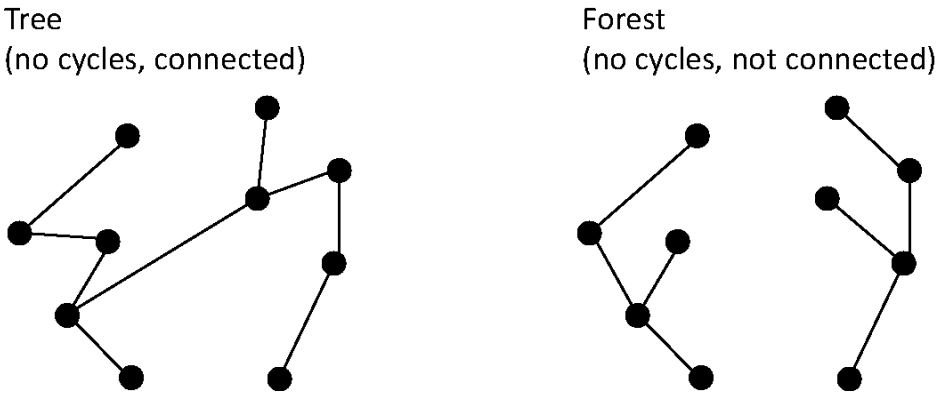
Zadatak 9

Stablo i šuma

Za prost graf G = (V, E) kažemo da je stablo ako važi:

- G je povezan graf
- G je acikličan graf (ne sadrži nijednu konturu)



slika 1. Prikaz stabla i šume

***Teorema 1**:*
Neka je G = (V, E) i |V| = n ≥ 2. Tada je G stablo akko za svaka dva čvora u, v ∈ V postojijedinstven put uv.

Dokaz:
Za n = 2 tvrđenje sledi direktno. Pretpostavimo da je n ≥ 3.

Pretpostavimo suprotno, da u stablu G postoje čvorovi u i v sa osobinom da između njih postoje dva različita u-v puta. Neka su to putevi U_1 i U_2, uz osobinom da $\{u_i,u_{i+1}\}\in U_1$, $\{u_i,u_{i+1}\}\notin U_2$:

$U_1 = uu_1 \dots u_iu_{i+1}\dots u_mv$
$U_2 = uu_1\dots u_nv$

(ako su različiti putevi, onda postoji grana koja pripada jednom, a ne pripada drugom).

Ovde ćemo prikazati slučaj kada je $u,v\notin\{u_i,u_{i+1}\}$, ostali slučajevi se izvode slično. Sada je:

$u_i \dots uu_iv_i\dots u_nv_{mi}v\dots u_{i+1}$

Ako je u_iu_{i+1}šetnja u grafu G - $\{u_i,u_{i+1}\}$. Ako je u grafu G - $\{u_i,u_{i+1}\}$ postoji u_iu_{i+1}šetnja onda postoji i u_iu_{i+1}put. Dodavanjem grane u_iu_{i+1} dobijamo konturu u grafu G, što je u suprotnosti sa pretpostavkom da je G stablo. Ako za svaka dva čvora $u,v\in V$ postoji u-v put onda je G

po definiciji povezan graf. Treba pokazati da je G acikličan. Pretpostavimo suprotno, da u grafu G postoji kontura oblika:

$w_1w_2w_3\dots w_nw_1$

Tada postoje bar dva puta od w_1 do w_2 :

$w_1w_2\dots w_nw_1$, $w_1w_2\dots w_n$
Što je u kontradikciji sa pretpostavkom da za svaka dva čvora postoji jedinstven put od jednog do drugog. To znači da je naša pretpostavka netačna i da je G acikličan graf.

Šuma je neusmereni graf bez ciklusa koji može biti nepovezan. Drugim rečima, šuma je **skup disjunktnih stabala**.

Ključne osobine šume:

- Ako šuma ima kkk komponenti povezanosti (stabala) i nnn čvorova, broj grana je n−k.
- Svaka komponenta šume je stablo.

Stablo i njegove ekvivalentne tvrdnje

Definicija:
Stablo je povezan, prost graf koji ne sadrži cikluse.

