1. Introduction. This program was written in May 2016 by Robert Clausecker* for the *Digitale Systeme* class at Humboldt University of Berlin. The program is written in the *Literate Programming* idiom using the `CWEB` system for structured documentation of Donald E. Knuth at Stanford University. From the source file `hierholzer.w`, a C program `hierholzer.c` is created using the `CTANGLE` program. At the same time, the program can be rendered into a plain TEX document using `CWEAVE`.

Please be careful to compile the program as C99. Correct function cannot be ensured when compiled in other dialects like GNU89. On a POSIX system, the invocation (1) is suitable.

```
c99 -o hierholzer hierholzer.c
```
(1)

2. The program computes Eulerian paths in undirected graphs. A path, if it exists, is computed using the algorithm of Hierholzer. The implementation admits parallel edges.

The program's complexity is linear in the sum of the number of edges and the declared number of vertices in the input. The dependency on the number of vertices comes from the need to create the graph structure in memory. This term could be eliminated by choosing a more sophisticated data structure for the graph but it does not matter except for degenerate cases where not all vertices are connected by edges.

3. Input is provided as a list of decimal integers separated by white space. The first number is *node_count*, the number of vertices in the graph. After that, the edges of the graph are described as pairs of integers from 0 to *node_count* − 1.

If an Eulerian path can be found it is printed as a white space separated list of the path's nodes terminated with a newline. If no path was found, `"-1\n"` is printed instead.

4. The source code loosely follows the stylistic guidelines of the OpenBSD kernel, described in manual page **style**(9). For readability, the **edge**, **node** and **allocator** types are emphasized and some operators are displayed differently from how they would appear in C.

in C	in `CWEB`
`A = B, A == B, A != B`	$A \leftarrow B, A = B, A \neq B$
`A < B, A > B, A <= B, A >= B`	$A < B, A > B, A \leq B, A \geq B$
`A.B, A->B, A[B]`	$A.B, A{\rightarrow}B, A[B]$
`A && B, A || B, A & B, A | B, A ^ B`	$A \wedge B, A \vee B, A \mathbin{\&} B, A \mid B, A \oplus B$
`A << B, A >> B`	$A \ll B, A \gg B$
`A + B, A - B, A * B, A / B, A % B`	$A + B, A - B, A * B, A/B, A \mathbin{\%} B$
`+A, -A, ~A, !A, ++A, --A`	$+A, -A, \sim A, \neg A, {+}{+}A, -{-}A$
`NULL`	Λ

format *edge* *int*
format *node* *int*
format *uintptr_t* *int*
format *allocator* *int*

* Robert Clausecker <fuzxxl@gmail.com>

5. Program Structure. Apart from custom functions, only the C standard library is used. If an error occurs in one of the steps, the program is terminated prematurely. Great care has been taken to ensure that all resources are released regardless of where the program terminates.

```
#include <stdlib.h>
#include <stdint.h>
#include <stdio.h>
  ⟨ type declarations 6 ⟩
  ⟨ the nonempty_node function 18 ⟩
  extern int main(int argc, char *argv[])
  {
    FILE *input_file ← Λ;
    struct allocator alloc ← {0, BUCKET_SIZE, Λ};
    struct edge path ← {Λ, Λ};
    node *graph ← Λ;
    size_t node_count ← 0;
    int status ← EXIT_FAILURE;
    ⟨ parse the command line and open input_file 13 ⟩
    ⟨ read the graph from input_file 14 ⟩
    status ← EXIT_SUCCESS;
    ⟨ decide on a vertex to start out from 17 ⟩
    ⟨ find an Eulerian path 19 ⟩
    ⟨ check if any edges remain 23 ⟩
    ⟨ print the Eulerian path 15 ⟩
  cleanup: ⟨ release all resources 12 ⟩
    return (status);
  }
```

6. The graph is represented as an adjacency list. The list is an array of vertices, each of which has a linked list of edges. During the algorithm, edges are gradually removed from the edge lists and added to *path*. The vertices are identified through their position in the *graph* array using the *node_num* macro.

```
#define node_num(n) n − graph
```

```
⟨ type declarations 6 ⟩ =
  typedef struct edge *node;
```

See also sections 7 and 8.

This code is used in section 5.

