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

Lab 3 Report

Your name here: Zheng Luo *Due 2/19/2013*

Part A. MaxdMatch

1. (30 points) Paste a copy of your changes to *maxdMatch* below. Highlight your changes by making them bold. You may omit methods you did not change.

```
   Did not change maxdMatch.h

   Here is the change to maxdMatch.cpp

/** Extend the matching, so it covers at least one more max degree vertex.
 *  @param e is the number of an edge; there are two possible cases;
 *  if e is a matching edge, we flip the edges on the path from e
 *  to the root of the tree; otherwise e connects a free vertex to
 *  a vertex in the tree and the tree path plus e forms an augmenting path.
 */
void maxdMatch::extend(edge e) {

   int t0 = Util::getTime();
    vertex u; edge ee;

    // if e is a matching edge, we flip the edges on the path
    // from e to the root of the tree
    if (match->member(e)) {
        // we choose the vertext's pEdge = e to do flip first
        u = pEdge[graf->left(e)] == e ? graf->left(e) : graf->right(e);
        // do the flip the edge
        while (pEdge[u] != 0) {
            ee = pEdge[u]; match->remove(ee);  u = graf->mate(u,ee);
            ee = pEdge[u]; match->addLast(ee); u = graf->mate(u,ee);
        }
    } else {
        // if e is not in the matching, e connects a free vertex to a vertex
        // in the tree and the tree path plus e forms an augmenting path
        // we choose the vertext's pEdge != e to do flip first
        u = pEdge[graf->left(e)] != e ? graf->left(e) : graf->right(e);
        // do the flip the edge
        while (pEdge[u] != 0) {
            ee = pEdge[u]; match->remove(ee);  u = graf->mate(u,ee);
            ee = pEdge[u]; match->addLast(ee); u = graf->mate(u,ee);
        }
        // add e to the matching
        match->addLast(e);
    }
   stats->extend += Util::getTime() - t0;

}

/** Find a path in graf that can be used to add another max degree
 *  vertex to the matching.
```

```
 *  @return an edge e that is at the "far end" of a tree path
 *  to the root of the tree defined by pEdge[];
 *  e may be either a matching edge, or an edge that connects
 *  a tree node to an edge that is not in the tree.
 */
edge maxdMatch::findPath() {
   int t0 = Util::getTime();
   //initialization code
   vertex r,u,v,w,x; edge e,f;
   enum stype { unreached, odd, even };
   stype state[graf->n()+1];
   edge mEdge[graf->n()+1];  // mEdge[u] = matching edge incident to u
    // initialize state[], mEdge[], pEdge[]
   for (u = 1; u <= graf->n(); u++) {
        state[u] = unreached; mEdge[u] = pEdge[u] = 0;
   }
    // update mEdge[] if the elment in it is not empty
    for (e = match->first(); e != 0; e = match->next(e)) {
        u = graf->left(e); v = graf->right(e);
        mEdge[u] = mEdge[v] = e;

   }
    // find vertex r that has max degree and has not been matched
    for (u = 1; u <= graf->n(); u++) {
        // if the max degree vertex has not been matched, set it even
        if (d[u] == maxd && mEdge[u] == 0) {
            r = u;
            state[r] = even;
            break;
        }
    }
    // define a list to store even vertex
   List q(maxe);
   for (e = graf->first(); e != 0; e = graf->next(e)) {
        if (state[graf->left(e)] == even || state[graf->right(e)] == even)
            q.addLast(e);

   }
   // update performance of initialization code
   stats->fpInit += Util::getTime() - t0;
   t0 = Util::getTime();
    // main loop - search for path
    while (!q.empty()) {
        e = q.first(); q.removeFirst();
        // let v be the even vertex, w is mate(v,e)
        v = (state[graf->left(e)] == even ? graf->left(e) : graf->right(e));
        w = graf->mate(v,e);
        // 1. If w is not unreached, ignore e and proceed to the next edge.
        if (state[w] != unreached)
            continue;
        // 2. If w is unreached and unmatched, then the edge e
        // together with the tree path from v to r is an augmenting path,
        // and we terminate the path search.
        if (state[w] == unreached && mEdge[w] == 0) {
            pEdge[w] = e; // w's parent edge is e
            return e;
        }
        // 3. If w is unreached and matched, let {w,x} be the matching edge
        // incident to w. Expand the tree by making pEdge[w] = e,pEdge[x] =
```

```
            // mEdge[x], state(w)=odd and state(x)=even. If x is not a maximum
            // degree vertex, then the tree path from x to r can be flipped
            // to extend the matching, and we terminate the path search.
            if (state[w] == unreached && mEdge[w] != 0) {
                x = graf->mate(w,mEdge[w]);
                state[w] = odd; pEdge[w] = e;
                state[x] = even; pEdge[x] = mEdge[x];
                if (d[x] != maxd)
                    return mEdge[x];
                // update even edge in the list
                for (f = graf->firstAt(x); f != 0; f = graf->nextAt(x,f)) {
                    if ((f != mEdge[x]) && !q.member(f))
                        q.addLast(f);
            }
        }
    }
    // update performance of main loop - search for path code
    stats->fpLoop += Util::getTime() - t0;
    // if no path is found, return 0
     return 0;
}
```

