1. Knuth Pre-Fascicle 8A, Exercise 65.

A search for the smallest Hamiltonian graph on which Warnsdorf's Algorithm W always fails.

Shisheng Li (李世胜) ⟨daizisheng@gmail.com⟩

April 2026

2. Introduction. *The Art of Computer Programming*, Volume 4, Pre-Fascicle 8A, Exercise 65 [rating 25] asks:

> *Is there a Hamiltonian graph for which Algorithm W always fails to find a Hamiltonian path, regardless of the starting point and the ordering of arcs?*

Knuth's answer is *yes*: he exhibits a specific 14-vertex graph with 22 edges. The parenthetical question "(Is there a smaller one?)" in the published answer is then flagged in the fascicle's preface as one of several implicitly posed open questions.

Our answer to Knuth's parenthetical: *no* — **Knuth's 14-vertex, 22-edge graph is minimal in vertex count, and within $n = 14$ it is also uniquely minimal in edge count.**

More precisely, we prove by exhaustive search that

> Every Hamiltonian graph on which Algorithm W always fails has $n \geq 14$.

This is tight: Knuth's graph realises $n = 14$. We further enumerate *all* Algorithm-W counter-examples at $n = 14$ with max-degree ≤ 5, obtaining exactly seven non-isomorphic graphs (Knuth's plus six new ones).

The proof combines exhaustive enumeration (via *nauty* − *geng*) with an exact filter that branches on every tied-minimum candidate in Warnsdorf's rule, guaranteeing that the reported verdict "W always fails" holds simultaneously over all arc-orderings and all start vertices.

3. Research framework: our filter + nauty-geng + parallel slicing. Our work decomposes cleanly into three pieces:

A. A small decision procedure (about 150 lines of C, Sections §11–§15 below). Given an adjacency-bitmask representation of a graph G, the two-stage filter *W_always_fails* followed by *has_ham_cycle* decides whether G is a counter-example. Typical cost is $O(n)$ per graph (W succeeds quickly from vertex 0, the graph is rejected); worst-case cost, exponential in n, arises only on the tiny minority of graphs that survive both filters and trigger full DFS branching. Algorithm W is modelled *exactly*: all tied-minimum candidates are branched upon, so the verdict "W always fails" holds simultaneously for every start vertex and every arc-ordering.

B. nauty-geng for exhaustive, isomorph-free enumeration (see below). We don't roll our own graph generator; instead we compile-in McKay and Piperno's `geng` via its `OUTPROC` / `GENG_MAIN` hooks. Every connected simple graph on n vertices with minimum degree ≥ 2 (and, when set, maximum degree $\leq D$) is streamed into our decision procedure exactly once, in a single process with no disk I/O.

C. Embarrassingly-parallel slicing via `geng`'s `res/mod` flag. `geng` accepts a final positional argument of the form r/m that restricts output to the r-th of m disjoint residue classes of its canonical-construction tree. We launch one copy of the binary per CPU thread, each covering its assigned slice; the experiments below used $m = 128$, 256, or 384 depending on the machine. No coordination is needed, and no graphs are duplicated or lost. The "CPU-time" column of the Summary-of-findings table is the *sum* of per-slice wall times across all slices; because `geng`'s residue classes are not perfectly balanced, this sum typically exceeds the wall-clock elapsed time by the usual tail-latency gap (see the Hardware section below for actual min/max/mean per sweep).

The upshot is that our original contribution, the counter-example decision procedure, is tiny; nauty-geng carries the combinatorial load; the operating system carries the parallelism. This is what makes the largest sweep reported below—all 4.8×10^{13} connected min-deg-2 simple graphs on 13 vertices— feasible on commodity cloud hardware.

4. About nauty and `geng`. *Nauty* and its companion graph-enumeration utility `geng`, both by Brendan D. McKay (Australian National University) and Adolfo Piperno (Sapienza University of Rome), have been the de-facto standard tools for graph isomorphism and canonical labeling since 1981; for our purposes they are *the* reason an exhaustive $n = 14$ sweep is feasible at all.

Why `geng`. `geng` generates all pairwise non-isomorphic simple graphs on n vertices satisfying optional filters (`-c` connected, `-d#` minimum degree, `-D#` maximum degree, `-e␣p:q` edge-count range, `-C` 2-connected, `-b` bipartite, etc.). Two properties are essential for us:

(i) **Exactly one representative per isomorphism class.** `geng` enumerates "canonical constructions" (see the reference below): it extends partial graphs one vertex at a time, and at each step keeps only those extensions where the just-added vertex is in the canonical-form vertex equivalence class of the new graph. Every isomorphism class is therefore emitted *exactly once*, with no post-hoc deduplication (McKay, "Isomorph-free exhaustive generation," *J. Algorithms* **26** (1998), 306–324).

