From Algorithms to Working Programs: On the Use of Program Checking in LEDA

Kurt Mehlhorn[1] and Stefan Näher[2]

[1] Max-Planck-Insitut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany
(mehlhorn@mpi-sb.mpg.de)
[2] Martin-Luther-Universität Halle-Wittenberg, FB Mathematik und Informatik, Kurt-Mothes-Str. 1, D-06099 Halle (Saale), Germany
(naeher@infsn.informatik.uni-halle.de)

Abstract. We report on the use of program checking in the LEDA library of efficient data types and algorithms.

1 Introduction

LEDA [MN95, MNU97, MN98] is a collection of implementations of data structures and combinatorial algorithms. In the almost ten years of the project we translated hundreds of algorithms into programs. For the purpose of this paper an *algorithm* is the description of a problem solving method intended for a human reader and a *program* is a description intended for machine execution. Clearly, algorithms and programs are quite different animals; algorithms are formulated in natural language and are published in papers and books, and programs are written in computer languages and are executed on machines. We expected the process of implementation to be tedious and time-consuming, indeed, it was, but not intellectually challenging. We now believe that the implementation process is very difficult and challenging. We encountered the following difficulties.

- We and our co-workers are not perfect programmers. We make mistakes. We use program checking [BK89, SM90, WB97] to cope with the possibility of error.
- Geometric algorithms are usually designed for a hypothetical machine, the so-called *Real RAM*, which is equipped with arithmetic over the real numbers. The efficient realization of the Real RAM is non-trivial.
- The primary goal for algorithm design is asymptotic running time, the secondary design goal is elegance (remember that algorithms are intended for human readers). Actual running time is usually not a design goal. The actual running time of algorithms with the same asymptotic behavior may differ widely.

In this paper we concentrate on the first item. For the other two items we refer the reader to [MN98] and the references therein.

2 Program Checking

We start with an example and then generalize.

2.1 Planarity Testing

A graph is *planar* if it can be drawn in the plane without edge crossings. A planarity tester

```
bool Is_Planar(const graph& G)
```

takes a graph G and returns true if G is planar and returns false if G is non-planar[1]. The planarity test played a crucial role in the development of LEDA.

There are several linear time algorithms for planarity testing [HT74, LEC67, BL76]. An implementation of the Hopcroft and Tarjan algorithm was added to LEDA in 91. The implementation had been tested on a small number of graphs. In 93 we were sent a graph together with a planar drawing of it. However, our program declared the graph non-planar. It took us some days to discover the bug. More importantly, we realized that a complex question of the form "is this graph planar" deserves more than a yes-no answer. We adopted the thesis that

a program should justify (prove) its answers in a way that is easily checked by the user of the program.

What does this mean for the planarity test?

If a graph is declared planar, a proof should be given in the form of a combinatorial embedding or a planar drawing. If a graph is declared non-planar, a proof should be given in the form of a Kuratowski subgraph[2]. Linear time algorithms for computing planar embeddings are described in [CNAO85, NC88, MM95] and linear algorithms for the computation of Kuratowski subgraphs are given in [Wil84, Kar90, HMN96]. The function

```
bool Is_Planar(graph& G, list<edge>& K)
```

returns true if G is planar and returns false otherwise. If G is planar, it also reorders the adjacency lists of G such that G becomes a plane map[3]. If G is non-planar, a set of edges forming a Kuratowski subgraph is returned in K. The function runs in linear time $O(n + m)$, where n and m are the number of nodes and edges of G, respectively. Its implementation is discussed in the chapter on

[1] We use the syntax of C++ in our examples. All functions mentioned in the paper are available in LEDA.

[2] Kuratowski [Kur30] has shown that every non-planar graph contains a subdivision of either K_5, the complete graph on five nodes, or $K_{3,3}$, the complete bipartite graph with three nodes on either side.

