CSE 542 – Advanced Data Structures and Algorithms *Jon Turner*

Lab 1 Report

*Your name here:*_______*Zheng Luo*_____________ *Due 1/29/2013*

Part A. Modifications to Dheap

1. (10 points) Your *svn* repository includes a directory called *grafalgo* that contains various data structures and graph algorithms. Modify the *Dheap* data structure that you will find there, as discussed in the lab instructions. Paste a copy of your modified code below. You may leave out methods that you did not change. Please highlight your changes by making them bold. Notice that the lines below are formatted using the "code" paragraph style, which uses a fixed width font and has appropriate tab stops. Please use this paragraph style for all code.

```
Dheap.h

class Dheap : public Adt {
public:
    …
    // variables to count the number of steps
    int changeKeyCount;
    int siftupCount;
    int siftdownCount;
private:
    …
};


Dheap.cpp

void Dheap::changekey(index i, keytyp k) {
    changeKeyCount++;
    keytyp ki = kee[i]; kee[i] = k;
    if (k == ki) return;
    if (k < ki) siftup(i,pos[i]);
    else siftdown(i,pos[i]);
}

void Dheap::siftup(index i, int x) {
    int px = p(x);
    while (x > 1 && kee[i] < kee[h[px]]) {
            h[x] = h[px]; pos[h[x]] = x;
            x = px; px = p(x);
            siftupCount++;
    }
    h[x] = i; pos[i] = x;
}
```

```
int Dheap::minchild(int x) {
    int y; int minc = left(x);
    if (minc > hn) return 0;
    for (y = minc + 1; y <= right(x) && y <= hn; y++) {
        siftdownCount++;
        if (kee[h[y]] < kee[h[minc]]) minc = y;
    }
    return minc;
}
```

Part B. Common.cpp and basic verification tests.

1. (15 points) Paste a copy of the modified *common.cpp* below. You may leave out methods that you did not change. Please highlight your changes by making them bold.

```
void prim(Wgraph& wg, list<int>& mstree, int d, PrimStats& stats) {

    // Make changes to configure Dheap parameter and
    // return statistics
    // Get the time first
    int t0 = Util::getTime();
    vertex u,v; edge e;
    edge *cheap = new edge[wg.n()+1];
    bool *intree = new bool[wg.n()+1];
    // Create a d heap
    Dheap nheap(wg.n(),d);
    // Initialize all the variables to count the number of steps to 0
    nheap.changeKeyCount = 0;
    nheap.siftupCount = 0;
    nheap.siftdownCount = 0;
    for (e = wg.firstAt(1); e != 0; e = wg.nextAt(1,e)) {
        u = wg.mate(1,e); nheap.insert(u,wg.weight(e)); cheap[u] = e;
    }
    intree[1] = true;
    for (u = 2; u <= wg.n(); u++) intree[u] = false;
    while (!nheap.empty()) {
        u = nheap.deletemin();
        intree[u] = true; mstree.push_back(cheap[u]);
        for (e = wg.firstAt(u); e != 0; e = wg.nextAt(u,e)) {
            v = wg.mate(u,e);
            if (nheap.member(v) && wg.weight(e) < nheap.key(v)) {
                nheap.changekey(v,wg.weight(e)); cheap[v] = e;
            } else if (!nheap.member(v) && !intree[v]) {
                nheap.insert(v,wg.weight(e)); cheap[v] = e;
            }
        }
    }
    // Update the statistics
    stats.ckCount = nheap.changeKeyCount;
    stats.suCount = nheap.siftupCount;
    stats.sdCount = nheap.siftdownCount;
    int t1 = Util::getTime();
    stats.runtime = t1-t0;
}

void badcase(int n, int m, Wgraph& wg) {
    vertex u,v; edge e;
    // Creating edges from vertex 1 to every other vertex
    for (u = 2; u <= n; u++) wg.join(1,u);
    // Createing edges from each vertex u >= 2 to u+1.
    for (u = 2; u <= n-1; u++) wg.join(u,u+1);
    // Random edges are then added to bring the total to m (so m >= 2*n-
3).
    wg.addEdges(m);
    // Edge weights for edges (u,v) with u<v are set to v-u.
    for (e = wg.first(); e != 0; e = wg.next(e)) {
        u = min(wg.right(e),wg.left(e));
        v = max(wg.right(e),wg.left(e));
        wg.setWeight(e,v-u);
    }
}
void worsecase(int n, int m, Wgraph& wg) {
    vertex u,v; edge e;
    // Creating edges from vertex 1 to every other vertex
    for (u = 2; u <= n; u++) wg.join(1,u);
    // Createing edges from each vertex u >= 2 to u+1.
    for (u = 2; u <= n-1; u++) wg.join(u,u+1);
    // Random edges are then added to bring the total to m (so m >= 2*n-
3).
    wg.addEdges(m);
    // If {u,v} is an edge where v=u+1, the weight assigned to {u,v}
    // should be 1
    // Edge weights for edges (u,v) with u<v are set to n*(n-u)+(n-v).
    for (e = wg.first(); e != 0; e = wg.next(e)) {
        u = min(wg.right(e),wg.left(e));
        v = max(wg.right(e),wg.left(e));
        if (v-u == 1)
            wg.setWeight(e,v-u);
        else
            wg.setWeight(e,n*(n-u)+(n-v));
    }
}
```

