LEMON

It's lemon, it's not lime

Khue Do[1]
Quoc-Tuan Le[2]

[1]Institute of Mathematics, VAST

[2]Optimal seminar group, HUS

Overview

1. **Introduction to LEMON**
2. **Heap data structure**
3. **LEMON's performance**
4. **LEMON's graphic**

Introduction to LEMON

What is LEMON

- LEMON is an abbreviation for
  Library for Efficient Modeling and Optimization in Networks.

What is LEMON

- LEMON is an abbreviation for
  Library for Efficient Modeling and Optimization in Networks.
- It is an **open source C++ template library** for optimization tasks related to **graphs and networks**.

What is LEMON

- LEMON is an abbreviation for
  Library for Efficient Modeling and Optimization in Networks.
- It is an **open source C++ template library** for optimization tasks related to **graphs and networks**.
- It provides **highly efficient implementations** of common data structures and algorithms.

What is LEMON

- LEMON is an abbreviation for
  Library for Efficient Modeling and Optimization in Networks.
- It is an **open source C++ template library** for optimization tasks related to **graphs and networks**.
- It provides **highly efficient implementations** of common data structures and algorithms.
- It is maintained by the EGRES group at Eötvös Loránd University, Budapest, Hungary.

What is LEMON

- LEMON is an abbreviation for
  Library for Efficient Modeling and Optimization in Networks.
- It is an **open source C++ template library** for optimization tasks related to **graphs and networks**.
- It provides **highly efficient implementations** of common data structures and algorithms.
- It is maintained by the EGRES group at Eötvös Loránd University, Budapest, Hungary.
- https://lemon.cs.elte.hu/trac/lemon

Building Graphs

Creating a graph

```
using namespace lemon;
ListDiGraph g;
```

Adding nodes and arcs

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

Removing items

```
g.erase(a);
g.erase(v);
```

Iterators

Iteration on nodes

```
for(ListDigraph::NodeIt v(g); v != INVALID; ++v) {...}
```

Iteration on arcs

```
for(ListDigraph::ArcIt a(g); a != INVALID; ++a)
for(ListDigraph::OutArcIt a(g,v); a != INVALID; ++a)
for(ListDigraph::InArcIt a(g,v); a != INVALID; ++a)
```

Note: INVALID is a constant, which converts to each and every iterator and graph item type.

Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.

Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
- This provides a more convenient interface.

Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
- This provides a more convenient interface.
- The program context always indicates whether we refer to the iterator or to the graph item.

Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use `operator*()`.
- This provides a more convenient interface.
- The program context always indicates whether we refer to the iterator or to the graph item.

Iterators

- Contrary to C++ STL, LEMON iterators are convertible to the corresponding item types without having to use operator*().
- This provides a more convenient interface.
- The program context always indicates whether we refer to the iterator or to the graph item.

LEMON: Printing node identifiers

```
for(ListDigraph::NodeIt v(g); v!=INVALID; ++v)
  std::cout « "dist[v] = " « dist[v] « std::endl;
```

BGL: Printing node identifiers

```
graph_t::vertex_iterator vi, vend;
for(tie(vi, vend) = vertices(g); vi != vend; ++vi)
  std::cout « *vi « ": " « dist[*vi] « std::endl;
```

Maps

- The graph classes represent only the pure structure of the graph.

Maps

- The graph classes represent only the pure structure of the graph.
- All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Maps

- The graph classes represent only the pure structure of the graph.
- All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Maps

- The graph classes represent only the pure structure of the graph.
- All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Creating maps

```
ListDigraph::NodeMap<std::string> label(g);
ListDigraph::ArcMap<int> cost(g);
```

Maps

- The graph classes represent only the pure structure of the graph.
- All associated data (e.g. node labels, arc costs or capacities) are stored separately using so-called maps.