(ii) **Very small memory footprint.** The generation proceeds depth-first through a tree of partial graphs, so only an $O(n^2)$ scratch graph (plus canonical-labeling state) is in memory at any given moment, regardless of how many billions of graphs are emitted.

Together these mean we can hand the $\sim$154 billion connected min-degree-2 simple graphs on 12 vertices—and indeed the $\sim$48 trillion on 13 vertices—to our filter one at a time through a single in-process callback, with no disk I/O whatsoever.

Trustworthiness. Nauty is over 40 years mature (first released in 1981), used in essentially every combinatorial-enumeration paper that needs isomorphism, and its correctness has been cross-checked against the published sequences (OEIS A001349, A001350, A006125, ...) for every n in range. The present program's results accordingly inherit those guarantees: we do not reimplement graph enumeration or isomorphism, only compose our filter on top.

5. `geng`'s plugin interface. By defining the preprocessor symbols `OUTPROC` and `GENG_MAIN` before including `geng.c`, we tell geng to (1) call our *w_outproc* instead of writing to stdout, and (2) rename its own `main` to *geng_main* so we can call it as a function from our own *main*. This turns the pair *(nauty-geng, w_outproc)* into a single executable — no pipe, no I/O serialization.

Internal graph representation. Each invocation of our *w_outproc* receives nauty's internal *graph* $*g$, an array of *setword* (typically **unsigned long**, 64-bit). Bit at position WORDSIZE $-1-i$ of $g[v]$ is 1 iff $v-i$ is an edge — i.e. nauty packs adjacency rows MSB-first. We convert to our low-bit-first *mask_t* representation with a 4-step bit-reverse in the output handler below.

Graph6 output for hits. When a counter-example is found we emit its `graph6` encoding (a base-63 ASCII format also due to McKay), via nauty's *ntog6*( ). The raw graph6 strings are canonical under nauty's labelling and therefore serve as unambiguous identifiers of each hit, independent of any drawing we choose. The seven $n = 14$ hits are:

```
#1 (m=22):  M?AACGoIBCR?dOOe?
#2 (m=23):  M?AACGoIBCR?dOWe?
#3 (m=24):  M?AACGoIBCR?WibP?
#4 (m=24):  M?AACGoICoGSbQWh?
#5 (m=24):  M?AACGoICoQgWcWc_
#6 (m=25):  M?AACGoICoKSbQWh?
#7 (m=26):  M?AA@_gEDIHQhQQd?
```

Important: vertex labels in the diagrams differ from those implicit in the graph6 strings. Each graph6 string above encodes nauty's *canonical* labelling, which is determined by graph-isomorphism invariants and has no special relationship to any particular Hamilton cycle. For the diagrams in Section §7 below we chose, for each graph, an independent relabelling that places a selected Hamilton cycle on the outer circle $1{-}2{-}\cdots{-}14{-}1$. Both labellings describe the same underlying (isomorphism class of) graph, but the integers $1, 2, \ldots, 14$ refer to different vertices in the two conventions.

6. Summary of findings. The search covers all connected simple graphs with minimum degree ≥ 2 (a necessary condition for a Hamiltonian cycle), up to isomorphism, as enumerated by `nauty-geng`'s canonical construction. For each graph we run the exact "is W stuck from every start?" decision procedure *W_always_fails* (Section §13 below), and only on a positive answer do we verify the graph is Hamiltonian via *has_ham_cycle*.

Ruled-out range. $n \leq 13$: exhaustive, 0 counter-examples. For each n we enumerate all connected simple graphs with minimum degree ≥ 2 (the necessary conditions for a Hamiltonian cycle), up to isomorphism. At $n = 1$ the minimum-degree-2 constraint is vacuously unsatisfiable (a single vertex has degree 0); at $n = 2$ the only connected graph is an edge, whose two endpoints have degree 1. Hence both rows show 0 graphs.

n	**max-deg ($-D$)**	**# graphs scanned**	**# W fails**	**# hits**	**CPU-time**
1	not specified	0	0	0	—
2	not specified	0	0	0	—
3	not specified	1	0	0	< 0.01 s
4	not specified	3	0	0	< 0.01 s
5	not specified	11	0	0	< 0.01 s
6	not specified	61	2	0	< 0.01 s
7	not specified	507	11	0	< 0.01 s
8	not specified	7,442	143	0	< 0.01 s
9	not specified	197,772	1,808	0	0.12 s
10	not specified	9,808,209	40,819	0	5.3 s
11	not specified	902,884,343	1,200,537	0	458 s
12	not specified	153,723,152,913	58,562,666	0	38.4 h
13	not specified	48,443,147,912,137	3,994,121,519	0	7,707 h
14	5	251,277,392,850	192,193,330	**7**	178.4 h

Hardware. Three CPU models were used:

- Rows $n \leq 11$: Intel Xeon E5-2686 v4 at 2.30 GHz, single-threaded. Wall time equals CPU time.
- Rows $n = 12$ and $n = 14$, $-D5$: dual AMD EPYC 7742 (Zen 2, 2.25 GHz base, 64 cores × 2 sockets × SMT-2 = 256 threads). The "CPU-time" entry is the sum of per-slice wall times across all 256 slices. Individual slices differ noticeably — geng's canonical-construction tree is not perfectly balanced, so some residue classes contain more graphs, or harder graphs, than others. For $n = 12$, per-slice walls ranged from 433 s to 645 s (mean 540 s), wall-clock elapsed 645 s, sum 138,148 s = 38.4 h. For $n = 14$, $-D5$, per-slice walls ranged from 1,347 s to 4,382 s (mean 2,508 s), wall-clock elapsed 4,382 s, sum 642,141 s = 178.4 h.
- Row $n = 13$: dual AMD EPYC 9K84 (Zen 4, 3.70 GHz, 96 cores × 2 sockets × SMT-2 = 384 threads, 192 physical cores). Per-slice walls ranged from 48,747 s to 87,054 s (mean 72,252 s); wall-clock elapsed 24.2 h, sum of per-slice walls 27,744,593 s = 7,707 h ≈ 321 CPU-days. As a cross-check we re-ran $n = 14$, $-D5$ on this CPU and reproduced the identical graph-count, stage-1 count, and 7-HIT set in 28.2 min wall (vs. 73 min on the 7742).

Edge-count completeness at $n = 14$. To rule out the possibility that a counter-example with max-degree ≥ 6 might be hiding in the sparse-edge range $m \leq 21$ (below Knuth's 22), and to confirm that hits #1–#5 really exhaust the counter-examples at $m \in \{22, 23, 24\}$ regardless of max-degree, we re-ran $n = 14$ *without any max-degree restriction*, banded by exact edge count. Each row below is an independent full sweep over every connected simple graph on 14 vertices with min-deg ≥ 2 and the given m.

m	**# graphs scanned**	**# W fails**	**# hits**
≤ 21	242,053,705	48,271,577	0
22	1,015,006,822	144,790,281	1 (Knuth #1)
23	4,411,400,282	458,097,928	1 (#2)
24	16,944,249,400	1,239,929,795	3 (#3, #4, #5)

All five hits coincide with the corresponding counter-examples from the $-D5$ sweep, so the $-D5$ restriction *did not miss* anything at $m \le 24$. In particular **Knuth's 22-edge graph is the unique 14-vertex counter-example with $m \le 22$.**

Conclusion. For every $n \le 13$, exhaustive enumeration of all connected simple graphs with minimum degree ≥ 2 yields *zero* Algorithm-W counter-examples. This proves

Every Hamiltonian graph on which Algorithm W always fails has $n \ge 14$.

Knuth's 14-vertex, 22-edge graph (#1 in Section §7) is therefore minimal in vertex count, and within $n = 14$ it is additionally uniquely minimal in edge count. At $n = 14$ with max-degree ≤ 5 there are exactly 7 non-isomorphic counter-examples. The parenthetical "(Is there a smaller one?)" in the published answer to Exercise 65 is therefore answered: **no**.

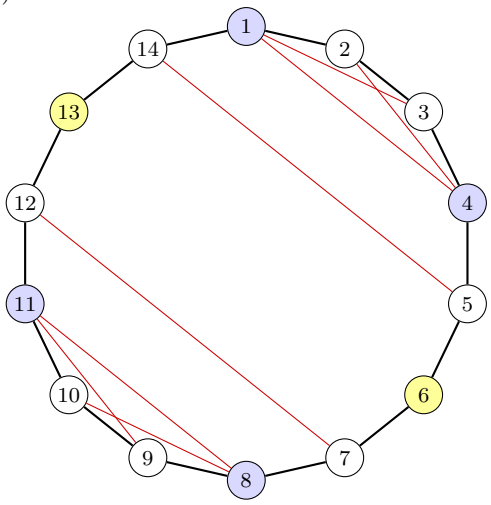
7. The seven $n = 14$ counter-examples (max-deg ≤ 5). All seven are connected, Hamiltonian, and defeat Algorithm W from every start under every arc-ordering. Nauty emits them as distinct isomorphism classes. Knuth's original is #1; the remaining six are new.

The $-D5$ sweep enumerates *all* $n = 14$ counter-examples with max-degree ≤ 5, producing exactly seven. The edge-banded no-D sweeps in Section §6 additionally certify that for $m \in \{22, 23, 24\}$ there are no counter-examples at any max-degree beyond those found at $-D5$, so hits #1–#5 are the only $n = 14$ counter-examples at those edge counts. Whether hits #6 ($m = 25$) and #7 ($m = 26$) remain the only ones when max-degree ≥ 6 is also allowed is not yet settled.

