Softverski algoritmi u sistemima automatskog upravljanja

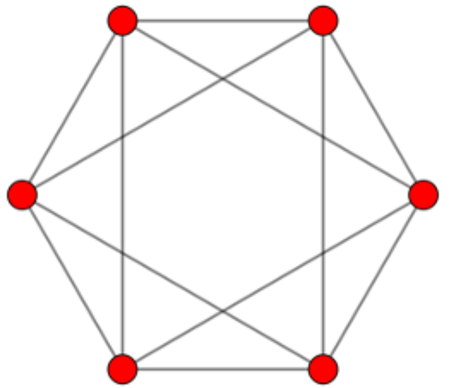
Grafovi 1

Danilo Kaćanski, Ivan Radman

Graf

❓ Šta je graf?

- Struktura koje se sastoji od **čvorova (V)** i **grana (E)**, koristi se za modelovanje odnosa između objekata
- Matematički zapis: $G = (V, E)$
- Red grafa: $|V|$ — **broj čvorova**
- Veličina grafa: $|E|$ — **broj grana**



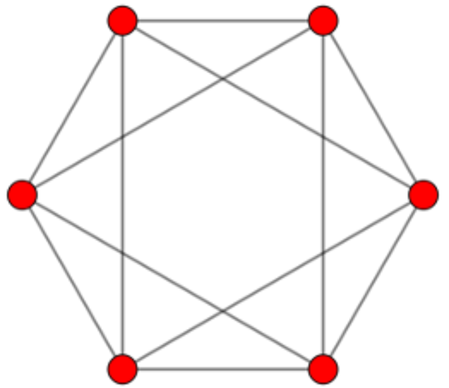
$$|V| = ?$$

$$|E| = ?$$

Graf

❓ Šta je graf?

- Struktura koje se sastoji od **čvorova (V)** i **grana (E)**, koristi se za modelovanje odnosa između objekata
- Matematički zapis: $G = (V, E)$
- Red grafa: $|V|$ — **broj čvorova**
- Veličina grafa: $|E|$ — **broj grana**



$$|V| = 6$$

$$|E| = 12$$

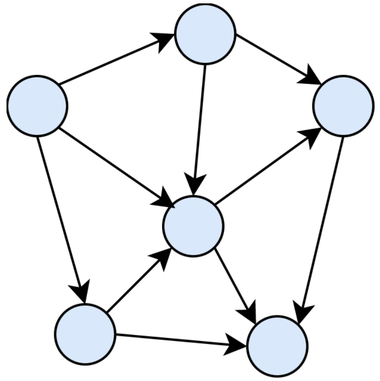
🌐 Gde se grafoci koriste?

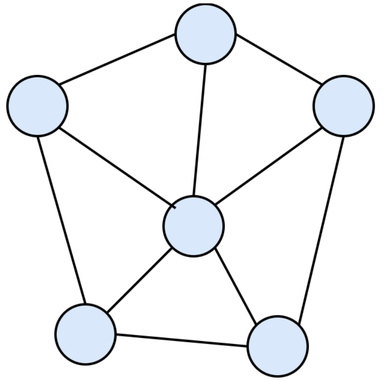
- **Društvene mreže** — korisnici su čvorovi, prijateljstva su grane
- **Google maps** — mesta su čvorovi, putevi su grane
- **Web stranice** — stranice su čvorovi, linkovi su grane
- **Operativni sistemi** — procesi su kao čvorovi, zavisnosti kao grane

Podela grafova prema osobinama

🔁 Po orjentaciji grana:

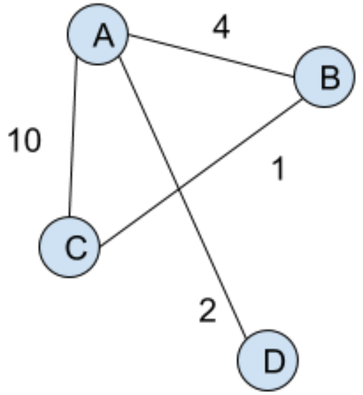
- **Usmereni** graf (*directed graph / digraph*)
  - Grane su **uređeni parovi** čvorova *(u,v)*
- **Neusmereni** graf (*undirected graph*)
  - Grane su **neuređeni parovi** čvorova *{u,v}*

