Euler Tours

Kurt Mehlhorn

June 28, 2000

Let G be a connected undirected graph in which every node has even degree. We show how to construct an Euler tour of G, i.e., a cycle containing all edges of G. Our strategy may be called "constructing and extending a cycle".

The Idea: The strategy is easy to explain. Assume that we have already constructed a cycle C. If C contains all edges of G we are done. Otherwise, there must a vertex v on C such that not all edges incident to v are contained in C (otherwise the graph would not be connected). In order to start the procedure, we take v as an arbitrary vertex of G. We trace a path starting in v and using only unused edges (unused = not contained in C) until we return to v. We incorporate P into C and continue.

More Details: How can we implement this efficiently?

An important subtask is to find unused edges. We maintain for each edge a label telling us whether the edge is used or not. Initially, all edges are unused and when an edge is added to P it becomes used. We assume that for each nodes v, the edges incident to v are contained in a linear list. We maintain an iterator $\mathit{current\_edge}[v]$ into the list with the property that all edges behind the iterator (= the iterator has scanned over them already) are used and the edges under the iterator or in front of the iterator may or may not be used. In order to find an unused edge we simply advance the iterator until we find an unused edge. If the iterator falls off the list, there are no unused edges anymore.

Consider a fixed node v and assume that we search l times for unused edges out of v. The total cost of the searches will be proportional to l plus the degree of v since we perform only one scan over the edges incident to v.

We also need to be able to find a node on our current cycle which has unused edges. We use a similar idea. We represent our cycle as a linear list C of edges and maintain an iterator *cycle_iterator* into the list. We maintain the property that only the targets of edges under the iterator or in front of the iterator have incident unused edges. In order to find a node v from which to grow a path we simply advance the iterator until we find an edge whose target has unused incident edges. We grow P and insert it into C after the iterator.

This completes the description of the algorithm. The running time is $O(n + m)$ since the number of attempts of finding unused edges is bounded by $2m$. We make at most one unsuccessful call for each node of the cycle and we make at total of m calls while extending the cycle.

A LEDA Implementation: We next give a complete implementation in LEDA. We assume that our input graph is a so-called bidirected graph, i.e., it is a directed graph in which for every edge $e = (u, v)$, the reversal $e^{rev} = (v, u)$ also belongs to G. Moreover, $G.\mathit{reversal}(e)$ gives access to the reversal of G. A bidirected graph represents an undirected graph with half the number of edges (every pair of directed edges corresponds to an undirected edge). We will construct a cycle which contains exactly one edge out of each pair of directed edges.

The procedure *Euler_Tour* takes a graph G and returns an Euler tour (as a list of edges). It starts by checking its preconditions. G must be a map, G must be connected, and all nodes must have even degrees.

We then define our local data structures: an *edge_array<bool> used* which tells which edges are used (initially, all edges are unused), an *node_array<edge> current_edge* storing the current edge of each node, and a list C of edges, the Euler tour to be constructed.

For each v, we initialize *current_edge*$[v]$ by setting it to the first edge out of v.

We initialize C by growing a cycle starting at some arbitrary node *v_0* of G. In each grow step we search for an edge leaving the cycle, trace a path starting with the edge until we return to its source node, and insert the path into C.

⟨*Euler_Tour.c*⟩≡

```
#include <LEDA/graph.h>
⟨find unused edge⟩
list<edge> Euler_Tour(const graph& G)
{ node v,v_0; edge e;
  if ( !G.is_map() ) error_handler(1,"Euler Tour: G must be a map");
  if ( !Is_Connected(G) ) error_handler(1,"Euler Tour: G must be connected");
  forall_nodes(v,G)
    if ( G.outdeg(v) % 2 != 0 )
      error_handler(1,"Euler Tour: node of odd degree");
  edge_array<bool> used(G,false);
  node_array<edge> current_edge(G);
  list<edge> C;
  forall_nodes(v,G) current_edge[v] = G.first_adj_edge(v);
  v_0 = G.first_node();
  ⟨grow a cycle starting and ending in v_0⟩
  list_item cycle_iterator = C.first();
  while ( true )
  { ⟨find an edge e leaving C, break if none (let v_0 be the source of e)⟩
    ⟨trace a path P and insert it into C⟩
  }
}
```

Before we give the details of cycle growing, path tracing and finding an unused edge leaving the cycle, we show how to find an unused edge out of a node v. We scan the adjacency list starting at *current_edge*$[v]$. When we find an unused edge, we return it. If we fall off the list, we return *nil*.

