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

Lab 2 Report

*Your name here:*_____ *Due 2/19/2013*

Part A. Modifications to augPath, shortPath

1. (15 points) Paste a copy of your changes to *augPath* and *shortPath* below. Highlight your changes by making them bold. You may omit methods you did not change.

   Here is the change to augPath.h.

```
class augPath {
public:
        augPath(Flograph&,int&);
        ~augPath();

        // statistics counters
        int     fpCount;
        int     fpSteps;
        int     augCount;
        int     augSteps;
        int     runtime;
protected:
        Flograph* fg;           ///< graph we're finding flow on
        edge *pEdge;            ///< pEdge[u] is edge to parent of u in spt

        int     augment();
        virtual bool findPath() = 0;    ///< find augmenting path
};
```

   Here's the change to augPath.cpp.

```
int augPath::augment() {
// Saturate the augmenting path p.
        vertex u, v; edge e; flow f;

        augCount++;
        // determine residual capacity of path
        f = Util::BIGINT32;
        v = fg->snk(); e = pEdge[v];
        while (v != fg->src()) {
                u = fg->mate(v,e);
                f = min(f,fg->res(u,e));
                v = u; e = pEdge[v];
        }
        // add flow to saturate path
        v = fg->snk(); e = pEdge[v];
        while (v != fg->src()) {
                augSteps++;
                u = fg->mate(v,e);
                fg->addFlow(u,e,f);
                v = u; e = pEdge[v];
```

```
        }
        return f;
}
```

Here is the change to shortPath.cpp.

```
shortPath::shortPath(Flograph& fg1, int& floVal) : augPath(fg1,floVal) {
// Find maximum flow in fg using the shortest augment path algorithm.
 int t0 = Util::getTime();
 floVal = 0;
 while(findPath()) {
     floVal += augment();
 }
 int t1 = Util::getTime();
 runtime = t1 - t0;
}

bool shortPath::findPath() {
// Find a shortest path with unused residual capacity.
 vertex u,v; edge e;
 List queue(fg->n());

 fpCount++;
 for (u = 1; u <= fg->n(); u++) pEdge[u] = 0;
 queue.addLast(fg->src());
 while (!queue.empty()) {
     u = queue.first(); queue.removeFirst();
     for (e = fg->firstAt(u); e != 0; e = fg->nextAt(u,e)) {
         fpSteps++;
         v = fg->mate(u,e);
         if (fg->res(u,e) > 0 && pEdge[v] == 0 &&
             v != fg->src()) {
             pEdge[v] = e;
             if (v == fg->snk()) {
                 return true;
             }
             queue.addLast(v);
         }
     }
 }
 return false;
}
```

2. (15 points) Compile the provided code in your lab1 directory using the makefile. Verify your changes to *augPath* and *shortPath* using the command *checkSpath* by typing

```
checkSpath verbose <random10
checkSpath <random20
checkSpath <random50
checkSpath verbose <hard3
checkSpath <hard10
```

