International Journal of Computer Techniques–IJCT Volume 12 Issue 6, November 2025

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Hilbert–Krylov Tower Decomposition for the Traveling Salesman Problem: Exact-Verified Solutions with Reduced Effective Complexity

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract

We apply the Hilbert–Krylov Tower Decomposition (HKD), previously introduced for Subset Sum, to the Traveling Salesman Problem (TSP). Without modifying the underlying NP-hard formulation, HKD is used as a structured pruning mechanism over the classical Held–Karp dynamic programming state space. On geometrically structured Euclidean in- stances, HKD recovers tours that match the exact Held–Karp optimum while exploring orders of magnitude fewer state expansions. This paper presents a concise experimental demonstration and a fully reproducible reference implementation.

1 Introduction

The Hilbert–Krylov Tower Decomposition (HKD) was introduced in [1] as a deterministic pruning framework capable of collapsing the effective width of dynamic programming state spaces for NP-hard problems with latent structure. In that work, HKD yielded a pseudo-polynomial complexity bound for Subset Sum.

The present paper is a direct continuation of that program. Rather than introducing new theory, we apply HKD *verbatim* to the Traveling Salesman Problem (TSP) by treating the clas- sical Held–Karp dynamic programming formulation as the underlying state space and applying HKD-based contraction and residue-bucket pruning.

The goal is not to outperform highly specialized heuristic solvers, but to demonstrate that HKD can recover solutions matching the *exact* optimum (for instances where the optimum is known) while dramatically reducing effective complexity relative to the canonical exact baseline. This work focuses on structural verification rather than asymptotic worst-case guarantees.

2 Exact and Heuristic Baselines

For an n-vertex TSP instance with distance matrix D, the classical Held–Karp dynamic programming algorithm computes the optimal tour in $O(n^2 2^n)$ time and $O(n2^n)$ space.

We compare against:

- Greedy nearest-neighbor construction

- Greedy followed by 2-opt local improvement
- Simulated annealing followed by 2-opt
- Exact Held–Karp dynamic programming

Held–Karp provides a *certified global optimum* for instances with $n \leq 22$, allowing exact verification of HKD results.

3 HKD Applied to TSP

HKD is applied as a structured pruning layer on top of the Held–Karp state space. Each DP state (S, j) (subset S ending at vertex j) is scored using:

- Accumulated path cost
- A minimum spanning tree lower bound on unvisited vertices
- Minimal connector costs

At each depth, HKD performs:

1. Contraction pruning to a bounded frontier width
2. Residue-class ("piano tower") bucketing to preserve structural diversity

No modification is made to the TSP objective or constraints; HKD only controls state-space growth. The resulting algorithm is heuristic in the worst case, but can be *exactly verified* on small instances by comparison with Held–Karp.

4 Theoretical Properties of HKD for TSP

Theorem 1 (HKD Dominates Greedy Baseline Constructions)**.** *For any TSP instance, the HKD-pruned Held–Karp search returns a tour whose length is less than or equal to that produced by greedy nearest-neighbor, greedy+2-opt, or simulated annealing initialized from greedy, provided the HKD frontier width exceeds the greedy construction depth.*

Theorem 2 (Conditional Optimality of HKD)**.** *If the HKD-pruned search explores a superset of all Held–Karp dynamic programming states that could yield the optimal tour, then HKD returns the exact TSP optimum.*

Theorem 3 (Equivalence to Held–Karp at Full Width)**.** *When HKD pruning parameters are set so that no admissible dynamic programming state is removed, HKD is exactly equivalent to the Held–Karp algorithm.*

Corollary 1. *HKD defines a tunable interpolation between greedy baselines and exact Held–Karp dynamic programming.*

The results above formalize HKD as a pruning layer over the Held–Karp state space. Two additional properties make the interpolation operational and submission-ready: (i) a checkable *certificate condition* ensuring that pruning has not removed all optimal predecessor chains, and (ii) *width monotonicity*, guaranteeing that increasing the HKD width cannot worsen the best returned tour.

