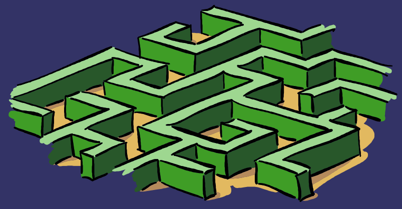
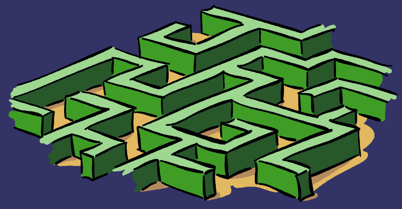
A Graph Program to Navigate a Route

- The application
- External data storage
- Dijkstra's Minimum Spanning Tree Algorithm
- Pseudocode
- Source code
- Test data
- Test results
- Conclusions



The Application

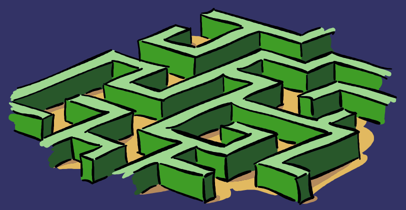
An undirected graph is well suited to modelling a set of roads between places for the purpose of automatically computing the shortest route. In this application, the vertices will be the names of towns or cities, and each edge will be a road segment with a start place and an end place, and a distance between these 2 places.



External Data Storage 1

To avoid repetitive data entry and to minimise data entry effort, graph data is stored externally using text files. One place name is stored directly in each vertex. In order to avoid having to type a long placename when details of a road endpoint are entered, the placename is also stored as a shorter mnemonic form. So the vertex for London is keyed as LO. Here is an exerpt from vertices.txt :

```
LO London
OX Oxford
```

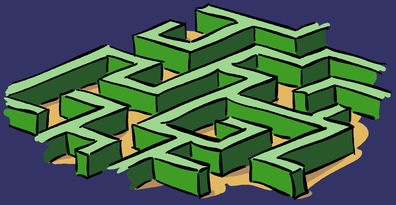


External Data Storage 2

This enables minimisation of the data entry needed for the road between Oxford and London which can be stored within edges.txt as the following text record:

OX LO 56

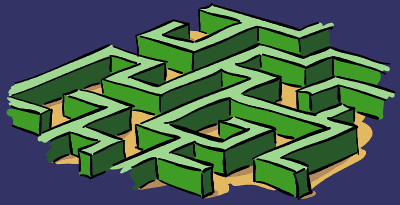
Indicating this road is 56 miles in length. Spaces are used between columns. This makes it easier if place names are not allowed embedded spaces. So a place-name consisting of more than 1 word, e.g. Newcastle upon Tyne has to be hyphenated as: Newcastle-upon-Tyne .



Dijkstra's Algorithm 1

This works by selecting a root for a Minimum Spanning Tree that will be created. A MST identifies a acyclic set of routes by which every vertex connects to the root using the shortest path between it and the root node.

The vertices to be scanned are given a starting distance assumed to exist between themselves and the root node of infinity in theory, or the maximum value of an integer or float in practice. The root node is given a distance to itself of zero. Vertices are then all placed in the set of unscanned vertices. Until all vertices have been scanned, the next vertex to be scanned is selected by finding the vertex with the shortest distance to the root.

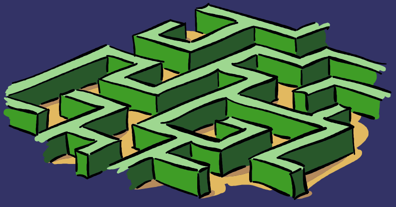


Dijkstra's Algorithm 2

The process of scanning a vertex involves checking the distance to root of all vertices connected to the scanned vertex by edges. This can be speeded up if the edge records were earlier connected to vertex records using adjacency lists When the distance to root of a connected vertex is checked, if the value it currently stores as its distance to root, is greater than the distance to root of the vertex being scanned plus the edge cost, the distance to root of the connected vertex is reduced to that of the vertex being scanned, plus the edge cost. Whenever the distance to root of a connected vertex is reduced, the identity of the previous vertex stored as part of the connected vertex record (i.e. the direction you have to travel to get from the connected vertex towards the root vertex) is updated to the identity of the vertex being scanned.

