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

Lab 5 Solution

Part A. Ddheap

1. (10 points) Your code repository includes a class *Ddheap* that extends the *Dheap* data structure, by adding the *addtokeys* operation. Complete the implementation of *Ddheap* and past a copy of your code below. Highlight your changes by making them bold. You may omit methods you did not change.

```
/** @file Ddheap.h
 *
 *  @author Jon Turner
 *  @date 2011
 *  This is open source software licensed under the Apache 2.0 license.
 *  See http://www.apache.org/licenses/LICENSE-2.0 for details.
 */

#ifndef DDHEAP_H
#define DDHEAP_H

#include "Dheap.h"

namespace grafalgo {

/** This class implements a heap data structure.
 *  The heap elements are identified by integers in 1..n where n
 *  is specified when an object is constructed.
 */
class Ddheap : public Dheap {
public:         Ddheap(int,int);
                ~Ddheap();

        // common methods
        void    clear();
        void    copyFrom(const Ddheap&);

        // access methods
        keytyp  key(index) const;

        // modifiers
        void    insert(index,keytyp);
        void    changekey(index,keytyp);
        void    addtokeys(keytyp);

        string& toString(string&) const;
private:
        keytyp  delta;
};

/** Get the key of item.
 *  @param i is the index of an item in the heap
```

```
 *  @return the value of i's key
 */
inline keytyp Ddheap::key(index i) const { return kee[i] + delta; }

/** Add to the keys of all items in the heap.
 *  @param x is value to add to all the keys.
 */
inline void Ddheap::addtokeys(keytyp x) { delta += x; }

} // ends namespace

#endif

/** @file Ddheap.cpp
 *
 *  @author Jon Turner
 *  @date 2011
 *  This is open source software licensed under the Apache 2.0 license.
 *  See http://www.apache.org/licenses/LICENSE-2.0 for details.
 */

#include "Ddheap.h"

#define p(x) (((x)+(d-2))/d)
#define left(x) (d*((x)-1)+2)
#define right(x) (d*(x)+1)

namespace grafalgo {

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

/** Destructor for Ddheap class. */
Ddheap::~Ddheap() { }

/** Copy into Ddheap from source. */
void Ddheap::copyFrom(const Ddheap& source) {
    this->Dheap::copyFrom(source);
    delta = source.delta;
}

/** Clear contents of heap. */
void Ddheap::clear() {
    this->Dheap::clear(); delta = 0;
}

/** Add item to the heap.
 *  @param i is the index of an item that is not in the heap
 *  @param k is the key value under which i is to be inserted
 */
void Ddheap::insert(index i, keytyp k) {
    kee[i] = k - delta; hn++; siftup(i,hn);
}
```

```
/** Change the key of an item in the heap.
 *  @param i is the index of an item that is not in the heap
 *  @param k is the new key value for i
 */
void Ddheap::changekey(index i, keytyp k) {
    this->Dheap::changekey(i,k-delta);
}

/** Construct a string representation of this object.
 *  @param s is a string in which the result is returned
 *  @return a reference to s
 */
string& Ddheap::toString(string& s) const {
    stringstream ss;
    for (int i = 1; i <= hn; i++) {
        ss << "(" << item2string(h[i],s) << "," << kee[h[i]] + delta
           << ")";
        if (i < hn) ss << " ";
    }
    s = ss.str();
    return s;
}

} // ends namespace
```

2. (10 points) Compile and run the program *testDdheap* that you will find in the *unit* subdirectory of the *heaps* directory. Paste a copy of the output below.

```
all tests passed
```

Part B. EdmondsBW and EdmondsBWmin.

