Bandwidth problem minimizacije

Seminarski rad u okviru kursa
Naučno izračunavanje
Matematički fakultet

Nevena Nikolić, 1021/2018
Luka Kalinić, 1058/2018

Avgust 2019

Sažetak

U ovom radu najpre definišemo, a potom rešavamo *bandwidth* problem minimizacije, koristeći metod promenljivih okolina. Rad se u velikoj meri oslanja na dokument [1].

Sadržaj

1 Uvod

Bandwidth problem minimizacije (eng. *Bandwidth Minimization Problem*) se sastoji u pronalaženju permutacije redova i kolona retke matrice sa ciljem da ne-nula elementi budu sadržani u okviru trake, koja je što je moguće bliža glavnoj dijagonali. Ovaj problem je nastao pedesetih godina prošlog veka i zasniva se na primedbi da kada ne-nula elemente grupišemo u okviru uzane trake oko glavne dijagonale, operacije poput inverzije matrice i računanja determinante postaju vremenski efikasnije.

Primene ovog problema su brojne, a najvažnija je u rešavanju velikih sistema linearnih jednačina. Gausova eliminacija može da se izvede u vremenu $O(nb^2)$ za matricu čija je dimenzija n i *bandwidth* b, što je brže od direktnog $O(n^3)$ algoritma kada je b manje od n. Dokazano je da je *bandwidth* problem minimizacije NP-kompletan, odnosno da nije veoma verovatno da postoji algoritam za njegovo rešavanje koji radi u polinomijalnom vremenu.

Algoritmi za rešavanje *bandwidth* problema minimizacije mogu da se podele na egzaktne i heurističke, a u novije vreme razvijeno je i više metaheuristika. Metaheuristike predstavljaju opšte šablone čijim preciziranjem se dolazi do heurističkih metoda za konkretne probleme. [3].

2 Formulacija problema

Bandwidth problem minimizacije može biti formulisan za matrice ili grafove, pri čemu važi da su obe formulacije problema ekvivalentne. Ekvivalencija je zasnovana na činjenici da se dati graf može predstaviti svojom matricom povezanosti i obratno.

2.1 *Bandwidth* problem minimizacije za matrice

Neka je data binarna (svi elementi su 0 ili 1), retka, simetrična matrica $A = \{a_{ij}\}$. Tada se *bandwidth* matrice A definiše kao

$$B(A) = max\{|i - j| : a_{ij} \neq 0\} \tag{1}$$

Bandwidth problem minimizacije za matrice (eng. *Matrix Bandwidth Minimization Problem*, skraćeno *MBMP*) se sastoji u permutovanju redova i kolona matrice A tako da ne-nula elementi budu sadržani u traci što je moguće bližoj glavnoj dijagonali, odnosno tako da se minimizuje *bandwidth* $B(A)$.

2.2 *Bandwidth* problem minimizacije za grafove

Neka je $G = (V, E)$ konačan neusmereni graf, gde je V skup čvorova i E skup grana. Neka je 1-1 funkcija $f : V \to \{1, 2, ..., n\}$ funkcija označavanja čvorova grafa. Tada se *bandwidth* čvora v definiše kao

$$B_f(v) = max_{j:(v,j)\in E}\{|f(v) - f(j)|\} \tag{2}$$

i *bandwidth* grafa G za f se definiše kao

$$B_f(F) = max\{|f(i) - f(j)| : (i, j) \in E\} \tag{3}$$

Bandwidth problem minimizacije za grafove (eng. *Graph Bandwidth Minimization Problem*, skraćeno *GBMP*) se sastoji u pronalaženju funkcije označavanja f koja minimizuje *bandwidth* grafa, odnosno veličinu $B_f(G)$.

Za graf sa n čvorova, broj mogućih označavanja je $n!$. Direktan pristup mogao bi da bude isprobavanje svih permutacija i izbor najboljeg rešenja. Međutim, vreme izvršavanja ovog metoda je $O(n!)$, te ovaj pristup postaje nepraktičan već za grafove koji imaju 10 čvorova. Imajući u vidu ekvivalenciju dve navedene verzije problema, u nastavku ćemo podrazumevati *bandwidth* problem minimizacije za grafove.

