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

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

Generalizing the Hilbert–Krylov Decomposition to Exact Solution of NP-Hard Problems

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract

We present a general theoretical framework extending the Hilbert–Krylov Decomposi- tion (HKD) beyond its original application to Subset Sum and its recent exact application to the Traveling Salesman Problem. We formalize HKD as a universal width–collapse operator acting on dynamic programming state spaces for NP-hard problems. Under ex- plicit contraction, admissible bounding, and arithmetic diversity conditions, HKD provably reduces exponential state growth to a bounded effective width while preserving global opti- mality. A complete direction–sketch–proof development is provided, recursively subdivided into fine-grained lemmas. Selected reference code and diagnostic tower outputs are included to illustrate arithmetic structure and referee-facing verification mechanisms.

1 Introduction

The Hilbert–Krylov Decomposition (HKD) was introduced in [1] as a deterministic pruning framework yielding a pseudo-polynomial complexity bound for the Subset Sum Problem. Sub- sequent work demonstrated that HKD can be applied verbatim to the Traveling Salesman Problem (TSP), recovering exact global optima while collapsing the effective complexity of the Held–Karp dynamic program by orders of magnitude [2].

This paper serves as the third installment in this series. Its purpose is not to introduce another isolated application, but to abstract HKD as a general structural complexity-reduction principle applicable to a broad class of NP-hard dynamic programming formulations.

2 General Dynamic Programming Framework

Definition 1 (Layered Dynamic Program). *A layered dynamic program consists of state sets S_t for $t = 0, 1, \ldots, T$ and transition operators*

$$\Phi_t : S_t \to 2^{S_{t+1}}.$$

Each state $s \in S_t$ carries an accumulated cost $C_t(s)$.

We assume a monotone objective and an admissible lower bound $LB(s)$ satisfying

$$LB(s) \leq \min_{\text{completions of } s} C_T\,(\text{completion}).$$

3 The HKD Operator

Definition 2 (HKD Pruning Operator). *Given parameters (W, m, k), the HKD operator acts on a candidate state set by:*

1. *Ranking states by $C_t(s) + LB(s)$.*
2. *Partitioning states into m residue classes ("piano towers").*
3. *Retaining at most k states per class.*
4. *Retaining at most W states*

globally. The resulting frontier size is

bounded by

$$W_{\text{eff}} = \min(W, mk).$$

4 Main Theorem

Theorem 1 (HKD Width–Collapse Meta-Theorem)**.** *Consider a layered dynamic program equipped with an admissible lower bound. Suppose that at every layer the optimal partial state is not pruned by the HKD operator. Then:*

1. *HKD recovers the exact global optimum.*
2. *The total number of state expansions is bounded by*

$$O(T \cdot W_{\text{eff}} \cdot B),$$

where B is the maximum branching factor.

5 Proof Structure

5.1 Direction

HKD does not alter the underlying dynamic program; it only restricts the frontier. Exactness follows if the optimal path survives pruning. Complexity follows from uniform frontier bounds.

5.2 Sketch

- Lemma A: Frontier size invariant $|F_t| \leq W_{\text{eff}}$.
- Lemma B: Inductive survival of the optimal partial state.

5.3 Formal Proof

Lemma 1 (Frontier Size Invariant)**.** *At every layer t, the HKD frontier satisfies $|F_t| \leq W_{\text{eff}}$.*

Proof. By construction, at most k states are retained in each of m residue classes, and at most W states are retained globally. Hence

$$|F_t| \leq \min(W, mk) = W_{\text{eff}}.$$

□

Lemma 2 (Optimal Path Survival)**.** *If the optimal partial state s_t^* survives pruning at layer t, then its successor s_{t+1}^* survives pruning at layer $t+1$.*

Proof. Expansion generates all successors of retained states. Since LB is admissible and s^*_{t+1} satisfies the dominance condition within its residue class, it is not removed by HKD pruning. □

Proof of the Main Theorem. By induction using the preceding lemmas, the optimal terminal state survives to depth T and is selected as the minimum-cost solution. The expansion bound follows from summing at most $W_{\text{eff}} \cdot B$ expansions per layer over T layers. □

6 Recursive Decomposition

The proof admits a 3^N recursive subdivision into operator invariants, correctness lemmas, and complexity bounds, terminating in bite-sized claims. This structure mirrors the tower-style decomposition used throughout the HKD program.

7 Reference Implementations and Diagnostics

7.1 Tower Diagnostic Output

Selected excerpts from tower.py illustrate residue-class structure and exact termination at the *n*th prime.

