```python
"""
Connect 4 Data Generation - SPEED OPTIMIZED
============================================
~10-20x faster than colab_data_generation_FINAL.py via:
  1. Numba @njit for rollouts, win checks, legal moves (C-speed)
  2. board.tobytes() dict keys (vs tuple(board.ravel()))
  3. math.sqrt/log for scalar ops (vs numpy)
  4. In-place board mods with undo (vs .copy() everywhere)
  5. int8 board dtype (42-byte keys vs 336-byte)
  6. imap_unordered for load balancing

Same zero-leakage guarantees: dedup across all 4 transformations
(original, mirror, perspective flip, mirror+flip).
"""

import numpy as np
import os
import time
import random
import math
import hashlib
import sys
from collections import Counter, defaultdict
from multiprocessing import Pool, cpu_count
import pickle

try:
    from numba import njit
    HAS_NUMBA = True
except ImportError:
    HAS_NUMBA = False
    def njit(*args, **kwargs):
        if args and callable(args[0]):
            return args[0]
        return lambda f: f

# ============================================================================
# NUMBA-JIT GAME ENGINE
# ============================================================================

@njit(cache=True)
def find_row(board, col):
    """Find lowest empty row in column. Returns -1 if full."""
    for r in range(5, -1, -1):
        if board[r, col] == 0:
            return r
    return -1

@njit(cache=True)
def find_legal(board):
    """Return array of legal column indices."""
    moves = np.empty(7, dtype=np.int32)
    n = 0
    for c in range(7):
        if board[0, c] == 0:
            moves[n] = c
            n += 1
    return moves[:n]
```

```
@njit(cache=True)
def check_win(board, row, col):
    """Check if piece at (row, col) forms 4-in-a-row."""
    p = board[row, col]
    if p == 0:
        return False

    # Vertical
    if row <= 2:
        if board[row+1,col]==p and board[row+2,col]==p and board[row+3,col]==p:
            return True

    # Horizontal
    count = 0
    for c in range(7):
        if board[row, c] == p:
            count += 1
            if count >= 4:
                return True
        else:
            count = 0

    # Diagonal backslash
    count = 0
    sr = row - min(row, col)
    sc = col - min(row, col)
    while sr < 6 and sc < 7:
        if board[sr, sc] == p:
            count += 1
            if count >= 4:
                return True
        else:
            count = 0
        sr += 1
        sc += 1

    # Diagonal slash
    count = 0
    sr = row + min(5 - row, col)
    sc = col - min(5 - row, col)
    while sr >= 0 and sc < 7:
        if board[sr, sc] == p:
            count += 1
            if count >= 4:
                return True
        else:
            count = 0
        sr -= 1
        sc += 1

    return False

@njit(cache=True)
def try_win(board, col, player):
    """Test if playing player at col wins. Restores board after."""
    row = find_row(board, col)
    if row < 0:
        return False
    board[row, col] = player
    won = check_win(board, row, col)
    board[row, col] = 0
    return won

@njit(cache=True)
def rollout_jit(board, next_player):
    """
    Heuristic rollout to game end. Returns winner: 1, -1, or 0 (tie).
    Checks winning/blocking moves before random play (same as professor's).
    """
    b = board.copy()
    p = next_player

    for _ in range(42):
        legal = find_legal(b)
        if len(legal) == 0:
            return 0

        opp = -p
        chosen = np.int32(-1)

        # 1. Winning move?
        for i in range(len(legal)):
            if try_win(b, legal[i], p):
                return p

        # 2. Blocking move?
        for i in range(len(legal)):
            if try_win(b, legal[i], opp):
                chosen = legal[i]
                break

        # 3. Center bias or random
        if chosen == -1:
            has_center = False
            for i in range(len(legal)):
                if legal[i] == 3:
                    has_center = True
                    break
            if has_center and np.random.random() < 0.2:
                chosen = np.int32(3)
            else:
                chosen = legal[np.random.randint(len(legal))]

        row = find_row(b, chosen)
        b[row, chosen] = p
        if check_win(b, row, chosen):
            return p
        p = opp

    return 0


# ==============================================================================
# OPTIMIZED MCTS (professor's algorithm, integer-based)
# ==============================================================================

def mcts_fast(board, color_val, nsteps):
    """
```

```
MCTS with nsteps rollouts. Uses professor's UCB1 algorithm.