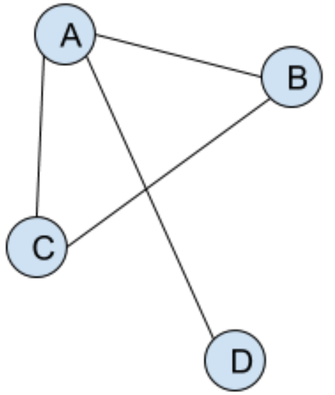




⚖️ Po postajanju težina na granama:

- **Težinski** graf (*weighted graph*)
  - Grane imaju dodeljene numeričke vrednosti
- **Besteżinski** graf (*unweighted graph*)
  - Grane su samo veze, bez težina

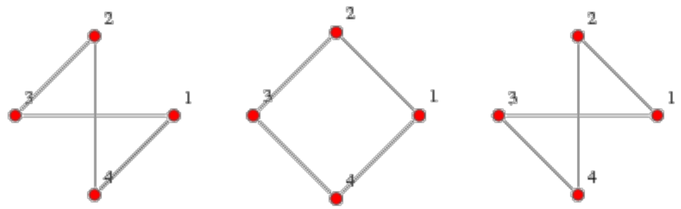




Reprezentacija grafova u Python-u

1. Matrica susedstva

- Graf se predstavlja kao 2D lista — matrica

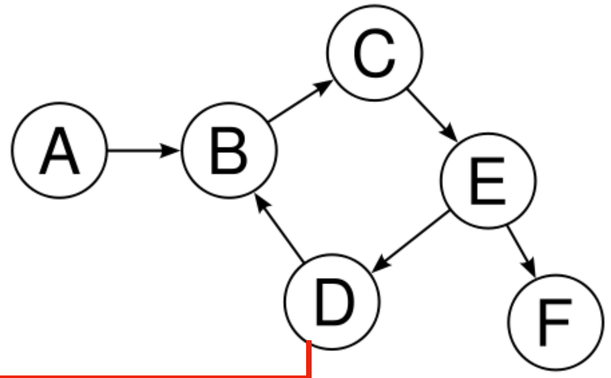


$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

```
graph = [
  [0,1,0,0,0,0],
  [0,0,1,0,0,0],
  [0,0,0,0,1,0],
  [0,1,0,0,0,0],
  [0,0,0,1,0,1],
  [0,0,0,0,0,0]
]
```

2. Lista susedstva (najčešće korišćena)

- Graf se predstavlja kao `dict`, gde su ključevi čvorovi, a vredosti liste susednih čvorova




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

Graf — Lista susedstva

Reprezentacija grafa kao `dict`:

```
graph = {}
```

Dodavanje grane u bestežinski graf:

```
def dodaj_granu(cvor1, cvor2):
    if cvor1 not in graph:
        graph[cvor1] = []
    if cvor2 not in graph:
        graph[cvor2] = []
    graph[cvor1].append(cvor2)
```

Primer:

```
dodaj_granu('A', 'B')
dodaj_granu('A', 'C')
```

Dodavanje grane u težinski graf:

```
def dodaj_granu(cvor1, cvor2, tezina):
    if cvor1 not in graph:
        graph[cvor1] = []
    if cvor2 not in graph:
        graph[cvor2] = []
    graph[cvor1].append((cvor2, tezina))
```

Primer:

```
dodaj_granu('A', 'B', 5)
dodaj_granu('A', 'C', 9)
```

📝 Napomena:

- Čvorovi se **automatski dodaju** ako ne postoje
- U težinskom grafu grane se čuvaju kao `(cvor, tezina)` parovi

Obilazak grafa

Kada imamo graf, često želimo da ga obiđemo — da posetimo sve čvorove i/ili grane, obično krećući iz jednog čvora

🔍 Zašto obilazimo graf?

