CSE 542 – Advanced Data Structures and Algorithms

Lab 1 Solution

Jon Turner

Part A. Modifications to Dheap (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 source 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.

```
...
class Dheap : public Adt {
public:    Dheap(int,int);
    ...
    // performance statistics
    int changekeyCount;
    int siftupCount;
    int siftdownCount;
private:
    ...
};

...

/** Constructor for Dheap class.
 *  @param size is the number of items in the contructed object
 *  @param D1 is the degree of the underlying heap-ordered tree
 */
Dheap::Dheap(int size, int dd) : Adt(size), d(dd) {
        makeSpace(size);
        changekeyCount = siftupCount = siftdownCount = 0;
}
...

/** Perform siftup operation to restore heap order.
 *  This is a private helper function.
 *  @param i is the index of an item to be positioned in the heap
 *  @param x is a tentative position for i in the heap
 */
void Dheap::siftup(index i, int x) {
      int px = p(x);
      siftupCount++;  // update performance stats
      while (x > 1 && kee[i] < kee[h[px]]) {
            siftupCount++;  // ditto
            h[x] = h[px]; pos[h[x]] = x;
            x = px; px = p(x);
```

```
     }
     h[x] = i; pos[i] = x;
}

/** Find the position of the child withthe smallest key.
 *  This is a private helper function, used by siftdown.
 *  @param x is a position of an item in the heap
 *  @return the position of the child of the item at x, that has
 *  the smallest key
 */
int Dheap::minchild(int x) {
     int y; int minc = left(x);
     siftdownCount++;      // update performance stats
     if (minc > hn) return 0;
     for (y = minc + 1; y <= right(x) && y <= hn; y++) {
          siftdownCount++;      // ditto
          if (kee[h[y]] < kee[h[minc]]) minc = y;
     }
     return minc;
}

/** Change the key of an item in the heap.
 *  @param i is the index of an item in the heap
 *  @param k is a new key value for item i
 */
void Dheap::changekey(index i, keytyp k) {
     changekeyCount++;     // update performance stats
     keytyp ki = kee[i]; kee[i] = k;
     if (k == ki) return;
     if (k < ki) siftup(i,pos[i]);
     else siftdown(i,pos[i]);
}

...
```

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.

```
// class used to return performance statistics for Prim's algorithm
class PrimStats {
        public:
        int     ckCount;        // changekeyCount from Dheap
        int     suCount;;       // siftupCount from Dheap
        int     sdCount;;       // siftdownCount from Dheap
        int     runtime;;       // total runtim of Prim

        /** Set all counters to specified value. */
        void set(int x) { ckCount = suCount = sdCount = runtime = x; }
};

/** Find a minimum spanning tree using Prim's algorithm.
 *  @param wg is a reference to a weighted graph object
 *  @param mstree is a reference to a list used to return
 *  the edges in the mst; it is assumed to be empty when prim is called
 *  @param d is the heap parameter for the heap used by Prim's algorithm.
 *  @param stats is a reference to a PrimStats object in which performance
 *  statistics are returned.
 */

void prim(Wgraph& wg, list<int>& mstree, int d, PrimStats& stats) {
    int t0 = Util::getTime();   // record start time
    vertex u,v; edge e;
    edge *cheap = new edge[wg.n()+1];
    bool *intree = new bool[wg.n()+1];
    Dheap nheap(wg.n(),d);

    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;
            }
        }
    }
    delete [] cheap;
    int t1 = Util::getTime();   // record elapsed time
    stats.runtime = t1 - t0;
    // record performance stats from Dheap
    stats.ckCount = nheap.changekeyCount;
    stats.suCount = nheap.siftupCount;
    stats.sdCount = nheap.siftdownCount;
}
```

```
/** Generate a weighted graph on which Prim's algorithm performs poorly.
 *  @param n is the desired number of vertices
 *  @param m is the desired number of edges
 *  @param wg is a reference to a weighted graph object
 *
 *  The graph is generated by first creating edges from vertex 1 to
 *  every other vertex, and then from each vertex u >= 2 to u+1.
 *  Random edges are then added to bring the total to m (so m >= 2*n-3).
 *  Edge weights for edges (u,v) with u<v are set to v-u.
 */
void badcase(int n, int m, Wgraph& wg) {
    m = max(m,2*n);
    wg.resize(n,m);
    for (vertex u = 2; u <= n; u++) {
        wg.join(1,u);
        if (u > 2) wg.join(u-1,u);
    }
    wg.addEdges(m);
    for (edge e = wg.first(); e != 0; e = wg.next(e)) {
        vertex u = min(wg.left(e),wg.right(e));
        vertex v = wg.mate(u,e);
        wg.setWeight(e,v-u);
    }
}

/** Generate a weighted graph on which Prim's algorithm performs more
poorly.
 *  @param n is the desired number of vertices
 *  @param m is the desired number of edges
 *  @param wg is a reference to a weighted graph object
 *
 *  The graph is generated as in badcase.
 *  Edge weights for edges (u,v) with u<v are set to n*(n-u)+(n-v).
 */
void worsecase(int n, int m, Wgraph& wg) {
    m = max(m,2*n);
    wg.resize(n,m);
    for (vertex u = 2; u <= n; u++) {
        wg.join(1,u);
        if (u > 2) wg.join(u-1,u);
    }
    wg.addEdges(m);
    for (edge e = wg.first(); e != 0; e = wg.next(e)) {
        vertex u = min(wg.left(e),wg.right(e));
        vertex v = wg.mate(u,e);
        if (v == u+1) wg.setWeight(e,1);
        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 source 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) e(4)]
[b: d(22) f(27) h(5)]
[c: d(22) e(13) g(24)]
[d: a(1) b(22) c(22) g(0) h(30)]
[e: a(4) c(13)]
[f: b(27) h(17)]
[g: c(24) d(0) h(12)]
[h: b(5) d(30) f(17) g(12)]
}

(a,d,1) (d,g,0) (a,e,4) (g,h,12) (b,h,5) (c,e,13) (f,h,17)
stats 3 14 12 7
```