board: 6x7 int8 array, values {-1, 0, 1}
color_val: 1 or -1 (current player)
Returns: best column (0-6)
"""
# Immediate win
legal0 = find_legal(board)
for c in legal0:
    if try_win(board, int(c), color_val):
        return int(c)

# Find non-losing moves
opp = -color_val
allowed = []
for c in legal0:
    col = int(c)
    row = int(find_row(board, col))
    board[row, col] = color_val

    opp_legal = find_legal(board)
    loses = False
    for oc in opp_legal:
        if try_win(board, int(oc), opp):
            loses = True
            break

    board[row, col] = 0
    if not loses:
        allowed.append(col)

if not allowed:
    allowed = [int(c) for c in legal0]

# MCTS search tree: tobytes_key -> [visits, value]
root_key = board.tobytes()
md = {root_key: [0, 0]}

_sqrt = math.sqrt
_log = math.log

for _ in range(nsteps):
    cv = color_val
    b = board.copy()
    path = [root_key]

    while True:
        legal = find_legal(b)
        n_legal = len(legal)
        if n_legal == 0:
            # Draw
            for key in path:
                md[key][0] += 1
            break

        # Generate child keys (in-place + undo)
        keys = [None] * n_legal
        for i in range(n_legal):
            col = int(legal[i])
            row = int(find_row(b, col))
            b[row, col] = cv
            k = b.tobytes()
            keys[i] = k
            if k not in md:
                md[k] = [0, 0]
            b[row, col] = 0

        # UCB1 selection
        parent_n = md[path[-1]][0]
        best_ucb = -1e30
        best_i = 0

        if parent_n == 0:
            best_i = 0
        else:
            log_p = _log(parent_n)
            for i in range(n_legal):
                nd = md[keys[i]]
                if nd[0] == 0:
                    best_i = i
                    break
                ucb = nd[1] / nd[0] + 2.0 * _sqrt(log_p / nd[0])
                if ucb > best_ucb:
                    best_ucb = ucb
                    best_i = i

        # Make chosen move
        chosen_col = int(legal[best_i])
        row = int(find_row(b, chosen_col))
        b[row, chosen_col] = cv
        path.append(keys[best_i])

        # Win during tree traversal?
        if check_win(b, row, chosen_col):
            winner = cv
            for j, key in enumerate(path):
                md[key][0] += 1
                if winner == color_val:
                    md[key][1] += 1 if j % 2 == 1 else -1
                else:
                    md[key][1] += -1 if j % 2 == 1 else 1
            break

        cv = -cv

        # Unvisited node -> rollout
        if md[keys[best_i]][0] == 0:
            result = int(rollout_jit(b, cv))
            for j, key in enumerate(path):
                md[key][0] += 1
                if result == 0:
                    pass
                elif result == color_val:
                    md[key][1] += 1 if j % 2 == 1 else -1
                else:
                    md[key][1] += -1 if j % 2 == 1 else 1
            break

# Select best move
```

```
    best_val = -1e30
    best_col = allowed[0]
    for col in allowed:
        row = int(find_row(board, col))
        board[row, col] = color_val
        k = board.tobytes()
        board[row, col] = 0

        if k in md and md[k][0] > 0:
            val = md[k][1] / md[k][0]
            if val > best_val:
                best_val = val
                best_col = col

    return best_col


# ==============================================================================
# DATA GENERATION
# ==============================================================================

def board_to_canonical(board, current_player_val):
    """Convert int8 board to (6,7,2) float32. Ch0=current, Ch1=opponent."""
    canonical = np.zeros((6, 7, 2), dtype=np.float32)
    canonical[:, :, 0] = (board == current_player_val).astype(np.float32)
    canonical[:, :, 1] = (board == -current_player_val).astype(np.float32)
    return canonical


def play_single_game(args):
    """Play one MCTS self-play game. Returns list of (board, move, metadata)."""
