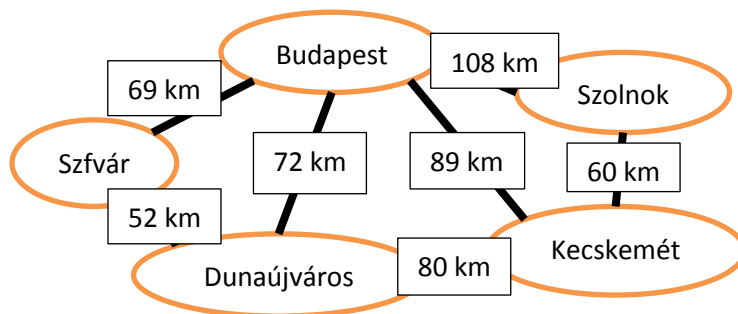


Útvonal-tervező

Specifikáció:

A program feladata az, hogy bekér a bemenetről két várost, és meghatározza ezek között a lehető legrövidebb utat. A kimenetre kiírja azt, hogy milyen városokat érintünk az út során, ezen városok közötti útvonalszakaszok hosszát, és a teljes távolságot. A program a nagyobb magyar városokat, és a köztük lévő főutakat kezeli. Ezt az adathalmazt egy gráfként lehet elképzelni, melynek csúcsaiban a városok vannak, éleihez pedig egy-egy pozitív egész számot rendelünk.

A gráf egy részlete:



Minden városhoz és úthoz azonosító számokkal tárolunk el, hogy a program egyszerűbb legyen, de a bekérésnél és a kiíratásnál a városokat írjuk ki, nem csak az azonosítójukat.

A program az adatokat két adatállományban tárolja:

1. Egy bináris fájlban tároljuk a városok azonosítóit, és nevüket (a gráf pontjai).
2. Az utakat egy szöveges fájlban tároljuk (a gráf élei). Egy út adatstruktúra tartalmazza az út azonosítóját, a két város azonosítóját, melyeket összeköti, és az út hosszát.

A program nagyjából 30-40 várost, és a köztük lévő kb. 80 utat fogja tartalmazni.

Példa a program be- és kimenetére:

Be:

Sopron Szolnok

Ki:

Sopron ---> Győr, 88 km

Győr ---> Tatabánya, 66 km

Tatabánya ---> Budapest, 60 km

Budapest ---> Szolnok, 108 km

Összesen: 322 km

Tervezés:

A problémát Dijkstra-algoritmussal a legegyszerűbb megoldani, mivel leggyorsabban (n^2 lépésszám alatt) ezzel az algoritmussal lehet gráfban megtalálni két pont közötti legrövidebb utat, ha az élekhez pozitív számokat rendelünk.

A városok azonosítóit és neveit egy bináris fájlban tároljuk (varosok.dat), a következőképpen:

- 1 Sopron 2 Győr 3 Szombathely 4 Veszprém 5 Zalaegerszeg 6 Nagykanizsa 7 Keszthely 8 Tatabánya 9 Székesfehérvár...
- Ezeket egy „varos” típusú struktúrába olvassuk be (int azonosito, char varosnev[30]), és ezekből csinálunk egy tömböt(varosok[100]), azaz legfeljebb 100 várost képes a program kezelni. Láncolt listával sokkal bonyolultabb és lassabb lenne, hiszen a Dijkstra-algoritmus során egy-egy városhoz sokszor kell távolságokat hozzárendelni.

A városok közötti utakat egy szöveges fájlban tároljuk (utak.txt), ez tartalmazza az út azonosítóját, a két város azonosítóját, melyeket összeköti, és az út hosszát.

- 1. 1 2 88
2. 1 3 73
3. 2 3 107
4. 2 4 81
5. 2 8 66
6. 2 9 85
7. 3 4 115
8. 3 5 54
...- Ezeket egy „ut” típusú struktúrába olvassuk be (int azonosito, int varos1, int varos2, double tavolsag). Az utakból készítünk egy egyirányú láncolt listát, strázsa nélkül.

Most már megvan a gráfunk, ezen kéne lefuttatni a Dijkstra-algoritmust.

