```
1: //  Solve MLST by MVCA
2:
3: #include <stdio.h>
4: #include <stdlib.h>
5: #include <time.h>
6: #include <math.h>
7: #include <string.h>
8: #include <iostream.h>
9: #include <conio.h>
10:
11: #define VN 500 // maximum number of vertices
12: #define CN 625 // maximum number of colors
13:
14: int *Temporary_Vector;
15: int dim_Temp;
16: int Demo;
17:
18: typedef struct edge_type {
19:     int u; // one vertex
20:     int v; // the other vertex
21:     int c; // color
22: struct edge_type *next; // next edge
23: } edge;
24:
25:
26: typedef struct color_type {
27:     int c; // color
28:    //int freq; // the net frequency of the color
29:     int en; // the number of edges
30:    edge *root; // point to a list of edges
31: } color;
32:
33:
34: class Graph {
35:    public: color L[CN]; // list of colors
36:    int vn; // number of vertices
37:    int cn; // number of colors
38:    int sol_cn; // number of colors
39:    void q_sortGF(int left, int right);  //quick sort
40:    //int backup_cn;
41:    //private: int backup_C[CN];
42:    public: int label[VN]; // from pepe
43:    int NumComps;
44:    public: int ReadNumber(FILE *fp);
45:     void InitGraph(FILE *fp);
46:     void InitEmptyGraph(Graph G);
47:    void PrintResultsToFile(FILE *results);
48:     void PrintGraph(); // Print this graph
49:     void PrintSolution(); // Print the feasible_solution
50:     public: void AddColor(Graph G, int c);
51:     //void RemoveColor(Graph G, int c);
52:    //public: int NumComponents(); // Find the number of connected components of the g
53:     void ClearAll();
54:     void RemoveAllColors();
55:     void my_MVCA(Graph G); // heuristic
56:    void PostOptimization(); // Post Optimization phase
57:    void RemoveUselessColor(int x);
58:     //private: void SortByGeneral(int numb_col); // Sort by the general frequency
59: };
```

```
 60:
 61:
 62: int Graph::ReadNumber(FILE *fp) {
 63:     char buf[10];
 64:     char c;
 65:     int i,n;
 66:     int num;
 67:
 68:    c=fgetc(fp);
 69:     i=0;
 70:     while (!feof(fp) && (c<'0' || c>'9')) c=fgetc(fp);
 71:     while (!feof(fp) && (c>='0' && c<='9')) {
 72:         buf[i]=c;
 73:       c=fgetc(fp);
 74:       i++;
 75:     }
 76:     //buf[i]='\0';
 77:     num=0;
 78:     n=i;
 79:     for (i=0; i<n; i++) {
 80:         num=num*10+(int)(buf[i]-'0');
 81:     }
 82:
 83:    return num;
 84: }
 85:
 86:
 87: void Graph::InitEmptyGraph(Graph G) {
 88:     int i;
 89:
 90:     vn=G.vn;
 91:    cn=G.cn;
 92:    sol_cn=0;
 93:    for (i=0; i<vn; i++) {
 94:         label[i]=i;
 95:     }
 96:    NumComps=vn;
 97:
 98:     for (i=0; i<cn; i++) {
 99:         L[i].en=0;
100:       L[i].c=i;
101:       L[i].root=NULL;
102:    }
103: }
104:
105:
106: void Graph::InitGraph(FILE *fp) {
107:     int i,j,k;
108:     edge *e;
109:
110:    sol_cn=cn;
111:    for (i=0; i<vn; i++) {
112:         label[i]=i;
113:    }
114:     for (i=0; i<cn; i++) {
115:         L[i].en=0;
116:       L[i].c=i;
117:       L[i].root=NULL;
118:    }
```

```
119:                     // Add edges in the empty graph:
120:     for (i=0; i<vn-1; i++) {
121:         for (j=i+1; j<vn; j++) {
122:             //if (i==j) continue;
123:             k=ReadNumber(fp);
124:             if (k==cn) continue;
125:                     // Add color k for edge (i,j):
126:             e=new edge;
127:             e->u=i;
128:          e->v=j;
129:          e->c=k;
130:          e->next=L[k].root;
131:          L[k].root=e;
