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

RESEARCH ARTICLE OPEN ACCESS

HKD∞ Incremental Nash: Exact Equilibrium Computation with Linear-in-N Cycle Reduction in Sparse Potential Games

Dependency-Localized Regret Maintenance, Exact-Cover Parallel Deviations, and Verified Exact Equilibria

Michael S. Yang
Independent Researcher
yangofzeal@gmail.com

Abstract

This paper studies exact pure Nash-equilibrium computation in sparse binary-action exact-potential graphical games. A conventional deterministic best-improvement implementation recomputes every player's unilateral deviation gain after every accepted move. The HKD∞ incremental formulation instead maintains exact regret values and invalidates only the deviating player and its graph neighbors. For bounded interaction degree d and T accepted deviations, the resulting gain-evaluation count is $N + (d+1)T$, compared with $N(T+1)$ for a full-rescan implementation. Thus, when $d = O(1)$ and $T = \Theta(N)$, the operation-count reduction grows linearly with N while preserving the identical deterministic equilibrium trajectory.

A reproducible Python benchmark over $N = 250, 500, 1000, 2000$, and 4000 players verifies zero positive unilateral regret at termination, identical final strategy profiles, identical flip counts, and cycle-count reductions from 26.23× to 408.81×. On the author's reported Python 3.10 run, wall-clock speedup rises from 20.46× to 289.68×, with the 4000-player case decreasing from 4.026344 s to 0.013899 s. We additionally describe an exact-cover layer for selecting compatible profitable deviations in parallel; this layer is not included in the reported timing table and therefore does not affect the measured claims. The public reproduction code contains no private or paid HKD implementation details and independently checks exact equilibrium conditions.

Keywords: Nash equilibrium; potential games; graphical games; incremental algorithms; sparse computation; exact equilibrium; HKD∞; exact cover.

1 Introduction

Nash equilibrium is a foundational solution concept for noncooperative games. Nash established existence of equilibrium points in finite n-person games [1]. The computational question is subtler: general Nash-equilibrium computation is not known to admit a polynomial-time solution, and broad exact claims must therefore be stated with care. This paper does not claim an algorithmic breakthrough for unrestricted normal-form games. Instead, it isolates a structured class in which exactness, sparse dependence, and incremental state can be exploited transparently.

The class considered here consists of binary-action graphical games whose incentives admit an exact potential. Potential games provide a scalar function whose unilateral change agrees with a player's incentive change [2]. That property makes deterministic best-improvement dynamics an exact equilibrium-finding process: every accepted move strictly raises the finite potential, so the process terminates at a pure Nash equilibrium.

The key engineering observation is locality. In a sparse graphical game, a move by player i cannot alter every other player's unilateral gain; it changes only the gain of i and players whose payoff neighborhoods include i. Graphical games were introduced specifically to encode such local payoff dependence compactly [3]. The benchmark therefore compares two implementations of the same deterministic best-improvement rule: a full-rescan baseline and a dependency-localized incremental solver called HKD∞ in this paper.

This distinction is important because the general Nash problem remains computationally hard. PPAD-completeness

contribution here is narrower and directly testable: on bounded-degree exact-potential graphical games, maintain exact gains only where dependency changes make recomputation necessary, prove preservation of the baseline path, and measure the resulting operation-count and wall-clock reductions.

For context, mature packages such as Gambit provide a broad collection of equilibrium-computation methods for finite games [5]. The present benchmark is not a head-to-head claim against every Gambit algorithm or every game representation. Its baseline is the explicit full-rescan implementation of the same best-improvement dynamics, which allows the incremental work reduction and exact path equivalence to be measured without conflating different equilibrium-selection rules.

2 Game Class and Exact Equilibrium Condition

Let $G = (V, E)$ be an undirected interaction graph with $|V| = N$. Each player i chooses a binary action $x_i \in \{0, 1\}$. The benchmark uses an integer-valued exact potential

$$\Phi(x) = \sum_i a_i x_i + \sum_{(i,j)\in E} J_{ij} \mathbf{1}[x_i = x_j],$$

where a_i and J_{ij} are bounded integers. Player utilities can be chosen so that each unilateral utility difference equals the corresponding potential difference. Therefore

$$\Delta_i(x) = \Phi(1 - x_i, x_{-i}) - \Phi(x_i, x_{-i})$$

is an exact regret/improvement signal.

