Kőrösi Gábor

Algoritmusok és adatszerkezetek a gyakorlatban

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Gráfok

Összefoglaló

Az első olyan feladat, amelyet gráfok segítségével oldottak meg, a königsbergi hidak problémája volt. A város lakói olyan sétát szerettek volna tenni, hogy mind a hét hídon átsétáljanak (de mindegyiken csak egyszer), és visszajussanak a kiindulópontra. Nem sikerült, ezért a kor híres matematikusához, Eulerhez fordultak. Euler bebizonyította, hogy a kívánt séta nem lehetséges. Az eddigiekben a lineáris adatstruktúrákkal, és a bináris fákkal foglalkoztunk. Most egy újabb nem lineáris adatszerkezetekkel, a gráfokkal fogunk megismerkedni.

Lecke fejezetei:

- Mi is az a gráf? – Olvasó (2 perc)
- A gráfok ábrázolása – Olvasó (8 perc)
- A gráfok bejárása – Olvasó (35 perc)
- Összetett időkomplexitás – Olvasó (10 perc)
- Gyakorló feladatok – Gyakorlati (50 perc)

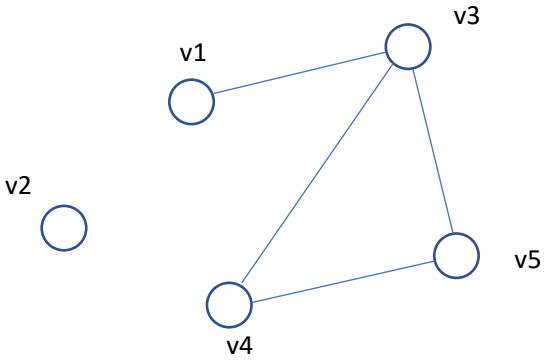
Téma típusa: Gyakorlati
Olvasási és gyakorlási idő: 90 perc

Mi is az a gráf? (2 perc)

A gráfok az informatikában gyakran előforduló matematikai struktúrák. Használata számos tudományterületen jelen van. Az internet is felfogható egy gráfként, de akár egy adott épület villamos hálózata is. Az emberek közötti ismeretségek is kezelhetők gráfként (Facebook). A GPS is gráfként kezeli a térképeket, és gráfelméleti algoritmusok használatával állítja elő az útvonalat. A molekulák is vizsgálhatók gráfként, mint az atomok és a köztük lévő kötések.
Ahogy a bináris fáknál, úgy itt is vannak élek (edge) és csomópontok, csúcsok (node, vertex). A gráfoknál kevésbé szigorúak a kapcsolódási szabályok. A gráfokat a matematikában mélyrehatóan alkalmazzák és tanulmányozzák. Az informatikában nem a matematikai modellezéssel, hanem az adatstruktúrával foglalkozunk.

Gráf: **G(V,E)** ahol:
G – gráf pontjainak halmaza
V – csúcsok (az angol vertex = csúcs szóból)
E – élek (az angol edge = él szóból)

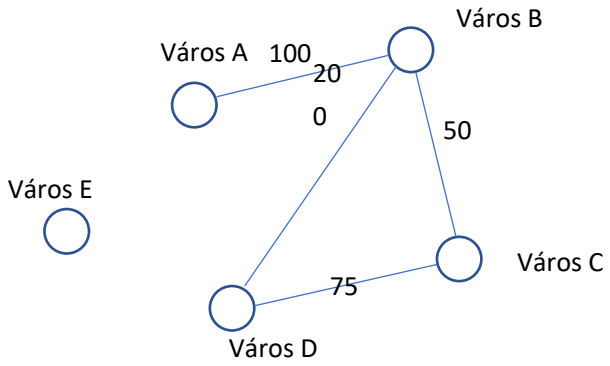
Példa egy gráfra:



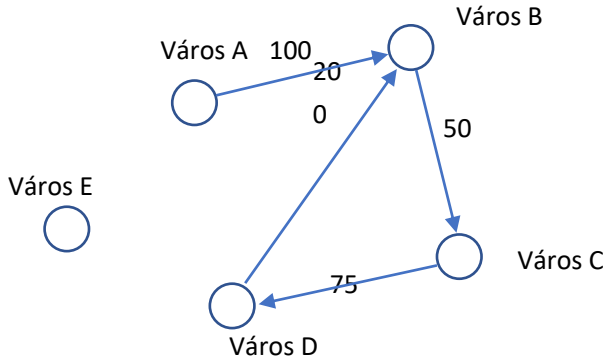
Csomópontok halmaza:
V(G) = {v1,v2,v3,v4,v5}
Élek halmaza:
E(G) = {(v1,v3), (v3,v4), (v3,v5), (v4,v5) }

Az éleket kétféleképpen tudjuk ábrázolni, hiszen lehetnek irányítottak és irányítatlanok is. A facebook is egy ilyen irányítatlan gráf. A párkeresés esetén a gráf irányított marad, ha nem kölcsönös a vonzalom.

A gráfok egy másik tulajdonsága a súlyozás. Egy gráf súlyozása alatt értjük azt a folyamatot, melynek során az egyes éleknek különböző nagyságú étékeket adunk. Egy gráf ebből adódóan lehet súlyozott vagy súlyozatlan. A súlyozott gráfokkal számtalan problémára adhatunk választ. Pl. egy vasúti hálózat gráf modellje:



Amennyiben egyirányú a forgalom, úgy lehet irányított is a gráf.



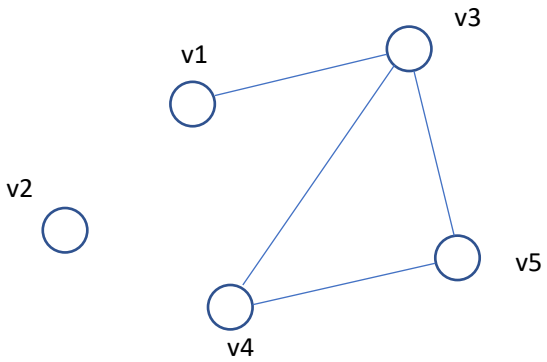
A gráfok ábrázolása (8 perc)

A gráfok pontokból (NODE, Vertex) és élekből (Edge) épülnek fel. Adja magát a kérdés, hogy hogyan tudnánk ezt ábrázolni egy programozás nyelvben?

A legegyszerűbb megoldás, ha létrehozunk két listát. A csomópontokhoz elegendő egy egydimenziós tömb, míg az élekhez tárolásához ábrázolásra van szükség. A tároláshoz a fához hasonló szerkezetet használhatunk fel. A struktúra ismeretében már feltölthetjük az élek adatait. Mivel tudjuk, hogy ez egy irányítatlan gráf, így mindegy, hogy melyik a kezdő, és melyik a fejező pont a táblázatunkban

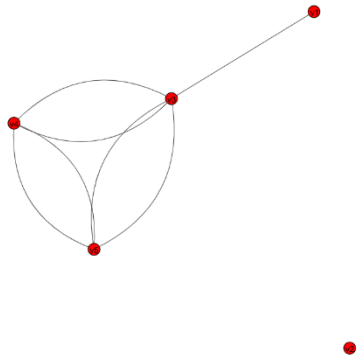
Vertex lista
1
2
3
4
5

Él lista	
1	3
3	4
3	5
4	3
4	5
5	3
5	4



Természetesen a gráf éle lehet súlyozott is. Ilyenkor a csatlakozási pontok mellett ezt is fel kell tüntetnünk. Egy ilyen ábrázolási formát szemléltet a következő Python kód:

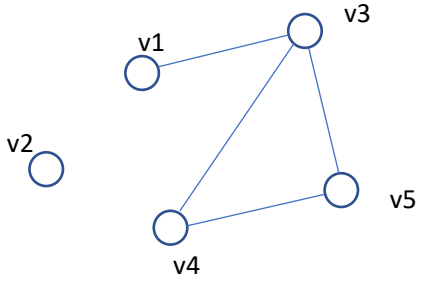
```
from igraph import *
g = Graph()
g.add_vertices(6)
g.add_edges([(1,3),(3,4),(3,5),(4,3),(4,5),(5,3),(5,4)])
g.vs["label"] = ["v1","v2","v3","v4","v5"]
layout = g.layout("kk")
plot(g, layout = layout)
```



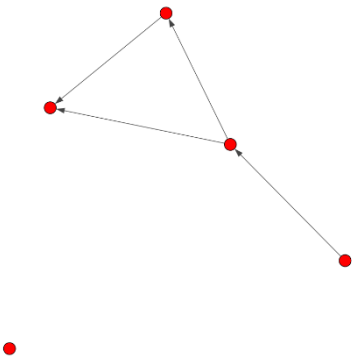
Ez a megoldás azonban nem hatékony, mivel el kell gondolkodunk egy-egy adatmozgatás „költségén” is. Az előző megoldás túl bonyolult keresésének és kezelésének megoldására alkalmazzák a SZOMSZÉDOSSÁGI MÁTRIX-okat. A szomszédsági mátrixban az oszlopok és sorok szemléltetik a NODE-jainkat még a táblázat értékei a csomópontok közötti éleket. Súlyozatlan gráf esetén ezek az értékek 0 és 1 értékűek lesznek. A táblázatunk irányítatlan gráfok esetén az átlón tükrözve egyforma lesz.

Vertex lista
1
2
3
4
5

	V1	V2	V3	V4	V5
V1	0	0	1	0	0
V2	0	0	0	0	0
V3	0	0	0	1	1
V4	0	0	0	0	0
V5	0	0	0	1	0



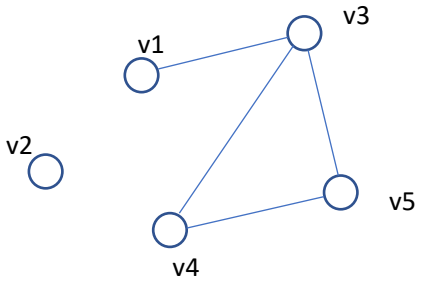

```
from igraph import *
import numpy as np

G = np.array([[0,0,1,0,0],
              [0,0,0,0,0],
              [0,0,0,1,1],
              [0,0,0,0,0],
              [0,0,0,1,0]])

g = Graph.Adjacency((G > 0).tolist())
layout = g.layout("kk")
plot(g, layout = layout)
```



A tárolás már egyértelmű, de mennyi időt igényel egy-egy elem elérése?

Mivel a VERTEX listán és az EDGE listán is át kell haladnunk így O(v). A szomszédossági mátrixok kezelésének számtalan előnye van, azonban O(v*v) helyfoglalása miatt aligha nevezhető memóriabarát folyamatnak. Épp ezért, a tömbök és a szomszédossági lista mellett ábrázolhatjuk a gráfokat SZOMSZÉDOSSÁGI LISTÁBAN is. A szomszédsági lista, egy adat struktúra mely az élek tárolásához tömbben tárolt láncolt listákat használ fel. Ennek felépítését a következő ábra és kódrészlet szemlélteti.