Paste a copy of your output below.

```
checkSpath verbose <random10
{
[b: c(17,0) d(18,0) f(16,16) h(19,0)]
[c: d(11,11) e(5,0)]
[d: a(1,0) g(14,11) h(11,0)]
[e: a(12,0) c(7,0)]
```

```
[f: h(2,0) j(60,16)]
[g: c(14,0) d(3,0) j(70,11)]
[h: f(17,0) g(5,0)]
[i->: b(16,16) c(82,11)]
[->j:]
}

stats 27 3 60 2 7 6
% checkSpath <random20
stats 58 11 562 10 45 18
% checkSpath <random50
stats 524 30 12046 29 159 218

% checkSpath verbose <hard3
{
[1->: 2(9,9) 6(9,9) 10(9,9) 14(9,9) 18(9,9) 22(9,9)]
[2: 3(27,9)]
[3: 4(27,9)]
[4: 5(27,9)]
[5: 6(27,9)]
[6: 7(27,18)]
[7: 8(27,18)]
[8: 9(27,18)]
[9: 10(27,18)]
[10: 11(27,27)]
[11: 12(27,27)]
[12: 13(27,27)]
[13: 28(9,9) 29(9,9) 30(9,9)]
[14: 15(27,9)]
[15: 16(27,9)]
[16: 17(27,9)]
[17: 18(27,9)]
[18: 19(27,18)]
[19: 20(27,18)]
[20: 21(27,18)]
[21: 22(27,18)]
[22: 23(27,27)]
[23: 24(27,27)]
[24: 25(27,27)]
[25: 26(27,27)]
[26: 27(27,27)]
[27: 31(9,9) 32(9,9) 33(9,9)]
[28: 31(1,0) 32(1,0) 33(1,0) 34(9,9)]
[29: 31(1,0) 32(1,0) 33(1,0) 34(9,9)]
[30: 31(1,0) 32(1,0) 33(1,0) 34(9,9)]
[31: 48(9,9)]
[32: 48(9,9)]
[33: 48(9,9)]
[34: 35(27,27)]
[35: 36(27,27)]
[36: 37(27,27)]
[37: 38(27,27)]
[38: 39(27,27)]
[39: 40(27,18) 60(9,9)]
[40: 41(27,18)]
[41: 42(27,18)]
[42: 43(27,18)]
[43: 44(27,9) 60(9,9)]
[44: 45(27,9)]
```

```
[45: 46(27,9)]
[46: 47(27,9)]
[47: 60(9,9)]
[48: 49(27,27)]
[49: 50(27,27)]
[50: 51(27,27)]
[51: 52(27,18) 60(9,9)]
[52: 53(27,18)]
[53: 54(27,18)]
[54: 55(27,18)]
[55: 56(27,9) 60(9,9)]
[56: 57(27,9)]
[57: 58(27,9)]
[58: 59(27,9)]
[59: 60(9,9)]
[->60:]
}

stats 54 55 7272 54 1134 241

% checkSpath <hard10
stats 2000 2001 1153760 2000 98000 18112
```

Part B. FaugPath and fshortPath.

1. (30 points) Paste a copy of your code for *faugPath* and *fshortPath* below. Highlight your changes by making them bold. You may omit methods you did not change.

```
/** Find maximum flow in a flow graph.
 *  Base class constructor initializes dynamic data common to
 *  all algorithms. Constructors for derived classes actually
 *  implement specific algorithms.
 */
faugPath::faugPath(Flograph& fg1, int& flow_value) : fg(&fg1) {
    pEdge = new edge[fg->n()+1];
    d = new int[fg->n()+1];
    fpCount = fpSteps = augCount = augSteps = 0;
    runtime = 0;
}

/** Saturate augmenting paths.
 *  This method uses the pEdge array to discover as many augmenting
 *  paths as it can.
 */
int faugPath::augment() {
    flow f, fsum;
    augCount++;
    fsum = 0;
    while ((f = reaugment()) > 0) { fsum += f; }
    return fsum;
}

/** Try to augment a path and if successful, patch the pEdge array.
 *  This method follows edges in the pEdge array back towards the sink.
 *  If an augmenting path is found, it adds flow to the path and attempts
 *  to patch the pEdge array, in order to replace saturated edges with
 *  replacement edges that may define another augmenting path.
 *  @return the amount of flow added to the path (0 if no path found)
 */
int faugPath::reaugment() {
    vertex u, v; edge e; flow f;

    // determine residual capacity of path
    f = Util::BIGINT32;
    v = fg->snk(); e = pEdge[v];
    while (v != fg->src() && e != 0) {
        u = fg->mate(v,e);
        f = min(f,fg->res(u,e));
        v = u; e = pEdge[v];
    }
    if (v != fg->src()) return 0;
    // add flow to saturate path
    v = fg->snk(); e = pEdge[v];
    while (v != fg->src()) {
        u = fg->mate(v,e);
        fg->addFlow(u,e,f);
        augSteps++;
        if (fg->res(u,e) == 0) {
            pEdge[v] = 0;
            for (edge ee = fg->firstAt(v); ee != 0;
                  ee = fg->nextAt(v,ee)) {
```

```
                vertex w = fg->mate(v,ee);
                augSteps++;
                if (fg->res(w,ee) > 0 && d[w] == d[u]) {
                    pEdge[v] = ee; break;
                }
            }
        }
        v = u; e = pEdge[v];
    }
    return f;
}
```

```
#include "fshortPath.h"

fshortPath::fshortPath(Flograph& fg1,int& floVal):faugPath(fg1,floVal) {
// Find maximum flow in fg using the shortest augment path algorithm.
        int t0 = Util::getTime();
        floVal = 0;
        while(findPath()) {
                floVal += augment();
        }
        int t1 = Util::getTime();
        runtime = t1 - t0;
}

bool fshortPath::findPath() {
// Find a shortest path with unused residual capacity.
        vertex u,v; edge e;
        List queue(fg->n());

        fpCount++;
        for (u = 1; u <= fg->n(); u++) { pEdge[u] = 0; d[u] = fg->n(); }
        d[fg->src()] = 0;
        queue.addLast(fg->src());
        while (!queue.empty()) {
                u = queue.first(); queue.removeFirst();
                for (e = fg->firstAt(u); e != 0; e = fg->nextAt(u,e)) {
                        fpSteps++;
                        v = fg->mate(u,e);
                        if (fg->res(u,e) > 0 && pEdge[v] == 0 &&
                            v != fg->src()) {
                                pEdge[v] = e; d[v] = d[u] + 1;
                                if (v == fg->snk()) return true;
                                queue.addLast(v);
                        }
                }
        }
        return false;
}
```

