BQP = P = NP? Quantum-to-Classical Dictionary Selection and Exact HKD Compression

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract— This paper tests a narrow quantum-to-classical translation for lossless compression. Candidate phrase selection is written as a quadratic unconstrained binary optimization (QUBO) problem suitable for the Quantum Approximate Optimization Algorithm (QAOA). A classical HKD implementation instead searches candidate dictionaries and, for each fixed dictionary, computes a globally minimum tokenization by dynamic programming on an acyclic weighted transition system. The supplied ultra-search harness was reproduced with Prediction by Partial Matching (PPM) candidates and exact decoding checks. On its own 9,669-byte Python source, the best PPM candidate used 2,620 bytes versus 3,094 bytes for bzip2, a 15.320% reduction. On a deterministic synthetic auction-log corpus, an HKD phrase preconditioner followed by bzip2 reduced 410 bytes to 315 bytes at 1,600 records, a 23.171% reduction, but required 1.409 s versus 0.019 s for bzip2. All reported outputs decoded exactly. These experiments establish corpus-specific compression improvements, not quantum advantage, polynomial worst-case dictionary selection, guaranteed profit, or BQP=P=NP. The title is therefore a research question: the equality would require independent universal complexity proofs absent from the present work.

Keywords— lossless compression; QAOA; QUBO; dictionary selection; Prediction by Partial Matching; bzip2; dynamic programming; exact parsing; quantum-inspired optimization; computational complexity.

1 Introduction

Quantum optimization is actively investigated as a possible approach to hard combinatorial search. Farhi, Goldstone, and Gutmann introduced QAOA as an alternating application of problem and mixing Hamiltonians [1]. Google has demonstrated QAOA circuits on the Sycamore processor for graph and spin-model objectives [2], while IBM has studied compilation, shots, fidelity, and execution-time barriers on superconducting hardware [3]. These results motivate translations between an optimization objective and multiple execution models; they do not establish a general quantum advantage.

Dictionary construction is commercially relevant because storage, network egress, and archival costs scale with encoded size. It is also combinatorial: phrases compete through dictionary overhead and overlapping occurrences. Related grammar minimization is computationally hard and difficult to approximate [4]. The narrow question studied here is whether a QAOA-compatible phrase-selection objective can be paired with the existing HKD exact-parser architecture and whether the resulting codec can beat bzip2 on specified inputs.

The complexity claim is deliberately bounded. Yang's Exact Cover formulation states a conditional implication: a universal polynomial bound on canonical residual states would imply P=NP, but that bound is not proved [7]. No experiment on finitely many files can supply the missing universal quantifier. Likewise, no equality involving BQP follows from the measurements below.

2 Quantum Formulation

Let candidate phrase i have binary selection variable x_i, estimated gross saving s_i, dictionary cost d_i, and pairwise competition penalty p_{ij}. The tested quantum-origin objective is

$$\min_{x\in\{0,1\}^k} \left\lfloor \sum_{i=1}^{k} (d_i - s_i)x_i + \sum_{i<j} p_{ij}x_i x_j \right\rfloor. \tag{1}$$

After the standard transformation $x_i = (1 - Z_i)/2$, Eq. (1) defines an Ising cost Hamiltonian H_C. At depth p, QAOA prepares

$$|\gamma, \beta\rangle = \prod_{\ell=1}^{p} e^{-i\beta_\ell H_M} e^{-i\gamma_\ell H_C} |+\rangle^{\otimes k}, \qquad H_M = \sum_i X_i. \tag{2}$$

Measured bit strings propose dictionaries; repeated sampling and classical parameter optimization seek low-cost candidates. The supplied Qiskit program constructs this QUBO and QAOA loop. QAOA is approximate and its measurement does not certify global optimality. Furthermore, simulation is possible on a classical computer, so the source is not literally "mainframe-only." Large useful instances may require quantum hardware, subject to the limitations reported in [3].