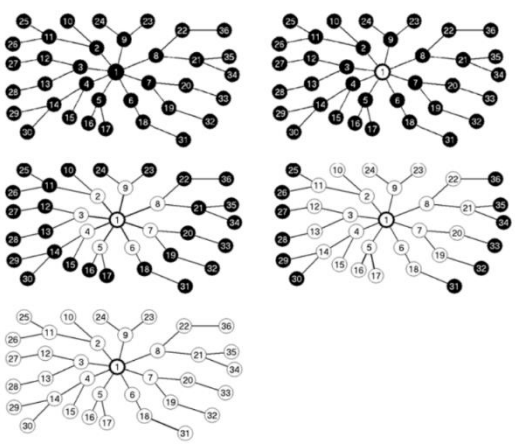
Vertex lista			
1	V3		
2			
3	V1	V4	V5
4	V3	V5	
5	V3	V4	




```
# A szomszedsagi lista abrazolasa
class AdjNode:
    def __init__(self, data):
        self.vertex = data
        self.next = None

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [None] * self.V

    # adjunk uj elet a grafhoz
    def add_edge(self, src, dest):
        node = AdjNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node

        node = AdjNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node

    # irjuk ki a grafot
    def print_graph(self):
        for i in range(self.V):
            print("Szomszedsagi vertex lista. Vertex {}\n head".format(i), end="")
            temp = self.graph[i]
            while temp:
                print(" -> {}".format(temp.vertex), end="")
                temp = temp.next
            print(" \n")
V = 5
graph = Graph(V)
graph.add_edge(0, 1)
graph.add_edge(0, 4)
graph.add_edge(1, 2)
graph.add_edge(1, 3)
graph.add_edge(1, 4)
graph.add_edge(2, 3)
graph.add_edge(3, 4)

graph.print_graph()
```

A gráfok bejárása (35 perc)

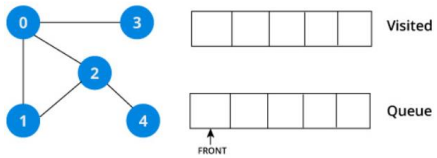
A gráfok két alapvető bejárási módja a mélységi és a szélességi bejárás. Az ezekkel a módszerekkel bejárt útvonalat mélységi vagy szélességi feszítőfának is szokták nevezni. A bejárási stratégiák egyszerű példája, egy öreg városka girbe-görbe utcáin bolyongó, kopott emlékezetű lámpagyújtogató.



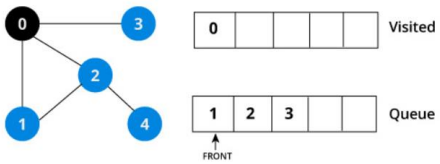
A szélességi bejárás szerint a lámpagyújtogatót egy este több barátja elkíséri a városka főterétre, ahol együtt meggyújtják az első lámpát. Utána annyi felé oszlanak, ahány utca onnan kivezet. A különvált kis

csapatok meggyújtják az összes szomszédos lámpát, majd tovább osztódnak. A városka lámpáit ilyen módon széltében terjeszkedve érik el.

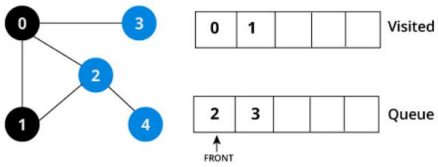
A megvalósításhoz, egy már tanult adatstruktúrára, a sorra van szükségünk. A sor képes az elemeket tárolni, és ez által megfelelő sorrendben visszaadni. Ez úgy valósul meg, hogy az első lámpa összes szomszédját beírjuk a sorba, majd ezeket sorra kivesszük, miközben az ő szomszédjaikat is beírjuk a sorba. Azonban, rögtön láthatóvá válik, ha minden elem minden szomszédját beírnánk a sorba, akkor rengeteg redundáns (ismétlődő) adatunk lenne. E probléma megoldására valahogy jeleznünk kell a programnak, hogy egy-egy VERTEX-et már kiírtunk. Ehhez egy új mezőt hozunk létre, ez pedig a megnézve boolean változó lesz.
Nézzünk egy példát, amelyen keresztül egyértelművé válik a bejárás működése! Vegyünk egy egyszerű gráfot:



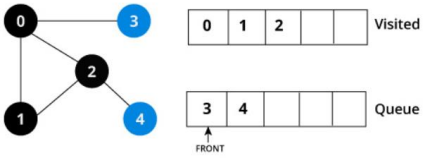
Először kiválasztjuk az induló elemünket. Esetünkben ez a 0 lesz.
Beolvassuk az elemünkhöz kapcsolódó szomszédokat a SORBA, átállítjuk azok megnézve változóját TRUE-ra és kiírjuk az elemet a képernyőre.