132:          L[k].en++;
133:         }
134:     }
135:
136:    if (Demo==1){
137:     printf("\ncolor(frequency): ");
138:         for (i=0; i<cn; i++) printf("%d(%d) ",L[i].c,L[i].en);
139:         printf("\n");
140:     getch();
141:    }
142:
143:    //q_sortGF(0, cn - 1);
144:
145:    if (Demo==1){
146:     printf("\n-- SORTED -- color(frequency): ");
147:         for (i=0; i<cn; i++) printf("%d(%d) ",L[i].c,L[i].en);
148:         printf("\n\n");
149:     getch();
150:    }
151: }
152:
153:
154: void Graph::q_sortGF(int left, int right) {
155:     int pivot, index_pivot, l_hold, r_hold;
156:    color L_support;
157:
158:    l_hold = left;
159:     r_hold = right;
160:    index_pivot=right;
161:    L_support=L[index_pivot];
162:    pivot = L[index_pivot].en;
163:    while (left < right){
164:     while ((L[left].en >= pivot) && (left < right)) left++;
165:         if (right != left){
166:          L[right] = L[left];
167:          right--;
168:         }
169:       while ((L[right].en <= pivot) && (left < right))  right--;
170:         if (right != left){
171:         L[left] = L[right];
172:         left++;
173:         }
174:     }
175:    L[right] = L_support;
176:    pivot = right;
177:    left = l_hold;
```

```
178:     right = r_hold;
179:    /*
180:    int k;
181:    if (Demo==1){
182:     printf("\n");
183:     printf("\nSort List: ");
184:     printf("\ncolor(frequency): ");
185:         for (k=0; k<cn; k++) printf("%d(%d) ",L[k].c,L[k].en);
186:         printf("\n\n");
187:     getch();
188:    }
189:    */
190:    if (right > pivot){
191:     q_sortGF(pivot+1, right);
192:    }
193:    if (left < pivot)    {
194:     q_sortGF(left, pivot-1);
195:    }
196: }
197:
198:
199: void Graph::my_MVCA(Graph G) {
200:     int num,i,j,p,k,Add_p;
201:     int flag;
202:     int temp[CN];
203:    edge *e;
204:    int label_app[CN][VN]; // support variable
205:    int Labeli,Labelj;
206:    int app_Num_Comp[VN];
207:
208:    Temporary_Vector=new int[G.cn];
209:
210:    if (Demo==1){
211:     printf("--- MVCA ---\n\n\n");
212:     getch();
213:    }
214:    if (sol_cn!=0) RemoveAllColors();
215:    num=vn;
216:    //Add_p=0;
217:    flag=1;
218:     for (p=0; p<G.cn; p++){
219:       if (G.L[p].root==NULL) temp[p]=1;
220:       else temp[p]=0;
221:       if (Demo==1)  printf("temp[%d]=%d\n",p,temp[p]);
222:    }
223:    if (Demo==1) getch();
224:    while (flag==1) {
225:       dim_Temp=0;
226:     for (p=0; p<G.cn; p++) {
227:          if (Demo==1)   printf("temp[%d]=%d\n",p,temp[p]);
228:             if (temp[p]==1) continue;
229:          for (i=0; i<vn; i++) {
230:             label_app[p][i]=label[i];
231:             if (Demo==1)    printf("label_app[%d][%d]=%d, label[%d]=%d\n",p,i,label_app[p]
232:          }
233:          app_Num_Comp[p]=NumComps;
234:          e=G.L[p].root;
235:          while (e!=NULL) {
236:                 i=e->u;
```

```
237:             j=e->v;
238:             if (Demo==1)    printf("\ni=%d, j=%d\n",i,j);
239:             if (i==j) {
240:                     e=e->next;
241:                 continue;
242:             }
243:             if(label_app[p][i]!=label_app[p][j]){
244:                 app_Num_Comp[p]=app_Num_Comp[p]-1;
245:                 Labeli=label_app[p][i];
246:                 Labelj=label_app[p][j];
247:                 for (k=0; k<vn; k++){   // := 1 to N
248:                     if (label_app[p][k]==Labelj)    label_app[p][k]=Labeli;
249:                 }
