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

Lab 2 Report

Your name here:_____Zheng Luo_____ *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.

```
your augPath.h file here
class augPath {
public:
     augPath(Flograph&,int&);
     ~augPath();
    // counters
    int fpCount;
    int fpSteps;
    int augCount;
    int augSteps;
    int runtime;

     …
};


your augPath.cpp file here
augPath::augPath(Flograph& fg1, int& flow_value) : fg(&fg1) {
   pEdge = new edge[fg->n()+1];
   fpCount = fpSteps = augCount = augSteps = 0; // Initialize counters to 0
}

augPath::~augPath() { delete [] pEdge; }

/** Saturate the augmenting path defined by the pEdge array. */
int augPath::augment() {
 augCount++; // increment counter
 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()) {
     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++; // increment counter
     u = fg->mate(v,e);
     fg->addFlow(u,e,f);
     v = u; e = pEdge[v];
```

```
 }
 return f;
}

your shortPath.cpp file here
shortPath::shortPath(Flograph& fg1, int& floVal) : augPath(fg1,floVal) {
   int t0 = Util::getTime(); // record start time
   floVal = 0;
   while(findPath()) {
      floVal += augment();
   }
    int t1 = Util::getTime(); // record elapsed time
    runtime = t1 - t0;
}

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

 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++; // increment counter
         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.

```
{
[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 8

stats 58 11 562 10 45 44

stats 524 30 12046 29 159 351

{
[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 364

stats 2000 2001 1153760 2000 98000 27011
```

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.

```
your faugPath.cpp file here
/** 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; // Initialize counters to
0
}

faugPath::~faugPath() { delete [] pEdge; delete [] d;}


/** Augment calls reaugment repeatedly, until reaugment returns 0.
 *  Augment then returns the total flow added during all calls to
reagument.
 */
int faugPath::augment() {
    augCount++; // increment counter
    int f = 0, totalReAugFlow = 0; // initialize flow
    bool canAugment = true;
    // call reaugment repeatedly, until reaugment returns 0.
    while (canAugment) {
        f = reaugment();
        if (f > 0) totalReAugFlow += f;
        else canAugment = false;
    }
```

```
        return totalReAugFlow;
    }

/*
 * Reaugment attempts to add flow to a single augmenting path defined by the
pEdge array.
 * If it succeeds in doing so, it attempts to fixup the pEdge array to enable
another path
 * to be found on a subsequent call. When reaugment is done, it returns the
amount of flow
 * added to the path it found or 0 if no flow was added.
 */
int faugPath::reaugment() {
    vertex u, v, w; edge e; flow f;
    // 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];
    }
    // if we can not find path with positive residual capacity
    if (f == 0) return 0;
    // add flow to saturate path
    v = fg->snk(); e = pEdge[v];
    while (v != fg->src()) {
        augSteps++; // increment counter
        u = fg->mate(v,e);
        fg->addFlow(u,e,f);
        // check if the edge is saturated
        if (fg->res(u,e) == 0) {
            // go through v's neighbour to find w as replacement
            for (e = fg->firstAt(v); e != 0; e = fg->nextAt(v,e)) {
                augSteps++; // increment counter
                w = fg->mate(v,e);
                if (d[u] == d[w] && fg->res(w,e) > 0) {
                    pEdge[v] = e; // update parent pointer
                    break;
                }
            }
        }
        // if the edge is not saturated
        v = u; e = pEdge[v];
    }
    return f;
}
    your fshortPath.cpp file here
    /** Find a shortest path with unused residual capacity.
     */
    bool fshortPath::findPath() {
        fpCount++; // increment counter
        vertex u,v; edge e;
        List queue(fg->n());

        for (u = 1; u <= fg->n(); u++) {
            pEdge[u] = 0;
            d[u] = fg->n();
        }
```