- Da proverimo da li postoji put između čvorova
- Da pronađemo najkraći put
- Da obradimo sve čvorove u nekom redosledu
- Da detektujemo cikluse, komponente povezanosti itd.

📌 Dve osnovne strategije:

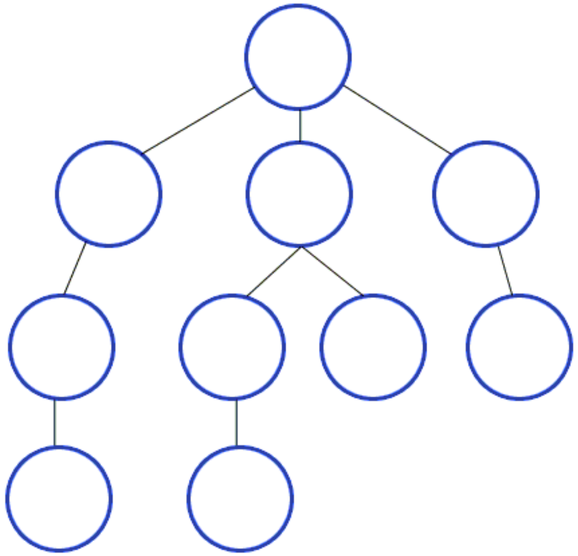
Strategija	Kratko objašnjenje
BFS (*Breadth-First Search*)	Obilazi nivo po nivo (širinu) — koristi red (`queue`)
DFS (*Depth-First Search*)	Ide što dublje može pre nego što se vraća — koristi stek (`stack`) ili rekurziju

BFS (*Breadth-First Search*)

Najjednostavniji algoritam pretrage grafa

❇️Opis:

- Počinje od **početnog čvora** i istražuje **susede pre nego što pređe** na sledeći nivo
- Koristi **red** (*queue*) da bi zadržao redosled obilaska
- Idealno za **pronalaženje najkraćeg puta** u **ne-težinskim grafovima**



🧠Osnovna ideja:

- Prati koje je čvorove **posetio** i koje još treba da **poseti** (*FIFO struktura*)

⏱️Vremenska složenost:

- $O(V + E)$, gde je ***V*** broj čvorova, a ***E*** broj grana

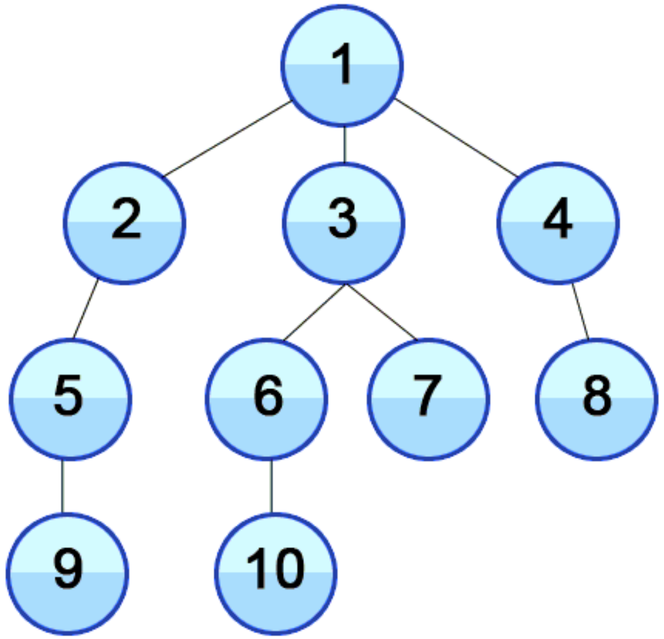
🧩Korisno za:

- Detekciju povezanosti, računanje rastojanja, provere puta između čvorova

BFS (*Breadth-First Search*)