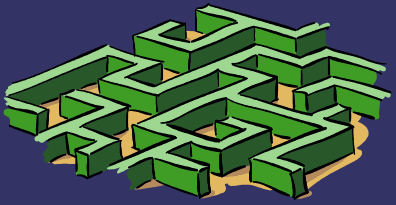
Pseudocode: preparation

```
For each edge:
        Add edge to adjacency list of vertex at from end
        Add edge to adjacency list of vertex at to end
For each vertex:
        Assign scanned = False
        Assign distance to root = infinity
        Assign identity of previous vertex as NULL
For root vertex, assign distance to root = zero.
```



Pseudocode: creation of MST

```
While unscanned vertices exist:
    Extract unscanned vertex with minimum distance to root
        as vertex being scanned (VBS)
    For each edge of VBS:
        DTRVBS = distance to root of vertex being scanned
        DTRVOE = distance to root of vertex at other end,
            (VOE) of edge
        If DTRVOE >= DTRVBS + edge cost:
            Assign DTRVOE = DTRVBS + edge cost
            Assign previous vertex of VOE as VBS
    Assign VBS as scanned = True
```



Source 1: comments

```
/* minspan.c
 * Richard Kay
 * April 2006
 * Implements Dijkstra's Minimum Spanning Tree Algorithm
 * in order to compute and print shortest route between
 * 2 places. To be used together with edges.txt and
 * vertices.txt which supply details of roads
 * and towns respectively */

/* format of vertices.txt :
 * key placename
 * e.g.
 * LO London
 * OX Oxford
 *
 * format of edges.txt
 * startkey endkey distance
 * e.g.
 * LO OX 56
 * indicates road from London to Oxford is 56 miles */
```

Source 2 : edge typedefs

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <limits.h>

typedef struct edge {
        int from; /* index of from place in vertices */
        int to; /* index of to place in vertices */
        int dist; /* distance in miles */
} EDGE;


typedef struct edgelist { /* Linked List of edges */
        int edgidx; /* index of edge in edges[] */
        struct edgelist *next;
} EDGELIST;
```

Source 3: vertex and graph types

```
typedef struct vertex { /* details about town */
        char key[3]; /* short code e.g. LO for London */
        char *place; /* name of town */
        int previous; /* index of place closer to root in MST */
        int dfr; /* distance from root */
        int scanned; /* 1 if this vertex has been scanned */
        EDGELIST *roads; /* list of connecting roads (edges) */
} VERTEX;

typedef struct graphd { /* graph data structure collection */
        VERTEX *vertices; /* array of vertices */
        int nvertices; /* no. of vertices in array */
        EDGE *edges; /* array of edges */
        int nedges; /* no. of edges in array */
} GRAPHD;
```

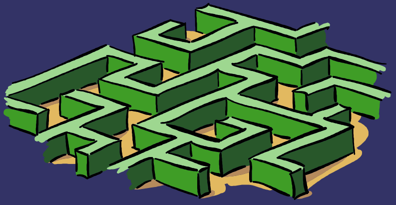
Source 4: function prototypes

```
int countfile(char *fname);
  /* counts \n terminated records in file */
void readedges(char *fname, GRAPHD *gd);
  /* reads edges into edges array */
void readvertices(char *fname, GRAPHD *gd);
  /* reads data into vertices array */
void list_adjacent(GRAPHD *gd);
  /* adds linked list of edges to each vertex */
int get_place(char *type,GRAPHD *gd);
  /* gets journey endpoint */
void print_route(int from,GRAPHD *gd);
  /* Prints all place names and distances along route */
int findplace(char *place,GRAPHD *gd);
  /* returns index of place within vertices array */
void addroad(GRAPHD *gd,int vi,int ei);
   /* adds road index ei to adjacency list of vertex index vi */
```

Source 5: more prototypes etc.

