EECS 477. HOMEWORK 4 SOLUTIONS.

1. Random Graph Generation and Visualization (10pts)

In this assignment you will need to generate random undirected graphs and visualize them.

First, generate N random points uniformly distributed within the unit square $[0,1] \times [0,1]$. These points will form the vertex set V of your graph $G = (V, E)$. Then randomly generate edges so that the probability of an edge connecting two vertices v and v' is given as:

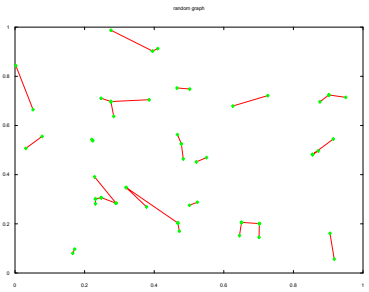
$$P(\{v, v'\} \in E) = \begin{cases} H, & \text{when } \rho(v, v') < D, \\ He^{A(D-\rho(v,v'))} & \text{when } \rho(v, v') \geq D. \end{cases}$$

Here $\rho(v, v')$ denotes the Euclidean distance between the points v and v'. $0 \leq H \leq 1$, $A > 0$, and $D > 0$ are constants.

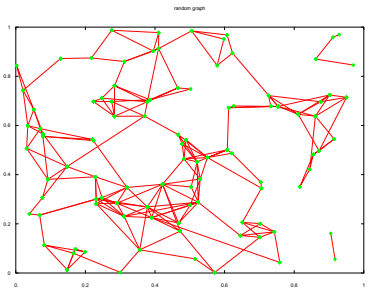
Submit the printout of your algorithm and the graph plots with parameters set to $N = 1000$, $H = 0.8$, $A = 25$ and different values of $D = 0, D = 0.1, D = 0.5, D = 1$. You may use `gnuplot` for the visualization.

Solution: See the graphs below. The source code is attached in a separate file.

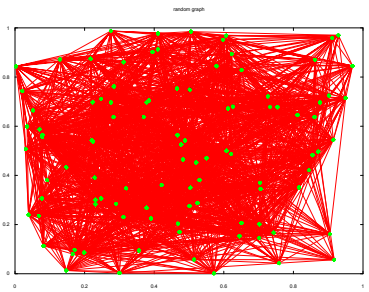
$D = 0.0$



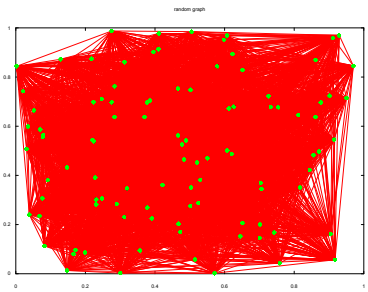
$D = 0.1$



$D = 0.5$



$D = 1.0$



2. k-ary heap Dijkstra algorithm(25pts)

Implement Dijkstra algorithm from Section 6.4 (pp.189–202) with k-ary heap (see Problem 6.16 for the details). You will need to implement your own k-ary heap functionality (see Problem 5.23). The input graphs for the Dijkstra algorithm will come from the first part of this homework: you will use undirected graphs with edge lengths generated randomly so that they are distributed uniformly in $[0, 1]$.

Submit a printout of your program together with two of example runs on some simple graphs: assume that the first vertex is the starting one, then the result will be the distance assignment from that vertex.

Analyze the asymptotic runtime of your algorithm both theoretically and experimentally. Plot the runtime of the algorithm varying the number of vertices and edges in your graph (one way to leave the number of edges approximately constant is to double the number of vertices and at the same time divide the constant H by four in your graph generation procedure).

Change the value of the heap parameter k between 2 and $k' = \max(2, \lfloor a/n \rfloor)$ (do it gradually in ten increments of the size $\lfloor (k' - 2)/10 \rfloor$. Do you see the performance boost as expected from the description in Problem 6.16? Plot the resulting performance plot (runtime against k) for a graph with the number of vertices greater than 5,000 and the number of edges greater than 100,000. Try to match your asymptotic analysis and the observed performance.

