Jacsev Sámuel C0UWDY

Generikus irányítatlan gráf

programozói dokumentáció a javadoc kiegészítéseként

A program írása közben két fő szempontra fordítottam kiemelt figyelmet. Egyik, hogy az osztályokat és a metódusokat igyekeztem a lehető legalacsonyabb láthatósági szintre állítani, hogy mindegyik csak onnan legyen elérhető vagy módosítható, ahonnan valóban szükséges. Másik, az olvashatóság és átláthatóság, ami magába foglalja a kommenteket, a kód konzisztens jellegét, amit persze Ctrl+Shift+F-el kiválóan el lehet érni, de legfőképpen a változónevek beszédes elnevezésére törekedtem.

A Main-en kívül három fő osztályba rendeztem a programot. Egyik természetesen a *Graph* amely egy gráf és tulajdonságaitnak összessége. A gráfot Node csúcsok alkotják, amiket Edge élek kötnek össze.

A **Graph** osztálynak három egyszerű tárolója van:

- *numberOfNodes*, ami a csúcsok számát tartalmazza. Ez minden esetben tudjuk. Amennyiben a felhasználó ad meg szomszédossági mátrixot, úgy tőle megkérdezi a program, ha beolvas fájlból, úgy egy helyes mátrixból egyértelműen meghatározható mondjuk annak első sora alapján.
- *adjacencyMatrix* ami a szomszédossági mátrixot tartalmazza. Mivel tudjuk a csúcsok számát, így lefoglalható egy n×n-es 2D tömb, ami a program működése során sosem változtat méretet. Statikus, egyszerű, indexelhető indexelhető, tehát ez a legjobb, amit használni lehet erre a célra.
- *nodes* tömb, ami a csúcsokat azaz Node objektumokat tárol. Itt is fontos volt hogy indexelhető legyen, hiszen ez alapján szerepelnek benne sorban a csúcsok, mindegyiket a szomszédossági mátrixból kiolvasható sora indexel 0...n-1 -ig.

A **Node** osztálynak négy fő tulajdonsága van. Tartalmazza a csúcs *id*- ját, ami a konstruktor egyetlen paramétere is egyben, és ez indexeli a csúcs helyét a *nodes* tömbben. Van egy *dataBank* ArrayList generikus tárolója, amiben előre nem definiált mennyiségű és paraméterként megadott típusú objektumokat lehet elhelyezni, ezért egy dinamikus tároló kellett. Ehhez hasonló az *edgeList* is, ami *Edge* objektumokat tárol. Persze az élek száma kiolvasható a szomszédossági mátrixból is, de nem logikus minden csúcshoz külön hosszúságú tömböt definiálni. A specifikációban halmaznak képzeltem- és valóban az is volt nagyon sokáig- aztán egy további feature érdekében amit később leírok majd, ArrayListre cseréltem. A *Node* objektum további funkciója, hogy van egy boolean *visited* nevű változója, ami a különböző bejáró algoritmusok használatakor azért előnyös, mert így nem kell új adathalmazt definiálni ennek tárolására.

Az **Edge** osztály kikerülhető lett volna, de szerintem sokat dob a programon, hogy nem a szomszédossági mátrixot használja mindenre. Jelen elrendezésben a mátrix alapján csak létrejönnek a *Node* és *Edge* objektumok példányai, amik önmagukban leírják az egész gráfot. Az élek természetesen Node- okat kötnek össze. Minden *Edge*-nek van egy *from* és egy to mutatója, melyek csúcsokra referálnak, így kicsit láncolt lista érzete van. Az élek saját tulajdonsága a *weight,* ami az élsúlyát tárolja; ezt az értéket 1...9- ig korlátoztam.

Alapvetően minden metódus olyan egyszerű, hogy nem igazán lehet őket jobban kommentálni mint azt a javadocban tettem, de igyekeztem izgalmasabbá tenni a kódot azzal, hogy a BFS-t rekurzívan írtam meg. Ennek röviden leírom a működését.

Ketté kellett bontani, hogy először elkészüljön a *BFSrecursive()* két paramétere; egy queue ami a *nextStart* nevet kapta, és egy ArrayList ami a *path*, azaz bejárt út sorrendjét rögzíti. A külső függvény kap egy kezdőpontot paraméterül, amit belerak a *nextStart*-ba, és innen fog indulni a rekurzió. A belső függvény meghívás után kiveszi a *nextStart*-ból a csúcs azonosítóját és beleteszi a *path*-ba aztán átállítja az adott *Node visited* változóját *true*-ra. Megkeresi a csúcsot a *nodes* tömbben, végig nézi az éllistáját, és amennyiben talál olyan Node-ot ami még nem volt bejárva, úgy azt belerakja a *nextStart* sorba. Innen meghívódik a *BFSrecursive()* újra ugyanazokkal a paramméterekkel, csak közben a *nextStart* első helyezettje más lett, és a *path* lista sem üres már. A rekurzió akkor ér véget, amikor kiüresedik a *nextStart* queue vagy a paraméterül kapott csúcs már be van járva. Ez az érték a csúcs létrejöttekor hamis, és minden algoritmus után ami ezt változtatja, megívható metódus a *resetVisited().*
Fontos, hogy mivel a BFS éllista és nem mátrix alapján fut, ezért alapból az élek az *edgeList*-be kerülésük sorrendjében lennének elérhetőek. Ez ugyan nem befolyásolja az eredeti célt miszerint határozza meg a program hogy a gráf egybefüggő e, de nem a legszebb megoldás. Azt az elvet követem, hogy ha az a neve hogy BFS akkor a metódus ne csak egy másik metódus kiegészítése legyen, hanem önállóan szélességi bejárásként tudjon funkcionálni, ezért elérhetővé tettem a felhasználó számára is. Viszont hogy a működése kiszámítható legyen, rendezni kellett az éllistákat, amire a halmaz nem nyújt megoldást, így lett újra gondolva. A csúcsok éllistái két komparátorral rendezhetőek. Egyik az éleket a *to Node id-ja* alapján rendezi sorba, a másik egyszerűen az él súlya alapján. Így végeredményben a legkisebb *id-val* rendelkezőcsúcs fele vagy legkisebb élsúlyú úton történhet a bejárás.