Creating maps

```
ListDigraph::NodeMap<std::string> label(g);
ListDigraph::ArcMap<int> cost(g);
```

Accessing map values

```
label[s] = "source";
cost[e] = 2*cost[f];
```

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.
  - When you leave its scope, the map will be deallocated automatically.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.
  - When you leave its scope, the map will be deallocated automatically.
- **Automatic.** The maps are updated automatically on the changes of the graph.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.
  - Whenever you need, you can allocate a new map.
  - When you leave its scope, the map will be deallocated automatically.
- **Automatic.** The maps are updated automatically on the changes of the graph.
  - If you add new nodes or arcs to the graph, the storage of the existing maps will be expanded and the new slots will be initialized.

Benefits of Graph Maps

- **Efficient.** Accessing map values is as fast as reading or writing an array.
- **Dynamic.** You can create and destruct maps freely.
    - Whenever you need, you can allocate a new map.
    - When you leave its scope, the map will be deallocated automatically.
- **Automatic.** The maps are updated automatically on the changes of the graph.
    - If you add new nodes or arcs to the graph, the storage of the existing maps will be expanded and the new slots will be initialized.
    - If you remove items from the graph, the corresponding values in the maps will be properly destructed.

Compile your first code

```cpp
#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 LEMON library here." << endl;
    cout << "We have a directed graph with " << countNodes(g) << " nodes "
    << "and " << countArcs(g) << " arc." << endl;
    return 0;
}
```

Build and Install from Source

LEMON is basically a large collection of C++ header files plus a small static library.

Supporting various operating systems (Windows; Linux, Solaris, OSX, AIX and other Unices), and compilers/IDEs (GCC, Intel C++, IBM XL C++, Visual C++, MinGW, CodeBlocks).

- Installation guide for Linux
- Installation guide for Windows

Compile your first code

If LEMON is installed **system-wide** (into directory /usr/local):

```
g++ −o hello_lemon hello_lemon.cc −lemon
```

Compile your first code

If LEMON is installed **system-wide** (into directory /usr/local):

```
g++ −o hello_lemon hello_lemon.cc −lemon
```

If LEMON is installed **user-local** into a directory (e.g. /lemon)

```
g++ −o hello_lemon −I ~/lemon/include hello_lemon.cc −L ~/lemon/lib −lemon
```

Compile your first code

If LEMON is installed **system-wide** (into directory /usr/local):

```
g++ −o hello_lemon hello_lemon.cc −lemon
```

If LEMON is installed **user-local** into a directory (e.g. /lemon)

```
g++ −o hello_lemon −I ~/lemon/include hello_lemon.cc −L ~/lemon/lib −lemon
```

Then, you can run by the following command

```
./hello_lemon
```

Compile your first code

If LEMON is installed **system-wide** (into directory /usr/local):

```
g++ −o hello_lemon hello_lemon.cc −lemon
```

If LEMON is installed **user-local** into a directory (e.g. /lemon)

```
g++ −o hello_lemon −I ~/lemon/include hello_lemon.cc −L ~/lemon/lib −lemon
```

Then, you can run by the following command

```
./hello_lemon
```

If everything has gone well, then our program prints out the followings

```
Hello World! This is LEMON library here.
We have a directed graph with 2 nodes and 1 arc.
```

