```
/**
 * \file graph.c
 *
 *
 *
 * \author jtd7
 * \date 20-10-2011
 */

#define _POSIX_C_SOURCE 200112L

#include "graph.h"
#include <math.h>
#include <stdio.h>

int numEdges = 0;

/* creates a new point structure */
struct points* newPoints(float *xs, float*ys, int np){
        struct points *p = malloc(sizeof(struct points));
        p->xs = xs;
        p->ys = ys;
        p->num_points = np;
        return p;
}


/* computes the distance between two points */
float distance(struct points* p, int i, int j){
        float x1 = p->xs[i];
        float x2 = p->xs[j];
        float y1 = p->ys[i];
        float y2 = p->ys[j];
        return sqrt(pow((x1 - x2),2) + pow((y1 - y2),2));
}

/* constructs a graph from a set of points.
   once we have the graph, we don't need the points,
   so maybe I can free it here? */
struct graph* newGraph(struct points* p, double max_distance){
        struct graph * g = malloc(sizeof(struct graph));
        struct edges ** edgeList;

        if(p->num_points == 0){
                printf("0.0");
                exit(0);
        }

        /* probably want to fill in your graph fields */
        edgeList = makeEdges(p, max_distance, p->num_points);
        g->edgeList = edgeList;
        g->numEdges = numEdges;
        if(numEdges == 0){
                printf("0.0");
                exit(0);
        }
        g->vertexList = p;

        return g;
}

/* adds an edge (weight or cost, starting vertex, and ending vertex)
   to the set of all edges, returns the index of the next
   edge, or the number of edges once the last edge has been added */
int addEdge(struct edges ** edges, double cost, int start, int end){
        /* probably want to make an edge and add it here. */
        edges[numEdges] = newEdge(cost, start, end);
        numEdges++;
```



```
        return 0;
}

struct edges * newEdge(double cost, int start, int end){
        struct edges * edge = malloc(sizeof(struct edges));
        if(!edge){
                perror("malloc");
                exit(1);
        }
        edge->cost = cost;
        edge->start = start;
        edge->end = end;
        return edge;
}

struct vertex * newVert(int vertex, int setId){
        struct vertex * vert = malloc(sizeof(struct vertex));
        if(!vert){
                perror("malloc");
                exit(1);
        }
        vert->vertex = vertex;
        vert->next = NULL;
        vert->setId = setId;
        return vert;
}

struct set * newSet(struct vertex * head){
        struct set * set = malloc(sizeof(struct set));
        if(!set){
                perror("malloc");
                exit(1);
        }
        set->head = head;
        set->numMembers = 1;
        return set;
}

int sameSet(struct vertex * v1, struct vertex * v2){
        return v1->setId == v2->setId;
}

/* always merge v2 into v1 */
void unionSets(struct vertex * v1, struct vertex * v2){
        struct set * s1 = v1->setHead;
        struct set * s2 = v2->setHead;
        /*struct vertex * test1;
        struct vertex * test2;*/
        struct vertex * cur;
        int newMembers = 0;

        /*fprintf(stderr, "\nv1 before\n");
        test1 = v1->setHead->head;
        while(test1){
                fprintf(stderr, "%d ", test1->vertex);
                test1 = test1->next;
        }

        fprintf(stderr, "\nv2 before\n");
        test2 = v2->setHead->head;
        while(test2){
                fprintf(stderr, "%d ", test2->vertex);
                test2 = test2->next;
        }*/


        /* loop to the end of set 1 */
        cur = s1->head;
        while(cur->next){
```

```
                cur = cur->next;
        }

        /* set cur to be s2's head, and loop to set vertex setIds.*/
        cur->next = s2->head;
        cur = cur->next;
        while(cur){
                cur->setId = v1->setId;
                cur->setHead = v1->setHead;
                cur = cur->next;
                newMembers++;
        }


        s1->numMembers += newMembers;