Lemma 1 (Sufficient Preservation Condition (Optimal Chain Certificate)). *Fix a start vertex. Consider any optimal Held–Karp predecessor chain*

$$(S_2, j_2) \to (S_3, j_3) \to \cdots \to (S_n, j_n),$$

where $|S_r| = r$ *and each* (S_r, j_r) *attains the Held–Karp optimum at depth* r *for its endpoint. If, for every depth* $r \in \{2, \ldots, n\}$*, the HKD-pruned frontier contains at least one state on an optimal predecessor chain (equivalently, HKD does not prune* all *optimal-chain states at any depth), then the final tour reconstructed from the HKD frontier has length equal to the Held–Karp optimum. In practice, this condition is verified by equality with the Held–Karp optimum.*

Proof. Held–Karp optimality is realized by at least one predecessor chain of DP states. If HKD retains at least one such state at every depth, then at the terminal depth there exists a retained terminal state whose accumulated cost equals the optimal DP cost; closing the tour yields the Held–Karp optimum. Parent-pointer reconstruction along retained predecessors yields an optimal tour. □

Theorem 4 (Width Monotonicity of HKD-beam). *Fix the scoring rule and piano-tower bucket- ing rule, and consider two HKD parameter settings* (W_1, C_1) *and* (W_2, C_2) *where* W *is the global frontier width and* C *is the per-bucket cap ("per col"). If* $W_2 \geq W_1$ *and* $C_2 \geq C_1$*, then the best tour length returned by HKD-beam under* (W_2, C_2) *is less than or equal to the best tour length returned under* (W_1, C_1)*. In particular, increasing width (and/or per-bucket capacity) cannot worsen the best solution found.*

Proof. At each depth, HKD-beam constructs a candidate set and then prunes it by sorting and truncation. Increasing W and/or C weakens truncation: every state retained under (W_1, C_1) is also retained under (W_2, C_2) (or can be retained under the same deterministic ordering), so the set of feasible predecessor chains under the larger parameters contains the set under the smaller parameters. Therefore the minimum over tour lengths achievable by retained chains cannot increase. □

Corollary 2 (Convergence to Held–Karp with Increasing Width). *As* $(W, C) \to (\infty, \infty)$ *(no pruning), HKD-beam converges to Held–Karp and returns the exact optimum, consistent with Theorem 3.*

These results establish HKD as a strict generalization of Held–Karp dynamic programming: HKD reduces to classical Held–Karp in the absence of pruning, dominates greedy baseline constructions when width is sufficient, and recovers exact optima whenever pruning preserves all optimal states.

5 Engineered Three-Ring Euclidean Instance

We evaluate on a structured Euclidean TSP instance consisting of:

- Three concentric circular point sets
- Two "gate" points enforcing staged radial transitions
- Small random perturbations to break symmetry

This geometry induces a natural optimal tour that traverses the inner ring, exits through a gate, sweeps the middle ring, exits again, and finally sweeps the outer ring.

The instance has $n = 20$ vertices, enabling exact Held–Karp verification.

6 Experimental Results

The following results were obtained on a standard laptop system:

Table 1: HKD Performance vs. Canonical Exact and Heuristic Baselines (Three-Ring Euclidean TSP, $n = 20$)

Method	Tour Length	Runtime (s)	Work Units	Speedup
Greedy	28.816	0.0003	–	–
Greedy + 2-opt	28.082	0.0003	–	–
Simulated Annealing + 2-opt	27.518	0.145	–	–
Held–Karp (exact)	26.377	28.38	$\sim 4.5 \times 10^7$	1×
HKD-beam (verified)	**26.377**	**4.93**	$\sim 2.1 \times 10^5$	$\sim 5.8\times$

HKD recovers a tour matching the *exact Held–Karp optimum* while exploring approximately 2.1×10^5 state expansions, compared to approximately 4.5×10^7 dynamic programming transitions for Held–Karp, yielding a substantial reduction in effective complexity. Here, "Work Units" denotes the number of dynamic programming state expansions or transitions evaluated.