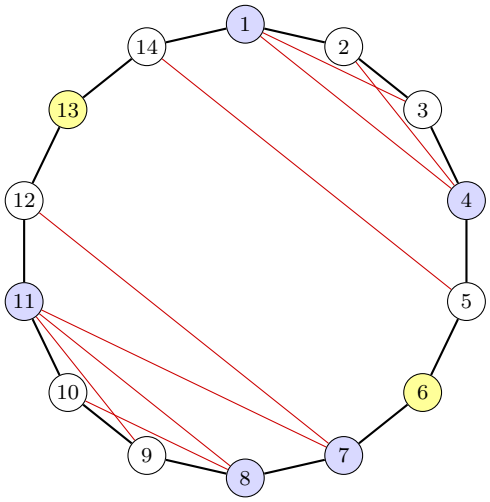
Identification of #1 with Knuth's graph. Knuth's $n = 14$ example in the published answer has $m = 22$ edges and maximum degree 4. Our $-D4$ sweep (#graphs scanned = 388,337,048; see Section §6, subsumed into the $-D5$ row of the main table) found exactly one counter-example, and the $m = 22$ row of the edge-banded table shows that graph to be the unique counter-example with $m = 22$ at *any* max-degree. Since Knuth's graph is a counter-example with $m = 22$, it must coincide with our #1 ($g6 =$ `M?AACGoIBCR?dOOe?`).

Each graph is drawn below with its 14 vertices on a circle, labeled 1–14 (1-indexed). **Black edges** belong to a chosen Hamiltonian cycle; **red edges** are the non-cycle (chord) edges. Degree-2 vertices are shaded yellow, degree-≥ 4 vertices shaded light blue, for quick visual orientation.

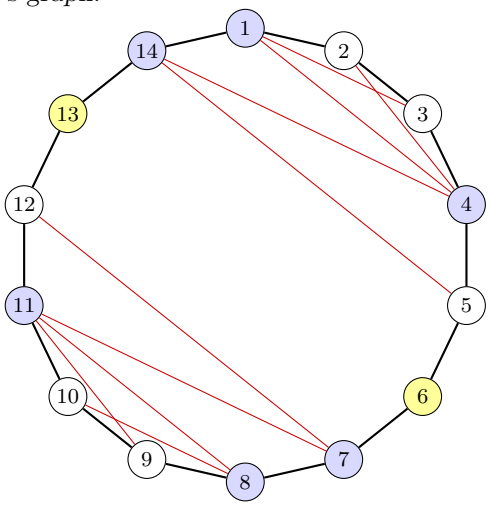
#1: $m = 22$ — Knuth's original, max-deg 4
deg seq $= [2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4]$
Two K_4 lobes $\{1, 2, 3, 4\}$ and $\{8, 9, 10, 11\}$ connected by two length-3 bridges $5{-}6{-}7$ and $12{-}13{-}14$, with crossing chords $(5, 14)$ and $(7, 12)$.



#2: $m = 23$ — **Knuth** + $(7, 11)$
deg seq = $[2, 2, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5]$
Adding the chord $(7, 11)$ *to Knuth's graph lifts vertex 11 from degree 4 to degree 5; all other degrees are unchanged.*

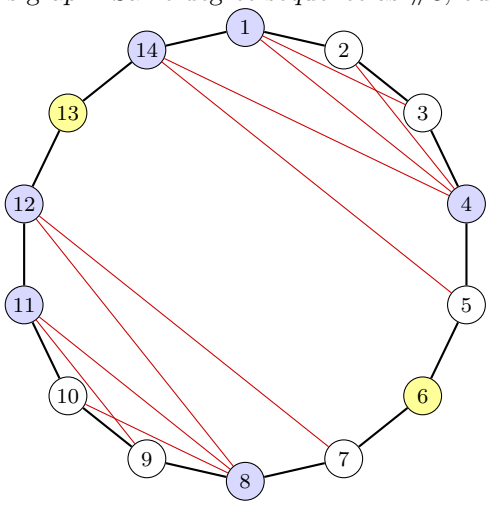


#3: $m = 24$ — **Knuth** + $(7, 11) + (4, 14)$
deg seq = $[2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5]$
Both chords are added to Knuth's graph.

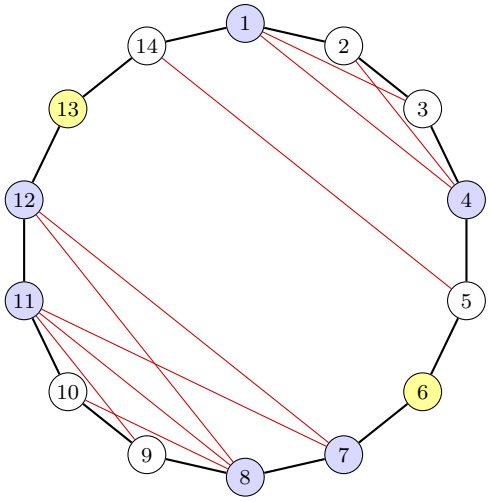


#4: $m = 24$ — **Knuth** + $(4, 14) + (8, 12)$
deg seq = $[2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5]$

Both chords are added to Knuth's graph. Same degree sequence as #3, but not isomorphic to it.