        /*fprintf(stderr, "\nv1 after\n");
        test1 = v1->setHead->head;
        while(test1){
                fprintf(stderr, "%d ", test1->vertex);
                test1 = test1->next;
        }

        fprintf(stderr, "\nv2 after\n");
        test2 = v2->setHead->head;
        while(test2){
                fprintf(stderr, "%d ", test2->vertex);
                test2 = test2->next;
        }*/

}



/* this is code for adding to a vector.  I'm using it to pair down on the
   number of points that need compared in order to construct the set of
   all edges. You're more than welcome to change this, but you shouldn't
   need to. */
void add(struct vect* v, int i){
        static int next_size = 0;
        if (v->index >= v->num_elms){
                next_size = ceil(v->num_elms * ar_scale);
                v->elms = realloc(v->elms, sizeof(int) * next_size);
                if(v->elms == 0) exit(EXIT_FAILURE);
                v->num_elms = next_size;
        }
        v->elms[v->index] = i;
        v->index++;
}

/* this constructs a 2-d array of vectors for containing points.  Basically
   I'm splitting up the entire graph into sectors s.t. there can only be edges
   between adjacent sectors.  This converts then n^2 cost of making all edges
   into something much more palatable */
struct vect **make_buckets(struct points *p,
                           double max_dist,
                           int num_buckets){
        int x = 0;
        int y = 0;
        int i = 0;
        struct vect **to_ret = malloc(sizeof(struct vect*) *
                                      num_buckets);
        for(i = 0; i < num_buckets; i++){
                to_ret[i] = malloc(sizeof(struct vect) * num_buckets);
                for(x = 0; x < num_buckets; x++){
                        to_ret[i][x].num_elms = 1;
                        to_ret[i][x].elms = malloc(sizeof(int));
                        if(to_ret[i][x].elms == 0) exit(EXIT_FAILURE);
                        to_ret[i][x].index = 0;
```



```
                }
        }

        for(i = 0; i < p->num_points; i++){
                x = (int)(p->xs[i] / max_dist);
                y = (int)(p->ys[i] / max_dist);
                if (x == num_buckets) x -= 1;
                if (y == num_buckets) y -= 1;
                add(&to_ret[x][y], i);
        }
        for(x = 0; x < num_buckets; x++)
                for(y = 0; y < num_buckets; y++){
                        /* I probably over-alloced for my buckets, so
                           now I'm resizing them to save memory */
                        to_ret[x][y].elms = realloc(to_ret[x][y].elms,
                                                    to_ret[x][y].index *
                                                    sizeof(int));
                        to_ret[x][y].num_elms = to_ret[x][y].index;
                }
        return to_ret;
}

/* This builds all of the edges in the graph by calling addEdge a bunch of
   times. It's an unholy mess because I refused to use c99, so I'm declaring a
   boatload of values up front.  In short, it iterates over the buckets created
   by the preceeding function, calling add edge on every valid edge in the
   graph. The horribly nested for loops save you a bunch of work and let you
   tackle truly huge graphs in reasonable times. */
struct edges ** makeEdges(struct points* p, double max_dist, int num_points){
        int maxEdges = (num_points * (num_points - 1)) / 2;
        struct edges ** to_ret = malloc(maxEdges * sizeof(struct edges *));
        int x = 0, y = 0, dx = 0, dy = 0, xp = 0, yp = 0,
            i = 0, j = 0, p1 = 0, p2 = 0, num_buckets = ceil(1 / max_dist);
        double d = 0;
        struct vect **buckets = make_buckets(p,max_dist,num_buckets);
        for(x = 0; x < num_buckets; x++){
           for(y = 0; y < num_buckets; y++){  /* for each grid cell */
              for(dx = 0; dx <= 1; dx++){
                 for(dy = 0; dy <= 1; dy++){  /* for each neighbor */
                    xp = x + dx;