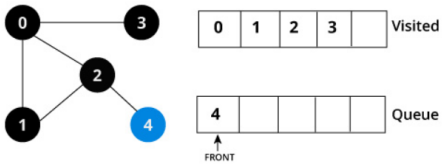
A következő lépésben az 1-es VERTEX-et írjuk ki, illetve a SORBA beírjuk a hozzá tartozó olyan elemeket, melynek megnézve értéke FALSE. Esetünkben egy ilyen elem sincs.



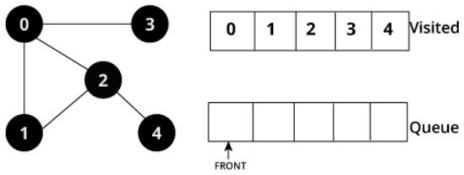
A következő lépésben a 2-es VERTEX-et írjuk ki, illetve a SORBA beírjuk a hozzá tartozó olyan elemeket, melynek megnézve értéke FALSE. Esetünkben ez a 4-es elem.



A következő lépésben a 3-as VERTEX-et írjuk ki, illetve a SORBA beírjuk a hozzá tartozó olyan elemeket, melynek megnézve értéke FALSE. Esetünkben egy ilyen elem sincs.



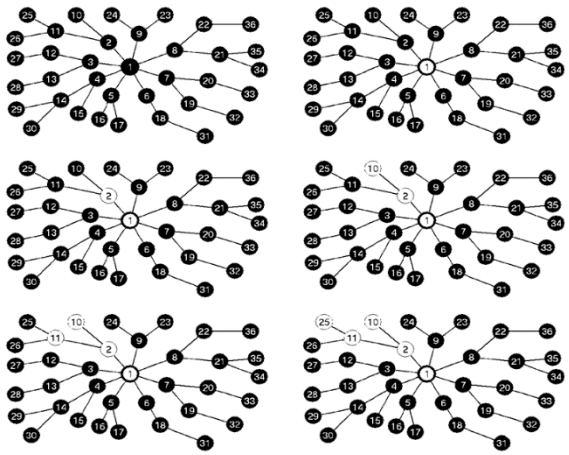
A következő lépésben a 4-es VERTEX-et írjuk ki, illetve a SORBA beírjuk a hozzá tartozó olyan elemeket, melynek megnézve értéke FALSE. Esetünkben egy ilyen elem sincs.



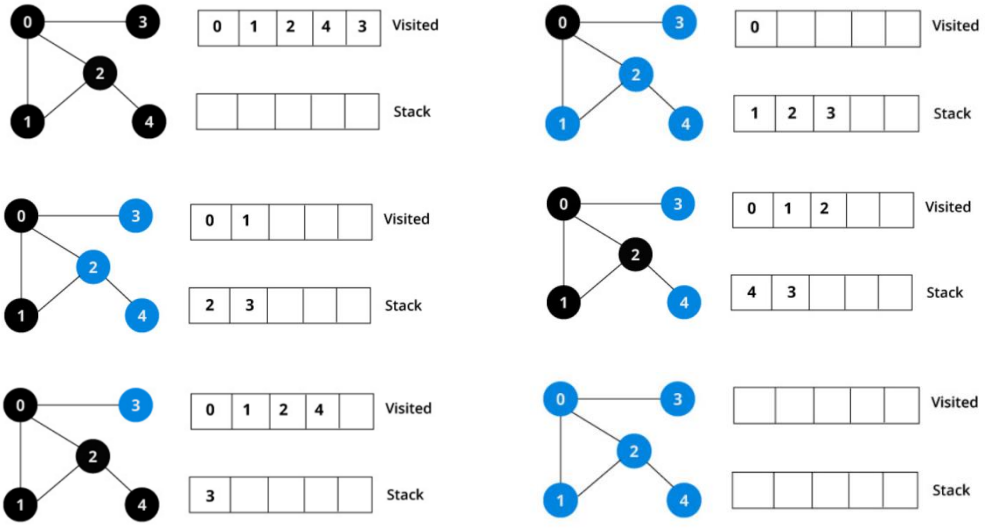
Következő lépés a 4-es VERTEX-et írjuk ki, illetve a SORBA beírjuk a hozzá tartozó olyan elemeket, melynek megnézve értéke FALSE. Esetünkben egy ilyen elem nincs, tehát kiürült a listánk, és leállt a programrész futása.