3 Metode za rešavanje

Kao što smo već pomenuli, algoritmi za rešavanje *bandwidth* problema minimizacije mogu da se podele na egzaktne i heurističke (odnosno metaheurističke).

Egzaktni algoritmi se najčešće baziraju na pristupu poznatom kao *grananje i ograničavanje* (eng. *branch and bound*). Del Korso i Manzini (*Del Corso G.M., G. Manzini*, 1999) su uspeli da reše problem za instance male i srednje veličine. Kaprara i Salazar (*Caprara A. and J.J. Salazar*, 2005) su uspeli da prošire prethodni rezultat uvodeći uže donje granice. Međutim, treba uzeti u obzir i cenu izračunavanja za dobijanje optimalnog rešenja. Stoga ovi metodi mogu rešavati samo probleme čija je veličina dovoljno mala da vreme izvršavanja bude razumno.

Heuristički pristupi zasnovani su na iskustvu i dobijeno rešenje ne mora biti optimalno. Ipak, heurističkim pristupom možemo brzo pronaći rešenje koje je dovoljno dobro za dati problem kombinatorne optimizacije. Metaheuristike, kao što smo pomenuli, predstavljaju tehniku koja je opštija od heuristika i zahtevaju manje računskih pretpostavki, te je poslednjih decenija akcenat stavljen na razvijanje metaheurističkih metoda za rešavanje *bandwidth* problema minimizacije. U literaturi se najčešće pominju tri ovakve metaheuristike: simulirano kaljenje (eng. *Simulated Annealing*, SA), tabu pretraga (eng. *Tabu Search*, TS) i metod promenljivih okolina (eng. *Variable Neighborhood Search*, VNS). U nastavku ćemo se fokusirati na rešavanje *bandwidth* problema minimizacije koristeći metod promenljivih okolina, za koji je eksperimentalno pokazano da ima bolje performanse od ostalih pristupa.

4 VNS pristup rešavanju problema

Metod promenljivih okolina (VNS) je metaheuristika predložena 1997. godine koja se pokazala veoma korisnom za dobijanje približnog rešenja optimizacionih problema. Predstavlja uopštenje lokalne pretrage, gde su definisane okoline po kojima će se sistematski vršiti pretraživanje [2]. VNS se bazira na sledećim principima:

- Lokalni minimum za jedan tip okoline ne mora nužno i da bude lokalni minimum za drugi tip okoline.
- Globalni minimum predstavlja lokalni minimum za sve tipove okolina.

- Za mnoge probleme u praksi, lokalni minimumi više tipova okolina su relativno blizu jedan drugog.

U literaturi postoji nekoliko varijanti osnovnog algoritma, među kojima se najčešće sreću redukovani, osnovni i uopšteni metod promenljivih okolina. U svim slučajevima se, ukoliko nije pronađeno bolje rešenje, prelazi u narednu okolinu, a inače, ukoliko se pronađe bolje rešenje, algoritam se resetuje na prvi razmatrani tip okoline.

Osnovni VNS algoritam uključuje tri postupka: razmrdavanje, lokalnu pretragu i promenu okoline. Razmrdavanje predstavlja pokušaj iskakanja iz tekućeg lokalnog optimuma i pronalaženja novog lokalno optimalnog rešenja koje će biti bliže globalno optimalnom rešenju. Za pronalaženje lokalnog optimuma koristi se lokalna pretraga kako bi se unapredila preciznost pretrage. *Pomeriti se ili ne* (eng. *move or not*) označava menjanje okoline, što nam daje iterativni metod i kriterijum zaustavljanja. Pseudokod osnovnog VNS algoritma može se prikazati na sledeći način:

- Odabrati tipove okolina $U_l(x), 1 \leq l \leq l_{max}$
- Generisati početno rešenje x i postaviti vrednost najboljeg rešenja $f^* = f(x)$
- Dok nije zadovoljen kriterijum zaustavljanja, ponavljati sledeće korake:
  - $l = 1$
  - Dok je $l \leq l_{max}$ ponavljati sledeće korake:
    - Izabrati proizvoljno rešenje x' u okolini $U_l(x)$
    - Primenom nekog tipa lokalne pretrage na početno rešenje x' naći rešenje x'' sa najmanjom vrednošću funkcije cilja
    - Ako rešenje x'' ima manju vrednost funkcije cilja od rešenja x, onda dodeliti $x = x''$ i $l = 1$, a inače $l = l + 1$
  - Po potrebi, ažurirati vrednost f^*
- Ispisati vrednost f^*

4.1 Početno rešenje

Kvalitet početnog rešenja će direktno uticati na performanse algoritma. Dobro početno rešenje može garantovati da će za kratko vreme biti dobijeno rešenje koje je globalno optimalno ili blisko optimalnom.