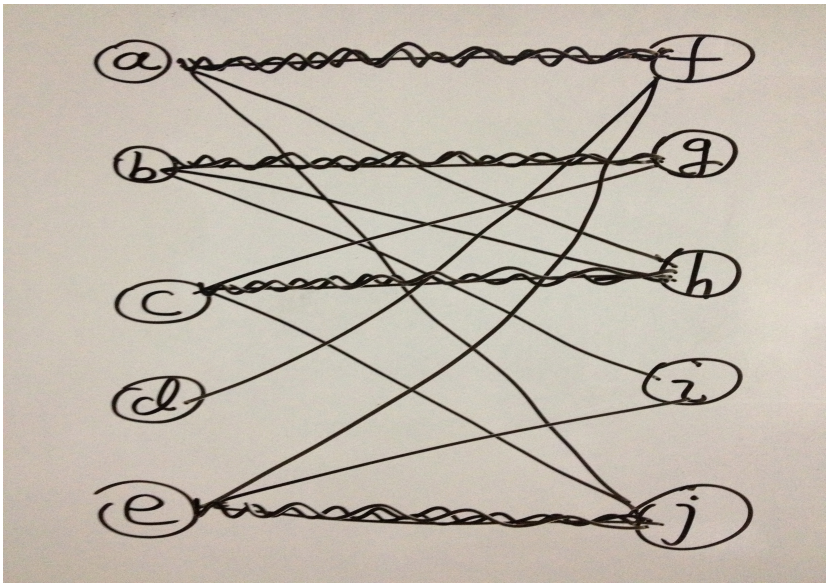
2. (10 points) Compile the provided code in your *lab3* directory using the *makefile*. Verify your code for *maxdMatch* using the command *checkMaxdMatch* by typing

```
checkMaxdMatch <bg5
checkMaxdMatch <bg10
checkMaxdMatch <bg50
```

Paste a copy of your output below.

```
1 6 0 3 12
(f,a) (g,b) (h,c) (j,e)
2 9 0 1 21
(l,e) (k,f) (m,g) (o,b) (r,d) (s,i)
3 34 1 1 50
(53,30) (52,33) (56,37) (73,41) (69,47) (57,44) (72,50) (74,13) (79,35)
(87,27) (96,25)
```

3. (5 points) Draw the graph in *bg5* and highlight the edges in the computed matching by making them heavier weight.



Part B. EdgeColor

1. (20 points) Paste a copy of your changes to *edgeColor* below. Highlight your changes by making them bold. You may omit methods you did not change.

```
Here is edgeColor.cpp
/** Find a minimum edge coloring in a bipartite graph.
 *  The algorithm used here finds a series of matchings, where each
 *  matching includes an edge incident to every max degree vertex.
 *  @param graf1 is a reference to the graph
 *  @param colorSets is a reference to a set of circular lists; on return,
 *  each list in the set defines a set of edges of the same color
 *  @param stats is a reference to a Stats object that on return,
 *  contains the total running time associated with various parts of
 *  the max matching algorithm, which is used as a subroutine.
 */

void edgeColor(Graph& graf1, ClistSet& colorSets, maxdMatch::Stats& stats)
{
    edge e, ee;
    Graph graf; graf.copyFrom(graf1); // copied graph can be used to
remove edges
    Dlist match(graf.m()); // store matched edge
    stats.clear();
    maxdMatch::Stats stats1;
    while (true) {
        // remove matched edge from graph
        for (e = match.first(); e != 0; e = match.next(e))
            graf.remove(e);
        // clear the match for the next maxMatch call
        match.clear();
        // find another matching
        maxdMatch(graf, match, stats1);
        stats.add(stats1);
        // if there is no edge to match, so every edge is matched, break
        if (match.length() == 0)
            break;
        // when match is found, we add matched edges to the colorSets
        e = match.first();
        for (ee = match.next(e); ee != 0; ee = match.next(ee)) {
            colorSets.join(e,ee); // make e and ee belong to the same set
            e = ee;
        }
    }
}
```

2. (15 points) Verify your code for *edgeColor* using the command *checkEdgeColor* by typing

```
checkEdgeColor <bg5a
checkEdgeColor <bg10a
checkEdgeColor <bg50a
```