250:             }
251:             if (Demo==1){
252:                 printf("e->next\n\n\n");
253:                 getch();
254:             }
255:             e=e->next;
256:         }
257:         if (Demo==1){
258:             getch();
259:             printf("\nOLD TEMP num=%d\n",num);
260:             getch();
261:         }
262:             if (app_Num_Comp[p]<num) {
263:             Temporary_Vector[0]=p;
264:             dim_Temp=1;
265:                 //Add_p=p;
266:             num=app_Num_Comp[p];
267:             if (Demo==1){
268:                 printf("\nNEW TEMP num=%d\n",num);
269:                 getch();
270:             }
271:         }
272:         else{
273:             if ((app_Num_Comp[p]==num)&&(dim_Temp!=0)) {
274:                if (Demo==1) printf("\nTHIS COLOR HAS THE SAME NUMBER OF CONN.COMPS. NEW EN
275:                 Temporary_Vector[dim_Temp]=p;
276:                 dim_Temp++;
277:             }
278:         }
279:         if (Demo==1){
280:             printf("\nTEMPORARY VECTOR: ");
281:             for (i=0; i<dim_Temp; i++) {
282:                 printf("%d ",Temporary_Vector[i]);
283:             }
284:             printf("\nWe start again with another color\n");
285:             getch();
286:         }
287:       }
288:       if (Demo==1){
289:         printf("\n!!!No we scan all the colors: we add the best color\n");
290:         getch();
291:       }
292:       Add_p=Temporary_Vector[rand()%dim_Temp];
293:       AddColor(G,Add_p);
294:       temp[Add_p]=1;
295:       NumComps=num;
```

```
296:       if (Demo==1){
297:         printf("\nPosition of the color added=%d, color added=%d, New num=%d\n",Add_p,G.L[
298:         getch();
299:       }
300:       for (i=0; i<vn; i++) {
301:          label[i]=label_app[Add_p][i];
302:          if (Demo==1)   printf("label[%d]=%d ",i,label[i]);
303:       }
304:       if (NumComps==1) flag=0;
305:       if (Demo==1){
306:         printf("\nflag=%d\n",flag);
307:         getch();
308:       }
309:    }
310: }
311:
312:
313: void Graph::PostOptimization(){
314:     int i,j,p,k;
315:     int flag;
316:     int x;
317:    edge *e;
318:    int Labeli,Labelj;
319:    int rem=0;
320:
321:    if (sol_cn!=0){
322:       if (Demo==1){
323:          printf("\n--------------- SOLUTION ");
324:         printf("\ncolor(frequency): ");
325:             for (i=0; i<sol_cn; i++) printf("%d(%d) ",L[i].c,L[i].en);
326:             printf("\n");
327:         getch();
328:     }
329:       x=sol_cn-2;
330:       //x=0;
331:       if (Demo==1){
332:         printf("\nStart of the Post Optimization Phase\n");
333:          printf("\nWe start with the last color: %d\n",L[x].c);
334:       }
335:     flag=1;
336:       while (flag==1) {
337:             for (i=0; i<vn; i++) {
338:                 label[i]=i;
339:             }
340:         NumComps=vn;
341:          for (p=0; p<sol_cn; p++) {
342:             if (p==x) continue;
343:             e=L[p].root;
344:             while (e!=NULL) {
345:                     i=e->u;
346:                 j=e->v;
347:                 if (i==j) {
348:                         e=e->next;
349:                     continue;
350:                 }
351:                 if(label[i]!=label[j]){
352:                     NumComps=NumComps-1;
353:                     Labeli=label[i];
354:                     Labelj=label[j];
```

```
355:                     for (k=0; k<vn; k++){   // := 1 to N
356:                         if (label[k]==Labelj)   label[k]=Labeli;