```c
void dijkstras_mst(int root,GRAPHD *gd);
  /* apply Dijkstra's minimum spanning tree algorithm */
int some_unscanned(GRAPHD *gd);
  /* returns true if not all vertices have been scanned */
int closest_unscanned(GRAPHD *gd);
  /* returns index of unscanned vertex closest to root */
int end_road(int start,int edgidx,GRAPHD *gd);
  /* returns index of the other end of a road */

#define VERTFIL "vertices.txt"
#define EDGEFIL "edges.txt"
```

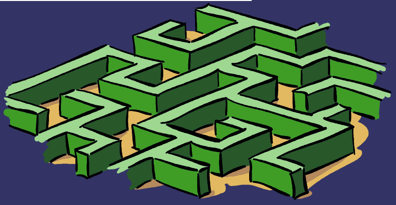


Source 6: main function

```
int main(void){
        GRAPHD gd; /* set of graph data */
        int from, to; /* journey endpoints */
        /* read in vertex then edge data */
        gd.nvertices=countfile(VERTFIL);
        gd.vertices=(VERTEX*)malloc(gd.nvertices*sizeof(VERTEX));
        readvertices(VERTFIL,&gd);
        gd.nedges=countfile(EDGEFIL);
        gd.edges=(EDGE*)malloc(gd.nedges*sizeof(EDGE));
        readedges(EDGEFIL,&gd);
        /* add adjacency lists to vertices */
        list_adjacent(&gd);
        /* get journey start and enpoints */
        from=get_place("start",&gd);
        to=get_place("end",&gd);
        /* create min span tree with root at endpoint */
        dijkstras_mst(to,&gd);
        /* print out route details from startpoint */
        print_route(from,&gd);
        return 0;
}
```

Source 7: count lines in file

```
int countfile(char *fname){
        /* counts no. of \n terminated records in file */
        int lines=0;
        char dummy[BUFSIZ];
        FILE *fh;
        fh=fopen(fname,"r");
        if(!fh){
                fprintf(stderr,"countfile: didn't open file");
                exit(1);
        }
        while(fgets(dummy,BUFSIZ,fh)) lines++;
        fclose(fh);
        return lines;
}
```



Source 8: read edges

```
void readedges(char *fname, GRAPHD *gd){
        /* reads edges into edges array */
        int i=0,ifrom,ito;
        char buff[BUFSIZ],sfrom[30],sto[30],sdist[30];
        FILE *fh;
        fh=fopen(fname,"r");
        for(i=0;i < gd->nedges;i++){
                fgets(buff,BUFSIZ,fh);
                sscanf(buff,"%s%s%s",sfrom,sto,sdist);
                if(strlen(sfrom)!=2||strlen(sto)!=2||
                    strlen(sdist)>3||strlen(sdist)<1){
                        fprintf(stderr,
                        "readedges edge %d invalid rec format",i);
                        exit(1);
                }
                ifrom=findplace(sfrom,gd);
                ito=findplace(sto,gd);
                if(ifrom<0||ito<0){
                        fprintf(stderr,"readedges unconnected edge %d",i);
                        exit(1);
                }
                gd->edges[i].from=ifrom;
                gd->edges[i].to=ito;
                gd->edges[i].dist=atoi(sdist);
        }
        fclose(fh);
}
```

Source 9: read vertices

```
void readvertices(char *fname, GRAPHD *gd){
        /* reads data into vertices array */
        char buff[BUFSIZ],key[20],town[BUFSIZ];
        int i;
        FILE *fh;
        fh=fopen(fname,"r");
        for(i=0;i < gd->nvertices;i++){
                fgets(buff,BUFSIZ,fh);
                sscanf(buff,"%s%s",key,town);
                if(strlen(key)!=2||strlen(town)>60){
                        fprintf(stderr,
                        "readvertices vertex %d invalid rec format",i);
                        exit(1);
                }
                strcpy(gd->vertices[i].key,key);
                gd->vertices[i].place=(char*)malloc(strlen(town)+1);
                strcpy(gd->vertices[i].place,town);
                /* initial sentinels for vertex */
                gd->vertices[i].previous=-1;
                gd->vertices[i].dfr=INT_MAX; /* from limits.h */
                gd->vertices[i].scanned=0; /* not yet scanned */
                gd->vertices[i].roads=NULL;
        }
        fclose(fh);
}
```

Source 10: adjacency listing