🔍**Zadatak:**

- Implementirati BFS algoritam i testirati ga na sledećem grafu:



🔄Oblilazak treba da ispiše sve čvorove **po nivoima** počevši od čvora 3

BFS (*Breadth-First Search*)

✅ BFS:

```python
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(graph[node])
```

🗺️ Reprezentacija grafa sa slike:

```python
graph = {
    1: [2, 3, 4],
    2: [5],
    3: [6, 7],
    4: [8],
    5: [9],
    6: [10],
    7: [],
    8: [],
    9: [],
    10: []
}
```

▶️ Poziv funkcije:

```python
bfs(graph, 1)
```

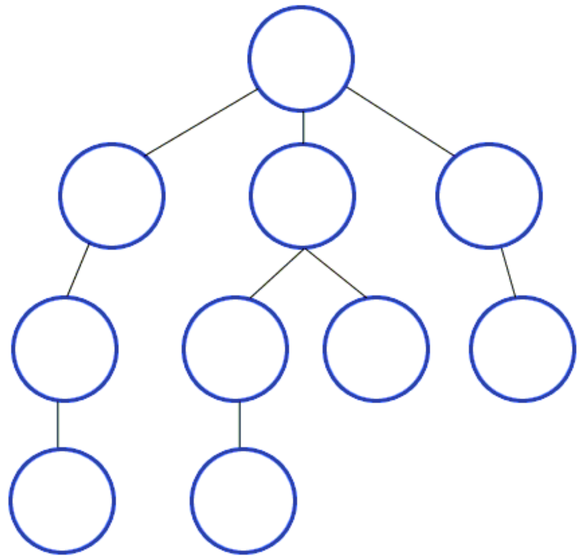
📤 Očekivani izlaz:

```
1 2 3 4 5 6 7 8 9 10
```

DFS (*Depth-First Search*)

✳️ Opis:

- **Rekurzivno** ili uz pomoć **steka**
- Počinje od početnog čvora i **istražuje sve do kraja puta** pre nego što se vrati nazad
- Koristi **stek** (*stack*) da bi zapamtio gde treba da se vrati



🧠 Osnovna ideja:

- Prati koji čvorovi su **posećeni**
- Koristi *LIFO* logiku — **poslednji dodat, prvi se obrađuje**

⏱️ Vremenska složenost:

- $O(V + E)$, gde je ***V*** broj čvorova, a ***E*** broj grana

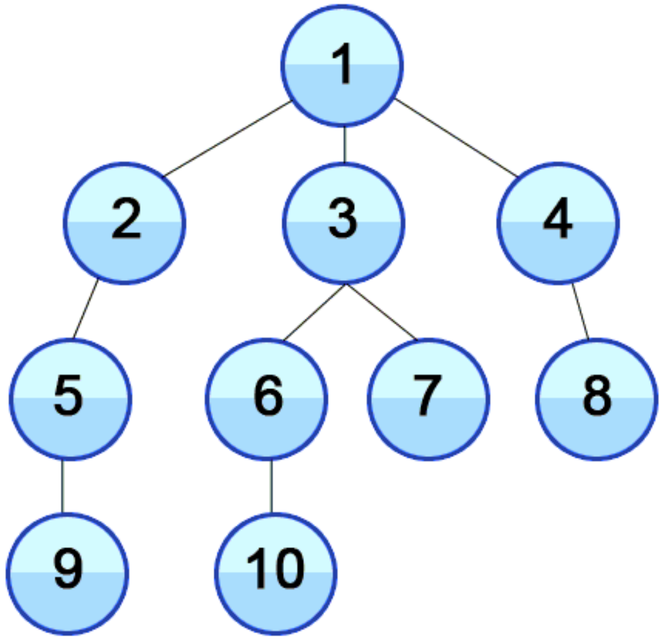
🧩 Korisno za:

- Detekcija ciklusa, kompnente povezanosti, topološko soritraje

DFS (*Depth-First Search*)