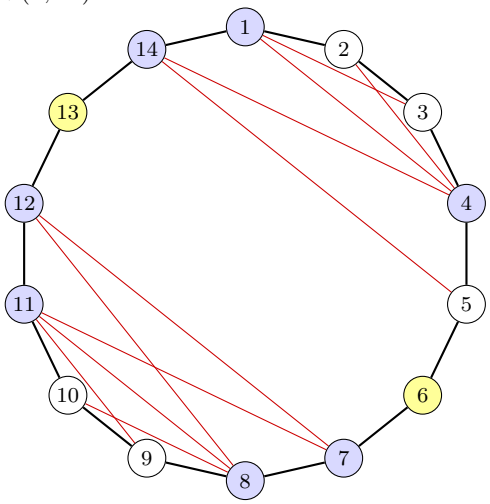


#5: $m = 24$ — **Knuth** + $(7,11) + (8,12)$
deg seq $= [2,2,3,3,3,3,3,3,4,4,4,4,5,5]$
Both chords are added to Knuth's graph. Same degree sequence as #3 and #4; all three are pairwise non-isomorphic.

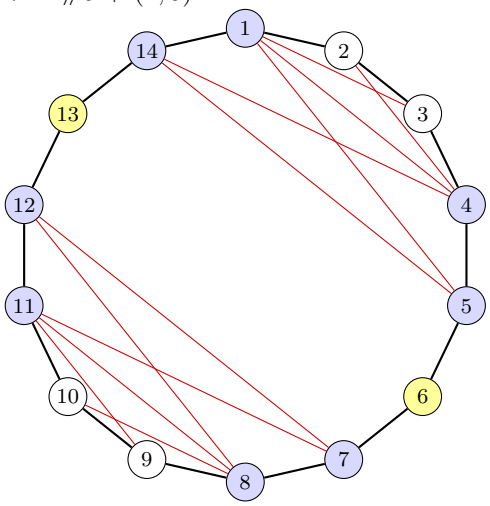


#6: $m = 25$ — **Knuth** + $(7,11) + (4,14) + (8,12)$
deg seq $= [2,2,3,3,3,3,3,4,4,4,4,5,5,5]$

All three of the chords appearing singly or pairwise in #2–#5 are added to Knuth's graph at once; equivalently #3 $+(8,12)$*, or #5* $+(4,14)$.



#7: $m = 26$ — **Knuth** $+ (1,5) + (4,14) + (7,11) + (8,12)$
deg seq $= [2,2,3,3,3,3,4,4,4,4,5,5,5,5]$
Knuth's 22 edges plus four chords: the same three $\{(7,11),(4,14),(8,12)\}$ *that generate #2–#6, plus one new chord* $(1,5)$*. Equivalently, #7 = #6 +* $(1,5)$.



8. Program structure. This single-file `CWEB` program pulls together a hand-coded 150 lines of C, glued to `nauty`'s graph-generation engine via the `OUTPROC` / `GENG_MAIN` hooks that `nauty-geng` provides. The build invocation is documented in Section §16.

⟨ Header includes 9 ⟩
⟨ Constants and typedefs 10 ⟩
⟨ Ham-cycle DFS 11 ⟩
⟨ Warnsdorf W with tie-branching 12 ⟩
⟨ Top-level filter *W_always_fails* 13 ⟩
⟨ Nauty `OUTPROC`: receive each graph and decide 14 ⟩
⟨ Main: set up geng invocation 15 ⟩

9. The program limits vertex count to 16 so that each vertex's adjacency list fits in a *uint16_t*. Knuth's question concerns $n = 14$, well within bounds.

⟨ Header includes 9 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <time.h>
#include "gtools.h"      /* nauty's in-tree I/O + graph types */
```
This code is used in section 8.

10. ⟨ Constants and typedefs 10 ⟩ ≡
```
#define MY_MAXN 16
  typedef uint16_t mask_t;      /* low-bit-first adjacency bitmask */      /* The bit-reversal in w_outproc
      below assumes 64-bit setwords; fail the build loudly if that ever stops holding. */
  _Static_assert(WORDSIZE ≡ 64, "this␣program␣requires␣nauty␣WORDSIZE=64");
```
This code is used in section 8.

11. Hamiltonian-cycle check (DFS). We verify a graph has a Hamiltonian cycle by a plain depth-first search rooted at vertex 0, closing back to 0 when all n vertices are visited. Early exit on the first cycle found keeps this fast in the typical case: a graph that *has* a Hamilton cycle can usually be accepted after a shallow DFS, whereas a graph that doesn't must exhaust the search tree and is substantially slower.

