Algoritmusok és adatszerkezetek

Készítette: Komlós Vivien [LIN0C8]

Feladat:

Írjon programot, amely egy gyártási folyamat *ütemterv gráfjában* megkeres egy *kritikus utat*!
*Ütemterv gráf***:** olyan gráf, melynek csúcsai a gyártási folyamat egyes állapotai, a gráf élei, pedig tevékenységek, amelyek a tevékenység időtartamával vannak súlyozva. A gráf tartalmaz két kitüntetett állapotot (csúcsot), a start- (S) és a végállapot (V), amelyek a gyártási folyamat kezdetét, illetve végét jelölik.
*Kritikus út***:** S-ből V-be vezető maximális időtartamú („maximális költségű") út. Tehát a „termék" gyártási ideje legalább akkora, mint a kritikus út időtartama („hossza").

Futtatás:

A programozáshoz Visual Studio 2012 fejlesztőkörnyezetet használtam. Az AlgoBead.sln fájlt kell megnyitni a forráskód megtekintéséhez, program futtatásához.

Az inputfájl szerkezete:

```
7
A B C D E F G
0 100 200 0 0 300 0
0 0 0 0 0 0 0
0 0 0 100 0 0 20
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 40 1000 0
A
B
```

- csúcsok száma
- csúcsok nevei
- gráf csúcsmátrixa

- kezdő csúcs neve
- végpont neve

A gráf szerkezete:

A gráf szerkezetéhez két osztályt hoztam létre. Az egyik a *Vertex*, ami a csúcsokat reprezentálja, a másik az *Edge*, ami az éleket.
A *Vertex* osztályt a csúcs nevének megadásával lehet példányosítani.
Adattagjai:

- *név*
- *edges* – dinamikus tömb (vektor), mely a csúcshoz tartozó éleket tartalmazza
- *max_dist* – a Dijsktra algoritmushoz szükséges, a maximum távolságot tárolja, kezdetben -1
- *parent* – a Dijsktra algoritmushoz szükséges, a szülő csúcs pointerét tárolja, kezdetben nullptr
- *color* – a Dijsktra algoritmushoz szükséges, a csúcs színét tárolja (false-fehér, true-fekete), kezdetben false
- *depthNumber* – a Mélységi bejáráshoz szükséges, a mélységi számot tárolja, kezdetben 0
- *traversalNumber* – a Mélységi bejáráshoz szükséges, a bejárási számot tárolja, kezdetben 0
- *depthColor* – a Mélységi bejáráshoz szükséges, a csúcs színét tárolja, (0 – fehér, 1 – szürke, 2 – fekete), kezdetben 0
- *deepParent* – a Mélységi bejáráshoz szükséges, a szülő csúcs pointerét tárolja, kezdetben nullptr

Az *Edge* osztályt a kezdőcsúcs, a végcsúcs és a távolság megadásával lehet példányosítani. Adattagjai:

- *origin* – az él kiinduló csúcsának pointere
- *destination* – az él végpontjának pointere
- *distance* – a két csúcs közötti távolság

A csúcsokat egy *Vertex* objektumokat tartalmazó dinamikus tömb tárolja. Ehhez a C++ *vector* adatszerkezetét használtam, mivel a tömb adatszerkezet nem hozható létre változó méret megadásával.

Beolvasás:

A fájl beolvasását a *read()* nevű függvény soronként végzi, majd a *split()* függvény segítségével szóközönként darabolja.
Elsőként a csúcsok számát tárolja el a *numberOfVertices* nevű változóban, majd *Vertex* példányokat hoz létre, amiket a *vertices* vektorhoz ad.