Leme:

Prije nego što formulišemo **5 ekvivalentnih tvrdnji** za stablo, objasnićemo **4 leme** koje definišu potrebne i dovoljne uslove za svaka dva ekvivalentna tvrđenja.

Lema 1:

Ako je graf G povezan i ima $|E(G)|=|V(G)|-1$ granu, onda je G stablo.

- Ova lema pokazuje da ako je graf povezan i ima tačno n−1 granu (gde je $n=|V(G)|$), onda ne može sadržatii ciklus.

Lema 2:

Ako između svakog para čvorova $u,v\in V(G)$ postoji jedinstven put, onda je G stablo.

- Graf koji ima jedinstven put između svakog para čvorova mora biti povezan i ne može imati ciklus (jer bi dodatni put stvorio ciklus).

Lema 3:

Ako je G povezan i brisanjem bilo koje grane postaje nepovezan, onda je G stablo.

- Ova lema tvrdi da je G **minimalan povezan graf** (povezanost se gubi kada se bilo koja grana ukloni). To povlači da ima $n-1$ granu i da ne sadrži ciklus.

Lema 4:

Ako je G acikličan i dodavanjem bilo koje grane stvara se ciklus, onda je G stablo.

- Ova lema tvrdi da je G **maksimalan acikličan graf** – ne može se dodati nijedna grana bez stvaranja ciklusa, što znači da je povezan i ima tačno $n-1$ granu.

Pet ekvivalentnih tvrdnji za stablo:

Teorema 78 (Karakterizacija stabla)
Neka je $G=(V,E)$ prost graf. Sledeća tvrđenja su ekvivalentna:

(i) G je stablo.

(ii) Za svaka dva čvora $u,v \in V(G)$ **postoji jedinstven put od u do v.**.

(iii) G je povezan i $|E(G)|=|V(G)|-1$.

(iv) G je povezan i brisanjem proizvoljne grane dobija se nepovezan graf
(tj. G je minimalan povezan graf).

(v) G je acikličan i dodavanjem grane se dobija graf koji sadrži konturu
(tj. G je maksimalan acikličan graf).

Definisati težinski graf

Težinski graf je graf u kome nas ne zanimaju samo čvorovi i grane već i mogućnosti stizanja iz tačke A u tačku B i to na najbolji mogući način. Najbolji način zavisi od problema koji treba riješiti, to je najkraći put, nekada najjeftiniji, put na kome se troši najmanje energije i sl. Iz tih razloga svakoj grani se dodjeljuje realan broj, njegova težina, odnosno mjera. Ako želimo, na primjer, da nađemo najkraći put između gradova težina je udaljenost, ili cijena avionske karte koja spaja udaljene gradova i sl. Težina ne mora da bude pozitivan broj, ali uobičajeno je da

se takav koristi, ne umanjujući opštost razmatranja. Ako neka grana ne postoji, tada se na pomenutu poziciju stavlja neki poseban simbol na primjer ∞.

Težinski graf je graf u kojem je svakoj grani ili čvoru pridružena težina, odnosno brojčana vrijednost koja predstavlja određenu osobinu, kao što su udaljenost, trošak, kapacitet ili snaga veze. Formalno, težinski graf se može definisati kao uređena trojka:

$$G = (V, E, w)$$

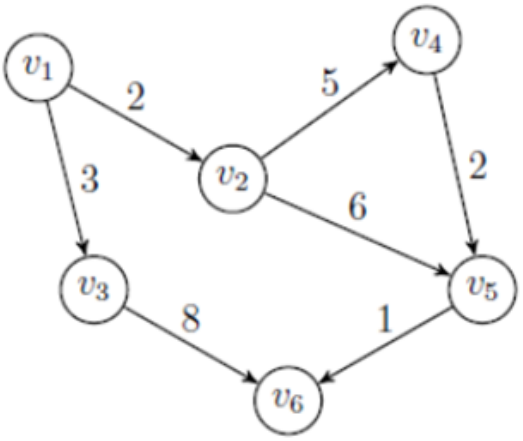
gdje su:

1. V: skup čvorova,
2. E: skup grana, gde je svaka grana uređeni par (u,v), pri čemu $u,v \in V$,
3. $w: E \to R$: funkcija težina koja svakoj grani $e \in E$ dodeljuje težinu $w(e)$.