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

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 22 18 7
stats 35 130 119 15
stats 137 653 886 53
```

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.

   Badcase and worsecase are both designed to incur the maximum possible number of changkey operations. For badcase, we also incur nearly the largest possible number of siftup steps. Whenever an edge {u,v} is processed by Prim's algorithm, the old key value for v is the largest in the heap, while after the changekey, it is either the smallest or second smallest. Consequently, these nodes are at the bottom of the heap before the changekey and move up to nearly the top of the heap, during the changekey.

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) 128.8 (113,147) 173.6 (162,183) 14.3 (13,17)
     badcase 32 100 2 69 (69,69) 142.9 (133,150) 191.8 (182,198) 11.1 (10,14)
   worsecase 32 100 2 69 (69,69) 306.9 (298,313) 190.3 (186,196) 12.1 (10,21)
   ```

   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.

   Since the base of the heap is 2 in this case, the number of levels is at most $\log_2 n=5$.

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

   Siftup is used by both insert and changkey. There are $n-1=31$ inserts plus 32 changekeys in the case of random graphs. Assuming the siftup steps are divided evenly between the two, we have about 2 siftup operations per changekey.

   *For the badcase graphs, there are about twice as many changekeys as there are inserts, so about 2/3 of the siftup steps are associated with changekeys. This gives us an estimate of (2/3)*143/69 which is about 4/3.*

   *For the worsecase graphs, we have about (2/3)*307/69 which is about 3.*

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

   There are 31 deletemin operations, so the number of siftdown steps per deletemin operation are about 5.3, 6 and 6 in the three cases.

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

   14.3 μs, 11.1μs and 12.1 μs.

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 (x1000)				*ratios*			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*	*changekey*	*siftup*	*siftdown*	*runtime*
random										
1	64	2	4.1	13.2	15.9	4.0				
2	128	2	8.1	27.4	35.8	9.8	2.00	2.08	2.25	2.44
4	256	2	16.3	57.7	79.6	38.2	2.00	2.10	2.22	3.89
8	512	2	32.6	120.5	175.1	122.8	2.00	2.09	2.20	3.22
16	1024	2	65.3	252.5	382.3	314.7	2.01	2.10	2.18	2.56
badcase										
1	64	2	63.0	91.7	15.8	4.2				
2	128	2	126.0	201.2	35.7	12.7	2.00	2.19	2.25	3.02
4	256	2	252.0	438.3	79.3	41.5	2.00	2.18	2.22	3.27
8	512	2	504.0	949.4	174.7	126.9	2.00	2.17	2.20	3.05
16	1024	2	1008.0	2043.2	381.3	323.0	2.00	2.15	2.18	2.55
worsecase										
1	64	2	63.0	319.7	15.9	6.9				
2	128	2	126.0	754.8	35.9	18.0	2.00	2.36	2.25	2.61
4	256	2	252.0	1749.4	79.7	52.1	2.00	2.32	2.22	2.89
8	512	2	504.0	3990.4	175.4	159.9	2.00	2.28	2.20	3.07
16	1024	2	1008.0	8981.1	382.8	394.6	2.00	2.25	2.18	2.47

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.

   According to the analysis, the worst-case number is m, so for the graph sizes here, that's 64K, 128K, 256K, 512K and 1024K.

The random graphs produce results that are much smaller. There are two reasons. First, the number of changekeys is much smaller than the m predicted by the analysis. Second, the number of siftup steps per changekey is much smaller. With random edges (and edge weights), most new edges examined by the algorithm have a larger weight than those already examined. Consequently, they do not require a call changekey. For those that do require a call to changekey, the reduction in edge weight is often small enough that the vertices stored in the heap do not move very far up in the heap.

For the badcase and worsecase graphs, the number of changekeys is fairly close to the worst-case bound.

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 number of changekeys should grow in proportion to the number of edges. So, it should double in each case. Surprisingly, this is exactly what happens, even for the random graphs. So even though the number of changekeys is far below the upper for random graphs, the growth rate is consistent with the worst-case analysis.

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.

