

A Conditional Route to $BQP = P = NP$: Canonical Exact-Cover States and a QAOA Attachment

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract

This article formalizes the relationship between a previously published HKD-infinity transfer–distillation reduction for Exact Cover by 3-Sets and a quantum approximate optimization attachment. The classical reduction is exact but conditional: if the number of distinct canonical residual states is polynomially bounded for every input, then Exact Cover by 3-Sets is in polynomial time and hence $P = NP$. A quadratic unconstrained binary optimization model permits candidate-state or dictionary selection to be processed by the Quantum Approximate Optimization Algorithm. That attachment does not establish a quantum speedup and does not imply $BQP \subseteq P$. We therefore state the strongest valid combined result: the polynomial canonical-state hypothesis together with an independent efficient classical-simulation hypothesis for bounded-error quantum computation implies $BQP = P = NP$. Verbatim Python code performs exhaustive finite-instance checks of the exact-cover recursion, and a SymPy program verifies the algebraic QUBO-to-Ising substitution and the logical dependency of the class equality. These checks improve auditability but cannot decide either universal hypothesis. The contribution is a falsifiable separation of the classical quotient-growth obligation, the quantum-simulation obligation, and the finite computations that bear on neither as a proof.

Keywords—computational complexity; exact cover; quantum algorithms; state-space reduction; symbolic verification.

1 Introduction

Whether $P = NP$ remains open, and current complexity theory also does not establish $BQP = P$. Any manuscript asserting the combined equality must therefore separate proved implications from conjectural premises. This paper performs that separation for two supplied components: the HKD-infinity transfer–distillation reduction for Exact Cover by 3-Sets (X3C) [7], and a QAOA-compatible phrase-selection program based on a quadratic unconstrained binary optimization (QUBO) objective.

The classical component memoizes canonical residual exact-cover states. Its correctness is exact, but its polynomial running time requires a universal polynomial bound on the number of distinct residual states. The earlier paper explicitly labels that bound as unresolved [7]. The quantum component maps a finite binary objective to an Ising Hamiltonian and uses QAOA to seek a low-cost bit string [3]. QAOA is an approximation framework; its presence neither certifies a global optimum nor yields a containment between quantum and classical complexity classes.

The result is consequently conditional, but mathematically precise. It also guards against a common invalid inference: $P = NP$ would give $NP \subseteq BQP$ because $P \subseteq BQP$, but would not supply the reverse containment $BQP \subseteq P$.

2 Complexity Preliminaries

The standard containments needed here are

$$P \subseteq NP, \quad P \subseteq BQP, \quad BQP \subseteq PP \subseteq PSPACE. \quad (1)$$

The inclusion of classical polynomial-time computation in bounded-error quantum polynomial time follows by reversible

simulation. The upper bounds on BQP are classical complexity bounds, not known polynomial-time simulations [2, 1, 5].

An X3C instance is $I = (U, S)$, where $|U| = 3q$ and each member of S has cardinality three. The question is whether exactly q pairwise disjoint members cover U . X3C is NP-complete [4]. Thus a polynomial-time exact X3C algorithm would imply $P = NP$.

3 The Prior Conditional Reduction

A residual state is $X = (C, A)$, with covered elements $C \subseteq U$ and surviving compatible sets $A \subseteq S$. For an uncovered pivot u , define $A(u) = \{S \in A : u \in S\}$. Exact-cover semantics give

$$\text{SAT}(X) \iff \bigvee_{S \in A(u)} \text{SAT}(X|S). \quad (2)$$

If $|A(u)| = 1$, the sole set is forced. Both branching and propagation are exact.

Let $T(I)$ be the logical recursion tree and let $\Phi : T(I) \rightarrow Z(I)$ map histories to canonical residual states. Sound merging requires

$$\Phi(X) = \Phi(Y) \implies \text{Comp}(X) = \text{Comp}(Y), \quad (3)$$

where $\text{Comp}(X)$ is the set of exact completions below X . Memoization then solves each member of $Z(I)$ at most once.

Theorem 1 (Conditional HKD-infinity reduction). *Suppose there exist universal constants $C, k, c, r > 0$ such that, for every X3C instance of encoding length L , the canonical quotient satisfies $|Z(I)| \leq CL^k$, and each canonical transition costs at most cL^r . Then $P = NP$.*

Proof. Equation (2) exhausts all exact covers, while Equation (3) permits only semantics-preserving merges. Persistent

memoization therefore takes at most $|Z(I)|cL^r \leq CcL^{k+r}$ work. Hence X3C belongs to P. Its NP-completeness yields $\text{NP} \subseteq \text{P}$, and the reverse inclusion is standard. $\square$

The theorem is an implication, not an unconditional proof.

