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

Lab 3 Solution

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.

```
/** Extend 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;

   if (match->member(e)) {
       u = graf->left(e);
       if (pEdge[u] != e) u = graf->right(e);
       while (pEdge[u] != 0) {
           e = pEdge[u]; match->remove(e);  u = graf->mate(u,e);
           e = pEdge[u]; match->addLast(e); u = graf->mate(u,e);
       }
       stats->extend += Util::getTime() - t0;
       return;
   }
   match->addLast(e);
   u = graf->left(e);
   if (pEdge[u] == 0) u = graf->right(e);
   while (pEdge[u] != 0) {
       e = pEdge[u]; match->remove(e);  u = graf->mate(u,e);
       e = pEdge[u]; match->addLast(e); u = graf->mate(u,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();
   enum stype { unreached, odd, even };
   stype state[graf->n()+1];
   edge mEdge[graf->n()+1];  // mEdge[u] = matching edge incident to u
```

```
    for (vertex u = 1; u <= graf->n(); u++) {
        state[u] = unreached; mEdge[u] = pEdge[u] = 0;
    }
    for (edge e = match->first(); e != 0; e = match->next(e)) {
        vertex u = graf->left(e); vertex v = graf->right(e);
        state[u] = state[v] = unreached;
        mEdge[u] = mEdge[v] = e;
    }

    // find a max degree vertex that's unmatched
    vertex root = 0;
    for (vertex u = 1; u <= graf->n(); u++) {
        if (d[u] == maxd && mEdge[u] == 0) {
            root = u; break;
        }
    }
    if (root == 0) return 0;
    state[root] = even;

    List q(maxe);
    for (edge e = graf->firstAt(root); e != 0; e = graf->nextAt(root,e)) {
        q.addLast(e);
    }

    stats->fpInit += Util::getTime() - t0;

    t0 = Util::getTime();

    edge e;
    while (!q.empty()) {
        e = q.first(); q.removeFirst();
        vertex v = (state[graf->left(e)] == even ?
                graf->left(e) : graf->right(e));
        vertex w = graf->mate(v,e);
        if (state[w] != unreached) continue;
        if (mEdge[w] == 0) break;
        vertex x = graf->mate(w,mEdge[w]);
        state[w] = odd; pEdge[w] = e;
        state[x] = even; pEdge[x] = mEdge[x];
        if (d[x] < maxd) { e = pEdge[x]; break; }
        for (edge ee = graf->firstAt(x); ee != 0;
                  ee = graf->nextAt(x,ee)) {
            if ((ee != mEdge[x]) && !q.member(ee)) {
                q.addLast(ee);
            }
        }
    }
    stats->fpLoop += Util::getTime() - t0;
    return e;
}
```

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

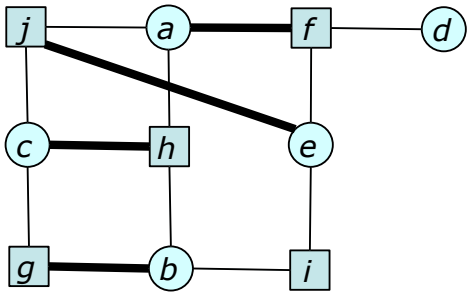
```
checkMaxdMatch <bg5
0 2 1 1 7
(f,a) (g,b) (h,c) (j,e)

checkMaxdMatch <bg10
1 2 0 1 7
```

```
(l,e) (k,f) (m,g) (o,b) (r,c) (s,a)

checkMaxdMatch <bg50
2 9 3 1 19
(65,25) (53,30) (52,33) (78,35) (56,37) (73,41) (51,44) (69,47) (57,3)
(72,19) (74,7) (79,15) (87,1) (96,9)
```

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.

```
#include "maxdMatch.h"

using namespace grafalgo;

/** 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)
{
    colorSets.clear();
    Graph graf(graf1.n(), graf1.m());
    graf.copyFrom(graf1);
    Dlist match(graf.m());

    stats.clear();
    maxdMatch::Stats stats1;
    while (true) {
        if (graf.m() == 0) return;
        maxdMatch(graf,match,stats1);
        stats.add(stats1);
        edge e = match.first();
        match.removeFirst();
        graf.remove(e);
        while (not match.empty()) {
            edge ee = match.first();
            colorSets.join(e,ee);
            match.removeFirst();
            graf.remove(ee);
        }
    }
}
```

