developer version
- Doxygen (`http://www.doxygen.org`)
  - for making documentations
- GLPK, CLP/CBC, CPLEX
  - for solving LP/IP problems

Dependencies (Windows)

- IDE (`Visual Studio`, `CodeBlocks`)
- CMAKE (`http://cmake.org`)
- Mercurial — TortoiseHg (`http://tortoisehg.bitbucket.org/`)

Download and Install (developer version)

Basic steps

1. `hg clone http://lemon.cs.elte.hu/hg/lemon-main mylemon`
2. `cd mylemon`
3. `mkdir build; cd build`
4. `cmake` [*options*] `..`
5. `make`
6. [`make install`]

CMAKE options

- `--help`
- `-D CMAKE_BUILD_TYPE=`build-type
  - `Release`, `Debug`, `Maintainer`
- `-G` generator
  - `"CodeBlocks - Unix Makefiles"`,
  - `"Visual Studio 12 2013"`,

CMAKE GUI

- `cmake-gui ..`

First program

hello_lemon.cc

```
#include <iostream>
#include <lemon/list_graph.h>

using namespace lemon;
using namespace std;

int main()
{
  ListDigraph g;

  ListDigraph::Node u = g.addNode();
  ListDigraph::Node v = g.addNode();
  ListDigraph::Arc  a = g.addArc(u, v);

  cout << "Hello World! This is the LEMON library here." << endl;
  cout << "We have a directed graph with " << countNodes(g) << " nodes "
  << "and " << countArcs(g) << " arc." << endl;

  return 0;
}
```

First program

hello_lemon.cc

```
#include <iostream>
#include <lemon/list_graph.h>

using namespace lemon;
using namespace std;

int main()
{
  ListDigraph g;

  auto u = g.addNode();
  auto v = g.addNode();
  g.addArc(u, v);

  cout << "Hello World! This is the LEMON library here." << endl;
  cout << "We have a directed graph with " << countNodes(g) << " nodes "
  << "and " << countArcs(g) << " arc." << endl;

  return 0;
}
```

Basics

Header files

`#include<lemon/list_graph.h>`	The graph data structure
`#include<lemon/bfs.h>`	BFS algorithm
`#include<lemon/dijkstra.h>`	Dijkstra algorithm

Namespace

```
using namespace lemon;
```

Graph operations

Adding vertices and arcs

```
ListDigraph::Node x = g.addNode();
ListDigraph::Node y = g.addNode();
ListDigraph::Node z = g.addNode();

g.addArc(x,y); g.addArc(y,z); g.addArc(z,x);
```

Reading from file

```
#include <lemon/list_graph.h>
#include <lemon/lgf_reader.h>

using namespace lemon;

int main()
{
  ListDigraph g;
  digraphReader(g, "digraph.lgf").run();
}
```

LEMON Graph Format

digraph.lgf

```
@nodes
label
0
1
...
41

@arcs
@
0    1
0    2
2    12
...
36   41
```

LEMON Graph Format

digraph.lgf

```
@nodes
label
0
1
...
41

@arcs
          label
0    1    0
0    2    1
2    12   2
...
36   41   123
```

LEMON Graph Format

digraph.lgf

```
@nodes
label size
0     12
1     3
...
41    12

@arcs
          label length
0    1    0     16
0    2    1     12
2    12   2     20
...
36   41   123   21
```

ListDigraph

Adding and removing vertices and arcs

```
ListDigraph g;
Node n; n = g.addNode();
Arc e; e = g.addArc(n1,n2);
g.erase(n); g.erase(e);
```

Parallel and loop arcs

```
ListDigraph::Arc parallel = g.addArc(x,y);
ListDigraph::Arc loop = g.addArc(x,x);
```

Head and tail of an arc

```
if (g.source(loop) == g.target(loop))
  std::cout << "This is a loop arc" << std::endl;
```

Node and arc ID

- `g.id(n)` és `g.id(a)`
- Unique and permanent
- It is a “not too big” nonnegative integer
  - They doesn’t always form and interval

Iterators (Old style)

- Used for enumerating/listing nodes and arcs
- Differ from the STL iterators! (Or not)
- On construction they point to the forst element
- The prefix ++ goes through the elements
- When we passed the last one it equals to `INVALID`.