🔍**Zadatak:**

- Implementirati DFS algoritam i testirati ga na sledećem grafu:



DFS (*Depth-First Search*)

✅ DFS:

```
def DFS(visited, graph, cvor):
    if cvor not in visited:
        print(cvor)
        visited.append(cvor)
        for sused in graph[cvor]:
            DFS(visited, graph, sused)
```

🗺️ Reprezentacija grafa sa slike:

```
graph = {
    1: [2, 3, 4],
    2: [5],
    3: [6, 7],
    4: [8],
    5: [9],
    6: [10],
    7: [],
    8: [],
    9: [],
    10: []
}
```

▶️ Poziv funkcije:

```
visited = []
DFS(visited, graph, 1)
```

📤 Očekivani izlaz:

```
1 2 5 9 3 6 10 7 4 8
```

BFS vs DFS

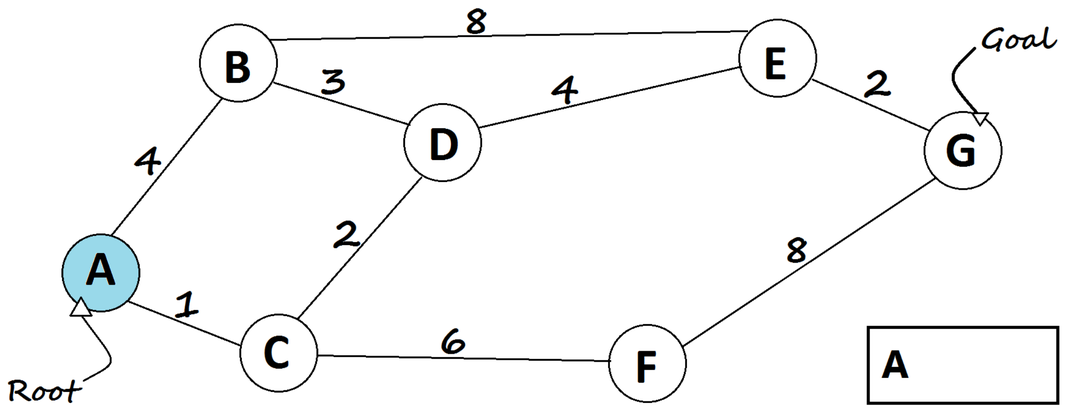
Osobina	BFS (Breadth-First Search)	DFS (Depth-First Search)
Princip	*FIFO* (red)	*LIFO* (stek ili rekurzija)
Redosled obilaska	Istražuje čvorove blizu izvornog	Istražuje čvorove što dalje od izvornog
Brzina (u teoriji)	Sporiji u širokim grafovima	Brži, ali može da ide previše u dubinu
Zahtev memorije	Više memorije (pamti sve susede po nivou)	Manje memorije (ide u dubinu)
Optimalnost puta	Pronalazi najkraći put (u ne-težinskim grafovima)	Ne garantuje najkraći put
Opasnost od petlje	Ne može da uđe u beskonačnu petlju (ako se pamte posećeni)	Može da uđe ako se ne pamti stanje
Pogodan za	Najkraći put, pretraga u širinu	Otkrivanje komponente, topološko sortiranje
Može da se koristi za	Nivoe grafa, minimum koraka	Provera ciklusa, generisanje lavirinata
Implementacija	Lako uz red i listu posećenih	Lako rekurzivno ili pomoću steka

BFS vs DFS

Pronalazak najkraćeg puta

🔍Cilj:

- Pronaći **najkraći put u težinskom grafu** od polaznog čvora do svih ostalih čvorova



🧠Gde se koristi:

- **Navigacija** (*npr. Google Maps*)
- **Mreže** (*npr. Najmanji broj hopova*)
- **Planiranje i optimizacija** (*npr. trošak, vreme*)

📌Napomena:

- Ako je **graf bez težina**, koristi se **BFS**
- Ako se **težine pozitivne**, koristi se **Dijkstrin algoritam**
- Ako graf sadrži **negativne težine**, koristi se **Bellman-Ford algoritam**

