Biztonságos hálózatrész

Gráf:

A gráfot éllistás reprezentációval valósítottam meg. Konstruktor hívásakor paraméterben megadott egész szám a csúcsok száma, és ennek megfelelő nagyságú tömböt foglalunk le.

Adattagjai:

- **verticesCount:** csúcsok száma
- **edgesCount:** élek száma
- **adjacent:** éllisták tömbje

Metódusai:

- **getVerticesCount():** visszatér az csúcsok számával
- **getEdgesCount():** visszatér az élek számával
- **getAdjacent(int i):** visszatér a megadott csúcsindex éllistájával
- **addEdge(int i1, int i2, double w):** felvesz egy w súlyú i1-ből i2-be vezető élt a gráfba
- **removeEdge(int i1, int i2, double w):** töröl egy w súlyú i1-ből i2-be vezető élt a gráfból, ha van ilyen

Él típus:

Egy célcsúcs indexet illetve egy súlyt tárol. Értelmezve van rá az egyenlő/ nem egyenlő operátor, ami az adattagok teljes egyenlőségét vizsgálja.

Éllista:

Az éllista egy fejelemes, egyirányú, aciklikus láncolt lista, és mindegy egyes eleme egy élt tárol.

Metódusai:

- **addEdge(Edge e)**: A lista elejére szúr egy élt.
- **remove(Edge e):** Megkeresi a listában a megadott élt, ha megtalálja kifűzi a listából és igazzal tér vissza, ellenkező esetben pedig hamissal.

Továbbá értelmezve van az éllista osztályra egy konstans iterátor.

Prioritási sor:

A prioritási sort kupac adatszerkezettel reprezentáltam és template osztályként hoztam létre, hogy bármilyen típussal működőképes legyen. Konstruktor hívásakor meg kell adni a típust, a rendezés relációját és a sor méretét.

Megvalósítottam a prioritási sor ismert metódusait: **Sorból**(*pop*), **Sorba**(*push*), **Első**(*first*), **Üres-e**(*empty*), **Teli-e**(*full*), **Üres**(*clear*). Ezenkívül létrehoztam egy **Módosít**(*modify*) metódust, mely megkapja a keresendő elemet, illetve annak a módosítását. A metódus megkeresi a sorban az elemet, ha megtalálja akkor módosítja és helyreállítja a kupacot.

A kupac adatszerkezetet a következő privát metódusok tartják fent: **Felcsúsztat**(*increase*), **Lecsúsztat**(*decrease*), amik relációnak megfelelően rendezik a kupacot.

Csúcs típus:

A csúcs indexét, szülőjét illetve az elérés megbízhatóságát tárolja. Értelmezve van rá a kisebb/nagyobb operátor ami az elérés megbízhatósága szerint rendez, illetve egyenlő/nem egyenlő operátor, ami az adattagok teljes egyenlőségét vizsgálja.

A feladat megoldása:

Egy kommunikációs hálózatot egy G=(V,E) irányított, élsúlyozott gráffal adunk meg. Minden (u,v) eleme E él rendelkezik egy 0 <= r(u,v) <=1 kapcsolat értékkel, amely az u csúcsból a v csúcsba vezető kommunikációs csatorna megbízhatóságát fejezi ki. Az r(u,v)-t annak valószínűségeként értelmezhetjük, hogy az u-ból v-be vezető csatornán az adat nem sérül meg, és feltesszük, hogy ezek a valószínűségek függetlenek.

A feladatot megoldásához a **Dijkstra** algoritmus használható, az alábbi módosításokkal: (A továbbiakban a Dijsktraban használt távolságra megbízhatóságként fogok hivatkozni)

- A kezdő csúcs megbízhatósága 0-ról 1-re állítása
- Az összes többi csúcs megbízhatóságát végtelenről 0-ra állítása
- A prioritási sornál min sor helyett max sor választása és új megbízhatóság meghatározásánál ’kisebb’ helyett ’nagyobb’ reláció használata
- Új megbízhatóság számítása összeadás helyett szorzással

További optimalizálási módosítás: az algoritmus elején csak a kezdőcsúcsot rakjuk be a sorba, majd amikor egy csúcshoz érve új megbízhatóság értéket kapunk, akkor azt is berakjuk, később pedig csak módosítjuk.

Dijkstra(G, s)

$\forall v \in G.V$		$Műveletigény$:
$v.d := 0, v.p := NIL, kész[v] := \downarrow$		$\Theta(n)$
$s.d = 1$		$\Theta(1)$
$Q : MaxQ$		$\Theta(1)$
$Q.sorba(s)$		
$\neg Q.üres\_e()$		"$\lvert V \rvert := n - szer$"
$u := Q.sorból(), kész[u] := \uparrow$		$O(n * \log_2 n), O(n)$
$\forall v \in G.Adj[u]$		"$\lvert E \rvert := e - szer$"
$kész[v]$	–	$O(e)$
$v.d < u.d * w(u,v)$	–	$O(e)$
$v.d = 0$		$O(e)$
$v.d := u.d * w(u,v)$	$v.d := u.d * w(u,v)$	$O(e)$
$v.p := u$	$v.p := u$	$O(e)$
$Q.sorba(v)$	$Q.helyreállít(v)$	$O(e * \log_2 n)$

$T(n) = O(n + 1 + 1 + n * \log_2 n + 5 * e + e * \log_2 n) = O((n + e) * \log_2 n))$

Tehát összesen a műveletigény: $O((n + e) * \log_2 n)$

Megjegyzés: A helyreállítás függvényem tartalmaz egy lineáris keresést, így művelet igénye: $O(\log_2 n + n)$
Ez a függvény $e - n + 1$-szer lesz meghívva a programban (ha $e < n$ akkor pedig egyszer sem). Ez ritka gráfok esetén elenyésző, így az műveletigény számításnál kihagytam, viszont sűrű gráfok esetén $O(n^2 * \log_2 n)$ -ről n^3 -re ronthatja az algoritmust. Sűrű gráfok esetén érdemes az algoritmust rendezetlen tömb + szomszédsági mátrix reprezentációval megírni.