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

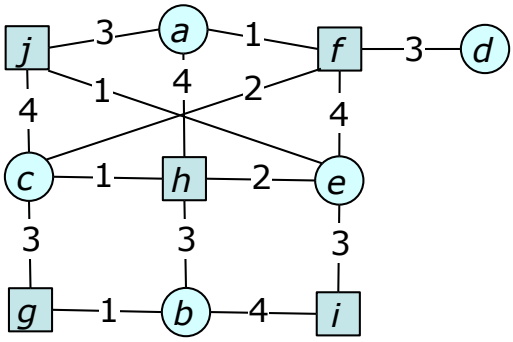
```
% checkEdgeColor <bg5a
1 6 2 5 17
{(f,a), (j,e), (h,c), (g,b)}
{(f,c), (h,e)}
{(f,d), (g,c), (h,b), (j,a), (i,e)}
{(f,e), (i,b), (h,a), (j,c)}
% checkEdgeColor <bg10a
2 20 5 0 33
{(k,i), (p,g), (l,f), (r,e), (m,d), (q,c), (o,b), (t,j), (s,a)}
{(k,f), (p,e), (l,c), (m,b), (r,h), (s,i), (o,g)}
{(l,g), (m,f), (s,e), (t,d), (r,c), (p,b), (o,i)}
{(l,b), (s,f), (r,i), (o,e), (m,c)}
{(r,d)}
{(r,f)}
```

```
% checkEdgeColor <bg50a
13 116 17 13 224
{(51,44), (77,41), (56,37), (53,30), (96,25), (87,10), (81,33), (80,18),
(79,27), (74,13), (72,50), (69,47)}
{(51,42), (100,41), (63,40), (85,39), (59,38), (93,37), (88,35), (95,34),
(61,33), (76,32), (74,30), (72,29), (66,28), (55,27), (94,25), (82,24),
(79,23), (70,22), (84,21), (83,20), (96,19), (86,18), (71,16), (68,15),
(89,14), (75,13), (81,12), (64,11), (78,10), (67,9), (91,7), (90,6),
(69,5), (52,4), (87,1), (97,50), (62,49), (58,48), (80,47), (57,46),
(77,45), (92,44), (65,43)}
{(51,30), (52,18), (87,23), (79,35), (69,27), (57,44), (73,41), (59,33)}
{(52,47), (70,4), (91,45), (86,44), (97,18), (85,41), (92,37), (65,35),
(80,33), (68,32), (88,30), (89,29), (72,25), (66,23), (56,22), (69,21),
(87,5), (79,19), (77,14), (61,12), (74,10), (84,7), (76,6), (100,3),
(96,9), (94,36), (93,27), (90,16), (71,46), (57,38), (81,2), (78,13),
(75,20), (58,26), (82,50), (54,48)}
{(52,33), (81,32), (89,30), (80,29), (67,27), (88,25), (76,23), (68,20),
(72,19), (71,13), (66,9), (74,7), (55,6), (100,5), (69,14), (96,41),
(93,18), (91,35), (87,21), (79,38), (57,3), (75,49), (94,50), (53,48),
(82,47), (90,45), (64,44), (58,37)}
{(52,6), (96,33), (93,12), (87,27), (81,42), (79,15), (72,48), (74,47),
(70,45), (62,44), (80,41), (63,38), (66,37), (78,35), (69,30), (56,29),
(65,25), (59,23), (57,18), (58,14)}
{(60,41), (87,34), (69,12)}
```

Also, type the following commands

```
% evalEdgeColor 10 20
10 20 1.5us 4.5us 1.2us 2.3us 16.4us 20.5us
% evalEdgeColor 100 200
100 200 11.8us 180us 15.8us 16.3us 282us 304us
% evalEdgeColor 1000 2000
1000 2000 112us 12.2ms 116us 107us 12.8ms 13ms
```

3. (5 points) Draw the graph in *bg5a* and show the coloring computed by *edgeColor*, by labeling each edge 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. 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	302us		186ms		285us		295us		188ms	
5K	7.5K	415us	1.37	279ms	1.50	413us	1.45	426us	1.44	281ms	1.49
5K	15K	1.05ms	2.53	705ms	2.53	1.05ms	2.54	925us	2.17	711ms	2.53
5K	30K	2.95ms	2.81	1.82sec	2.58	3.28ms	3.12	2ms	2.16	1.84sec	2.59
5K	60K	8.7ms	2.95	5.14sec	2.82	15.4ms	4.70	5.07ms	2.54	5.21sec	2.83
fixed average degree, increasing n											
2K	6K	379us		98.1ms		371us		328us		100ms	
4K	12K	837us	2.21	434ms	4.42	808us	2.18	709us	2.16	439ms	4.39
8K	24K	1.72ms	2.05	1.91sec	4.40	1.8ms	2.23	1.56ms	2.20	1.92sec	4.37
16K	48K	3.78ms	2.20	8.4sec	4.40	5.19ms	2.88	3.98ms	2.55	8.44sec	4.40
32K	96K	8.81ms	2.33	36.1sec	4.30	14ms	2.70	11.1ms	2.79	36.2sec	4.29

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?