                    yp = y + dy;
                    if(xp < num_buckets && yp < num_buckets){
                      for(i = 0; i < buckets[x][y].num_elms; i++){
                         for(j = 0; j < buckets[xp][yp].num_elms; j++){
                            /* this next one cuts down the points to compare
                               incase we're calculating edges in a region */
                            if(xp != x || yp != y || (j > i)){
                              p1 = buckets[x][y].elms[i];
                              p2 = buckets[xp][yp].elms[j];
                              d = distance(p,p1,p2);
                              if(d <= max_dist && p1 != p2){
                                      /* this is the important bit */
                                      addEdge(to_ret,d,p1,p2);
                              }
                            }
                         }
                      }
                    }
                 }
              }
           }
        }
        /* free the buckets, you don't need them */
        for(i = 0; i < num_buckets; i++){
                for(j = 0; j < num_buckets; j++)
                        free(buckets[i][j].elms);
                free(buckets[i]);
        }
        free(buckets);
```

```
        /* depending on what you do in add edge, you may want to do
           some additional work on to_ret */
        return to_ret;
}
```

```
/**
 * heap.c - Modified over from assignment 2
 *
 * Author: Stephen Chambers
 */
#include "graph.h"
#include "heap.h"
#include <stdlib.h>
#include <stdio.h>


struct edges * pop(struct edges ** edges, unsigned int * n){
        struct edges * e;
        e = edges[0];
        edges[0] = edges[*n-1];
        edges[*n-1] = NULL;
        *n = *n - 1;
        pushdown(edges, 0, *n);
        return e;
}

void pushdown(struct edges ** edges, unsigned int index, unsigned int n){
        unsigned int lIndex, rIndex, min = 0;
        struct edges * temp;

        lIndex = getl(index);
        rIndex = getr(index);

        /*fprintf(stderr, "index: %d\nl: %d\nr: %d\n", index, lIndex, rIndex);
        fprintf(stderr, "edges[0]: [%d,%d] %f\n", edges[0]->start, edges[0]->end
, edges[0]->cost);
        fprintf(stderr, "edges[1]: [%d,%d] %f\n", edges[1]->start, edges[1]->end
, edges[1]->cost);
        fprintf(stderr, "edges[2]: [%d,%d] %f\n", edges[2]->start, edges[2]->end
, edges[2]->cost);

        fprintf(stderr, "\nedges[2]: [%d,%d] %f\n", edges[2]->start, edges[2]->e
nd, edges[2]->cost);
        fprintf(stderr, "edges[5]: [%d,%d] %f\n", edges[5]->start, edges[5]->end
, edges[5]->cost);
        fprintf(stderr, "edges[6]: [%d,%d] %f\n", edges[6]->start, edges[6]->end
, edges[6]->cost);*/



        /* stop if my children go off the array
         * If they don't, find the smallest index */
        if((lIndex >= n && rIndex >= n)){
                return;
        }
        else if(lIndex >= n){
                if(edges[index]->cost < edges[rIndex]->cost)
                        min = index;
                else
                        min = rIndex;
        }
        else if(rIndex >= n){
                if(edges[index]->cost < edges[lIndex]->cost)
                        min = index;
                else
                        min = lIndex;
        }
        else{
                min = get_smallest(edges, index, lIndex, rIndex);
        }
        /* swap if a child is smaller than index */
        if ( min != index ){
                temp = edges[index];
                edges[index] = edges[min];
```



```
                edges[min] = temp;
                pushdown(edges, min, n);
        }
}

unsigned int get_smallest(struct edges ** edges, unsigned int a,
                unsigned int b, unsigned int c){
        if(edges[a]->cost <= edges[b]->cost && edges[a]->cost <= edges[c]->cost)
{
                return a;
        }
        else if(edges[b]->cost <= edges[a]->cost && edges[b]->cost <= edges[c]->
cost){
                return b;
        }