Its unresolved premise is the worst-case quotient-growth statement

$$\exists C, k \forall I: |\Phi(T(I))| \leq CL(I)^k. \quad (4)$$

Finite experiments cannot establish the universal quantifier in Equation (4).

4 Quantum Attachment

For binary candidates $x_i \in \{0, 1\}$, let s_i be gross benefit, d_i representation cost, and p_{ij} a pairwise overlap penalty. The supplied attachment minimizes

$$f(x) = \sum_i (d_i - s_i)x_i + \sum_{i < j} p_{ij}x_i x_j. \quad (5)$$

Substituting $x_i = (1 - Z_i)/2$ yields a diagonal Ising cost Hamiltonian. At depth p , QAOA prepares

$$|\gamma, \beta\rangle = \prod_{i=1}^p e^{-i\beta \sum_j H_j} e^{-i\gamma \sum_j H_j} \otimes_n |+\rangle, \quad H_M = \sum_i X_i. \quad (6)$$

Measurements produce candidate bit strings, while a classical optimizer varies the angles. The procedure may be simulated for small n ; neither a finite simulation nor execution on quantum hardware proves a complexity-class collapse. Known QAOA performance depends on the problem, depth, optimization, and noise [3, 6].

The QUBO can guide which states or phrases to materialize, but heuristic selection must not discard a branch required by Equation (2). It is therefore an optimization attachment to the exact solver rather than a replacement for exact enumeration unless accompanied by an independent correctness certificate.

5 Combined Conditional Statement

Define the two independent hypotheses:

H_C : Equation (4) and polynomial transition cost hold;

H_Q : $\text{BQP} \subseteq \text{P}$.

Theorem 2 (Conditional combined equality). *If $H_C \wedge H_Q$, then $\text{BQP} = \text{P} = \text{NP}$.*

Proof. By the preceding reduction, H_C gives $\text{P} = \text{NP}$. Standard reversible simulation gives $\text{P} \subseteq \text{BQP}$, while H_Q gives $\text{BQP} \subseteq \text{P}$; hence $\text{BQP} = \text{P}$. Combining the equalities proves the claim. $\square$

Proposition 1. *The hypothesis H_C alone does not establish $\text{BQP} = \text{P}$.*

Proof. The conclusion of H_C is $\text{P} = \text{NP}$. None of the containments used in that reduction provides $\text{BQP} \subseteq \text{P}$. The QUBO construction and QAOA execution are finite algorithms, not a uniform efficient classical simulation of every bounded-error quantum circuit. $\square$

Table 1: Claims and what would be required to establish them.

Claim	Required evidence
Finite X3C answer	Exhaustive oracle agrees with exact memoized recursion
QUBO mapping	Symbolic identity for all binary assignments
$\text{P} = \text{NP}$	Universal polynomial bound on canonical states
$\text{BQP} = \text{P}$	Uniform polynomial classical simulation or equivalent containment

6 Verification Protocol

The accompanying code has deliberately narrow claims. The Python verifier enumerates all subfamilies on small X3C instances, compares this oracle with the canonical memoized recursion, and reports the number of reached canonical states. Agreement checks correctness on those instances only. State counts are observations, not an asymptotic upper bound.

The SymPy program expands Equation (5) after $x_i = (1 - z_i)/2$, checks equality at every two-bit assignment, and uses Boolean formulas to show that the requested class equality

depends on both H_C and H_Q . Symbolic algebra validates the stated transformation; it does not prove either hypothesis.

7 Threats to Validity

Canonicalization can merge syntactically different histories while still leaving exponentially many semantically distinct residual states. Small or structured instances may therefore exhibit dramatic compression without controlling the worst case. Likewise, a state-vector QAOA simulation takes resources exponential in qubit count in the direct representation, and good objective values on selected instances do not demonstrate asymptotic quantum advantage. Finally, the QUBO objective represents candidate selection; it is not itself a reduction from arbitrary bounded-error quantum computation to a classical polynomial-time algorithm.

8 Conclusion

The precise defensible result is conditional. The prior HKD-infinity work reduces $\text{P} = \text{NP}$ to a universal polynomial bound on exact canonical X3C states. The QAOA attachment supplies a valid quantum optimization formulation but no proof that $\text{BQP} \subseteq \text{P}$. Assuming both missing statements does yield $\text{BQP} = \text{P} = \text{NP}$. The included exhaustive and symbolic programs make the finite algebra and solver behavior reproducible while explicitly leaving the two universal obligations open.

Acknowledgment

No external funding was declared. The author is solely responsible for the mathematical claims and the supplied software.