    game_id, mcts_rollouts, random_opening_moves = args

    board = np.zeros((6, 7), dtype=np.int8)
    cv = np.int8(1)  # player 1 starts
    game_data = []

    # Random opening for diversity
    n_random = random.randint(0, random_opening_moves)
    for _ in range(n_random):
        legal = find_legal(board)
        if len(legal) == 0:
            break
        col = int(random.choice(legal))
        row = int(find_row(board, col))
        board[row, col] = cv
        if check_win(board, row, col):
            return []  # ended during opening
        cv = np.int8(-cv)

    # MCTS self-play
    recorded = 0
    while True:
        legal = find_legal(board)
        if len(legal) == 0:
            break

        move = mcts_fast(board, int(cv), mcts_rollouts)
        # Record BEFORE executing
        canonical = board_to_canonical(board, int(cv))
        game_data.append((canonical, move, {
            'game_id': game_id,
            'move_number': recorded,
        }))
        recorded += 1

        # Execute
        row = int(find_row(board, move))
        board[row, move] = cv
        if check_win(board, row, move):
            break

        cv = np.int8(-cv)

    return game_data


# ==============================================================================
# ZERO-LEAKAGE DEDUPLICATION
# ==============================================================================

def hash_board(board):
    return hashlib.md5(board.tobytes()).hexdigest()

def get_all_equivalent_boards(board, move):
    equivalents = []
    equivalents.append((board.copy(), move, 'original'))
    mirror = board[:, ::-1, :].copy()
    equivalents.append((mirror, 6 - move, 'mirror'))
    flip = board[:, :, [1, 0]].copy()
    equivalents.append((flip, move, 'perspective_flip'))
    mirror_flip = mirror[:, :, [1, 0]].copy()
    equivalents.append((mirror_flip, 6 - move, 'mirror_and_flip'))
    return equivalents

def normalize_to_canonical_form(board, move):
    equivalents = get_all_equivalent_boards(board, move)
    min_hash = None
    canonical = None
    for equiv_board, equiv_move, transform in equivalents:
        h = hash_board(equiv_board)
        if min_hash is None or h < min_hash:
            min_hash = h
            canonical = (equiv_board, equiv_move, transform)
    return canonical

def deduplicate_all_transformations(raw_data):
    """
    Deduplicates across ALL 4 equivalent transformations.
    Each unique position appears EXACTLY ONCE.
    """
    print("  Phase 1: Normalizing all boards to canonical form...")
    canonical_to_entries = defaultdict(list)

    for idx, (board, label, metadata) in enumerate(raw_data):
        if idx % 50000 == 0 and idx > 0:
            print(f"    Processed {idx:,} / {len(raw_data):,} samples...")
        canon_board, canon_move, transform = normalize_to_canonical_form(board, 
label)
        canon_hash = hash_board(canon_board)
        canonical_to_entries[canon_hash].append({
            'board': canon_board, 'label': canon_move,
            'metadata': metadata, 'transform': transform,
            'original_index': idx
        })

    print(f"  Phase 1 complete: {len(raw_data):,} raw -> 
{len(canonical_to_entries):,} unique")

    print("  Phase 2: Majority voting for duplicates...")
    clean_data = []
    stats = {
        'total_raw': len(raw_data),
        'unique_canonical': len(canonical_to_entries),
        'transform_counts': Counter(),
        'duplicates_removed': 0,
        'conflicts_removed': 0
    }

    for canon_hash, entries in canonical_to_entries.items():
        for e in entries:
            stats['transform_counts'][e['transform']] += 1

        if len(entries) == 1:
            e = entries[0]
            clean_data.append((e['board'], e['label'], e['metadata']))
        else:
            labels = [e['label'] for e in entries]
            unique_labels = set(labels)
            if len(unique_labels) <= 2:
                label_counts = Counter(labels)
                majority_label = label_counts.most_common(1)[0][0]
                for e in entries:
                    if e['label'] == majority_label:
                        clean_data.append((e['board'], e['label'], e['metadata']))
                        break
                stats['duplicates_removed'] += len(entries) - 1
            else:
                stats['conflicts_removed'] += len(entries)

    stats['final_size'] = len(clean_data)
    print(f"  Phase 2 complete: {stats['duplicates_removed']:,} dupes, "
          f"{stats['conflicts_removed']:,} conflicts removed")
    print(f"  Final: {stats['final_size']:,} samples")
    return clean_data, stats


# ============================================================================
# LEAKAGE VERIFICATION
# ============================================================================

def verify_no_leakage(X, Y):
    print("=" * 70)
    print("LEAKAGE VERIFICATION")
    print("=" * 70)
    # Phase 1: canonical uniqueness
    print("Phase 1: Checking canonical form uniqueness...")
    canonical_hashes = set()
    for i in range(len(X)):
        if i % 50000 == 0 and i > 0:
            print(f"  {i:,} / {len(X):,}")
        canon_board, _, _ = normalize_to_canonical_form(X[i], Y[i])
        h = hash_board(canon_board)
        if h in canonical_hashes:
            print(f"  FAIL: duplicate canonical at index {i}")
            return False
        canonical_hashes.add(h)
    print(f"  PASS: {len(canonical_hashes):,} unique canonical forms")

    # Phase 2: no equivalent transformations
    print("Phase 2: Checking cross-transformation uniqueness...")