   Findpath should be called at most $n\Delta$ times. Since each call to findpath takes O(m) time, the overall running time is $O(mn\Delta)$. When we increase m, we also increase the average vertex degree and Δ by the same factor, so doubling the number of edges should increase the runtime by a factor of 4. We observe a slower growth rate in the data. Since most of the time is spent on initialization within findpath, it appears that the number of calls to findpath may be growing more slowly than predicted by the analysis.

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?

   In this case, the ratio should be a little larger than 4. The number of vertices and edges both double, but since the average degree is held constant, the max degree will grow only by a small amount. The observed growth rates are consistent with this expectation.

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 difference here is pretty stunning. Typically fpInit is typically hundreds of times larger than fpLoop and in a few cases it is more than a thousand times larger. The explanation appears to be that the actual search for a path often terminates after just examining a few edges in the graph. On the other hand, the initialization always takes the worst-case amount of time, since we have to do some operation for every edge and every vertex.

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.

   One change is to avoid most of the calls to findpath by constructing an initial matching without using findpath. One way to do this is to examine edges incident to max degree vertices, and add such an edge so long as it does not conflict with any edge already in the matching. This can be done quickly and will typically match most of the max degree vertices, reducing the number of times we need to call findpath.

   Another thing we can do is to avoid searching for a max degree vertex every time we call findpath. If we make a list of such vertices when we first start maxdMatch, we can select vertices from this list in findpath. Of course, to make this work, whenever a max degree vertex is matched, we also need to remove that vertex from the list.

   We can also eliminate the initialization associated with the state values in findpath, by using a different method of determining if a vertex has been visited in the current search. The idea is we number the calls to findpath 1, 2, 3, and so forth. We maintain an array visited[u] and set visited[u]=i if u is visited during the i-th call. This we way, we can still test a vertex in findpath to see if its been visited during the current call, but we don't need the state values to tell us this.

   If we're careful, we can also eliminate the need to initialize pEdge and mEdge. In the case of mEdge, this requires that we update the values of mEdge when flipping the edges along a path, in and out of the matching.

   One final thing we can do is move the declaration of the queue used by findpath outside of findpath itself. The declaration triggers a call to the constructor of the underlying Dlist object, which takes O(m) time to complete. We can do better by just initializing this once in maxdMatch and then just clearing it at the start of findpath.

