

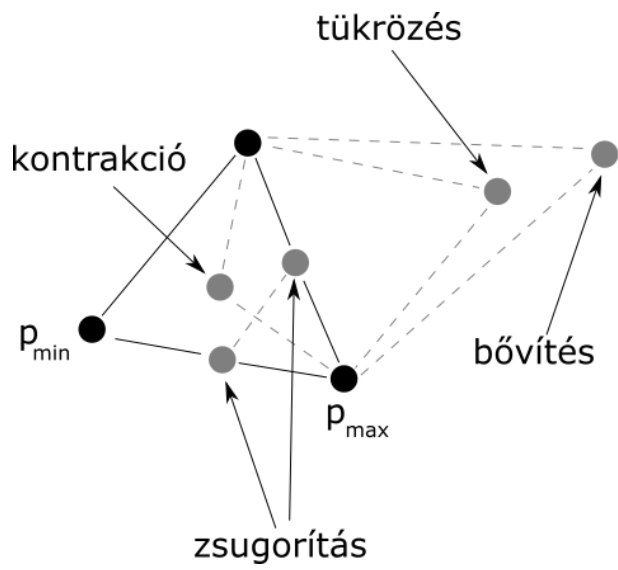
---

## A. Mellékletek

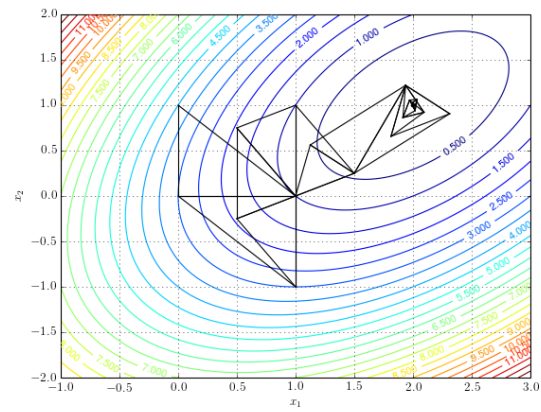
### A.1. Nelder-Mead algoritmus

Mivel a szivárgási a kiömlési együttható függvénye, ezért az optimalizálandó függvény jelen esetben egyváltozós, folytonos és egyetlen minimummal rendelkezik. A Python programozási csomagban szerencsére sok lokális optimalizáló algoritmus található. Az ilyen függvényekre legjobbnak mutatkozott a `scipy.optimize` csomagon belül a Nelder-Mead [9] módszer.

Ez egy gyakorta alkalmazott numerikus módszer, amely többváltozós függvények optimalizációjára használatos. Lényege, hogy a keresési tér  $n$ -dimenziós voltának esetén a legegyszerűbb geometriai alakzatot, azaz egy  $n + 1$ -dimenziós szimplexet hoz létre. Ez egydimenziós esetben egy szakasz, kétdimenziós esetben egy háromszög, háromdimenziós esetben pedig egy tetraéder képzését jelenti a keresési tér egy véletlenszerűen kiválasztott pontjából kiindulva. Ez a kiindulási pont lesz a kezdeti feltétele az algoritmusnak. Az algoritmus működése egy iterációs eljárás alapul, amely először a szimplex minden pontjában kiszámítja a függvény értékét. Ezek után megnézi, hogy melyik a legrosszabb értékkel ellátott pont és középpontosan tükrözi azt a többi,  $n$  darab pontból álló alakzat pontjainak közös centroidjára. Ha a tükrözés után a kapott pont értéke jobb mint az eddigi legjobb függvényérték, akkor a tér ezen irányában a centroidból az új pontba mutató vektort egy egynél nagyobb skaláris együtthatóval megszorozza, azaz bővíti. Ha az új pont függvényértéke jobb mint a tükrözötté, akkor ezzel a ponttal újabb szimplexet képez, ha viszont nem, akkor a tükrözött ponttal képezi ugyanezt. Ellenkező esetben, azaz ha a tükrözött pont értéke rosszabb mint az eddigi legjobb függvényérték, akkor a tükrözött pont irányába mutató vektort egy  $] - 1, 1[$  tartományban lévő együtthatóval szorozza, ezáltal újabb pontot képezve. Ha az új pont függvényértéke jobb mint a tükrözötté, akkor ezzel a ponttal képi az újabb, összehúzott szimplexet, ha viszont nem, akkor a tükrözött ponttal képezi azt. Ha egyik megoldás sem vezet jobb eredményre, akkor a zsugorítás következik, amely a rosszabb értékű pontok felől a legjobb pont felé kicsinyíti az alakzatot. A szimplex-képzési módszereket és a folyamatot a 21. ábra mutatja be.