7. Each edge is a pair of **struct edge**. Initially, *partner* points to the other node connected to the edge and *next* points to the next edge in the edge list. When an edge is removed from the graph, one half is inserted into the Eulerian cycle with *next* pointing to the next edge in the cycle. The other half is marked as "removed" by setting *partner* ← Λ. The Eulerian path is represented by a dummy-edge *path* whose *next* is the first edge and *partner* is the node the path starts out from.

A pair of **struct edge** are always allocated together. Some times, we need to find the other half of a pair of **struct edge**. To do this we use a trick: Let $s \leftarrow$ **sizeof**(**struct edge**), then s is of the form $a \cdot 2^b$ and $2^b = s \,\&\, -s$. Thus in a pair of **struct edge** one member is aligned to 2^{b+1} while the other is aligned only to 2^b. When we set things up such that the first member is always aligned to 2^{b+1}, we can find if it is the first in a pair by checking if bit b is set.

```
#define EDGE_ALIGNMENT sizeof(struct edge) & −sizeof(struct edge)
#define is_second_half(e) ((uintptr_t) e & EDGE_ALIGNMENT)
#define other_half(e) is_second_half(e) ? e − 1 : e + 1
```

```
⟨ type declarations 6 ⟩ +=
  struct edge {
    struct edge *next;
    node *partner;
  };
```

8. Memory Management. In order to quickly allocate and release many edges and to ensure the alignment constraints of **struct edge** we use a custom allocator *alloc*. The allocator allocates edges in buckets of `BUCKET_SIZE` each, keeping a table of all allocated buckets in *buckets*. The members *bucket_count* and *fill* keep track of how many buckets we have and how full the last bucket is, respectively.

```
#define BUCKET_SIZE °200000 /* 64 · 1024 entries per bucket */
⟨ type declarations 6 ⟩ +=
  struct allocator {
    size_t bucket_count, fill;
    struct edge **buckets;
  };
```

9. If the first entry in the bucket is not suitably aligned, throw it away. By construction, the second entry will.

```
⟨ allocate a new bucket 9 ⟩ =
  {
    struct edge **new_buckets ← realloc(alloc.buckets, ++alloc.bucket_count * sizeof *alloc.buckets);
    if (new_buckets ≠ Λ) alloc.buckets ← new_buckets;
    else {
      perror("allocation␣error");
      goto cleanup;
    }
    if (alloc.buckets[alloc.bucket_count − 1] ← malloc(BUCKET_SIZE * sizeof **alloc.buckets),
          alloc.buckets[alloc.bucket_count − 1] = Λ) {
      perror("allocation␣error");
      goto cleanup;
    }
    alloc.fill ← is_second_half(alloc.buckets[alloc.bucket_count − 1]) ? 1 : 0;
  }
```

This code is used in section 10.

10. The variable *epair* is assumed to be declared outside. Note that *alloc.fill* starts out at `BUCKET_SIZE` to do the right thing on the first edge we allocate.

```
⟨ allocate a pair of struct edge for epair 10 ⟩ =
  if (alloc.fill > BUCKET_SIZE − 2) ⟨ allocate a new bucket 9 ⟩
  epair ← alloc.buckets[alloc.bucket_count − 1] + alloc.fill;
  alloc.fill += 2;
```

This code is used in section 16.

11. Walk *alloc.buckets*, then free *alloc.buckets*.

```
⟨ release all edges 11 ⟩ =
  {
    size_t i;
    for (i ← 0; i < alloc.bucket_count; i++) free(alloc.buckets[i]);
    free(alloc.buckets);
  }
```

This code is used in section 12.

12. A label *cleanup* is placed just before this section so we can go there to end the program early.

```
⟨ release all resources 12 ⟩ =
  if (input_file ≠ Λ) fclose(input_file);
  ⟨ release all edges 11 ⟩
  free(graph);
```

This code is used in section 5.

13. Input and Output. The following sections implement input parsing and formatted output. Before input can be parsed, the input file must be opened. If no input file was provided a helpful message is printed instead.

```
⟨ parse the command line and open input_file 13 ⟩ =
  if (argc ≠ 2) {
    fprintf(stderr, "usage: %s file\n", argv[0]);
    goto cleanup;
  }
  if (input_file ← fopen(argv[1], "r"), input_file = Λ) {
    perror("cannot open input file");
    goto cleanup;
  }
```

This code is used in section 5.