1. (40 points). The *match* subdirectory includes a partial implementation of Edmond's algorithm for max weight matchings in bipartite graphs. Paste a copy of your completed version of Edmonds algorithm below.

```
// Header file for class that implements Edmond's algorithm for
// finding a maximum weight matching in a bipartite graph. To use,
// invoke the constructor.

#ifndef EDMONDSBW_H
#define EDMONDSBW_H

#include "stdinc.h"
#include "Wgraph.h"
#include "Dlist.h"
#include "Ddheap.h"

using namespace grafalgo;

/** This class implements Edmond's algorithm for bipartite weighted
graphs.
 *  This is a primal-dual algorithm.
 */
class edmondsBW {
public: edmondsBW(Wgraph&,Dlist&,int&,int&);
private:
    Wgraph* graf;       ///< graph we're finding matching for
    Dlist*  match;      ///< matching we're building

    enum stype {unreached, odd, even};
    stype   *state;     ///< state used in augmenting path search
    edge*   mEdge;      ///< mEdge[u] is matching edge incident to u
    edge*   pEdge;      ///< p[u] is parent of u in forest
    double* z;          ///< z[u] is label for vertex u (dual variable)

    Ddheap* h1o;        ///< heap of odd vertices by label
    Ddheap* h1e;        ///< heap of even vertices by label
    Ddheap* h2;         ///< edges joining even/unreached by slack
    Ddheap* h3;         ///< edges joining even/even by slack

    int maxwt;          ///< maximum edge weight

    double augment(edge);
    edge findpath();
};

#endif


#include "edmondsBW.h"

using namespace grafalgo;

/** Find a maximum weighted matching in a bipartite graph.
 *  @param graf1 is a reference to a bipartite graph
 *  @param match1 is a reference to a list of edges in which
 *  result is returned
 *  @param size is a reference to an integer in which the size
```

```
 *  of the matching is returned
  * @param weight is a reference to an integer in which the weight
 *  of the matching is returned
 */
edmondsBW::edmondsBW(Wgraph& graf1, Dlist& match1, int& size, int& weight)
         : graf(&graf1), match(&match1) {
    state = new stype[graf->n()+1];
    mEdge = new edge[graf->n()+1];
    pEdge = new edge[graf->n()+1];
    z = new double[graf->n()+1];

    h1o = new Ddheap(graf->n(),2); h1e = new Ddheap(graf->n(),2);
    int maxe = 0; maxwt = 0;
    for (edge e = graf->first(); e != 0; e = graf->next(e)) {
        maxe = max(e,maxe); maxwt = max(graf->weight(e),maxwt);
    }
    h2 = new Ddheap(maxe,2); h3 = new Ddheap(maxe,2);

    for (vertex u = 1; u <= graf->n(); u++) z[u] = maxwt/2.0;

    edge e; size = 0; weight = 0;
    while((e = findpath()) != 0) { weight += augment(e); size++; }

    delete [] state; delete [] mEdge;
    delete [] pEdge; delete [] z;
    delete h1o; delete h1e; delete h2; delete h3;
}

/** Augment the current matching, using the forest built by findpath.
 *  @param e is an edge joining two trees in the forest built by
 *  findpath
 */
double edmondsBW::augment(edge e) {
    vertex u = graf->left(e);
    double pathWeight = 0;;
    while (pEdge[u] != 0) {
        edge ee;
        ee = pEdge[u]; match->remove(ee);  u = graf->mate(u,ee);
        pathWeight -= graf->weight(ee);
        ee = pEdge[u]; match->addLast(ee); u = graf->mate(u,ee);
        pathWeight += graf->weight(ee);
    }
    vertex v = graf->right(e);
    while (pEdge[v] != 0) {
        edge ee;
        ee = pEdge[v]; match->remove(ee);  v = graf->mate(v,ee);
        pathWeight -= graf->weight(ee);
        ee = pEdge[v]; match->addLast(ee); v = graf->mate(v,ee);
        pathWeight += graf->weight(ee);
    }
    if (u == v) Util::fatal("edmondsBW::augment: graph not bipartite");
    match->addLast(e);
    pathWeight += graf->weight(e);
    return pathWeight;
}

/** Search for an augmenting path in the bipartite graph graf.
 *  @return the edge that joins two separate trees in the forest
 *  defined by pEdge; this edge, together with the paths
```

```
 *  to the tree roots forms an augmenting path; return 0 if
 *  no augmenting path, or if the current matching has maximum weight
 */
edge edmondsBW::findpath() {
    for (vertex u = 1; u <= graf->n(); u++) {
        state[u] = even; 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;
    }
    for (vertex u = 1; u <= graf->n(); u++) {
        if (state[u] == even) h1e->insert(u,z[u]);
    }
    if (h1e->size() < 2) return 0;
    for (edge e = graf->first(); e != 0; e = graf->next(e)) {
        vertex u = graf->left(e);
        vertex v = graf->right(e);
        if (state[u] == even || state[v] == even) {
            if (state[u] != state[v])
                h2->insert(e,z[u]+z[v]-graf->weight(e));
            else
                h3->insert(e,z[u]+z[v]-graf->weight(e));
        }
    }

    while (true) {
        if (!h3->empty() && h3->key(h3->findmin()) == 0) {
            edge e = h3->findmin();
            // update labels for vertices in heaps
            while (!h1e->empty()) {
                vertex u = h1e->findmin();
                z[u] = h1e->key(u);
                h1e->deletemin();
            }
            while (!h1o->empty()) {
                vertex u = h1o->findmin();
                z[u] = h1o->key(u);
                h1o->deletemin();
            }
            h2->clear(); h3->clear();
            return e;
        }
        if (!h2->empty() && h2->key(h2->findmin()) == 0) {
            edge e = h2->deletemin();
            vertex v = (state[graf->left(e)] == even ?
                    graf->left(e) : graf->right(e));
            vertex w = graf->mate(v,e);
            vertex x = graf->mate(w,mEdge[w]);
            state[w] = odd; pEdge[w] = e;
            state[x] = even; pEdge[x] = mEdge[x];
            h1e->insert(x,z[x]); h1o->insert(w,z[w]);

            // remove edges at w from h2 if present
            for (edge ee = graf->firstAt(w); ee != 0;
                  ee = graf->nextAt(w,ee)) {
                if (h2->member(ee)) h2->remove(ee);
```

```
            }

            // add edges at x to h3, and remove from h2 as needed
            for (edge ee = graf->firstAt(x); ee != 0;
                      ee = graf->nextAt(x,ee)) {
                if (ee == mEdge[x]) continue;
                vertex y = graf->mate(x,ee);
                if (state[y] == unreached && !h2->member(ee)) {
                    h2->insert(ee,z[x]+z[y]
                            -graf->weight(ee));
                } else if (state[y] == even) {
                    h2->remove(ee);
                    z[y] = h1e->key(y);
                    h3->insert(ee,z[x]+z[y]
                            -graf->weight(ee));
                }
            }
            continue;
        }
        // relabel
        double delta = h1e->key(h1e->findmin());
        if (delta == 0) return 0; // current matching is max weight
        if (!h2->empty()) delta = min(delta,h2->key(h2->findmin()));
        if (!h3->empty()) delta = min(delta,h3->key(h3->findmin())/2);
        h1e->addtokeys(-delta); h1o->addtokeys(delta);
        h2->addtokeys(-delta); h3->addtokeys(-2*delta);
    }
}
```

