A Program that Finds a Minimum Eccentricity Spanning Tree

David W. Krumme

Technical Report 99-2
November 1999

1 Introduction

A minimum-eccentricity spanning tree is a spanning tree of a weighted, undirected graph $G = (V, E)$ which minimizes the worst-case weighted distance between any source and any sink, where the sources and sinks are given subsets of V. An algorithm that solves this problem is described in a companion publication [1]. This report presents a C program that implements the algorithm.

2 Summary of the program

The program is presented on the following pages. It is also available as a file through `ftp:www.eecs.tufts.edu/tr` or `http:www.eecs.tufts.edu/tr`. Like the algorithm, the program finds one point which determines a minimum-eccentricity spanning tree. It does not attempt to find all such points. It does not attempt to actually describe the resultant tree because moving from the point to the tree can be done by a simple breadth-first search.

The published algorithm has steps 1 through 5. Steps 1 and 2 are performed by the `all_dist` procedure in the program which implements Dijkstra's algorithm. Step 3 is performed by the `build_h` procedure. Step 4 corresponds to the initialization of the variable `gbest`. Step 5.1 is performed by the `build_AB` procedure. Steps 5.2 and 5.3 are performed by the `find` procedure. After that, at

the end of `main`, the program prints out the result by describing where to center a minimum-distance spanning tree to obtain a minimum-eccentricity spanning tree. It also prints out the radius of the graph.

There is one fine point not covered in [1]. The algorithm naturally produces a range of points where edge it chooses can be subdivided, all of which yield one and the same tree. The `remember` procedure uses the `A` and `B` arrays to determine that range, and the mainline reports the result.

References

[1] D. W. Krumme and P. Fragopoulou. "Minimum Eccentricity Multicast Trees." Submitted to *Journal of Interconnection Networks*.