(a) Szimplex képzési metódusok



(b) Az algoritmus folyamatábrája

21. ábra. A Nelder-Mead algoritmus működésének szemléltetése

## A.2. Genetikus algoritmus

E módszer a természettudományok számtalan területén használt algoritmus-fajta. A technika a természetes kiválasztódás folyamatából inspirált, amely az evolúciós algoritmusok (EA) nagyobb osztályába tartozik. A genetikus algoritmusok általában arra használatosak, hogy kiváló minőségű megoldásokat hozzanak létre az optimalizálási és keresési problémákra olyan biológiai ihletésű operátorokra támaszkodva, mint a mutáció, a keresztezés és a szelekció. A Python programozási nyelv SciPy moduljának részét képezi egy genetikus algoritmos bővítmény is, amely szabadon implementálható. Az eljárást ebben a modulban futtattam, Ubuntu környezetben.

A genetikus algoritmus a Darwini evolúciót ülteti át algoritmizált formába. Alapelve, hogy az életképesebb egyed nagyobb valószínűséggel adja tovább génjeit mint egy kevésbé megfelelő példány. A természetben ez a természetes kiválasztódás útján történik, amelyet számtalan tényező befolyásolhat, mint a fizikai tulajdonságok, az intelligencia szintje, a tűrőképesség, és még sok más egyéb tényező. Ezen körülmények tehát tekinthetők irányító tényezőnek az evolúció során. A genetikus algoritmus ezeket a tulajdonságokat és tényezőket használja fel a mechanizmusainak kiépítésére.

A GA-nek egy- vagy több célfüggvény minimalizálására is alkalmas fajtái vannak. Jelen esetben az egyetlen célfüggvényre használt modulja került alkalmazásra, ez pedig a számított szivárgás és kieső fogyasztói hányad lineáris kombinációja volt.

Minden genetikus algoritmus legfontosabb bemenő (és kijövő) eleme a célfüggvény. Az ez által képzett keresési térből pontokat vételezve és ezeknek értékeit visszaadva hozza létre a **populációt**. Ennek mérete lehet állandó vagy változó, de alapvetően meghatározza, hogy egy adott lépésben az algoritmus hány különböző megoldást kezel és értékel ki. Ez megfelel az egyedszám definíciójának, amelyet az algoritmus az öröklődés szimulálására használ. Minél nagyobb ez a szám, annál több esély van a legjobb eredmény elérésére egy generáción belül, viszont bővítése a függvényhívások számának növekedését vonja maga után mellyel a futási idő is növekszik, éppen ezért fontos megtalálni a legjobb arányt a kettő között.

A populáció minden esetben egy generációt alkot, amelyet minden iterálásnál létrehoz az algoritmus. A generációk számát a **maximális iterációszám** paranccsal lehet változtatni. Minél nagyobb ez az érték, annál több ideje lesz a populációknak a legjobb megoldást tartalmazni, azonban a futási idő is megnőhet egy túl nagyra választott szám-

nál, holott már a konvergencia korábban megvalósult. Erre az esetre a Python létrehozta a **maximális javulás nélküli iterációszám** beállítást, amely a konvergencia megakadása esetén megakadályozza ezen eset kialakulását. Ha az adott megoldás már az ebben a paraméterben megadott számszor a populáció legjobb egyede volt, akkor leállítja az algoritmust.

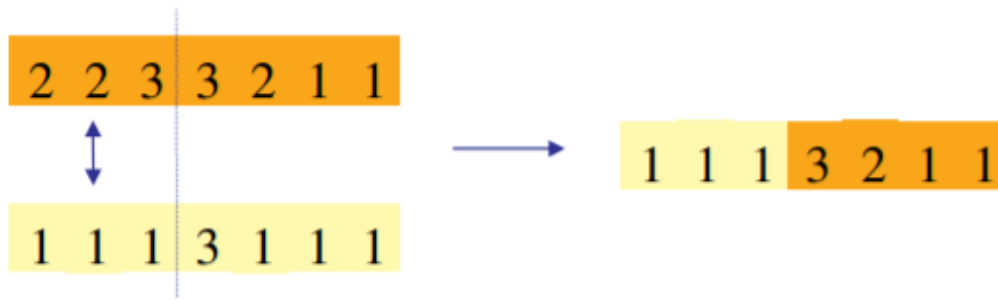
A populációk generálásánál a módszer minimálisan eltér a várttól, mert a genetikus sodródás elkerülése végett kívülről új egyedeket hoz be. Erre módot a **szülőarány** paraméter ad, amely a szülők részarányát határozza meg a következő generáció populációjában. Ennek növelésével bennfentesebb, lokálisan jól konvergáló generációkat, míg csökkentésével sokkal diverzebb és ezáltal globális optimumkeresésre jobban alkalmas generációsorokat lehet létrehozni. Ezt úgy érdemes elképzelni, mintha bizonyos eséllyel bizonyos populációs egyedek utazhatnának az időben és úgy generálhatnának utódokat.