   These should grow a little faster than a factor of 2, since the heap is getting deeper at each step. The data shows that siftupCount and siftdownCount grow by a factor of 2.2 on each step, again implying that the growth rate is consistent with the worst-case analysis.

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.

   These should also grow a little faster than a fact of 2 from one row to the next. Surprisingly, the growth rate is faster than that. The most plausible explanation is that the cache performance is deteriorating as the graphs get larger.

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 (x1000)				*ratios*			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*	*changekey*	*siftup*	*siftdown*	*runtime*
random										
4	16	2	5.2	37.2	76.9	2.0				
4	64	2	10.7	47.9	79.2	5.3	2.07	1.29	1.03	2.74
4	256	2	16.3	57.7	79.6	35.6	1.52	1.20	1.01	6.65
4	1024	2	21.8	67.4	79.7	311.3	1.34	1.17	1.00	8.76
4	4096	2	27.4	78.1	79.6	1399.5	1.25	1.16	1.00	4.50
badcase										
4	16	2	12.0	53.5	79.6	1.5				
4	64	2	60.0	141.0	79.4	5.5	5.00	2.64	1.00	3.62
4	256	2	252.0	438.3	79.3	39.3	4.20	3.11	1.00	7.17
4	1024	2	1020.0	1439.4	79.3	329.9	4.05	3.28	1.00	8.40
4	4096	2	4092.0	4482.0	79.4	1467.6	4.01	3.11	1.00	4.45
worsecase										
4	16	2	12.0	150.9	79.3	2.2				
4	64	2	60.0	531.0	79.5	8.9	5.00	3.52	1.00	4.02
4	256	2	252.0	1749.4	79.7	51.2	4.20	3.29	1.00	5.74
4	1024	2	1020.0	5346.2	79.9	369.7	4.05	3.06	1.00	7.21
4	4096	2	4092.0	41031.4	80.0	1783.3	4.01	7.67	1.00	4.82

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 number of siftup steps is at most (m+n)log$_d$n, or 20K*12=240K, 68K*12=792K, 260K*12=3M, 1028K*12=12M, 4100K*12=49M. The random and badcase graphs have much smaller siftup counts, but the worsecase graphs come reasonably close. This is consistent with what we saw earlier.*

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.

   We expect the values to grow by a factor of 4 from one row to the next. This is mostly true for the badcase and worsecase graphs. There is an anomaly in the first row that has no obvious explanation. For the random graphs, the growth rate is much smaller than expected, so in this case, the growth *rate is much smaller than the analysis suggests. It appears that as the number of edges per vertex increases, an ever smaller fraction of the edges examined leads to a reduction in the key value in the heap.*

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.

   The siftup steps should also grow by a factor of 4 at each step, but we see that they fall somewhat short in most cases. The biggest discrepancy is for the random graphs, and this is to be expected given the slower growth rate in the number of changekeys. For the badcase and worsecase graphs, it appears that the heap entries are not moving as far up the heap during each changekey. It's not clear why that should be.

   The number of siftdown steps should remain constant in this case and the ratios confirm this.

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.

   Again, we expect a factor of 4 in each row. The observed values are mostly larger than this. Again, the cache performance is the most likely explanation.

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 (x1000)			
n	*m*	*d*	*changekey*	*siftup*	*siftdown*	*runtime*
random						
4	256	2	16.3	57.7	79.6	36.7
4	256	4	16.3	38.4	85.7	35.3
4	256	8	16.3	31.7	117.4	35.9
4	256	16	16.3	28.3	179.2	38.1
4	256	64	16.3	25.0	489.5	36.9
badcase						
4	256	2	252.0	438.3	79.3	39.3
4	256	4	252.0	402.7	85.4	39.5
4	256	8	252.0	373.4	117.2	38.9
4	256	16	252.0	334.4	174.9	40.2
4	256	64	252.0	282.2	441.4	37.7
worsecase						
4	256	2	252.0	1749.4	79.7	55.8
4	256	4	252.0	977.7	86.8	46.4
4	256	8	252.0	707.5	120.0	42.3
4	256	16	252.0	566.0	181.2	42.4
4	256	64	252.0	427.3	482.2	40.6

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

   The number of levels drops as d increases, so in these cases we expect 12, 6, 4, 3 and 2, if we define the number of levels as the length of the longest path from a node to the root the heap.

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

   For random, the ratios are approximately 3.6 and 1.6. For badcase they are 1.7 and 1.1. For worsecase, they are 6.9 and 1.7.

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?

   For random and badcase, there is no strong relationship between this sum and the running time. In general, the runtimes are fairly insensitive to the change in d. For the worsecase graphs, the data is reasonably consistent with the expectation, although the match is not perfect.

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?*

   The only case in which the choice of d seems to matter is the worsecase graphs. In this case, we get almost a 30% reduction in runtime going from d=2 to d=64. While this is not huge, it's something. One could argue that since the worsecase graphs are highly contrived, there is little chance of this case arising in practice. On the other hand, there appears to be no downside to adjusting d to the graph density and since it does occasionally help, it's not unreasonable to include this adjustment in the implementation of Prim's algorithm.