Heap data structure

Heap data structure

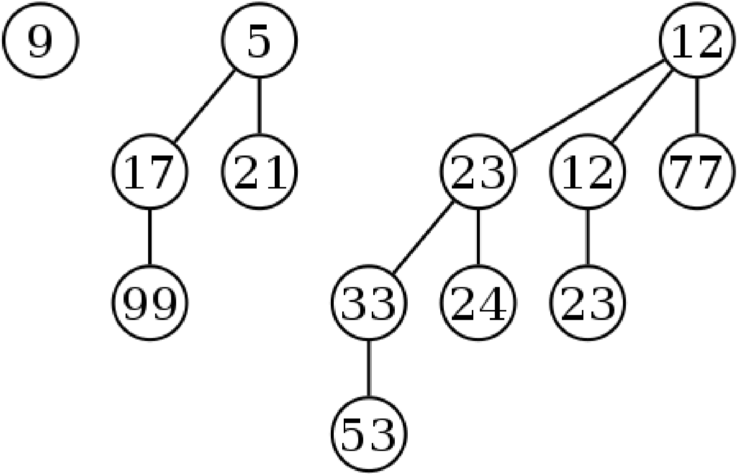
Definition

A **Heap** is a special tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

- **Max-Heap:** The root node hold the greatest value. Same property is hold for any sub tree.
- **Min-Heap:** he root node hold the smallest value. Same property is hold for any sub tree.

Examples of heap data structure

Example of a min-heap:



Binary Heap

Definition

A **binary heap** is a binary tree such that:

- It is a complete tree (as much complete as possible).
- It is either a Max Heap or Min Heap.

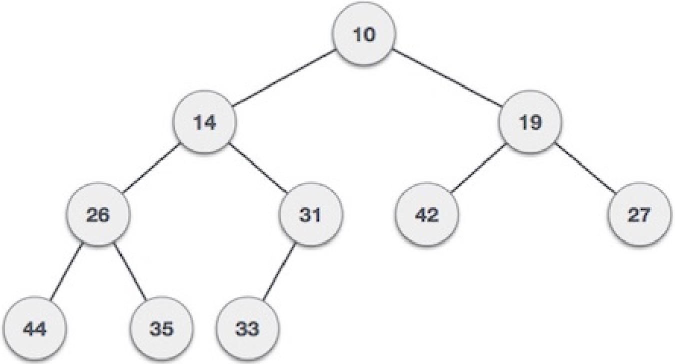
Representation of binary heap

Store the heap in an array **arr** such that:

1. The root element will be at **arr**[0].
2. **arr**$[(i-1)/2]$ returns the parent node.
3. **arr**$[2i+1]$ returns the left node.
4. **arr**$[2i+2]$ returns the right node.

Example

Example of a min-binary heap:



Implementation on heap

There are 5 fundamental operations can be implement on heap.

1. Return the smallest element on the heap.
2. Remove the smallest element on the heap.
3. Decrease the value of an element on the heap.
4. Insert a new element to the heap.
5. Delete an element on the heap.

Every of the above operations guaranteed to **preserve the structure of the heap**.

getSmallest() operation

- Return the smallest element on the heap.
- The smallest element is exactly the **root node**.
- The complexity of the **getSmallest()** operation is $O(1)$.

Heapify procedure

A heapify procedure is procedure to maintain the heap property.

Algorithm 1 Heapify procedure

```
procedure HEAPIFY(arr, i)
    $l \leftarrow Left(i)$
    $r \leftarrow Right(i)$
    if $l \leq$ arr.heap-size and arr[$l$] $<$ arr[$r$] then
        $smallest \leftarrow l$
    else
        $smallest \leftarrow i$
    end if
    if $r \leq$ arr.heap-size and arr[$r$] $<$ arr[$smallest$] then
        $smallest \leftarrow l$
    end if
    if $smallest \neq i$ then
        exchange arr[$i$] with arr[$smallest$]
        HEAPIFY(arr, $smallest$)
    end if
end procedure
```

Complexity of heapify procedure

- The heapify procedure is just a single direction traversal through the heap tree.
- The worst-case is that the heapify traversal through every layer of the heap tree.
- Hence, the running time of heapify procedure is $O(\log n)$.

extractMin() operation

- Remove the minimum element from the heap.
- Actually removing the root node of the heap.
- Call the heapify procedure to reconstruct the heap.
- The complexity of the **extractMin()** operation is $O(\log n)$.

decreaseKey() operation

- Decrease the value of a key on the heap.
- Call the heapify procedure to reconstruct the heap.
- The comlexity of the **decreaseKey()** operation is $O(\log n)$.

insert() procedure

- Add a new key at the end of the tree.
- Call the heapify procedure to reconstruct the heap.
- The comlexity of the **insert()** operation is $O(\log n)$.

delete() procedure

- Deleting a key from the procedure.
- Decrease the value of the chosen key to minus infinity by using **decreaseKey()** operation.
- The root key now becomes the minus infinity key.
- Apply the **extractMin()** operation to get rid the current root key.
- The complexity of **delete()** operation is $O(\log n)$.