Paste a copy of your output below.

```
3 13 1 5 36
{(f,a), (g,b), (h,c), (j,e)}
{(f,c), (h,e)}
{(f,d), (i,e), (j,a), (h,b), (g,c)}
{(f,e), (j,c), (h,a), (i,b)}


4 27 1 7 64
{(k,i), (s,a), (t,j), (o,b), (q,c), (m,d), (r,e), (l,f), (p,g)}
{(k,f), (o,g), (s,i), (r,h), (m,b), (l,c), (p,e)}
{(l,g), (o,i), (p,b), (r,c), (t,d), (s,e), (m,f)}
{(l,b), (m,c), (o,e), (r,i), (s,f)}
{(r,d)}
{(r,f)}

21 338 2 18 468
{(51,44), (69,47), (72,50), (74,13), (79,27), (80,18), (81,33), (87,10),
(96,25), (53,30), (56,37), (77,41)}
{(51,42), (65,43), (92,44), (77,45), (57,46), (80,47), (58,48), (62,49),
(97,50), (87,1), (52,4), (69,5), (90,6), (91,7), (67,9), (78,10), (64,11),
(81,12), (75,13), (89,14), (68,15), (71,16), (86,18), (96,19), (83,20),
(84,21), (70,22), (79,23), (82,24), (94,25), (55,27), (66,28), (72,29),
(74,30), (76,32), (61,33), (95,34), (88,35), (93,37), (59,38), (85,39),
(63,40), (100,41)}
{(51,30), (59,33), (73,41), (57,44), (69,27), (79,35), (87,23), (52,18)}
{(52,47), (70,4), (54,48), (82,50), (58,26), (75,20), (78,13), (81,2),
(57,38), (71,46), (90,16), (93,27), (94,36), (96,9), (100,3), (76,6),
(84,7), (74,10), (61,12), (77,14), (79,19), (69,21), (87,5), (56,22),
(66,23), (72,25), (89,29), (88,30), (68,32), (80,33), (65,35), (92,37),
(85,41), (86,44), (97,18), (91,45)}
{(52,33), (58,37), (64,44), (90,45), (82,47), (53,48), (94,50), (75,49),
(79,38), (57,3), (87,21), (91,35), (93,18), (96,41), (69,14), (100,5),
(55,6), (74,7), (66,9), (71,13), (72,19), (68,20), (76,23), (88,25),
(67,27), (80,29), (89,30), (81,32)}
{(52,6), (58,14), (57,18), (59,23), (65,25), (56,29), (69,30), (78,35),
(66,37), (63,38), (80,41), (62,44), (70,45), (74,47), (72,48), (79,15),
(81,42), (87,27), (93,12), (96,33)}
{(60,41), (69,12), (87,34)}
```

Also, type the following commands

```
evalEdgeColor 10 20
evalEdgeColor 100 200
evalEdgeColor 1000 2000
```

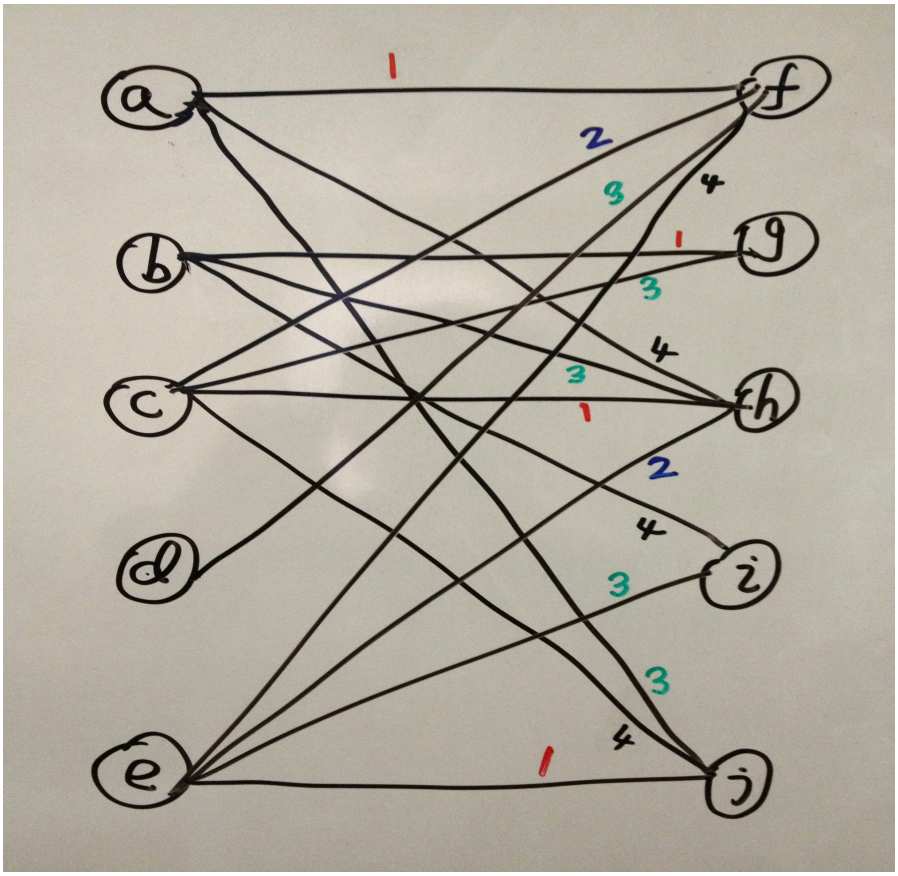
and paste the output below.

```
10 20 3.5us 15.2us 0.5us 2.5us 34.8us 45.3us

100 200 23.8us 460us 0.5us 21.1us 590us 624us

1000 2000 164us 32.3ms 1us 162us 33.1ms 33.4ms
```