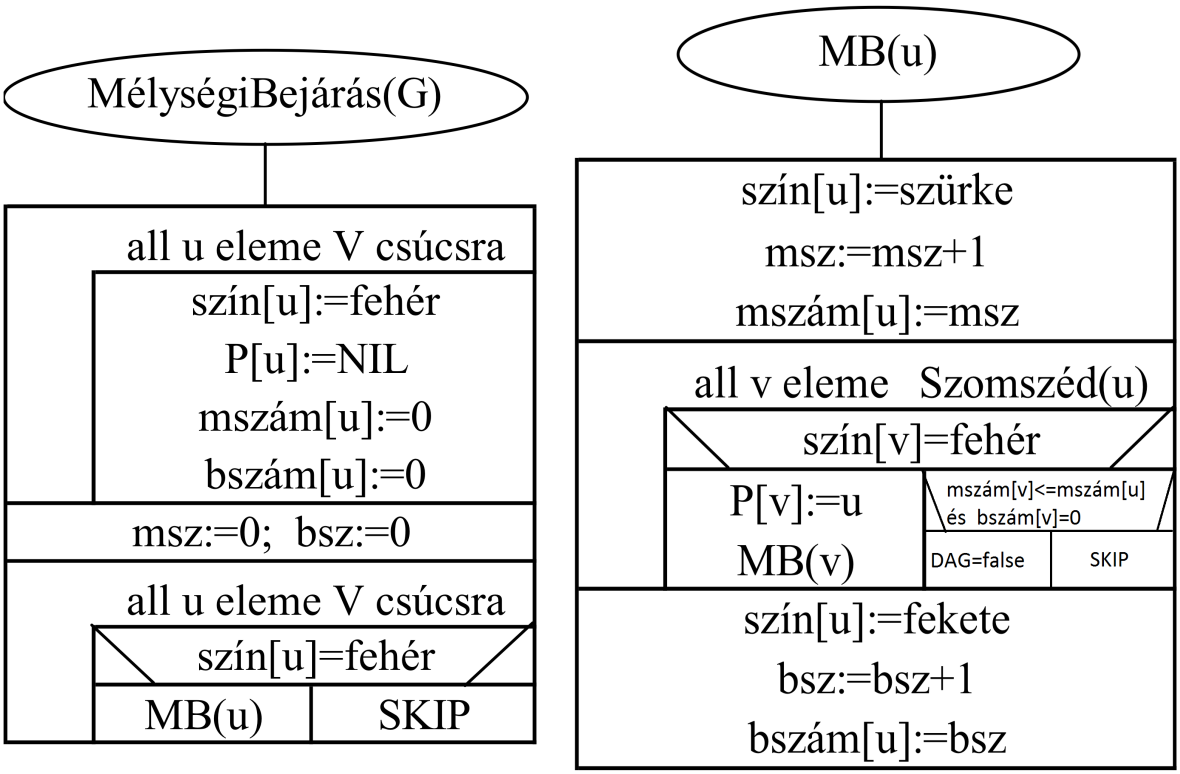
Ezután feltölti a csúcsokhoz tartozó *edges* vektorokat. Végül tárolja a kezdőcsúcsot a *start_vertex_name* nevű változban és a végállapotot a *destini_vertex_name* változóban.

A szükséges algoritmusok:

Mélységi bejárás

Ennek az algoritmusnak a segítségével meghatározható, hogy a gráf irányított körmentes-e (DAG). Egy kezdőpontból kiindulva addig megyünk egy él mentén, ameddig el nem jutunk egy olyan csúcsba, amelyből már nem tudunk továbbmenni, mivel nincs már meg nem látogatott szomszédja. Ekkor visszamegyünk az út utolsó előtti csúcsához, és onnan próbálunk egy másik él mentén továbbmenni. Ha ezen az ágon is minden csúcsot bejártunk, ismét visszamegyünk egy csúcsot, és így tovább.

A bejárás során tároljuk, hogy a csúcsot hányadikként értük el (hányadikként lett szürke), hogy hányadikként fejeztük be a csúcs és a belőle elérhető csúcsok bejárását (hányadikként lett fekete). Ezek a mélységi és a befejezési számok.



Implementálása

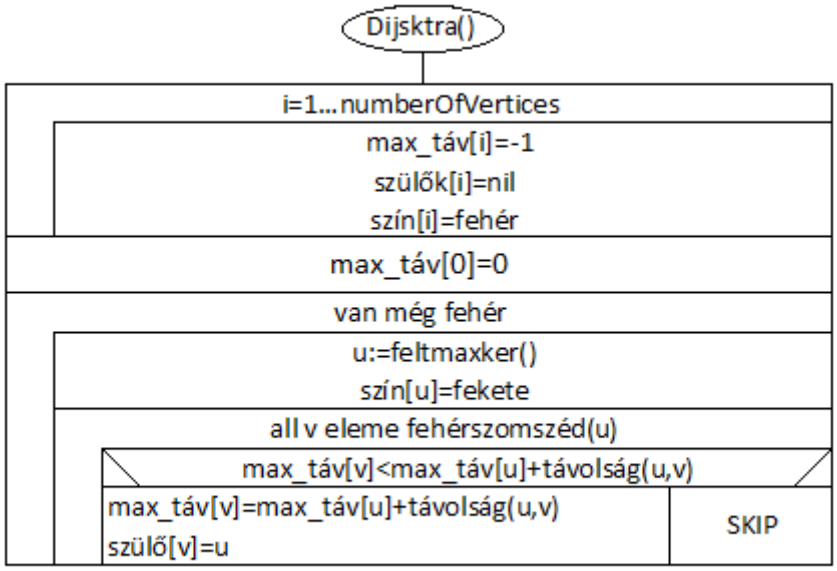
A programban a MélységiBejárás() függvényt *DepthFirstSearch()* néven implementáltam. Az első ciklust nem kellett megírnom mivel az értékek deklarálását elvégzem a csúcsok példányosításakor, ezek az értékek így nem tömbökben, hanem az osztály adattagjaiként tárolódnak.