The program

```
/* Program "mest" finds a minimum-eccentricity spanning tree.
 * Author:  David W. Krumme
 * Usage:
 *      mest inputgraph
 *      mest - inputgraph
 *      mest -- inputgraph
 * The "-" options evoke extra printouts.
 * To compile with gcc:  gcc mest.c -o mest
 *
 * The input graph, in standard input or a file named on the command line,
 * is assumed to be in this form:
 *      The edges and their weights, each on a line
 *      then a blank line
 *      then the members of S1, one per line
 *      then a blank line
 *      then the members of S2, one per line
 * Arbitrary ascii names are used for vertices.
 * Comment lines are initiated with #
 * Here is a four-cycle, all weights 10, two antipodes in S1 and two in S2:
 *      NW NE 10
 *      NE SE 10
 *      SE SW 10
 *      SW NW 10
 *
 *      NW
 *      SE
 *
 *      NE
 *      SW
 */
#define MAXNODES 63
#define MAXNAME 39
#include <stdio.h>
#include <strings.h>
int verbose = 0, veryverbose = 0;
```

```
//////////////////////////////
//      Arrays used          //
//////////////////////////////

char name[MAXNODES+1][MAXNAME+1]; // name[x] is the print-name of node x

int wt[MAXNODES+1][MAXNODES+1]; // wt[x][y] = weight (x,y), 0 if no edge
int maxpt;                      // vertices are numbered 1..maxpt

int S[3][MAXNODES+1];           // Sources and sinks: S1 = S[1], S2 = S[2]
int N[3];                       // N[1] = number of sources, N[2] no. of sinks
int inS[3][MAXNODES+1];         // Characteristic function for S1 and S2

int d[MAXNODES+1][MAXNODES+1];  // d[x][y] = weighted distance from v to w

// For each i and x, { h[i][x][1..maxpt] } = S[i]
//   and d[x][v] <= d[x][w] if v = h[i][x][k] and w = h[i][x][k+1] for some k
int h[3][MAXNODES+1][MAXNODES+1]; // vertex list ordered by distance

int A[3][MAXNODES];     // A and B are the special arrays
int B[3][MAXNODES];     // (described separately).
int NAB[3];             // size of corresponding A and B array
```

```
//////////////////////////////
//  Calculate A[] and B[]     //
//////////////////////////////

void build_AB(int i, int u, int w) {
    int k, p, v; // v moves through members of S[i] by decreasing d[v][W]
    A[i][0] = d[ h[i][w][N[i]] ][ w ] - wt[u][w];
    B[i][0] = d[ h[i][w][N[i]] ][ w ] + 1;      // fake entry, fixed later
    k = 0;
    A[i][1] = A[i][0];
    for (p = N[i]; p >= 1; p--) {
        v = h[i][w][p];
        if (d[v][u] > A[i][k+1]) {
            if (d[v][w] < B[i][k]) B[i][++k] = d[v][w];
            A[i][k+1] = d[v][u];
        }
    }
    B[i][0] = d[ h[i][u][N[i]] ][ u ] - wt[u][w];
    if (k==0 || B[i][k] > B[i][0]) B[i][++k] = B[i][0];
    NAB[i] = k;
}

void print_AB(int x, int y) {
  int i,k;
  for (i=1; i<=2; i++) {
    printf("(%s,%s) AB[%d] =", name[x],name[y],i);
    for (k=1; k<=NAB[i]; k++) printf(" (%d,%d)", A[i][k], B[i][k]);
    printf("\n");
  }
}

void print_D(int x, int y) {
  int i,j;
  for (i=1; i<=2; i++) {
    printf("(%s,%s) D[%d] =", name[x],name[y],i);
    for (j=1; j<=N[i]; j++) {
            printf(" (%d,%d)", d[x][S[i][j]], d[y][S[i][j]]);
    }
    printf("\n");
  }
}
```

```
//////////////////////////////
//  Find the minimum value    //
//////////////////////////////

int gbest=((unsigned)(1<<31)-1), gx, gy, lowerlim, upperlim;
void remember(int newbest, int x, int y, int k, int j) {
        int alt1, alt2;
        gbest = newbest; gx = x; gy = y;
        if (k==1) alt1 = -wt[x][y];
        else alt1 = A[1][k] - B[1][k-1];
        if (j==1) alt2 = -wt[x][y];
        else alt2 = A[2][j] - B[2][j-1];
        if (alt1 > alt2) lowerlim = alt1; else lowerlim = alt2;
        if (k==NAB[1]) alt1 = wt[x][y];
        else alt1 = A[1][k+1] - B[1][k];
        if (j==NAB[2]) alt2 = wt[x][y];
        else alt2 = A[2][j+1] - B[2][j];
        if (alt1 < alt2) upperlim = alt1; else upperlim = alt2;
}

void find(int x, int y) {
        int k, j, cc, dd;
        k = 1; j = 1;
        while (k <= NAB[1] && j <= NAB[2]) {
            cc = A[1][k] + wt[x][y] + B[2][j];
            dd = A[2][j] + wt[x][y] + B[1][k];
            if (cc > dd) {
                if (verbose) {
                  printf("A[1][%d]+w(%d,%d)+B[2][%d] = %d + %d + %d = %d\n",
                         k, x,y, j, A[1][k], wt[x][y], B[2][j], cc);
                }
                if (cc < gbest) remember(cc, x, y, k, j);
                j++;
            } else {
                if (verbose) {
                  printf("A[2][%d]+w(%d,%d)+B[1][%d] = %d + %d + %d = %d\n",
                         j, x,y, k, A[2][j], wt[x][y], B[1][k], dd);
                }
                if (dd < gbest) remember(dd, x, y, k, j);
                k++;
                if (cc == dd) j++;
            }
        }
}
```

```
//////////////////////////////
//      Misc. support         //
//////////////////////////////

void build_h() {        // create h[1] and h[2] arrays as subsets of h[0]
        int i, j, k1, k2, v;
   for (i=1; i<=maxpt; i++) {
      k1 = 0; k2 = 0;
      for (j=1; j<=maxpt; j++) {
        v = h[0][i][j];
        if (inS[1][v]) h[1][i][++k1] = v;
        if (inS[2][v]) h[2][i][++k2] = v;
}  }  }

void all_dist() {       // Dijkstra alg.
    int i,j,k,v, bestd, old, new;
    char taken[MAXNODES+1];
    for (i=1; i<=maxpt; i++) {
        for (j=1; j<=maxpt; j++) { d[i][j] = 999999; taken[j] = 0; }
        d[i][i] = 0; taken[i] = 1;
        v = i; k = 1; h[0][i][k] = i;
        while (1) {
            for (j=1; j<=maxpt; j++) if (wt[v][j]) {
                old = d[i][j];
                new = d[i][v] + wt[v][j];
                if (new < old) d[i][j] = new;
            }
            bestd = 999999;
            for (j=1; j<=maxpt; j++) if (!taken[j]) {
                if (d[i][j] < bestd) { bestd = d[i][j]; v = j; }