3. (5 points) Draw the graph in *bg5a* and show the coloring computed by *edgeColor*, by labeling each with an integer to indicate its "color set" (so edges in the first color set are labeled 1, the edges in the next are labeled 2, etc).



Part C. Evaluating edgeColor.

1. (10 points) Run the provided *script1* and use the resulting data to complete the table below. Note that the performance reported by *evalEdgeColor*. Show the units. For each performance counter, compute the ratios of the values from one row to the next.

		maxdInit		*fpInit*		*fpLoop*		*extend*		*total*	
n	*m*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*
fixed n, increasing m											
5K	5.5K	414.3us		348.6ms		0.1us		415.1us		350.9ms	
5K	7.5K	608.2us	1.47	547.7ms	1.57	0.2us	2.00	583.8us	1.41	551.1ms	1.57
5K	15K	1.5ms	2.54	1.6sec	2.90	0.8us	4.00	1.3ms	2.27	1.6sec	2.90
5K	30K	4.3ms	2.77	4.9sec	3.08	1.1us	1.38	3.1ms	2.32	4.9sec	3.08
5K	60K	12.6ms	2.95	16.0sec	3.27	1.8us	1.64	8.1ms	2.63	16.1sec	3.28
fixed average degree, increasing n											
2K	6K	564.8us		237.9ms		1.2us		481.4us		240.6ms	
4K	12K	1.2ms	2.18	1.0sec	4.39	0.5us	0.42	1.1ms	2.34	1.1sec	4.37
8K	24K	2.5ms	2.03	4.2sec	4.05	0.9us	1.80	2.3ms	2.03	4.2sec	4.04
16K	48K	5.5ms	2.19	17.8sec	4.20	0.7us	0.78	7.8ms	3.40	17.9sec	4.21
32K	96K	12.7ms	2.32	84.6sec	4.75	1.4us	2.00	26.4ms	3.40	84.9sec	4.75

2. (10 points) Give an expression (in terms of n, m and the maximum vertex degree Δ), for the worst-case number of calls to *findpath* over all calls to *maxdMatch*. Now, give an expression for the worst-case asymptotic running time of *edgeColor*? Let T_1 be the runtime of *edgeColor* on a graph with n vertices and m edges, and T_2 be the runtime of *edgeColor* on a graph with n vertices and $2m$ edges. Based on the worst-case analysis, what would you expect the ratio T_2/T_1 to be? How does this compare with the data in the top portion of the table?

   *We know that each findpath operation matches a max degree vertex one at a time, in worst case; all vertices have to be matched. We assume each vertex have maximum degree, which equals 2*m/n, so worst-case number of calls to findpath in one maxdMatch is n/2 = O(n). Also, we know that total number of maxdMatch equals* Δ. *Therefore, worst-case number of calls to findpath over all calls to maxdMatch = O(n**Δ*). Each findpath operation takes O(m) time to find a path. Therefore, based on the worst-case analysis, worst-case asymptotic running time of edgeColor is O(m*n**Δ*).*

   I expect the ratio T_2/T_1 *to be 2, because the n and* Δ *do not change, only m doubles. Base on T = O(m*n**Δ*), it should double. The table shows that the running time increased more than rate of m increased. When m is 1.4 times as before, new T is 1.8 times as before and m is twice as before, new T is more than twice as before, sometimes about 3 times. I think this is because in the execution of the algorithm, we very rally hit the worst case. In the beginning, we almost never hit the worst case, in the later state; we may probably hit the worst case, but can be very rare. Also, the structure of the graph may be one factor too, so we rarely hit the worst case.*

3. (5 points) Let T_1 be the runtime of *edgeColor* on a graph with n vertices and kn edges, and T_2 be the runtime of *edgeColor* on a graph with $2n$ vertices and $2kn$ edges. Based on the worst-case analysis, what would you expect the ratio T_2/T_1 to be? How does this compare with the data in the bottom portion of the table?