Dobro početno rešenje može se generisati korišćenjem pretrage u širinu (eng. *breadth first search*, BFS). Osnovna ideja je da bi povezani čvorovi trebalo da imaju bliske oznake. Struktura nivoa (*level structure*) grafa, u oznaci $L(G)$, jeste particionisanje njegovih čvorova u nivoe $L_1, L_2, ..., L_k$ koji zadovoljavaju sledeće uslove:

1. Čvorovi povezani sa čvorom iz nivoa L_1 su ili u L_1 ili u L_2
2. Čvorovi povezani sa čvorom iz nivoa L_k su ili u L_k ili u L_{k-1}
3. Čvorovi povezani sa čvorom iz nivoa L_i (za $1 < i < k$) su ili u L_{i-1} ili u L_i ili u L_{i+1}

Prema tome, mogu se dobiti razumno dobra rešenja. Dakle, procedura za generisanje početnog rešenja se sastoji u primeni BFS algoritma sa nasumičnim izborom početnog čvora; različiti izbori početnog čvora vode različitim početnim rešenjima. Prema strukturi nivoa, sva početna rešenja su bolja od originalnog dodeljivanja. Očigledno, *bandwidth* koji se dobija

ovim metodom ne može biti gori od maksimalnog *bandwidth*-a grafa, jer su susednim čvorovima dodeljeni uzastopni brojevi. BFS metod daje gornju granicu za kvalitetno rešenje.

4.2 Razmrdavanje

Označavanje f' je u k-toj okolini označavanja f ako se ta dva označavanja razlikuju u $k+1$ oznaci. Preciznije, rastojanje ρ između dva rešenja f i f' se definiše kao:

$$\rho(f, f') = \sum_{i=1}^{n} \eta(i) - 1, \eta(i) = \begin{cases} 1, f(i) \neq f'(i) \\ 0, f(i) = f'(i) \end{cases} \tag{4}$$

Na primer, ako su data označavanja $f = (3, 2, 1, 5, 4)$ i $f' = (4, 1, 3, 2, 5)$, rastojanje između njih je 4. U cilju odabira čvorova za razmenu njihovih oznaka, uvodimo još dve definicije:

$$f_{max}(v) = max\{f(u), u \in N(v)\} \tag{5}$$

$$f_{min}(v) = min\{f(u), u \in N(v)\} \tag{6}$$

$f_{max}(v)$ predstavlja najveću oznaku među čvorovima susednim čvoru v, dok $f_{min}(v)$ predstavlja najmanju.

Najpre se generiše skup čvorova $K \subset V$ takav da mu je kardinalnost veća od datog broja k. Zatim se nasumično bira čvor u iz skupa K i pronađe se njegov kritični čvor. Kritični čvor čvora u je čvor v za koji važi

$$|f(u) - f(v)| = B_f(u) \tag{7}$$

Dalje, bira se čvor w tako da zadovoljava sledeća dva uslova:

- vrednost $max\{f_{max}(w) - f(v), f(v) - f_{min}(w)\}$ je minimalna
- važi nejednakost $f_{min}(u) \leq f(w) \leq f_{max}(u)$

Konačno, zamenjuju se oznake čvorovima v i w. Opisani postupak se može predstaviti narednim pseudokodom (slika 1).

Algorithm 1 Shaking (k, f)

Initialization:

Let $K = \{v | B_f(v) \geq B'\}$, B' is chosen such that $|K| \geq k$;

Iteration:

1: **for** $i = 1$ to k **do**
2: $u \leftarrow$ RandomInt $(1, |K|)$;
3: $v \leftarrow$ such that $|f(u) - f(v)| = B_f(u)$;
4: **if** $(u, v) \in E$ **then**
5: $w \leftarrow \arg\min_w \{\max\{f_{max}(w) - f(v), f(v) - f_{min}(w)\} | f_{min}(u) \leq f(w) \leq f_{max}(u)\}$;
6: $swap(f(u), f(v))$
7: **end if**
8: **end for**

Slika 1: Algoritam 1 - razmrdavanje

4.3 Lokalna pretraga

Koristimo lokalnu pretragu da bismo konstruisali skup čvorova pogodnih za zamenu oznake. Najbolje označavanje za tekući čvor v se definiše kao:

$$mid(v) = [\frac{max(v) + min(v)}{2}] \tag{8}$$

Onda je skup čvorova pogodnih za zamenu oznake za čvor v definisan kao:

$$N'(v) = \{u : |mid(v) - f(u)| < |mid(v) - f(v)|\} \tag{9}$$

Prema ovako definisanom skupu $N'(v)$, zamena oznake trenutnog čvora v sa čvorom $u \in N'(v)$ pokušava se jedna po jedna u poretku rastuće vrednosti $|mid(v) - f(u)|$, sve dok rešenje nije unapređeno. Pored toga, ako *bandwidth* grafa nije smanjen, ali broj kritičnih grana jeste, ovaj uslov se takođe može posmatrati kao unapređenje rešenja. Kritična grana je ona grana (u, v) takva da važi $B_f(u) = B_f(G)$ i $B_f(v) = B_f(G)$. Opisani postupak može se predstaviti narednim pseudokodom (slika 2).

Algorithm 2 Local Search (f)

```
1:  while CanImprove do
2:    CanImprove = False;
3:    for v = 1 to n do
4:      if B_f(v) = B_f(G) then
5:        for all u such that u ∈ N'(v) do
6:          swap (f(v), f(u)) and update
            (B_f(w), B_f(G)), ∀w ∈ (N(v) ∪ N(u));
7:          if number of critical edges reduced
            then
8:            CanImprove = True;
9:            break;
10:         end if
11:         swap (f(v), f(u)) and update
            (B_f(w), B_f(G)), ∀w ∈ (N(v) ∪ N(u));
12:       end for
13:     end if
14:   end for
15: end while
```

Slika 2: Algoritam 2 - lokalna pretraga

4.4 Pomeriti se ili ne

Nakon pronalaska lokalno optimalnog rešenja, moramo doneti odluku da li se trenutno rešenje f zamenjuje novim rešenjem f'. Razmatraju se sledeća tri slučaja:

- $B_{f'}(G) < B_f(G)$: ako je *bandwidth* novog rešenja bolji od trenutnog *bandwidth*-a, lako je zaključiti da se treba pomeriti.
- $|V_c(f')| < |V_c(f)|$: ako se *bandwidth* ne menja, odnosno $B_{f'}(G) = B_f(G)$, upoređujemo broj kritičnih čvorova (onih čiji je *bandwidth* jednak $B_f(G)$) za trenutno i novo rešenje da bismo videli da li je $|V_c(f')|$ smanjen.
- $\rho(f', f) > \alpha$: Ako prethodna dva uslova nisu ispunjena, onda poredimo rastojanje između trenutnog i novog rešenja sa korisnički zadatim koeficijentom α.

Opisani postupak može se predstaviti narednim pseudokodom (slika 3).

Algorithm 3 Move (f, f', α)

$Move \leftarrow False$;
if $B_{f'}(G) < B_f(G)$ **then**
  $Move \leftarrow True$;
else
  if $B_{f'}(G) = B_f(G)$ **then**
    if $|V_c(f')| < |V_c(f)|$ or $\rho(f', f) > \alpha$ **then**
      $Move \leftarrow True$;
    end if
  end if
end if

Slika 3: Algoritam 3 - pomeriti se ili ne

Konačno, VNS pristup rešavanju *bandwidth* problema minimizacije može se predstaviti narednim pseudokodom (slika 4).

Algorithm 4 VNS $(A, k_{min}, k_{max}, k_{step}, \alpha)$

Initialization:
$B^* \leftarrow \infty; t \leftarrow 0$;
$i_{max} = Int((k_{max} - kmin)/k_{step}))$;
$f \leftarrow InitSol(f); f \leftarrow LocalSearch(f);$;
$i \leftarrow 0; k \leftarrow k_{min}$;
while $i \leq i_{max}$ **do**
  $f' \leftarrow Shaking(f, k)$;
  $f' \leftarrow LocalSearch(f')$;
  if $Move(f, f', \alpha)$ **then**
    $f \leftarrow f'; k \leftarrow k_{min}; i \leftarrow 0$;
  else
    $k \leftarrow k + k_{step}; i \leftarrow i + 1$;
  end if
end while

Slika 4: Algoritam 4 - VNS

5 Python implementacija

Koristimo biblioteku **networkx** za rad sa grafovima (dokumentacija dostupna na sledećem link-u) i biblioteku **matplotlib.pyplot** za vizualizaciju grafa. Čvorove grafa označavamo sa $v_1, v_2, ..., v_n$. Funkcija *generate_graph* za date cele brojeve n i e formira graf sa n čvorova i e grana na pseudo-slučajan način.

```
def generate_graph(n, e):

    l = [i for i in range(1,n+1)]
    V = nodes_from_labeling(l)
    G = nx.Graph()

    for i in range(e):
        u = random.choice(V)
        v = random.choice(V)

        while u == v:
            v = random.choice(V)

        G.add_edge(u,v)


    return G
```

Funkcija *visualize_graph* vizualizuje prosleđeni graf G.

```
def visualize_graph(G):

    pos = nx.spring_layout(G)
    plt.figure(figsize=(10,10))
    nx.draw_networkx_nodes(G,pos,node_color='pink')
    nx.draw_networkx_labels(G,pos,font_color='magenta')
    nx.draw_networkx_edges(G,pos,edgelist=list(G.edges),color='purple')
    plt.axis('off')
    plt.show()
```

Funkciju označavanja f zadajemo kao listu $[l_1, l_2, ..., l_3]$. Vezu između čvorova grafa i označavanja f uspostavljamo funkcijom *label*, koja za dati graf G, čvor *node* i funkciju označavanja f vraća oznaku koju tom čvoru dodeljuje funkcija f.

```
def label(G, node, f):

    index_of_node = int(node.strip('v'))-1
    return f[index_of_node]

```

Definišemo zatim dve pomoćne funkcije. $labeling\_from\_nodes$ je funkcija koja nam od liste čvorova $v_j, 1 \leq j \leq n$, formira i vraća listu indeksa $j, 1 \leq j \leq n$ (tj. označavanje tih čvorova). $nodes\_from\_labeling$ je funkcija koja nam od liste indeksa $j, 1 \leq j \leq n$, formira i vraća listu čvorova $v_j, 1 \leq j \leq n$.

```
def labeling_from_nodes(nodes):
    n = len(nodes)
    return [int(nodes[i].strip('v')) for i in range(n)]

def nodes_from_labeling(labeling):
    n = len(labeling)
    return ['v'+str(labeling[i]) for i in range(n)]
```

Funkcija *bandwidth_of_a_node* za dati graf G, čvor *node* i funkciju označavanja f izračunava *bandwidth* čvora *node* pri označavanju f, prema formuli (2).

```
def bandwidth_of_a_node(G, node, f):
    neighbors = [u for u in G.neighbors(node)]
    label_node = label(G,node,f)
    maximum = float('-inf')
    for u in neighbors:
        label_u = label(G,u,f)
        diff = abs(label_u - label_node)
        if diff > maximum:
            maximum = diff
    return maximum
```

Funkcija *bandwidth* za dati graf G i funkciju označavanja f izračunava *bandwidth* grafa G pri označavanju f, prema formuli (3).

```
def bandwidth(G, f):
    nodes = [v for v in G.nodes]
    bandwidths = [bandwidth_of_a_node(G,node,f) for node in nodes]
    return max(bandwidths)
```

Funkcija *initial_solution* za dati graf G generiše početno rešenje - početno označavanje f čvorova grafa G, koristeći BFS sa nasumičnim izborom čvora od kojeg se započinje pretraga.

```
def initial_solution(G):
    nodes = [u for u in G.nodes]
    start = random.choice(nodes)
    edges_iterator = nx.bfs_edges(G,start)
    edges = [edge for edge in edges_iterator]
    solution = [start] + [edge[1] for edge in edges]
    return labeling_from_nodes(solution)
```

Funkcija *distance* za data dva rešenja (označavanja) f i f_p računa rastojanje između njih, prema formuli (4).

```python
def distance(f,f_p):
    n = len(f)
    dist = 0
    for i in range(n):
        if f[i] != f_p[i]:
            dist += 1
    dist -= 1
    return dist
```

Funkcija *max_labeled_neighbor* za dati graf G, čvor *node* i označavanje f pronalazi najveću oznaku među čvorovima susednim čvoru *node*, prema formuli (5).

```python
def max_labeled_neighbor(G, node, f):
    neighbors = [u for u in G.neighbors(node)]
    max_label = float('-inf')
    for u in neighbors:
        label_u = label(G,u,f)
        if label_u > max_label:
            max_label = label_u
    return max_label
```

Analogno, funkcija *min_labeled_neighbor* za dati graf G, čvor *node* i označavanje f pronalazi najmanju oznaku među čvorovima susednim čvoru *node*, prema formuli (6).

```python
def min_labeled_neighbor(G, node, f):
    neighbors = [u for u in G.neighbors(node)]
    min_label = float('inf')
    for u in neighbors:
        label_u = label(G,u,f)
        if label_u < min_label:
            min_label = label_u
    return min_label
```

Funkcija *set_K* za dati graf G, označavanje f i ceo broj k generiše podskup čvorova K, na osnovu *Initialization* koraka u algoritmu 1.

```python
def set_K(G, f, k):
    nodes = [u for u in G.nodes]
    B = bandwidth(G,f)
    B_p = random.randint(1,B)

    K = []
    while True:
        for node in nodes:
            bandwidth_node = bandwidth_of_a_node(G, node, f)
            if bandwidth_node >= B_p:
                K.append(node)

        if len(K) >= k:
            break
        else:
            K = []
            B_p = random.randint(1,B)

    return K
```

Funkcija *critical_node* za dati graf G, čvor *node* i označavanje f nalazi kritičan čvor čvora *node*, na osnovu formule (7).

```python
def critical_node(G, node, f):

    nodes = [u for u in G.nodes]
    for v in nodes:
        if abs(label(G, node, f) - label(G, v, f)) == bandwidth_of_a_node(G, node, f):
            return v
```

Funkcija *swap_labels* za dati graf G, označavanje f i čvorove u i v menja označavanje f, tako što datim čvorovima zameni oznake.

```python
def swap_labels(G, u, v, f):

    label_u = label(G, u, f)
    label_v = label(G, v, f)

    index_u = f.index(label_u)
    index_v = f.index(label_v)

    f[index_u] = label_v
    f[index_v] = label_u

    return f
```

Funkcija *shaking* za dati graf G, označavanje f i ceo broj k realizuje fazu razmrdavanja, prema algoritmu 1.

```
def shaking(G, f, k):

    K = set_K(G, f, k)

    edges = [edge for edge in G.edges]
    nodes = [node for node in G.nodes]

    for i in range(k):
        u = random.choice(K)
        v = critical_node(G, u, f)

        if (u,v) in edges:
            f_min_u = min_labeled_neighbor(G, u, f)
            f_max_u = max_labeled_neighbor(G, u, f)

            label_v = label(G, v, f)

            min_value = float('inf')
            min_w = None

            for w in nodes:
                label_w = label(G, w, f)
                if label_w <= f_max_u and label_w >= f_min_u:
                    current_value = max(max_labeled_neighbor(G, w, f) - label_v,
                                        label_v - min_labeled_neighbor(G, w, f))

                    if current_value < min_value:
                        min_value = current_value
                        min_w = w

            f = swap_labels(G,v,min_w,f)

    return f
```

Funkcija *best_labeling* za dati graf G, čvor v i označavanje f pronalazi najbolju oznaku čvora v, prema formuli (8).

```
def best_labeling(G, v, f):
    max_v = max_labeled_neighbor(G, v, f)
    min_v = min_labeled_neighbor(G, v, f)

    mid_v = (max_v + min_v) // 2

    return mid_v
```

Funkcija *suitable_swapping_nodes* za dati graf G, čvor v i označavanje f pronalazi skup čvorova pogodnih za razmenu oznake sa čvorom v, prema formuli (9).

```
def suitable_swapping_nodes(G, v, f):
    nodes = [u for u in G.nodes]
    mid_v = best_labeling(G, v, f)
    label_v = label(G, v, f)
    N_p = []

    for u in nodes:
        label_u = label(G, u, f)
        if abs(mid_v-label_u) < abs(mid_v-label_v):
            N_p.append(u)

    return N_p
```

Funkcija *number_of_critical_edges* za dati graf G i označavanje f izračunava broj kritičnih grana. Pojam kritične grane uveden je u poglavlju 4.3.

```
def number_of_critical_edges(G, f):

    number = 0
    B = bandwidth(G, f)
    edges = [edge for edge in G.edges]

    for edge in edges:
        v_from = edge[0]
        v_to = edge[1]

        if bandwidth_of_a_node(G, v_from, f) == B and bandwidth_of_a_node(G, v_to, f) == B:
            number += 1

    return number
```

Funkcija *local_search* za dati graf G i označavanje f realizuje fazu lokalne pretrage, prema algoritmu 2.

```
def local_search(G, f):

    canImprove = True
    nodes = [u for u in G.nodes]

    while canImprove:
        canImprove = False

        for v in nodes:

            number_of_critical = number_of_critical_edges(G, f)

            if bandwidth_of_a_node(G, v, f) == bandwidth(G, f):

                N_p = suitable_swapping_nodes(G,v,f)

                for u in N_p:
                    f = swap_labels(G,v,u,f)

                    if number_of_critical_edges(G, f) < number_of_critical:
                        canImprove = True
                        break

                    f = swap_labels(G,v,u,f)

    return f
```

Funkcija *number_of_critical_nodes* za dati graf G i označavanje f izračunava broj kritičnih čvorova. Pojam kritičnog čvora uveden je u poglavlju 4.4.

```
def number_of_critical_nodes(G, f):

    Vc = set([])
    B = bandwidth(G, f)
    nodes = [u for u in G.nodes]

    for u in nodes:
        if bandwidth_of_a_node(G, u, f) == B:
            Vc.add(u)

    return len(Vc)
```

Funkcija *move* za dati graf G, trenutno rešenje f, novo rešenje f_p i ceo broj α odgovara na pitanje "pomeriti se ili ne?", prema algoritmu 3.

```
def move(G, f, f_p, alpha):

    Move = False
    B_f = bandwidth(G, f)
    B_f_p = bandwidth(G, f_p)

    if B_f_p < B_f:
        Move = True

    else:
        if B_f == B_f_p:
            if number_of_critical_nodes(G,f_p) < number_of_critical_nodes(G,f) or distance(f, f_p) > alpha:
                Move = True

    return Move
```

Najzad, funkcija VNS za dati graf G, parametre k_{min}, k_{max}, k_{step} i α realizuje metod promenljivih okolina za rešavanje *bandwidth* problema minimizacije, prema algoritmu 4.

```
def VNS(G, k_min, k_max, k_step, alpha):
    B_star = float('inf')
    t = 0

    i_max = int((k_max - k_min)/k_step)

    f = initial_solution(G)
    f = local_search(G, f)
    i = 0
    k = k_min

    while i <= i_max:
        f_p = shaking(G, f, k)
        f_p = local_search(G, f)
        if move(G, f, f_p, alpha):
            f = f_p
            k = k_min
            i = 0
        else:
            k = k + k_step
            i = i + 1

    return f
```

Literatura

[1] Abdel Lisser Chen Wang, Chuan Xu. Bandwidth minimization problem. In *MOSIM 2014, 10ème Conférence Francophone de Modélisation, Optimisation et Simulation*, Nancy, France, November 2014. URL: https://hal.archives-ouvertes.fr/hal-01166658/document.

[2] Stefan Mišković. Metoda promenljivih okolina - beleške sa vežbi. URL: http://ni.matf.bg.ac.rs/vezbe/4/vns.html.

[3] Anđelka Zečević Mladen Nikolić. *Naučno Izračunavanje*. Matematički fakultet, Beograd.