A GA világában minden egyednek egyedi kromoszómái vannak. Azonban a diverzitás bővítése végett ezen kromoszómák monoton, generációkon átívelő ismétlődése helyett az algoritmus **mutációt** alkalmaz, amely a genetikai anyag megváltoztatásának felel meg. Ezzel a kromoszómák véletlenszerű változtatásán keresztüli diverzitás-növeléssel nagyobb keresési teret tud megnyitni.



22. ábra. A mutáció sematikus szemléltetése

A keresés diverzitását tovább növelve a kromoszómákat szokás keresztezni is, ezt teszi meg a **keresztezési valószínűség** paraméter. Minél nagyobb az értéke, annál nagyobb az esély a két szülő keresztezésének esélyére, azaz két egyed kromoszómáinak keverésére. Ez a Python által kiépített modulban háromféleképpen is megvalósítható: egyrésztől egy pontos keresztezés útján, ekkor a szülők kromoszómái által alkotott vektort az algoritmus kettévágja és a két rész összetételéből alkot egy utódot.



23. ábra. Az egypontos keresztezés szemléltetése

Ugyanezt meg lehet tenni kétpontos formában is, ezt ábrával most nem szemléltetem. A harmadik, leginkább diverz keresési tér letapogatást eredményező eljárás az egyenlő, azaz **'uniform'** névre hallgató. Ebben az algoritmus a vektorból véletlenszerűen kiválaszt pár felcserélendő kromoszómát, amelyeket az új egyedben már felcserélve fog megjeleníteni.



24. ábra. Egyenlő keresztezés szemléltetése

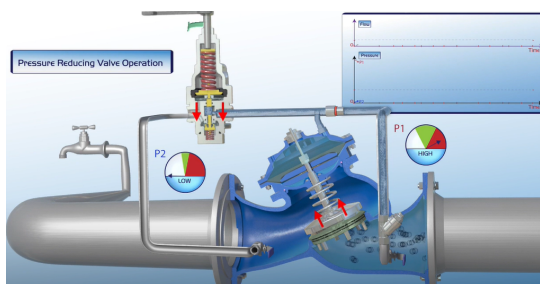
A beállítások sorát az **elitarány** zárja. A Pythonban alkalmazott genetikus algoritmus alapvetően elitista megközelítésű, ennek a paraméternek a kihagyása egy teljesen hagyományos GA-t adna vissza. Az elitista megközelítés lényege, hogy minden generációban az adott populáció legjobb egyedei, azaz elit rétege kromoszómaváltoztatás nélkül átkerül a következő generációba, ezzel elősegítve a konvergenciát. Értéke a populáció méretétől függ, például 100 egyed esetén 0,1-es elitérték 10 elitet jelent.

### A.3. Nyomáscsökkentő szelepek

A nyomáscsökkentő szelepek (angolul Pressure Reduction Valves, a továbbiakban pedig PRV-k) nevükből adódóan a nyomás csökkentésére alkalmazható áramlási ellenállások. Fő feladatuk a nyomásingadozások kiegyenlítése és a stabil alapnyomás biztosítása a hálózatokban.



