Problem set 1

Exercise 1) K-clique

* Input: G=(V,E), k and t=k/3
* Steps

1. Find all t-clusters S={v1,…,vt} in G

It can be done using brute-force search / exhaustive search

Amount of subgraphs of size t:

Each is determined to be a t-clique in: O(t2)

Total running time: O(ntt2)

1. Transform G into G’=(V’,E’)

For each cluster S create a node in G’ representing it

Two nodes S1 and S2 in V’ are connected by an edge if no vertex vi in V appears in both S1 and S2 and if for all vi in S1 and all uj in S2 there is an edge (vj,ui) in E

Creating nodes takes: n’<O(nt)

Potential edges: O(n’2)<O(n2t)

Checks per edge: O(t2)

Total: O(n2tt2)

1. Find triangles in G’

Iff a triangle is in G’ then a k-cluster is in G

Do a matrix multiplication on the adjacency matrix of G’ O(n’w)<O(ntw)

For each edge in V’ check if triangle exist in O(1) time O(n’2)<O(n2t)

Total: O(ntw)

* Running time:

Step 1: O(ntt2)

Step 2: O(n2tt2)

Step 3: O(ntw)

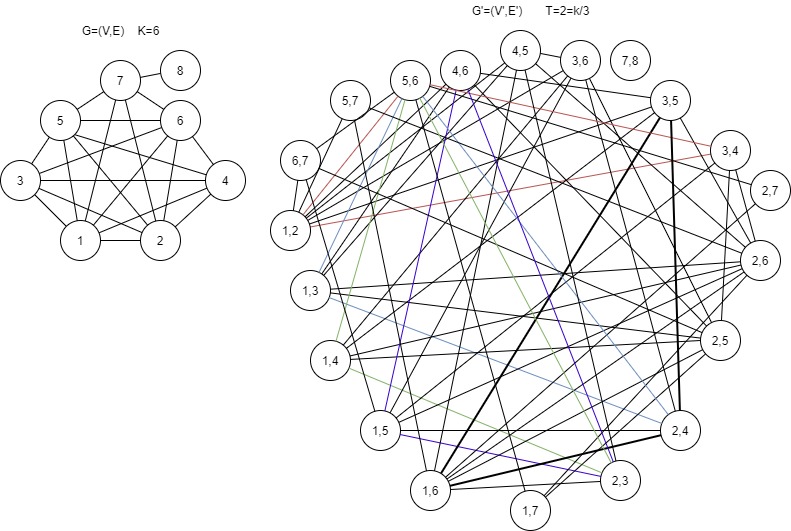
Total: O(ntw)= O(nwk/3)

Since ntw<n2tt2 since t<n

* Proof of correctness

If a k-cluster exists and no triangle is found, then there are no 3 nodes S1, S2, S3 in V’ which together represents k unique nodes that are fully connected. So if no triangle is found then the nodes in Si aren’t fully connected to the nodes in Sj for i!=j and if that is the case that some node vk in Si aren’t connected to some node vo in Sj then vk and vo can’t be part of the same clique and thus it isn’t contradicting.

If a triangle is found but no k-clique exists then the k nodes represented in S1={v1,…,vt}, S2={vt+1,…,v2t} and S3={v2t+1,…,vk} must be fully connected since if edge (vi,vj) isn’t in E for vj in Sj’ and vi in Si’ then edge (Si, Sj) wouldn’t be in E’. If (vi,vj) isn’t in E for vi,vj in Sk then the edges in Sk wouldn’t form a t-clique and thus Sk wouldn’t be in V’ and therefore also not part of any triangle.



Exercise 2) Dv in O(n log n)

* Input: D= and v=

Vandermonde matrix: M=

* Observation:

Dv forms a system of n linear equations working on the n roots of unity:

Pi(x)= for i=0,…,n and x=wn,i

* FFT can be used directly since as input it takes the coefficients v={a0,…,an-1} and its output is the result of the polynomial on each of the n roots of unity wn,0,…,wn,n-1 thus giving .

Therefore the result of Dv can be computed using FFT running in O(n log n) time

Exercise 3) any odd integer n>5 is writable as sum of 3 primes

1. Counting pairs(p1,p2,p3) for one value n%2=1

The P primes are received and converted to coefficients of a polynomial with prime pi indicating that coefficient with index pi is 1.

This polynomial P(x) with coefficients [0,{0,1}|P|] is then cubed by first computing p2(x). Each multiplication starts by appending zeroes until both have the same length that is a power of 2 and at least 2n the original largest.

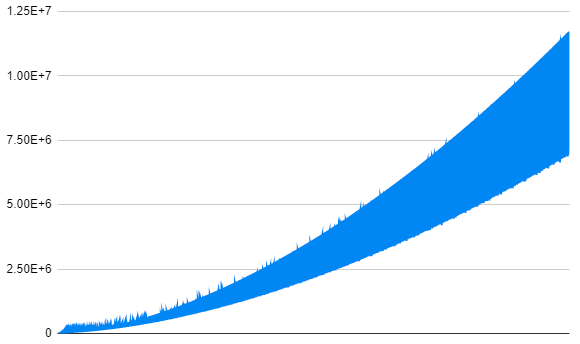
The result is then the n’t coefficient of p3(x).

Running time is bounded by doing FFT 6 times: O(n log n)

1. Bonus

Y-axis = combinations x-axis = n in range(7,…,99999) for n%2=0

To reduce time of running the p3(x) is only computed once and it starts with 10000 coefficients such that each n only requires O(1) time.



Exercise 4) compute det(A) in O(nw+1)

* The code can be found in Determinant.py. There are 3 steps, preparation, polynomial creation and polynomial multiplication.
* preparation step:

Compute all Ai for i=1,…,n-1 O(n\*nw)=O(nw+1)

Computing each tr(Ai) O(n\*n)=O(n2)

Computing all n factorials for O(n)

* Polynomial creation

Create n polynomials of degree n with coefficient cl,0=1 and the other cl,e being 0 if e isn’t divisible by l, else it is the Newton identity:

Computations: O(n2) since the Newton identity is with the preparation step a constant time operation.

* Matrix multiplication

Until the list of polynomials only contain 1 polynomial, 2 are popped then multiplied and the first n coefficients are then put back into the list. Thus there will be n calls to FFT.

The final result will be the coefficient for xn.

Total computations: O(n2 log n)

* Running time

Preparation step: O(nw+1)

Polynomial creation: O(n2)

Matrix multiplication: O(n2 log n)

Total: O(nw+1)