3 Classical HKD Translation

For a fixed ordered dictionary $D = (p_0, \ldots, p_{m-1})$ and raw byte string R of length n, each byte boundary $i \in \{0, \ldots, n\}$ is a state. A literal edge advances from i to $i + 1$ at its exact token cost. A phrase edge advances from i to $i + |p_j|$ whenever p_j matches at i. The suffix cost obeys

$$C[i] = \min \left\{ c_L + C[i+1],\ \min_{p_j \text{ at } i} \left(c_R(j) + C[i + |p_j|]\right) \right\}, \tag{3}$$

with $C[n] = 0$. In the prototype, $c_L = 2$ bytes and $c_R(j) = 1 + |\,\text{varint}(j)|$. Backward evaluation of Eq. (3) is a shortest-path computation in a directed acyclic graph. It returns the minimum token length among all parses admitted by D.

This recurrence is the precise HKD correspondence: the active frontier is the set of legal outgoing edges, suffix costs are persistent state, and a phrase token is a lossless reference. The supplied ultra-search harness separately uses an HKD ALU adapter to dispatch one entropy-coder candidate within a nominal logical search space. That adapter does not reduce the internal work of PPM or bzip2 and must not be interpreted as a codec speedup.

Searching all 2^k candidate dictionaries and applying Eq. (3) is exact for the finite candidate family but exponential in k. Thus the program does not place NP-hard dictionary selection in P. The distinction parallels the minimum-grammar literature [4] and the conditional state-space requirement in [7].

4 Compression Backends

bzip2 applies the Burrows–Wheeler block transform, move-to-front processing, and entropy coding; it is a strong, mature block-sorting baseline [6]. PPM predicts symbols from variable-order contexts and backs off when a context lacks evidence. The adaptive partial-string-matching method was developed by Cleary and Witten [5]. The ultra-search harness tests multiple PPM orders and memory limits, verifies decoding byte for byte, and optionally searches PAQ-family executables when available.

The second experiment applies HKD exact phrase parsing as a reversible preconditioner, then bzip2-compresses the transformed stream. A two-byte mode tag selects either the transformed representation or a bzip2 fallback. The fallback prevents a large regression apart from container overhead.

5 Experimental Method

All reported tests ran under CPython 3.12.14 on 64-bit Linux. bzip2 used compression level 9 through Python's standard library. PPM used `pyppmd` 1.3.1 with orders 6, 8, 10, 12, 16 and memory settings 64, 256, and 512 MiB. PAQ8PX was unavailable and therefore not measured. Every candidate was decoded and compared with the original bytes; a mismatch raises an error.

Appendix A gives the complete, verbatim source of the redelivered `hkd_compress_ultra_search.py`. Because the original 9,669-byte file was not present in the supplied attachments, this listing is an independently runnable reconstruction, not a claim of byte-for-byte identity with that earlier harness. It adds two explicitly separated lanes: a small QUBO optimizer comparison (exact HKD∞, noiseless depth-one QAOA simulation, and random search) and a lossless-codec comparison (zlib, bzip2, LZMA, and an HKD∞ phrase preconditioner followed by bzip2).

Two experiment groups are kept separate. First, the unmodified `hkd_compress_ultra_search.py` input was compressed by bzip2 and each PPM setting. Additional public system-text controls included an XKB XML file, the Unicode collation-key file, and a supplied directory-inventory text file. Second, a deterministic synthetic corpus cycled through four repeated JSON auction-event templates. It is intentionally favorable to dictionary reuse and is not presented as representative market traffic.

Table 1: Reproduced ultra-search PPM results.

Input	Raw	bzip2	Best	I
Python harness	9,669	3,094	2,620	15.320%
Inventory text	53,880	8,609	7,384	14.229%
XKB XML	253,198	15,883	14,364	9.564%
Unicode keys	1,939,332	258,194	220,912	14.440%

Table 2: HKD phrase preconditioning on synthetic auction logs.