357:                     }
358:                 }
359:                 e=e->next;
360:             }
361:         }
362:         if (Demo==1)   printf("\nNumComps=%d\n",NumComps);
363:         if (NumComps==1){
364:             if (Demo==1){
365:                 printf("\n *-*-* IT IS POSSIBLE TO DELETE COLOR %d FROM THE SOLUTION!\n",L
366:                 rem++;
367:             }
368:             RemoveUselessColor(x);
369:         }
370:         else{
371:             if (Demo==1)    printf("\nIT IS NOT POSSIBLE TO DELETE COLOR %d FROM THE SOLUT
372:         }
373:         if (x==0){
374:         //if (x>=sol_cn-2){
375:             flag=0;
376:             if (Demo==1){
377:                 if (rem!=0){
378:                     printf("\n --- Total removed=%d\n",rem);
379:                 PrintSolution();
380:                }
381:             }
382:         }
383:         else x=x-1;
384:         //else x=x+1;
385:         if (Demo==1){
386:             if (flag==1) printf("\nnew x=%d; We start again with another color: %d\n",x,L[
387:             getch();
388:         }
389:       }
390:    }
391:    else{
392:     printf("\n!!! DANGER: THE SOLUTION IS EMPTY !!!\n");
393:         printf("IMPOSSIBLE TO APPLY THE POST OPTIMIZATION PHASE\n");
394:       printf("\nPress a key to continue ...\n");
395:         getch();
396:    }
397: }
398:
399:
400: void Graph::RemoveAllColors() {
401:     int i;
402:
403:    sol_cn=0;
404:     for (i=0; i<cn; i++) {
405:         L[i].c=cn;
406:       L[i].en=0;
407:         L[i].root=NULL;
408:    }
409:    for (i=0; i<vn; i++) {
410:     label[i]=i;
411:    }
412:    NumComps=vn;
413: }
```

```
414:
415:
416: void Graph::AddColor(Graph G, int p) {
417:    L[sol_cn].root=G.L[p].root;
418:    L[sol_cn].c=G.L[p].c;
419:    L[sol_cn].en=G.L[p].en;
420:    sol_cn++;
421: }
422:
423:
424: void Graph::RemoveUselessColor(int x) {
425:    int p;
426:
427:    sol_cn--;
428:    for (p=x; p<sol_cn; p++) {
429:     L[p]=L[p+1];
430:    }
431:    if (Demo==1) PrintSolution();
432: }
433:
434: /*
435: void Graph::RemoveColor(Graph G, int p) {
436:    int k;
437:
438:    if (L[cn-1].c==G.L[p].c){
439:       L[cn-1].root=NULL;
440:         L[cn-1].en=0;
441:       L[cn-1].c=cn; //cn means no color
442:         cn--;
443:    }
444:    else{
445:     printf("\nError in removing! ");
446:       G.ClearAll();
447:         for (k=0; k<cn; k++) {
448:             L[k].c=cn;
449:         L[k].en=0;
450:             L[k].root=NULL;
451:         }
452:     cn=0;
453:       printf("\nPress a key to continue ...");
454:         getch();
455:       clrscr();
456:       return;
457:    }
458: } */
459:
460:
461: void Graph::ClearAll() {
462:     int i;
463:     edge *e,*e1;
464:
465:     for (i=0; i<cn; i++) {
466:         L[i].c=cn;
467:       L[i].en=0;
468:         e=L[i].root;
469:         while (e!=NULL) {
470:             e1=e;
471:          e=e->next;
472:             delete e1;
```

```
473:         }
474:         L[i].root=NULL;
475:     }
476:     cn=0;
477:     sol_cn=0;
478: }
479:
480:
481: void Graph::PrintSolution() {
482:     int p;
483:
484:     printf("Number of colors of the found solution: %d\n",sol_cn);
485:     printf("SOLUTION: ");
486:     for (p=0; p<sol_cn; p++) {
487:         if (L[p].root!=NULL) printf("%d ",L[p].c);
488:     }
489:     printf("\n");
490: }
491:
492:
493: void Graph::PrintGraph() {
494:     int i;
495:     edge *e;
496:
497:     printf("GRAPH ----------------------------------------\n");
498:     printf("position) color(freq): (arcs)\n");
499:     for (i=0; i<sol_cn; i++) {
500:         if (L[i].root==NULL) continue;
501:         printf("%d) %d(%d): ",i,L[i].c,L[i].en);
502:         e=L[i].root;
503:         while (e!=NULL) {
504:             printf("(%d,%d) ",e->u,e->v);
505:             e=e->next;
506:         }
507:         printf("\n");
508:     }
509:     printf("\n");
510:     printf("================================================================================
511:     printf("\n");
512: }
513:
514:
515: void Graph::PrintResultsToFile(FILE *results) {
516:     int i;
517:     int p;
518:     edge *e;
519:
520:     fprintf(results,"Number of colors of the found solution: %d\n",sol_cn);
521:     fprintf(results,"SOLUTION: ");
522:     for (p=0; p<sol_cn; p++) {
523:         if (L[p].root!=NULL) fprintf(results,"%d ",L[p].c);
524:     }
525:     fprintf(results,"\n");
526:     fprintf(results,"GRAPH ----------------------------------------\n");
527:     fprintf(results,"position) color(freq): (arcs)\n");
528:     for (i=0; i<sol_cn; i++) {
529:         if (L[i].root==NULL) continue;
530:         fprintf(results,"%d) %d(%d): ",i,L[i].c,L[i].en);
531:         e=L[i].root;
```

```
532:         while (e!=NULL) {
533:             fprintf(results,"(%d,%d) ",e->u,e->v);
534:             e=e->next;
535:         }
536:         fprintf(results,"\n");
537:     }
538:    fprintf(results,"\n");
539:    fprintf(results,"================================================================================
540:    fprintf(results,"\n");
541: }
542:
543:
544: //------------------------------- MAIN:-------------------------------------//
545:
546: void main() {
547:    FILE *fp;
548:    FILE *results;
549:    FILE *heuristic;
550:    Graph G,H;
551:     int k;
552:    int gn;
553:     int value;
554:    char answer;
555:     char filename[40];
556:    char buffer[40];
557:    char vector[40];
558:    double AvgValue;
559:     clock_t u1, u2;
560:     double u;
561:
562:    do{
563:       clrscr();
564:       textmode(_ORIGMODE);
565:       srand(1);
566:       printf("DEMO? \n");
567:     printf("(1) : YES \n");
568:         printf("(0) : NO \n");
569:       printf("\n-> TYPE YOUR CHOICE: ");
570:       scanf("%d",&Demo);
571:       printf("\n");
572:       while ((Demo!=1)&&(Demo!=0)) {
573:         clrscr();
574:         printf("!!! WRONG ANSWER !!! Demo: %d\n",Demo);
575:             printf("YOU MUST TYPE 1 OR 0!!! PLEASE, TRY AGAIN.");
576:         printf("\n\n\n\n\nPress a key to continue ...");
577:             getch();
578:             clrscr();
579:         printf(" Demo? \n");
580:         printf(" (1) : YES \n");
581:             printf(" (0) : NO \n");
582:          printf("\n-> TYPE YOUR CHOICE: ");
583:         scanf("%d",&Demo);
584:          printf("\n");
585:     }
586:
587:         printf("Filename: ");
588:         scanf("%s",filename);
589:       fp=fopen(filename,"rt");
590:         while (fp==NULL) {
```

```
591:         clrscr();
592:         printf("!!! DANGER !!! CANNOT OPEN FILE: %s\n",filename);
593:             printf("THE FILE DOESN'T EXIST! PLEASE, TRY AGAIN.");
594:         printf("\n\n\n\n\nPress a key to continue ...");
595:             getch();
596:             clrscr();
597:          printf("Filename: ");
598:             scanf("%s",filename);
599:          fp=fopen(filename,"rt");
600:       }
601:       if (Demo==1){