2. (20 points) The *match* subdirectory also includes a partial implementation of the variant of Edmond's algorithm that finds minimum weight matchings of maximum size. Paste a copy of your completed code below. Highlight the places where this version differs from the previous one, by making them bold.

```
// Header file for class that implements Edmond's algorithm for
// finding a minimum weight matching of maximum size matching in
// a bipartite graph. To use, invoke the constructor.

#ifndef EDMONDSBWMIN_H
#define EDMONDSBWMIN_H

#include "stdinc.h"
#include "Wgraph.h"
#include "Dlist.h"
#include "Ddheap.h"

using namespace grafalgo;

/** This class implements Edmond's algorithm for bipartite weighted
graphs.
 *  This is a primal-dual algorithm.
 */
class edmondsBWmin {
public: edmondsBWmin(Wgraph&,Dlist&,int&,int&);
private:
    Wgraph* graf;       ///< graph we're finding matching for
    Dlist*  match;      ///< matching we're building

    enum stype {unreached, odd, even};
```

```
    stype   *state;     ///< state used in augmenting path search
    edge*   mEdge;      ///< mEdge[u] is matching edge incident to u
    edge*   pEdge;      ///< p[u] is parent of u in forest
    double* z;          ///< z[u] is label for vertex u (dual variable)

    Ddheap* h1o;        ///< heap of odd vertices by label
    Ddheap* h1e;        ///< heap of even vertices by label
    Ddheap* h2;         ///< edges joining even/unreached by slack
    Ddheap* h3;         ///< edges joining even/even by slack

    int maxwt;          ///< maximum edge weight

    double augment(edge);
    edge findpath();
};

#endif

#include "edmondsBWmin.h"

using namespace grafalgo;

/** Find a maximum weighted matching in a bipartite graph.
 *  @param graf1 is a reference to a bipartite graph
 *  @param match1 is a reference to a list of edges in which
 *  result is returned
 *  @param size is a reference to an integer in which the size
 *  of the matching is returned
 */
edmondsBWmin::edmondsBWmin(Wgraph& graf1, Dlist& match1, int& size,
            int& weight) : graf(&graf1), match(&match1) {
    state = new stype[graf->n()+1];
    mEdge = new edge[graf->n()+1];
    pEdge = new edge[graf->n()+1];
    z = new double[graf->n()+1];

    // negate the edge weights
    int maxe = 0; maxwt = -graf->weight(graf->first());
    for (edge e = graf->first(); e != 0; e = graf->next(e)) {
        graf->setWeight(e,-graf->weight(e));
        maxe = max(e,maxe); maxwt = max(graf->weight(e),maxwt);
    }
    h1o = new Ddheap(graf->n(),2); h1e = new Ddheap(graf->n(),2);
    h2 = new Ddheap(maxe,2); h3 = new Ddheap(maxe,2);

    for (vertex u = 1; u <= graf->n(); u++) z[u] = maxwt/2.0;

    edge e; size = 0; weight = 0;
    while((e = findpath()) != 0) { weight -= augment(e); size++; }

    // restore original edge weights
    for (edge e = graf->first(); e != 0; e = graf->next(e))
        graf->setWeight(e,-graf->weight(e));

    delete [] state; delete [] mEdge;
    delete [] pEdge; delete [] z;
    delete h1o; delete h1e; delete h2; delete h3;
}
```

```
/** Augment the current matching, using the forest built by findpath.
 *  @param e is an edge joining two trees in the forest built by
 *  findpath
 */
double edmondsBWmin::augment(edge e) {
    vertex u = graf->left(e);
    double pathWeight = 0;;
    while (pEdge[u] != 0) {
        edge ee;
        ee = pEdge[u]; match->remove(ee);  u = graf->mate(u,ee);
        pathWeight -= graf->weight(ee);
        ee = pEdge[u]; match->addLast(ee); u = graf->mate(u,ee);
        pathWeight += graf->weight(ee);