2. (5 points) In the *lab1* directory, you will find a program *checkPrim.cpp* that can be used to verify the operation of your modified version of Prim's algorithm. Examine the code, then compile it, using the provided *makefile* (you may need to adjust the *makefile* to suit your environment) and run it by typing the command. (Note, the instructions here, assume you are using a Mac, Unix or Linux computer. If you're using Windows, you will need to make appropriate adjustments. Whatever system you use, make sure it's a lightly loaded system that's not being used by others, in order to ensure that you get reasonably accurate performance measurements. So do *not* use the CEC servers for this. If you don't have access to such a system, you can reserve a Linux server in the Open Network Lab, www.onl.wustl.edu. Reserve a single *pc1core* server and ssh to that server in order to run your experiments.)

   ```
   checkPrim verbose <wg8
   ```

   Paste a copy of the output below.

   ```
   (a,d,1) (d,g,0) (a,e,4) (g,h,12) (b,h,5) (c,e,13) (f,h,17)
   stats 3 4 2 8
   ```

   Verify that the list of edges does represent a minimum spanning tree of the graph in file *wg8*.

   The list of edges does represent a minimum spanning tree of the graph

3. (5 points) Also, run *checkPrim* on the graphs in files *wg10*, *wg25* and *wg100*, but in this case, do *not* use the verbose argument. Paste the output from your three runs below (they should be one line each).

```
stats 5 8 5 13
stats 35 71 48 22
stats 137 417 396 62
```

4. (5 points) In the *lab1* directory, you will find programs *badcase.cpp* and *worsecase.cpp*. These generate graphs using the methods in the *common.cpp* file that you wrote. Generate a *badcase* graph by typing the command

```
badcase 6 12
```

Paste the resulting graph below.

```
{
[a: b(1) c(2) d(3) e(4) f(5)]
[b: a(1) c(1) e(3) f(4)]
[c: a(2) b(1) d(1) e(2)]
[d: a(3) c(1) e(1)]
[e: a(4) b(3) c(2) d(1) f(1)]
[f: a(5) b(4) e(1)]
}
```

5. (5 points) Generate a *worsecase* graph by typing the command

```
worsecase 6 12
```

Paste the resulting graph below.

```
{
[a: b(1) c(33) d(32) e(31) f(30)]
[b: a(1) c(1) e(25) f(24)]
[c: a(33) b(1) d(1) e(19)]
[d: a(32) c(1) e(1)]
[e: a(31) b(25) c(19) d(1) f(1)]
[f: a(30) b(24) e(1)]
}
```

6. (10 points) Discuss how the edge weights differ in these two cases and explain the effect that difference can be expected to have on the performance of Prim's algorithm. You will need to refer to the source code for Prim's algorithm and *Dheap* in order to answer this question. Hint: the order of edges in the adjacency lists matters.

For the badcase:

for (e = wg.firstAt(1); e != 0; e = wg.nextAt(1,e))

u = wg.mate(1,e); nheap.insert(u,wg.weight(e)); cheap[u] = e;

We will insert n-1 vertex into the heap. The weight in the adjacent list is not necessarily reversely sorted, so it is not really bad here. However when a new vertex u is deletemin from heap, then we iterate through the adjacency list of u, say the vertex in it is v, but w(u,v) is smaller, so we have to do the decrease key operation many times.

nheap.changekey(v,wg.weight(e)); cheap[v] = e;

Both decrease key and insert operation will cause siftup operation in the heap, for the badcase the dominate factor is the decease key operation.