        else if(edges[c]->cost <= edges[b]->cost && edges[c]->cost <= edges[a]->
cost){
                return c;
        }

        /* theoretically unreachable */
        return 0;
}


unsigned int getp(unsigned int index){
        return (index - 1) / 2;
}

unsigned int getl(unsigned int index){
        return (index * 2) + 1;
}

unsigned int getr(unsigned int index){
        return (index * 2) + 2;
}

void print_heap(struct edges ** edges, unsigned int n){
        unsigned int i, l, r = 0;
        fprintf(stderr, "***************PRINTING HEAP!************\n");
        for(i = 0; i < n; i++){
                l = getl(i);
                r = getr(i);
                fprintf(stderr, "parent: [%d, %d] %f\n", edges[i]->start, edges[i]->e
nd, edges[i]->cost);
                if(l < n)
                        fprintf(stderr, "left child: [%d, %d] %f\n", edges[l]->start, ed
ges[l]->end, edges[l]->cost);
                if(r < n)
                        fprintf(stderr, "right child: [%d, %d] %f\n", edges[r]->start, e
dges[r]->end, edges[r]->cost);
        }
}
```

```
/**
 * \file spanning_tree.c
 *
 *
 *
 * \author jtd7
 * \date 20-10-2011
 */

#define _POSIX_C_SOURCE 200112L
#include <assert.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
#include "graph.h"
#include "heap.h"

void usage(){
        fprintf(stderr, "Usage:\ntree <infile> <max_distance>\n");
        exit(EXIT_FAILURE);
}

int solve(struct graph * g){
        unsigned int n = g->numEdges;
        int mid = (n/2) - 1;
        int i = 0;
        struct edges * cur;
        struct edges ** spt;
        struct vertex ** verts;
        int edgeIndex = 0;
        double sptSum = 0.0;

        /* initialize final spanning tree */
        spt = malloc((g->vertexList->num_points - 1) * sizeof(struct edges*));

        /* initialize vertex array */
        verts = malloc(g->vertexList->num_points * sizeof(struct vertex));

        /* make a set for every vertex v */
        for(i = 0; i < g->vertexList->num_points; i++){
                struct vertex * v = newVert(i, i);
                struct set * s = newSet(v);
                v->setHead = s;
                verts[i] = v;
        }

        /* convert edge list to a heap using floyd's algorithm */
        for(i = mid; i >= 0; i--){
                pushdown(g->edgeList, i, n);
        }

        /* pop off edge heap until empty */
        while(n != 0){
                cur = pop(g->edgeList, &n);

                /*print_heap(g->edgeList, g->numEdges);*/

                if(!sameSet(verts[cur->start], verts[cur->end])){
                        /* add edge to T */
                        spt[edgeIndex] = cur;
                        edgeIndex++;

                        /* union two sets */
                        if(verts[cur->start]->setHead->numMembers < verts[cur->e
nd]->setHead->numMembers){
                                /* merge start into end */
                                unionSets(verts[cur->end], verts[cur->start]);
                        }
                        else{
```



```
                                /* merge end into start */
                                unionSets(verts[cur->start], verts[cur->end]);
                        }
                }
        }

        for(i = 0; i < (g->vertexList->num_points - 1); i++){
                sptSum += spt[i]->cost;
                printf("%d %d\n", spt[i]->start, spt[i]->end);
        }
        printf("%f\n", sptSum);

        return 0;
}

int process(const char* infile, double max_distance){
        float *x_array;
        float *y_array;
        int num_points = -1;
        FILE *ifile = fopen(infile, "r");
        char buf[256];
        char c = '\0';
        int b_index = 0;
        int p_index = 0;
        float x = 0.;
        float y = 0.;
        int on_x = 1;
        struct points* p;
        struct graph* g;
        int err = 0;

        if(ifile == NULL){ /* fopen failed */
                fprintf(stderr, "Failed to open %s\n", infile);
                return EXIT_FAILURE;