    }
    vertex v = graf->right(e);
    while (pEdge[v] != 0) {
        edge ee;
        ee = pEdge[v]; match->remove(ee);  v = graf->mate(v,ee);
        pathWeight -= graf->weight(ee);
        ee = pEdge[v]; match->addLast(ee); v = graf->mate(v,ee);
        pathWeight += graf->weight(ee);
    }
    if (u == v) Util::fatal("edmondsBWmin::augment: graph not bipartite");
    match->addLast(e);
    pathWeight += graf->weight(e);
    return pathWeight;
}

/** Search for an augmenting path in the bipartite graph graf.
 *  @return the edge that joins two separate trees in the forest
 *  defined by pEdge; this edge, together with the paths
 *  to the tree roots forms an augmenting path; return 0 if
 *  no augmenting path, or if the current matching has maximum weight
 */
edge edmondsBWmin::findpath() {
    for (vertex u = 1; u <= graf->n(); u++) {
        state[u] = even; 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;
    }
    for (vertex u = 1; u <= graf->n(); u++) {
        if (state[u] == even) h1e->insert(u,z[u]);
    }
    if (h1e->size() < 2) return 0;
    for (edge e = graf->first(); e != 0; e = graf->next(e)) {
        vertex u = graf->left(e);
        vertex v = graf->right(e);
        if (state[u] == even || state[v] == even) {
            if (state[u] != state[v])
                h2->insert(e,z[u]+z[v]-graf->weight(e));
            else
                h3->insert(e,z[u]+z[v]-graf->weight(e));
        }
    }
```

```
while (true) {
    if (!h3->empty() && h3->key(h3->findmin()) == 0) {
        edge e = h3->findmin();
        // update labels for vertices in heaps
        while (!h1e->empty()) {
            vertex u = h1e->findmin();
            z[u] = h1e->key(u);
            h1e->deletemin();
        }
        while (!h1o->empty()) {
            vertex u = h1o->findmin();
            z[u] = h1o->key(u);
            h1o->deletemin();
        }
        h2->clear(); h3->clear();
        return e;
    }
    if (!h2->empty() && h2->key(h2->findmin()) == 0) {
        edge e = h2->deletemin();
        vertex v = (state[graf->left(e)] == even ?
                graf->left(e) : graf->right(e));
        vertex w = graf->mate(v,e);
        vertex x = graf->mate(w,mEdge[w]);
        state[w] = odd; pEdge[w] = e;
        state[x] = even; pEdge[x] = mEdge[x];
        h1e->insert(x,z[x]); h1o->insert(w,z[w]);

        // remove edges at w from h2 if present
        for (edge ee = graf->firstAt(w); ee != 0;
              ee = graf->nextAt(w,ee)) {
            if (h2->member(ee)) h2->remove(ee);
        }

        // add edges at x to h3, and remove from h2 as needed
        for (edge ee = graf->firstAt(x); ee != 0;
                  ee = graf->nextAt(x,ee)) {
            if (ee == mEdge[x]) continue;
            vertex y = graf->mate(x,ee);
            if (state[y] == unreached && !h2->member(ee)) {
                h2->insert(ee,z[x]+z[y]
                        -graf->weight(ee));
            } else if (state[y] == even) {
                h2->remove(ee);
                z[y] = h1e->key(y);
                h3->insert(ee,z[x]+z[y]
                        -graf->weight(ee));
            }
        }
        continue;
    }
    // relabel
    double delta;
    if (h2->empty() && h3->empty()) return 0;
    if (h2->empty()) delta = h3->key(h3->findmin())/2;
    else if (h3->empty()) delta = h2->key(h2->findmin());
    else delta = min(h2->key(h2->findmin()),
             h3->key(h3->findmin())/2);
    h1e->addtokeys(-delta); h1o->addtokeys(delta);
    h2->addtokeys(-delta); h3->addtokeys(-2*delta);
```

```
    }
}
```