For the worsecase: The weight in the adjacent list is reversely sorted, that mean first when we insert the vertices into heap we will trigger siftup operation as many as we can every time when we insert a vertex. Also when a new vertex u is deletemin from heap, then we iterate through the adjacency list of u, say the vertex in it is v, but w(u,v) is smaller, so we have to do the decrease key operation. But worse case is different from bad case, the weight of vertices in each adjacent list is reversely sorted, that means when we do the decrease key operation, we have to do more siftup to make the smaller key to the top than the bad case. Both decrease key and insert operation will cause siftup operation in the heap, for the worse the dominate factor is the both insert and decrease key operation.

7. (10 points) Examine the program *evalPrim.*cpp and make sure you understand what it does. Then, compile and run it by typing

```
evalPrim 32 100 2
```

Paste the results below.

```
random 32 100 2 31.9 (25,36) 65.9 (53,87) 71.5 (65,76) 14.7 (13,19)

badcase 32 100 2 69 (69,69) 42.9 (33,50) 81.7 (78,85) 11.3 (10,15)

worsecase 32 100 2 69 (69,69) 206.9 (198,213) 80.4 (77,82) 14.5 (12,23)
```

For the graphs generated by this *evalPrim* run, give a tight numerical upper bound on the number of levels in the heap used by Prim's algorithm, based on the worst-case analysis.

$\lceil log_2 100 \rceil$ = *7*

Estimate the average number of *siftup* steps that were executed per *changekey* operation for the *random* graphs. Repeat for the *badcase* and *worsecase* graphs.

random (65.9/2)/31.9 = 1.0

badcase (42.9/2)/69 = 0.3

worsecase (206.9/2)/69 = 1.5

Estimate the average number of *siftdown* steps per *deletemin* operation for each of the three types of graphs.

random 71.5/31 = 2.3

badcase 81.7/31 = 2.8

worsecase 80.4.7/31 = 2.6

What is the average running time (in microseconds), for each of the three graphs?

random 14.7

badcase 11.3

worsecase 14.5

Part C. Evaluating performance as the number of vertices increases.

1. (10 points) Run the provided *script1* and use the average count values to complete the middle columns of the table below. Note that the table has separate sections for *random*, *badcase* and *worsecase* graphs. Note that the vertex and edge counts shown at left are in thousands (so the first line is for 1,000 vertices and 64,000 edges). Enter the count data similarly, to make the numbers easier to compare. In the columns labeled *ratios*, you are to add data showing how the count values grow from one row to the next. So for example, in the second row, under *changekey*, enter the ratio of the second row *changekey* count to the first row *changekey* count. In the third row, under *changekey*, enter the ratio of the third row *changekey* count to the second row *changekey* count, and so forth. The first row in each delta section should be left blank. (You may find it more convenient to import the data produced by *script1* into a spreadsheet, use the spreadsheet to compute the ratios, then format a table like the one shown below and copy it from the spreadsheet to this Word document. If you choose to do this, make sure that your tables is formatted just like the one below, including column headers and so forth. And of course, delete the provided table in that case.)

			counts (1000s)				*ratios*			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*	*changekey*	*siftup*	*siftdown*	*runtime*
random										
1	64	2	4.07	8.14	7.51	6.07				
2	128	2	8.1	17.3	17	20.6	1.98	2.13	2.27	3.378
4	256	2	16.3	37.4	38.1	66.2	2.01	2.16	2.24	3.21
8	512	2	32.6	80	84.2	171.5	2	2.14	2.21	2.60
16	1024	2	65.3	171.2	184.4	336.5	2.00	2.14	2.19	1.96
badcase										
1	64	2	63	27.73	7.48	7.26				
2	128	2	126	73.2	17	22.6	2	2.64	2.27	3.10
4	256	2	252	182.3	38	73.9	2	2.50	2.26	3.27
8	512	2	504	437.4	83.9	174	2	2.40	2.21	2.35
16	1024	2	1008	1019.2	183.8	369.2	2	2.33	2.19	2.12
worsecase										
1	64	2	63	255.7	7.5	10.2				
2	128	2	126	626.8	17.2	30.1	2	2.45	2.29	2.95
4	256	2	252	1493.4	38.3	92.3	2	2.38	2.23	3.07
8	512	2	504	3478.4	84.6	225.9	2	2.38	2.21	2.48
16	1024	2	1008	7957	448.2	448.2	2	2.29	2.19	1.98

2. (10 points) The worst-case analysis for Prim's algorithm includes an upper bound on the number of *changekey* operations. Give an expression for this bound in terms of the number of vertices and edges (*n, m*). How does the experimental data compare to the worst-case bound in each of the three cases? Try to explain any differences you observe.

   changekey = $m - (n\text{-}1) = O(m - n)$

For the random case in the experiment, the number of changekey is 6% - 7% of the upperbound, it means, actually there is not much changekey operation in the random graph. However for the bad case and worse case, there will be maximum number of changekey involved.