```c
void list_adjacent(GRAPHD *gd){
        /* adds list of edges to each vertex */
        int i,fpi,tpi; /* from and to place indices */
        for(i=0;i < gd->nedges;i++){
                fpi=gd->edges[i].from;
                tpi=gd->edges[i].to;
                addroad(gd,fpi,i);
                addroad(gd,tpi,i);
        }
}

void addroad(GRAPHD *gd,int vi,int ei){
   /* adds road index ei to road linked list for vertex index vi */
        EDGELIST *el; /* pointer to LL node */
        el=(EDGELIST*)malloc(sizeof(EDGELIST)); /* space for LL node */
        el->next=gd->vertices[vi].roads; /* link into start of list */
        el->edgidx=ei;
        gd->vertices[vi].roads=el;
}
```

Source 11: prompt for route end

```c
int get_place(char *type,GRAPHD *gd){
        /* gets a journey endpoint */
        char buff[BUFSIZ];
        int lenstr,vidx;
        do {
          printf("input key or name for %s place\n",type);
          fgets(buff,BUFSIZ,stdin);
          /* chop \n */
          lenstr=strlen(buff);
          if(buff[lenstr-1]=='\n'){
                buff[lenstr-1]='\0';
                lenstr--;
          }
          vidx=findplace(buff,gd);
        } while(vidx < 0);
        return vidx;
}
```

Source 12: finding utility functions

```c
int findplace(char *place,GRAPHD *gd){
        /* return index of place within vertices array.
         * Finds either place name or key */
        int i;
        for(i=0;i < gd->nvertices;i++){
                if(strcmp(place,gd->vertices[i].place)==0)
                        return i;
                if(strcmp(place,gd->vertices[i].key)==0)
                        return i;
        }
        return -1;
}

int end_road(int start,int edgidx,GRAPHD *gd){
  /* returns index of other end of road */
        if(gd->edges[edgidx].from == start)
                return gd->edges[edgidx].to;
        else
                return gd->edges[edgidx].from;
}
```

Source : 13 Dijkstra's Algorithm

```
void dijkstras_mst(int root,GRAPHD *gd){
        /* Dijkstras algorithm to make
         * Minimum Spanning Tree */
        int i,disti,dist_oe,other_end;
        EDGELIST *el;
        gd->vertices[root].dfr=0; /* root at zero dist from itself */
        gd->vertices[root].previous=-1; /* nowhere closer to itself */
        while(some_unscanned(gd)){
                i=closest_unscanned(gd);
                gd->vertices[i].scanned=1; /* in set of scanned vertices */
                disti=gd->vertices[i].dfr;
                el=gd->vertices[i].roads;
                while(el){ /* check all edges from closest */
                        other_end=end_road(i,el->edgidx,gd);
                        dist_oe=gd->vertices[other_end].dfr;
                        if(dist_oe > disti + gd->edges[el->edgidx].dist){
                                /* update shorter route to other end */
                                gd->vertices[other_end].dfr =
                                disti + gd->edges[el->edgidx].dist;
                                gd->vertices[other_end].previous = i;
                        }
                        el=el->next;
                }
        }
}
```

Source 14: Dijkstra utility functions

```
int some_unscanned(GRAPHD *gd){
  /* returns true if not all vertices have been scanned */
        int i;
        for(i=0;i < gd->nvertices;i++){
                if(!gd->vertices[i].scanned)
                        /* found an unscanned vertex */
                        return 1;
        }
        /* no unscanned vertex found */
        return 0;
}