🛠️Napomena:

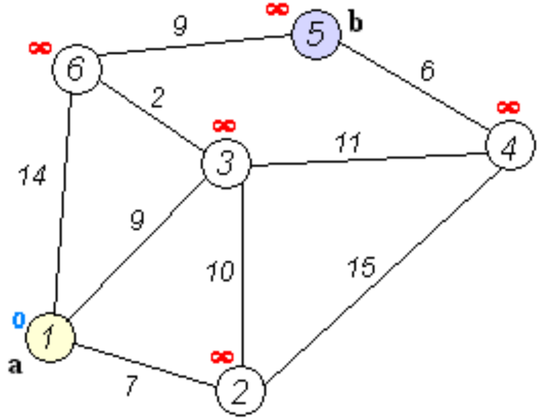
- **Dijkstra** → Brz, ali ne radi sa negativnim težinama
- **Bellman-Ford** → Sporiji, ali radi i sa negativnim težinama

Dijkstra

Osnovni algoritam za pronalaženje najkraćeg puta u grafu sa pozitivnim težinama

🛠️Ideja:

- Čuva se skup *neposećenih* čvorova
- Počinje se od *izvornog* čvora i njegova **udaljenost** se postavlja na **0**
- Svi ostali čvorovi se inicijalizuju na beskonačno (*inf*)
- U svakom koraku se bira čvor sa **najmanjom trenutno poznatom udaljenošću**
- Ažuriraju se udaljenosti do njegovih suseda
- Čvor se zatim označava kao *posećen*
- Postupak se ponavlja dok svi čvorovi ne budu posećeni



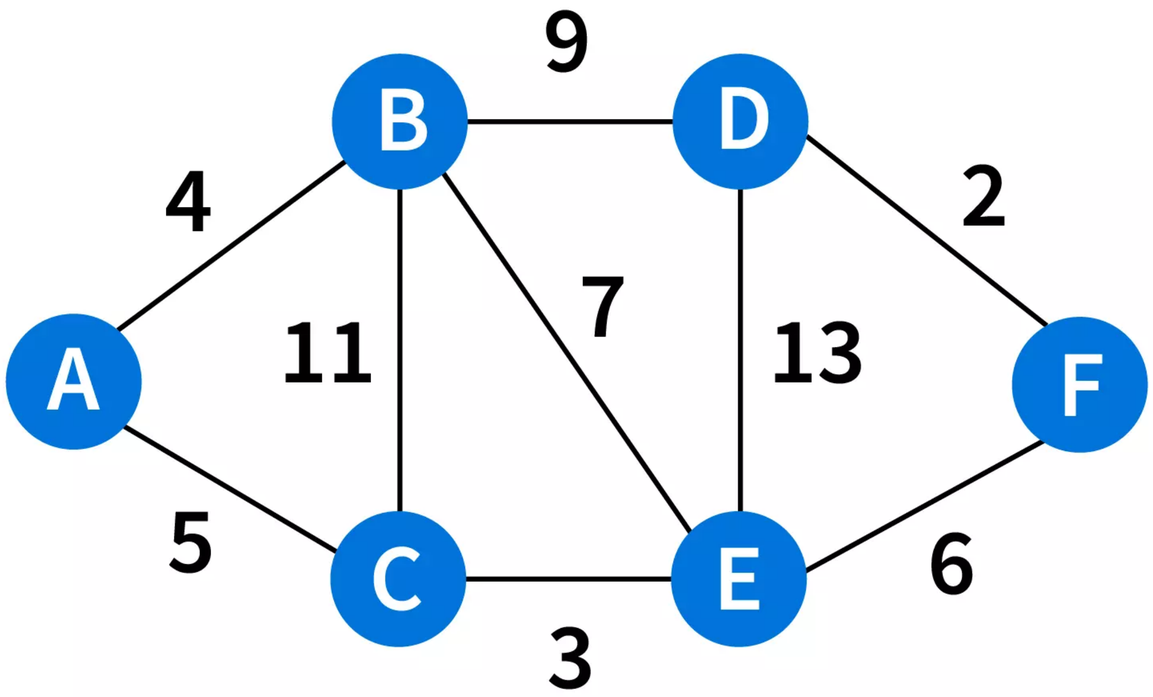
⏱️Kompleksnost:

- **Bez prioriteta:** $O(V^2)$
- **Sa prioritetnim redom** (*heap*)**:** $O(E + V \cdot log\ V)$

Dijkstra

🔍**Zadatak:**

- Implementirati Dijkstra algoritam i testirati ga na sledećem grafu:



Dijkstra

✅Dijkstra:

```python
def dijkstra(graph, source):
    unvisited = graph.copy()
    dist = {}

    for cvor in unvisited:
        dist[cvor] = math.inf
    dist[source] = 0

    while unvisited:
        min_cvor = None
        for cvor in unvisited:
            if min_cvor is None or dist[cvor] < dist[min_cvor]:
                min_cvor = cvor

        for sused, tezina in unvisited[min_cvor]:
            if dist[min_cvor] + tezina < dist[sused]:
                dist[sused] = dist[min_cvor] + tezina

        unvisited.pop(min_cvor)

    print("Udaljenost čvorova od početnog čvora:")
    for cvor in dist:
        print(f"{cvor}\t→\t{dist[cvor]}")
```

🗺️Reprezentacija grafa sa slike i poziv funkcije:

```python
graph = {
    'A': [('B', 4), ('C', 5)],
    'B': [('C', 11), ('D', 9)],
    'C': [('E', 3)],
    'D': [('E', 7), ('F', 2)],
    'E': [('F', 6)],
    'F': []
}
dijkstra(graph, 'A')
```

📤Očekivani izlaz:

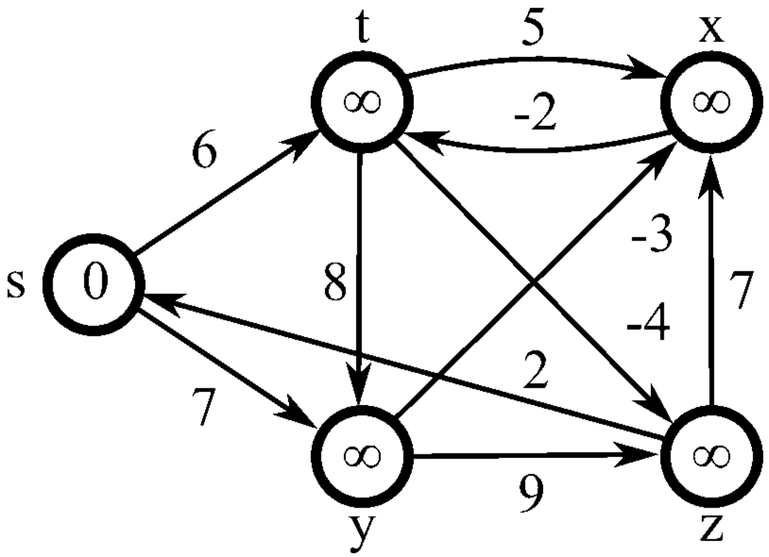
```
Udaljenost čvora od početnog čvora:
A       0
B       4
C       5
D       13
E       8
F       14
```

Bellman-Ford

Algoritam za pronalaženje najkraćeg puta u grafu koji može sadržati negativne težine

🛠️Ideja:

- Udaljenosti se **inicijalizuju na beskonačno** (*inf*), osim za izvorni čvor (*postavlja se na 0*)
- Za razliku od Dijsktre, algoritam **ne koristi skup posećenih čvorova**
- Tokom *V-1* iteracija, **za svaku granu** pokušava se ažuriranje udaljenosti
- Na kraju, dodatnom proverom se otkriva da li postoji **negativni ciklus**