[3] A map is an undirected graph in which a cyclic order is imposed on the edges incident to any node. A map is plane if there is a planar embedding preserving the cyclic orders. We use plane map and combinatorial embedding as synonyms.

embedded graphs of [MN98]. If LEDA is installed on your computer system, you may want to exercise the planarity test demo before proceeding.

```
int GENUS(const graph& G)
{ if ( !Is_Map(G) ) error_handler(1,"Genus only applies to maps");
  int n = G.number_of_nodes();
  if ( n == 0 ) return 0;
  int nz = 0;
  node v;
  forall_nodes(v,G)
     if ( outdeg(v) == 0 )  nz++;
  int m = G.number_of_edges();
  node_array<int> cnum(G);
  int c = COMPONENTS(G,cnum);

  edge_array<bool> considered(G,false);
  int fc = 0;
  edge e;
  forall_edges(e,G)
  { if ( !considered[e] )
    { // trace the face to the left of e
      edge e1 = e;
      do { considered[e1] = true;
           e1 = G.face_cycle_succ(e1);
         }
      while (e1 != e);
      fc++;
    }
  }
  return (m/2 - n - nz - fc + 2*c)/2;
}
```

Fig. 1. A map is plane if and only if its genus is zero. The genus of a map is computed according to Euler's formula: one half of the number of undirected edges minus the number of nodes minus the number of isolated nodes minus the number of face cycles plus twice the number of connected components. The face cycle successor of an edge e is the next edge out of the target of e in the cyclic order of the edges around the target of e. In the LEDA representation of maps every undirected edge is represented by a pair of directed edges.

The crucial observation is now that the justifications, which Is_Planar gives for its answers, are easily checked. It is well-known that a connected map is plane if it satisfies the so-called Euler-relation $f - m + n - 2 = 0$, where f is the number of face cycles. It is also well-known that a graph is non-planar if it contains a Kuratowski subgraph. The functions

```
int GENUS(const graph& G)
bool CHECK_KURATOWSKI(const graph& G, const list<edge>& K)
```

compute the genus of a map G and check whether K is a Kuratowski subgraph of G, respectively. The implementation of the former function is shown in Figure 1, the implementation of the latter function is equally simple.

		BL_PLANAR			HT_PLANAR	
	G	T	T + E or K	C	T	T + E
P	0.25	0.52	0.53	0.08	0.84	1.32
	0.52	1.05	1.09	0.16	1.74	2.69
	1.07	2.13	2.22	0.33	3.59	5.77
P + $K_{3,3}$	0.309999	0.379999	1.77	0.0600014	0.83	–
	0.529999	0.640001	2.68	0.129997	1.7	–
	1.12	1.27	5.44	0.23	3.45	–
P + K_5	0.32	0.369999	1.82	0.0600014	0.869999	–
	0.549999	0.530003	2.57	0.119999	1.68	–
	1.1	1.3	7.09	0.229996	3.56001	–
MP	0.269997	0.720001	0.720001	0.110001	1.21	1.91
	0.459999	1.46	1.45	0.220001	2.50999	4.01
	0.93	2.94	2.92	0.440002	5.06	8.3
MP + e	0.269997	0.550003	2.05	0.0699997	0.330002	–
	0.449997	1.07001	3.81	0.139999	0.659996	–
	0.93	2.47	8.55	0.289993	1.35	–