    all_hashes = set()
    symmetric = 0
    for i in range(len(X)):
        if i % 50000 == 0 and i > 0:
            print(f"  {i:,} / {len(X):,}")
        equivalents = get_all_equivalent_boards(X[i], Y[i])
        board_hashes = set()
        for eb, em, et in equivalents:
            h = hash_board(eb)
            if h in all_hashes:
                print(f"  FAIL: equivalent transformation at index {i}")
                return False
            board_hashes.add(h)
        if len(board_hashes) < 4:
            symmetric += 1
        all_hashes.update(board_hashes)

    print(f"  PASS: no cross-transformation leakage")
    print(f"  Unique forms: {len(canonical_hashes):,}, "
          f"variants: {len(all_hashes):,}, "
          f"ratio: {len(all_hashes)/len(canonical_hashes):.2f}x")
    if symmetric:
        print(f"  Symmetric boards: {symmetric:,} (normal)")
    print("VERIFIED: Zero leakage - safe for train/test split + augmentation")
    print("=" * 70)
    return True


# ==============================================================================
# NUMBA WARMUP
# ==============================================================================

def warmup_numba():
    """Pre-compile all Numba functions before spawning workers."""
    if not HAS_NUMBA:
        print("[WARMUP] Numba not available, running pure Python (SLOW)")
        return

    print("[WARMUP] Compiling Numba JIT functions...", end=" ", flush=True)
    t0 = time.time()
    b = np.zeros((6, 7), dtype=np.int8)
    _ = find_legal(b)
    _ = find_row(b, 3)
    b[5, 3] = np.int8(1)
    _ = check_win(b, 5, 3)
    _ = try_win(b, 2, np.int8(-1))
    b[5, 2] = np.int8(-1)
    _ = rollout_jit(b, np.int8(1))
    print(f"done in {time.time()-t0:.1f}s")


# ==============================================================================
# MAIN GENERATION FUNCTION
# ==============================================================================

def generate_dataset(
    num_games=50,
    mcts_rollouts=10000,
    random_opening_moves=6,
    num_workers=None,
    save_prefix="fast_10k",
    checkpoint_every=500
):
    if num_workers is None:
        num_workers = max(1, cpu_count())

    print("=" * 70)
    print("CONNECT 4 DATA GENERATION - SPEED OPTIMIZED")
    print("=" * 70)
    print(f"  Numba JIT:    {'YES' if HAS_NUMBA else 'NO (install numba!)'}")
    print(f"  Games:        {num_games:,}")
    print(f"  MCTS Rollouts:{mcts_rollouts:,}")
    print(f"  Opening:      0-{random_opening_moves} random moves")
    print(f"  Workers:      {num_workers}")
    print(f"  Checkpoints:  every {checkpoint_every} games")

    # Rough time estimate
    if HAS_NUMBA:
        secs_per_game = (mcts_rollouts / 5000) * 20  # ~2x faster with numba
    else:
        secs_per_game = (mcts_rollouts / 2500) * 20
    est_hours = (num_games * secs_per_game) / (num_workers * 3600)
    print(f"  Est. time:    {est_hours:.1f} hours")
    print("=" * 70)
    print()

    warmup_numba()

    all_data = []
    games_done = 0
    start = time.time()

    args_list = [(i, mcts_rollouts, random_opening_moves) for i in
range(num_games)]

    # Process in chunks for checkpointing
    chunk_size = checkpoint_every
    num_chunks = (num_games + chunk_size - 1) // chunk_size

    for chunk_idx in range(num_chunks):
        c_start = chunk_idx * chunk_size
        c_end = min((chunk_idx + 1) * chunk_size, num_games)
        chunk_args = args_list[c_start:c_end]

        print(f"Chunk {chunk_idx+1}/{num_chunks}: games {c_start+1}-{c_end}")
        chunk_t = time.time()

        with Pool(num_workers) as pool:
            for game_data in pool.imap_unordered(play_single_game, chunk_args,
chunksize=1):
                all_data.extend(game_data)
                games_done += 1

                if games_done % max(1, num_games // 100) == 0:
                    elapsed = time.time() - start
                    gps = games_done / elapsed
                    eta = (num_games - games_done) / gps if gps > 0 else 0
                    print(f"  [{games_done:,}/{num_games:,}] "
                          f"{gps:.2f} g/s | "
                          f"samples: {len(all_data):,} | "
                          f"ETA: {eta/60:.0f}m", flush=True)

        chunk_elapsed = time.time() - chunk_t
        print(f"  Chunk done in {chunk_elapsed:.0f}s | "
              f"total samples: {len(all_data):,}")

        # Checkpoint
        if games_done < num_games:
            cp = f"{save_prefix}_checkpoint_{games_done}.pkl"
            with open(cp, 'wb') as f:
                pickle.dump(all_data, f)
            print(f"  Checkpoint: {cp}")
        print()

    # Deduplication
    print("=" * 70)
    print("DEDUPLICATION")
    print("=" * 70)
    clean_data, stats = deduplicate_all_transformations(all_data)

    # Convert to arrays
    X = np.array([b for b, _, _ in clean_data], dtype=np.float32)
    Y = np.array([m for _, m, _ in clean_data], dtype=np.int64)
    metadata = [meta for _, _, meta in clean_data]

    # Verify
    is_clean = verify_no_leakage(X, Y)
    if not is_clean:
        raise ValueError("LEAKAGE DETECTED!")