14. We read the input as described earlier and construct the graph in memory. The number of nodes in the graph is stored in *node_count*.

```
⟨ read the graph from input_file 14 ⟩ =
  {
    size_t from, to;
    int items;
    if (fscanf(input_file, "%zu\n", &node_count) ≠ 1) {
      if (ferror(input_file)) perror("input error");
      else fprintf(stderr, "input malformed\n");
      goto cleanup;
    }
    if (graph ← calloc((size_t) node_count, sizeof *graph), graph = Λ) {
      perror("allocation error");
      goto cleanup;
    }
    while (items ← fscanf(input_file, "%zu %zu\n", &from, &to), items = 2) {
      if (from ≥ node_count ∨ to ≥ node_count) {
        fprintf(stderr, "input malformed\n");
        goto cleanup;
      }
      ⟨ insert an edge between from and to 16 ⟩
    }
    if (items ≠ EOF) {      /* items ∈ {0, 1} if fscanf did not parse both from and to */
      fprintf(stderr, "input malformed\n");
      goto cleanup;
    }
    if (ferror(input_file)) {
      perror("input error");
      goto cleanup;
    }
  }
```

This code is used in section 5.

15. To output the path we follow the *next* pointers of *path* and print the vertices we encounter.

```
⟨ print the Eulerian path 15 ⟩ =
  {
    struct edge *edg;
    printf("%tu", node_num(path.partner));
    for (edg ← path.next; edg ≠ Λ; edg ← edg→next) printf(" %tu", node_num(edg→partner));
    putchar('\n');
  }
```

This code is used in section 5.

16. Graph Algorithms. Both parts of the edges are hooked into the beginnings of their edge lists.

```
⟨ insert an edge between from and to 16 ⟩ =
  {
    struct edge *epair;
    ⟨ allocate a pair of struct edge for epair 10 ⟩
    epair[0].next ← graph[from];
    epair[0].partner ← graph + to;
    graph[from] ← epair + 0;
    epair[1].next ← graph[to];
    epair[1].partner ← graph + from;
    graph[to] ← epair + 1;
  }
```

This code is used in section 14.

17. Only graphs with at most two vertices of odd degree can have Eulerian paths. The converse is not true: A disconnected graph might not have vertices of odd degree but it cannot have an Eulerian path.

This snippet finds out how many nodes of odd degree exist, assigns *path.partner* to one of the two odd nodes if exactly two nodes have odd degree or prints `"-1\n"` for "no Eulerian path" if more than two nodes have odd degree. If no nodes of odd degree are found, a random node is picked for *path.partner*. If no node exists, the path is empty and the program terminates prematurely.

```
⟨ decide on a vertex to start out from 17 ⟩ =
  {
    struct edge *edg;
    size_t i, degree, odd_count ← 0;
    for (i ← 0; i < node_count; i++) {
      for (degree ← 0, edg ← graph[i]; edg ≠ Λ; edg ← edg→next) degree++;
      if (degree % 2 ≠ 0) {
        if (++odd_count ≤ 2) path.partner ← graph + i;
        else {
          printf("-1\n");
          goto cleanup;
        }
      }
    }
    if (path.partner = Λ) path.partner ← nonempty_node(graph, node_count);
    if (path.partner = Λ) {
      printf("\n");
      goto cleanup;
    }
  }
```

This code is used in section 5.

18. Find a node with at least one edge, return Λ if none exists.

```
⟨ the nonempty_node function 18 ⟩ =
  static node *nonempty_node(node *graph, size_t node_count)
  {
    size_t i;
    for (i ← 0; i < node_count; i++)
      if (graph[i] ≠ Λ) return (graph + i);
    return (Λ);
  }
```

This code is used in section 5.

19. This is the rough structure of Hierholzer's algorithm. The algorithm alternately expands the current path by inserting a new subcycle between *chaser* and *chaser→next* and "chases" the path towards its end by advancing *chaser* until a node with edges remaining or the end is reached. When *chaser* reaches the end of the path the algorithm is complete.

```
⟨ find an Eulerian path 19 ⟩ =
  {
    struct edge *chaser ← &path;
    do {
      ⟨ greedily remove a path from chaser, at its end, follow it up with chaser→next 20 ⟩
      ⟨ move chaser along the path until a vertex with edge or Λ is reached 22 ⟩
    } while (chaser ≠ Λ);
  }
```

This code is used in section 5.