```
⟨ Ham-cycle DFS 11 ⟩ ≡
    /* Two GCC intrinsics used throughout this file: __builtin_ctz(x) = count
       trailing zeros = index of lowest set bit. __builtin_popcount(x) = number of 1-bits (POPCNT
       instruction on x86). The idiom "x &= x − 1" clears the lowest set bit of x (because x − 1 flips all
       trailing zeros to ones and the lowest 1 to 0, so AND leaves the higher bits intact). Combined with
       ctz, this is the standard "iterate over set bits low-to-high" loop. */
  static int has_ham_cycle_recur(const mask_t adj[], int n, int v, mask_t visited, int depth)
  {
    if (depth ≡ n) return (adj[v] & 1U) ≠ 0;      /* can we close back to vertex 0? */
    mask_t cands = adj[v] & ∼visited;      /* unvisited neighbors of v */
    while (cands) {
      int u = __builtin_ctz(cands);      /* lowest-index candidate */
      cands &= cands − 1;      /* clear the lowest set bit */
      if (has_ham_cycle_recur(adj, n, u, visited | (mask_t)(1U ≪ u), depth + 1)) return 1;
    }
    return 0;
  }
  static int has_ham_cycle(const mask_t adj[], int n)
  {
    if (n < 3) return 0;
    return has_ham_cycle_recur(adj, n, 0, 1, 1);
  }
```

This code is used in section 8.

12. Warnsdorf's rule (exact, branching on tied minima). Algorithm W in Knuth's §7.2.2.4 is a deterministic greedy heuristic: at each step it moves to the unvisited neighbor of minimum active degree (step W5), breaking ties by the arc-ordering in ARCS(v_k), and *terminates without backtracking* as soon as the current vertex has no unvisited neighbors (step W7). The active degree of an unvisited vertex u is the number of u's neighbors not yet in the path, i.e., $|\mathrm{adj}(u) \cap \overline{\mathrm{visited}}|$; we compute it via `POPCNT` on the fly instead of maintaining a `DEG[]` array, which saves the per-frame copy that an explicit `DEG[]` parameter would need.

```
⟨ Warnsdorf W with tie-branching 12 ⟩ ≡
  static int W_recurse(const mask_t adj[], int n, int v, mask_t visited)
  {   /* Success: every vertex is on the path, so we have a Hamilton path. */
    if (__builtin_popcount(visited) ≡ n) return 1;    /* popcount = number of 1-bits */
    mask_t unv = adj[v] & ~visited;    /* unvisited neighbors of v */
    if (¬unv) return 0;    /* dead end: nowhere to go */    /* First pass: find the minimum
          active-degree among the candidates. cand_deg[] is left uninitialised — we only ever write index
          u for u in unv here, and only ever read index u for u in unv in the second pass below, so reading
          uninitialised slots is unreachable. */
    int min_deg = MY_MAXN + 1;    /* +∞ sentinel */
    int cand_deg[MY_MAXN];
    mask_t bm = unv;
    while (bm) {
      int u = __builtin_ctz(bm);    /* lowest set bit of bm */
      bm &= bm − 1;    /* clear that bit */
      int d = __builtin_popcount(adj[u] & ~visited);
      cand_deg[u] = d;
      if (d < min_deg) min_deg = d;
    }    /* Second pass: recurse on every tied-minimum candidate, in bit (= arc) order. Any one success
          means some arc-ordering lets W succeed from v, so we return 1. */
    bm = unv;
    while (bm) {
      int u = __builtin_ctz(bm);
      bm &= bm − 1;
      if (cand_deg[u] ≡ min_deg) {
        if (W_recurse(adj, n, u, visited | (mask_t)(1U ≪ u))) return 1;
      }
    }
    return 0;
  }
```

This code is used in section 8.

13. By the Corollary of Section §12, the filter below (which invokes *W_recurse* from every start vertex and returns true only if all of them fail) decides exactly the "W always fails" property required by Exercise 65.