Beolvassuk a két várost a standard bemenetről, melyek közti legrövidebb útra a felhasználó kíváncsi. (string kiindulopont, string celpont). A varosok tömbben megkeressük ezek azonosítóit(int start és int finish). A továbbiakban a program az azonosítók szerint fog keresni, és nem a városok neve szerint, csak a végén a kiíratáskor fogjuk visszakeresni, hogy az az azonosító tulajdonképpen melyik városhoz tartozik.

Csinálunk egy double dist[100] tömböt, ebben távolságokat tárolunk, azonban ez nem feltétlenül a végleges távolság, hanem csak az algoritmus futtatása során addig megtalált legrövidebb út. A kezdőpont d-jét 0-ra állítjuk. A többi végtelenre kéne, de mivel a c-ben nincs ilyen, ezért 10000-re állítjuk, mert ekkora értékeket amúgy sem kaphatnánk az algoritmus során.

Ahhoz, hogy tudjuk, hogy egy adott pont esetén tényleg megtaláltuk-e a legrövidebb távolságot, csinálunk egy logikai változót (int megvan), melynek értéke 1, ha már megvan a végleges távolság, és 0, ha még nincs. Ebből is egy tömböt csinálunk, mely a városokra vonatkozik. A kezdőpont esetén ezt 1-re állítjuk, a többi város esetén 0-ra.

Ezután jön a Dijkstra-algoritmus fő része:

- Csinálunk egy int legutolso változót, ez mindig annak a városnak az azonosítóját fogja tartalmazni, amelyiknek legutoljára megtaláltuk a végleges távolságát. Ez kezdetben a kiindulópont.
- Végiglépkedünk a láncolt listán: Ha találunk egy olyan elemet, amiben a varos1 vagy a varos2 megegyezik a legutolso-val, akkor belépünk egy feltételbe:
HA $\text{dist}[\text{legutolso}] + \text{utak.ut.tavolsag} < \text{dist}[\text{utak.ut.varos1 vagy varos2}]$ (attól függően, hogy a varos1 vagy a varos2-ben van-e a legutolso)
AKKOR $\text{dist}[\text{utak.ut.varos1 vagy varos2}] = \text{dist}[\text{legutolso}] + \text{utak.ut.tavolsag}$
- Ezt addig csináljuk, amíg a láncolt lista végére nem érünk.
- Ezután azok közül, melyeknek nem tudjuk a végleges távolságát, megvizsgáljuk a legkisebbet, ennek a „megvan” értékét átállítjuk egyre, és a legutolsó változóba berakjuk ennek a városnak az azonosítóját:
- Ezt az egész folyamatot egy ciklusba rakjuk, és addig futtatjuk, amíg a célpont „megvan” értéke 1 lesz. Ekkor a $\text{dist}[\text{célpont}]$ lesz a probléma megoldása.

Ha azt is tudni akarjuk, hogy milyen úton jutottunk el a célpontba, nem csak a távolságot, a Dijkstra egy kicsivel bonyolultabbá válik: amikor egy város megvan értékét 1-re állítjuk, azt is el kell tárolnunk, hogy honnan jutottunk el oda egy külön int honnan változóba. Ennek segítségével a végén vissza tudjuk fejteni visszafelé (a célpont felől) az utat, és kiíratni.

Dokumentáció

Programozói dokumentáció:

A program bemenete a fájlokból egy gráfnak megfeleltethető térkép, pontokkal (városokkal) és a köztük futó élekkel (utak) megadva. A városokat egy `typedef struct { int azonosito; char varosnev[30]; double dist; int megvan; int honnan; } varos;` tömbben tároljuk, ezt a main függvény elején deklaráljuk. A városok adatait (azonosito, varosnev) egy bináris fájlból olvassuk be (varosok.dat). Ezt a beolvasbin függvényt a main hívja meg. Először csinálunk egy ideiglenes struktúratömböt: `typedef struct { int azonosito; char varosnev[30]; } beolvasott;`, ebbe olvassuk be közvetlenül a bináris fájlban tárolt adatokat, majd ezeket átmásoljuk a `varos[]` tömb megfelelő számú elemeibe. Erre azért van szükség, mert a városok adatainak nyilvántartása során a Dijkstra algoritmus közben további információkat kell eltárolni. A függvény paraméterlistán visszaadja a városok tömbjét és a városok számát.