(a) Valós kinézet



(b) Vázlatos felépítés

25. ábra. A Bermad cég forgalomban lévő 'Bermad 720 ES' nyomáscsökkentő szelepje

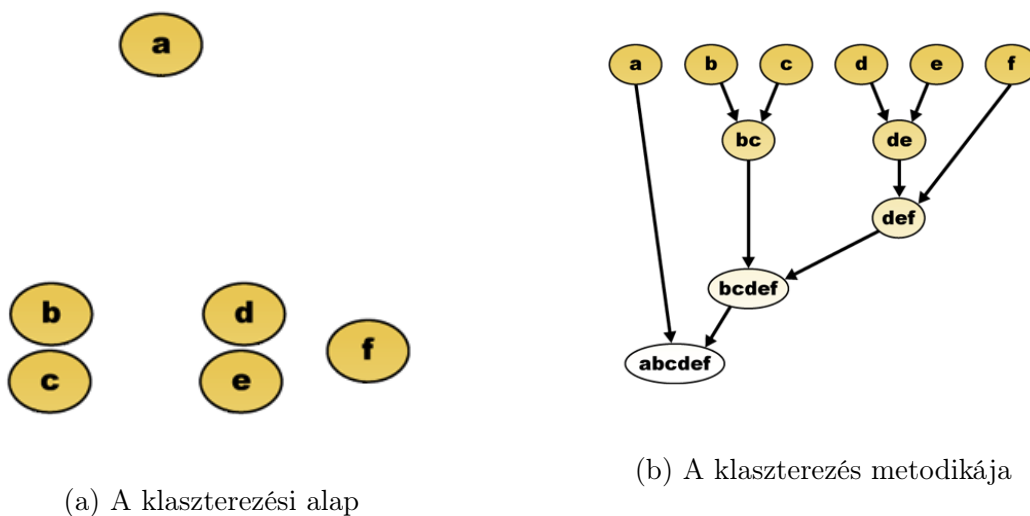
A PRV-k egy alsó és egy felső házból tevődnek össze. Az alsó ház a ki- és bemenő térfogatáramot tudja szabályozni, a felső vezérlőház pedig egy mintavételezési mechanikát foglal magába. Maga a szerkezet általában a beépítését tekintve célszerűen olyan helyeken van, amelyek könnyen hozzáférhetőek. Ilyenek például a csomópontok, hiszen a legtöbbnél vagy valamilyen tolózármechanika, vagy akna került beiktatásra, amely könnyebbé és olcsóbbá teszi a PRV-k beszerelését. Mivel a szelepek beépítése alapvetően csövekbe történik, és minden cső két csomópont között fut a hálózati modellben, ezért a szelepek lehelyezését a csövek elején vagy végén célszerű megtenni. Az alsó házban a bemenő és a kimenő térfogatáramot egy szeleptányér emelésével lehet szabályozni. A felső házban történik ennek az irányítása. Először a beállító csavar manuális elforgatásával beállítható a fenti rugó merevsége, amely így a vezérlődugattyú mozgására képes. A vezérlődugattyú lefelé nyomásával a betörő víz nyomáscsökkenést idéz elő a szelep felső vezérlőkamrájában, amely így felnyitja a rugós mechanizmussal ellátott szeleptányért és megindul az áramlás a kifelé menő ágban is. A nyomáscsökkentő szelep fő előnye, hogy az első beállítás után

stabil nyomástartományban tudja tartani az alsó ágot bármilyen fogyasztó igény jelentkezése esetén. Nem igényel különösebb karbantartást, működése megbízható. Beállításához legtöbbször mindössze egy csavarkulcsra van szükség, a megfelelő nyomástartományba hozásához pedig a gyártási katalógusban található berendezési jelleggörbére.

A nyomáscsökkentő szelepek széleskörű elterjedtsége így esélyt ad az azokkal való topológia optimalizációra. Munkámban megpróbálom az ivóvízhálózatok felépítését nyomáscsökkentő szelepekkel kiegészíteni abban a reményben, hogy az így kialakult nyomástérkép szivárgáscsökkenéshez vezet, amellet, hogy a fogyasztók a korábbival identikus módon kiszolgálásra kerülnek.

## A.4. Leiden-algoritmus

A klaszterezési eljárások alapvetően két csoportra bonthatóak: hierarchikus és nem hierarchikus elemzési fajtákra. Az itt bemutatott klaszteranalízis algoritmusok legnagyobb része Python-implementációja alapján tájékozódttam, a [1] forrásból. A hierarchikus klaszterezési eljárások között kettőt különítünk el: az agglomeratív, azaz összeadáson alapuló módszert és a divizív, azaz felosztáson alapulót. Előbbi sokkal gyakoribb, mivel alulról sokkal könnyebb haladni és összevonni, mint felülről kiválasztani a megfelelő osztódási csoportokat. Az agglomeratív módszer először minden egyes elemet külön klaszterként definiál, innen pedig az elemek valamilyen tulajdonságára támaszkodva egyre nagyobb és nagyobb csoportokat létrehozva csökkenti a klaszterszámot amíg el nem éri a kiinduló állapotot, azaz egy klaszterbe nem sorolja az összes elemet. Ilyen eljárás például az egyszerű láncmódszer [15], amely két klaszter közötti távolságot a két legközelebbi elem távolságával számolja, a Ward-féle varianciamódszer [10], amely azokat az elemeket vonja össze, melyeknél az összevonás során a legkisebb lesz a belső szórásnégyzet növekedése, de a centroidmódszer [12] is, ami a két klaszter közötti távolságot a klaszterek centroidja alapján határozza meg. A klaszterezés eredménye a folyamatnak az az eredménye, amely a felhasználó által megadott klaszterszámhoz legközelebbi megoldást biztosítja. Az agglomeratív módszer éppen ezért könnyen ábrázolható dendrogramon, melyet a 26. ábra szemléltet.