```
⟨ Top-level filter W_always_fails 13 ⟩ ≡
  static int W_always_fails(const mask_t adj[], int n)
  {
    for (int s = 0; s < n; s++) {
      if (W_recurse(adj, n, s, (mask_t)(1U ≪ s))) return 0;
    }
    return 1;
  }
```

This code is used in section 8.

14. Output handler. Nauty calls *w_outproc* for every canonical graph it emits. We translate the adjacency matrix, filter with the two-stage pipeline (Warnsdorf first, Hamilton-cycle verification second), and emit a `HIT` line when we find a counter-example.

The order is deliberately *W-first*. Warnsdorf's rule is cheap on almost every input: for the vast majority of graphs it quickly finds a completion from $s = 0$ alone and the graph is rejected immediately. The Hamilton-cycle check, by contrast, is fast on graphs that *have* a Hamilton cycle but noticeably slower on graphs that don't, since it must exhaust the search tree. Running W first thus filters out the overwhelming majority of inputs cheaply, so that only a tiny fraction of graphs ever incur the Hamilton-cycle check — making the combined pipeline much faster than running either test alone.

Bit-order conversion. Nauty represents each vertex's adjacency row in a 64-bit *setword*, packed *MSB-first*: bit WORDSIZE $-1-i$ of $g[v]$ is 1 iff $v-i$ is an edge. Our filter works with a 16-bit *mask_t* packed *LSB-first*: bit i of $adj[v]$ is 1 iff $v-i$ is an edge. The conversion below is a right-shift-then-reverse: first we right-shift $g[v]$ by WORDSIZE $-16 = 48$ bits to place the 16 relevant (MSB-first) bits into the low half of a **uint16_t**; then we reverse those 16 bits. The reversal is the standard divide-and-conquer ladder — swap adjacent bits, then adjacent pairs, then nibbles, then bytes — which is four ALU instructions on any modern CPU.

```
⟨ Nauty OUTPROC: receive each graph and decide 14 ⟩ ≡
  static int TARGET_N;
  static const char *TAG = "";
  static long PROGRESS_EVERY = 0;
  static long g_count = 0, g_stage1 = 0, g_hits = 0, g_next_report = 0;
  static struct timespec g_ts0;
  void w_outproc(FILE *outfile, graph * g, int n)
  {
    (void) outfile;
    if (n ≠ TARGET_N) return;      /* Convert nauty's MSB-first 64-bit setword rows to our LSB-first
          16-bit mask_t. See the prose above for the layout contract. */
    mask_t adj[MY_MAXN];
    int shift = WORDSIZE − 16;      /* = 48 when WORDSIZE = 64 */
    for (int v = 0; v < n; v++) {
      uint16_t x = (uint16_t)((unsigned long) g[v] ≫ shift);
        /* Now x has the 16 relevant bits of g[v], still MSB-first: bit 15 of x encodes edge v-0, bit 14
          encodes v-1, ..., bit 0 encodes v-15. Reverse the 16 bits in 4 divide-and-conquer steps. */
      x = ((x & #aaaa) ≫ 1) | ((x & #5555) ≪ 1);      /* swap adjacent bits */
      x = ((x & #cccc) ≫ 2) | ((x & #3333) ≪ 2);      /* swap adjacent pairs */
      x = ((x & #f0f0) ≫ 4) | ((x & #0f0f) ≪ 4);      /* swap adjacent nibbles */
      x = (uint16_t)((x ≫ 8) | (x ≪ 8));      /* swap the two bytes */
      adj[v] = (mask_t)x;      /* bit i = 1 iff edge v-i */
    }
    g_count++;      /* Stage 1: does W fail from every start under every arc-ordering? */
    if (¬W_always_fails(adj, n)) return;
    g_stage1++;      /* Stage 2: if so, confirm the graph actually has a Ham cycle. */
    if (has_ham_cycle(adj, n)) {
      int m = 0;
      for (int v = 0; v < n; v++) m += __builtin_popcount(adj[v]);
      m /= 2;
      g_hits++;      /* ntog6( ) returns a pointer into nauty's internal static buffer; the string already ends
          with a newline, so we put it last in the format and don't add another one. */
      char *g6 = ntog6(g, 1, n);
      printf("HIT␣%s␣n=%d␣m=%d␣g6=%s", TAG, n, m, g6);
```

```
    fflush(stdout);
    fprintf(stderr, "[%s] HIT after %ld graphs m=%d g6=%s", TAG, g_count, m, g6);
    fflush(stderr);
  }
  if (PROGRESS_EVERY > 0 ∧ g_count ≥ g_next_report) {
    struct timespec ts_now;
    clock_gettime(CLOCK_MONOTONIC, &ts_now);
    double secs = (ts_now.tv_sec − g_ts0.tv_sec) + (ts_now.tv_nsec − g_ts0.tv_nsec) * 1 · 10^−9;
    fprintf(stderr,
        "[%s] progress: %ld graphs in %.1fs (%.0f g/s), ""%ld stage1, %ld hits\n", TAG,
        g_count, secs, g_count / secs, g_stage1, g_hits);
    fflush(stderr);
    g_next_report += PROGRESS_EVERY;
  }
}
```

This code is used in section 8.

15. Main. We call geng via the `GENG_MAIN` hook with a fixed argument list: `-c -d2 -D5 -q -u`. The `-D` flag is hard-coded here; to sweep a different max-degree (or drop the limit entirely for the full $n \le 13$ run), edit the line marked below and rebuild. The last command-line argument, if present, is a "res/mod" slice marker passed directly to geng — this lets us split work across processes or machines.

⟨ Main: set up geng invocation 15 ⟩ ≡