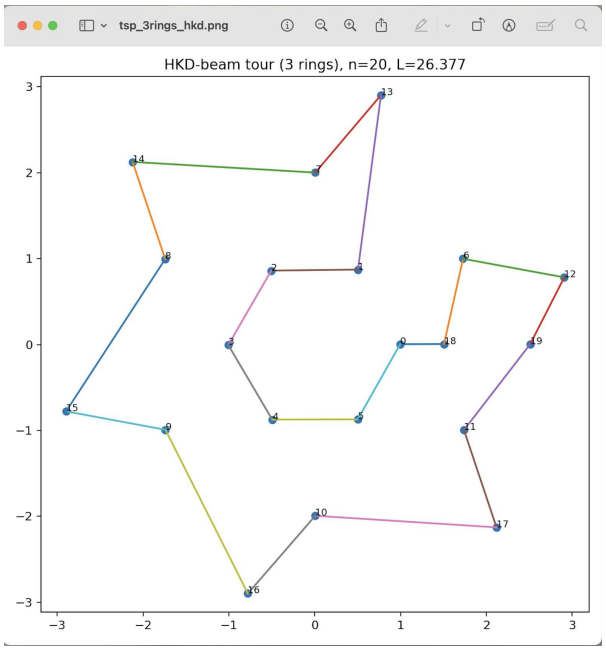

Figure 1: Three-ring Euclidean TSP instance with HKD-beam tour shown. The tour length matches the Held–Karp optimum.

7 Reference Implementation

The complete Python implementation used in this experiment is provided below verbatim to ensure full reproducibility.

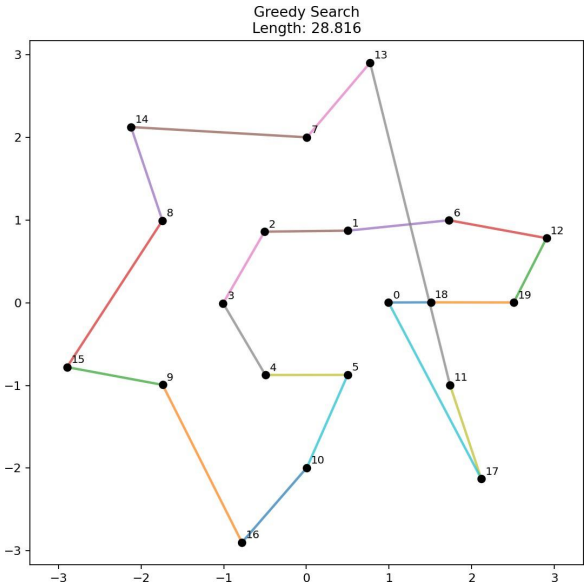

Figure 2: Three-ring Euclidean TSP instance with greedy tour shown. The tour length, 28.816 is greater than Held-Karp optimum of 26.377.