Example I: Count the number of nodes

```
int cnt = 0;
for (ListDigraph::NodeIt n(g); n != INVALID; ++n)
  cnt++;
std::cout << "Number of nodes: " << cnt << std::endl;
```

Example II: Create a full graph

```
for (ListDigraph::NodeIt u(g); u != INVALID; ++u)
  for (ListDigraph::NodeIt v(g); v != INVALID; ++v)
    if (u != v) g.addArc(u, v);
```

Iterators II. (Old style)

`ArcIt`: List the arcs of the graph

```
int cnt = 0;
for (ListDigraph::ArcIt a(g); a != INVALID; ++a)
  cnt++;
std::cout << "Number of arcs: " << cnt << std::endl;
```

`OutArcIt`: List the arcs leaving a node

```
int cnt = 0;
for (ListDigraph::OutArcIt a(g, n); a != INVALID; ++a)
  cnt++;
std::cout << "Num of arcs leaving the node 'n': "
          << cnt << std::endl;
```

`InArcIt`: List the arcs entering a node

```
int cnt = 0;
for (ListDigraph::NodeIt n(g); n != INVALID; ++n)
  for (ListDigraph::InArcIt a(g, n); a != INVALID; ++a)
    cnt++;
std::cout << "Number of arcs: " << cnt << std::endl;
```

Iterators (miscellaneous)

The order of enumeration

- The order of the enumeration is arbitrary
- Does not (necessarily) same as they were added
- What guaranteed: The order of subsequent iterations will be the same if we do not modify the graph in between.

Auxiliary tools

- `countNodes(g)`
- `countArcs(g)`
- `countInArcs(g,n)`
- `countOutArcs(g,n)`

The Node and Arc types are comparable by the '<' operator

```
for (ListDigraph::NodeIt u(g); u != INVALID; ++u)
  for (ListDigraph::NodeIt v(g); v != INVALID; ++v)
    if (u < v) g.addArc(u, v);
```

New Style (STL compatible) Iterators

STL compatible Iterators

```
for (auto n: g.nodes())
  cnt++;
```

Available “ranges”

- `g.nodes()`
- `g.arcs()`
- `g.inArcs(n)`
- `g.outArcs(n)`

Assign data to nodes/arcs: Maps

Usage (just like `std::vector` or `std::map`)

```
ListDigraph::NodeMap<int> map(g);

map[x] = 2;
map[y] = 3;
map[z] = map[x] + map[y];
```

Maps' properties

- Fast and efficient
  - like normal arrays (`std::vector`)
- Dynamic
  - we can allocate new maps at any time
- Automatic
  - When adding new arcs/nodes, then the corresponding maps will be augmented accordingly
  - Upon removing arcs/nodes, the corresponding map values will be destructed.

Maps II.

(Almost) arbitrary data type can be used

```
ListDigraph::NodeMap<std::string> name(g);
name[x] = "Node A";
name[y] = "Node B";
```

Example

```
ListDigraph::NodeMap<char> label(g);
char ch = 'A';
for (ListDigraph::NodeIt n(g); n != INVALID; ++n)
  label[n] = ch++;
```

Example: initial value

```
ListDigraph::NodeMap<int> out_deg(g, 0);
for (auto a: g.arcs())
  out_deg[g.source(a)]++;
```

Caution

The items added at a later time will be initialized by the default constructor!

Example

Breath First Search

```
#include<lemon/list_graph.h>
#include<queue>

void bfs(ListDigraph &g, Node s)
{
  ListDigraph::NodeMap<ListDigraph::Arc> pred(g,INVALID);
  ListDigraph::NodeMap<bool> visited(g,false);
  std::queue<Node> queue;
  queue.push(s);
  visited[s]=true;
  while(!queue.empty()) {
    for(auto a: g.outArcs(g,queue.front()) ) {
      n=g.target(a);
      if(!visited[n]) {
        queue.push(n);
        visited[n]=true;
        pred[n]=a;
      }
    }
    queue.pop();
  }
}
```

The “factory” BFS algorithm

The `Bfs` class (`#include <lemon/bfs.h>`)