*Worst-case asymptotic running time of edgeColor is T = O(m*n*Δ). In this case, n doubles, m doubles, and Δ doubles too. Therefore, I expect the ratio T_2/T_1 to be 8. But the running time is less than 8, probably around 4 in the table. I think this is because in the execution of the algorithm, we very rally hit the worst case. In the beginning, we almost never hit the worst case, in the later state; we may probably hit the worst case, but can be very rare. Also, the structure of the graph may be one factor too, so we rarely hit the worst case.*

4. (10 points) Compare the relative values of *fpInit* and *fpLoop*. Normally we expect initialization to be a small fraction of an algorithm's runtime, but that is not the case here. Explain why, as completely as you can.

   The values of *fpInit* is much larger than the *fpLoop. And fpInit is the dominant factor that effects the running time. In fact fpLoop is really small in contribution to the algorithm's runtime.*

   Every time we call findPath(), we will redo all the initialization again, which seems very redundant. Even we say that the running time of findPath() is O(m), the hidden constant factor can be really large. Let's look at the code here:

   a) *We are redoing initialize state[], mEdge[], pEdge[] again and again each time. O(n) time each call*

   ```
   for (u = 1; u <= graf->n(); u++) {
            state[u] = unreached; mEdge[u] = pEdge[u] = 0;
   }
   ```

   b) *We are redoing update mEdge[] again and again each time. O(m) time each call.*

   ```
   for (e = match->first(); e != 0; e = match->next(e)) {
                    u = graf->left(e); v = graf->right(e);
                    mEdge[u] = mEdge[v] = e;
   }
   ```

   c) *We are redoing find vertex r that has max degree and has not been matched again and again each time. O(n) time each call.*

   ```
   for (u = 1; u <= graf->n(); u++) {
       // if the max degree vertex has not been matched, set it even
       if (d[u] == maxd && mEdge[u] == 0) {
          r = u;
          state[r] = even;
          break;
       }
     }
   ```

   d) *We are redoing find edge which is next to vertex r that has max degree and has not been matched again and again each time. O(m) time each call.*

   ```
   for (e = graf->first(); e != 0; e = graf->next(e)) {
   ```

```
            if (state[graf->left(e)] == even || state[graf->right(e)] == even)
        q.addLast(e);
    }
```

5. (10 points) Discuss at least three changes you can make to *maxdMatch* that might significantly improve the overall performance. In each case, explain the change you have in mind and why you think it will improve the overall performance.

   a) *We maintain a maxVerts list which keep tracks the unmatched max degree vertices. Once a max degree vertex is matched, we remove the list. This save the running time of a constant factor times n. The procedure below in findPath() will be replace by the initialized maxVerts in init().*

```
for (u = 1; u <= graf->n(); u++) {
    // if the max degree vertex has not been matched, set it even
    if (d[u] == maxd && mEdge[u] == 0) {
        r = u;
        state[r] = even;
        break;
    }
}
```

   b) *We maintain a mEdge[] list which keep tracks matched edge number of each the vertex. Most importantly, when we do the extend step, we will keep mEdge[] up to date in the extend step. Not in the find step. This save the running time of a constant factor times m. The procedure below in findPath() will be replace by the some operation in extend().*

```
for (e = match->first(); e != 0; e = match->next(e)) {
        u = graf->left(e); v = graf->right(e);
        mEdge[u] = mEdge[v] = e;
}
```

   c) *We maintain visited[] which keeps track of the most recent phase in which each vertex has been visited, which means we only set initial state[] even in the init() and change state[] even or odd in the process of expending the tree, we will not do initial state[] every time in findPath(). This save the running time of a constant factor times m. The procedure below in findPath() will be replace by the some operation in init() and expend the tree.*

```
for (u = 1; u <= graf->n(); u++) {
        state[u] = unreached; mEdge[u] = pEdge[u] = 0;
}

for (e = graf->first(); e != 0; e = graf->next(e)) {
        if (state[graf->left(e)] == even || state[graf->right(e)] == even)
                q.addLast(e);
}
```

Part D. FmaxdMatch and fedgeColor

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

Here is the change to fmaxdMatch.h

```
class fmaxdMatch {
public:
    fmaxdMatch(Graph&,Dlist&,maxdMatch::Stats&);
protected:
    Graph*   graf;       // graph we're finding matching for
    Dlist*   match;      // matching we're building
    Dlist*   q;          // stores edges incident to even vertices
    Dlist*  maxVerts;   // keeps track the umatched max degree vertices
    edge*    pEdge;      // pEdge[u] is edge to parent of u in forest
    edge*   mEdge;      // stores vertex incident to matched edges
    int*    visit;      // keeps track most recent phase each visited
vertex
    int*     d;          // d[u] is degree of u
    int     maxd;       // maximum degree
    int     maxe;       // largest edge number
    int     sNum;       // index of current search
    enum stype {odd, even};
    stype* state;       // stores the state of the vertex

    maxdMatch::Stats *stats;

    void extend(edge);
    edge findPath();
    void init(Graph&,Dlist&,maxdMatch::Stats&);
    void cleanup();
};
```