    # Save
    print("\nSaving...")
    x_path = f'{save_prefix}_X.npy'
    y_path = f'{save_prefix}_Y.npy'
    meta_path = f'{save_prefix}_metadata.pkl'
    ids_path = f'{save_prefix}_game_ids.npy'
    np.save(x_path, X)
    np.save(y_path, Y)
    with open(meta_path, 'wb') as f:
        pickle.dump(metadata, f)
    game_ids = np.array([m['game_id'] for m in metadata], dtype=np.int32)
    np.save(ids_path, game_ids)
    print("Saved files:")
    for path in (x_path, y_path, meta_path, ids_path):
        print(f"  {os.path.abspath(path)}")

    total_time = time.time() - start
    print(f"\n{'='*70}")
    print(f"DONE in {total_time/60:.1f}min ({total_time/3600:.2f}hr)")
    print(f"  Games: {num_games:,} | Final samples: {len(X):,}")
    print(f"  X: {X.shape} {X.dtype} | Y: {Y.shape} {Y.dtype}")
    print(f"  Avg samples/game: {len(X)/num_games:.1f}")
    print(f"\nLabel distribution:")
    for col in range(7):
        n = (Y == col).sum()
        print(f"  Col {col}: {n:,} ({100*n/len(Y):.1f}%)")
    print(f"\nSafe for train/test split + augmentation")
    print(f"{'='*70}")
    return X, Y, metadata


# ============================================================================
# QUICK TEST
# ============================================================================

def run_test(n_games=3, rollouts=200):
    """Quick test to verify everything works."""
    print(f"\n{'='*70}")
    print(f"QUICK TEST: {n_games} games, {rollouts} rollouts")
    print(f"{'='*70}")

    warmup_numba()

    t0 = time.time()
    all_data = []
    for i in range(n_games):
        gt = time.time()
        data = play_single_game((i, rollouts, 6))
        all_data.extend(data)
        print(f"  Game {i+1}: {len(data)} samples in {time.time()-gt:.1f}s")

    elapsed = time.time() - t0
    print(f"\nTotal: {len(all_data)} samples in {elapsed:.1f}s")
    print(f"Speed: {n_games/elapsed:.2f} games/sec")

    # Test dedup
    clean, stats = deduplicate_all_transformations(all_data)
    X = np.array([b for b, _, _ in clean], dtype=np.float32)
    Y = np.array([m for _, m, _ in clean], dtype=np.int64)

    ok = verify_no_leakage(X, Y)
    print(f"\nLeakage test: {'PASS' if ok else 'FAIL'}")

    # Extrapolate overnight run
    secs_per_game = elapsed / n_games
    for target in [10000]:
        for workers in [cpu_count(), 30]:
            est = target * secs_per_game / workers
            print(f"  {target:,} games @ {rollouts} rollouts, "
                  f"{workers} workers: ~{est/3600:.1f} hours")
    # Now extrapolate for 10K rollouts
    scale = 10000 / rollouts
    secs_10k = secs_per_game * scale
    for workers in [cpu_count(), 30]:
        est = 10000 * secs_10k / workers
        print(f"  10,000 games @ 10,000 rollouts, "
              f"{workers} workers: ~{est/3600:.1f} hours")

    return ok


# ==============================================================================
# MAIN
# ==============================================================================

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--test", action="store_true", help="Run quick test")
    parser.add_argument("--games", type=int, default=60000)
    parser.add_argument("--rollouts", type=int, default=10000)
    parser.add_argument("--opening", type=int, default=6)
    parser.add_argument("--workers", type=int, default=None)
    parser.add_argument("--prefix", type=str, default="fast_10k")
    parser.add_argument("--checkpoint", type=int, default=500)
    args = parser.parse_args()

    if args.test:
        run_test()
    else:
        generate_dataset(
            num_games=args.games,
            mcts_rollouts=args.rollouts,
            random_opening_moves=args.opening,
            num_workers=args.workers,
            save_prefix=args.prefix,
            checkpoint_every=args.checkpoint,
        )
```