Mergeable heap

Definition

A **mergeable heap** is a data structure that supports the following operations:

- Create a new heap containing no elements.
- Inserts an element into the heap.
- Return the element whose hold the minimum value.
- Delete the element whose hold the minimum value.
- **Create a new heap that contains all the elements of the heap H_1 and H_2.**
- Decrease the value of a chosen element in the heap.
- Delete an element from the heap.

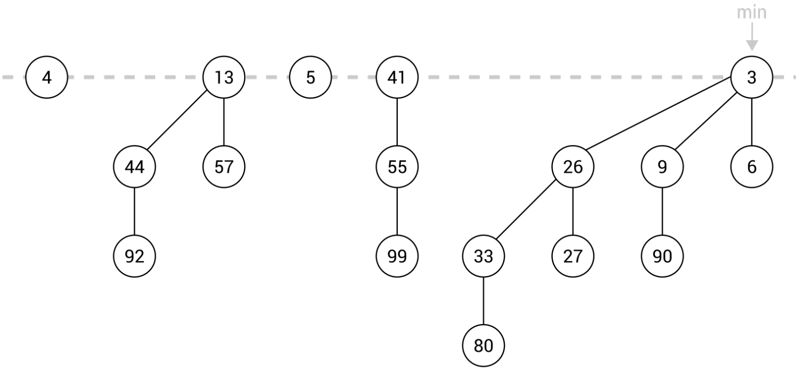
Fibonacci heap

Definition

A **fibonacci heap** is a collection of rooted trees such that each tree obeys the **min-heap property**.

Example of Fibonacci heap

Example of a min-Fibonacci heap:

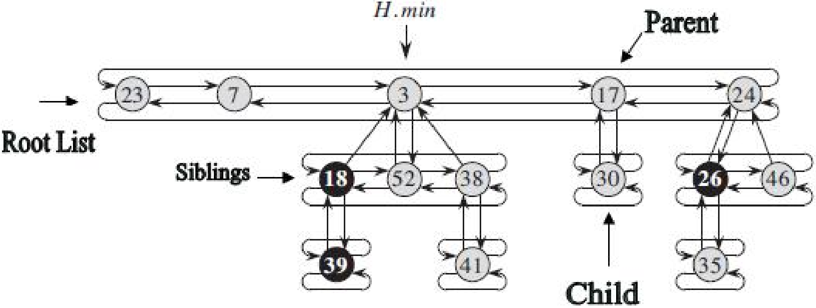


Detail structure of Fibonacci heap

- Collection of rooted **min-heap** tree.
- A **pointer** to the minimum value.
- **Circular doubly linked list** to connect all tree roots.

Example of Fibonacci heap

More detail example of a min-Fibonacci heap:



Fibonacci heap insert() procedure

- Make a new tree with root is inserted element.
- Check whether if the new element has the smallest value.
- Hence, the complexity of **insert()** operation is $O(1)$.

Fibonacci heap merge() procedure

- Simply merge two lists together.
- Hence, the complexity of **merge()** operation is $O(1)$.