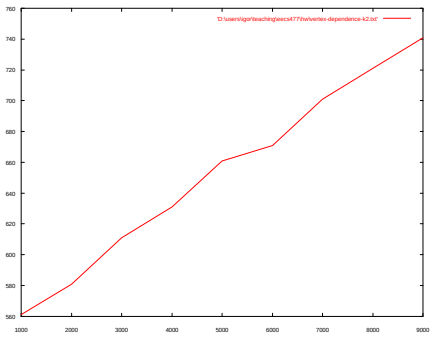
Solution: Two example runs and the code are attached in a separate file. We shall analyze the runtime of `dijkstra` function below:

```
void dijkstra(GraphT& g, int K, vector<float>& d) {
  // finds distances for all the points from the origin

  vector<bool> cand(g.size(), true);
  cand[0] = false;

  assert(g.size()==d.size());

  for(int i=1; i<d.size(); ++i)
    d[i] = 2*INFINITY;
  d[0] = 0.0f;

  for(int j=0; j<g.nei(0).size(); ++j) {
    GraphT::EdgeT& e = g.nei(0)[j];
    if(cand[e.dest])
      d[e.dest] = min(d[e.dest], d[0]+e.length);
  }

  vector<int> v(g.size()-1);
  vector<int> order(g.size());
```

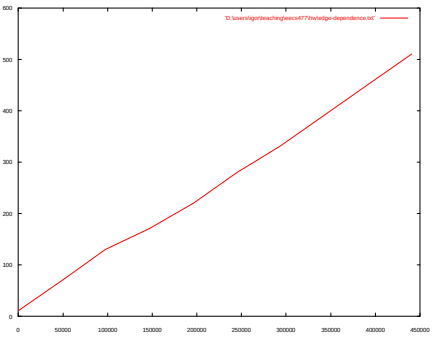
```
  order[0] = -1;

  for(int i=1; i<g.size(); ++i) {
    v[i-1] = i;
    order[i] = i-1;
  }

  WorseThanPred wpred(d);
  make_kheap(v, order, v.size(), wpred, K);
  for(int k=v.size(); k>0; --k) {
    pop_kheap(v, order, k, wpred, K);
    int itop = v[k-1];
    cand[itop] = false;
    for(int j=0; j<g.nei(itop).size(); ++j) {
      GraphT::EdgeT& e = g.nei(itop)[j];
      if(cand[e.dest]) {
        d[e.dest] = min(d[e.dest], d[itop]+e.length);
        percolate_kheap(v, order, order[e.dest], wpred, K);
      }
    }
  }
}
```

The `for(int k=)` loop dominates the runtime of the algorithm; within that loop, we have $|V|$ `pop_kheap` calls. Additionally, the `if(cand[e.dest])` statement will be called twice for every edge and `percolate_kheap` will be called once for every edge. The runtime of a single call to `pop_kheap` is $O(k \log_k |V|)$ and a single call to `percolate_kheap` takes $O(\log_k |V|)$. Thus the overall runtime will be $T(|V|, |E|, k) = O(|V|k \log_k |V| + |E| \log_k |V|)$.

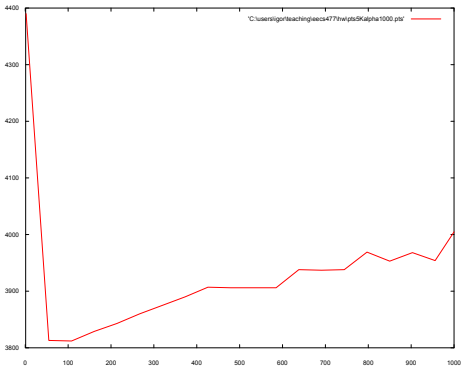
First, let's experiment with the performance keeping $k = 2$. Here we plot the runtime changing vertex count keeping edge count the same. We see roughly linear growth as expected (the effect of the logarithm is not seen).



Now let's plot performance changing edge count keeping vertex count the same. We see linear growth as expected.

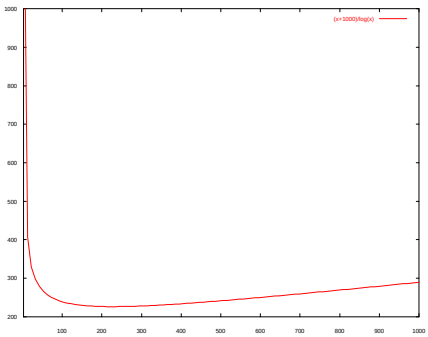


Now let's plot performance for a fixed graph changing the parameter k of the heap.