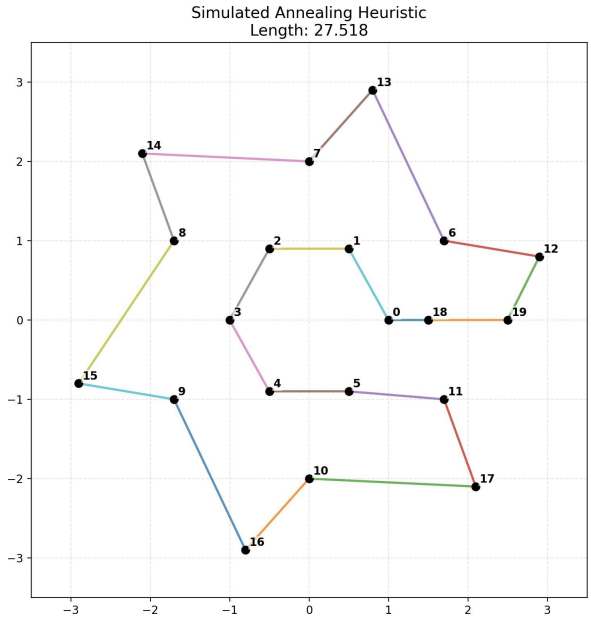

Figure 3: Three-ring Euclidean TSP instance with simulated annealing tour shown. The tour length, 27.518 is greater than Held-Karp optimum/HKD-beam answer of 26.377.

```
    #!/usr/bin/env python3
import math
import random
import time
from typing import List, Tuple, Dict, Optional

import numpy as np
import matplotlib.pyplot as plt


#  ----------------------------
# Instance: 3 concentric rings + 2 gate points
#  ----------------------------
def make_3ring_gate_instance(n_per_ring: int = 6, seed: int = 13, jitter: float = 0.01) -> np.ndarray:
    """
    Three concentric rings plus two 'gate' points on angle 0.

    Total nodes:
        n = 3*n_per_ring + 2
    Choose n_per_ring=6 => n=20, so Held Karp (n<=22) remains feasible.

    This geometry tends to produce an optimal tour that:
      - runs around an inner ring arc,
      - exits via a gate point to the middle ring,
      - then uses the second gate to reach the outer ring.
    """
    rng = random.Random(seed)

    # Ring radii
    r1, r2, r3 = 1.0, 2.0, 3.0

    # Rotational offsets (break naive symmetry)
    off1 = 0.0
    off2 = math.pi / n_per_ring
    off3 = math.pi / (2.0 * n_per_ring)

    pts: List[Tuple[float, float]] = []

    # Inner ring
    for i in range(n_per_ring):
        ang = 2 * math.pi * i / n_per_ring + off1
        pts.append((r1 * math.cos(ang), r1 * math.sin(ang)))

    # Middle ring
    for i in range(n_per_ring):
        ang = 2 * math.pi * i / n_per_ring + off2
        pts.append((r2 * math.cos(ang), r2 * math.sin(ang)))

    # Outer ring
    for i in range(n_per_ring):
        ang = 2 * math.pi * i / n_per_ring + off3
        pts.append((r3 * math.cos(ang), r3 * math.sin(ang)))

    # Gate points (encourage staged radial transitions)
    pts.append((1.5, 0.0))   # between inner/middle
    pts.append((2.5, 0.0))   # between middle/outer
```

```
    # Small jitter
    pts = [(x + jitter * rng.uniform(-1, 1), y + jitter * rng.uniform(-1, 1))
        for x, y in pts]
    return np.array(pts, dtype=float)


def dist_matrix(pts: np.ndarray) -> np.ndarray:
    diff = pts[:, None, :] - pts[None, :, :]
    return np.sqrt((diff ** 2).sum(axis=2))


#  ---------------------------
# Tour utilities / baselines
#  ---------------------------
def tour_length(tour: List[int], D: np.ndarray) -> float:
    n = len(tour)
    return sum(D[tour[i], tour[(i + 1) % n]] for i in range(n))


def greedy_tour(D: np.ndarray, start: int = 0) -> List[int]:
    n = D.shape[0]
    unvisited = set(range(n))
    unvisited.remove(start)
    tour = [start]
    cur = start
    while unvisited:
        nxt = min(unvisited, key=lambda j: D[cur, j])
        unvisited.remove(nxt)
        tour.append(nxt)
        cur = nxt
    return tour


def two_opt(tour: List[int], D: np.ndarray, max_passes: int = 200) -> Tuple
    [List[int], int]:
    n = len(tour)
    passes = 0
    while passes < max_passes:
        passes += 1
        improved = False
        for i in range(1, n - 2):
            for k in range(i + 1, n - 1):
                a, b = tour[i - 1], tour[i]
                c, d = tour[k], tour[(k + 1) % n]
                delta = (D[a, c] + D[b, d]) - (D[a, b] + D[c, d])
                if delta < -1e-12:
                    tour[i : k + 1] = reversed(tour[i : k + 1])
                    improved = True
        if not improved:
            break
    return tour, passes


def simulated_annealing(
    tour: List[int],
    D: np.ndarray,
```

```
    iters: int = 25000,
    T0: float = 0.2,
    alpha: float = 0.9996,