Table 1. The running times of functions related to planarity: The first column shows the type of the input graph, the second column shows the time for the call *BL_PLANAR*(G), the third column shows the time for the call *BL_PLANAR*(G, K), the fourth column shows the time required to check the result of the computation in the third column, i.e, the time for the call *Genus*$(G) == 0$, if G is planar, and the call *CHECK_KURATOWSKI*(G, K) if G is non-planar, the fifth column shows the time for the call *HT_PLANAR*(G), and the last column shows the time for the call *HT_PLANAR*(G, K). The last call is only made when G is planar, since there there is no efficient Kuratowski finder implemented for the Hopcroft-Tarjan planarity test.
The meaning of the first column is as follows: P stands for a random planar map with n nodes and m uedges, P + $K_{3,3}$ stands for a random planar map with n nodes and m uedges plus a $K_{3,3}$ on six randomly chosen nodes, P + K_5 stands for a random planar mao with n nodes and m uedges plus a K_5 on five randomly chosen nodes, MP stands for a maximal planar map with n nodes, and MP + e stands for a maximal planar graph plus one additional edge between two random nodes that are not connected in G. In all cases the edges of the graph were permuted before the tests were started.
For each type of graph we used $n = 2^i \cdot 1000$, $m = 2^i \cdot 2000$ for $i = 0$, 1, and 2.

Table 1 shows the running times of several functions related to planarity. In this table BL_PLANAR stands for the planarity test of Lempel, Even, and Cederbaum with PQ-tree data structure of Booth and Luecker, the embedding algorithm of Chiba et al., and the Kuratowski finder of Hundack et al., and HT_PLANAR stands for the planarity test of Hopcroft and Tarjan and the embedding algorithm of Mehlhorn and Mutzel.

2.2 General Remarks

What have we achieved?

Verification for every Problem Instance: When a graph is declared planar, the resulting plane map is checked by testing whether its genus is zero, and if a graph is declared non-planar, the subgraph K is checked by CHECK_KURATOWSKI. In this way, the correctness of Is_Planar is established for each problem instance.

Trust with Minimal Investment: A user of Is_Planar does *not* have to understand the intricacies of the planarity test. It suffices to understand the functions GENUS and CHECK_KURATOWSKI. The implementation of either function is less than a page long and the underlying mathematics is simple compared to the mathematics underlying the planarity test. Observe that one only needs to understand that maps of genus zero are plane (about a two page proof) and that the existence of a Kuratowski subgraph implies non-planarity (again about a two page proof). There is, however, no need to understand why every non-planar graph contains a Kuratowski subgraph. The implementation proves this fact for every problem instance. In this way checkers allow to develop trust in an implementation with only minimal intellectual investment. It is even conceivable that checkers can be formally verified by means of automatic program verification. [BSM97] is a first example.

Program Libraries: Program libraries contain implemented algorithms. The implementor of a library may want to hide his code (after all, the source code of the programs constitutes his intellectual capital), but he may also want to make a convincing case that his code is correct. Program checkers resolve the conflict. We use the following guidelines for the specification and implementation of functions.

(1) Define the problem to be solved and what constitutes a justification for an answer.
(2) Prove that the suggested justification indeed proves correctness for any particular instance.
(3) Define the interface of the function.
(4) Define the interface of the checker and give its implementation.
(5) Give the implementation of the function. There is no need to make the implementation public.

A *matching* in a graph G is a subset M of the edges of G such that no two share an endpoint.

An odd-set cover OSC of G is a labeling of the nodes of G with non-negative integers such that every edge of G (which is not a self-loop) is either incident to a node labeled 1 or connects two nodes labeled with the same i, $i \geq 2$.

Let n_i be the number of nodes labeled i and consider any matching N. For i, $i \geq 2$, let N_i be the edges in N that connect two nodes labeled i. Let N_1 be the remaining edges in N. Then $|N_i| \leq \lfloor n_i/2 \rfloor$ and $|N_1| \leq n_1$ and hence

$$|N| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$$

for any matching N and any odd-set cover OSC.