3. (10 points) Now, consider the ratios. Note that the number of vertices and edges doubles from one row to the next. Based on the worst-case analysis, by how much would you expect the number of *changekey* operations to grow from one row to the next? Justify your answer using the worst-case analysis. How does this compare to the ratios in the table? Try to explain any differences you observe.

   The changekey operation should double by the worst case analysis. Because the upper bound for the change key is the number of edges – number of vertices. For the all the cases in the table, the ration is approximately 2, even some of the case is not the worst case, but the ration is what we expected.

4. (10 points) Now consider the ratios for *siftupCount* and *siftdownCount*. By how much should *siftupCount* and *siftdownCount* increase from one row to the next? Justify your answer using the worst-case analysis. How does this compare to the ratios in the table? Try to explain any differences you observe.

   *By the worst case analysis for the siftupCount, it could be increase on when we insert vertex to heap or the changekey is called. The worst case should be number of insert * depth of the heap + changekey * depth of the heap. But we know insertion to heap is O(n*logn) operation, and changekey is O((m-n)* logn), total number of siftupCound = O(m *logn), so the ratio should be more than 2, but not too much. The data in the table shows that.*

   Siftdown will be increased when we do the extraction, the cost of it for the worst case is O(nlogn), so the ratio should be more than 2, but not too much. The data in the table shows that.

5. (10 points) Finally, consider the ratios for the *runtime* values. By how much should the runtime increase from one row to the next. Justify your answer using the worst-case analysis. How does this compare to the experimental data? Try to explain any differences you observe.

   *By worst case analysis, the running time for the Prim is O((m+n)*logn), we know the ratio should be more than 2, but not too much. The data in the table shows that.*

Part D. Evaluating performance as the number of edges increases.

1. (10 points) Run the provided *script2* and use the average count values to complete the middle columns of the table below. Compute the deltas as in the previous part.

			counts (1000s)				*ratios*			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*	*changekey*	*siftup*	*siftdown*	*runtime*
random										
4	16	2	5.2	28	36.8	2.8				
4	64	2	10.7	33.1	37.9	7.9	2.06	1.18	1.03	2.82
4	256	2	16.3	37.4	38.1	69	1.52	1.13	1.01	8.73
4	1024	2	21.8	41.6	38.1	342.4	1.34	1.11	1	5.0
4	4096	2	27.3	46.7	38.1	1524	1.25	1.12	1	4.45
badcase										
4	16	2	12	37.4	38	2.2				
4	64	2	60	77	38	8.5	5	2.06	1	3.86
4	256	2	252	182.3	38	76.6	4.2	2.37	1	9.01
4	1024	2	1020	415.4	38	356.1	4.05	2.28	1	4.65
4	4096	2	4092	386	38	1611	4.01	0.93	1	4.5
worsecase										
4	16	2	12	134.9	37.4	3.5				
4	64	2	60	467	38.1	13.3	5	3.46	1.02	3.8
4	256	2	252	1491.2	38.3	97.7	4.2	3.19	1.01	7.35
4	1024	2	1020	4322.1	38.4	425.1	4.05	2.9	1.0	4.35
4	4096	2	4092	36935.4	38.4	2067.7	4.01	8.55	1	4.87

2. (10 points) The worst-case analysis for Prim's algorithm includes an upper bound on the number of *siftup* steps. Give an expression for this bound in terms of the number of vertices and edges (*n, m*). How does the experimental data compare to the worst-case bound in each of the three cases? Try to explain any differences you observe.

   *The worst case should be number of insert * depth of the heap + changekey * depth of the heap. We inserting building heap is O(n*logn) operation, and changekey is O((m-n)* logn), so the worst case siftup is O(m*logn). The experiment data is a little different, it should be approximately 4 time increase, but there is not. I guess it is because even we change the key of a vertex in the heap, there will not be that many siftup operations, maybe sometime a changekey is going to trigger a few or no siftup operations, the random case have fewest siftup while badcase has more, and worst case has most, the ratio is like that.*

3. (10 points) Now, consider the ratios. Note that the number of edges increases by a factor of 4 from one row to the next. Based on the worst-case analysis, by how much would you expect the number of *changekey* operations to grow from one row to the next? Justify your answer using the worst-case analysis. How does this compare to the experimental data? Try to explain any differences you observe.