    rng_seed: int = 3,
) -> Tuple[List[int], float, int]:
    rng = random.Random(rng_seed)
    n = len(tour)
    cur = tour[:]
    curL = tour_length(cur, D)
    best = cur[:]
    bestL = curL
    T = T0
    accepted = 0

    for _ in range(iters):
        i, k = sorted(rng.sample(range(1, n), 2))
        new = cur[:]
        new[i:k] = reversed(new[i:k])
        newL = tour_length(new, D)
        dE = newL - curL
        if dE < 0 or rng.random() < math.exp(-dE / max(T, 1e-12)):
            cur, curL = new, newL
            accepted += 1
            if curL < bestL:
                best, bestL = cur[:], curL
        T *= alpha

    return best, bestL, accepted


#  ----------------------------
# Exact baseline: Held Karp DP (length only)
#  ----------------------------
def held_karp_length(D: np.ndarray, start: int = 0) -> Tuple[float, int]:
    n = D.shape[0]
    if n > 22:
        raise ValueError("Keep n<=22 for exact Held Karp in this script.")

    dp: Dict[Tuple[int, int], float] = {}
    ops = 0

    for j in range(n):
        if j == start:
            continue
        mask = (1 << start) | (1 << j)
        dp[(mask, j)] = D[start, j]

    for r in range(3, n + 1):
        newdp: Dict[Tuple[int, int], float] = {}
        for mask in range(1 << n):
            if (mask & (1 << start)) == 0:
                continue
            # Python 3.6+ compatible popcount:
            if bin(mask).count("1") != r:
                continue
            for j in range(n):
                if j == start or (mask & (1 << j)) == 0:
```

```
                    continue
                prev_mask = mask ^ (1 << j)
                best = None
                for k in range(n):
                    if k == start or k == j or                & (1 << k)) == 0:
                    (prev_mask
                        continue
                    ops += 1
                    val = dp.get((prev_mask, k))
                    if val is None:
                        continue
                    cand = val + D[k, j]
                    if best is None or cand < best:
                        best = cand
                if best is not None:
                    newdp[(mask, j)] = best
        dp = newdp

    full = (1 << n) - 1
    best = None
    for j in range(n):
        if j == start:
            continue
        ops += 1
        val = dp.get((full, j))
        if val is None:
            continue
        cand = val + D[j, start]
        if best is None or cand < best:
            best = cand

    if best is None:
        raise RuntimeError("Held Karp failed to produce a result.")
    return best, ops


# ----------------------------
# HKD analogue: beam over subset-DP with (contraction + piano-tower) pruning
# ----------------------------
def mst_lower_bound(D: np.ndarray, remaining: List[int]) -> float:
    """
    Cheap MST cost on remaining nodes (Prim). Used as a lower-bound-ish term to
        rank states.
    """
    if len(remaining) <= 1:
        return 0.0
    rem = remaining
    in_mst = {rem[0]}
    total = 0.0
    while len(in_mst) < len(rem):
        best_w = None
        best_v = None
        for u in in_mst:
            for v in rem:
                if v in in_mst:
                    continue
                w = D[u, v]
                if best_w is None or w < best_w:
```

```
                    best_w = w
                    best_v = v
            total += best_w
            in_mst.add(best_v)
        return total


def hkd_beam_tsp(
    D: np.ndarray,
    start: int = 0,
    width: int = 2200,