Težinski graf koji je usmjeren zove se mreža.

Upotreba težinskih grafova:

1. **Putanja najmanje težine**: Algoritmi poput Dijkstrinog ili Bellman-Fordovog za pronalaženje najkraće putanje.
2. **Maksimalni protok**: Algoritmi za analizu kapaciteta mreža.
3. **Optimizacija**: Korišćenje težina za rešavanje problema kao što su putujući trgovac ili minimalno razapinjuće stablo (Kruskalov ili Primov algoritam).



SLIKA 2. Težinski usmjereni graf

Priferov kod:

Priferov kod je način kodiranja stabala pomoću niza brojeva. Ovaj kod pruža kompaktnu reprezentaciju stabala sa nnn čvorova koristeći niz dužine n−2n-2n−2. Ova reprezentacija je korisna u teoriji grafova i algoritmima.

Definicija

Priferov kod pretvara **prosto stablo** (usmereni ili neusmereni graf koji je povezan i ne sadrži cikluse) sa n čvorova u niz dužine n−2.

Ulaz:

Stablo sa n čvorova, gde su čvorovi numerisani od 1 do n.

Izlaz:

Niz P=[p1,p2,…,pn−2], koji je **Priferov kod** stabla.

Ključne osobine Priferovog koda:

1. **Jedinstvenost**: Svako stablo sa nnn čvorova ima jedinstven Priferov kod dužine n−2.
2. **Broj stabala**: Broj različitih stabala sa n čvorova je n^{n-2} (Cayleyjeva formula).
3. **Efikasnost**: Priferov kod omogućava jednostavnu rekonstrukciju stabla.

Primer 1: Stablo → Priferov Kod

Razmotrite stablo sa čvorovima {1,2,3,4,5,6} i sledećim ivicama: {(1,2),(1,3),(3,4),(3,5),(5,6)}

Koraci za generisanje Priferovog koda:

1. Listovi su {2,4,6}. Najmanji list je 2. Sused 1 se dodaje u kod. Uklanjamo 2.
2. Sada je stablo: {1,3,4,5,6}. Listovi su {4,6,1}. Najmanji list je 4. Sused 3 se dodaje u kod. Uklanjamo 4.
3. Sada je stablo: {1,3,5,6}. Listovi su {6,1}. Najmanji list je 6. Sused 5 se dodaje u kod. Uklanjamo 6.
4. Sada je stablo: {1,3,5}. Listovi su {1,5}. Najmanji list je 1. Sused 3 se dodaje u kod. Uklanjamo 1.
5. Ostaje stablo: {3,5}. List je 5. Sused 3 se dodaje u kod. Uklanjamo 5.

Priferov kod: {1,3,5,3}.

Primer 2: Priferov Kod → Stablo

Dati Priferov kod: {1,3,5,3}.

Koraci za rekonstrukciju stabla:

1. Napravimo listu svih čvorova: {1,2,3,4,5,6}.
2. Brojimo pojavljivanja svakog čvora u Priferovom kodu:
    - 1:1, 3:2, 5:1, ostali (2,4,6):0.
3. Pronađemo najmanji čvor koji nije u Priferovom kodu (sa 0 pojavljivanja) i povežemo ga sa prvim čvorom iz koda:
    - Povezujemo 2 sa 1. Uklanjamo 1 iz koda i smanjujemo broj pojavljivanja 1.
4. Ažuriramo: {3,5,3}, pojavljivanja {3:2,5:1,4:0,6:0}.
    - Povezujemo 4 sa 3. Uklanjamo 3 iz koda i smanjujemo broj pojavljivanja 3.
5. Ažuriramo: {5,3}, pojavljivanja {5:1,3:1,6:0}.
    - Povezujemo 6 sa 5. Uklanjamo 5 iz koda i smanjujemo broj pojavljivanja 5.
6. Ažuriramo: {3}, pojavljivanja {3:1}.
    - Povezujemo 1 sa 3. Uklanjamo 3 iz koda.
7. Ostaju čvorovi 3 i 5. Povezujemo ih.