Fibonacci heap extractMin() procedure

Algorithm 2 extractMin procedure

```
procedure FIB-EXTRACT-MIN($H$)
    $z = H.min$
    if $z \neq NIL$ then
        for each child $x$ of $z$ do
            add $x$ to the root lists of $H$
            $x.p = NIL$
        end for
    end if
    if $H.n = 1$ then
        $H.min = NIL$
    else
        $H.min = z.right$
        $CONSOLIDATE(H)$
    end if
    remove $z$ from $H$
    $H.n = H.n - 1$
end procedure
```

Heap consolidate procedure

- The consolidate procedure can be described as below:
  1. Find two roots x and y which have the same degree. WLOG, let $x.key \leq y.key$.
  2. Link y to x by making y a child of x.
  3. Find the minimum root z. Let $H.min = z$.
- The amortized for each above operation take maximum $O(\log n)$ time.
- Hence, the amortized complexity of consolidate procedure and **extractMin()** procedure for so on is $O(\log n)$.

Fibonacci heap decreaseKey() procedure

Algorithm 3 decreaseKey procedure

```
procedure FIB-HEAP-DECREASE-KEY($(H, x, k)$)
    if $k > x.key$ then error "new key is greater than current key"
    end if
    $x.key = k$
    $y = x.p$
    if $y \neq NIL$ and $x.key < y.key$ then
        $CUT(H, x, y)$
        $CASCADING - CUT(H, y)$
    end if
    if $x.key < H.min.key$ then
        $H.min = x$
    end if
end procedure
```

Fibonacci heap decreaseKey() procedure

The **CUT** function in **decreaseKey()** procedure

Algorithm 4 CUT procedure

```
procedure CUT($H, x, y$)
    remove $x$ from the child list of $y$, decrementing $y.degree$
    add $x$ to the root list of $H$
    $x.p = NIL$
    $x.mark =$ FALSE
end procedure
```

Fibonacci heap decreaseKey() procedure

The **CASCADING-CUT** function in **decreaseKey()** procedure

Algorithm 5 CASCADING-CUT procedure

```
procedure CASCADING-CUT($H, y$)
    $z = y.p$
    if $z \neq NIL$ then
        if $y.mark == $ FALSE then
            $y.mark = $ TRUE
        else
            $CUT(H, y, z)$
            $CASCADING - CUT(H, z)$
        end if
    end if
end procedure
```

Fibonacci heap decreaseKey() procedure

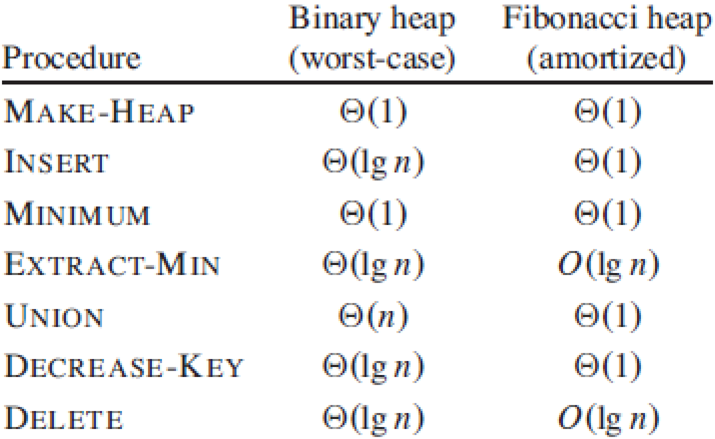
The amortized complexity of decreaseKey() procedure is $O(1)$.

Fibonacci heap delete() procedure

- Using the same argument as for the binary heap.
- Hence, the amortized complexity of **delete()** operation is $O(\log n)$.

comparison between binary heap and Fibonacci heap

The running time of each operation is being compared via the below table:



Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Application of heap data structure

- As a sorting algorithm.
- To create a **priority queues**, which is used in many algorithm such as:
    - Prim's algorithm for finding minimum spanning tree.
    - **Dijkstra algorithm for finding all pair shortest path.**
    - **Perform better than search tree.**

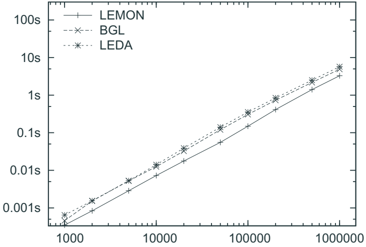
Good page for data visualization

This page contains some fancy for data visualization.