2. (15 points) Check your new classes by typing in the lab1 directory.

```
checkFspath verbose <random10
checkFspath <random20
checkFspath <random50
checkFspath <hard3
checkFspath <hard10
```

Paste a copy of your output below. Note that these should produce the same flows as in part1, although you will see differences in the performance counter values.

```
% checkFspath verbose <random10
{
[b: c(17,0) d(18,0) f(16,16) h(19,0)]
[c: d(11,11) e(5,0)]
[d: a(1,0) g(14,11) h(11,0)]
[e: a(12,0) c(7,0)]
[f: h(2,0) j(60,16)]
[g: c(14,0) d(3,0) j(70,11)]
[h: f(17,0) g(5,0)]
[i->: b(16,16) c(82,11)]
[->j:]
}

stats 27 3 60 2 22 5
% checkFspath <random20
stats 58 11 562 10 107 25
% checkFspath <random50
stats 524 20 7621 19 403 153
% checkFspath <hard3
stats 54 15 1852 14 1488 84
% checkFspath <hard10
stats 2000 183 104108 182 113880 4006
```

Part C. Evaluating performance on random graphs as the number of edges increases.

1. (10 points) Run the provided *script1* and use the data from the first half of the output file to complete the "count" columns of the table below. Note that the table has separate sections for *shortPath* and *fshortPath* graphs. To make the numbers easier to interpret, enter values like 34538 as 34.6K and values like 1234567 as 1.2M, and so forth. For each performance counter, compute the ratios of the values from one row to the next, as we did in lab 1.

		fpCount		*fpSteps*		*augCount*		*augSteps*		*runtime*	
n	*m*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*
shortPath											
200	800	46		62K		45		320		990	
200	1.6K	158	3.43	450K	7.23	157	3.49	861	2.69	5.8K	5.81
200	3.2K	504	3.19	2.9M	6.37	503	3.20	2.4K	2.76	35K	6.13
200	6.4K	1.6K	3.25	17M	6.07	1.6K	3.26	7K	2.93	311K	8.81
200	12.8K	4.1K	2.50	79M	4.53	4.1K	2.50	16K	2.35	1.4M	4.64
fshortPath											
200	800	34		45K		33		665		731	
200	1.6K	84	2.47	231K	5.07	83	2.52	2,565	3.86	3K	4.13
200	3.2K	166	1.98	909K	3.94	165	1.99	10K	3.97	11.4K	3.79
200	6.4K	216	1.30	2.2M	2.37	215	1.30	41K	3.99	29.3K	2.56
200	12.8K	226	1.05	4.1M	1.88	225	1.05	152K	3.73	78.6K	2.68

2. (5 points) Give an expression for the worst-case number of calls to *findpath* (in *shortPath*). How does this compare to the observed data? How does the growth rate of *fpCount* compare to the worst-case analysis?

   According to the worst-case analysis, there are at most mn calls to findpath, so here the values would range form 160 thousand to 2.56 million. The data shows 46 to 4.1 thousand, so only a tiny fraction of the predicted number.

   Concerning the growth rate, n is fixed, while m is doubling at each step, so we would expect fpCount to grow by a factor of 2. We observe a higher growth rate, so apparently as the graph becomes more dense, the number of path searches is growing more quickly than the number of edges (although it remains well below the absolute bound).