A pure strategy profile x is a pure Nash equilibrium exactly when

$$\Delta_i(x) \le 0 \qquad \text{for every player } i.$$

Because all coefficients and actions are integral, every Δ_i is evaluated exactly with integer arithmetic. No floating-point tolerance or ϵ-equilibrium convention is required. The public verifier independently recomputes all N unilateral gains from the final profile and requires

$$\max_i \Delta_i(x) \le 0.$$

The generated graphs have bounded degree six. Random seeds determine integer bias terms, integer symmetric interaction weights, and the initial binary strategy profile. The same game and same starting profile are passed to both solvers. A deterministic tie-break chooses the smallest player index among equal positive gains, so path identity is a meaningful and reproducible invariant.

3 Full-Rescan and HKD∞ Incremental Dynamics

The baseline performs a complete scan after every accepted move. At each iteration it evaluates Δ_i for all N players, selects the largest positive gain using the fixed tie-break, flips that player, and repeats. If T moves are accepted before equilibrium, the exact number of gain evaluations is

$$C_{\text{full}} = N(T + 1),$$

including the final scan that proves no positive deviation remains.

HKD∞ begins with the same N gain evaluations but stores the results in a versioned priority queue. When player i moves, only i and its graph neighbors can have changed gain values. All unaffected entries remain valid. If the graph degree is bounded by d, the gain-evaluation count obeys

$$C_{\text{HKD}} \le N + (d + 1)T.$$

Thus

$$\frac{C_{\text{full}}}{} = \frac{N(T+1)}{}.$$

$$C_{\text{full}} = \Theta(N^2), \qquad C_{\text{HKD}} = \Theta(N),$$

giving a $\Theta(N)$ reduction in gain-evaluation cycles. This is a work-reduction statement, not a claim that disk or any physical memory tier is intrinsically faster than RAM.

4 Exact Path-Preservation Theorem

Theorem 1 (Localized exactness)**.** *Consider a binary-action graphical exact-potential game and a deterministic best-improvement rule with a fixed total tie-break order. Suppose an incremental implementation stores exact unilateral gains and, after a move by player i, recomputes precisely every player whose unilateral gain can depend on x_i, while leaving all other stored gains unchanged. Then the incremental implementation selects the same next player as a full rescan at every iteration, terminates after the same number of moves, and returns the identical pure Nash equilibrium profile.*

Proof. Initially both methods compute the same exact gain vector. Assume their profiles and valid gain values agree immediately before an iteration. If player i is selected and flipped, any player j whose payoff neighborhood is independent of x_i has exactly the same unilateral gain before and after the move, so retaining its stored value is exact. Every player whose gain may depend on x_i is recomputed exactly. Thus the localized data structure represents precisely the gain vector that a full rescan would compute.

Because both methods use the same maximization and tie-break rule, they select the same next player. Induction establishes identical trajectories. The finite exact potential strictly increases on every accepted positive-gain move, so termination occurs. At termination no positive unilateral gain exists; hence the common terminal profile is a pure Nash equilibrium. □

5 Exact-Cover Parallel Deviation Layer

The localized theorem naturally supports an additional batching layer. Let P be the current set of positive-gain players. Associate with each candidate i a dependency footprint F_i containing i and every state component needed to keep i's precomputed gain valid. Two candidates conflict if their footprints overlap in a way that makes simultaneous execution invalidate either stored gain. Selecting a maximum-weight compatible subset of P can then be expressed as a weighted set-packing/exact-cover-style problem, using gain or another deterministic priority as the weight.

An exact-cover solver is useful here as a selection layer, not as a source of Nash equilibrium existence or game-theoretic alpha. If it returns a set of pairwise noninterfering deviations, those deviations can be independently verified against the public conflict relation before execution. A paid hkd_exact_cover implementation may accelerate this selection in a private environment, but its implementation details are neither needed nor exposed by the public reproduction program.

Crucially, the timing results in Section 7 were obtained from the sequential localized priority-queue implementation and do not include the exact-cover batch layer. Consequently, the measured 20.46x-289.68x wall-clock values and 26.23x-408.81x cycle reductions stand without attributing any unmeasured performance to the paid exact-cover component.

6 Experimental Method

The reproducibility program uses Python 3.10-compatible standard-library components only. For each $N \in \{250, 500, 1000, 2000, 4000\}$, three games are generated from seeds 11, 12, and 13. Each game has a sparse ring-like symmetric degree-six interaction graph, bounded integer coefficients, and a reproducible initial binary profile. Both solvers run from an independent copy of the identical initial profile.