- Initialization

```
ListDigraph g;
Bfs<ListDigraph> bfs(g);
```

- Execution

```
bfs.run(s);
bfs.run(s,t);
```

- Query the result

```
bfs.dist(n)
bfs.predNode(n)
bfs.predArc(n)
bfs.path(n)
...
```

A `bfs()` function (`#include <lemon/bfs.h>`)

```
ListDigraph::NodeMap<int> dist_map(g);
bfs(g).distMap(dist_map).run(s);
```

BFS example

Print paths to the nodes no farther than `max_dist`

```
Bfs<ListDigraph> bfs(g);
bfs.run(s);

for (auto n: g.nodes()) {
  if (bfs.reached(n) && bfs.dist(n) <= max_dist) {
    std::cout << gr.id(n);
    auto prev = bfs.prevNode(n);
    while (prev != INVALID) {
      std::cout << "<-" << gr.id(prev);
      prev = bfs.prevNode(n);
    }
    std::cout << std::endl;
  }
}
```

BFS/DFS execution

BFS execution

```
bfs.run(s);
```

BFS/DFS execution

BFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
bfs.start();
```

BFS/DFS execution

BFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
while (!bfs.emptyQueue()) {
  bfs.processNextNode();
  ...
}
```

BFS/DFS execution

BFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
while (!bfs.emptyQueue()) {
  bfs.processNextNode();
  ...
}
```

DFS execution

```
bfs.run(s);
```

BFS/DFS execution

BFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
while (!bfs.emptyQueue()) {
  bfs.processNextNode();
  ...
}
```

DFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
bfs.start();
```

BFS/DFS execution

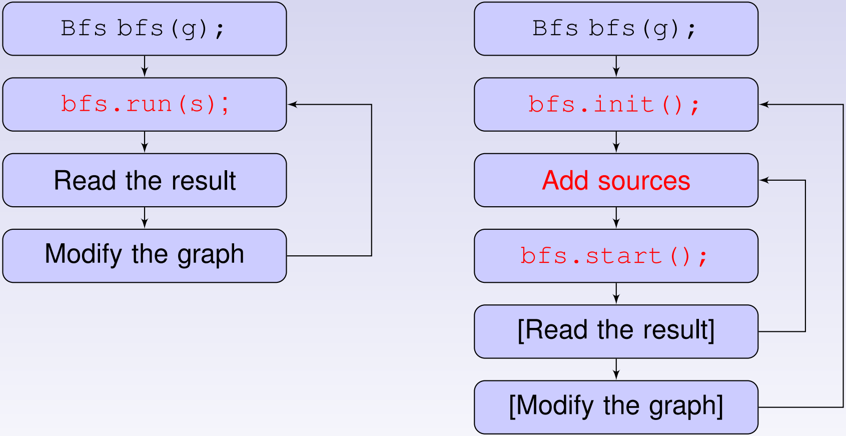
BFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
while (!bfs.emptyQueue()) {
  bfs.processNextNode();
  ...
}
```

DFS execution

```
bfs.init();
bfs.addSource(s);
[bfs.addSource(s2);]
while (!bfs.emptyQueue()) {
  bfs.processNextArc();
  ...
}
```

BFS/DFS repeated executions



BFS example II

Count the number of connected components

```
template<class G>
int connected_components(const G &g)
{
  int cnt = 0;
  Bfs<ListDigraph> bfs(g);
  bfs.init();
  for (auto n: g.nodes())
    if(!bfs.reached(n)}
      {
        cnt++;
        bfs.addSource(n);
        bfs.start();
      }
  return cnt;
}
```

The Kruskal algorithm (for min. cost spanning tree)

```
kruskal(g,in,out);
```

`g`

Can either be a directed or undirected graph

`in`

- `EdgeMap<>` or `ArcMap<>` describing the edge costs
- An arbitrary STL “Forward container” (e.g. `std::vector<T>` or `std::list<T>`), where `T` is `std::pair<GR::Arc/Edge,C>` and the costs are in increasing order.

