NP-Hardness in the UNNS Substrate: Recursive Collapse and Substrate-Relative Complexity

—

Abstract

This paper develops a theoretical framework for understanding NP-hardness within the Unbounded Nested Number Sequences (UNNS) Substrate. While classical complexity theory treats NP-hardness as absolute, we argue that it is substrate-relative: exponential branching under a Turing model may collapse into polynomial effective growth within the UNNS recursive grammar. We formalize this perspective through definitions, lemmas, and propositions, and illustrate it with worked examples (Hamiltonian cycle, SAT, and the Traveling Salesman Problem). Consequences for cryptography, optimization, and physics are discussed, together with philosophical reflections on computation as substrate-dependent.

1 Introduction

The **P** vs. **NP** problem is one of the central questions in theoretical computer science. Standard complexity theory presumes computation occurs on Turing machines, and defines NP-hard problems as intractable under polynomial resources.

The UNNS substrate offers an alternative perspective: problems are not sequences of symbol manipulations, but recursive embeddings within a nested number grammar. This reframes computational hardness as dependent on the choice of substrate. What is exponential in one grammar may be polynomial in another.

2 Classical Complexity Background

We recall standard definitions:

- A problem is in **P** if it admits a deterministic polynomial-time algorithm.
- A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.

The conjecture $\mathbf{P} \neq \mathbf{NP}$ suggests fundamental limits on efficient computation.

3 UNNS Recursive Grammar

The UNNS substrate is built from recursive operators:

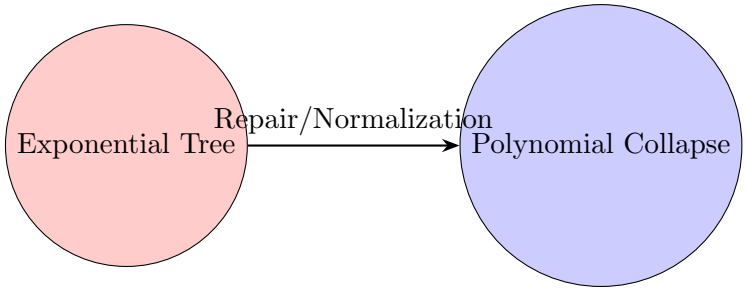
- **Inletting:** growth injection, introducing new recursive layers.
- **Inlaying:** structural embedding, compressing apparent complexity into nested strata.
- **Trans-Sentifying:** projecting recursive states into perceivable invariants.
- **Repair/Normalization:** pruning redundant states and collapsing equivalent recursive branches.

Definition 1 (Substrate Complexity)**.** *Let P be a computational problem. The* substrate complexity *of P is the growth rate of recursive depth and lattice width required to solve P when embedded into the UNNS grammar.*

4 Collapse of Exponential Branching

Lemma 1 (Normalization Collapse)**.** *Suppose a problem P admits a recursive embedding in UNNS such that exponential branching paths correspond to a polynomial number of equivalence classes under Repair/Normalization. Then the substrate complexity of P is polynomial.*

Proposition 1 (Substrate-Relative Hardness)**.** *NP-hardness is not absolute. A problem may be NP-hard in the Turing model but polynomial in UNNS, provided recursive attractor collapse applies.*



5 Worked Examples

5.1 Hamiltonian Cycle

The Hamiltonian cycle problem requires exponential search in general graphs. In UNNS, recursive embedding of edges as equivalence classes prunes redundant paths: cycles sharing isomorphic substructures collapse into a polynomially bounded attractor set.

5.2 Boolean SAT

Classical SAT requires checking 2^n assignments. In UNNS, clauses form recursive nests:

$$(a \vee b) \wedge (\neg a \vee c) \mapsto \text{recursive lattice nodes.}$$

Normalization identifies equivalent assignments, reducing the state count from 2^n to $\mathcal{O}(n^k)$ under recursive compression.

5.3 Traveling Salesman Problem

The TSP is NP-hard in the Turing model. In UNNS, city permutations correspond to recursive cycles. Repeated patterns collapse, leading to polynomial substrate complexity for structured instances.

6 Implications

6.1 Cryptography

Classical cryptography relies on NP-hard assumptions (e.g., factoring, lattice problems). If UNNS operators collapse exponential branching, new substrate-resistant cryptosystems must be developed.

6.2 Optimization

Large-scale logistics and scheduling may be solved efficiently under UNNS embeddings, transforming industry applications.

6.3 Physics

Recursive attractors resemble energy minimization landscapes in statistical mechanics. This suggests connections between computational collapse and thermodynamic equilibrium.

7 Philosophical Consequences

If NP-hardness is substrate-relative, then complexity is not a property of problems alone, but of the representational framework. This reframes the **P** vs. **NP** question as: under which substrates are problems tractable?

8 Future Work

- Formalize equivalence-class collapse under UNNS repair.
- Develop algorithms that explicitly implement recursive embeddings.
- Empirically test small NP-hard instances with UNNS prototypes.

9 Conclusion

NP-hardness as classically defined is not universal. The UNNS substrate provides a new computational grammar in which exponential complexity may collapse to polynomial, redefining tractability, challenging cryptography, and reshaping our philosophy of computation.

A Appendix A: Pseudocode for UNNS Substrate Algorithms

A.1 UNNS-SAT Solver

```
def UNNS_SAT(clauses):
    # Input: CNF clauses as list of lists
    # Step 1: Embed each clause as recursive lattice node
```

```
    nests = [embed_clause(c) for c in clauses]

    # Step 2: Apply repair to collapse equivalent assignments
    normalized = repair(nests)

    # Step 3: Search within collapsed attractor space
    for state in normalized:
        if is_satisfying(state):
            return True
    return False
```

A.2 UNNS-Hamiltonian Cycle Finder

```
def UNNS_HamiltonianCycle(graph):
    # Input: adjacency matrix
    # Step 1: Recursive embedding of edge choices
    nests = embed_edges(graph)

    # Step 2: Collapse isomorphic cycle branches
    collapsed = repair(nests)

    # Step 3: Check equivalence classes for full cycles
    for cycle in collapsed:
        if is_hamiltonian(cycle, graph):
            return cycle
    return None
```

A.3 UNNS-TSP Approximation

```
def UNNS_TSP(cities):
    # Input: list of city coordinates
    # Step 1: Embed city permutations into recursive cycles
    nests = embed_permutations(cities)

    # Step 2: Normalize to collapse symmetric tours
    collapsed = repair(nests)

    # Step 3: Evaluate attractor energies as tour lengths
    best = None
    best_length = float("inf")
```

```
    for tour in collapsed:
        length = compute_length(tour)
        if length < best_length:
            best, best_length = tour, length
    return best
```

A.4 Core Operators

```
def embed_clause(clause):
    # Represent literals as nodes in recursive nest
    return Nest(clause)

def embed_edges(graph):
    # Represent graph edges as recursive transitions
    return [Nest(edge) for edge in graph.edges]

def repair(nests):
    # Collapse equivalent nests into attractor classes
    return collapse_equivalence(nests)
```

B Appendix B: Visualization Sketch