Az MB() függvényt *DF()* néven implementáltam. Ha v csúcs színe nem fehér megvizsgálom, hogy visszaél-e.

Dijsktra-algoritmus

Ez az algoritmus egy gráfban megkeresi a legrövidebb utakat egy adott csúcsból indulva, így maximális utak keresésére átírva alkalmazható a fenti feladat megoldásához.

A módosított algoritmus stuktogrammja



Implementálása

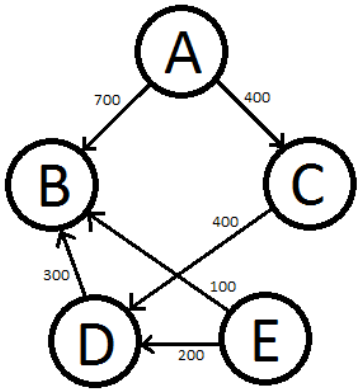
Az első ciklust itt sem kellett megírnom mivel az értékek deklarálását elvégzem a csúcsok példányosításakor, ezek az értékek így nem tömbökben, hanem az osztály adattagjaiként tárolódnak.

Tesztelés

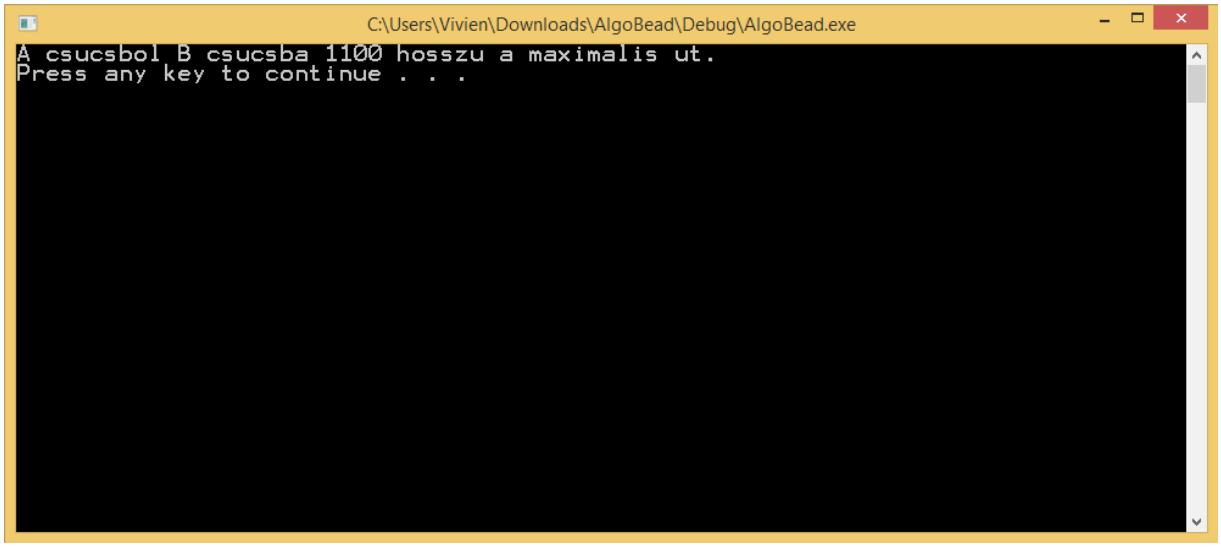
1. Irányított körmentes gráf, a start és a végpont között van út:
Ebben az esetben a program helyesen lefut, megtalálja a maximális utat a két csúcs között.