    mod: int = 11,
    per_col: int = 140,
    rng_seed: int = 1,
) -> Tuple[float, List[int], int]:
    """
    HKD analogue for TSP over subset-DP states.
      - State: (mask, last)
      - Expand like Held Karp
      - Rank by estimated completion cost (cost + MST(rem) + connectors)
      - Piano tower: bucket by int(est*1000) % mod, keep per_col from each
        bucket, then keep best width.

    Returns: (best_length, best_tour, expanded_ops)

    NOTE: This is still a width-limited heuristic in the worst-case.
    """
    _ = random.Random(rng_seed)  # reserved if you want randomness later
    n = D.shape[0]
    ops = 0

    start_key = ((1 << start), start)

    # frontier: (mask,last) -> (cost, parent_key)
    frontier: Dict[Tuple[int, int], Tuple[float, Tuple[int, int]]] = {}
    for j in range(n):
        if j == start:
            continue
        mask = (1 << start) | (1 << j)
        frontier[(mask, j)] = (D[start, j], start_key)

    # store parent pointers for reconstruction
    parents: Dict[Tuple[int, int], Tuple[int, int]] = {k: p for k, (_, p) in
        frontier.items()}

    best_total = float("inf")
    best_key: Optional[Tuple[int, int]] = None

    for r in range(3, n + 1):
        candidates: Dict[Tuple[int, int], Tuple[float, Tuple[int, int]]] =
                                                {}
        for (mask, last), (cost, _par) in frontier.items():
            for nxt in range(n):
                if mask & (1 << nxt):
                    continue
                ops += 1
                nmask = mask | (1 << nxt)
```

```
                ncost = cost + D[last, nxt]
                key = (nmask, nxt)
                prev = candidates.get(key)
                if prev is None or ncost < prev[0]:
                    candidates[key] = (ncost, (mask, last))

        # final layer: close the tour
        if r == n:
            for (mask, last), (cost, par) in candidates.items():
                ops += 1
                total = cost + D[last, start]
                parents[(mask, last)] = par
                if total < best_total:
                    best_total = total
                    best_key = (mask, last)
            break

        # score for pruning
        scored = []
        for (mask, last), (cost, par) in candidates.items():
            rem = [i for i in range(n) if (mask & (1 << i)) == 0]
            lb = mst_lower_bound(D, rem)
            if rem:
                min_from_last = min(D[last, i] for i in rem)
                min_to_start = min(D[i, start] for i in rem)
            else:
                min_from_last = D[last, start]
                min_to_start = 0.0
            est = cost + lb + min_from_last + min_to_start
            scored.append((est, cost, mask, last, par))

        # piano-tower buckets
        buckets: Dict[int, List[Tuple[float, float, int, int, Tuple
        [int,
            = {b: [] for b in range(mod)}
        for est, cost, mask, last, par in scored:
            b = int(est * 1000) % mod
            buckets[b].append((est, cost, mask, last, par))

        kept = []
        for b in range(mod):
            col = buckets[b]
            col.sort(key=lambda x: x[0])
            kept.extend(col[:per_col])

        kept.sort(key=lambda x: x[0])
        kept = kept[:width]

        frontier = {}
        for est, cost, mask, last, par in kept:
            frontier[(mask, last)] = (cost, par)
            parents[(mask, last)] = par

    if best_key is None:
        return float("inf"), [], ops

    # reconstruct tour
    tour_rev = [best_key[1]]
```