References

- [1] L. M. Adleman, J. DeMarrais, and M.-D. A. Huang, “Quantum computability,” *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1524–1540, 1997, doi: 10.1137/S0097539795293639.
- [2] E. Bernstein and U. Vazirani, “Quantum complexity theory,” *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1411–1473, 1997, doi: 10.1137/S0097539796300921.
- [3] E. Farhi, J. Goldstone, and S. Gutmann, “A quantum approximate optimization algorithm,” arXiv:1411.4028, 2014. [Online]. Available: <https://arxiv.org/abs/1411.4028>.
- [4] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA, USA: W. H. Freeman, 1979.
- [5] J. Watrous, “Quantum computational complexity,” in *Encyclopedia of Complexity and Systems Science*, R. A. Meyers, Ed. New York, NY, USA: Springer, 2009, pp. 7174–7201, doi: 10.1007/978-0-387-30440-3_428.
- [6] J. Weidenfeller *et al.*, “Scaling of the Quantum Approximate Optimization Algorithm on superconducting qubit based hardware,” *Quantum*, vol. 6, p. 870, 2022, doi: 10.22331/q-2022-12-07-870.
- [7] M. S. Yang, “HKD-infinity transfer–distillation reduction for exact cover,” *International Journal of Computer Techniques*, vol. 13, no. 5, pp. 345–351, 2026. [Online]. Available: IJCT article PDF.

A Inline Color-Formatted Python Code

This appendix embeds the complete reproducibility code directly in the manuscript source. It therefore remains visible even when the PDF is compiled without the companion .py files.

A.1 Finite X3C Verification

```

1  #!/usr/bin/env python3
2  """Finite X3C verification; this is not an asymptotic complexity proof."""
3  from __future__ import annotations
4
5  from functools import lru_cache
6  from itertools import combinations
7
8
9  def exhaustive_x3c(universe, sets):
10     """Definition-level oracle: enumerate every q-set subfamily."""
11     universe = frozenset(universe)
12     triples = tuple(map(frozenset, sets))
13     if len(universe) % 3:
14         return False
15     for choice in combinations(triples, len(universe) // 3):
16         if sum(map(len, choice)) == len(universe) and frozenset().union(*choice) == universe:
17             if all(a.isdisjoint(b) for i, a in enumerate(choice) for b in choice[i + 1 :]):
18                 return True
19     return False
20
21
22  def canonical_solver(universe, sets):
23     """Exact pivot recursion memoized by a canonical residual state."""
24     universe = frozenset(universe)
25     triples = tuple(sorted((tuple(sorted(s)) for s in sets)))
26
27     @lru_cache(maxsize=None)
28     def solve(uncovered, active):
29         uncovered = frozenset(uncovered)
30         active_sets = tuple(frozenset(triples[i]) for i in active)
31         if not uncovered:
32             return True
33         supports = {
34             u: tuple(i for i, s in zip(active, active_sets) if u in s)
35             for u in uncovered
36         }
37         pivot = min(uncovered, key=lambda u: (len(supports[u]), u))
38         if not supports[pivot]:
39             return False
40         for i in supports[pivot]:
41             chosen = frozenset(triples[i])
42             if not chosen <= uncovered:
43                 continue
44             next_uncovered = tuple(sorted(uncovered - chosen))
45             next_active = tuple(
46                 j for j in active
47                 if j != i and frozenset(triples[j]).isdisjoint(chosen)
48             )
49             if solve(next_uncovered, next_active):
50                 return True
51         return False
52
53     answer = solve(tuple(sorted(universe)), tuple(range(len(triples))))
54     return answer, solve.cache_info().currsize
55
56
57  def main():
58     cases = [
59         (range(6), [(0, 1, 2), (3, 4, 5)], True),
60         (range(6), [(0, 1, 2), (0, 4, 5), (1, 3, 5)], False),
61         (range(9), [(0, 1, 2), (3, 4, 5), (6, 7, 8),
62                    (0, 3, 6), (1, 4, 7), (2, 5, 8)], True),
63     ]
64     for n, (universe, sets, expected) in enumerate(cases, 1):
65         oracle = exhaustive_x3c(universe, sets)
66         memoized, states = canonical_solver(universe, sets)
67         assert oracle == memoized == expected
68         print(f"case={n} answer={oracle} canonical_states={states}")

```

```

69 print("finite verification passed; no universal polynomial bound was tested")
70
71
72 if __name__ == "__main__":
73     main()

```

Listing 1: Exact finite-instance checker with canonical memoization.