Here is the change to fmaxdMatch.cpp

```
/** Find a matching in the bipartite graph graf that includes an
 *  edge at every vertex of maximum degree.
 *  graf1 is a reference to the graph
 *  match1 is a reference to a list in which the matching is returned
 */
fmaxdMatch::fmaxdMatch(Graph& graf1, Dlist& match1, maxdMatch::Stats&
stats1) {
   int t0 = Util::getTime();
    // do the initalize
    init(graf1, match1, stats1);
    // repeated max all max degree vertex
     while(true) {
     edge e = findPath();
     if (e == 0) break;
     extend(e);
 }
  // do clean up
 cleanup();
  // update performance statistic
 stats->total = Util::getTime() - t0;
}

/** Initizalize all data structures used by the algorithm.
 *  Includes allocation and initialization of dynamic data structures.
```

```
 *  In addition to the data structures provided by the base class,
 *  we add an mEdge array, a list maxdVerts containing unmatched
 *  vertices of maximum degree, the queue used by findpath
 *  and the array visit[] which keeps track of the most recent
 *  phase in which each vertex has been visited.
 */

void fmaxdMatch::init(Graph& graf1, Dlist& match1, maxdMatch::Stats&
stats1) {
 int t0 = Util::getTime();
 graf = &graf1; match = &match1; stats = &stats1;
 // Initialization of dynamic data structures
 pEdge = new edge[graf->n()+1]; mEdge = new edge[graf->n()+1];
    visit = new int[graf->n()+1]; d = new int[graf->n()+1];
    maxVerts = new Dlist(graf->n()); state = new stype[graf->n()+1];
    // initialize variables
    stats->clear(); sNum = maxd = maxe = 0;
 for (vertex u = 0; u <= graf->n(); u++) {
        d[u] = visit[u] = mEdge[u] = pEdge[u] = 0;
    }
 // compute vertex degrees and max degree
 // find largest edge number
 for (edge e = graf->first(); e != 0; e = graf->next(e)) {
     vertex u = graf->left(e); vertex v = graf->right(e);
     d[u]++; d[v]++;
     maxd = max(maxd,d[u]); maxd = max(maxd,d[v]);
     maxe = max(e,maxe);
 }
    // initialize queue which stores even vertices
    q = new Dlist(maxe);
    // add max degree vertex to the maxVerts
    for (vertex u = 1; u <= graf->n(); u++) {
        if (d[u] == maxd) maxVerts->addLast(u);
    }
 stats->maxdInit = Util::getTime() - t0;
}

void fmaxdMatch::cleanup() {
    delete [] pEdge; delete [] mEdge; delete [] visit;
    delete [] d; delete [] state; delete q; delete maxVerts;
}

/** Extend the matching, so it covers at least one more max degree vertex.
 *  @param e is the number of an edge; there are two possible cases;
 *  if e is a matching edge, we flip the edges on the path from e
 *  to the root of the tree; otherwise e connects a free vertex to
 *  a vertex in the tree and the tree path plus e forms an augmenting
path.
 */
void fmaxdMatch::extend(edge e) {
    int t0 = Util::getTime();
    vertex u,v; edge ee;
    // if e is a matching edge, we flip the edges on the path
    // from e to the root of the tree
    if (match->member(e)) {
        // we choose the vertext's pEdge = e to do flip first
        u = pEdge[graf->left(e)] == e ? graf->left(e) : graf->right(e);
        // do the flip the edge
        mEdge[u] = 0;
```

```
        // do the flip the edge
        while (pEdge[u] != 0) {
            ee = pEdge[u]; match->remove(ee);  v = graf->mate(u,ee);
            ee = pEdge[v]; match->addLast(ee); u = graf->mate(v,ee);
            mEdge[u] = mEdge[v] = ee;
        }
    } else {
        // if e is not in the matching, e connects a free vertex to a
vertex in
        // the tree and the tree path plus e forms an augmenting path
        // we choose the vertext's pEdge != e to do flip first
        u = pEdge[graf->left(e)] != e ? graf->left(e) : graf->right(e);
        // do the flip the edge
        while (pEdge[u] != 0) {
            ee = pEdge[u]; match->remove(ee);  v = graf->mate(u,ee);
            ee = pEdge[v]; match->addLast(ee); u = graf->mate(v,ee);
            mEdge[u] = mEdge[v] = ee;
        }
        // add e to the matching
        match->addLast(e);
        // update mEdge
        mEdge[graf->left(e)] = mEdge[graf->right(e)] = e;
    }
 stats->extend += Util::getTime() - t0;
}

/** Find a path in graf that can be used to add another max degree
 *  vertex to the matching.
 */
edge fmaxdMatch::findPath() {
     int t0 = Util::getTime();
    // do some initialization
    vertex r,v,w,x; edge e,f;
    sNum++; q->clear();
    // make sure r is a valid vertex
    if (maxVerts->length() == 0) return 0;
    // get the root vertex
    r = maxVerts->first(); maxVerts->removeFirst();
    // update the state of root
    state[r] = even; visit[r] = sNum;
    // add edges incident to even vertices to the queue
    for (e = graf->firstAt(r); e != 0; e = graf->nextAt(r,e)) {
        q->addLast(e);
    }
    // main loop - search for path
    while (!q->empty()) {
     e = q->first(); q->removeFirst();
     // let v be the even vertex, w is mate(v,e)
     v = (state[graf->left(e)] == even ? graf->left(e) : graf->right(e));
     w = graf->mate(v,e);
        // 1. If w is visited in this phase, ignore e and proceed to the
next edge.
        if (visit[w] == sNum)
            continue;
        // 2. If w is not visited in this phase and unmatched, then the
edge e

        // together with the tree path from v to r is an augmenting path,
```

```
        // and we terminate the path search. However if w has maximum
degree,
        // we have to remove it from the maxVerts.
        if (visit[w] < sNum && mEdge[w] == 0) {
            pEdge[w] = e; // w's parent edge is e
            visit[w] = sNum; // update visit of w
            //if w has maximum degree, remove it from the maxVerts.
            if (d[w] == maxd) maxVerts->remove(w);
            return e;
        }
        // 3. If w visited in this phase and matched, let {w,x} be the
matching edge incident
        // to w. Expand the tree by making pEdge[w] = e,pEdge[x] =
mEdge[x], state(w)=odd and
        // state(x)=even. If x is not a maximum degree vertex, then the
tree path from x to r
        // can be flipped to extend the matching, and we terminate the
path search.
     if (visit[w] < sNum && mEdge[w] != 0) {
         x = graf->mate(w,mEdge[w]);
         state[w] = odd; pEdge[w] = e;
         state[x] = even; pEdge[x] = mEdge[x];
         visit[w] = visit[x] = sNum;
         if (d[x] != maxd) return mEdge[x];
         // update even edge in the list
         for (f = graf->firstAt(x); f != 0; f = graf->nextAt(x,f)) {
             if ((f != mEdge[x]) && !q->member(f)) q->addLast(f);
         }
     }
 }
    // update performance of main loop - search for path code
    stats->fpLoop += Util::getTime() - t0;
    // if no path is found, return 0
    return 0;
}
```