- Heap
- Binomal Queue
- Fibonaci Heaps
- Leftist Heap
- Skew Heap
- . . .

LEMON's performance

Performance[1]

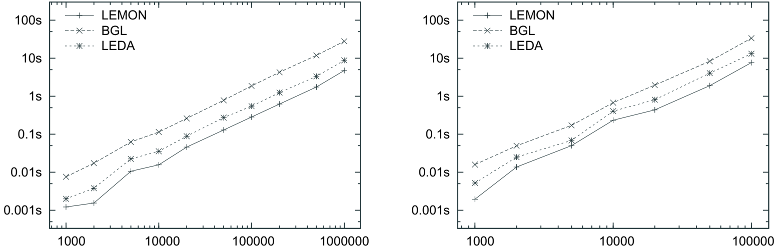


Sparse graphs ($m \approx n \log_2 n$)

Dense graphs ($m \approx n\sqrt{n}$)

Figure 1: Benchmark results for the Dijkstra algorithm.

Performance[1]

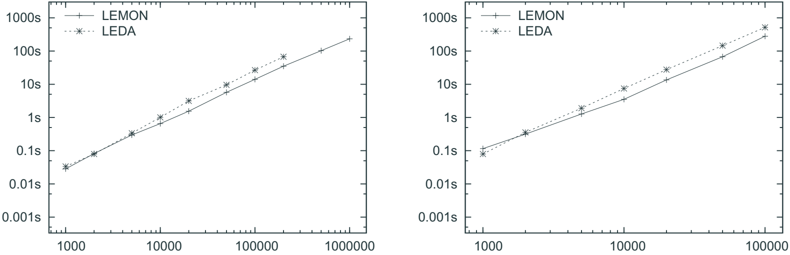


Sparse graphs ($m \approx n \log_2 n$)

Dense graphs ($m \approx n\sqrt{n}$)

Figure 2: Benchmark results for maximum flow algorithms.

Performance[1]



Sparse graphs ($m \approx n \log_2 n$)

Dense graphs ($m \approx n\sqrt{n}$)

Figure 3: Benchmark results for minimum cost flow algorithms.

Performance[1]

Graph type	Algorithm	Sparse graph	Dense graph
LEMON	LEMON	3.27s	1.13s
LEMON	BGL	4.36s	1.07s
BGL	LEMON	3.55s	1.56s
BGL	BGL	4.90s	2.08s

Table 1: Benchmark results for the largest instances of the shortest path problem combining LEMON and BGL implementations.

---

[1] The benchmark tests were performed on a machine with AMD Opteron Dual Core 2.2 GHz CPU and 16 GB memory (1 MB cache), running openSUSE 10.1 operating system. The codes were compiled with GCC version 4.1.0 using -O3 optimization flag.

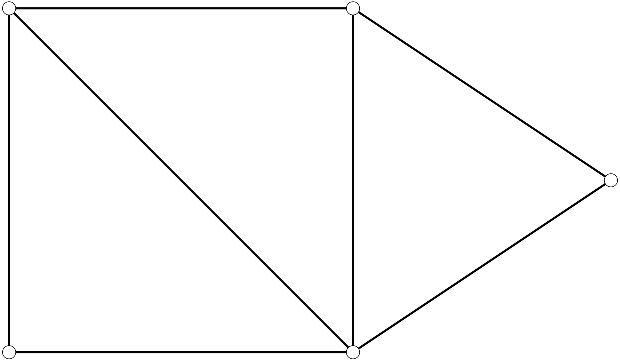
Heap performance

n Type	10	100	1000
BinHeap	0.000857	0.01636	0.1152
QuadHeap	0.000847	0.01748	0.113
Dheap	0.000872	0.01652	0.1156
FibHeap	0.001063	0.01932	0.1372
PairingHeap	0.001153	0.022	0.1764
RadixHeap	0.000992	0.02948	0.1956
BinomialHeap	0.0003	0.01632	0.1094
BucketHeap	0.000545	0.02976	0.218