3. (5 points) Give a bound on the number of steps per call to *findpath* (in *shortPath*). How does this compare to the data in the table?

   We expect at most 2m steps per call to findpath, since each edge is considered at most two times over all iterations of the inner loop. For m=800, the observed values are (62K/46))≈1,300 vs a bound of 1,600. For m=1600, the observed values are (450K/158)≈2,848 vs a bound of 3,200. In general the data match the bounds fairly well.

4. (5 points) How would you expect the runtime of *shortPath* to grow (based on the worst-case analysis)? How does this compare to the data? Try to explain any differences you observe.

The worst-case runtime grows in proportion to m^2n, so we would expect a factor of 4 increase at each step. The observed growth rate is generally more than this. This is consistent with the fact that fpCount grows faster than the predicted rate.

5. (5 points) Compare the *fpCount* values and growth rates for *shortPath* vs. *fshortPath*. What does this tell you about the number of augmenting paths found during each execution of the *augment* method?

   The absolute difference is relatively modest for small graphs, but for the largest graphs, shortPath requires about 20 times as many calls as fshortPath. So, in this latter case, fshortPath is finding an average of 20 augmenting paths every time augment is called. What's most striking is that as the number of edges grows, this advantage is increasing.

6. (5 points). Compare the *augSteps* values for *shortPath* vs. *fshortPath.* Explain the observed differences. What are the implications of this comparison for the overall running time?

   The augSteps values are much larger fshortPath (nearly 10 times larger for the largest graphs). This makes sense, since fshortPath spends extra steps "patching" the pEdge array. Still, the augSteps values are much smaller than the fpSteps values, so overall we would still expect the running time for shortPath to be much smaller than for shortPath.

7. (5 points) Compare the *runtime* values for *shortPath* and *fshortPath*. Consider both the absolute values and the growth rate.

   We see only a modest advantage for small graphs, but for the largest graphs, fshortPath is almost 20 times faster than shortPath. The growth rates for fshortPath are also consistently smaller and generally less than the worst-case growth rate (which would give a factor of 4 per step). In contrast, shortPath grows by more than a factor of 4 per step, as noted above. Again, this is happening because shortPath's performance on these graphs is better than the absolute worst-case bounds. As m gets larger, its performance gets closer to the bounds.

Part D. Evaluating performance on random graphs as the number of vertices and edges both increase.

1. (10 points) Use the data from the second half of the *script1* output to complete the count columns of the table below. Compute ratios as before.

		fpCount		*fpSteps*		*augCount*		*augSteps*		*runtime*	
n	*m*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*
shortPath											
50	800	254		307K		253		1.1K		3.4K	
100	1.6K	430	1.69	1.1M	3.67	429	1.70	1.9K	1.79	13K	3.93
200	3.2K	504	1.17	2.9M	2.55	503	1.17	2.4K	1.23	35K	2.64
400	6.4K	575	1.14	6.9M	2.39	574	1.14	2.8K	1.20	90K	2.56
800	12.8K	622	1.08	15M	2.19	621	1.08	3.2K	1.13	285K	3.18
fshortPath											
50	800	55		63K		54		4.4K		758	
100	1.6K	109	1.98	271K	4.29	108	2.00	7.8K	1.79	3.4K	4.51
200	3.2K	166	1.52	909K	3.36	165	1.53	10K	1.31	11K	3.32
400	6.4K	212	1.28	2.5M	2.71	211	1.28	12K	1.20	32K	2.89
800	12.8K	265	1.25	6.2M	2.54	264	1.25	14K	1.17	119K	3.64

2. (5 points) How does the growth rate of *fpCount* compare to the worst-case analysis in this case? Try to explain any differences you observe.

   According to the worst-case analysis, there are at most mn calls to findpath. Since m and n are both doubling at each step, we expect an increase of a factor of 4 at each step. The actual rate of increase is much smaller.

   Why should this be? Well, these are random graphs and the average out degree is held constant at 16 as n increases, what's probably happening is that the minimum cut is close to either the source or the sink, and the capacity of this cut is not increasing very much as the graph gets larger. So, even though there are more "potential" paths from source to sink, most of them never get used because the min cut gets saturated so quickly.

3. (5 points) How does *fshortPath* compare to *shortPath* in this case? Discuss how this differs from the case where *n* is held constant.