3. (20 points) Demonstrate your programs by typing and pasting the results below.

```
% checkEdmonds <wbg5
4 313 (d,g,97) (b,h,90) (c,f,79) (a,j,47)
5 296 (c,h,23) (a,j,47) (d,f,72) (e,g,72) (b,i,82)
% checkEdmonds <wbg10
8 544 (a,q,90) (j,l,93) (c,n,86) (b,s,73) (e,r,45) (g,m,40) (d,p,75)
(h,t,42)
8 484 (a,t,11) (e,r,45) (d,q,78) (h,p,84) (j,l,93) (b,s,73) (g,m,40)
(c,k,60)
% checkEdmonds <wbg50
48 34993 (30,100,997) (34,65,989) (49,90,948) (35,75,933) (13,81,891)
(33,52,890) (8,79,888) (2,94,879) (11,70,882) (17,72,828) (25,96,828)
(48,85,956) (36,88,852) (43,61,816) (32,64,801) (19,92,752) (16,67,945)
(24,86,720) (47,83,710) (29,73,646) (12,80,630) (18,71,577) (45,57,943)
(28,55,742) (4,62,659) (3,74,795) (31,76,950) (41,78,472) (15,91,440)
(20,89,429) (7,56,378) (38,97,988) (26,53,817) (21,68,362) (40,99,622)
(37,51,466) (1,87,594) (46,58,872) (5,60,821) (6,82,900) (9,95,533)
(23,77,620) (27,98,638) (50,63,243) (10,93,817) (22,66,612) (39,84,677)
(42,59,245)
50 17090 (6,89,101) (2,92,103) (33,55,108) (34,93,111) (3,57,129)
(12,75,131) (25,97,143) (50,100,153) (20,51,158) (16,54,159) (29,66,229)
(31,98,130) (13,73,248) (10,70,175) (42,59,245) (26,77,278) (47,88,310)
(43,91,207) (9,84,154) (18,74,185) (32,94,342) (30,68,179) (45,76,328)
(39,69,372) (36,83,270) (44,67,176) (37,78,190) (7,56,378) (22,52,412)
(38,58,462) (48,63,477) (14,80,132) (41,61,503) (19,85,556) (1,87,594)
(23,71,196) (40,99,622) (15,95,387) (17,82,467) (35,53,656) (4,62,659)
(11,65,141) (27,72,398) (28,96,186) (46,64,611) (24,86,720) (5,60,821)
(49,90,948) (21,81,562) (8,79,888)
```