```
    queue.addLast(fg->src()); // add source
    d[fg->src()] = 0; // the initial distance
    while (!queue.empty()) {
        u = queue.first(); queue.removeFirst();
        for (e = fg->firstAt(u); e != 0; e = fg->nextAt(u,e)) {
            fpSteps++; // increment counter
            v = fg->mate(u,e);
            if (fg->res(u,e) > 0 && pEdge[v] == 0 &&
                v != fg->src()) {
                    pEdge[v] = e; // update parent edge
                    d[v] = d[u] + 1; // update distance
                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 verbose <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.

```
{
[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 9

stats 58 11 562 10 107 55

stats 524 20 7621 19 403 237

{
[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 15 1852 14 1488 223

stats 2000 183 104108 182 113880 6202
```

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		62.4K		45		320		1.3K	
200	1.6K	158	3.43	450.9K	7.23	157	3.49	861	2.69	8.5K	6.42
200	3.2K	504	3.19	2.9M	6.37	503	3.20	2.4K	2.76	48.1K	5.65
200	6.4K	1.6K	3.25	17.4M	6.07	1.6K	3.26	7.0K	2.93	311.1K	6.47
200	12.8K	4.1K	2.50	79.0M	4.53	4.1K	2.50	16.4K	2.35	1.9M	6.21
fshortPath											
200	800	34		45.5K		33		665		991	
200	1.6K	84	2.47	230.6K	5.07	83	2.52	2.6K	3.86	4.4K	4.47
200	3.2K	166	1.98	908.6K	3.94	165	1.99	10.2K	3.97	15.9K	3.59
200	6.4K	216	1.30	2.2M	2.37	215	1.30	40.6K	3.99	39.7K	2.50
200	12.8K	226	1.05	4.1M	1.88	225	1.05	151.6K	3.73	105.1K	2.65

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?

   Number of findpath = O(mn). Because level(t) is at least 1 when the algorithm starts and can never be larger than n-1, the total number of augmenting path step is O(mn). The data is much smaller than O(mn). Since n stays 200 in the running time, m doubles every time. The number of findpath should double each time, but the data is more than 2 twice of previous. I guess because the number of findpath is much smaller than worst case, we can get more than twice increase.

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?

   *Number of steps per call to findpath = O((m^2)*n), because we have most O(mn) find path step, each step has most O(m) step to find shortest path. Also the data in the table is much smaller than the O((m^2)*n).*

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.

   Base on *O((m^2)*n), it should be 4 times as before. The running time in the table grows more than 4 times, until the m becomes really big. I think cache is a fator that effects the performance, also we don't really hit the worst case, the structure of the random graph may be one factor too.*

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 values of fpCount in fshortPath is much smaller than shortPath. The growth rates of fpCount in fshortPath is smaller than shortPath, especially graph becomes more and more dense. It tells that fshortPath find more reaugment path than shortPath in augment method, we know shortPath only find one augment path per exection of augment method.

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 values of augSteps in fshortPath is much larger than shortPath. It means during each execution of the augment method, fshortPath did much more augSteps than the shortPath. values for fshortPath minus values for fshortPath roughly equals to number of augSteps is that fshortPath try to find w as a replacement. Because in fshortPath we try to do as many augSteps as we can during each execution of the augment method. This means that the overall running time of fshortPath is faster than shortPath, because fshortPath did less bfs, when the graph becomes more dense, fshortPath seem s have more advantage.

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

   The absolute value runtime of fshortPath is smaller than shortPath, when the graph becomes more dense, fshortPath seem s have more advantage. The growth rage of fshortPath is smaller than shortPath, I think because we don't really hit the worst case, the structure of the random graph may be one factor to effect growth rate, sometime the growth rate is bigger than 4 for the shortPath.