For each size, the program reports the median elapsed time, median accepted flips, median baseline gain-evaluation count, and median localized gain-evaluation count across the three seeds. Exactness requires all three runs to satisfy four checks simultaneously: baseline maximum regret is nonpositive, localized maximum regret is nonpositive, the final strategy profiles are identical, and the accepted flip counts are identical.

game generator, seeds, and deterministic rule, the reported flip counts and gain-evaluation counts are algorithmic observables. The public code was sanitized for distribution by removing any paid/private HKD API references; the mathematical localized-update procedure itself remains visible because it is necessary for reproducibility. A rerun of the sanitized source reproduced all flip counts, cycle counts, cycle ratios, and exact=True outcomes; timings varied as expected with the execution environment.

7 Results

The author's supplied Python 3.10 run produced the measurements in Table 1. Every tested size returned exact=True. The operation-count ratio grows nearly proportionally with N: 26.23x, 50.85x, 103.42x, 203.02x, and 408.81x as N doubles from 250 to 4000. At $N = 4000$, the full-rescan implementation performs 5,736,000 unilateral-gain evaluations, whereas HKD∞ performs 14,031, while both execute 1,433 accepted deviations and terminate at the identical zero-positive-regret equilibrium.

Measured wall-clock speedup also increases with N, reaching 289.68x in the 4000-player run: 4.026344 seconds for the full rescan versus 0.013899 seconds for the localized solver. Timing growth is not itself the theorem; the stronger reproducible structural result is the exact operation-count reduction coupled with path identity and independent equilibrium verification.

Table 1: Exact equilibrium benchmark reported by the author.

N	flips	base s	HKD∞ s	speedup	base cycles	HKD cycles	cycle x	exact
250	95	0.017031	0.000832	20.46x	24000	915	26.23x	True
500	173	0.059730	0.001584	37.71x	87000	1711	50.85x	True
1000	371	0.259203	0.003504	73.97x	372000	3597	103.42x	True
2000	698	0.977364	0.006706	145.75x	1398000	6886	203.02x	True
4000	1433	4.026344	0.013899	289.68x	5736000	14031	408.81x	True

8 Limitations and Claim Boundary

The results apply to the benchmarked class of sparse binary-action exact-potential graphical games and to deterministic best-improvement dynamics. They do not establish a faster algorithm for arbitrary mixed Nash equilibria, arbitrary normal-form games, or unrestricted graphical games. The theorem depends on exact dependency localization: if a player's gain can change because of a remote state variable, that dependency must be included in the affected set.

The full-rescan baseline is intentionally simple and algorithmically matched to HKD∞. A broader systems paper should add optimized compiled implementations, additional graph families and degrees, adversarial potential landscapes, larger action sets, and comparisons with established game-theory packages where equivalent solution concepts can be enforced. Such comparisons must distinguish different equilibrium-selection rules and representations from the incremental-vs-rescan question isolated here.

The exact-cover batching layer is presented as an extension and verification-compatible acceleration opportunity; no exact-cover timing is included in the headline table. Any future claim for that layer should report its own wall-clock, operation count, solution verification, and comparison baseline.

9 Conclusion

Sparse dependence can change the computational shape of exact equilibrium dynamics even when it does not change the underlying game-theoretic solution concept. For bounded-degree exact-potential graphical games, a full rescan spends work on N players after each local move, whereas HKD∞ retains unaffected exact gains and refreshes only the changed dependency neighborhood. The resulting count $N + (d + 1)T$ versus $N(T + 1)$ yields a linear-in-N cycle reduction when d is bounded and T grows linearly with N.

The reported benchmark verifies this scaling experimentally through 4000 players, culminating in a 408.81x reduction

deviations, reach the identical strategy profile, and independently satisfy zero positive unilateral regret. The public reproduction program exposes the theorem-level algorithm while withholding unrelated private HKD and paid exact-cover implementation details.

Reproducibility and Code Availability

The accompanying file `hkd_inf_nash_benchmark_public.py` is the public reference implementation. It contains no paid `hkd_exact_cover` implementation, license material, private HKD package API, model state, or other proprietary acceleration machinery. Reviewers should rerun wall-clock measurements on their own hardware.

10 Public Reproduction Program

For independent reproduction, the complete public benchmark used to implement the experiment is reproduced below as `hkd_inf_nash_benchmark_public.py`. The program uses only the Python standard library. It constructs the benchmark games, runs both the full-rescan and dependency-localized implementations, independently verifies the terminal unilateral gains, checks equality of the final profiles and flip counts, and reports both wall-clock and gain-evaluation counts. No paid or private HKD implementation is required to run this publication benchmark.