Here the ratio $\alpha = |E|/|V|$ is around 1000, so the best parameter will be obtained by minimizing the function $\tau(k) = C(k + \alpha)/\log k$.

The optimal value will not be achieved at $k = \alpha$ but depending on α will be closer to zero. Below we plot the function $\tau(k)$ for $\alpha = 1000$ and see that the overall behavior matches the performance above (although the minimum is shifted closer to the origin in our implementation).



3. Linear inhomogeneous recurrencies (20pts)

Solve the following recurrencies exactly and express your answer as simply as possible using Θ notation.

A.

$$t(n) = 5 * t(n-1) - 4 * t(n-2) + 3 * 2^n,$$
$$t(0) = 1,$$
$$t(1) = 2.$$

Solution: First rewrite the equation in the canonical form:

$$t(n) - 5 * t(n-1) + 4 * t(n-2) = 3 * 2^n$$

The characteristic equation is

$$(x^2 - 5x + 4)(x - 2) = 0$$

The roots are $x_1 = 1$, $x_2 = 4$, and $x_3 = 2$. There are no repeating roots and so the general solution will have the form

$$t(n) = A_1 1^n + A_2 4^n + A_3 2^n$$

Here the first two terms always satisfy the homogeneous equation (without right hand side), and the last term is used to satisfy the inhomogeneous equation.

Substitute this into the original equation to get A_3 (the terms with A_1 and A_2 will disappear, make sure to understand why):

$$A_3 2^n - 5A_3 2^{n-1} + 4A_3 2^{n-2} = 3\,2^n$$

It follows that $A_3 = -6$. Thus our solution has the form:

$$t(n) = A_1 + A_2 4^n - 6 2^n$$

We have two unknowns and two additional initial conditions. Substitute $t(n)$ for $n = 0$ and 1 to get:

$$A_1 + A_2 - 6 = 1$$
$$A_1 + 4A_2 - 12 = 2$$

so that $A_1 = 14/3$ and $A_2 = 7/3$

Finally, the answer is

$$t(n) = \frac{14}{3} + \frac{7}{3} * 4^n - 6 * 2^n.$$

B.

$$t(n) = 5 * t(n-1) - 4 * t(n-2) + (n+1) * 4^n,$$
$$t(0) = 1,$$
$$t(1) = 2.$$

Solution: First rewrite the equation in the canonical form:

$$t(n) - 5 * t(n-1) + 4 * t(n-2) = (n+1) * 4^n$$

The characteristic equation is

$$(x^2 - 5x + 4)(x-4)^2 = 0$$

which is the same as

$$(x-1)(x-4)^3 = 0$$

The roots are $x_1 = 1$, $x_{2,3,4} = 4$. The general solution will have the form

$$t(n) = A_1 + A_2 4^n + A_3 n 4^n + A_4 n^2 4^n$$

Here the first two terms always satisfy the homogeneous equation (without right hand side), and the last two terms is used to satisfy the inhomogeneous equation.

Substitute this into the original equation to get $A_3 = 14/9$ and $A_4 = 2/3$.

Now we know that the solution has the form

$$t(n) = A_1 + A_2 4^n + \frac{14}{9} n 4^n + \frac{2}{3} n^2 4^n,$$

and now the remaining two coefficients can be found from initial conditions

$$A_1 + A_2 = 1$$
$$A_1 + 4A_2 = -62/9$$

so that $A_1 = 98/27$ and $A_2 = -71/27$.

The answer is then

$$t(n) = \frac{98}{27} - \frac{71}{27} 4^n + \frac{14}{9} n 4^n + \frac{2}{3} n^2 4^n.$$

4. Master theorem (20pts)

Analyze the following recurrencies using the master theorem and express your answer as simply as possible using Θ notation.