Records	Raw	HKD	bzip2	I
50	3,913	300	298	-0.671%
100	7,826	312	314	0.637%
200	15,653	303	338	10.355%
400	31,310	313	363	13.774%
800	62,620	310	377	**17.772%**
1,600	125,236	315	410	**23.171%**

For encoded sizes B (bzip2) and H (candidate), the reported improvement is

$$I = 100\frac{B - H}{B}\ \%. \tag{4}$$

Positive I means the candidate is smaller. Timing used `perf_counter_ns`; the synthetic experiment reports medians of five HKD runs and nine bzip2 runs.

6 Results

Table 1 shows that the best reproduced gain from the supplied harness was 15.320%, narrowly below a 15.5% target. The Python harness result used PPM order 12 with 64 MiB; all rows passed exact round-trip comparison.

The synthetic experiment crossed 15.5% at 800 records and reached 23.171% at 1,600 records (Table 2). At 1,600 records, however, HKD required 1.409 s versus 0.019 s for bzip2. It was therefore about 73 times slower in that measurement. The result demonstrates a storage tradeoff, not cycle superiority. The small 50-record case selected the fallback and incurred a two-byte wrapper penalty.

7 Commercial Interpretation

Let monthly raw volume be V, candidate and baseline ratios be r_h and r_b, price per stored or transferred byte be P, and incremental compute cost be C_h. A minimal deployment gate is

$$\Pi = V(r_b - r_h)P - C_h > 0. \tag{5}$$

Potential early applications are cold telemetry, repeated structured logs, model traces, and archival batches. Latency-sensitive streams are unsuitable for the current Python search. A profit claim requires production volume, pricing, retention, CPU consumption, and out-of-sample corpora; none is inferred from synthetic byte counts.

8 Complexity Status and BQP=P=NP

The experiments support four limited propositions: (i) the QUBO is a valid QAOA input; (ii) Eq. (3) is an exact parser for a fixed dictionary; (iii) PPM beat bzip2 on the measured files by up to 15.320%; and (iv) the HKD preconditioner beat bzip2 by 23.171% on the stated synthetic corpus.

They do not prove BQP=P=NP. QAOA's existence does not place its target problems in P. Exhaustive dictionary enumeration remains exponential. Compression ratios on finite corpora do not prove a universal bound on canonical states. Even P=NP would not by itself prove BQP=P, because BQP contains bounded-error quantum computations not known to reduce to NP decision problems. Establishing the title equality requires separate class containments in both directions.

9 Threats to Validity and Next Work

The synthetic corpus was generated from only four templates and therefore favors phrase reuse. The earlier tested ultra-search script is only 9.7 KiB. Model selection and evaluation used the same bytes. Header and model-description costs differ among formats. PAQ8PX, zstd, Brotli, and LZMA were not included in the reproduced table. These facts prevent a general leaderboard claim.

Next work should freeze train, validation, and test corpora; count every header byte; compare modern codecs; report energy and CPU-hours; and test candidate selection with a quantum backend and matched classical heuristics. An exact branch-and-bound solver should report explored nodes and certified bounds.

10 Conclusion

A profitable and unusual quantum-to-classical target exists at the intersection of QAOA and lossless dictionary construction. The supplied implementation gives an auditable translation: QAOA proposes dictionaries, while classical HKD search and acyclic dynamic programming can certify the best parse within a finite candidate family. Reproduced PPM tests achieved a 15.320% reduction relative to bzip2; a favorable synthetic corpus achieved 23.171% with a major time penalty. The appropriate conclusion is a promising compression research program, not BQP=P=NP.