Példa: test1.txt

```
5
A B C D E
0 700 400 0 0
0 0 0 0 0
0 0 0 400 0
0 300 0 0 0
0 100 0 200 0
A
B
```



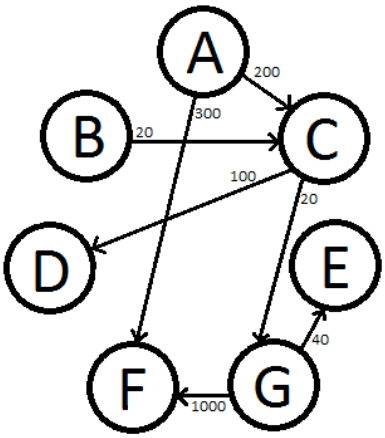
A csúcsból vezet egy 700 hosszú él is B-be, de a program a megfelelő, A-C-D-B 1100 hosszú utat választja ki.

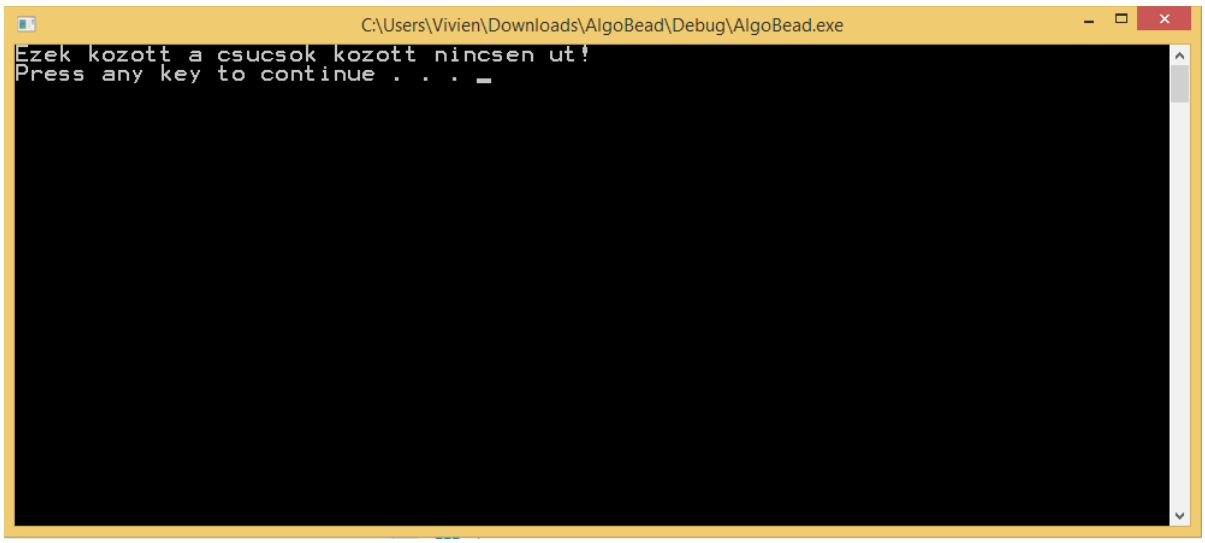


2. Irányított körmentes gráf, a start és a végpontok között nincs út:
Ebben az esetben a program felismeri, hogy nincsen út és ezt jelzi a felhasználó felé.

Példa: test2.txt

```
7
A B C D E F G
0 0 200 0 0 300 0
0 0 20 0 0 0 0
0 0 0 100 0 0 20
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 40 1000 0
A
B
```

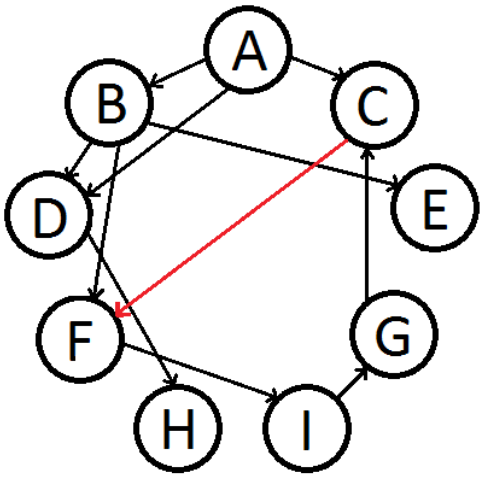




3. Irányított nem körmentes gráf:
Ebben az esetben a program felismeri és kiírja a hibát.

Példa: test3.txt

```
9
A B C D E F G H I
0 1 1 1 0 0 0 0 0
0 0 0 1 1 1 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1
0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0
A
B
```



A gráfban C-F él visszaél.