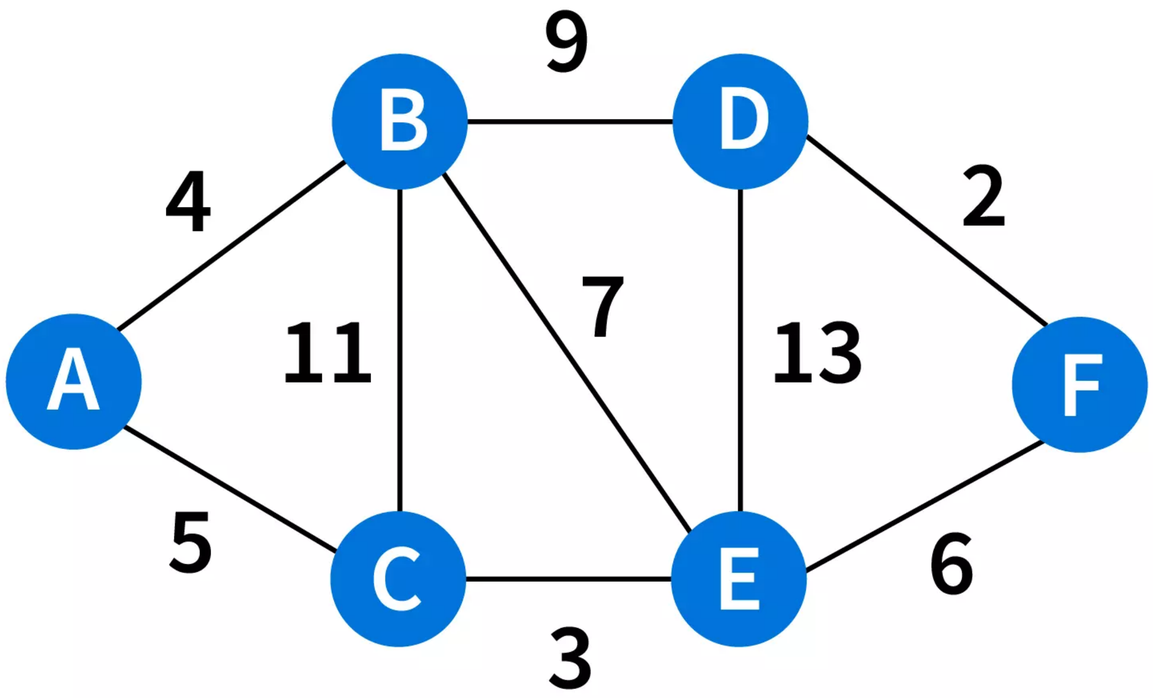
⏱️Kompleksnost:

- **Bez unapređenja:** $O(V \cdot E)$

Bellman-Ford

🔍**Zadatak:**

- Implementirati Bellman-Ford algoritam i testirati ga na sledećem grafu:



Bellman-Ford

✅Bellman-Ford

```python
def bellmanFord(graph, source):
    dist = {node: math.inf for node in graph}
    dist[source] = 0

    for _ in range(len(graph) - 1):
        for u in graph:
            for v, weight in graph[u]:
                if dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight

    # Provera negativnog ciklusa
    for u in graph:
        for v, weight in graph[u]:
            if dist[u] + weight < dist[v]:
                print("Graf sadrži negativnu kružnu putanju!")
                return

    print("Najkraće rastojanje od čvora", source)
    for node in dist:
        print(f"{node}: {dist[node]}")
```

🗺️Reprezentacija grafa sa slike i poziv funkcije:

```python
graph = {
    'A': [('B', 4), ('C', 5)],
    'B': [('D', 9), ('C', 11)],
    'C': [('D', 7), ('E', 3)],
    'D': [('F', 2)],
    'E': [('F', 6)],
    'F': []
}
bellmanFord(graph, 'A')
```

📤Očekivani izlaz:

```
Udaljenost čvora od početnog čvora:
A       0
B       4
C       5
D       13
E       8
F       14
```