Listing 1: Complete public reproduction program: hkd_inf_nash_benchmark_public.py

```
import random, time, heapq, statistics

def make_game(n, deg=6, seed=1):
    rng=random.Random(seed)
    a=[rng.randint(-7,7) for _ in range(n)]
    nbr=[{} for _ in range(n)]
    # ring-like guaranteed degree-ish sparse graph, avoid dupes
    for i in range(n):
        for k in range(1, deg//2+1):
            j=(i+k)%n
            if j==i or j in nbr[i]: continue
            w=rng.randint(-5,5)
            if w==0: w=1
            nbr[i][j]=w; nbr[j][i]=w
    x=[rng.randrange(2) for _ in range(n)]
    return a,nbr,x

def delta(i,x,a,nbr):
    # Phi = sum a_i x_i + sum_(i,j) J_ij * 1[x_i==x_j]
    old=x[i]; new=1-old
    d=a[i]*(new-old)
    for j,w in nbr[i].items():
        d += w*((1 if new==x[j] else 0)-(1 if old==x[j] else 0))
    return d

def verify(x,a,nbr):
    ds=[delta(i,x,a,nbr) for i in range(len(x))]
    return max(ds), ds

def baseline(a,nbr,x0):
    x=x0[:]; n=len(x); cycles=0; flips=0
    t=time.perf_counter()
    while True:
        best_d=0; best_i=None
        for i in range(n):
            d=delta(i,x,a,nbr); cycles+=1
            if d>best_d or (d==best_d and d>0 and (best_i is None or i<best_i)):
                best_d=d; best_i=i
        if best_i is None: break
        x[best_i]^=1; flips+=1
    return x,flips,cycles,time.perf_counter()-t

def hkd(a,nbr,x0):
    x=x0[:]; n=len(x); cycles=0; flips=0
    vers=[0]*n; heap=[]
    t=time.perf_counter()
    for i in range(n):
        d=delta(i,x,a,nbr); cycles+=1
        heapq.heappush(heap,(-d,i,0))
    while True:
        while heap:
            nd,i,v=heapq.heappop(heap)
```

```
                d=-nd; break
        else: break
        if d<=0: break
        x[i]^=1; flips+=1
        affected=[i]+list(nbr[i].keys())
        for j in affected:
            vers[j]+=1
            dj=delta(j,x,a,nbr); cycles+=1
            heapq.heappush(heap,(-dj,j,vers[j]))
    return x,flips,cycles,time.perf_counter()-t

for n in [250,500,1000,2000,4000]:
    rows=[]
    for seed in [11,12,13]:
        a,nbr,x=make_game(n,6,seed)
        b=baseline(a,nbr,x)
        h=hkd(a,nbr,x)
        vb=verify(b[0],a,nbr)[0]; vh=verify(h[0],a,nbr)[0]
        rows.append((b,h,vb,vh,b[0]==h[0]))
    btime=statistics.median(r[0][3] for r in rows)
    htime=statistics.median(r[1][3] for r in rows)
    bcy=statistics.median(r[0][2] for r in rows)
    hcy=statistics.median(r[1][2] for r in rows)
    flips=statistics.median(r[0][1] for r in rows)
    print(n, 'flips',flips,'baseline_s',f'{btime:.6f}','hkd_s',f'{htime:.6f}', 'speedup',f'{btime/htime:.2f}', 'base_cycles',int(bcy),'
      hkd_cycles',int(hcy),'cycle_x',f'{bcy/hcy:.2f}', 'exact', all(r[2]<=0 and r[3]<=0 and r[4] and r[0][1]==r[1][1] for r in rows))
```

References

[1] J. F. Nash, Jr., “Equilibrium Points in n-Person Games,” *Proceedings of the National Academy of Sciences*, vol. 36, no. 1, pp. 48–49, 1950. doi:10.1073/pnas.36.1.48. https://doi.org/10.1073/pnas.36.1.48

[2] D. Monderer and L. S. Shapley, “Potential Games,” *Games and Economic Behavior*, vol. 14, no. 1, pp. 124–143, 1996. doi:10.1006/game.1996.0044. https://doi.org/10.1006/game.1996.0044

[3] M. Kearns, M. L. Littman, and S. Singh, “Graphical Models for Game Theory,” *Proc. 17th UAI*, pp. 253–260, 2001. https://www.cis.upenn.edu/~mkearns/papers/old-graphgames.pdf

[4] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou, “The Complexity of Computing a Nash Equilibrium,” *SIAM Journal on Computing*, vol. 39, no. 1, pp. 195–259, 2009. doi:10.1137/070699652. https://doi.org/10.1137/070699652

[5] R. Savani and T. L. Turocy, “Gambit: The package for computation in game theory,” Version 16.1.0, 2023. https://gambitproject.readthedocs.io/en/v16.1.0/