            }
            if (bestd == 999999) break;
            h[0][i][++k] = v;
            taken[v] = 1;
}   }   }

int lookup(char *n) {   // Map vertex name to index number
    for (int x=1; x<=MAXNODES; x++) {
        if (name[x][0]==0) strcpy(name[x], n);
        if (strcmp(name[x], n) == 0) return x;
    }
    fprintf(stderr, "Too many vertices!\n");
    return exit(1);
}
```

```
//////////////////////////////
//         Input            //
//////////////////////////////

void read_graph(FILE *f) {      // Input the edges
    int x,y,w; char *px,*py,*pw;
    char line[200];
    while (1) {
        if (!fgets(line, 200, f)){fprintf(stderr, "premature EOF\n");exit(1);}
        if (px = strtok(line, " \n")) {
            if (*px == '#') continue;
            if ( (py = strtok(NULL, " \n"))
              && (pw = strtok(NULL, " \n"))) {
                x = lookup(px); y = lookup(py); w = atoi(pw);
                if (w <= 0) {fprintf(stderr, "Negative weight!\n"); exit(1);}
                wt[x][y] = w;
                wt[y][x] = w;
                if (y > maxpt) maxpt = y;
                if (x > maxpt) maxpt = x;
            } else return;
        } else return;
    }
}

void read_source(FILE *f, int which) {  // Input the source and sink lists
    int x; char *px;
    char line[200];
    while (1) {
        if (!fgets(line, 200, f)){
            if (which == 2) return;
            fprintf(stderr, "%d premature EOF\n", which);exit(1);
        }
        px = strtok(line, " \n");
        if (px && *px) {
            x = lookup(px);
            if (1 <= x && x <= maxpt) {
                S[which][++N[which]] = x;
                inS[which][x] = 1;
            } else { fprintf(stderr, "Unknown: %s\n", px); exit(1); }
        } else return;
    }
}
```

```
//////////////////////////////
//      Debug support         //
//////////////////////////////
void print_graph() {
    int i,j;
    for (i=1; i<=maxpt; i++) {
        printf("%s:", name[i]);
        for (j=1; j<=maxpt; j++) if (wt[i][j])
                printf(" %s[%d]", name[j],wt[i][j]);
        printf("\n");
    }
}
void print_sources() {
    int i,j;
    for (i=1; i<=2; i++) {
        for (j=1; j<=N[i]; j++) printf("%s ", name[S[i][j]]);
        printf("\n");
    }
}
void print_dist() {
    int i,j,k;
    printf("   ");
    for (j=1; j<=maxpt; j++) printf(" %2d", j);
    printf("\n");
    for (i=1; i<=maxpt; i++) {
        printf("%2d:", i);
        for (j=1; j<=maxpt; j++) printf(" %2d", d[i][j]);
        printf("\n");
    }
}
void print_h() {
    int i,k;
    for (i=1; i<=maxpt; i++) {
        printf("%2d =>", i);
        for (k=1; k<=maxpt; k++) printf(" %d", h[0][i][k]);
        printf("\n");
        printf(" S1 --");
        for (k=1; k<=N[1]; k++) printf(" %d@%d", h[1][i][k], d[i][h[1][i][k]]);
        printf("\n");
        printf(" S2 --");
        for (k=1; k<=N[2]; k++) printf(" %d@%d", h[2][i][k], d[i][h[2][i][k]]);
        printf("\n");
    }
}
```

```
//////////////////////////////
//       Mainline           //
//////////////////////////////
main(int argc, char *argv[]) {
    int x,y;
    FILE *fin;

    if (argc > 1 && argv[1][0]=='-') {
        verbose = 1;
        if (argv[1][1]=='-') veryverbose = 1;
        argc--; argv++;
    }
    fin = NULL;
    if (argc > 1) fin = fopen(argv[1], "r");
    if (fin == NULL) fin = stdin;
    read_graph(fin);                    /* print_graph(); // for debugging */
    read_source(fin, 1);
    read_source(fin, 2);                /* print_sources(); // debug */

    // Phase one (does not use S)
    all_dist();                         /* print_dist(); // debug */

    // Phase two
    // customize the h list to S[1] and S[2]
    build_h();                          /* print_h(); // debug */
    // check each edge
    for (x=1; x<=maxpt; x++) for (y=x+1; y<=maxpt; y++) if (wt[x][y]) {
        if (veryverbose) print_D(x, y);
        // construct A and B arrays for (x,y)
        build_AB(1, x, y); build_AB(2, x, y);
        if (verbose && NAB[1] >= 1 && NAB[2] >= 1) print_AB(x,y);
        // optimize over edges and AB points
        find(x,y);
    }
```

```
    // print answer
    { char *phrase = "Divide";
      if (upperlim == wt[gx][gy]) {
        printf("Set c = vertex %s.\n", name[gx]);
        phrase = "Or divide";
      }
      if (lowerlim == -wt[gx][gy]) {
        if (phrase[0] == 'O') phrase = "Or set"; else phrase = "Set";
        printf("%s c = vertex %s.\n", phrase, name[gy]);
        phrase = "Or divide";
      }
      if (lowerlim < upperlim) {
        printf("%s (%s,%s) into (%s,c) and (c,%s),\n",
                phrase, name[gx], name[gy], name[gx], name[gy]);
        printf("   with wt(%s,c) = W and wt(c,%s) = %d-W,\n",
                name[gx], name[gy], wt[gx][gy]);
        printf("      where W can be any value in %.1f < W < %.1f.\n",
                0.5*(wt[gx][gy]-upperlim),
                0.5*(wt[gx][gy]-lowerlim));
      } else { // the following probably cannot happen
        printf("%s (%s,%s) into (%s,c) and (c,%s),\n",
                phrase, name[gx], name[gy], name[gx], name[gy]);
        printf("   with wt(%s,c) = %.1f.\n",
                name[gx], 0.5*(wt[gx][gy]-upperlim));
      }
      printf("Minimum-weight spanning tree to be centered at c.\n");
      printf("Eccentricity = %d.\n", gbest);
  }
    // calculate radius, just for curiosity, will be <= eccentricity/2
  { int radius = -1, r, center;
    for (x=1; x<=maxpt; x++) {
        r = 0;
        for (y=1; y<=maxpt; y++) if (d[x][y] > r) r = d[x][y];
        if (radius < 0 || r < radius) { radius = r; center = x; }
    }
    printf("radius(G) = %d with center %s.\n", radius, name[center]);
  }
    return(0);
}
```