   Here, fshortPath has a much a smaller advantage over shortPath. It appears that with these graphs, there few opportunities to "patch" the pEdge array. This could happen if for many edges (u,v) in the min cut, there are no "replacement edges" (w,v) where d[w]=d[v]. That is, there is only one "parent" of v in the breadth-first search tree from the source. This could happen frequently, if the min cut is defined by s and its neighbors on the left side of the cut, and all other vertices on the right side.

Part E. Evaluating performance on "hard"graphs as k1 and k2 both increase.

1. (10 points) Use the data from the second half of the *script2* output to complete the count columns of the table below. Compute ratios as before.

				fpCount		*fpSteps*		*augCount*		*augSteps*		*runtime*	
k1	*k2*	*n*	*m*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*	*count*	*ratio*
shortPath													
2	2	42	52	17		1.4K		16		272		35	
4	4	78	112	129	7.59	24K	17.14	128	8.00	3,200	11.76	451	12.89
8	8	150	256	1K	7.95	440K	18.56	1K	8.00	42K	13.12	6.8K	15.03
16	16	294	640	8.2K	7.99	9.1M	20.66	8K	8.00	598K	14.24	123K	18.17
32	32	582	1.8K	65K	8.00	212M	23.34	65K	8.00	9M	15.01	2.6M	20.74
fshortPath													
2	2	42	52	7		512		6		416		20	
4	4	78	112	27	3.86	4,730	9.24	26	4.33	3,978	9.56	148	7.40
8	8	150	256	115	4.26	48K	10.25	114	4.38	49K	12.38	1.6K	10.71
16	16	294	640	483	4.20	533K	10.99	482	4.23	686K	13.93	16K	10.10
32	32	582	1.K	2K	4.11	6.4M	12.06	1,986	4.12	10M	14.86	207K	12.91

2. (5 points) The flow graphs used in this part are structured similarly to the example graphs on slide 12 of the max flow lecture. Look at the source code to make sure you understand the role of the parameters k_1 and k_2. Give an upper bound on the number of calls to *findpath* in *shortPath* as a function of k_1 and k_2. How does the data for *shortPath* compare to the bound?

   The number of calls to shortPath should be $2k_1k_2*k_2+1$, where the +1 accounts for the last unsuccessful call to findpath This gives us 17, 129, 1025, 8193 and 65,537. These are exactly the values shown in the table. These values are growing by just under a factor of 8 at each step, which is what we expect in this case.

3. (5 points) Find an upper bound on the number of calls to *findpath* in *fshortPath* as a function of k_1 and k_2. How does the data for *fshortPath* compare to the bound?

   For these graphs, each call to augment should find k_2 augmenting paths, so the number of calls to findpath should be at most $2k_1*k_2+1$, This would give us 9, 33, 129, 513 and 2049. The actual values are a little smaller, because towards the end of the run, the edges connecting the central bipartite graph become saturated, allowing the reaugment method to patch the pEdge array for these edges as well.

4. (5 points) For large values of k_1 and k_2, how quickly would you expect the run time of *shortPath* to grow, based on the worst-case analysis. How does this compare with the data? Explain any discrepancy.

   For large values of k_1 and k_2, the runtime should grow in proportion to $k_1(k_2)^4$. In this case, that would imply that the runtime would grow by a factor of 32 for each doubling of k_1 and k_2. In this case, the growth rate is a bit slower, but the reason is that k_1 and k_2 are still small enough that the chains leading to and from the central bipartite subgraph still constitute a signifcant fraction of the total

number of edges. If we go to substantially larger values, the central subgraph dominates and we can expect to see the ratios converge to 32.

5. (5 points) Explain how you could modify the hard-case graphs so as to eliminate *fshortPath's* advantage over *shortPath*. Comment on the general utility of the method used by *fshortPath* to reduce the running time.

   A general way to eliminate the advantage of fshortPath is to replace any edge of capacity c with a path of three edges. In this path, the central edge would have capacity c, while the other 2 would have capacity c+1. With this change, only the central edges on such paths would ever become saturated, and since these edges have no potential "replacement edges" that the reaugment method could use to patch the pEdge array, the augment method would find just one augmenting path for each call to findpath.

   While this makes it clear that fshortPath is no better than shortPath in the worst case, the data for random graphs suggests that it can often be significantly faster. So, it seems like a worthwhile refinement to the basic algorithm. On the other hand, we'll later study other algorithms that are substantially faster.