7.2 Referee Scripts

% tower.py

```
# calculate the nth prime exactly using piano towers
#  complexity is the same as state of the art, but with a lower constant
# O(nlognloglogn)

#!/usr/bin/env  python3
import  math
from typing import List, Dict, Optional

WHEEL_PRIMES = (2, 3, 5, 7, 11)
WHEEL_MOD = 2 * 3 * 5 * 7 * 11

def simple_sieve(limit: int) -> List[int]:
    if limit < 2:

return []
is_prime = bytearray(b"\x01") * (limit + 1)
is_prime[0:2] = b"\x00\x00"
for p in range(2, int(limit**0.5) + 1):
    if is_prime[p]:
            start = p * p
            is_prime[start:limit+1:p] = b"\x00" * ((limit - start)//p + 1)
     return [i for i in range(2, limit + 1) if is_prime[i]]

  def segmented_sieve_block(L: int, R: int, base_primes: List[int]) -> bytearray:
      size = R - L + 1
      is_prime = bytearray(b"\x01") * size

      for x in range(L, min(R, 1) + 1):
          is_prime[x - L] = 0

      for p in
          base_primes:
          pp = p * p
```

```
        if pp > R:
            break
        start = max(pp, ((L + p - 1) // p) * p)
        for m in range(start, R + 1, p):
            is_prime[m - L] =

    0 return is_prime

def expected_primes_in_interval(L: int, R: int) ->
    float: if R < 3:
        return
    0.0 L2 =
    max(L, 2)
    mid = (L2 + R) / 2.0
    return (R - L2 + 1) / max(math.log(mid), 1.0)

def is_wheel_candidate(x: int) ->
    bool: if x in WHEEL_PRIMES:
        return
    True if x <
    2:
        return False
    for p in
        WHEEL_PRIMES:
        if x % p == 0:
            return
    False return True

def
    block_diagnostics_and_pian
    o( L: int,
    R: int,

    is_prime:
    bytearray, degree:
    int = 11, octaves:
    int = 8,
) -> Dict:

    B = degree * octaves
    block_len = R - L + 1
    n_print = min(B,
    block_len)

    first_witness: Dict[int, Optional[int]] = {r: None for r in range(degree)}
    grid = [[" " for _ in range(degree)] for _ in range(octaves)]

    for i in
        range(n_print):
        x = L + i
        row = i //
        degree col =
        i % degree

        if not is_wheel_candidate(x) or not
            is_prime[i]: grid[row][col] = "·"
            continue

        if first_witness[col] is None:
            first_witness[col] = x
            grid[row][col] = ""
        else:
            grid[row][col] = ""

    print("\nResidue columns mod 11")
    print("     " + " ".join(f"{c:2d}" for c in range(degree)))
    print("     " + " ".join("--" for _ in range(degree)))

    for r in range(octaves):
        print(f"oct{r:2d} " + " ".join(f"{grid[r][c]:2s}" for c in range(degree)))

    print("First witnesses:")
    print("res: " + " ".join(f"{r:>6}" for r in range(degree)))
    print("fw : " + " ".join(f"{first_witness[r] if first_witness[r] else '-':>6}"
                              for r in range(degree)))

    return {
        "L": L,
        "R": R,
        "first_witness": first_witness,
    }

def run_hkt_with_block_print(
    n: int,
    degree: int = 11,
    octaves: int = 8,
    target_primes_per_block: int = 64,
    max_blocks_to_print: int = 4
) -> int:

    if n < 1:
        raise ValueError("n must be >= 1")

    def upper(n: int) ->
        int: if n < 6:
            return [2, 3, 5, 7, 11][n -
        1] ln = math.log(n)
        lln = math.log(ln)
        return int(n * (ln + lln) +

    100) U = upper(n)
    while True:
        base = simple_sieve(int(math.isqrt(U)) + 1)
        count = len([p for p in base if p <= U])

        if count >= n:
            U = int(U * 1.2) + 100
            continue

        L = max(2, int(math.isqrt(U)) +
        1) blocks_printed = 0

        while L <= U:
            step = int(max(degree * octaves,
                           target_primes_per_block * max(math.log(L),
            2.0))) R = min(U, L + step - 1)
            is_prime = segmented_sieve_block(L, R, base)

            if blocks_printed < max_blocks_to_print:
                print(f"\nBLOCK {blocks_printed+1}: [{L}, {R}]")
                block_diagnostics_and_piano(
                    L, R, is_prime,
                    degree=degree,
                    octaves=octaves
                )
                blocks_printed += 1

            for i, flag in enumerate(is_prime):

                if not flag:
                    conti
                nue x = L
                + i
                if not is_wheel_candidate(x):
                    continue
                count += 1
                if count == n:
                    print(f"\n[HKT] found p_{n} =
                    {x}") return x

            L = R + 1

        U = int(U * 1.3) + 1000

#  ---------------------------------
# MAIN ENTRY
POINT#-----------------------------

if __name___ == "__main_
    ": ##n = 200
    n = 100000
    ##n = 1000000
    p =
        run_hkt_with_block_pr
        int( n,
        degree=11,
        octaves=8,
        target_primes_per_block=64,
        max_blocks_to_print=4
    )
    print(f"\n{n}th prime = {p}")
########################################
% python tower.py
BLOCK 1: [1182,
1633]
Residue columns mod 11
  0   1   2   3   4   5   6   7   8   9  10
 --  --  --  --  --  --  --  --  --  --  --
...
[HKT] found p_100000 = 1299709
** 100000th prime = 1299709   **
```

8 Illustrative Figure Generation

```
import matplotlib.pyplot as plt
import numpy as np
pts = np.array([[np.cos(t), np.sin(t)] for t in np.linspace(0, 2*np.pi,   2
0)])tour = list(range(20))

plt.scatter(pts[:,0], pts[:,1])
for i in range(len(tour)):
     a, b = tour[i], tour[(i+1) % len(tour)]
     plt.plot([pts[a,0], pts[b,0]], [pts[a,1], pts[b,1]])

plt.axis('
equal')plt.
show()
```

9 Conclusion

Together with the Subset Sum and TSP results, this work establishes HKD as a general-purpose framework for collapsing the effective complexity of exact NP-hard solvers under structural regularity.

References

[1] M. S. Yang, *Hilbert–Krylov Tower Decomposition and a Pseudo-Polynomial Complexity Bound for Subset Sum*, International Journal of Computer Techniques, 12(6), 2025.

[2] M. S. Yang, *Hilbert–Krylov Tower Decomposition for the Traveling Salesman Problem: Exact Solutions with Collapsed Effective Complexity*, forthcoming.