2. (15 points) Verify your code using the command *checkFedgeColor* by typing

```
checkFedgeColor <bg5a
checkFedgeColor <bg10a
checkFedgeColor <bg50a
```

Paste a copy of your output below.

```
11 0 0 4 31
{(f,a), (g,b), (h,c), (j,e)}
{(f,c), (h,e)}
{(f,d), (i,e), (j,a), (h,b), (g,c)}
{(f,e), (j,c), (h,a), (i,b)}
17 0 0 2 53
{(k,i), (s,a), (t,j), (o,b), (q,c), (m,d), (r,e), (l,f), (p,g)}
{(k,f), (o,g), (s,i), (r,h), (m,b), (l,c), (p,e)}
{(l,g), (o,i), (p,b), (r,c), (t,d), (s,e), (m,f)}
{(l,b), (m,c), (o,e), (r,i), (s,f)}
{(r,d)}
{(r,f)}
50 0 0 24 167
{(51,44), (69,47), (72,50), (74,13), (79,27), (80,41), (81,33), (87,10),
(96,25), (52,18), (53,30), (56,37)}
```

```
{(51,42), (65,43), (92,44), (77,45), (57,46), (52,47), (58,48), (62,49),
(97,50), (70,4), (87,5), (90,6), (91,7), (67,9), (78,10), (64,11),
(81,12), (75,13), (69,14), (68,15), (71,16), (86,18), (96,19), (83,20),
(84,21), (56,22), (79,23), (82,24), (72,25), (55,27), (66,28), (89,29),
(74,30), (76,32), (80,33), (95,34), (88,35), (94,36), (93,37), (59,38),
(85,39), (63,40), (100,41)}
{(51,30), (61,33), (73,41), (62,44), (69,27), (79,35), (87,23), (57,18)}
{(52,4), (76,6), (84,7), (74,10), (61,12), (79,19), (69,21), (89,14),
(66,23), (72,29), (94,25), (88,30), (68,32), (59,33), (65,35), (92,37),
(63,38), (85,41), (86,44), (97,18), (80,47), (54,48), (82,50), (58,26),
(70,22), (75,20), (78,13), (81,2), (71,46), (90,16), (57,3), (91,45),
(93,27), (96,9), (100,5), (87,1)}
{(52,33), (58,37), (57,44), (90,45), (82,47), (53,48), (94,50), (75,49),
(79,38), (87,21), (91,35), (93,18), (96,41), (100,3), (69,5), (55,6),
(74,7), (66,9), (71,13), (77,14), (72,19), (68,20), (76,23), (88,25),
(67,27), (80,29), (89,30), (81,32)}
{(52,6), (58,14), (80,18), (59,23), (65,25), (56,29), (69,30), (96,33),
(78,35), (66,37), (57,38), (77,41), (64,44), (70,45), (74,47), (72,48),
(79,15), (81,42), (87,27), (93,12)}
{(60,41), (69,12), (87,34)}
```

