Sidorenko_Graphon

Raymond Sun

2025-06-06

Helper Functions

```
import numpy as np
import itertools
def count_homomorphisms_backtrack(H_adj, G_adj):
    h = H_adj.shape[0]
    g = G_adj.shape[0]
    count = 0

    mapping = [-1] * h

    def backtrack(pos):
        nonlocal count
        if pos == h:
            count += 1
            return

        for candidate in range(g):
            # Check edges from pos to previously mapped vertices
            valid = True
            for prev in range(pos):
                if H_adj[pos, prev] == 1 and G_adj[candidate, mapping[prev]] != 1:
                    valid = False
                    break
                if H_adj[prev, pos] == 1 and G_adj[mapping[prev], candidate] != 1:
                    valid = False
                    break
            if valid:
                mapping[pos] = candidate
```

```
                backtrack(pos + 1)
                mapping[pos] = -1

    backtrack(0)
    return count

def compute_t_H_G_backtrack(H_adj, G_adj):
    g = G_adj.shape[0]
    total_maps = g ** H_adj.shape[0]
    hom_count = count_homomorphisms_backtrack(H_adj, G_adj)
    return hom_count / total_maps

def average_degree(G_adj):
    n = G_adj.shape[0]
    total_degree = np.sum(G_adj)
    return total_degree / (n ** 2)

def sidorenko_ratio(H_adj, G_adj):
    # Computes p^|E(H)| / t - 1
    t = compute_t_H_G_backtrack(H_adj, G_adj)
    p = average_degree(G_adj)
    num_edges_H = np.sum(H_adj) // 2
    return (p ** num_edges_H / t - 1)
def compute_t_G_W(H, W_block):
    # Compute t for nxn graphon, brute force n^|V(H)| calculations
    n = H.shape[0]
    edges = [(i, j) for i in range(n) for j in range(i + 1, n) if H[i, j] == 1 or H[j, i] ==
    num_blocks = W_block.shape[0]
    block_volume = 1.0 / num_blocks
    t = 0.0
    for assignment in product(range(num_blocks), repeat=n):
        prob = 1.0
        for (u, v) in edges:
            prob *= W_block[assignment[u], assignment[v]]
        t += prob * (block_volume ** n)
    return t

def sidorenko_gap(H, W_block):
    # Compute p^|E(H)| - t
    t = compute_t_G_W(H, W_block)
    p = np.mean(W_block)
    num_edges = int(np.sum(H) // 2)
```

```
    return p**num_edges - t, t, p**num_edges

def perturb(W):
    W_new = W.copy()
    n = W.shape[0]

    # Flatten the indices of W into a list of (i, j) pairs
    all_indices = [(i, j) for i in range(n) for j in range(n)]

    # Shuffle and pick 1 indice to increase by 0.005, then 1 indice to decrease by 0.005
    random.shuffle(all_indices)
    increase_indices = all_indices[:1]
    decrease_indices = all_indices[1:2]

    # Check that the proposed updates stay in [0, 1]
    for i, j in increase_indices:
        if W_new[i, j] + 0.005 > 1.0:
            return W  # Reject and return original

    for i, j in decrease_indices:
        if W_new[i, j] - 0.005 < 0.0:
            return W  # Reject and return original

    # Apply the updates
    for i, j in increase_indices:
        W_new[i, j] += 0.005
    for i, j in decrease_indices:
        W_new[i, j] -= 0.005

    return W_new
```

Backtrack Homomorphism Counter Demo

```
H = np.array([
    [0,0,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,1,0,0,1,1],
    [0,0,0,0,0,1,1,0,0,1],
    [0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,1,1,1,0],
    [0,1,1,1,0,0,0,0,0,0],
```

```
    [0,0,1,1,1,0,0,0,0,0],
    [1,0,0,1,1,0,0,0,0,0],
    [1,1,0,0,1,0,0,0,0,0],
    [1,1,1,0,0,0,0,0,0,0]
])
G = np.array([
    [0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0],
    [1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1],
    [1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1],
    [0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1],
    [0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0],
    [0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0],
    [0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0],
    [1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1],
    [1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0],
    [0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0]
])

G1 = np.array([
  [0, 1],
  [1, 0]
  ])

G2 = np.array([
  [0, 1, 1, 1, 1, 1],
  [1, 0, 1, 0, 1, 1],
  [1, 1, 0, 1, 1, 1],
  [1, 0, 1, 0, 1, 1],
  [1, 1, 1, 1, 0, 1],
  [1, 1, 1, 1, 1, 0],
])
#print("Number of homomorphisms:",
#compute_t_H_G(H, G_adj)
count_homomorphisms_backtrack(H, G1)
count_homomorphisms_backtrack(H, G2)
gap = sidorenko_ratio(H, G2)
print("p^|E(H)| / t(H, G) - 1 =", gap)
count_homomorphisms_backtrack(H, G)
```

```
p^|E(H)| / t(H, G) - 1 = -0.2579700137774835
```