Part C. Karp's Algorithm.

1. (40 points) The *lab5* subdirectory includes a file *karp.cpp* that contains a template for an implementation of Karp's algorithm for the asymmetric traveling salesman problem. Complete the implementation and paste a copy of your code below.

```
#include "stdinc.h"
#include "Dlist.h"
#include "CListSet.h"
#include "Wdigraph.h"
#include "edmondsBWmin.h"

using namespace grafalgo;

class KarpStats {
public:
    int numCycles;      // number of cycles
    int maxCycleSize;   // size of largest cycle
    int cyclesWeight;   // weight of collection of cycles
    int tourWeight;     // weight of final tour
    int matchTime;      // time to compute min-weight perfect matching
    int totalTime;      // total run time

    KarpStats() { clear(); }
    void clear() {
        cyclesWeight = tourWeight = 0;
        numCycles = maxCycleSize = 0;
        matchTime = totalTime = 0;
    }
    void add(KarpStats& from) {
        numCycles += from.numCycles;
        maxCycleSize += from.maxCycleSize;
        cyclesWeight += from.cyclesWeight;
        tourWeight += from.tourWeight;
        matchTime += from.matchTime;
        totalTime += from.totalTime;
    }
    string& toString(string& s) {
        stringstream ss;
        ss << numCycles << " " << maxCycleSize << " "
           << cyclesWeight << " " << tourWeight << " "
           << matchTime << " " << totalTime;
        s = ss.str();
        return s;
    }
};

/** Find a traveling salesman tour using Karp's algorithm.
 *  @param graf1 is a reference to a weighted digraph object
 *  @param tour is a reference to a circular list set object
 *  in which the vertices in the tour are returned in order
 *  @param stats is a reference to a KarpStats object in which
 *  performance statistics are returned.
 */
void karp(Wdigraph& graf, ClistSet& tour, KarpStats& stats) {
    int t0 = Util::getTime();
    // construct bipartite graph corresponding to graf1
    int maxe = graf.first();;
    Wgraph bigraf(2*graf.n(),graf.m());
```

```
for (edge e = graf.first(); e != 0; e = graf.next(e)) {
    edge ee = bigraf.join(graf.tail(e),graf.n()+graf.head(e));
    bigraf.setWeight(ee,graf.length(e));
    maxe = max(e,maxe);
}
Dlist match(bigraf.m());
int t1 = Util::getTime();
int mSize, mWeight;
edmondsBWmin(bigraf,match,mSize,mWeight);
stats.matchTime = Util::getTime() - t1;
stats.cyclesWeight = mWeight;

// construct collection of cycles based on matching list
// first identify the tour edge leaving each vertex
tour.clear();
edge tEdge[graf.n()+1];
for (vertex u = 1; u <= graf.n(); u++) tEdge[u] = 0;
for (edge e = match.first(); e != 0; e = match.next(e)) {
    vertex u = bigraf.left(e);
    vertex v = bigraf.right(e) - graf.n();
    edge ee = graf.getEdge(u,v);
    tEdge[graf.tail(ee)] = ee;
}
bool partialTour = false;
for (vertex u = 1; u <= graf.n(); u++) {
    if (tEdge[u] == 0) { partialTour = true; continue; }
    vertex v = graf.mate(u,tEdge[u]);
    if (v != tour.pred(u)) tour.join(u,v);
}

// patch the cycles into a single cycle
// first identify a "reference edge" for each cycle and
// identify the longest cycle
Dlist cycRef(graf.n());     // list with one vertex from each cycle
bool mark[graf.n()+1];      // used to mark visited edges
vertex longest = 0;     // reference vertex for longest cycle
int longestLen = 0;     // length of longest cycle
for (vertex u = 1; u <= graf.n(); u++) mark[u] = false;
stats.numCycles = 0;
for (vertex u = 1; u <= graf.n(); u++) {
    if (mark[u] == true) continue;
    mark[u] = true;
    stats.numCycles++;
    cycRef.addLast(u);
    int len = 1;
    for (vertex v = tour.suc(u); v != u; v = tour.suc(v)) {
        len++; mark[v] = true;
    }
    if (len > longestLen) { longest = u; longestLen = len; }
}
stats.maxCycleSize = longestLen;

// next, move longest cycle to end of list
cycRef.remove(longest); cycRef.addLast(longest);

// merge first and last cycle until only one cycle left
stats.tourWeight = stats.cyclesWeight;
while (cycRef.length() > 1) {
    // find best place to patch first and last cycle
```

```
        vertex u1 = cycRef.first(); vertex u2 = cycRef.last();
        edge e1 = tEdge[u1]; edge e2 = tEdge[u2];
        if (e1 == 0) { cycRef.removeFirst(); continue; }
        if (e2 == 0) { cycRef.removeLast();  continue; }
        vertex best1, best2; int bestcost; best1 = 0;
        edge x1 = graf.getEdge(u1,graf.head(e2));
        edge x2 = graf.getEdge(u2,graf.head(e1));
        if (x1 != 0 && x2 != 0) {
            best1 = u1; best2 = u2;
            bestcost = graf.length(x1) + graf.length(x2)
                   - (graf.length(e1) + graf.length(e2));
        }
        for (vertex v1 = tour.suc(u1); v1 != u1; v1 = tour.suc(v1)) {
            for (vertex v2 = tour.suc(u2); v2 != u2;
                  v2 = tour.suc(v2)) {
                e1 = tEdge[v1]; e2 = tEdge[v2];
                x1 = graf.getEdge(v1,graf.head(e2));
                x2 = graf.getEdge(v2,graf.head(e1));
                if (x1 == 0 || x2 == 0) continue;
                int cost = graf.length(x1) + graf.length(x2)
                     - (graf.length(e1) + graf.length(e2));
                if (best1 == 0 || cost < bestcost) {
                    best1 = v1; best2 = v2;
                    bestcost = cost;
                }
            }
        }
        // patch the cycles together in tour
        if (best1 == 0) {
            partialTour = true;
        } else {
            stats.tourWeight += bestcost;
            tour.join(best1, graf.head(tEdge[best2]));
        }
        cycRef.removeFirst();
    }
    if (partialTour) {
        cerr << "karp: failed to complete tour\n";
    }
    stats.totalTime = Util::getTime() - t0;
}
```