26. ábra. Agglomeratív klaszterezési technika szemléltetése

A másik, nem hierarchikus klaszterezési fajták közül egyértelműen a K-Means algoritmus [4] a legnépszerűbb, amelynek működését a ?? ábra szemlélteti. Működése során az adathalmazba az általunk kiválasztott klaszterszámnak megfelelő random pontot generál, amelyek középpont szerepet betöltve magukhoz csoportosítják a legközelebb hozzájuk lévő pontokat. Ezek után a képzett klaszterek centroidjának kiszámítása következik, amelyet új középpontként használ az elemek újraklaszterezésére. Az iteráció addig folytatódik, amíg több lépés már nem elérhető, azaz az újraklaszterezés nem ad az előzőtől különböző eredményt.

Az ivóvízhálózatok klaszterezésére a szakirodalomban nem találtam bejáratott módszert. Ezért a legcélszerűbbnek a Python csomagban elérhető beépített algoritmusok használatát láttam. Itt több is szóba kerülhetett volna, azonban az átfogó tesztek után legmegfelelőbbnek a K-Means és a spektrális klaszterezés bizonyult. A teszteket a Sopron közeli Balf város ivóvízhálózatán kezdtem, amely leginkább egy fa gráfra hasonlít. Ennek nyomástérképe és gráfmodellje a 27. ábrán látható.



27. ábra. Balf városának gráfmodellje és nyomástérképe

A klaszterezés nehézsége jelen esetben abban gyökerezik, hogy nincs egyértelműen megállapítható következetesség a klaszterek száma és az elvágtott élek között. Miután az agglomeratív klaszterezési eljárások 1 és 50 közötti klaszterszámnál is általában nagyon általános megoldásokat adtak (például a medence kivezetésére helyezendő nyomáscsök-

kentő szelep vagy egy messzi, kieső csomópont levágása) ezért használatukat kizártam. A nem-hierarchikus metódusok először ugyan ígéretes eredményeket mutattak, hiszen nem a triviális megoldásokat találták meg, azonban az algoritmusok kezdő lépésének számító véletlenszerű pontgeneráció miatt nem adtak stabil megoldást. Ugyan előnyük volt, hogy a klaszterszám definiálása megtehető volt, mégsem bizonyultak elég jó megoldásnak a genetikus algoritmus értékeihez képest.

Emiatt újabb eljárások után kellett néznem. Miután a fő célom az volt, hogy a kiválasztott eljárás homogén nyomáshozonát hozzon létre és ezen keresztül csökkentse a szivárgást, ezért célszerűnek láttam nyomásokkal súlyozni az adathalmazt. Ez azt jelentette, hogy a szivárgási modellben kiszámított csomópontokhoz tartozó nyomásokból képzett valamilyen értéket a klaszterező algoritmusnak adott gráf topológiailag megfelelő pontjához rendelve súlyként funkcionált.

A csomag **igraph** modulja számtalan, gráfokkal kapcsolatos programot tartalmaz. Az egyik ilyen a Leiden-algoritmus [21], amely a többszintes klaszterezésnek, másnéven Louvain-algoritmusnak a stabilabb és megbízhatóbb verziójaként van említve a szakirodalomban. Ez egy agglomeratív eljárás, tehát a klaszterszámot folyamatosan csökkentve dolgozik. Az algoritmus felépítésének célfüggvényeként a rendkívül ismert és sok helyen használt modularitást használja. A modularitás egy olyan célfüggvény, amely a gráf legmegfelelőbb klaszterezésével kapott közösségek élszámainak várható számát adja meg. A Leiden-algoritmus megpróbálja maximalizálni a különbséget a klaszter éleinek tényleges száma és az ilyen élek a modularitás szerint várható száma között. Ezt a (18) egyenlettel teszi, amely a modularitást adja meg.

$$\mathcal{H} = \frac{1}{2m} \sum_c (e_c - \gamma \frac{K_c^2}{2m}) \quad (18)$$

ahol  $e_c$  -vel jelöljük az élek tényleges számát a klaszterben. Az élek várható száma  $\frac{K_c^2}{2m}$  -ként fejezhető ki, ahol  $K_c$  a  $c$  klaszter csomópontjainak összege,  $m$  pedig a hálózat összes éleinek száma. Bár eredetileg a modularitásra lett definiálva, a Leiden algoritmus más függvények optimalizálására is használható. Egy alternatív célfüggvényként a Konstans Potts-Model (CPM) amely leküzdí a modularitás bizonyos korlátait. A CPM a következőképpen van definiálva:

$$\mathcal{H} = \sum_c [e_c - \gamma \binom{n_c}{2}] \quad (19)$$

ahol  $n_c$  a csomópontok száma a  $c$  klaszterben.