`out`

- `EdgeMap<bool>` or `ArcMap<bool>`
- An iterator of an arbitrary Arc/Edge STL container type.

```
std::vector<Arc> tree(53);
kruskal(g,cost,tree.begin());
```

or if we don’t know the size of the tree in advance:

```
std::vector<Arc> tree;
kruskal(g,cost,std::back_inserter(tree));
```

Tools

Count items

- `countNodes()`, `countArcs()`, `countOutArcs()`, `countInArcs()`
- If a graph type records the number of items, then these functions run in $O(1)$ time.

Graph copying (there is no copy constructor!)

```
ListDigraph sg;
SmartDigraph tg;
digraphcopy(sg, tg).run();
```

Tools

Count items

- `countNodes()`, `countArcs()`, `countOutArcs()`, `countInArcs()`
- If a graph type records the number of items, then these functions run in $O(1)$ time.

Graph copying (there is no copy constructor!)

```
ListDigraph sg;
SmartDigraph tg;
ListDigraph::NodeMap<SmartDigraph::Node> nr(sg);
digraphcopy(sg, tg).nodeRef(nr).run();
```

Tools

Count items

- `countNodes()`, `countArcs()`, `countOutArcs()`, `countInArcs()`
- If a graph type records the number of items, then these functions run in $O(1)$ time.

Graph copying (there is no copy constructor!)

```
ListDigraph sg;
SmartDigraph tg;
ListDigraph::NodeMap<SmartDigraph::Node> nr(sg);
SmartDigraph::ArcMap<ListDigraph::Arc> acr(tg);
digraphcopy(sg, tg).nodeRef(nr).arcCrossRef(acr).run();
```

Arcs between two nodes

Simple search (it iterates through to outgoing arcs)

```
for(ListDigraph::Arc e=findArc(g,u,v); e!=INVALID; e=findArc(g,u,v,e))
{...  }
for(ConArcIt<ListDigraph> a(g,u,v); a!=INVALID; ++a) { ...  }
```

Static Arc Lookup Table (in $O(\log d)$ time)

```
ArcLookUp<ListDigraph arcs(g);
ListDigraph::Arc a=arcs(u,v);
...  //Modify the graph
arcs.refresh();
```

Static Arc Lookup Table (lists all parallel arcs, in $O(\log d)$ time)

```
AllArcLookUp<ListDigraph arcs(g);
for(ListDigraph::Arc a=arcs(u,v); a!=INVALID; a=arcs(u,v,a)) { ...  }
...  //Modify the graph
arcs.refresh();
```

Dynamic Arc Lookup Table (with automatic update, in $O(\log d)$ time)

```
DinArcLookUp<ListDigraph arcs(g);
for(ListDigraph::Arc a=arcs(u,v); a!=INVALID; a=arcs(u,v,a)) { ...  }
```

Graphs II.

A "graph" is not a data structure, but a "concept"

- There are various graph implementations to meet special needs.
- We can write our own (though it is not easy)
- There are also graph adaptors (see later)
- the algorithms work with any graph types

Some graph implementations

`List(Di)Graph`, the "swiss army knife".

`Smart(Di)Graph`, a more memory efficient (and faster) graph (nodes/arcs cannot be erased).

`FullGraph`,

`GridGraph`.

Graphs II.

A “graph” is not a data structure, but a “concept”

- There are various graph implementations to meet special needs.
- We can write our own (though it is not easy)
- There are also graph adaptors (see later)
- the algorithms work with any graph types

Some graph implementations

`List(Di)Graph`, the “swiss army knife”.

`Smart(Di)Graph`, a more memory efficient (and faster) graph (nodes/arcs cannot be erased).

`FullGraph`,

`GridGraph`.

Graphs III: Graph-adaptors

Shortest path from one node to all others

```
bfs(g,s).predMap(pred).run();
```

Shortest path to one node from all others

```
bfs(revGraphAdaptor(g),s).predMap(pred).run();
```

Some adaptors

`RevGraphAdaptor`, Reverses the arcs of the graph

`SubGraphAdaptor`, allows switching on/off the nodes and the arcs.

`BidirGraphAdaptor`, Every arcs will be bidirected.

...