5654308

4x4 Graphon Optimization Demo

Fixed p and Sidorenko gap as reward function, randomly pick one weight to increase by 0.005 and one to decrease by 0.005 every action:

```
from itertools import product
import random
# ---- Define the graph G ----
H = np.array([
    [0,0,0,0,0,0,0,1,1,1],
    [0,0,0,0,0,1,0,0,1,1],
    [0,0,0,0,0,1,1,0,0,1],
    [0,0,0,0,0,1,1,1,0,0],
    [0,0,0,0,0,0,1,1,1,0],
    [0,1,1,1,0,0,0,0,0,0],
    [0,0,1,1,1,0,0,0,0,0],
    [1,0,0,1,1,0,0,0,0,0],
    [1,1,0,0,1,0,0,0,0,0],
    [1,1,1,0,0,0,0,0,0,0]
])
  # W with average p = 0.80
W = np.array([
 [0.78, 0.86, 0.69, 0.87],
 [0.91, 0.81, 0.6, 0.88],
 [0.8, 0.68, 0.96, 0.76],
 [0.71, 0.85, 0.95, 0.69]
])
sidorenko_gap(H, W)
```

```
(np.float64(-3.4164493493207826e-05),
 np.float64(0.03521853658232524),
 np.float64(0.03518437208883203))
```

```
# --- Optimization loop
best_gap, best_t, best_p_e = sidorenko_gap(H, W)
W_best = W.copy()  # initialize best

for step in range(100): # Custom amount of iterations
    W_new = perturb(W)
```

```
    new_gap, new_t, new_p_e = sidorenko_gap(H, W_new)
    delta = abs(new_gap) - abs(best_gap)

    if delta < 0:
        W = W_new  # <-- update W!
        if abs(new_gap) < abs(best_gap):
            W_best = W.copy()
            best_gap, best_t, best_p_e = new_gap, new_t, new_p_e

# --- Final output
print("\nFinal optimized W:")
print(np.round(W_best, 3))  # <-- use W_best
print(f"Final Sidorenko gap: {best_gap:.3e}")
```

```
Final optimized W:
[[0.775 0.86  0.7   0.865]
 [0.905 0.81  0.605 0.875]
 [0.8   0.675 0.96  0.765]
 [0.715 0.855 0.94  0.695]]
Final Sidorenko gap: -3.048e-05
```

Example: Start from 4x4 graphon with weight average p = 0.5:

```
W = np.array([
 [0.89, 0.49, 0.35, 0.5],
 [0.31, 0.78, 0.2, 0.59],
 [0.39, 0.71, 0.51, 0.2],
 [0.11, 0.87, 0.39, 0.68],
])
sidorenko_gap(H, W)
```

```
(np.float64(-6.471107749430073e-05),
 np.float64(9.355637888835444e-05),
 np.float64(2.8845301394053712e-05))
```

After running loop for many iterations, approaches uniform weights and stabilizes here:

```
W_best = np.array([
  [0.54, 0.47, 0.495, 0.485],
 [0.5, 0.495, 0.48, 0.515],
 [0.485, 0.53, 0.52, 0.46 ],
 [0.47, 0.495, 0.495, 0.535]
 ])
sidorenko_gap(H, W_best)
```

```
(np.float64(-1.6290915704104958e-09),
 np.float64(2.884693048562417e-05),
 np.float64(2.884530139405376e-05))
```

```
W = W_best
best_gap, best_t, best_p_e = sidorenko_gap(H, W)
W_best = W.copy()  # initialize best
for step in range(100):
    W_new = perturb(W)
    new_gap, new_t, new_p_e = sidorenko_gap(H, W_new)
    delta = abs(new_gap) - abs(best_gap)

    if delta < 0:
        W = W_new  # <-- update W!
        if abs(new_gap) < abs(best_gap):
            W_best = W.copy()
            best_gap, best_t, best_p_e = new_gap, new_t, new_p_e

# --- Final output
print("\nFinal optimized W:")
print(np.round(W_best, 3))  # <-- use W_best
print(f"Final Sidorenko gap: {best_gap:.3e}")
```

```
Final optimized W:
[[0.54  0.47  0.495 0.485]
 [0.5   0.495 0.48  0.515]
 [0.485 0.53  0.52  0.46 ]
 [0.47  0.495 0.495 0.535]]
Final Sidorenko gap: -1.629e-09
```

Thoughts:

1. Changing only two weights would still affect 74% of the mappings in 4*4 case, so just storing the last value of t and updating doesn't save much time. It saves over 50% of time for 5*5 graphons, although 5^{10} is almost 10 times the value of 4^{10}
2. For H with 10 vertices and 4*4 graphon, the iterations already take a lot of time, and is exponential so approximate methods would be needed to explore cases like C4xC4 and 6*6 graphons.
3. In 4x4 case, weights approach the uniform case where Sidorenko gap = 0 as expected with the current random update strategy with fixed p, but maybe some better RL techniques could help find escape local maxima to find weight distributions that potentially cross 0.
4. Backtrack algorithm seems to give accurate numbers and is many times faster than brute force, but it is slower compared to Sagemath so it would be helpful if we could find how to make Sagemath algorithm work for $|V(H)| > 8$.