Part B. 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.

```
#ifndef FMAXDMATCH_H
#define FMAXDMATCH_H

#include "maxdMatch.h"
#include "Dlist.h"

using namespace grafalgo;

/** This class encapsulates data and methods used to find a matching
 *  that matches all vertices of maximum degree. The algorithm is
 *  invoked using the constructor.
 */
class fmaxdMatch : public maxdMatch {
public:

     fmaxdMatch(Graph&,Dlist&,Stats&);
private:
     edge*   mEdge;      ///< mEdge[u] is matching edge incident to  u
     Dlist*  maxdVerts;  ///< list of max degree vertices
     List*   q;      ///< queue of edges used in findpath
     int*visited;///< visited[u]=i if u visited in phase i
     int phase;      ///< each call to findpath starts new phase

     void    extend(edge);
     edge    findPath();

     voidinit(Graph&, Dlist&, Stats&);
     voidcleanup();
};

#endif

#include "fmaxdMatch.h"

using namespace grafalgo;

/** 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, Stats& stats1) {
     int t0 = Util::getTime();

     maxdMatch::init(graf1,match1,stats1);

     // find an initial matching, by examining edges at max degree
     // vertices and adding the first non-conflicting edge we find;
     // account for the time spent on this using fpLoop
     int t1 = Util::getTime();
     for (vertex u = 1; u <= graf->n(); u++) {
         if (d[u] != maxd || mEdge[u] != 0) continue;
         for (edge e = graf->firstAt(u); e != 0; e = graf->nextAt(u,e)) {
             vertex v = graf->mate(u,e);
```

```
            if (mEdge[v] == 0) {
                match->addLast(e);
                mEdge[u] = mEdge[v] = e;
                if (maxdVerts->member(u))
                    maxdVerts->remove(u);
                if (maxdVerts->member(v))
                    maxdVerts->remove(v);
                break;
            }
        }
    }
    stats->fpLoop += Util::getTime() - t1;

    edge e;
    phase = 1;
    while((e = findPath()) != 0) {
        extend(e);
        phase++;
    }

    maxdMatch::cleanup(); cleanup();
    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 visited[] which keeps track of the most recent
 *  phase in which each vertex has been visited.
 */
void fmaxdMatch::init(Graph& graf1, Dlist& match1, Stats& stats1) {
    int t0 = Util::getTime();

    // initialize stuff in base class
    maxdMatch::init(graf1,match1,stats1);

    // allocate storage for added data structures
    mEdge = new edge[graf->n()+1];
    maxdVerts = new Dlist(graf->n());
    visited = new int[graf->n()+1];
    q = new List(maxe);

    for (vertex u = 1; u <= graf->n(); u++) {
        pEdge[u] = mEdge[u] = visited[u] = 0;
        if (d[u] == maxd) maxdVerts->addLast(u);
    }

    stats->maxdInit += Util::getTime() - t0;
}

void fmaxdMatch::cleanup() {
    delete [] mEdge;  delete [] visited; delete maxdVerts;
}

/** 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;

     if (match->member(e)) {
         u = graf->left(e);
         if (pEdge[u] != e) u = graf->right(e);
         mEdge[u] = 0;
         while (pEdge[u] != 0) {
             e = pEdge[u]; match->remove(e);  u = graf->mate(u,e);
             e = pEdge[u]; match->addLast(e);
             mEdge[u] = e; u = graf->mate(u,e); mEdge[u] = e;
         }
         stats->extend += Util::getTime() - t0;
         return;
     }
     match->addLast(e);
     u = graf->left(e); v = graf->right(e);
     if (maxdVerts->member(u)) maxdVerts->remove(u);
     if (maxdVerts->member(v)) maxdVerts->remove(v);
     mEdge[u] = mEdge[v] = e;
     if (pEdge[u] == 0) u = v;
     while (pEdge[u] != 0) {
         e = pEdge[u];
         match->remove(e);  u = graf->mate(u,e);
         e = pEdge[u]; match->addLast(e);
         mEdge[u] = e; u = graf->mate(u,e); mEdge[u] = 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();

     // find a max degree vertex that's unmatched
     vertex root = maxdVerts->first();
     if (root == 0) return 0;
     maxdVerts->removeFirst();
     visited[root] = phase;

     q->clear();
     for (edge e = graf->firstAt(root); e != 0; e = graf->nextAt(root,e)) {
         q->addLast(e);
     }
     stats->fpInit += Util::getTime() - t0;
     t0 = Util::getTime();

     edge e;
     while (!q->empty()) {
         e = q->first(); q->removeFirst();
         vertex v = (visited[graf->left(e)] == phase ?
                 graf->left(e) : graf->right(e));
```

```
            vertex w = graf->mate(v,e);
            if (visited[w] == phase) continue;
            if (mEdge[w] == 0) { pEdge[w] = 0; break; }
            vertex x = graf->mate(w,mEdge[w]);
            visited[w] = phase; pEdge[w] = e;
            visited[x] = phase; pEdge[x] = mEdge[x];
            if (d[x] < maxd) { e = pEdge[x]; break; }
            for (edge ee = graf->firstAt(x); ee != 0;
                      ee = graf->nextAt(x,ee)) {
                if ((ee != mEdge[x]) && !q->member(ee))
                    q->addLast(ee);
            }
        }
        stats->fpLoop += Util::getTime() - t0;
        return e;
    }
```