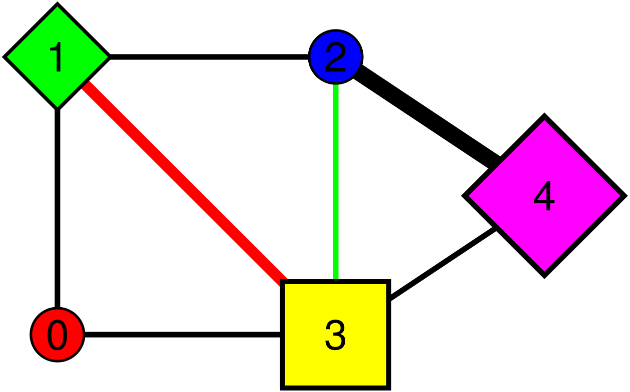
Table 2: Results for the Dijkstra algorithm compiling with LEMON heap options.

LEMON's graphic

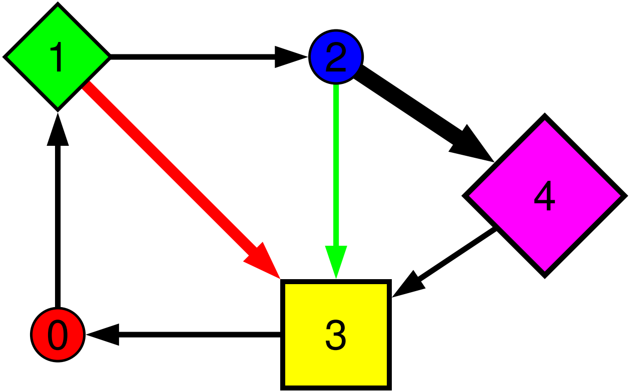
Graphic



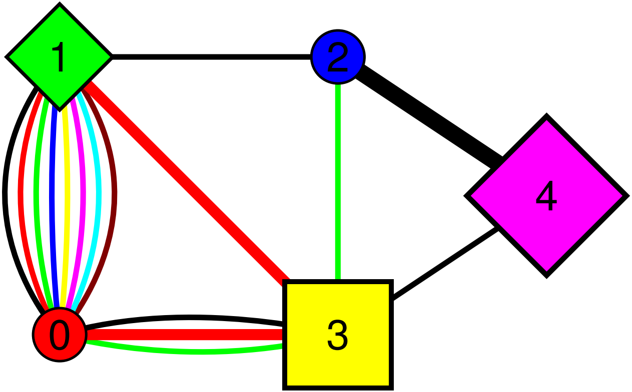
Graphic



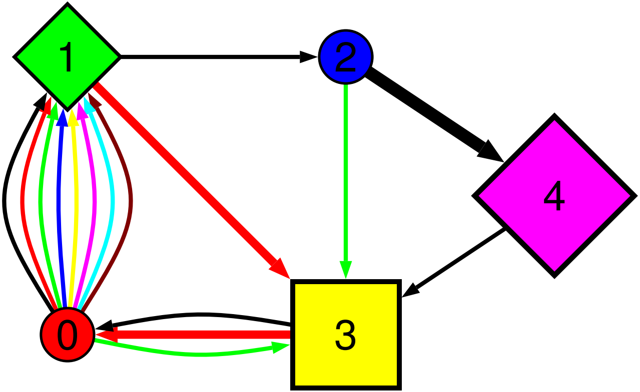
Graphic



Graphic



Graphic



Graphic