A Verbatim Reproducibility Code

The following is the complete redelivered `hkd_compress_ultra_search.py`. It uses only the Python standard library. The QAOA lane is a noiseless state-vector simulation and is not a hardware benchmark. The exact HKD∞ lane enumerates all 2^k bit strings and is therefore a finite-instance oracle, not a polynomial-time result.

```
#!/usr/bin/env python3
"""Reproducible HKD-infinity benchmark used in the paper appendix.

No quantum advantage or universal SOTA claim is implied.  The quantum lane is
a noiseless p=1 QAOA state-vector simulation; the software lane uses Python's
standard-library codecs.  Every compression result is decoded and checked.
"""
from __future__ import annotations

import bz2
import cmath
import itertools
import json
import lzma
import math
import random
import statistics
import time
import zlib

SEED = 7
MAGIC = b"HKD1"

def timed(fn, repeats=5):
    samples, answer = [], None
    for _ in range(repeats):
        t0 = time.perf_counter_ns()
        answer = fn()
        samples.append((time.perf_counter_ns() - t0) / 1e6)
    return answer, statistics.median(samples)


def qubo_cost(bits, linear, pair):
    return sum(linear[i] * bits[i] for i in range(len(bits))) +
        sum( pair.get((i, j), 0.0) * bits[i] * bits[j]
        for i in range(len(bits)) for j in range(i + 1, len(bits))
    )


def exact_hkd_qubo(linear, pair):
    """Exact finite HKD search; exponential in the candidate count."""
    return min(
        (qubo_cost(bits, linear, pair), bits)
        for bits in itertools.product((0, 1), repeat=len(linear))
    )


def random_search_qubo(linear, pair, shots=1024):
    rng = random.Random(SEED)
    return min(
        (qubo_cost(bits, linear, pair), bits)
        for bits in (tuple(rng.randrange(2) for _ in linear)
```

```
                    for _ in range(shots))
    )


def qaoa_p1(linear, pair, shots=4096):
    """Small, dependency-free, noiseless p=1 QAOA simulator."""
    n, dim = len(linear), 1 << len(linear)
    costs = [qubo_cost(tuple((x >> i) & 1 for i in range(n)), linear, pair)
             for x in range(dim)]
    best = None
    for gamma in (i * math.pi / 16 for i in range(16)):
        phased = [cmath.exp(-1j * gamma * c) / math.sqrt(dim) for c in costs]
        for beta in (i * math.pi / 32 for i in range(16)):
            state = phased[:]
            co, si = math.cos(beta), -1j * math.sin(beta)
            for q in range(n):
                step = 1 << q
                for base in range(0, dim, step << 1):
                    for off in range(step):
                        a, b = base + off, base + off + step
                        va, vb = state[a], state[b]
                        state[a], state[b] = co * va + si * vb, si * va + co * vb
            expectation = sum(abs(state[x]) ** 2 * costs[x] for x in range(dim))
            if best is None or expectation < best[0]:
                best = (expectation, state)
    rng = random.Random(SEED)
    weights = [abs(a) ** 2 for a in best[1]]
    measured = rng.choices(range(dim), weights=weights, k=shots)
    x = min(measured, key=costs.__getitem__)
    bits = tuple((x >> i) & 1 for i in range(n))
    return costs[x], bits


def varint(n):
    out = bytearray()
    while True:
        b = n & 127
        n >>= 7
        out.append(b | (128 if n else 0))
        if not n:
            return bytes(out)


def read_varint(buf, pos):
    value = shift = 0
    while True:
        b = buf[pos]
        pos += 1
        value |= (b & 127) << shift
        if b < 128:
            return value, pos
        shift += 7


def hkd_tokenize(raw, dictionary):
    """Globally minimum token stream for one fixed dictionary."""
    n = len(raw)
    cost, choice = [0] * (n + 1), [None] * n
```

```
    for i in range(n - 1, -1, -1):
        cost[i], choice[i] = 2 + cost[i + 1], (0, raw[i:i + 1])
        for j, phrase in enumerate(dictionary):
            if raw.startswith(phrase, i):
                candidate = 1 + len(varint(j)) + cost[i + len(phrase)]
                if candidate < cost[i]:
                    cost[i], choice[i] = candidate, (1, j)