2. (20 points) Demonstrate the operation of your code by typing the following commands and pasting the results below.

```
% checkKarp <wdg5
2 3 132 159 330 343: {[a e b c d]}
% checkKarp <wdg10
3 6 326 335 73 85: {[a d e c f j h b i g]}
% checkKarp <wdg50
4 24 65524 66663 5363 5767: {[1 10 32 17 50 5 44 31 14 26 8 29 16 20 27 41
24 49 46 13 43 25 18 19 15 22 47 6 11 38 39 23 4 33 9 34 28 3 35 42 2 12
45 30 21 40 37 36 7 48]}
```

Part D. Evaluating running time.

3. (10 points) Run the provided *script1* and use the resulting data to complete the table below. Show the units. For each performance metric, compute the ratios of the values from one row to the next.

		matching time		*total time*	
n	*m*	*time*	*ratio*	*time*	*ratio*
50	1K	2.28ms		2.46ms	
100	3K	11.9ms	5.2	12.8ms	5.2
200	9K	74ms	6.2	82ms	6.4
400	27K	466ms	6.3	533ms	6.5
800	81K	3.1sec	6.7	3.6sec	6.8

4. (10 points) Give an expression for the worst-case running time for the matching algorithm. Based on this, by how much would you expect the running time to increase from one row in the table, to the next. Using this algorithm, what is the largest TSP instance you could solve if you were limited to one hour of compute time (assume that both vertices and edges continue to scale up as in the table)? You need not be precise. Just give rough bounds on the problem size. What if you had 24 hours of compute time? What if you had a week?