Also, type the following commands

```
evalFedgeColor 10 20
evalFedgeColor 100 200
evalFedgeColor 1000 2000
```

and paste the output below.

```
10 20 12.8us 0us 0.4us 3.2us 32.3us 41.4us
100 200 96.5us 0us 0.8us 25.3us 232us 275us
1000 2000 619us 0us 0.4us 157us 1.39ms 1.65ms
```

3. (10 points) Run the provided *script2* and use the resulting data the table below. Compute ratios as before.

		maxdInit		*fpInit*		*fpLoop*		*extend*		*total*	
n	*m*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*	*time*	*ratio*
fixed n, increasing m											
5K	5.5K	1.7ms		0us		0.4us		396us		3.6ms	
5K	7.5K	2.3ms	1.24	0us	N/A	1.5us	0.91	611.4us	1.37	5.3ms	1.34
5K	15K	4.4ms	1.92	0us	N/A	1us	1.70	1.1ms	1.96	10.6ms	2.04
5K	30K	9.6ms	2.18	0us	N/A	2.9us	2.18	2.3ms	2.02	25.6ms	2.39
5K	60K	24.6ms	2.49	0us	N/A	7.3us	2.08	5.8ms	2.40	82.5ms	3.08
fixed average degree, increasing n											
2K	6K	1.7ms		0us		1.4us		447.5us		4.1ms	
4K	12K	3.5ms	2.08	0us	N/A	1.4us	1.00	914.1us	2.04	8.6ms	2.09
8K	24K	7.3ms	2.07	0us	N/A	2.9us	2.07	1.9ms	2.07	18.5ms	2.15
16K	48K	15.2ms	2.08	0us	N/A	2.7us	0.93	4.6ms	2.43	43.4ms	2.35
32K	96K	33.9ms	2.23	0us	N/A	3.8us	1.41	11.5ms	2.50	117.0ms	2.71

3. (10 points) Compare the running time of *fedgeColor* to that of *edgeColor*. How big an improvement did you get? Where is most of the time being spent now? Do you think there is still room to improve this further?

   The running time of fedgeColor is several hundreds times faster than the edgeColor. For example: 16.1sec/85.2ms = 188.3, 81.9/117ms = 725.6 and so on. The most of the time being spent now is in maxdInit. I think if we want to improve the asymptotical running time, we have to change the algorithm, that is one feasible way. Because there might be some asymptotically faster algorithms, I am not sure about this. If we cannot find asymptotically faster algorithms, we can reduce the some constant factor in the algorithm we use now. I there is a possible improvement for the algorithm, that is cut down the maxdInit times. I think we don't have to do the following every time in the fmaxdMatch.

   ```
   // compute vertex degrees and max degree
   // find largest edge number
   for (edge e = graf->first(); e != 0; e = graf->next(e)) {
           vertex u = graf->left(e); vertex v = graf->right(e);
           d[u]++; d[v]++;
           maxd = max(maxd,d[u]); maxd = max(maxd,d[v]);
           maxe = max(e,maxe);
   }
   ```

   We can store the d[u] information in the edgeColor, that means, once d[u] is initialized for the first time, when remove graph edge happens, we decrease the d[u] at the same time. Because we know what edge e we will remove, we can do similal things like

   ```
   for (e = match.first(); e != 0; e = match.next(e)) {
          graf.remove(e);
           vertex u = graf->left(e); vertex v = graf->right(e);
           d[u]--; d[v]--;
   }
   ```

   I think by doing that it can save us another constant factor times m running time.

4. (10 points) How do the growth rates for the running time compare to the original version. How do you account for the differences?

   The growth rates for the optimized version are smaller than the original version. We say the worst-case asymptotic running time of edgeColor is $T = O(m*n*\Delta)$. For the maxdMatch we did O(m) operation per findPath() to initialize mEdge[]. In fmaxdMatch, we only do one initialize mEdge[] per fmaxdMatch call. Both version rarely hit worst cases, so the growth rate of maxdMatch should be larger than fmaxdMatch for the O(m) and O(n) operations per findPath(). That accounts for the growth rate of edgeColor larger than fedgeColor.