```
int GENG_MAIN(int argc, char *argv[]);
int main(int argc, char **argv)
{
  if (argc < 2) {
    fprintf(stderr, "Usage: %s N [tag] [progress_every] [res/mod]\n"
        "  N                 vertex count (must equal the TARGET_N geng generates)\n"
        "  tag               label prepended to each stderr line (default empty)\n"
        "  progress_every    emit a progress line after every K graphs (0 = silent)\n"
        "  res/mod           geng slice specifier: process only residue class r of m\n", argv[0]);
    return 1;
  }
  TARGET_N = atoi(argv[1]);
  if (argc >= 3) TAG = argv[2];
  if (argc >= 4) PROGRESS_EVERY = atol(argv[3]);
  if (TARGET_N < 3 ∨ TARGET_N > MY_MAXN) {
    fprintf(stderr, "Bad n=%d\n", TARGET_N);
    return 1;
  }
  clock_gettime(CLOCK_MONOTONIC, &g_ts0);
  g_next_report = PROGRESS_EVERY;
  char nstr[8];
  snprintf(nstr, sizeof (nstr), "%d", TARGET_N);
  const char *resmod = (argc >= 5) ? argv[4] : Λ;
  char *geng_argv[10];
  int ai = 0;
  geng_argv[ai++] = "geng";
  geng_argv[ai++] = "-c";
  geng_argv[ai++] = "-d2";
  geng_argv[ai++] = "-D5";      /* change per build if desired */
  geng_argv[ai++] = "-q";
  geng_argv[ai++] = "-u";
  geng_argv[ai++] = nstr;
  if (resmod) geng_argv[ai++] = (char *) resmod;
  geng_argv[ai] = Λ;
  int geng_argc = ai;
  GENG_MAIN(geng_argc, geng_argv);
  struct timespec ts_now;
  clock_gettime(CLOCK_MONOTONIC, &ts_now);
  double secs = (ts_now.tv_sec - g_ts0.tv_sec) + (ts_now.tv_nsec - g_ts0.tv_nsec) * 1 · 10^-9;
  fprintf(stderr,
      "[%s] DONE n=%d: scanned %ld graphs in %.1fs (%.0f g/s), "
      "%ld stage1, %ld hits.\n",
      TAG, TARGET_N, g_count, secs, g_count / (secs > 0 ? secs : 1), g_stage1, g_hits);
  return 0;
}
```

This code is used in section 8.

16. Build instructions.

1. Obtain nauty. All experiments in this document used *nauty 2.8.8*: the local single-threaded $n \le 11$ runs used the Debian/Ubuntu package `libnauty-dev 2.8.8+ds-5`, and the cloud multi-thread runs for $n = 12, 13, 14$ used the upstream source tarball `nauty2_8_8.tar.gz` from `https://pallini.di.uniroma1.it/`. Unpack, `./configure`, `make nauty.a`.
2. Tangle this CWEB file:
   ```
   ctangle solution.w
   ```
3. Compile, linking nauty's geng as a plugin:
   ```
   gcc -O3 -march=native \
     -DMAXN=WORDSIZE -DOUTPROC=w_outproc -DGENG_MAIN=geng_main \
     -I /path/to/nauty2_8_8 solution.c /path/to/nauty2_8_8/geng.c \
     /path/to/nauty2_8_8/nauty.a -lm -o solution
   ```
4. Run:
   ```
   ./solution 14 # full n=14 sweep with -D5
   ./solution 14 p0 2000000 0/256 # slice 0 out of 256, progress every 2M
   graphs
   ```
5. For N-way parallel execution (set `N` to your thread count, e.g. 128, 256, or 384):
   ```
   N=256
   for r in $(seq 0 $((N-1))); do
     ./solution 14 p$r 2000000 $r/$N done
   wait
   ```

17. Closing notes.

Main result. For every $n \leq 13$, exhaustive enumeration of all connected simple graphs with minimum degree ≥ 2 produces zero Algorithm-W counter-examples. Knuth's 14-vertex, 22-edge graph therefore realises the tight lower bound $n \geq 14$ on the vertex count of any Algorithm-W counter-example.

Acknowledgments. The fine-grained filtering machinery of `nauty-geng` (McKay and Piperno, "Practical graph isomorphism, II," *J. Symbolic Comput.* **60** (2014), 94–112) carries essentially all the combinatorial load; our own contribution is the $\sim$150 lines of filter and glue in this file.

Collaboration with Claude Code. I carried out this investigation jointly with Anthropic's Claude Code. The division of labour was:

- **Claude Code** wrote the C filter and its integration with `nauty-geng`, rented and configured the cloud server, deployed and executed the parallel sweeps, collected and summarized the resulting data, and drafted the prose of this document.
- **I** reviewed the code and cross-checked every algorithmic-logic step that underpins the verdicts reported here.

Any remaining errors, omissions, or misinterpretations are, of course, mine.