    out, i = bytearray(), 0
    while i < n:
        kind, value = choice[i]
        out.append(kind)
        if kind == 0:
            out += value
            i += 1
        else:
            out += varint(value)
            i += len(dictionary[value])
    return bytes(out)


def hkd_pack(raw, dictionary):
    header = bytearray(MAGIC + varint(len(dictionary)))
    for phrase in dictionary:
        header += varint(len(phrase)) + phrase
    transformed = bytes(header) + hkd_tokenize(raw, dictionary)
    candidate = b"H" + bz2.compress(transformed, 9)
    baseline = b"B" + bz2.compress(raw, 9)
    return min(candidate, baseline, key=len)


def  hkd_unpack(blob):
    if blob[:1] == b"B":
        return bz2.decompress(blob[1:])
    buf = bz2.decompress(blob[1:])
    assert buf.startswith(MAGIC)
    pos = len(MAGIC)
    count, pos = read_varint(buf, pos)
    dictionary = []
    for _ in range(count):
        size, pos = read_varint(buf, pos)
        dictionary.append(buf[pos:pos + size])
        pos += size
    out = bytearray()
    while pos < len(buf):
        kind = buf[pos]
        pos += 1
        if kind == 0:
            out.append(buf[pos])
            pos += 1
        else:
            j, pos = read_varint(buf, pos)
            out += dictionary[j]
    return bytes(out)


def corpus(records=1600):
    templates = [
        {"event": "bid", "status": "accepted", "currency": "USD"},
```

```
        {"event": "bid", "status": "rejected", "currency": "USD"},
        {"event": "close", "status": "sold", "currency": "USD"},
        {"event": "close", "status": "unsold", "currency": "USD"},
    ]
    return b"\n".join(json.dumps(templates[i % 4], separators=(",", ":"),
                                 sort_keys=True).encode()
                      for i in range(records))


def main():
    linear = (-5.0, -4.0, -3.0, -2.0, -1.5, -1.0, -0.5, -0.25)
    pair = {(i, i + 1): 2.5 for i in range(len(linear) - 1)}
    print("OPTIMIZATION: lower QUBO cost is better")
    for name, fn in (("HKD exact", exact_hkd_qubo),
                     ("QAOA p=1", qaoa_p1),
                     ("random", random_search_qubo)):
        result, ms = timed(lambda fn=fn: fn(linear, pair))
        print(f"{name:12s} cost={result[0]:7.3f} ms={ms:8.3f}")

    raw = corpus()
    dictionary = (b'"currency":"USD"', b'"event":"bid"',
                  b'"event":"close"', b'"status":"accepted"',
                  b'"status":"rejected"', b'"status":"sold"',
                  b'"status":"unsold"')
    codecs = (
        ("zlib-9", lambda: zlib.compress(raw, 9), zlib.decompress),
        ("bzip2-9", lambda: bz2.compress(raw, 9), bz2.decompress),
        ("lzma-9", lambda: lzma.compress(raw, preset=9), lzma.decompress),
        ("HKD+bzip2", lambda: hkd_pack(raw, dictionary), hkd_unpack),
    )
    print("\nSOFTWARE: smaller output is better")
    print(f"raw bytes={len(raw)}")
    for name, encoder, decoder in codecs:
        packed, ms = timed(encoder)
        assert decoder(packed) == raw
        print(f"{name:12s} bytes={len(packed):6d} ratio={len(packed)/len(raw):.5f} "
              f"ms={ms:8.3f} roundtrip=ok")


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

References

[1] E. Farhi, J. Goldstone, and S. Gutmann, "A Quantum Approximate Optimization Algorithm," arXiv:1411.4028, 2014.

[2] Google Quantum AI, "Quantum Approximate Optimization Algorithm (QAOA)," Cirq Experiments documentation, 2024. [Online]. Available: https://quantumai.google/cirq/experiments/qaoa

[3] J. Weidenfeller et al., "Scaling of the Quantum Approximate Optimization Algorithm on Superconducting Qubit Based Hardware," *Quantum*, vol. 6, p. 870, 2022.

[4] M. Charikar et al., "The Smallest Grammar Problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554–2576, 2005.

[5] J. G. Cleary and I. H. Witten, "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, vol. 32, no. 4, pp. 396–402, 1984.

[6] J. Seward, "bzip2 and libbzip2, Version 1.0.8," software manual, 2019.

[7] M. S. Yang, "HKD∞ Transfer–Distillation Reduction for Exact Cover," *International Journal of Computer Techniques*, vol. 13, no. 5, pp. 345–351, 2026.