A városok közötti utakat egy szöveges fájlból olvassuk be (utak.txt), ebben egy úthoz 3 adat tartozik: a két város, melyeket összeköt, és az út hossza. Egy ilyen adatstruktúra a következőképpen néz ki: `typedef struct { int varos1; int varos2; double tavolsag; } utak;`. Ezeket egy egyirányú, strázsa nélküli láncolt listában tároljuk el: `typedef struct lista { utak ut; struct lista *next; } lista;`. A beolvasás a fájlból a következőképpen van megoldva:

```
while (fscanf(fp, "%d %d %lf", &(p->ut.varos1), &(p->ut.varos2), &(p->ut.tavolsag)) == 3)
{
    p->next = (lista*)malloc(sizeof(lista));
    p = p->next;
}
```

Ez azt eredményezi, hogy az utoljára beolvasott adat már nem helyes, az `fscanf` nem 3-at ad vissza, de a program mégis beolvassa. Emiatt ki kell törölnünk a láncolt lista utolsó elemét. Ezt a szöveges fájlból beolvasást, és a törlést a `beolvastxt` függvény csinálja meg, mely paraméterként a main-ben létrehozott listára mutató head-pointert kapja meg.

Ezután következik a konzolról való beolvasás, hogy a felhasználó melyik két város távolságára kíváncsi. A függvény paraméterként megkapja a városok tömbjét, és a városok darabszámát, majd a két város azonosítóját adja vissza paraméterlistán.

Az első három függvény `int` típusú. Ez a hibakezelés miatt van így. Ha nincs hiba, akkor 0-t adnak vissza, ha van benne hiba, akkor 1-et.

A `varosok[]` tömb `varos` típusú struktúrákból áll: `typedef struct { int azonosito; char varosnev[30]; double dist; int megvan; int honnan; } varos;` Ezeknek már beolvastuk az azonosítóját és a nevüket a fájlból, a többi értéket kell meghatároznunk. Ezeket a beállított függvénnyel inicializáljuk. A `dist` a kiindulóponttól vett távolságot tárolja el, értéke a kiindulópontra kezdetben 0, a többi pontra 10000 (ennél biztosan kisebb lesz a végleges `dist`). A `megvan` azt jelenti, hogy egy adott városnak már megtaláltuk –e a végleges távolságát. Ez a kiindulópontra 1, a többire 0. A `honnan` érték azért kell, hogy vissza tudjuk fejteni az útvonalat: Minden városra eltárolja, hogy honnan

jutottunk el oda a Dijkstra-algoritmus során, azaz ennek segítségével visszafelé meg tudjuk mondani nemcsak a legrövidebb út hosszát, hanem magát a legrövidebb utat is. Ezt csak a kezdőpontra inicializáljuk, értéke -1.

Ezután jön a Dijkstra-algoritmust lefuttató függvény, a `dijkstra`. Ez addig fut, amíg a célpont város megvan értéke nem egy, azaz még nincs meg a célpont végleges távolsága. A függvény nem ad vissza értéket, a városok tömböt módosítja, annak a honnan, dist, és megvan értékeit.

A honnan értékeket az `utvonalterv` függvény dolgozza fel. Ez fejt vissza a legrövidebb utat, azt, hogy mely városokat kell érintenünk. Az út során érintett városok számát nem tudjuk, ezért azokat egy láncolt listába rakjuk bele. A listában kezdetben a célpont azonosítója van. Annak van egy honnan értéke. Tehát egy listaelem típusa: `typedef struct utvonalt { int honnan; int mostani; struct utvonalt *next; } utvonalt;` . Mindig az első elem elé szúrjuk be azt, ahonnan oda érkeztünk. Így a végén az út vonal szerinti sorrendben lesznek a városok. Ez a függvény egy ilyen típusú elemre mutató eleje pontert kap, és azt is ad vissza, hiszen a lista elejére fűzés során a kezdőcím megváltozik.

A main ezután a `kiir` függvényt hívja meg. Ez végigmegy az előzőleg létrehozott cikluson, és minden lépésben 3 dolgot ír ki: a honnan azonosítójú város nevét, a mostani azonosítójú város nevét, ill. a távolságot. A távolság megkeresésére a `kiir` a ciklus minden lépésében meghívja a `szakasz_hossz` függvényt: ez megkeresi az utak listájában azt az utat, amelyek a honnan és a mostani várost kötik össze, majd ezen út hosszát adja vissza. A `kiir` függvény még ezután kiírja a a célpont dist-jét, mint összes távolságot.