Rekonstruisano stablo: {(1,2),(3,4),(5,6),(1,3),(3,5)}.

Minimalno pokrivajuće stablo (MST)

Minimalno pokrivajuće stablo grafa je podskup grana povezanog grafa koji uključuje sve čvorove i nema cikluse, a suma težina grana u tom podskupu je minimalna.

1. Kruskalov algoritam

Kruskalov algoritam koristi **strategiju "pohlepnog" pristupa** za gradnju MST-a.

Koraci algoritma:

1. Sortiraj sve grane u grafu prema težini u rastućem redosledu.
2. Iterativno dodaj najmanju granu u stablo, pod uslovom da ne formira ciklus.
3. Za proveru ciklusa koristi se **Union-Find struktura** (disjunktni skupovi).

Složenost:

- **Sortiranje grana:** O(ElogE),
- **Union-Find operacije:** $O(E \cdot \alpha(V))$, gde je α inverzna Akermanova funkcija.

Ukupna složenost: O(ElogE).

Prednosti:

- Efikasan za retke grafove (kada je broj grana manji od |V|^2).

- Jednostavan za implementaciju.

Mane:

- Nije idealan za grafove sa velikim brojem čvorova i gusto povezanim strukturama.

2. Primov algoritam

Primov algoritam takođe koristi **pohlepnu strategiju**, ali gradi stablo iterativno, šireći ga iz jednog čvora.

Koraci algoritma:

1. Započni iz proizvoljnog čvora.
2. Dodaj najmanju granu koja povezuje čvor izvan stabla sa nekim čvorom unutar stabla.
3. Ponavljaj dok svi čvorovi ne budu uključeni.

Složenost:

- **Korišćenjem heap-a:** O(ElogV),
- **Korišćenjem matrice susedstva:** (V^2).

Prednosti:

- Pogodan za guste grafove (kada je broj grana blizu |V|^2).
- Prirodno širi stablo, što ga čini lakšim za implementaciju u aplikacijama gde je potrebno proširenje mreže.

Mane:

- Manje efikasan za retke grafove.

3. Boruvkin algoritam

Boruvkin algoritam je jedan od najstarijih algoritama za MST. Radi iterativno, nalik Kruskalovom algoritmu.

Koraci algoritma:

1. Započni sa svim čvorovima kao zasebnim grupama.
2. U svakoj iteraciji, za svaki čvor, odaberi granu minimalne težine koja povezuje njegovu komponentu sa drugom komponentom.
3. Nastavi dok sve komponente ne budu spojene u jedno stablo.

Složenost:

- U naivnoj implementaciji: O(VE),
- Uz efikasne strukture podataka: O(ElogV).

Prednosti:

- Lako se paralelizuje.
- Pogodan za distribuirane sisteme.

Mane:

- Manje efikasan od Kruskalovog i Primovog algoritma za standardne implementacije.

4. Sollinov (Parallel Boruvka) algoritam

Sollinov algoritam je varijanta Boruvkinog algoritma optimizovana za paralelno računanje.

Koraci:

1. Početno sve komponente čvorova su nezavisne.
2. Svaka komponenta nalazi najbližu granu.
3. Spajanje se vrši paralelno za sve komponente.
4. Proces se ponavlja dok ne ostane jedna komponenta.

Složenost:

- Efikasno paralelno izvođenje.

Prednosti:

- Idealan za velike grafove u distribuiranom okruženju.

Mane:

- Implementacija je složenija u odnosu na sekvencijalne algoritme.

Algoritam	**Strategija**	**Složenost**	**Prednosti**	**Mane**
Kruskal	Grabežljiva	O(ElogE)	Efikasan za retke grafove	Ciklus-provera može biti skupa
Prim	Grabežljiva	O(ElogV)	Efikasan za guste grafove	Teži za implementaciju za velike mreže
Boruvka	Iterativna	O(ElogV)	Lako paralelizuje	Sporiji za sekvencijalno izvođenje