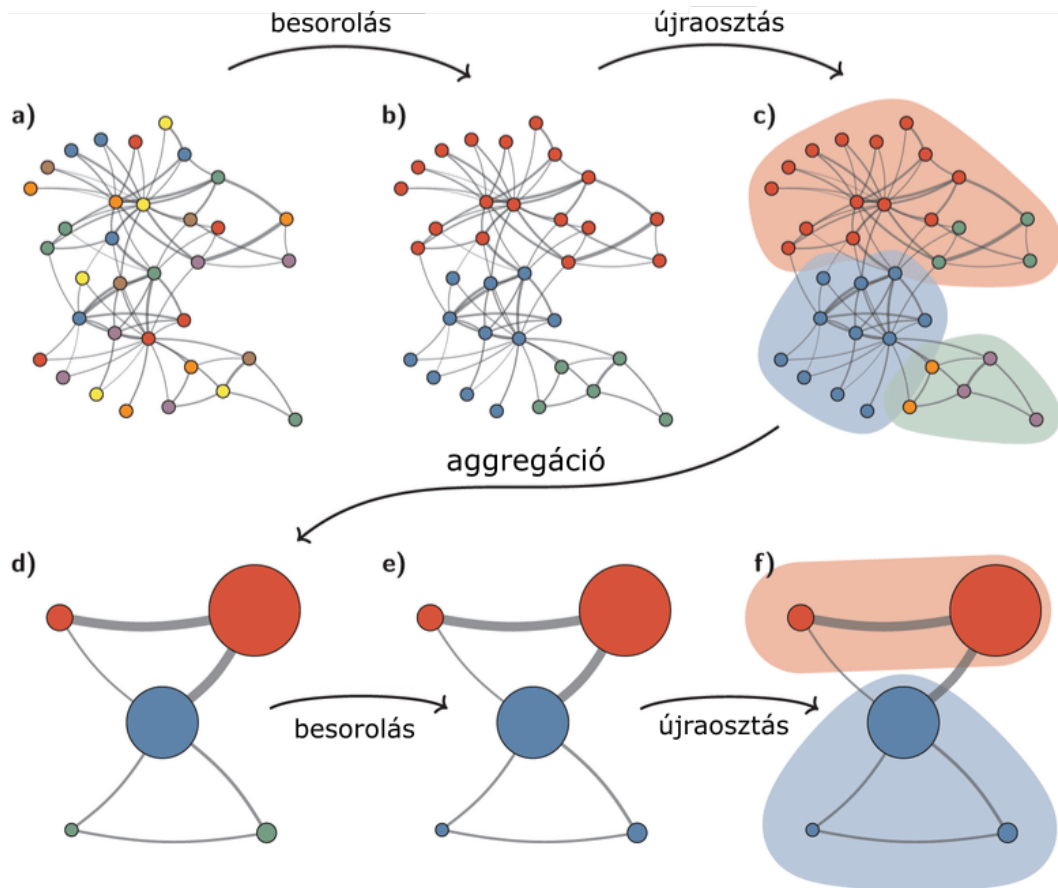
A  $\gamma$  felbontási paraméter értelmezése meglehetősen egyszerű. A paraméter egyfajta küszöbként funkcionál: a közösségek sűrűségének legalább  $\gamma$ -nak kell lennie, míg a közösségek közötti sűrűségnek  $\gamma$ -nál alacsonyabbnak kell lennie. A felbontási paraméter az (18) és (19) egyenletekben kitüntetett szerepet kap, hiszen értékének változtatásával kapható több vagy éppen kevesebb klaszter. Jellemzően kisebb felbontási paraméter értékek kevesebb, nagyobbak pedig több klaszter formálásához vezetnek.

A Leiden-algoritmus három fázisból áll: elsőként a csomópontok helyi klaszterezése kerül bevezetésre, ahol az agglomeratív algoritmusokhoz hasonlóan a pontok valamilyen tulajdonságuk alapján kerülnek klaszterekbe sorolásra. Miután ezen algoritmusnál lehetséges mind a pontok, mind pedig az élek súlyozása, lévén jelen problémakörben a nyomás a szivárgás fő okozója, súlyként valamilyen formáját használva számomra megfelelőbb irányba tudom eltolni a klaszterek kialakítását.

Második lépésként a partíció finomítása következik, amely igazán különlegessé és érdekessé teszi a Leiden algoritmust, hiszen ezzel a lépéssel elkerüli a triviális megoldások létrejöttét. Az első lépés után létrejött egy előzetes klaszterezés, amelyet a célfüggvényének megfelelően hozott létre az algoritmus a beállított gráfsúlyokat is figyelembe véve. Legyen ez a megoldás a  $\mathcal{K}$ . A Leiden-algoritmus ez után egy véletlenszerű faktort visz bele a klaszterezésbe, amelynél definiál egy  $\mathcal{K}_{fin}$  megoldást is, amely egy teljesen új klaszterezésből kialakulva minden pontot saját klaszterébe oszt be. Ezek után a  $\mathcal{K}$  megoldást alapul véve lokálisan egyesíti a csomópontokat a  $\mathcal{K}_{fin}$  megoldásban: a csomópontok, amelyek önállóan vannak egy klaszterben a  $\mathcal{K}_{fin}$  megoldásban, összevonhatók egy másik klaszterrel. Fontos, hogy az egyesítéseket csak a  $\mathcal{K}$  partíció klaszterein belül hajtja végre és csak azon pontoknál vizsgálja ennek lehetőségét, amelyeknél a partícióbontás célfüggvényérték-növekedést eredményezne, ezáltal gyorsítva az eljárást. Ezen kívül egy csomópont csak akkor kerülhet be egy klaszterbe, ha mindkettő kellően jól kapcsolódik azok  $\mathcal{K}$ -ban definiált klaszteréhez. A csomópontok helye természetesen csak a célfüggvény értékének növekedése esetén változtatható meg. Azonban lévén rendkívül sok ilyen megoldás előállhat, a legnagyobb célfüggvény-érték emelkedéssel járó partíció-változtatás kiválasztása végeredményében nem biztos, hogy a célnak legmegfelelőbb klaszterezést eredményezné. Éppen ezért a partíció-változtatási eljárásokat az algoritmus a függvényérték változtatásának mértékében egy  $\theta$  faktorral súlyozva, de véletlenszerűen választja ki. Így ugyan a

partíciós keresési teret bővítette, de a megoldási valószínűséget a legnagyobb célfüggvény-érték növekedés felé tolja el. A Leiden algoritmus folyamatábráját a 28. ábra szemlélteti.

Harmadik lépésként az aggregált klaszter létrehozása történik, amely aztán a következő iteráció első lépése lesz. A finomítási és aggregációs szakaszok befejezése után az aggregált  $\mathcal{K}$  klaszterezés gyakran klaszterszám-növekedést szenved. Ez ugyan rossz és felesleges iránynak tűnhet, azonban lévén egy véletlen faktort is integrálva bővíti a partícionálási teret, ezáltal megelőzve a túl triviális megoldások kialakulását, amelyek a korábbi agglomeratív algoritmusoknál a medencére és a távoli csomópontokra feleslegesen lehelyezett nyomáscsökkentő szelepeket okozták. Éppen ezért az algoritmus kiválóan alkalmas volt ivóvízhálózatok gráfmodelljének szivárgás optimalizációs szempontból való klaszter-analízisére.



28. ábra. A Leiden-klaszterezés folyamatábrája, forrás: [21]

## A.5. Differenciális evolúció

A differenciális evolúció (továbbiakban DE) nevéből adódóan egy evolúciós megközelítést alkalmazó optimumkereső. Az algoritmus függvénye egy vektor-skalár függvény, amelynek bemenete a kezdeti változókból álló vektor, kimenete pedig a függvény által visszaadott skalár. Maga a differenciális evolúció lévén az evolúciós algoritmusok közé tartozik, a genetikus algoritmushoz nagyon hasonló, azonban fő eltérési pontjait az alábbiakban be szeretném mutatni.

A DE egyik legfontosabb paramétere a GA-hez hasonlóan a populációszám. Ez a legfőbb beállítás, melynek segítségével a keresési tér nagysága definiálható. A populációkon belül egyedek reprezentálják az optimalizációs keresés megoldásait. A DE alapvetően úgy optimalizál egy problémát, hogy a populáció számának fenntartása mellett új egyedeket hoz létre a meglévők megfelelően történő kombinálásával. Ezek után a legalkalmasabb, azaz a célfüggvény értékére legkisebb skalárt visszaadó egyedet örökíti tovább a populációba. Ily módon a populáció az iteráció során egyre jobb és jobb egyedeket tartalmaz, amelyek végül konvergenciához vezethetnek.

Szemléltetésképp legyen  $f$  az a vektor-skalár függvény, amelyet optimalizálni kívánunk. Az algoritmus célja, hogy megtalálja azt az  $\mathbf{m}$  vektorjelöltet, melyre  $f(\mathbf{m})$  a globális minimum értékét veszi fel. Ehhez először a felhasználónak kell biztosítania egy kezdeti egyedekből álló populációt, legyenek ezen egyedek tagjai  $\mathbf{x}$ .

Első lépésként az algoritmus két legfontosabb paraméterét, a  $CR$  keresztezési valószínűség és a  $F$  differenciális súly beállítását kell megtenni. Jellemző értékeik  $CR \in [0, 1]$  és  $F \in [0, 2]$ . Ezek utána a kezdeti egyedek a keresési térben való random elhelyezése következik és megkezdődik az iteráció.

Az iterációs lépésben Minden egyes  $\mathbf{x}$  egyedre kiválaszt a populációból három további egyedet, legyen ezek  $\mathbf{a}$ ,  $\mathbf{b}$ , és  $\mathbf{c}$ . Ezek után véletlenszerűen kiválaszt az  $R$  listából egy indexet, ahol  $R=1,2,...,n$  és  $n$  az egyedek dimenziójának száma. Ezek után a kijelölt  $\mathbf{x}$  egyedből megkezdődik az utódjának ( $\mathbf{y}$ )-nak képzése.

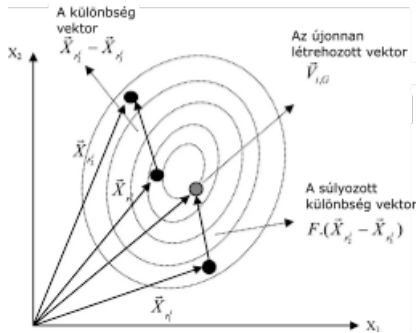
Ehhez először egy  $i$  index választása szükséges, ez a  $i=1,2,...,n$  listából választható és amellyel az iterációt az algoritmus génenként végezni fogja. Az  $i$  index az egyedek vektorában lévő elemek számozására használatos, illetve biztonsági funkciót lát el, amely a keresztezési paraméter zérusnak választásával kap csak kiemelt szerepet. Ezen kívül következik még egy véletlenszerűen generált  $[0, 1]$  közötti  $r_i$  szám képzése. Ennek jelentősége

a differenciálképzésben van, mivel ha  $r_i < CR$ , illetve ha  $i = R$ , akkor az  $\mathbf{y}$  egyed  $i$ . génje a következőképpen áll elő:

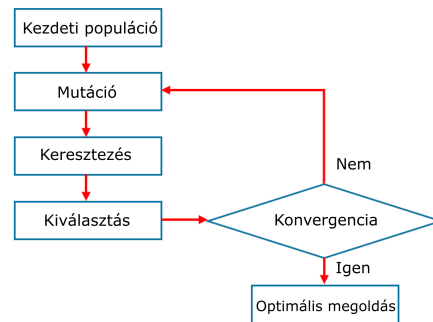
$$y_i = a_i + F * (b_i + c_i) \quad (20)$$

Ezt szemlélteti a 29a ábra is. A (20) egyenletben kiemelt szerepet kap  $F$  paraméter, hiszen lévén vektorok képzését valósítja meg, ezen paraméter értéke szabja meg a keresési térben való irányokban való haladás nagyságát. A  $R$  érték szerepe pedig az, hogy a keresztezési valószínűség nagyon kicsi értékei esetén is legyen változatosság a vektorokban. Egyéb esetben, tehát ha  $CR < r_i$  vagy  $i \neq R$ , akkor az adott gén továbbörökitése történik ( $y_i = x_i$ )

Az iterációs eljárás akkor fejeződik be, amikor az eredeti  $\mathbf{x}$  és az eljárással kapott  $\mathbf{y}$  egyedek függvényértékeinek összehasonlítása megtörtént. Ha  $f(\mathbf{y}) \leq f(\mathbf{x})$ , akkor a populációban  $\mathbf{x}$ -et  $\mathbf{y}$ -ra cseréli, ellenkező esetben  $\mathbf{x}$  egyed marad a populáció eleme. Az iterációt az algoritmus addig végzi, ameddig már a fent leírt lépésekkel nem tud újabb egyedet képezni a populációba, azaz a függvény valamilyen minimumhelyéhez ért. Ha ez túl sokáig tartana, akkor lehetőség van a maximális iterációszám megkötésére is. A folyamatot a 29b ábra mutatja be.



(a) Megoldásképzése



(b) Folyamatábrája

29. ábra. Differenciális evolúció bemutatása, forrás: [20]

A szakirodalomban a két legfontosabb paraméterre a leginkább elterjedt alapbeállítások  $CR = 0.9$  és  $F = 0.8$ . A kiértékelést emiatt én is ezen paraméterértékekkel végeztem.