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		307.5K		253		1.1K		4.6K	
100	1.6K	430	1.69	1.1M	3.67	429	1.70	1.9K	1.79	18.5K	4.01
200	3.2K	504	1.17	2.9M	2.55	503	1.17	2.4K	1.23	48.1K	2.60
400	6.4K	575	1.14	6.9M	2.39	574	1.14	2.9K	1.20	125.9K	2.62
800	12.8K	622	1.08	15.1M	2.19	621	1.08	3.2K	1.13	383.8K	3.05
fshortPath											
50	800	55		63.1K		54		4.4K		1.0K	
100	1.6K	109	1.98	270.6K	4.29	108	2.00	7.8K	1.79	4.7K	4.44
200	3.2K	166	1.52	908.6K	3.36	165	1.53	10.2K	1.31	15.8K	3.40
400	6.4K	212	1.28	2.5M	2.71	211	1.28	12.3K	1.20	44.9K	2.84
800	12.8K	265	1.25	6.2M	2.54	264	1.25	14.3K	1.17	161.3K	3.59

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.

   Number of findpath = O(mn). Because level(t) is at least 1 when the algorithm starts and can never be larger than n-1, the total number of augmenting path step is O(mn). The data is much smaller than O(mn). Since m both and n doubles every time. The number of findpath should be 4 times as before each time, but the data is less than 2 twice of previous. I guess because the graph is sparser than previous case, the number of paths in this case is fewer than previous case. We don't have as many paths to saturate as previous one, the number of paths is growing very slowly, so growth rate of fpCount is less than previous case.

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

   The absolute value runtime of fshortPath is smaller than shortPath, because graph in this case is sparser than previous case. If n is a constant, the graph will be more and more dense. This time, n grows with m, so the graph is not as dense as before, so fshortPath seems have less advantage than previous case. Because when graph is sparser, fshortPath will have less changes to do reaugmentation, the augment method may have to do breadth first scan more frequently to find a new path instead of just doing the reaugmentation.

Part E. Evaluating performance on "hard" graphs as k_1 and k_2 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		60	
4	4	78	112	129	7.59	23.7K	17.14	128	8	3.2K	11.76	667	11.12
8	8	150	256	1.0K	7.95	440.2K	18.56	1.0K	8	42.0K	13.12	9.1K	13.57
16	16	294	640	8.2K	7.99	9.1M	20.66	8.2K	8	598.0K	14.24	179.8K	19.86
32	32	582	1.8K	65.5K	8.00	212.2M	23.34	65.5K	8	9.0M	15.01	3.5M	19.20
fshortPath													
2	2	42	52	7		512		6		416		30	
4	4	78	112	27	3.86	4.7K	9.24	26	4.33	4.0K	11.76	220	7.33
8	8	150	256	115	4.26	48.5K	10.25	114	4.38	49.2K	13.12	1.9K	8.67
16	16	294	640	483	4.20	532.5K	10.99	482	4.23	686.1K	14.24	22.4K	11.73
32	32	582	1.8K	2.0K	4.11	6.4M	12.06	2.0K	4.12	10.2M	15.01	292.2K	13.07

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?

   number of calls to findpath = $O(2 * k_1 * k_2^2)$

   The bound is very tight, every time number of calls to findpath hits the worst 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?

   number of calls to findpath = $O(2 * k_1 * k_2)$

   The bound is tight, every time number of calls to findpath roughly hits the worst case.

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.

   We know the running time is $O(k_1 * k_2^4)$, because k is doubling, the running time should be $2^5 = 32$ times than before. I think the main problem here is that k is not big enough, if k is big enough, the constant factor will effect the result. We will have total path = $2 * k_1 * k_2^2$, each path will take O(m) to find, and $m = 20 * k_1 + k_2^2 + 4 * k_2$, so $m = O(k_2^2)$, and $k_1 = k_2$, finally we get is $O(k_1 * k_2^4)$ running time, ignoring 20 k_1 if k_2 is really large.

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.

*We can add extra vertex before we enter the bipartite graph. Then fshortPath dose not have advantage over shortPath, because each time the fshortPath can not do the reaugment of the previous path, it has to find a whole new path. Below is how we add extra vertex, then both fshortPath and shortPath have the same running time O(k_1 *k_2^4).*