602:         gn=1;
603:         }else{
604:         if (Demo==0){
605:                 printf("Number of samples: ");
606:                 scanf("%d",&gn);
607:          }
608:       }
609:       printf("\n==================================== RESULTS ============================
610:     printf("\n");
611:     G.vn=G.ReadNumber(fp);
612:     if (G.vn>VN){
613:         clrscr();
614:         printf("!!! DANGER !!! CANNOT OPEN FILE: %s\n",filename);
615:             printf("THE TOTAL NUMBER OF VERTICES IS %d AND IT EXCEEDS THE MAXIMUM LIMIT OF
616:         printf("\n\n\n\n\nPress a key to continue ...");
617:             getch();
618:         fclose(fp);
619:             clrscr();
620:         return;
621:     }
622:         G.cn=G.ReadNumber(fp);
623:     if (G.cn>CN){
624:         clrscr();
625:         printf("!!! DANGER !!! CANNOT OPEN FILE: %s\n",filename);
626:             printf("THE TOTAL NUMBER OF COLOURS IS %d AND IT EXCEEDS THE MAXIMUM LIMIT OF
627:         printf("\n\n\n\n\nPress a key to continue ...");
628:             getch();
629:         fclose(fp);
630:             clrscr();
631:         return;
632:     }
633:       sprintf(buffer,"%c%c_results_MVCA_%d_%d.txt",filename[0],filename[1],G.vn,G.cn);
634:       results=fopen(buffer,"wt");
635:     if (results==NULL) {
636:         printf("CANNOT OPEN FILE: %s\n",buffer);
637:         printf("\n\n\n\nPress a key to continue ...");
638:             getch();
639:             clrscr();
640:             return;
641:         }
642:     fprintf(results,"%d %d\n\n",G.vn,G.cn);
643:     H.InitEmptyGraph(G);
644:       sprintf(vector,"heur%d%c%c%d.txt",G.vn,filename[0],filename[1],G.cn);
645:       heuristic=fopen(vector,"wt");
646:       value=0;
647:         u=0.000;
648:         for (k=0; k<gn; k++) {
649:             u1=clock();
```

```
650:             G.InitGraph(fp);
651:         H.my_MVCA(G);
652:          H.PostOptimization();
653:         u2=clock();
654:         printf("\t\t\t\t    SAMPLE %d \n",k+1);
655:         fprintf(results,"\t\t\t\t    SAMPLE %d \n",k+1);
656:         H.PrintSolution();
657:         H.PrintGraph();
658:         H.PrintResultsToFile(results);
659:          fprintf(heuristic,"%d\n",H.sol_cn);
660:         value=value+H.sol_cn;
661:          // u = u+ (double(u2 - u1)/1000); //sec
662:             u = u+ (double(u2 - u1));   //msec
663:     }
664:
665:         AvgValue=(value+0.0)/gn;
666:     printf("\n********************** SPERIMENTALS EXECUTING VALUES: ********************
667:     printf("\t\t\t   Average Value: %f\n",AvgValue);
668:         printf("\t\t\t   Average Time (msec): %f\n",(double(u/gn)));
669:     printf("\n\t\tResults saved in the file: \\%s\n",buffer);
670:     fprintf(results,"\n********************** SPERIMENTALS EXECUTING VALUES: **********
671:     fprintf(results,"\t\t\t   Average Value: %f\n",AvgValue);
672:         fprintf(results,"\t\t\t   Average Time (msec): %f\n",(double(u/gn)));
673:     fclose(fp);
674:     fclose(results);
675:       fclose(heuristic);
676:
677:     G.ClearAll();
678:         for (k=0; k<H.cn; k++) {
679:             H.L[k].c=H.cn;
680:         H.L[k].en=0;
681:             H.L[k].root=NULL;
682:         }
683:     H.cn=0;
684:     H.sol_cn=0;
685:       delete Temporary_Vector;
686:
687:         printf("\n\n\n\n\n\n\t\t\t   ANOTHER SIMULATION? (y/n): ");
688:        answer = getch();
689:       textmode(C80);
690:       clrscr();
691:    }while (answer=='y'||answer=='Y');
692: }
693:
```