   The algorithm finds a min weight maximum size matching in O(mn log n) time. In these experiments, n doubles from one row to the next, while m grows by a factor of 3. So, we should expect the run time to grow by a little more than a factor of 6 with each step. The measured data is consistent with this.

   An hour is 3600 seconds and the total runtime for the largest case above is 3.6 seconds. So the ratio between the two is 1000. This is more than 6^3 and less than 6^4=1296, so we could expect to get somewhere in the range n=6400, m=2.2M to n=12,800, m=6.5M.

   24 hours is 24,000 times longer than the 3.6 seconds above. Since 6^5<24,000<6^6, we can expect to get to somewhere in the range n=26K, m=19M to n=51K, m=57M.

   A week gives us another factor of 7 in the runtime, so it allows us to roughly double the number of vertices and triple the number of edges. So could expect to get in the range n=51K, m=57M to n=102K, m=170M.

Part E. Evaluating solution quality.

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

n	*m*	*num cycles*	*max cycle length*	*matching weight*	*tour weight*
50	1K	3.7	28	7.6K	8.5K
100	3K	3.9	67	32K	35K
200	9K	5.6	118	127K	140K
400	27K	5.6	261	519K	556K
800	81K	6.4	524	2.05M	2.14M

2. (10 points) Give an expression for the maximum number of cycles that could be defined by the matching. Compare this to the number observed. How is the number of cycles increasing with n? If we ran the algorithm on an instance with n=64,000, m=2.2M, how many cycles would you expect?

   The maximum number of cycles is n/2, which is much larger than what we observe. In these experiments, n grows by a factor of 16, while the average number of cycles grows from 3.7 to 6.4, a ratio of just 1.7.

   *Let's assume that the number of cycles grows by a factor of 1.7 every time n grows by a factor of 16. Then if n grows by a factor of 256, the cycle count grows by a factor of 2.9, which we'll round to 3. Since 200*16*16<64,000<400*16*16, we can estimate that for n=64,000 the cycle length will be in the neighborhood of 5.6*3 or about 17.*

3. (10 points) Compare the maximum cycle length to n. Discuss how the number of cycles and the maximum cycle length affect the quality of the approximate TSP tour computed by the algorithm.

   For the smallest problem instances, the maximum cycle length is about .56n. For the largest, it is about .66n.

   If the number of cycles is k, the number of patching operations required to produce a TSP tour is k–1. Since each patching operation increases the cost, it's important to minimize the total number. The maximum cycle length is probably less important than the number of cycles. The main advantage of a large cycle is that it provides more choices for the patching operations than a shorter cycle. Consequently, the larger the cycle, the more likely it is that we'll find an inexpensive place to patch.

4. (20 points) Compare the average weight of the TSP tour to the average weight of the matching used to define the initial set of cycles. What does this tell you about how close the computed tours are to the optimal tour value? Give an example of a TSP instance for which the patching algorithm's performance is extremely poor and show that it is (note, this requires showing that there is a low-cost tour that Karp's algorithm does not find).

   For the smallest problem size, the ratio of the TSP tour weight to the matching weight is about 1.11. For the largest problem size, the ratio is about 1.04. In each case, this ratio provides a bound on the solution quality, since the matching weight can be no larger than the weight of an optimum tour.

   The fact that the ratio is getting smaller suggests that as the problem instances become larger, the performance will only improve.

Consider the graph with vertices a, b, c, d, e, f. Edges ab, bc, cd, de, ef and fa all have cost 2. Edges ac, ce, ea, bf, fd and db all have cost 1. All other pairs are joined by edges with a cost of 1 million. Note that the first part of Karp's algorithm finds the two cycles ace and bfd that each have a cost of 3, for a total of 6. The tour abcdef has a cost of 12 and is optimal. But there is no way to patch the two cycles found by Karp's algorithm without using one of the expensive edges. So the resulting tour has a cost that is much larger that of the optimum tour.