It can be shown that for a maximum cardinality matching M there is always an odd-set cover OSC with

$$|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

thus proving the optimality of M. In such a cover all n_i with $i \geq 2$ are odd, hence the name.

list<edge> MAX_CARD_MATCHING(*graph G*, *node_array<int>& OSC*,
int heur = 0)

computes a maximum cardinality matching M in G and returns it as a list of edges. The algorithm ([Edm65, Gab76]) has running time $O(nm \cdot \alpha(n, m))$. With *heur* = 1 the algorithm uses a greedy heuristic to find an initial matching. This seems to have little effect on the running time of the algorithm.
An odd-set cover that proves the maximality of M is returned in OSC.

bool CHECK_MAX_CARD_MATCHING(*graph G*, *list<edge> M*,
node_array<int> OSC)

checks whether M is a maximum cardinality matching in G and OSC is a proof of optimality. Aborts if this is not the case.

Fig. 2. The manual page for maximum cardinality matching. The first paragraph defines the problem, the second paragraph defines the notion of proof and the third and fourth paragraph establish that an odd-set cover constitutes a proof of optimality. Observe, that is is *not* necessary to understand why odd-set covers proving optimality exist.

Figures 2 and 3 illustrate items (1) to (4) for the case of maximum cardinality matchings in general graphs.

Debugging: Program checking amounts to a complete check of the post-condition of a program. It allows to assume that potentially incorrect programs are correct.

```
static bool False(string s)
{ cerr << "CHECK_MAX_CARD_MATCHING: " << s << "\n"; return false; }

bool CHECK_MAX_CARD_MATCHING(const graph& G, const list<edge>& M,
                                    const node_array<int>& OSC)
{ int n = Max(2,G.number_of_nodes());
  int K = 1;
  array<int> count(n);
  for (int i = 0; i < n; i++) count[i] = 0;
  node v; edge e;

  forall_nodes(v,G)
  { if ( OSC[v] < 0 || OSC[v] >= n )
     return False("negative label or label larger than n - 1");
    count[OSC[v]]++;
    if (OSC[v] > K) K = OSC[v];
  }

  int S = count[1];
  for (int i = 2; i <= K; i++) S += count[i]/2;
  if ( S != M.length() )
    return False("OSC does not prove optimality");

  node_array<int> deg(G,0);
  forall(e,M) { deg[G.source(e)]++; deg[G.target(e)]++; }
  forall_nodes(v,G)
    if (deg[v] > 1) return False("M is not a matching");

  forall_edges(e,G)
  { node v = G.source(e); node w = G.target(e);
    if ( v == w || OSC[v] == 1 || OSC[w] == 1 ||
           ( OSC[v] == OSC[w] && OSC[v] >= 2) ) continue;
    return False("OSC is not a cover");
  }
  return true;
}
```

Fig. 3. The checker for maximum cardinality matchings.

If a program operates correctly on a particular instance, fine, and if it operates incorrectly, it is caught by the checker. Thus, if all subroutines of a function f are checked, no checker of a subroutine fires, and an error occurs during the execution of f, the error must be in f. This feature of program checking is extremely useful during the debugging phase of program development.

Testing: Program checking supports testing. Traditionally, testing is restricted to problem instances for which the solution is known by other means. Program checking allows to test on *any* instance. For example, we use the following program (among others) to check the matching algorithm.

```
for (int n = 0; n < 100; n++)
  for (int m = 0; m < 100; m++)
    { random_graph(G,n,m); // random graph with n nodes and m edges
      list<edge> M = MAX_CARD_MATCHING(G,OSC);
      CHECK_MAX_CARD_MATCHING(G,M,OSC);
    }
```

Hidden Assumptions: A checker can only be written if the problem at hand is rigorously defined. We noticed that some of our specifications contained hidden assumptions which were revealed during the design of the checker. For example, an early version of our biconnected components algorithm assumed that the graph contains no isolated nodes.

3 Conclusion

At the time of this writing LEDA contains checkers for most network algorithms (mostly based on linear programming duality), for planarity testing, for priority queues, and for the basic geometric algorithms (convex hulls, Delaunay diagrams, and Voronoi diagrams). Program checking has greatly increased our confidence in the correctness of our implementations. For further reading on program checking we refer the reader to [SM90, BS94, SM91, BSM97, BS95, BSM95, SWM95, BK89, BLR90, BW96, WB97], [MN98, AL94, OLPT97, MNS+96].