Maps III.: The ReadMap concept

ReadMap

```
struct MyMap {
  typedef ListDigraph::Arc Key;
  typedef double Value;
  Value operator[](Key &k) const { return PI; }
};
```

Or:

```
struct MyMap:  MapBase<ListDigraph::Arc,double> {
  Value operator[](Key &k) const { return PI; }
};
```

- Wherever an algorithm except a read-only map, we can use a custom one.
- The map and the algorithm are linked in compile time $\Longrightarrow$ very efficient
- The are a lot of “map adaptors” (see later)

Readmap example

Dijkstra execution with reduced costs

```
class ReducedLengthMap  : public MapBase<Digraph::Arc,double>
{
  const Digraph &g;
  const Digraph::ArcMap<double> &orig_len;
  const Digraph::NodeMap<double> &pot;

public:
  Value operator[](Key e) const {
    return orig_len[e]-(pot[g.target(e)]-pot[g.source(e)]);
  }

  ReducedLengthMap(const Digraph &_g,
                   const Digraph::ArcMap &_o,
                   const Digraph::NodeMap &_p)
    : g(_g), orig_len(_o), pot(_p) {};
};

...
ReducedLengthMap rm(g,len,pot);
Dijkstra<Digraph,ReducedLengthMap> dij(g,rm);
dij.run(s);
...
```

Maps IV.: The WriteMap and ReferenceMap conceptek

WriteMap

```
struct MyMap:  MapBase<ListDigraph::Arc,double> {
  Value operator[](Key &k) const { return PI; }
  void set(Key &k, Value v) { ...  }
};
```

RefenenceMap

```
struct MyMap:  MapBase<ListDigraph::Arc,double> {
  const Value &operator[](Key &k) const { ...  }
  Value &operator[](Key &k) { ...  }
  void set(Key &k, Value v) { ...  }
};
```

Map adaptors

Example

```
ListDigraph graph;

typedef ListDigraph::ArcMap<double> DoubleArcMap;
DoubleArcMap length(graph);
DoubleArcMap speed(graph);

typedef DivMap<DoubleArcMap, DoubleArcMap> TimeMap;
TimeMap time(length, speed);

Dijkstra<ListDigraph, TimeMap> dijkstra(graph, time);
dijkstra.run(source, target);
```

Or:

```
dijkstra(graph, divMap(length, speed).run(source,target));
```

- `AddMap<M1,M2>`, `SubMap<M1,M2>`, `MulMap<M1,M2>` stb.
- `ComposeMap<M1,M2>`, `CombineMap<M1,M2,F,V>`, `FunctorToMap<F,K,V>`, `CombineMap<M1,M2,F,V>`, `ConvertMap<M,V>` stb.
- `ForkMap<M1,M2>`, `NotWriteMap<M>`, `NegWriteMap<M>`

Undirected graphs

```
ListGraph g;
ListGraph::Edge e = g.addEdge(u,v);
```

In one hand: works exactly like a directed graph

On the other hand: Undirected graph operations

Undirected graphs

```
ListGraph g;
ListGraph::Edge e = g.addEdge(u,v);
```

In one hand: works exactly like a directed graph

- Every `Edge` corresponds to two oppositely directed `Arcs` $\Longrightarrow$ Thus all directed graph algorithms will work.
  - Even if it doesn't make sense.

On the other hand: Undirected graph operations

Undirected graphs

```
ListGraph g;
ListGraph::Edge e = g.addEdge(u,v);
```

In one hand: works exactly like a directed graph

On the other hand: Undirected graph operations

- `ListGraph::Edge e = g.addEdge(u,v);`
  - An `ListGraph::Arc` will convert to this
  - Backward conversion: `a = direct(e,true)` and `a = direct(e, u)`
  - `oppositeArc(a)`: The oppositely directed arc
- `ListGraph::EdgeMap<int> emap(g);`
- `ListGraph::IncEgdeIt it(g,u);`
- End nodes of an `Edge`
  - This and that: `g.u(e)` and `g.v(e)`
  - The “other” end: `oppositeNode(u, e)`
  - For `IncEdge` iterators (also for `*ArcIt`, even in digraphs)
    - the node it is anchored to: `g.baseNode(e)`
    - the other: `g.runningNode(e)`