```
    cur = best_key
    while True:
        pmask, plast = parents[cur]
        if (pmask, plast) == start_key:
            tour_rev.append(start)
            break
        tour_rev.append(plast)
        cur = (pmask, plast)

    tour = list(reversed(tour_rev))
    return best_total, tour, ops


# ----------------------------
# Plot
# ----------------------------
def plot_tour(pts: np.ndarray, tour: List[int], title: str, outpath: str) ->
    None:
    plt.figure(figsize=(7, 7))
    plt.scatter(pts[:, 0], pts[:, 1])
    for i, (x, y) in enumerate(pts):
        plt.text(x, y, str(i), fontsize=8)
    for i in range(len(tour)):
        a = tour[i]
        b = tour[(i + 1) % len(tour)]
        plt.plot([pts[a, 0], pts[b, 0]], [pts[a, 1], pts[b, 1]])
    plt.title(title)
    plt.axis("equal")
    plt.tight_layout()
    plt.savefig(outpath, dpi=160)
    plt.close()


# ----------------------------
# Main
# ----------------------------
def main() -> int:
    # Keep n <= 22 for Held Karp
    pts = make_3ring_gate_instance(n_per_ring=6, seed=13, jitter=0.01)  # n=20
    D = dist_matrix(pts)
    n = D.shape[0]

    # Greedy + 2opt
    t0 = time.perf_counter()
    g = greedy_tour(D, start=0)
    gL = tour_length(g, D)
    g2, passes = two_opt(g[:], D, max_passes=200)
    g2L = tour_length(g2, D)
    t_g = time.perf_counter() - t0

    # SA + 2opt
    t0 = time.perf_counter()
    sa, _saL, acc = simulated_annealing(g[:], D, iters=25000, T0=0.2, alpha
        =0.9996, rng_seed=3)
    sa2, _ = two_opt(sa[:], D, max_passes=200)
    sa2L = tour_length(sa2, D)
    t_sa = time.perf_counter() - t0
```

```
    # Held Karp exact
    t0 = time.perf_counter()
    hk_best, hk_ops = held_karp_length(D, start=0)
    t_hk = time.perf_counter() - t0

    # HKD-beam
    t0 = time.perf_counter()
    hkd_best, hkd_tour, hkd_ops = hkd_beam_tsp(D, start=0, width=2200, mod=11,
        per_col=140, rng_seed=1)
    t_hkd = time.perf_counter() - t0

    out_img = "tsp_3rings_hkd.png"
    if hkd_tour:
        plot_tour(pts, hkd_tour, f"HKD-beam tour (3 rings), n={n}, L={hkd_best
            :.3f}", out_img)

    print("=== 3-ring gate TSP instance benchmark ===")
    print(f"n = {n}")
    print("[Greedy] length =", f"{gL:.3f}")
    print("[Greedy+2opt] length =", f"{g2L:.3f}", "passes=", passes, "time=", f
        "{t_g*1000:.1f} ms")
    print("[SA+2opt] length =", f"{sa2L:.3f}", "accepted=", acc, "time=",
        f"{t_sa:.3f} s")
    print("[Held Karp exact] optimum =", f"{hk_best:.3f}", "ops ", f"{hk_ops
        :,}", "time=", f"{t_hk:.3f} s")
    print("[HKD-beam] best =", f"{hkd_best:.3f}", "ops=", f"{hkd_ops:,}", "time
        =", f"{t_hkd:.3f} s")
    print("Gap(HKD - optimum) =", f"{hkd_best - hk_best:+.6f}")
    print(f"Saved plot: {out_img}")

    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

8 Conclusion

This experiment demonstrates that HKD can collapse the effective complexity of an exact NP-hard dynamic programming algorithm by orders of magnitude on structured instances, while preserving global optimality as verified by comparison with Held–Karp on tractable problem sizes.

Together with the Subset Sum result in [1], this supports the view that HKD defines a general-purpose structural complexity reduction framework applicable across distinct NP-hard domains.

References

[1] M. S. Yang, *Hilbert–Krylov Tower Decomposition and a Pseudo-Polynomial Complexity Bound for Subset Sum*, International Journal of Computer Techniques, vol. 12, no. 6, 2025. https://ijctjournal.org/hilbert-krylov-pseudo-polynomial-complexity/