References

[AL94] N.M. Amato and M.C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, pages 164–173, 1994.

[BK89] M. Blum and S. Kannan. Programs That Check Their Work. In *Proc. of the 21th Annual ACM Symp. on Theory of Computing*, 1989.

[BL76] K.S. Booth and G.S. Luecker. Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using *PQ*-tree Algorithms. *Journal of Comp. and Sys. Sciences*, 13:335–379, 1976.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Annual ACM Symp. on Theory of Computing*, pages 73–83, 1990.

[BS94] J. D. Bright and G. F. Sullivan. Checking mergeable priority queues. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing*, pages 144–153, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[BS95] J. D. Bright and G. F. Sullivan. On-line error monitoring for several data structures. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 392–401, Pasadena, California, 1995.

[BSM95] J. D. Bright, G. F. Sullivan, and G. M. Masson. Checking the integrity of trees. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 402–413, Pasadena, California, 1995.

[BSM97] J.D. Bright, G. F. Sullivan, and G. M. Masson. A formally verified sorting certifier. *IEEE Transactions on Computers*, 46(12):1304–1312, 1997.

[BW96] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Trans. Comput.*, 45(4):385–393, April 1996.

[CNAO85] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using PQ-trees. *Journal of Computer and System Sciences*, 30(1):54–76, February 1985.

[Edm65] J. Edmonds. Maximum matching and a polyhedron with 0,1 - vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[Gab76] H.N. Gabow. An efficient implementation of Edmond's algorithm for maximum matching on graphs. *JACM*, 23:221–234, 1976.

[HMN96] C. Hundack, K. Mehlhorn, and S. Näher. A Simple Linear Time Algorithm for Identifying Kuratowski Subgraphs of Non-Planar Graphs. Manuscript, 1996.

[HT74] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21:549–568, 1974.

[Kar90] A. Karabeg. Classification and detetection of obstructions to planarity. *Linear and Multilinear Algebra*, 26:15–38, 1990.

[Kur30] C. Kuratwoski. Sur le problème the courbes guaches en topologie. *Fundamenta Mathematicae*, 15:271–283, 1930.

[LEC67] A. Lempel, S. Even, and I. Cederbaum. An Algorithm for Planarity Testing of Graphs. In P. Rosenstiehl, editor, *Theory of Graphs, International Symposium, Rome*, pages 215–232, 1967.

[MM95] K. Mehlhorn and P. Mutzel. On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm. *Algorithmica*, 16(2):233–242, 1995.

[MN95] K. Mehlhorn and S. Näher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.

[MN98] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1998. Draft versions of some chapters are available at `http://www.mpi-sb.mpg.de/~mehlhorn`.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and Ch. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proc. of the 12th Annual Symposium on Computational Geometry*, pages 159–165, 1996.

[MNU97] Kurt Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.5). Technical report, Max-Planck-Institut für Informatik, 1997. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[NC88] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms.* Annals of Discrete Mathematics (32). North-Holland Mathematics Studies, 1988.

[OLPT97] O.Devillers, G. Liotta, F.P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. Technical report, Center for Geometric Computing, Department of Computer Science, Brown University, 1997.

[SM90] G. F. Sullivan and G. M. Masson. Using certification trails to achieve software fault tolerance. In Brian Randell, editor, *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS '90)*, pages 423–433, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society Press.

[SM91] G. F. Sullivan and G. M. Masson. Certification trails for data structures. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pages 240–247, 1991.

[SWM95] G.F. Sullivan, D.S. Wilson, and G.M. Masson. Certification of computational results. *IEEE Transactions on Computers*, 44(7):833–847, 1995.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[Wil84] S.G. Williamson. Depth-First Search and Kuratowski Subgraphs. *JACM*, 31(4):681–693, 1984.