BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Critical node detection problem in complex networks

Abstract

EZ AZ OLDAL NEM RÉSZE A DOLGOZATNAK!

Ezt az angol kivonatot külön lapra kell nyomtatni és alá kell írni!

A DOLGOZATTAL EGYÜTT KELL BEADNI!

Kötelező befejezés:

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

2020 BÉCZI ELIÉZER

ADVISOR:
ASSIST PROF. DR. GASKÓ NOÉMI

BABEŞ–BOLYAI UNIVERSITY OF CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND INFORMATICS
SPECIALIZATION: COMPUTER SCIENCE

Diploma Thesis

Critical node detection problem in complex networks

ADVISOR:
ASSIST PROF. DR. GASKÓ NOÉMI

STUDENT:
BÉCZI ELIÉZER

2020

UNIVERSITATEA BABEŞ–BOLYAI, CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

Identificarea nodurilor critice în rețele complexe

CONDUCĂTOR ŞTIINŢIFIC:
LECTOR DR. GASKÓ NOÉMI

ABSOLVENT:
BÉCZI ELIÉZER

2020

BABEŞ–BOLYAI TUDOMÁNYEGYETEM KOLOZSVÁR
MATEMATIKA ÉS INFORMATIKA KAR
INFORMATIKA SZAK

Szakdolgozat

Kritikus csomópontok meghatározása komplex hálózatokban

TÉMAVEZETŐ:

DR. GASKÓ NOÉMI,
EGYETEMI ADJUNKTUS

SZERZŐ:

BÉCZI ELIÉZER

2020

Tartalomjegyzék

1. fejezet

Bevezető

1.1. Áttekintés

Hálózatok terén nem minden csomópont egyforma fontosságú. A kulcsfontosságú csomópontok keresésével hálózatokban széles körben foglalkoznak, különösképpen olyan csomópontok esetén, melyek a hálózat konnektivitásához köthetők. Ezeket a csomópontokat általában úgy nevezzük, hogy Kritikus Csomópontok.

Kritikus Csomópontok Meghatározásának Problémája (CNDP) egy optimalizációs feladat, amely egy olyan csoport csomópont megkereséséből áll, melyek törlése maximálisan rontja a hálózat konnektivitását bizonyos predefiniált konnektivitási metrikák szerint.

A CNDP számos alkalmazási területtel rendelkezik. Például, közösségi hálók nagy befolyással bíró egyedeinek azonosítása, komputációs biológiában kapcsolatok definiálására jelút vagy fehérje-fehérje kölcsönhatás hálózatokban, smart grid sebezhetőségének vizsgálata, egyének meghatározása védőoltással való ellátásra vagy karanténba való zárásra egy fertőzés terjedésének gátlása érdekében.

A CNDP egy $\mathcal{NP}$-teljes feladat. Adva van egy $G = (V, E)$ gráf, ahol $|V| = n$ a csomópontok száma, és $|E| = m$ pedig az élek száma. A feladat k kritikus csomópont meghatározása, amelyek törlése a bemeneti gráfból minimalizálja a hálózat páronkénti konnektivitását. Az alapján, hogy mit értünk egy hálózat konnektivitása alatt, a CNDP-nak van egycélú illetve többcélú megfogalmazása is.

1.2. Hozzájárulásaink

Ebben a dolgozatban többek között egy bi-objektív megfogalmazásával fogunk foglalkozni a CNDP-nak. Standard evolúciós algoritmusokat fogunk összehasonlítani egymással különböző szintetikus bemenetekre, illetve való világból inspirált bemenetekre, ugyanakkor célunk egy új hibrid algoritmus fejlesztése, melynek eredményei összehasonlíthatók a standard algoritmusok eredményeivel.

Az algoritmusokat Python-ban fogjuk bemutatni, és a NetworkX könyvtárat [Hagberg et al., 2008] fogjuk használni ahhoz, hogy gráfokat tudjunk manipulálni.

Benchmark tesztelés végett egy olyan gráfhalmazt fogunk használni, amelyben 4 alapvető típus jelenik meg, mindegyik a maga jellegzetességeivel.

2. fejezet

Egycélú CNDP

2.1. Páronkénti konnektivitás

Egycélú CNDP esetén a kihívás abban áll, hogy találjunk egy olyan konnektivitási metrikát, amely alkalmazási területtől függően megfelelően leírja egy gráf összefüggőségét. S-el fogjuk jelölni a törlendő csomópontok halmazát, míg az $f(S)$ jóság függvény fogja jellemezni a $G[V \setminus S]$ feszített részgráf összefüggőségét. Ha H-val jelöljük a $G[V \setminus S]$ feszített részgráf összefüggő komponenseinek a halmazát, akkor a jóság függvény a következő képlettel írható le:

$$f(S) = \sum_{h \in H} \frac{|h| \cdot (|h| - 1)}{2}, \tag{2.1}$$

amelyet az irodalom [Aringhieri et al., 2016; Ventresca, 2012] úgy tart számon, hogy **páronkénti konnektivitás**. Tehát a feladat a 2.1 függvénynek a minimalizálása:

$$\min_{S \subseteq V} f(S). \tag{2.2}$$

A 2.1 fitnesz függvény implementációját a 2.1. kódrészlet szemlélteti Python-ban.

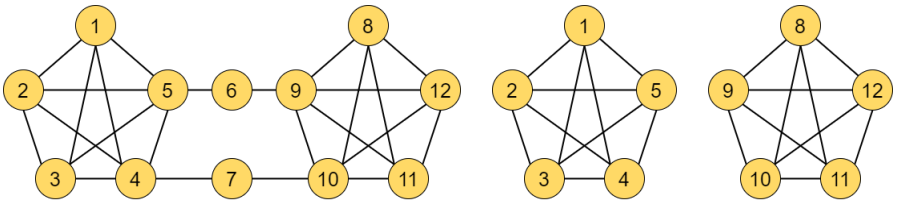
2.1. Listing. Páronkénti konnektivitás

```
def pairwise_connectivity(G):
    components = networkx.algorithms.components.connected_components(G)
    result = 0

    for component in components:
        n = len(component)
        result += (n * (n - 1)) // 2

    return result
```

Egy példa

A 2.1. ábrán látható gráfban, ha $k = 2$ kritikus csomópontot kell azonosítanunk, akkor $S = \{1, 2\}$ eredményezi az optimális megoldást. A $G[V \setminus S]$ feszített részgráf két, egyenként öt csomópontból álló összefüggő komponensre esik szét, vagyis $|H| = 2$. Így a 2.1 jóság függvény a következőképpen számolódik:

$$f(S) = \frac{5 - (5 - 1)}{2} + \frac{5 - (5 - 1)}{2} = 20$$



2.1. ábra. Példa egy kis méretű gráfra (bal oldalt), amely a 6. és 7. csomópontok törlése után szétesik két összefüggő komponensre (jobb oldalt).

2.2. Mohó algoritmus

2.2.1. Általánosan

Egy mohó algoritmus egy egyszerű és intuitív algoritmus, amely gyakran használt optimalizációs feladatok megoldására. Az algoritmus helyi optimumok megvalósításával próbálja megtalálni a globális optimumot.

Habár a mohó algoritmusok jól működnek bizonyos feladatok esetében, mint pl. Dijkstra-algoritmus, amely egy csomópontból kiindulva meghatározza a legrövidebb utakat, vagy Huffman-kódolás, amely adattömörítésre szolgál, de sok esetben nem eredményeznek optimális megoldást. Ez annak köszönhető, hogy míg a mohó algoritmus függhet az előző lépések választásától, addig a jövőben meghozott döntésektől független.

Az algoritmus minden lépésben mohón választ, folyamatosan lebontva a feladatot kisebb feladattá. Más szavakkal, a mohó algoritmus soha nem gondolja újra választásait.

2.2.2. Saját mohó algoritmus

A mohó algoritmus kiindul a gráf csúcslefedéséből. [1] Ez lesz a kezdeti S megoldásunk. A maradék csomópontok $V \setminus S$ a gráf maximális független csúcshalmazát [2] *MIS* alkotják. Mivel majdnem biztos, hogy a megoldásunkban több, mint k csomópont lesz, ezért mohón elkezdünk kivenni csomópontokat S-ből, majd ezeket hozzáadni *MIS*-hez, amíg $|S| > k$. A hozzáadott csomópont az lesz, amelyiket ha visszatesszük az eredeti gráfba, akkor a minimum értéket téríti vissza a páronkénti konnektivitásra a keletkezett gráfban.

Mivel több olyan csomópont lehet, amelyeket ha visszateszünk az eredeti gráfba, akkor ugyanazt a minimális értéket adják vissza a páronkénti konnektivitásra, ezért ezeket eltároljuk a B halmazban, és minden lépésben random módon határozzuk meg, hogy melyik kerüljön vissza *MIS*-be. Ezzel az eljárással garantáljuk, hogy a mohó algoritmusunk különböző megoldásokat fog adni többszöri futtatások esetén. A CNDP esetén a mohó algoritmust a 2.1 kódrészlet szemlélteti.

1. Angolul: vertex cover.
2. Angolul: maximal independent set.

Algorithm 2.1 Greedy CNP

```
1: function GREEDY(G, k)
2:     S ← VERTEX COVER(G)
3:     while |S| > k do
4:         B ← arg min_{i∈S} f(S \ {i})
5:         S ← S \ {SELECT(B)}
6:     end while
7:     return S
8: end function
```

2.3. Genetikus algoritmus

2.3.1. Általánosan

A genetikus algoritmus a metaheurisztikák osztályába tartozik, és a természetes kiválasztódás inspirálta. Egy globális optimalizáló, amely gyakran használt optimalizációs és keresési problémák esetében, ahol a sok lehetséges megoldás közül a legjobbat kell megkeresni. Azt hogy egy megoldás mennyire jó, a fitnesz függvény mondja meg.

A genetikus algoritmus mindig egy populációnyi megoldással dolgozik. A populációba egyedek tartoznak, amelyek egyenként egyed esetén megoldásai a feladatnak. Az algoritmus minden iterációban egy új populációt állít elő az aktuális populációból úgy, hogy a **szelekciós operátor** által kiválasztott legrátermettebb szülőkön alkalmazza a **rekombinációs** és **mutációs operátorokat**.

Ezen algoritmusok alapötlete az, hogy minden újabb generáció az előzőnél valamelyest rátermettebb egyedeket tartalmaz, és így a keresés folyamán egyre jobb megoldások születnek.

2.3.2. Saját genetikus algoritmus

Egy Genetikus Algoritmus (GA) standard algoritmikus keretrendszerét használjuk fel. Generálunk egy kezdeti populációt megoldásokkal. Utána keresztezzük őket, hogy új megoldásokat kapjunk, amelyeket pedig mutálunk. Ezután rendezzük a régi és új megoldásokat egy fitnesz függvény alapján, és létrehozunk egy új populációt eltávolítva a rossz megoldásokat. A folyamatot addig ismételjük, amíg az iterációk száma el nem ér egy felső korlátot. Az algoritmus végén visszatérítjük a legjobb megoldást. A CNDP esetén a genetikus algoritmust a 2.2 kódrészlet szemlélteti.

Inicializáció

A kezdeti populáció egyedeit random generáljuk ki. Ez azt jelenti, hogy minden egyed kromoszómája egy k csomópontból álló részhalmaza lesz a bemeneti gráf csomóponthalmazának. Ezt szemlélteti a 2.3 kódrészlet.

Egy új fitnesz függvényt vezetünk be egyed esetén egyed jóságának felmérése végett. Ez abban tér el a 2.1 részben tárgyaltaktól, hogy nem csak a páronkénti konnektivitás mértékét vesszük figyelembe egy egyed esetén, hanem hogy az eddigi talált legjobb megoldástól mennyire tér el. Ezt a fitnesz függvényt a

Algorithm 2.2 Genetic Algorithm

```
1: function GA(G, k, N, π_min, π_max, Δπ, α, t_max)
2:     t ← 0
3:     INIT(N, P, S*, γ, π)
4:     while t < t_{t_max} do
5:         P' ← CROSSOVER(k, N, P)
6:         P' ← MUTATION(k, N, P', π)
7:         P ← SELECTION(N, P, P')
8:         S*, γ, π = UPDATE(N, P, S*, π, π_min, π_max, Δπ, α)
9:         t ← t + 1
10:    end while
11:    return P
12: end function
```

következő képlettel írjuk le:

$$g(S, S^*) = f(S) + \gamma \cdot |S \cap S^*|. \tag{2.3}$$

A képletben szereplő S^* jelenti az eddig talált legjobb megoldást. A γ egy változó, amely abban segít, hogy fenntartsuk a változatosságot a populáció egyedei között, megbüntetve azokat, amelyek túl közel vannak a legjobbhoz. A γ változót minden iterációban a következő képlettel számoljuk újra:

$$\gamma = \frac{\alpha \cdot f(S^*)}{\langle |S \cap S^*| \rangle_{S \in P}}, \tag{2.4}$$

ahol a nevező a populáció egyedeinek és a legjobb egyed közötti átlagos hasonlóságot fejezi ki. Az α pedig a képletben található változók egymás feletti fontosságát befolyásolja.

A π paraméter a mutáció valószínűségét fejezi ki egy egyed esetén. Ezt kezdetben $\pi_{\min}$-re állítjuk, de minden iterációban frissítjük aszerint, hogy találtunk-e az új generációban egy olyan megoldást, amely jobb, mint a globális legjobb. Ha találtunk az eddigieknél jobb megoldást, akkor a π értékét $\pi_{\min}$-re állítjuk, különben a $\pi = \min(\pi + \Delta\pi, \pi_{\max})$ képlet szerint növeljük. Ez arra jó, hogy fenntartsuk a populáció sokféleségét abban az esetben, amikor nem tudunk javítani az eddig talált legjobb megoldáson, mindezt úgy, hogy megnöveljük a mutációk kialakulásának a valószínűségét.

Az S^*, γ és π változók frissítését a 2.4 kódrészlet mutatja be.

Algorithm 2.3 Random Solution

```
1: function RAND SOL(k)
2:     S ← V
3:     while |S| > k do
4:         elem ← SELECT(S)
5:         S ← S \ {elem}
6:     end while
7:     return S
8: end function
```

Algorithm 2.4 Update S^*, γ and π variables

function UPDATE($N, P, S^*, \pi, \pi_{\min}, \pi_{\max}, \Delta\pi, \alpha$)
  $avg \leftarrow 0$
  for $i \leftarrow 1, N$ **do**
    $S \leftarrow P[i]$
    $avg \leftarrow avg + |S \cap S^*|$
  end for
  $avg \leftarrow \frac{avg}{N}$
  $\gamma \leftarrow \frac{\alpha \cdot f(S^*)}{avg}$
  $S \leftarrow P[0]$
  if $f(S) < f(S^*)$ **then**
    $S^* \leftarrow S$
    $\pi \leftarrow \pi_{\min}$
  else
    $\pi \leftarrow \min(\pi + \Delta\pi, \pi_{\max})$
  end if
  return S^*, γ, π
end function

Reprodukció

A genetikus algoritmus egy kulcsfontosságú fázisa a reprodukció. Itt döntjük el, hogy a meglévő populációból miként jöjjön létre az új generáció. Ez azt jelenti, hogy meghatározzuk, hogy az S_1 és S_2 szülők kromoszómáit hogyan olvasztjuk egybe annak érdekében, hogy egy új S' egyed szülessen.

Esetünkben úgy történik egy új egyed létrehozása, hogy random módon kiválasztunk 2 különböző szülőt, és ezek kromoszómáit egybevonjuk: $S' = S_1 \cup S_2$. Mivel majdnem biztos, hogy az így kapott egyed kromoszómája több, mint k csomópontot tartalmaz, ezért szükséges törölnünk belőle nódusokat, amíg $|S'| > k$. Az hogy melyik nódus kerül törlésre az új egyed kromoszómájából, random módon történik. A reprodukciós folyamatot a 2.5 kódrészlet szemlélteti.

Fontos megemlítenünk, hogy mivel a szülőket random módon választjuk ki egyed esetén egyed létrehozásához, ezért a populáció egyedei között nem teszünk különbséget. Vagyis keresztezéskor nem nézzük, hogy csak a legrátermettebb szülőket válasszuk, hanem egyenlő eséllyel választunk kevésbé jó fitnesz értékkel rendelkező egyedet is szülőnek. Ez lelassítja a populáció uniformizálódásának folyamatát, de segíti a megoldástér bejárását. Ez azért jó, mert nem tudjuk előre, hogy a csomópontok mely kombinációja fogja eredményezni a bemeneti gráf maximális szétesését, ha ezeket együtt töröljük a gráfból. Ezért a kevésbé jó fitnesz értékkel rendelkező egyedeket sem kell figyelmen kívül hagyni, mert kombinálva őket jó megoldásokhoz juthatunk.

Mutáció

A következő nagy jelentőséggel bíró fázisa a genetikus algoritmusnak a mutáció. Mutáció alatt azt értjük, hogy vesszük az újonnan létrejött populációt, és a populációban található egyedek génjeit perturbáljuk

Algorithm 2.5 Recombination Operator

```
1: function CROSSOVER(k, N, P)
2:     P' ← ∅
3:     for i ← 1, N do
4:         S_1 ← SELECT(P)
5:         S_2 ← SELECT(P)
6:         S' ← S_1 ∪ S_2
7:         if |S'| = k then
8:             P' ← P' ∪ {S'}
9:         else
10:            S' ← RANDOM SAMPLE(S', k)        ▷ Take k random elements from S'
11:            P' ← P' ∪ {S'}
12:        end if
13:    end for
14:    return P'
15: end function
```

valamilyen csekély valószínűséggel. A mutáció azért tartozik a nagy döntések halmazába, mert a mutáció révén fenntartjuk a populáció sokféleségét, és elkerüljük a korai konvergenciát. [3]

A populáció minden egyes új egyede esetén, a mutáció valószínűségét a π paraméter befolyásolja. Generálunk egy egyenletes eloszlású véletlen számot 1 és 100 között, és ha ez kisebb, mint π, akkor módosítjuk a megoldást. A módosítás úgy történik, hogy leszögezzük, hogy a megoldás hány génjét szeretnénk változtatni. Ezt a számot tükrözi az n_g változó, amely értékét a $[0, k]$ intervallumból veszi, és random generáljuk. A következő lépés, hogy kitörlünk n_g csomópontot a megoldásból, de mivel majdnem biztos, hogy a megoldásunk így nem-optimális, mert $|S| < k$, ezért szükséges visszaadogatnunk csomópontokat S-be. Ennek érdekében véletlenszerűen kiválasztunk egy csomópontot a $V \setminus S$ halmazból, és a kiválasztott csomópontot visszatesszük a megoldásba. A 2.6 kódrészlet a mutáció műveletét hívatott bemutatni.

Szelekció

Az utolsó fázisa a genetikus algoritmusunknak a szelekció. Itt döntjük el, hogy mely egyedek fogják alkotni a következő nemzedéket. Jelen esetben ez úgy megy végbe, hogy összefésüljük a régi P és az újonnan létrejött P' populációkat, és rendezzük az egyedeket a 2.3 fitnesz függvény alapján. Növekvő sorrendbe rendezzük őket, mivel nem szabad elfelejtenünk, hogy célunk végső soron a páronkénti konnektivitás minimalizálása. Ezután kiválasztjuk az első N egyedet, és ezeket visszük tovább a következő iterációba. Genetikus algoritmusunk szelekciós szakaszát a 2.7 kódrészlet ismerteti.

3. Angolul: premature convergence.

Algorithm 2.6 Mutation Operator

```
1: function MUTATION(k, N, P, π)
2:     P' ← ∅
3:     for i ← 1, N do
4:         r ← RAND INT(1, N)
5:         if r ≤ π then
6:             S' ← P[i]
7:             n_g ← RAND INT(0, k)                ▷ Number of genes to mutate
8:             for j ← 1, n_g do
9:                 elem ← SELECT(S')
10:                S' ← S' \ {elem}
11:            end for
12:            MIS ← V \ S'
13:            while |S'| < k do
14:                elem ← SELECT(MIS)
15:                S' ← S' ∪ {elem}
16:            end while
17:            P' ← P' ∪ {S'}
18:        else
19:            S ← P [i]
20:            P' ← P' ∪ {S}
21:        end if
22:    end for
23:    return P'
24: end function
```

Algorithm 2.7 Selection Operator

```
1: function SELECTION(N, P, P')
2:     P ← P ∪ P'
3:     SORT(P)                          ▷ Sort individuals by fitness function in ASC order
4:     return P [:N]                    ▷ Take best N solutions
5: end function
```

2.4. GA, de okos inicializálással

Ahhoz, hogy ne teljesen véletlen megoldásokból induljunk ki a 2.2 kódrészlettel szemléltetett genetikus algoritmus esetén, ezért a kezdeti populáció egy részét a 2.1 algoritmus segítségével fogjuk kigenerálni. Ugyan a populáció inicializálása így több időt fog igénybe venni, de a megoldások egy része a bemeneti gráf struktúráját figyelembe véve lesznek meghatározva. Ezt szemlélteti a 2.8 algortimus, amely a kezdeti populáció 10%-át okosan generálja ki, a maradék 90%-át pedig véletlenül, felhasználva a 2.3 algoritmust.

Algorithm 2.8 Smart Initialization

```
1:  function SMART INIT(G, k, N)
2:      P ← ∅
3:      for i ← 1, N · 10/100 do
4:          P ← P ∪ {GREEDY(G, k)}
5:      end for
6:      while |P| < N do
7:          P ← P ∪ {RAND SOL(k)}
8:      end while
9:      return P
10: end function
```

3. fejezet

Kétcélú CNDP

3.1. A CNDP-től a BOCNDP-ig

Az egycélú CNDP-től úgy jutunk el a kétcélú CNDP-ig, hogy nem egy függvényt fogunk optimalizálni, hanem kettőt. Míg a CNDP esetén a 2.1 képlettel leírt függvény minimalizálása volt a feladat, addig a BOCNDP esetén két célfüggvényünk van, amelyeket optimalizálni szeretnénk k csomópont kitörlése után a G gráfból:

1. Maximalizálni szeretnénk az összefüggő komponensek számát.
2. Minimalizálni szeretnénk az összefüggő komponensek számosságának a varianciáját.

Ennek érdekében a következő két célfüggvényt vezetjük be:

$$\max \quad |H|, \tag{3.1}$$

$$\min \quad var(H), \tag{3.2}$$

ahol H-val jelöljük a $G\left[V \setminus S\right]$ feszített részgráf összefüggő komponenseinek a halmazát, és $var(H)$ jelöli az összefüggő komponensek számosságának nem szabályos mintavételének a varianciáját. A H halmaz varianciáját a következő képlet segítségével számoljuk ki:

$$\frac{1}{|H|}\sum_{h\in H}\left(|h| - \frac{n^*}{|H|}\right)^2, \tag{3.3}$$

ahol $n^* = \sum_{h\in H}|h|$ a $G\left[V \setminus S\right]$ feszített részgráf csomópontjainak a száma.

A 3.1 és a 3.3 képletekkel leírt problémát úgy ismerjük az irodalomban [Ventresca et al., 2018], mint BOCNDP. Habár a CNDP is ugyanerre a problémára próbál megoldást nyújtani, mégsem hasonlítható össze a BOCNDP-vel, mert alapjában véve különbözik tőle.

A H halmaz számosságának meghatározását a 3.1. kódrészlet mutatja be Python-ban, míg a 3.3 képlet implementációját a 3.2. kódrészlet.

3.1. Listing. A feszített részgráf összefüggő komponenseinek a száma

```
1 def connected_components(exclude=None):
      if exclude is None:
          exclude = {}
```

```
    S = set(exclude)
    subgraph = networkx.subgraph_view(G, filter_node=lambda n: n not in S)
    return networkx.number_connected_components(subgraph)
```

3.2. Listing. Az összefüggő komponensek számosságának a varianciája

```
def cardinality_variance(exclude=None):
    if exclude is None:
        exclude = {}

    S = set(exclude)
    subgraph = networkx.subgraph_view(G, filter_node=lambda n: n not in S)
    components = list(networkx.connected_components(subgraph))

    num_of_components = len(components)
    num_of_nodes = subgraph.number_of_nodes()
    variance = 0

    for component in components:
        cardinality = len(component)
        variance += (cardinality - num_of_nodes / num_of_components) ** 2

    variance /= num_of_components
    return variance
```

Irodalomjegyzék

Aringhieri, R., Grosso, A., Hosteins, P., és Scatamacchia, R. A general evolutionary framework for different classes of critical node problems. *Engineering Applications of Artificial Intelligence*, 55: 128–145, 2016.

Hagberg, A., Swart, P., és S Chult, D. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

Ventresca, M. Global search algorithms using a combinatorial unranking-based problem representation for the critical node detection problem. *Computers & Operations Research*, 39(11):2763–2775, 2012.

Ventresca, M., Harrison, K. R., és Ombuki-Berman, B. M. The bi-objective critical node detection problem. *European Journal of Operational Research*, 265(3):895–908, 2018.