A.2 SymPy Algebra and Logic Verification

```

1  #!/usr/bin/env python3
2  """Symbolic checks for the QUBO attachment and conditional class logic."""
3  from itertools import product
4
5  from sympy import And, Equivalent, Implies, Not, symbols, simplify, expand
6  from sympy.logic.inference import satisfiable
7
8
9  def verify_qubo_to_ising():
10     a0, a1, p01 = symbols("a0 a1 p01", real=True)
11     x0, x1, z0, z1 = symbols("x0 x1 z0 z1")
12     qubo = a0 * x0 + a1 * x1 + p01 * x0 * x1
13     ising = expand(qubo.subs({x0: (1 - z0) / 2, x1: (1 - z1) / 2}))
14     expected = expand(
15         (a0 / 2 + a1 / 2 + p01 / 4)
16         + (-a0 / 2 - p01 / 4) * z0
17         + (-a1 / 2 - p01 / 4) * z1
18         + (p01 / 4) * z0 * z1
19     )
20     assert simplify(ising - expected) == 0
21     for b0, b1 in product((0, 1), repeat=2):
22         lhs = qubo.subs({x0: b0, x1: b1})
23         rhs = ising.subs({z0: 1 - 2 * b0, z1: 1 - 2 * b1})
24         assert simplify(lhs - rhs) == 0
25     print("QUBO-to-Ising identity verified for symbolic coefficients")
26
27
28  def verify_conditional_logic():
29     hc, hq, p_eq_np, bqp_subset_p, p_subset_bqp = symbols(
30         "H_C H_Q P_eq_NP BQP_subset_P P_subset_BQP"
31     )
32     premises = And(Implies(hc, p_eq_np), Equivalent(hq, bqp_subset_p), p_subset_bqp)
33     target = And(p_eq_np, bqp_subset_p, p_subset_bqp)
34     assert satisfiable(And(premises, hc, hq, Not(target))) is False
35     # H_C alone permits BQP not to be contained in P.
36     countermodel = satisfiable(And(premises, hc, Not(hq), Not(bqp_subset_p)))
37     assert countermodel is not False
38     print("conditional dependency verified: H_C and H_Q are both required")
39
40
41  if __name__ == "__main__":
42     verify_qubo_to_ising()
43     verify_conditional_logic()

```

Listing 2: Symbolic QUBO-to-Ising and conditional-logic checks.

A.3 Supplied QAOA Attachment

```

1  #!/usr/bin/env python3
2  """QAOA origin model for profitable phrase selection."""
3  from __future__ import annotations
4
5  from dataclasses import dataclass
6
7
8  @dataclass(frozen=True)
9  class Candidate:
10     phrase: bytes
11     gross_saving: int
12     dictionary_cost: int
13
14

```

```

15 def build_qubo(candidates: list[Candidate], overlap_penalty: dict[tuple[int, int], int]):
16     """Return QUBO coefficients for min x'Qx."""
17     q: dict[tuple[int, int], float] = {}
18     for i, c in enumerate(candidates):
19         q[i, i] = float(c.dictionary_cost - c.gross_saving)
20     for (i, j), penalty in overlap_penalty.items():
21         key = (min(i, j), max(i, j))
22         q[key] = q.get(key, 0.0) + penalty
23     return q
24
25
26 def qaoa_select(q: dict[tuple[int, int], float], n: int, reps: int = 2):
27     """Execute QAOA with Qiskit; return selected bits and objective value."""
28     try:
29         from qiskit.primitives import StatevectorSampler
30         from qiskit_optimization import QuadraticProgram
31         from qiskit_optimization.algorithms import MinimumEigenOptimizer
32         from qiskit_algorithms.minimum_eigensolvers import QAOA
33         from qiskit_algorithms.optimizers import COBYLA
34     except ImportError as exc:
35         raise SystemExit(
36             "Install qiskit, qiskit-algorithms, and qiskit-optimization."
37         ) from exc
38
39     qp = QuadraticProgram("hkd_dictionary_selection")
40     for i in range(n):
41         qp.binary_var(f"x_{i}")
42     linear = {f"x_{i}": q.get((i, i), 0.0) for i in range(n)}
43     quadratic = {
44         (f"x_{i}", f"x_{j}"): value
45         for (i, j), value in q.items() if i != j
46     }
47     qp.minimize(linear=linear, quadratic=quadratic)
48     result = MinimumEigenOptimizer(
49         QAOA(
50             sampler=StatevectorSampler(),
51             optimizer=COBYLA(maxiter=120),
52             reps=reps,
53         )
54     ).solve(qp)
55     return [int(round(v)) for v in result.x], float(result.fval)
56
57
58 if __name__ == "__main__":
59     demo = [
60         Candidate(b"quantum", 120, 12),
61         Candidate(b"compression", 180, 16),
62     ]
63     qubo = build_qubo(demo, {(0, 1): 8})
64     print(qaoa_select(qubo, len(demo)))

```

Listing 3: QAOA model for QUBO phrase selection.