The worst case will be 4 times than the previous data. Because the number of changekey = O(m - n). However, in the experiment, for the random case, the ratio is between 1 and 2, the bad case and worse cases are around 4. The change key operation really depends on how the weight of the graph is organized. For the random case, it is just random organized; the bad case is really bad, worse case is more worse, so there are more change key operations.

4. (10 points) Now consider the ratios for *siftupCount* and *siftdownCount*. By how much should *siftupCount* and *siftdownCount* increase from one row to the next. Justify your answer using the worst-case analysis. How does this compare to the experimental data? Try to explain any differences you observe.

   *By the worst case analysis for the siftupCount, it could be increase on when we insert vertex to heap or the changekey is called. The worst case should be number of insert * depth of the heap + changekey * depth of the heap. But we know insertion to heap is O(n*logn) operation, and changekey is O((m-n)* logn), total number of siftupCound = O(m*logn), so the ratio should be around 4. The ratio in the table is less than 4. I guess it is because even we change the key of a vertex in the heap or we insert a vertex to the heap, there will not be that many siftup operations, maybe sometime a changekey is going to trigger a few or no siftup operations.*

   Siftdown will be increased when we do the extraction, the cost of it for the worst case is O(nlogn), so the ratio should not change much. The data in the table shows that.

5. (10 points) Finally, consider the ratios for the *runtime* values. By how much should the *runtime* increase from one row to the next. Justify your answer using the worst-case analysis. How does this compare to the experimental data? Try to explain any differences you observe.

   *By worst case analysis, the running time for the Prim is O((m+n)*logn), since m is the dominate factor, much bigger than n, so we know the ratio should be around 4, and the data in the table shows that.*

Part E. Effect of d on performance. (10 points)

1. (10 points) Run the provided *script3* and use the average count values to complete the middle columns of the table below. You need not compute deltas for this part.

			counts (1000s)			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*
random						
4	256	2	16.2	37.4	38.1	64.9
4	256	4	16.2	18.2	61.5	71.5
4	256	8	16.2	11.4	99.4	64.1
4	256	16	16.2	8	164.4	64.6
4	256	64	16.2	4.7	477.8	67.4
badcase						
4	256	2	252	182.3	37.9	71.6
4	256	4	252	146.7	61.2	67.9
4	256	8	252	117.4	99.1	67.1
4	256	16	252	78.4	160	68.7
4	256	64	252	26.2	430	66.2
worsecase						
4	256	2	252	1493.4	38.3	88.3
4	256	4	252	721.7	62.5	88.3
4	256	8	252	451.5	101.6	72.2
4	256	16	252	310	166.2	71.7
4	256	64	252	171.3	470.6	70.9

2. (5 points) For each row in the table, give a (tight) numerical upper bound on the number of levels in the heap.

   $\lceil log_2 4000 \rceil$ = 12

   $\lceil log_4 4000 \rceil$ = 6

   $\lceil log_8 4000 \rceil$ = 4

   $\lceil log_{16} 4000 \rceil$ = 3

   $\lceil log_{64} 4000 \rceil$ = 2

3. (5 points) For each of the three cases, compute the ratio of *siftupCount* to *changekeyCount* for *d*=2 and *d*=64.

			counts (1000s)				*ratios*			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*	*changekey*	*siftup*	*siftdown*	*runtime*
random										
4	256	2	16.2	37.4	38.1	64.9				

4	256	64	16.2	4.7	477.8	67.4	1	8.0		
badcase										
4	256	2	252	182.3	37.9	71.6				
4	256	64	252	26.2	430	66.2	1	7.0		
worsecase										
4	256	2	252	1493.4	38.3	88.3				
4	256	64	252	171.3	470.6	70.9	1	8.7		

4. (5 points) For each of the three cases, consider how *siftupCount* and *siftdownCount* change with *d*. We would expect the smallest overall runtime for the value of *d* that minimizes *siftupCount*+*siftdownCount*. Are the data consistent with that expectation?

   When d increases, the siftup operation will decrease because the depth of the heap decrease, but the siftdown opearation will increase because the depth of the heap decreases too. Based on the value of the experiment, when d increase, the running time will decrease more rapidly on the graph which will involve more siftup operation, the best value should be 2 + m/n.

5. (5 points) Compare the sensitivity of the runtime to *d* in each of the three cases. How significant is the improvement in each case? Based on this data, how important do you think it is (as a practical matter) to adjust *d* as a function of *m* and *n?*

   For the random case, increase the value of d, the performance is not guaranteed. For the bad and worse case, when d = 64, we can find the minimum running time. When d increase, the running time will decrease more rapidly on the graph which will involve more siftup operation, the best value should be d = 2 + m/n for general purpose regardless of the graph.