20. In this snippet a new path is assembled beginning from *chaser→partner* until it gets stuck somewhere. This can only happen if we are back were we came from or when this is the first path we add in which case *chaser_next* is Λ.

```
⟨ greedily remove a path from chaser, at its end, follow it up with chaser→next 20 ⟩ =
  {
    struct edge *leader, *chaser_next ← chaser→next;
    for (leader ← chaser; *leader→partner ≠ Λ; leader ← leader→next) {
      leader→next ← *leader→partner;
      ⟨ remove leader→next from the graph 21 ⟩
    }
    leader→next ← chaser_next;
  }
```

This code is used in section 19.

21. Unhook *leader→next* from its edge list and mark *other_half*(*leader→next*) as removed by setting *partner* ← Λ. No memory is deallocated. This function assumes that *leader→next* is the first edge in its edge list which always holds when this snippet runs. The *normalize* macro advances *n*'s edge list to skip removed edges so that *n* = Λ when the node is empty or points to an actual edge.

```
#define normalize(n) while (n ≠ Λ ∧ n→partner = Λ) n ← n→next
⟨ remove leader→next from the graph 21 ⟩ =
  {
    struct edge *back ← other_half(leader→next);
    *back→partner ← leader→next→next;
    normalize((*back→partner));
    back→partner ← Λ;
    normalize((*leader→next→partner));
  }
```

This code is used in section 20.

22. If we reached Λ and have not seen a vertex with at least one edge, the path is complete.

```
⟨ move chaser along the path until a vertex with edge or Λ is reached 22 ⟩ =
  while (chaser ≠ Λ ∧ *chaser→partner = Λ) chaser ← chaser→next;
```

This code is used in section 19.

23. Even if all vertices have even degree the node may not have an Eulerian path if it is not connected. In this case an Eulerian path is seemingly found but some edges remain in the graph.

```
⟨ check if any edges remain 23 ⟩ =
  if (nonempty_node(graph, node_count) ≠ Λ) {
    printf("-1\n");
    goto cleanup;
  }
```

This code is used in section 5.

24. References and Index.

- Carl Hierholzer, *Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren,* in *Mathematische Annalen,* vol. VI., no. 1, p. 30–32, March 1873.
- The OpenBSD Kernel Developer's Manual, *Kernel source file style guide,* **style**(9).
- ISO/IEC JTC 2/SC 22: ISO/IEC 9899:2011, *"Programming Languages – C"*, The International Organization for Standardization.

25. This index contains all identifiers as well as some other keywords. The numbers refer to sections, not pages. The section with the definition of an identifier is underlined.

⟨ allocate a new bucket 9 ⟩ Used in section 10.
⟨ allocate a pair of **struct edge** for *epair* 10 ⟩ Used in section 16.
⟨ check if any edges remain 23 ⟩ Used in section 5.
⟨ decide on a vertex to start out from 17 ⟩ Used in section 5.
⟨ find an Eulerian path 19 ⟩ Used in section 5.
⟨ greedily remove a path from *chaser*, at its end, follow it up with *chaser*→*next* 20 ⟩ Used in section 19.
⟨ insert an edge between *from* and *to* 16 ⟩ Used in section 14.
⟨ move *chaser* along the path until a vertex with edge or Λ is reached 22 ⟩ Used in section 19.
⟨ parse the command line and open *input_file* 13 ⟩ Used in section 5.
⟨ print the Eulerian path 15 ⟩ Used in section 5.
⟨ read the graph from *input_file* 14 ⟩ Used in section 5.
⟨ release all edges 11 ⟩ Used in section 12.
⟨ release all resources 12 ⟩ Used in section 5.
⟨ remove *leader*→*next* from the graph 21 ⟩ Used in section 20.
⟨ the *nonempty_node* function 18 ⟩ Used in section 5.
⟨ type declarations 6, 7, 8 ⟩ Used in section 5.

The Algorithm of Hierholzer