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

```
% checkFedgeColor <bg10a
10 1 8 0 25
{(k,i), (m,f), (o,e), (l,b), (s,a), (r,h)}
{(k,f), (r,c)}
{(l,c), (m,b), (s,i), (o,g), (r,f), (p,e)}
{(l,g), (s,f), (r,e), (m,d), (q,c), (p,b), (t,j), (o,i)}
{(l,f), (t,d), (m,c), (o,b), (r,i), (s,e), (p,g)}
{(r,d)}
% checkFedgeColor <bg50a
43 3 21 2 70
{(51,44), (77,41), (56,37), (52,33), (53,30), (67,27), (65,25), (57,18),
(96,19), (87,10), (81,42), (80,29), (79,15), (74,13), (72,50), (69,47)}
{(51,42), (100,41), (63,40), (85,39), (79,38), (92,37), (91,35), (95,34),
(81,33), (68,32), (88,30), (72,29), (66,28), (93,27), (96,25), (82,24),
(76,23), (56,22), (75,20), (97,18), (89,14), (78,13), (61,12), (64,11),
(74,10), (67,9), (84,7), (90,6), (69,5), (70,4), (57,3), (87,1), (94,50),
(62,49), (58,48), (80,47), (71,46), (77,45), (86,44), (65,43)}
{(51,30), (55,27), (52,18), (87,23), (79,35), (69,21), (57,44), (73,41),
(59,33)}
{(52,4), (100,3), (57,46), (71,16), (96,33), (81,32), (68,15), (91,7),
(84,21), (89,29), (72,48), (58,26), (70,22), (74,47), (78,10), (92,44),
(93,37), (97,50), (90,45), (80,41), (63,38), (88,35), (69,30), (87,27),
(94,25), (79,23), (83,20), (86,18), (77,14), (75,13), (66,9), (76,6)}
{(52,47), (64,44), (96,41), (59,38), (58,37), (78,35), (80,33), (76,32),
(89,30), (56,29), (79,27), (88,25), (66,23), (68,20), (69,14), (71,13),
(74,7), (55,6), (93,18), (91,45), (100,5), (94,36), (90,16), (87,21),
(81,12), (75,49), (72,19), (82,50), (54,48)}
{(52,6), (96,9), (93,12), (87,5), (81,2), (79,19), (53,48), (82,47),
(70,45), (62,44), (85,41), (57,38), (66,37), (65,35), (61,33), (74,30),
(69,27), (72,25), (59,23), (80,18), (58,14)}
{(60,41), (87,34), (69,12)}
```

Also, type the following commands

```
% evalFedgeColor 10 20
10 20 10.8us 0.3us 2.3us 0.3us 15us 19.7us
% evalFedgeColor 100 200
100 200 51.6us 0.8us 24.5us 1.1us 72.4us 95.3us
% evalFedgeColor 1000 2000
1000 2000 448us 7.9us 185us 8us 530us 711us
```

3. (10 points) Run the provided *script2* and use the resulting data to complete 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	823us		10.4us		438us		9.60us		1.27ms	
5K	7.5K	1.04ms	1.32	18.9us	1.82	568us	1.30	18.80us	1.96	1.66ms	1.31
5K	15K	2.17ms	2.23	80.6us	4.26	1.25ms	2.20	69.8us	3.71	3.62ms	2.18
5K	30K	5.39ms	2.57	247us	3.06	2.94ms	2.35	180us	2.58	8.81ms	2.43
5K	60K	14.6ms	2.85	783us	3.17	8.33ms	2.83	466us	2.59	24.6ms	2.79
fixed average degree, increasing n											
2K	6K	887us		26.9us		465us		26us		1.45ms	
4K	12K	1.74ms	1.99	60.1us	2.23	977us	2.10	53us	2.06	2.86ms	1.97
8K	24K	3.61ms	2.07	133us	2.21	2.16ms	2.21	116us	2.16	6.08ms	2.13
16K	48K	7.59ms	2.14	274us	2.06	4.69ms	2.17	248us	2.14	13ms	2.14
32K	96K	18.1ms	2.36	697us	2.54	12.2ms	2.60	643us	2.59	32ms	2.47

4. (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?

   For the largest graphs, the running times have improved by more than a factor of 1000. Most of the time is now being spent in the initialization for maxdMatch. This has actually gone up, relative to the earlier case. Now however, the path search in findpath does account for a significant fraction of the total time (typically about one third of the total) and the initialization in findpath has dropped to a much smaller fraction of the running time.

   I do not see a lot of opportunity for further improvement, at least not using this algorithm. It turns out that there are other algorithms for graph edge coloring that can perform better, but they use a different approach.

5. (10 points) How do the growth rates for the running time compare to the original version. How do you account for the differences?

   When n is held fixed and m is increased, the growth rates are pretty similar to those for the original version. On the other hand, when we increase n while holding the average degree fixed, the growth rates are much better. Previously fpInit was growing by a factor of 4 or more each time we increased n. Now that fpInit has been substantially reduced, it no longer causes this faster growth rate.