Ezután a `listatorol` függvény törli mindkét láncolt listát, hogy ne legyen memóriaszivárgás.

Felhasználói dokumentáció:

A fájlok tartalma:

bináris fájl: `varosok.dat` azonosító városnév(ékezet nélkül, nagy kezdőbetűvel)

szöveges fájl: `utak.txt` azonosító1 azonosító2 úthossz

Az azonosítók 0-tól kezdődő egész számokat.

A konzolra beolvasásnál:

A két várost kell beírni, szóközzel vagy enterrel elválasztva, nagy kezdőbetűvel, ékezet nélkül. Rossz megadás esetén a program hibaüzenetet ír ki.

Tesztelési dokumentáció:

A teszteléshez használt fájlok:

- `varosok.dat` (szöveges formában):

```
{0, "Sopron"},  
{1, "Gyor"},
```

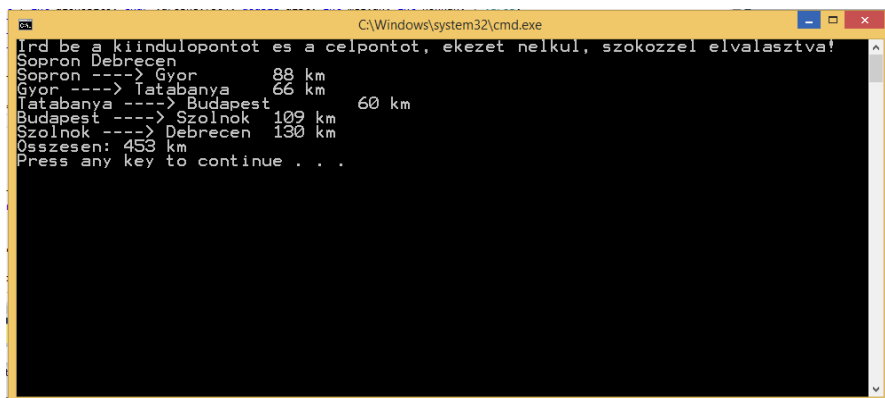
{2, "Szombathely"},
{3, "Veszprem"},
{4, "Zalaegerszeg"},
{5, "Keszthely"},
{6, "Tatabanya"},
{7, "Szekesfehervar"},
{8, "Budapest"},
{9, "Dunaujvaros"},
{10, "Szekszard"},
{11, "Siofok"},
{12, "Pecs"},
{13, "Kaposvar"},
{14, "Nagykanizsa"},
{15, "Kecskemet"},
{16, "Szolnok"},
{17, "Salgotarjan"},
{18, "Eger"},
{19, "Miskolc"},
{20, "Baja"},
{21, "Szeged"},
{22, "Hodmezovasarhely"},
{23, "Bekescsaba"},
{24, "Debrecen"},
{25, "Nyiregyhaza"}

- utak.txt

0 1 88
0 2 73
1 2 107
1 3 81
1 7 85
1 6 66
2 3 115
2 4 54
3 5 76
3 7 47
4 5 43
6 7 55
4 14 51
5 14 53
5 11 86
3 11 46
7 11 46
6 8 60
7 8 64

8 9 84
7 9 52
7 10 102
9 10 81
10 11 93
10 12 60
11 12 115
11 13 80
12 13 65
5 13 87
13 14 80
8 17 114
8 18 135
17 18 64
18 19 65
19 25 92
24 25 50
16 24 130
16 18 110
8 16 109
8 15 87
9 15 82
15 16 60
16 23 108
23 24 124
22 23 69
16 22 94
21 22 26
15 21 90
15 20 108
10 20 39
20 21 101

A program megfelelően működik. Például:



```
C:\Windows\system32\cmd.exe
Ird be a kiindulopontot es a celpontot, ekezet nelkul, szokozzel elvalasztva!
Sopron Debrecen
Sopron ----> Gyor      88 km
Gyor ----> Tatabanya  66 km
Tatabanya ----> Budapest 60 km
Budapest ----> Szolnok  109 km
Szolnok ----> Debrecen 130 km
Osszesen: 453 km
Press any key to continue . . .
```