Sollin	Paralelna	O(ElogV)	Idealan za velike grafove	Kompleksnija implementacija

Kruskalov algoritam:

```python
class UnionFind:
    def __init__(self, size):
        self.parent = list(range(size))
        self.rank = [0] * size

    def find(self, node):
        if self.parent[node] != node:
            self.parent[node] = self.find(self.parent[node])  # Path compression
        return self.parent[node]

    def union(self, node1, node2):
        root1 = self.find(node1)
        root2 = self.find(node2)

        if root1 != root2:
            # Union by rank
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1
            return True
        return False
```

```python
def kruskal(graph, num_nodes):
    """
    graph: List of edges (u, v, weight)
    num_nodes: Number of nodes in the graph
    Returns: Minimum Spanning Tree (list of edges) and its total weight
    """
    # Sort edges by weight
    graph.sort(key=lambda edge: edge[2])

    uf = UnionFind(num_nodes)
    mst = []
    total_weight = 0

    for u, v, weight in graph:
        if uf.union(u, v):  # If adding this edge doesn't form a cycle
            mst.append((u, v, weight))
            total_weight += weight

    return mst, total_weight
```

```python
if __name__ == "__main__":
    graph = [
        (0, 1, 10),
        (0, 2, 6),
        (0, 3, 5),
        (1, 3, 15),
        (2, 3, 4)
    ]
    num_nodes = 4

    mst, total_weight = kruskal(graph, num_nodes)
    print("Minimalno pokrivajuće stablo:", mst)
    print("Ukupna težina MST:", total_weight)

```

Rezultat :

```
Minimalno pokrivajuce stablo: [(2, 3, 4), (0, 3, 5), (0, 1, 10)]
Ukupna tezina MST: 19
```

Algoritam za odrejdivanje najkraceg puta u grafu

Za određivanje najkraćeg puta u grafu postoje različiti algoritmi u zavisnosti od tipa grafa (težinski, netežinski, usmereni, neusmereni) i uslova (negativne težine grana ili ne).

Evo nekoliko poznatih algoritama:

1. **Dijkstra**: Za grafove sa ne-negativnim težinama grana.
2. **Bellman-Ford**: Za grafove koji mogu imati negativne težine grana.
3. **Floyd-Warshall**: Za pronalaženje najkraćeg puta između svih parova čvorova.
4. **BFS (Breadth-First Search)**: Za netežinske grafove.

Algoritam: Dijkstra (za ne-negativne težine)

Opis

Dijkstra-ov algoritam pronalazi najkraći put od jednog izvornog čvora sss do svih ostalih čvorova u grafu. Radi isključivo za grafove sa ne-negativnim težinama grana.

Ulaz:

- Graf predstavljen listom susedstva ili matricom težina.
- Izvorni čvor s.

Izlaz:

- Najkraće udaljenosti od s do svih ostalih čvorova.
- Najkraći putevi (opciono).

Koraci algoritma:

1. Postavite udaljenost izvornog čvora sss na 0, a svih ostalih čvorova na ∞ (beskonačnost).
2. Kreirajte skup S posetjenih čvorova, inicijalno prazan.
3. Dok svi čvorovi nisu obrađeni:
    - Izaberite čvor uuu sa najmanjom trenutnom udaljenošću koji nije u S.
    - Dodajte u u S.
    - Za svakog suseda v čvora u:
        - Ažurirajte udaljenost do v ako je kraća preko u: udaljenost[v] = min(udaljenost[v], udaljenost[u] + težina(u,v)) Kada su svi čvorovi obrađeni, najkraće udaljenosti su poznate.

Implementacija u programu:

```python
import heapq

def dijkstra(graph, source):
    # graph: dict {node: [(neighbor, weight), ...]}
    # source: početni čvor

    # Inicijalizacija
    distances = {node: float('inf') for node in graph}
    distances[source] = 0
    priority_queue = [(0, source)]  # (trenutna udaljenost, čvor)
    previous_nodes = {node: None for node in graph}  # Za rekonstrukciju puta

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Ako je trenutna udaljenost veća od poznate, preskočite
        if current_distance > distances[current_node]:
            continue

        # Obradite sve susede
        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight

            # Ako je kraći put pronađen
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances, previous_nodes

# Rekonstrukcija najkraćeg puta
def reconstruct_path(previous_nodes, target):
    path = []
    while target is not None:
        path.append(target)
        target = previous_nodes[target]
    return path[::-1]
```