⟨*find unused edge*⟩≡

```
edge find_unused_edge(const graph& G, node v,
                      node_array<edge>& current_edge,
                      const edge_array<bool>& used)
{ edge e = current_edge[v];
  while ( e != nil && used[e] ) e = current_edge[v] = G.adj_succ(e);
  current_edge[v] = e;
  return e;
}
```

The total cost for the calls to *find_unused_edge*$(v,\ldots)$ for a fixed node v is the degree of v plus the number of calls. This follows from the fact that the cost of a call is $O(1)$ plus the number of times the iterator is advanced.

Finding a cycle through *v_0* is simple. We set a node v to *v_0* and then repeat two steps. Find an unused edge e out of v and advance v to the target node of e. Each edge of the path is added to the cycle and e and its reversal are declared used.

⟨*grow a cycle starting and ending in v_0*⟩≡

```
v = v_0;
do { e = find_unused_edge(G,v,current_edge,used);
     used[e] = used[G.reversal(e)] = true;
     C.append(e);
     v = G.target(e);
} while ( v != v_0 );
```

We next show how to find an edge leaving our current cycle. We have a *cycle_iterator* that points to an item in C. For all items behind the iterator (i.e., the iterator has visited them already), the target node of the edge under the iterator has no unused edge. If the cycle iterator has fallen off C, we are done and return C. Otherwise, we let *v_0* be the target node of the edge under the iterator. If there is an unused edge e out of *v_0* we are done. Otherwise we advance the cycle iterator.

⟨*find an edge e leaving C, break if none (let v_0 be the source of e)*⟩≡

```
while ( true )
{ if ( cycle_iterator == nil ) return C;
  v_0 = G.target(C[cycle_iterator]),
  e = find_unused_edge(G,v_0,current_edge,used);
  if ( e !=  nil ) break;
  cycle_iterator = C.succ(cycle_iterator);
}
```

When we have found an edge e leaving the cycle, say at *v_0*, we trace a path of unused edges starting with e until we reach *v_0*. The new edges are inserted into C one after the other.

⟨*trace a path P and insert it into C*⟩≡

```
list_item P_it = cycle_iterator;
v = v_0;
do { e = find_unused_edge(G,v,current_edge,used);
     used[e] = used[G.reversal(e)] = true;
     P_it = C.insert(e,P_it,LEDA::after);
     v = G.target(e);
}
while ( v != v_0 );
```

Remark: The program contains two almost identical chunks. One should be able to combine the two chunks into one. I found no way which did not obscure the logic of the program.

A Checker: The procedure *Euler_Tour* is sufficiently complex to deserve a checker. We write a function that takes a list C of edges of a bidirected graph G and checks that C is an Euler tour. We trace the cycle. As we trace the cycle we check whether consecutive edges can follow each other on a path and we count the number of occurrences of edges. If some edge does not appear at all or more than once we shout “foul”.

⟨*Euler_Tour.c*⟩+≡

```
bool Check_Euler_Tour(const graph& G, const list<edge>& C)
{ edge_array<int> number_of_occurrences(G,0);
  edge e, e_prev = C.tail(); // last edge in C
  forall(e,C)
  { number_of_occurrences[e]++; number_of_occurrences[G.reversal(e)]++;
    if ( G.target(e_prev) != G.source(e) )
      { cout << "C is not a path"; return false; }
    e_prev = e;
  }
  forall_edges(e,G)
    if ( number_of_occurrences[e] != 1 )
      { cout << "some edge either not used or used more than once";
        return false;
      }
  return true;
}
```

A Test: The most complicated part of the test is to create suitable input data. We want to have the bidirected version of connected undirected graphs in which every node has even degree.

We proceed as follows. We generate a random directed graph G with n nodes and m edges. We remove all self-loops and we make sure that the underlying undirected graph is connected. We then turn the graph into a map (= LEDA-speak for a bidirected graph with defined reversal information). Finally, we iterate over the nodes of G. Whenever we find a pair of nodes v and w of odd degrees we add the edges (v, w) and (w, v) and pair them.

⟨*Euler_Tour_test.c*⟩≡

```
⟨Euler_Tour.c⟩
int main(){
for (int n = 2; n < 100; n++)
for (int m = 0; m < 400; m++)
  {
graph G;
random_graph(G,n,m);
Delete_Loops(G);
Make_Connected(G);
list<edge> R; G.make_map(R);
node w = nil, v;
forall_nodes(v,G)
{ if ( G.outdeg(v) % 2 == 0 ) continue;
  if ( w == nil ) { w = v; continue; }
  G.set_reversal(G.new_edge(w,v),G.new_edge(v,w)); w = nil;
}
list<edge> C = Euler_Tour(G);
if ( !Check_Euler_Tour(G,C) ) G.write("test_graph");
}
return 0;
}
```

The test above takes less than a minute to run.