A szélességi bejárás Python kóddal megvalósítva:

```
# This code is contributed by Neelam Yadav
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):

        # allitsuk be az osszes vertex VISITED cimkejet false-ra
        visited = [False] * (len(self.graph))
        # keszitsunk egy sort
        queue = []
        # Irjuk be az elemet a sorba, es allitsuk be a VISITED cimkejet
        queue.append(s)
        visited[s] = True

        while queue: #amig a sor nem urul ki

            # vegyuk ki a vertexet a sorbol es irjuk ki
            s = queue.pop(0)
            print (s, end = " ")

            # Az osszes olyan szomsedot amit meg nem latogattunk be tegyuk bele a sorba
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True

# keszitsunk egy grafot
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

# jarjuk be a grafot a szelessegi bejarassal
g.BFS(2)
```



A mélységi bejárás szemléltetéséhez is „az öreg városka girbe-gurba utcáin bolyongó kopott emlékezetű lámpagyújtogató” példát vesszük szemügyre. A lámpagyújtogató az első, majd a további lámpák felgyújtása után mindig egy olyan utcán indul tovább, amelyikben még nem lát fényt, és elérve a következő sarokhoz, meggyújtja az ott elhelyezett lámpát. Ha egy lámpa alól körül nézve már minden onnan kiinduló utcából fényt lát, akkor visszamegy arra, ahonnan ide érkezett, és onnan egy még meg nem gyújtott lámpa irányába indul. Ha ilyet nem lát, akkor innen is visszamegy a megelőző sarokra. Egy idő után visszaért eredeti kiinduló helyére, és ekkor már minden innen elérhető köztéri lámpa világít.
A mélységi bejárás is hasonló elven működik, mint a szélességi bejárás. Itt is használnunk kell a megnézve változót. A megvalósításhoz használhatunk vermet vagy rekurzív metódust. A mélységi bejárás során keletkezett útvonalat mélységi feszítőfának is szokás nevezi.
Nézzünk egy példát a mélységi bejárás megvalósítására!



A mélységi bejárás Python kóddal megvalósítva:

```
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        # jeloljuk meg, hogy mar lattuk ezt az eleme, es irjuk ki
        visited[v] = True
        print(v, end = ' ')
        # tegyuk be az osszes meg nem latott szomszedot a verembe
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    def DFS(self, v):

        # allitsuk be az osszes vertex VISITED cimkejet false-ra
        visited = [False] * (max(self.graph)+1)

        self.DFSUtil(v, visited)
```

```
# keszitsunk egy grafot
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

# jarjuk be a grafot a melysegi bejarassal
g.DFS(2)
```

Gyakorló feladatok (45 perc)

1. Készítsünk egy programot melyben a emberek baráti hálózatát ábrázoljuk. A program képes legyen megmondani egy-egy emberről, hogy barátok-e, és kik a közvetlen közeli/közös barátaik.
2. Szemléltessük az előző feladatot az *igraph* csomag segítségével!
3. Szemléltessük, hogy maximum hány „kézfogásra” vagyunk a gráf összes emberétől.

Ajánlott kitekintő anyag (78 perc)

Belgiumi telefonbeszélgetések – Flamand/Vallon **(5 perc)** – Angol nyelvű cikk linkje

Ízek hálózata **(5 perc)** - Angol nyelvű cikk linkje

Szélességi és mélységi bejárás **(18 perc)** – Youtube link angol nyelven

Szélességi bejárás **(50 perc)** – MIT video link angol nyelven

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE