Zadatak #8

Definicija stabla i šume

Definicija stabla

- **Stablo** je povezani neusmeren graf bez ciklusa.
- Graf G=(V,E) je stablo ako zadovoljava sledeće:
  1. Konektovan je (postoji put između bilo koja dva čvora).
  2. Nema ciklusa.
- Za graf G=(V,E), sledeći uslovi su ekvivalentni i opisuju stablo:
  1. G je stablo.
  2. G ima |E|=|V|-1.
  3. Postoji tačno jedan put između svaka dva čvora.
  4. Dodavanjem bilo koje grane, G formira ciklus.
  5. Uklanjanjem bilo koje grane, G postaje nepovezan.

Osobine stabla

1. **Broj grana** – ako stablo ima n čvorova (|V|=n), onda ima n-1 grana (|E|=n-1).
2. **Jedinstven put** – između bilo koja dva čvora postoji tačno jedan put.
3. **Listovi** - čvor koji nema potomaka naziva se list. Ako stablo ima L listova, njihova distribucija zavisi od strukture stabla (npr. binarno, n – stablo).
4. **Visina stabla** – visina stabla je maksimalna udaljenost od korena do bilo kog lista.

Matematički dokaz

- Dokaz : Broj grana je n-1
- Indukcija po n:
  - **Baza**: Za n=1, jednočvorno stablo nema grana (|E|=0).
  - **Induktivni korak**: Pretpostavimo da važi za n. Dodavanjem jednog čvora i povezivanjem sa jednim postojećim čvorom, dodajemo tačno jednu granu. Dakle, |E|=(n-1)+1=n.

- Dokaz: Stablo je graf sa tačno jednim putem između svaka dva čvora
  - **Pretpostavka**: Postoje dva različita puta između čvorova u i v.
  - $P_1 i\ P_2$ formiraju ciklus, što je kontradikcija definiciji stabla.
  - Dakle, između u i v postoji tačno jedan put.

Definicija šume

- Šuma je nepovezani graf bez ciklusa.
- Graf F=(V,E) je šuma ako:

1. Svaka povezana komponenta CF je stablo.
2. Broj grana E je $|V| - k_r$ gde je k broj komponenti (stabala).

Primer stabla I šume

1. Stablo:

V={A,B,C,D},E={(A,B),(A,C),(C,D)}.

- Ovo je stablo jer ima 4 čvora i 3 grane (|E|=|V|-1), i svaki par čvorova je povezan tačno jednim putem.

2. Šuma:

V={A,B,C,D,E,F},E={(A,B),(C,D),(E,F)}.

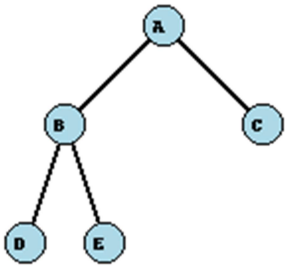
- Ovo je šuma jer ima 3 povezana stabla: {(A,B)},{(C,D)},{(E,F)}.

Pet tvrdnji i četiri leme da bi graf bio stablo

Pet tvrđenja:

Neka je G = (V,E) neusmeren, povezan graf sa n čvorova.

Sledećih pet tvrdnji su međusobno ekvivalentne:



1. *G* je stablo.
2. *G* je povezan i ima tačno *n−1* granu.
3. *G* je acikličan i ima tačno *n−1* granu.
4. *G* je minimalno povezan (povezan, ali uklanjanje bilo koje grane ga čini nepovezanim).
5. *G* je maksimalno acikličan (acikličan, ali dodavanje bilo koje grane stvara ciklus).

Leme sa dokazima:

Lema 1:
G je stablo ⇔ *G* je povezan i ima tačno $n-1$ granu.

Dokaz:

- (⇒) Ako je *G* stablo, po definiciji je povezan i acikličan. Za takav graf važi da ima tačno $n-1$ granu.
- (⇐) Ako je *G* povezan i ima $n-1$ granu, ne može imati ciklus (jer bi imao više grana). Dakle, stablo.

Lema 2:
G je stablo ⇔ *G* je acikličan i ima tačno $n-1$ granu.

Dokaz:

- (⇒) Stablo nema cikluse i ima $n-1$ granu.
- (⇐) Ako je *G* acikličan i ima $n-1$ granu, mora biti povezan (inače bi imao manje grana), pa je stablo.

Lema 3:
G je stablo ⇔ *G* je minimalno povezan.

Dokaz:

- (⇒) U stablu, uklanjanjem bilo koje grane se narušava povezanost. Dakle, stablo je minimalno povezano.
- (⇐) Ako je *G* minimalno povezan, onda je povezan, a svaka grana je neophodna → nema ciklusa → stablo.

Lema 4:
G je stablo ⇔ *G* je maksimalno acikličan.

Dokaz:

- (⇒) Dodavanje bilo koje grane u stablo pravi ciklus.
- (⇐) Ako je *G* maksimalno acikličan, onda je već acikličan, a dodavanje bilo koje grane pravi ciklus → *G* je povezano sa $n-1$ grana → stablo.

Priferov kod

Priferov kod (takođe poznat kao Prüferov kod) je kod koji predstavlja stablo (tj. povezani, aciklični graf) sa n čvorova u obliku niza celih brojeva. To je kompaktan način kodiranja stabla koji se koristi u kombinatorici i računarstvu, često za generisanje i manipulaciju stablima.

Zašto je Priferov kod važan?

1. Jedinstvenost: Svako stablo s n čvorova ima tačno jedan Prüferov kod dužine n−2, i svaki niz od n−2 celih brojeva (sa vrednostima između 1 i n) odgovara jedinstvenom stablu.
2. Kompaktnost: Umesto da se stablo predstavi skupom čvorova i grana, ono se može sažeti u niz od n−2 brojeva.
3. Primena u algoritmima: Priferov kod se koristi za:
    - Generisanje svih mogućih stabala na n čvorova.
    - Kodiranje i dekodiranje stabala.
    - Optimizaciju zadataka kao što su problemi sa povezivanjem i strukturisanjem stabala.
4. Intuitivan odnos sa čvorovima: Kod sadrži informacije o "strukturi povezanosti" stabla, što omogućava rekonstrukciju bez dodatnih podataka.

Primer određivanja Priferovog koda za dato stablo

Algoritam za generisanje Pruferovog koda

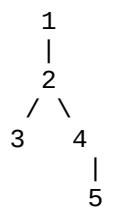
1. Identifikujte list sa najmanjim indeksom.
2. Zapišite čvor koji je povezan sa tim listom.
3. Uklonite list iz stabla.
4. Ponovite postupak dok u stablu ne ostanu samo dva čvora.

Rezultat će biti niz brojeva koji predstavlja Pruferov kod stabla.

Evo jednostavnog primjera stabla i objašnjenja kako generisati njegov Pruferov kod korak po korak

Primer stabla:

Imamo sledeće stablo sa 5 čvorova:



Grane stabla:
1−2, 2−3, 2−4, 4−5

Koraci za generisanje Pruferovog koda:

Korak 1: Pronalaženje lista sa najmanjim indeksom.

- Listovi su čvorovi koji imaju samo jednog susjeda.
  Trenutni listovi: **1, 3, 5**.
  Najmanji indeks: **1**.
- Zapišemo njegovog susjeda (2) u Pruferov kod.
- Uklonimo list 1.

Pruferov kod: **[2]**
Preostale grane: 2−3, 2−4, 4−5.

Korak 2: Ponovo pronađemo najmanji list.

- Listovi: **3, 5**.
  Najmanji indeks: **3**.
- Zapišemo njegovog susjeda (2) u Pruferov kod.
- Uklonimo list 3.

Pruferov kod: **[2, 2]**
Preostale grane: 2−4, 4−5.

Korak 3: Ponovo pronađemo najmanji list.

- Listovi: **5**.
  Najmanji indeks: **5**.
- Zapišemo njegovog suseda (4) u Pruferov kod.
- Uklonimo list 5.

Pruferov kod: **[2, 2, 4]**
Preostale grane: 2−4.

Korak 4: Završni korak.

- Ostaju dva čvora: **2** i **4**.
  Algoritam se završava.

Konačni Pruferov kod:

[2, 2, 4]

Objašnjenje:

- Pruferov kod uvek ima tačno n−2 elemenata, gde je n broj čvorova.
  U ovom slučaju, n = 5, pa Pruferov kod ima 5−2 = 3 elementa.

Primer određivanja stabla za dat Priferov kod

Koraci za određivanje stabla:

1. Odredimo koliko čvorova ima stablo
2. Pronađemo broj pojavlljivanja svakog čvora
3. Rekonstruišemo stablo

Primer koda:

[1, 1, 3, 5, 5]

Korak 1: Odredjivanje broja cvorova

- Spomenuli smo da kod uvek ima tacno n-2 elementa, gde je n broj cvorova.
- Duzina koda je 5, samim tim broj cvorova je 5 + 2 = 7

  V = {1, 2, 3, 4, 5, 6, 7}

Korak 2: Broj pojavljivanja stepeni

- Sada kada imamo broj cvorova, ispisemo koliko svaki od njih ima pojavljivanja u Priferovom kodu

  Kod = [1, 1, 3, 5, 5]

  Pojavljivanja:

  1: 2

  2: 0

  3: 1

  4: 0

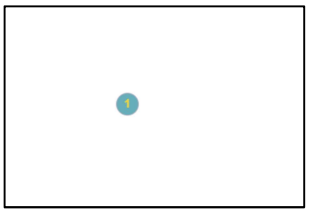
  5: 2

  6: 0

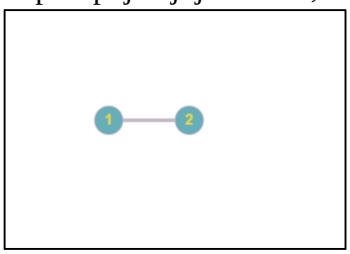
  7: 0

Korak 3: Rekonstruisemo stablo

- Mozemo krenuti od crtanja prvog cvora kojeg imamo u kodu, **u nasem slucaju 1**



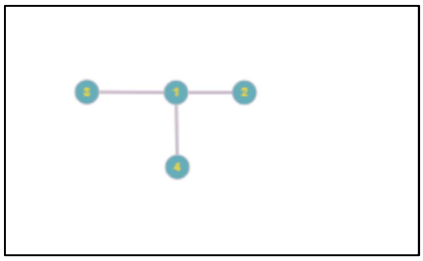
- Dalje gledamo koji je najmanji cvor koji se pojavljuje 0 puta i vezujemo ga za cvor koji se prvi pojavljuje u kodu, **u nasem slucaju je to 2 I imamo granu 1-2**



- Sada mozemo zaboraviti na cvor 2, posto on vise nema pojavljivanja.

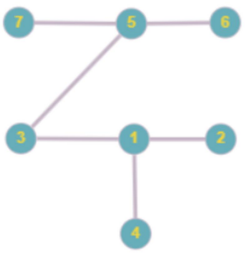
Korak 4: Ponavljanje.

- Sledeca grana ce nam biti izmedju novog najmanjeg broja i sledeceg cvora u Priferovom kodu, **u nasem slucaju 1 i 4**
- Nakon ovoga ce nam ponestati jedinica iz Priferovog koda, sto znaci da ce nam u sledecem koraku 1 biti najmanji cvor sa nula pojavljivanja u kodu. **Sledeca grana je onda 3-1**.



- Tim sistemom dobijamo redom sledece grane: **3-5, 5-6, 5-7**

Konačni izgled stabla:



Definicija težinskog grafa

Težinski graf je graf gde su granama (ivice) pridružene vrednosti koje se nazivaju težine. Te težine mogu predstavljati različite osobine, kao što su udaljenost, trošak, trajanje, kapacitet i slično. Formalno, težinski graf može biti predstavljen kao G=(V,E,w), gde:

- V je skup čvorova (tačaka, temena).
- E je skup grana (ivica) koje povezuju čvorove.
- $w: E \rightarrow R$ je funkcija koja svakoj grani $e \in E$ dodeljuje težinu w(e).

Algoritmi za određivanje minimalnog pokrivajućeg stabla

Def: **Minimalno pokrivajuće stablo** - sadrži sve čvorove grafa koji pokriva, ima najmanju moguću sumu težina grana i nema cikluse.

Najpoznatiji algoritmi za pronalazak minimalnog pokrivajućeg stabla su:

1. Kruskalov algoritam
2. Primov algoritam
3. Boruvkin algoritam

Radi prikazivanja python implementaciju svakog algoritma, pokazaću kako će izgledati naša struktura grafa u kodu:

```python
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices  # Broj čvorova u grafu
        self.edges = []  # Lista grana (u obliku: [težina, u, v])

    # Funkcija za pronalaženje korena (uz kompresiju puta)
    def find(self, parent, node):
        if parent[node] != node:
            parent[node] = self.find(parent, parent[node])
        return parent[node]

    # Funkcija za spajanje dva skupa (Union by rank)
    def union(self, parent, rank, u_root, v_root):
        if rank[u_root] < rank[v_root]:
            parent[u_root] = v_root
        elif rank[u_root] > rank[v_root]:
            parent[v_root] = u_root
        else:
            parent[v_root] = u_root
            rank[u_root] += 1
```

KRUSKALOV ALGORITAM

Kruskalov algoritam je algoritam pohlepe, koji sortira grane po težini i dodaje ih redom u minimalno pokrivajuće stablo ako ne formiraju ciklus.

Algoritam radi tako što se grane prvo sortiraju rastuće, prema težini. Proverava se da li grana formira ciklus, i ako ne formira, dodaje se u minimalno pokrivajuće stablo. Ako formira ciklus, preskače se. Provera da li grana formira ciklus se vrši pomoću strukture disjunktnih skupova.

Složenost algoritma: O(nlogn) gde je n broj grana.

Pogodan je za manje grafove i algoritam je jednostavan za implementaciju.

```python
# Implementacija Kruskalovog algoritma
def kruskal(self):
    # Sortiranje grana po težini
    self.edges.sort()

    parent = []
    rank = []

    # Inicijalizacija disjunktnih skupova
    for node in range(self.vertices):
        parent.append(node)
        rank.append(0)

    mst = []  # Lista grana u MST

    # Prošetajmo kroz sve grane
    for edge in self.edges:
        weight, u, v = edge

        # Pronalaženje korena za čvorove u i v
        u_root = self.find(parent, u)
        v_root = self.find(parent, v)

        # Ako u i v nisu u istom skupu, dodajemo granu u MST
        if u_root != v_root:
            mst.append([u, v, weight])
            self.union(parent, rank, u_root, v_root)

    return mst
```

PRIMOV ALGORITAM

Primov algoritam je algoritam pohlepe. On gradi minimalno pokrivajuće stablo koristeći iterativni postupak, u kom stablu dodaje najbližu granu koja povezuje već pokriveni deo grafa sa nepokrivenim.

Algoritam najpre počne od nekog čvora (nije bitno od kog se čvora kreće). Ovaj čvor se sada nalazi u stablu. Zatim dodaje granu najmanje težine koja povezuje čvorove u stablu sa ostalim čvorovima. Dodaje u stablo i čvor sa druge strane te grane. Postupak se nastavlja dok svi čvorovi ne budu u stablu.

Složenost algoritma: O(ElogV) gde je E broj grana, a V broj čvorova.

Pogodan je za grafove koji imaju mnogo grana (mnogo više grana u odnosu na čvorove)

```python
# Implementacija Primovog algoritma
def prim(self):
    mst = []  # Lista grana u MST
    visited = [False] * self.vertices
    edges = []

    # Početak od čvora 0
    visited[0] = True
    for weight, u, v in self.edges:
        if u == 0 or v == 0:
            edges.append([weight, u, v])

    edges.sort()

    while edges:
        weight, u, v = edges.pop(0)
        if visited[u] and visited[v]:
            continue

        mst.append([u, v, weight])
        next_vertex = v if not visited[v] else u
        visited[next_vertex] = True

        for edge_weight, edge_u, edge_v in self.edges:
            if edge_u == next_vertex or edge_v == next_vertex:
                if not visited[edge_u] or not visited[edge_v]:
                    edges.append([edge_weight, edge_u, edge_v])
        edges.sort()

    return mst
```

BORUVKIN ALGORITAM

Boruvkin algoritam je jedan od prvih algoritama za pronalaženje minimalnog pokrivajućeg stabla. Zasniva se na tome da se čvorovi dele u odvojene komponente, koje se spajaju tako što se nađe najmanje teška grana između dve komponente, dok ne ostane samo jedna komponenta.

Algoritam počinje tako što je svaki čvor odvojena komponenta. Za svaku komponentu se pronađe najmanje teška grana i preko nje se dve komponente spajaju. Postupak se ponavlja dok nas konačno spajanje komponenti ne dovede do samo jedne preostale komponente.

Složenost algoritma: O(ElogV)

Ovaj algoritam je pogodan za paralelizaciju, ali je implementacija teža nego kod prethodna dva algoritma

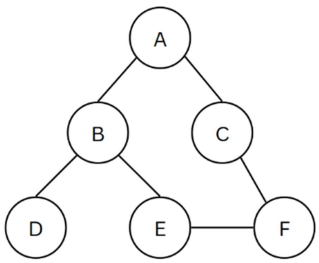
```python
# Implementacija Borůvkinog algoritma
def boruvka(self):
    parent = list(range(self.vertices))
    rank = [0] * self.vertices

    mst = []
    components = self.vertices

    while components > 1:
        cheapest = [-1] * self.vertices

        for weight, u, v in self.edges:
            u_root = self.find(parent, u)
            v_root = self.find(parent, v)
            if u_root != v_root:
                if cheapest[u_root] == -1 or cheapest[u_root][0] > weight:
                    cheapest[u_root] = (weight, u, v)
                if cheapest[v_root] == -1 or cheapest[v_root][0] > weight:
                    cheapest[v_root] = (weight, u, v)

        for node in range(self.vertices):
            if cheapest[node] != -1:
                weight, u, v = cheapest[node]
                u_root = self.find(parent, u)
                v_root = self.find(parent, v)
                if u_root != v_root:
                    mst.append([u, v, weight])
                    self.union(parent, rank, u_root, v_root)
                    components -= 1

    return mst
```

Algoritam za određivanje najkraćeg puta u grafu

Za pronalaženje najkraćeg puta u usmerenom ili neusmerenom grafu možemo koristiti BFS (Breadth first) algoritam.

Kao reprezentaciju grafa u python-u koristićemo rečnik u kom je ključ tipa int i predstavlja vrednost čvora, a vrednost povezana sa tim ključem je lista koja sadrži sve čvorove koji su povezani sa početnim čvorom.

```python
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```



Odabraćemo dva čvora između kojih želimo da nađemo najkraću putanju, recimo A i F. Koristeći BFS algoritam počevši od početnog čvora prolazimo kroz svaki susedni čvor i čim dođemo do krajnjeg čvora, znamo da smo pronašli najkraću putanju. Za detaljnije objašnjenje koda pogledati komentare u kodu.

```python
from collections import deque

def bfs_shortest_path(graph, start, goal):
    # Provera da li su pocetni i krajnji cvor isti
    if start == goal:
        return [start]

    # Inicijalizacija reda sa početnim čvorom
    queue = deque([[start]])
    visited = set()

    while queue:
        # Iz queue uzimamo putanju
        path = queue.popleft()
        node = path[-1]

        # Ako je čvor već posećen, preskočićemo ga
        if node in visited:
            continue

        # Obeležimo čvor kao posećen
        visited.add(node)

        # Za svaki susedni čvor, dodajemo novu putanju u red
        for neighbor in graph.get(node, []):
            new_path = list(path)
            new_path.append(neighbor)
            queue.append(new_path)
            print(f"Current path: {new_path}")

            # Ako smo stigli do krajnjeg čvora, vraćamo putanju
            if neighbor == goal:
                return new_path

    # Ako ne postoji putanja do krajnjeg čvora, vraćamo None
    return None
```