In this assignment, you measure the performance of Dijkstra's algorithm and Floyd's algorithm for the all pairs shortest path problem. As Dijkstra's is designed for the single source problem, you need to repeat the algorithm by changing the source n times, where n is the number of vertices. A samples program with a binary heap is attached.

**Problem 1.** Draw a graph of your choice with more than 15 vertices and 30 edges. You can choose your own cost for each edge. Enhance both algorithms so that they can actually output shortest paths in addition to shortest distances. Output the shortest paths between two pairs of vertices by both algorithms using a computer and moving object. The moving object can move with the return key, or actually move.

**Problem 2.** Organize the frontier set F into a binary heap and a ternary heap, described in page 40 of the notes and tutorial. Compare computing times of Floyd, the plain version, and the versions with the binary and ternary heap. Those new versions are supposed to be efficient for sparse graphs. To define sparse graphs, we incorporate a parameter p which is the edge existence probability from vertex i to vertex j for i=1, …, n; j=1, …, n. Thus the expected value of edges, m, is pn^2. Sparse graphs are those with small p, thus small m. See attachment for Dijkstra with binary heap. Draw the results of time and comparison measurements in graphs, and discuss the performance of each method. Comparison means key (distance) comparison. By changing n and p, measure the computing time of the three algorithms. The table for time or key comparison will look like:

<table>
<thead>
<tr>
<th>n</th>
<th>p</th>
<th>Floyd</th>
<th>Plain Dijkstra</th>
<th>binary Dijkstra</th>
<th>ternary Dijkstra</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>similar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>similar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>similar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Present your assignment work in print, which includes outlines of algorithms, drawing of input graph, outputs of paths, tables of time measurements, graphs of performance measurements, and discussions on the results.

Also submit files of source code and a file of your graph with “submit” command. Make empty “ff.dat”, then use the interface program to generate a graph in “ff.dat”. Shortest paths between user-specified vertices (by mouse click) must be shown on the display.

**Due date:** October 11, 2006, 5pm. Drop due one week later. Worth 30%
// This is to maintain frontiers in a binary heap, August 2006
// This is a solution for assignment Sept 2006
#include <stdio.h>
FILE *myfile;
float p;
int i, j, k, kmin, n, v, w, t; int V[1000]; int out[801][801];
int d[1000], f[1000], s[1000]; int a[801][801]; int d1[801][801];
int H[1000], P[1000];
int b[801][801];
int end;
int x, y, dummy;
output(int a[][801]){int i, j;
        for(i=1;i<=n;i++){
          for(j=0;j<=n;j++)printf("%4d ",a[i][j]);
          printf("\n");
        }
}
output1(int a[]){int i;  //For debugging
        for(i=1;i<=n;i++) printf("%4d ",a[i]);
        printf("\n");
}
siftup(int p, int q, int H[]){
        int j, k; int x, y, z;  // x, y, z : vertices
        y=H[p];
        j=p; k=2*j;
        while (k<=q) {
            z = H[k];
            y=H[p];
            if (k<q) if (d[z]>d[H[k+1]]) {
                k++; z = H[k]; } // Take z with smallest key
            if (d[y]<=d[z]) break;
            H[j] = z; P[H[j]]=j; j = k; k = 2*j;
        }
        H[j]=y; P[H[j]]=j; // vertex y settles down at the j-th of H
    }
decrease_key(int w, int H[]){
        int i, j, k;
        j=P[w]; i=j/2;
        while((i>0)&&(d[H[i]]>d[w])){
            H[j]=H[i]; P[H[j]]=j;
            j=i; i=i/2;
        }
        H[j]=w; P[w]=j;
    }
insert(int w, int H[]){
        int i, j, k;
        end++; j=end;
        i=end/2;
        while((i>0)&&(d[H[i]]>d[w])){
            H[j]=H[i]; P[H[j]]=j;
            j=i; i=i/2;
        }
        H[j]=w; P[w]=j;
    }
    // H is the heap H[1..end]
    // s and f are membership arrays for S and F
    // siftup is from the lecture notes. Actually siftdown
    main(){
init(); // output(a); printf("\n"); output(out); printf("\n");
t=clock();
/*** Dijkstra for all pairs ***/
for(i=1;i<=n;i++) // Source is i
// printf("Start from %d\n", i);
end=0;
for (j=1;j<=n;j++) {d[j]=999; s[j]=0; f[j]=0;}
H[1]=i; f[i]=1; d[i]=0;

for (j=1;j<=n;j++) {
    /*** delete min ***/
    v=H[1]; s[v]=1; f[v]=0;
    if(end>=1) // end is the last position of heap H
        {H[1]=H[end]; H[end]=0; end--; siftup(1, end, H);}  // siftdown

    /*** update or expand F ***/
    for(k=1;k<=out[v][0];k++)
        w=out[v][k];
    if (s[w]==0) {  /** if w is not in S **/
        if (f[w]==1) {  /** if w is in F **/
            /*** decrease-key ***/
            if (d[v]+a[v][w]<d[w])
                {d[w]=d[v]+a[v][w];
                decrease_key(w, H);
                }
        } else {
            /*** insert ***/
            d[w]=d[v]+a[v][w];
            insert(w, H); f[w]=1;
        }
    }

    for(j=1;j<=n;j++) b[i][j]=d[j]; // b is the container for all-pairs
}
output(b);
printf("time= %d\n", (clock()-t)/1000);
getchar();
/*** Floyd for all pairs ***/
t=clock();
printf("Floyd starts\n");
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        for (k=1;k<=n;k++)
            if (d1[i][k]+d1[k][j]<d1[i][j])
                {d1[i][j]=d1[i][k]+d1[k][j]; /* k may be on the path */
                printf("time= %d\n", (clock()-t)/1000);
            } getchar();
output(d1);}
init(){int i,j,k;
/** Graph initialization **/  
printf("Input n and p"); scanf("%d %f",n, &p);  
for (i=1; i<=n; i++) {  
  for (j=1; j<=n; j++) {a[i][j]=random()%n+1;  
    if(i==j)a[i][j]=0;}  
};  

for(i=1;i<=n;i++){  
  for(j=1;j<=n;j++)  
    if((a[i][j]>(int)(n*p))&&(j!=(i+1)%n)) a[i][j]=99;  
}

for(i=1;i<=n;i++){ /* out[i][0] is number of edges from i */  
  out[i][0]=0;  
  for(j=1;j<=n;j++)  
    if((a[i][j]<99)&&(i!=j)){  
      out[i][0]++; out[i][out[i][0]]=j;  
    }  
  }
}