        }else{  /* get input size */
                while((c = getc(ifile)) != '\n'){
                        buf[b_index] = c;
                        b_index++;
                }
                num_points = atoi(buf);
                x_array = malloc(sizeof(float) * num_points);
                y_array = malloc(sizeof(float) * num_points);
                b_index = 0;
                while((c = getc(ifile)) != EOF){
                        buf[b_index] = c;
                        if(c == '\n' || c == ' '){
                                if (on_x){
                                        x = atof(buf);
                                        on_x = 0;
                                }else{
                                        y = atof(buf);
                                        x_array[p_index] = x;
                                        y_array[p_index] = y;
                                        p_index ++;
                                        on_x = 1;
                                }
                                b_index = 0;
                        }else b_index++;
                }
        }

        p = newPoints(x_array, y_array, num_points);
        g = newGraph(p, max_distance);

        /* solve the graph */
        err = solve(g);
        /* cleanup here */
        fclose(ifile);
        return err;
```

```
}

int main(int argc, const char *const argv[]){
        double d;
        if (argc != 3)  usage();
        d = atof(argv[2]);
        return process(argv[1], d);
}
```

```
/**
 * \file graph.h
 *
 *
 *
 * \author jtd7
 * \date 20-10-2011
 */

#ifndef GRAPH_H
#define GRAPH_H

#include <stdlib.h>
#include <math.h>

/* I do a lot of dynamic resizing arrays in my code.
   I used 1.5 instead of 2 to avoid overshooting my end memory too much. */
const static float ar_scale = 1.5;

/* rather than storing an array of tuples, I store two arrays,
   one for x, and one for y. */
struct points {
        float *xs;
        float *ys;
        int num_points;
};

struct edges {
/* your fields here */
        double cost;
        int start;
        int end;
};


struct graph {
        /* your fields here */
        struct points * vertexList;
        struct edges ** edgeList;
        int numEdges;
};

struct vertex {
        int vertex;
        int setId;
        struct set * setHead;
        struct vertex * next;
};

struct set {
        struct vertex * head;
        int numMembers;
};

/* an auxilliary data structure that has nothing to do with your code,
   used for making all of the valid edges efficiently */
struct vect {
        int* elms;
        int num_elms;
        int index;
};

struct points* newPoints(float *xs, float *ys, int np);
struct edges** makeEdges(struct points *p, double max_dist, int num_points);
struct graph* newGraph(struct points* p, double max_dist);
int addEdge(struct edges ** e, double cost, int start, int end);
struct edges * newEdge(double cost, int start, int end);
struct set * newSet(struct vertex * head);
struct vertex * newVert(int vertex, int setId);
```



```
int sameSet(struct vertex * v1, struct vertex * v2);
void unionSets(struct vertex * v1, struct vertex * v2);

#endif
```

```
/**
 * heap.h- header file for heap
 *
 * Author: Stephen Chambers
 */

#ifndef HEAP_H
#define HEAP_H

#include "graph.h"

void pushdown(struct edges **, unsigned int, unsigned int);
unsigned int get_smallest(struct edges **, unsigned int, unsigned int, unsigned
int);
unsigned int getp(unsigned int);
unsigned int getl(unsigned int);
unsigned int getr(unsigned int);
void print_heap(struct edges ** edges, unsigned int n);
struct edges * pop(struct edges **, unsigned int *);

#endif
```

```
CC=gcc
CFLAGS=-lm -ansi -pedantic -Wall -Werror

SRC=    spanning_tree.c \
        graph.c heap.c

all: tree tree_debug

tree: $(SRC)
        $(CC) $(CFLAGS) -DNDEBUG -O3 $^ -o tree


tree_debug: $(SRC)
        $(CC) $(CFLAGS) -g $^ -o tree_debug

clean:
        rm -f tree tree_debug
        rm -f *.o
```