int closest_unscanned(GRAPHD *gd){
  /* returns index of unscanned vertex closest to root */
        int i,mindist=INT_MAX,nearest=-1;
        for(i=0;i < gd->nvertices;i++){
                if(! gd->vertices[i].scanned &&
                gd->vertices[i].dfr<=mindist){
                        mindist=gd->vertices[i].dfr;
                        nearest=i;
                }
        }
        return nearest;
}
```

Source 15: outputting the route

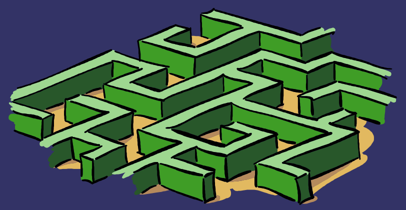
```c
void print_route(int from,GRAPHD *gd){
	/* Prints all place names and distances along route */
	do {
		if(gd->vertices[from].dfr==INT_MAX ||
		gd->vertices[from].previous<0){
			fprintf(stderr,
			"print_route: unconnected vertex");
			exit(1);
		}
		printf("At: %s. Miles to go: %d\n",
				gd->vertices[from].place,
				gd->vertices[from].dfr);
		from=gd->vertices[from].previous;
	} while(gd->vertices[from].dfr);
	/* finally print destination,i.e. root of MST */
	printf("At: %s. Miles to go: %d\n",
		gd->vertices[from].place,
		gd->vertices[from].dfr);
}
```

Test Data

Files vertices.txt and edges.txt were created using a text editor. Details for 55 towns and 93 roads in mainland Britain were input. Some distances were taken from a UK road map and some were guessed. 10 lines from each file are shown.

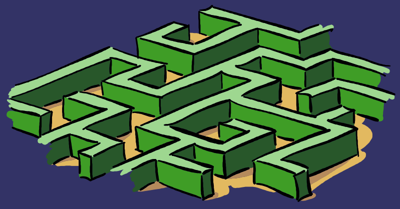
AB Aberdeen
AW Aberystwyth
BK Birkenhead
BI Birmingham
BG Brighton
BR Bristol
CM Cambridge
CA Cardiff
CL Carlisle
CN Carmarthen

PE PL 77
PL EX 44
PL TO 29
TO EX 17
EX PE 110
EX BR 84
EX SA 90
EX SO 109
SA SO 23
SO WN 15



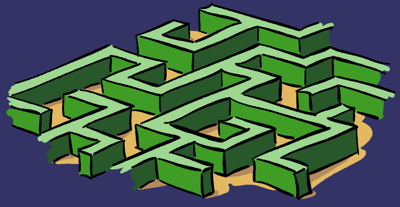
Test Results: Plymouth to Aberdeen

input key or name for start place
Plymouth
input key or name for end place
Aberdeen
At: Plymouth. Miles to go: 698
At: Exeter. Miles to go: 654
At: Bristol. Miles to go: 570
At: Gloucester. Miles to go: 535
At: Cheltenham. Miles to go: 523
At: Worcester. Miles to go: 488
At: Birmingham. Miles to go: 458
At: Manchester. Miles to go: 369
At: Leeds. Miles to go: 325
At: Newcastle-upon-Tyne. Miles to go: 231
At: Edinburgh. Miles to go: 125
At: Aberdeen. Miles to go: 0



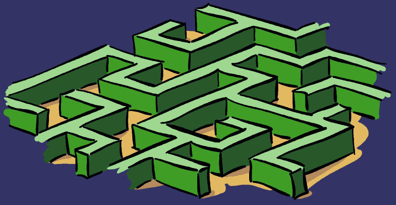
Test Results: Margate to Holyhead

input key or name for start place
Margate
input key or name for end place
Holyhead
At: Margate. Miles to go: 396
At: Dover. Miles to go: 374
At: London. Miles to go: 295
At: Reading. Miles to go: 260
At: Swindon. Miles to go: 220
At: Gloucester. Miles to go: 185
At: Hereford. Miles to go: 140
At: Shrewsbury. Miles to go: 104
At: Holyhead. Miles to go: 0



Test Results: Hastings to Birkenhead

input key or name for start place
Hastings
input key or name for end place
Birkenhead
At: Hastings. Miles to go: 316
At: Brighton. Miles to go: 281
At: London. Miles to go: 222
At: Milton-Keynes. Miles to go: 162
At: Coventry. Miles to go: 125
At: Birmingham. Miles to go: 103
At: Chester. Miles to go: 37
At: Birkenhead. Miles to go: 0



Conclusions

This program solves a moderately complex problem. Design of the program required a study of graph theory and the selection of a standard graph algorithm.

The internal data was designed around the algorithm to minimise programming complexity. The external data was designed to minimise data entry input and errors.

The processing was divided into many small functions each of which could perform a well-contained task. Writing smaller functions around well-designed data is much easier than attempting to debug large functions written to process poorly structured data.