A. $t(n) = 4t(n/3) + n^2$
Solution: $f(n) = n^2 = \Omega(n^{\log_3 4 + 0.0001})$ so it's Case 3 of MT, which results in $t(n) = \Theta(n^2)$

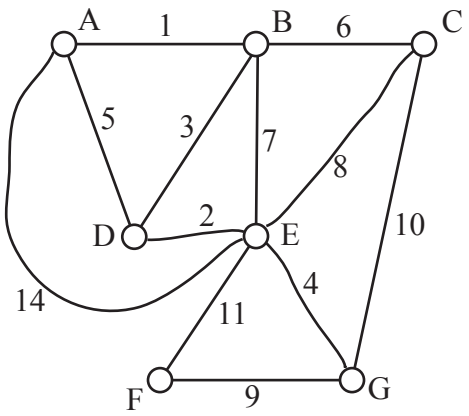
B. $t(n) = 4t(n/3) + n$
Solution: $f(n) = n = O(n^{\log_3 4 - 0.0001})$ so it's Case 1 of MT, which results in $t(n) = \Theta(n^{\log_3 4})$

C. $t(n) = 3t(n/3) + \log n$
Solution: $f(n) = \log n = O(n^{\log_3 3 - 0.0001}) = O(n^{1-0.0001})$ so it's Case 1 of MT, which results in $t(n) = \Theta(n)$

D. $t(n) = 2t(n/3) + \log n$

Solution: $f(n) = \log n = O(n^{\log_3 2 - 0.0001})$ so it's Case 1 of MT, which results in $t(n) = \Theta(n^{\log_3 2})$

5. MST and Union-find (25pts)



In this part, we will be looking for the minimum spanning tree of the undirected graph above.

Give the order in which edges will be added to the MST by Prim's algorithm. Start the algorithm from vertex A.

Repeat for Kruskal's algorithm.

Show the final configuration of the disjoint set structure tree at the end of Kruskal's algorithm (use the disjoint set structure with path compression, and in case of ties, use the alphabetically larger node as the root).

Solution: Prim's algorithm order: `AB, BD, DE, EG, BC, GF`.

Kruskal's algorithm order:

```
AB accepted
DE accepted
BD accepted
EG accepted
* AD discarded
BC accepted
* BE discarded
* CE discarded
FG accepted
stop
```

The disjoint set structure will evolve as follows:

```
Initially:
abcdefg
ABCDEFG: pointers array
```

```
0000000: ranks

AB accepted
find A, find B, merge AB: B becomes root
abcdefg
BBCDEFG: pointers array
0100000: ranks

DE accepted
find D, find E, merge DE: E becomes root
abcdefg
BBCEEFG: pointers array
0100100: ranks

BD accepted

find B: returns B
abcdefg
BBCEEFG: pointers array
0100100: ranks

find D: returns E
abcdefg
BBCEEFG: pointers array
0100100: ranks

merge B and E: E becomes root
abcdefg
BECEEFG: pointers array
0100200: ranks

EG accepted
find E: returns E
abcdefg
BECEEFG: pointers array
0100200: ranks

find G: returns G
abcdefg
BECEEFG: pointers array
0100200: ranks

merge E and G: E becomes root
abcdefg
BECEEFE: pointers array
0100200: ranks

* AD discarded
```

```
find A: returns E, compresses path ABE
abcdefg
EECEEFE: pointers array
0100200: ranks

find D: returns E
abcdefg
EECEEFE: pointers array
0100200: ranks

BC accepted
find B: returns E
abcdefg
EECEEFE: pointers array
0100200: ranks

find C: returns C
abcdefg
EECEEFE: pointers array
0100200: ranks

merge E and C: E becomes root
abcdefg
EEEEEFE: pointers array
0100200: ranks

* BE discarded
find B: returns B
find E: returns E
abcdefg
EEEEEFE: pointers array
0100200: ranks

* CE discarded
find C: returns E
find E: returns E
abcdefg
EEEEEFE: pointers array
0100200: ranks

FG accepted
find F: returns F
abcdefg
EEEEEFE: pointers array
0100200: ranks

find G: returns E
abcdefg
```

```
EEEEEFE: pointers array
0100200: ranks

merge F and E: E is a root again

abcdefg
EEEEEEE: pointers array
0100200: ranks

STOP
```