**SVEUČILIŠTE U ZAGREBU**
**FAKULTET ORGANIZACIJE I INFORMATIKE**
**V A R A Ž D I N**

**Jakov Begović**

# APSTRAKTNI TIP PODATAKA GRAF

**SEMINARSKI RAD**

**Varaždin, 2023.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Jakov Begović**

**Studij: Informacijski i poslovni sustavi**

# APSTRAKTNI TIP PODATAKA GRAF

## SEMINARSKI RAD

**Mentor:**

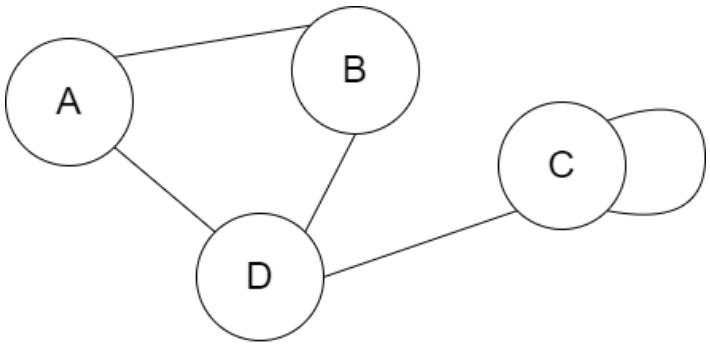
Prof. dr. sc. Alen Lovrenčić

**Varaždin, siječanj 2023.**

# 1. Sadržaj

# 2. Uvod

Graf (engl. graph) je apstraktni tip podatka, kraće ATP, koji opisuje odnose između objekata. Njega koristimo kako bi pojednostavili neki objekt iz stvarnog života. On sadrži čvorove (drugim riječima točke ili vrhove, engl. *vertex*) i bridove (drugim riječima veze, engl. *edge*). Njegovi čvorovi (oznaka V) i bridovi (oznaka E) ga definiraju.



Slika 1: primjer ATP-a graf

Za graf sa slike 1 formalno vrijedi:

$V = \{A, B, C, D\}$

$E = \{ (A, B), (A, D), (B, D), (D, C), (C, C) \}$

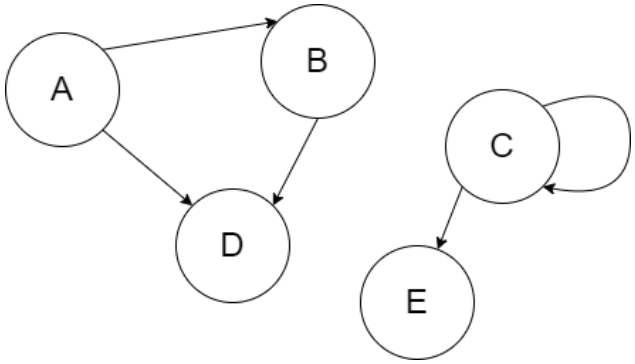
Gore prikazan graf ima 4 čvora i 5 bridova. Vrijedi da je $|V| = 4$ , a $|E| = 5$. Brid (C,C) nazivamo petljom jer počinje i završava u istom čvoru.

Razlikujemo više vrsta grafova. Različite vrste grafova služe za predstavljanje različitih objekata u stvarnome svijetu. Jednostavni grafovi nemaju višestruke bridove između istih točaka i nemaju petlje. Graf koji ne zadovoljava ove kriterije naziva se opći graf. Povezani graf je graf koji ima jednu povezanu komponentu. Nepovezani ne zadovoljavaju ovaj kriterij. Potpuni graf sadrži maksimalan broj bridova. Odnosno, $|V| \times (|V| - 1)/2$ bridova. Stablo je najrjeđi povezani graf.

Razlikujemo i usmjerene i neusmjerene grafove, te težinske i netežinske grafove. Težinski grafovi bridovima pridružuju težinu. Težina može predstavljati razne vrijednosti iz stvarnog života. Slika 1 ilustrira neusmjereni graf. U usmjerenom grafu svakim bridom se može proći samo jednom. Na slici 2 je prikazana ilustracija usmjerenog, nepovezanog, općeg grafa.

Primjeri korištenja grafova:

- Predstavljanje gradova i udaljenosti između njih
- Predstavljanje zračnih linija i cijene njihovog korištenja
- Predstavljanje toka rijeka
- Predstavljanje međuljudskih odnosa i prijateljskih poveznica

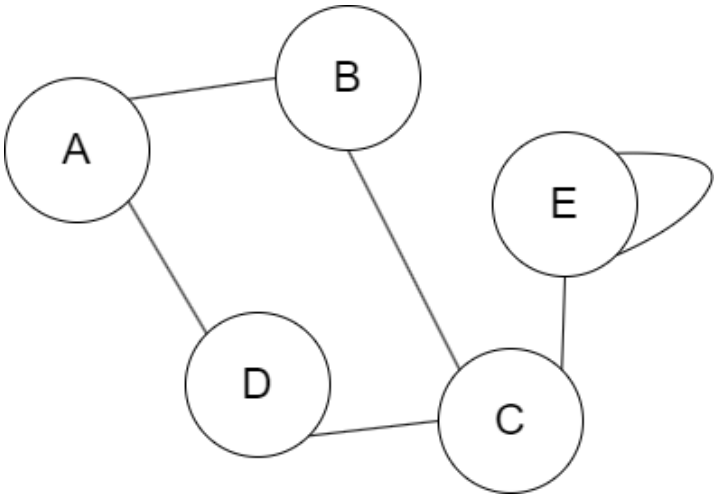



Slika 2: primjer ATP-a graf

# 3. Razrada teme

## 3.1. Implementacije tipa podatka

ATP graf se može na razne načine predstaviti u računalu. Moguća je implementacija pomoću matrice susjedstva, matrice incidencije i listom susjedstva.




Slika 3 – primjer neusmjerenog netežinskog grafa

### 3.1.1. Matrica susjedstva

Kod implementacije pomoću matrice susjedstva vrijednost na poziciji $M_{i,j}$ matrice M ukazuje na odnos između čvorova i i j. Kod neusmjerenih grafova vrijednosti na pozicijama $M_{i,j}$ i $M_{j,i}$ imaju istu vrijednost. Isto ne mora vrijediti kod usmjerenog grafa.

Tablica 1: prikaz grafa sa slike 3 pomoću matrice susjedstva

| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 0 | 1 | 0 |
| B | 1 | 0 | 1 | 0 | 0 |
| C | 0 | 1 | 0 | 1 | 1 |
| D | 1 | 0 | 1 | 0 | 0 |
| E | 0 | 0 | 1 | 0 | 1 |

Kod težinskih grafova na pozicije u matrice upisujemo težine tih bridova. Kod netežinskih, u slučaju da je vrijednost na poziciji $M_{i,j}$ 1, postoji brid koji povezuje čvorove i i j.

U slučaju da je vrijednost 0, ne postoji takav brid. U tablici 2 je prikazan graf na slici 3 pomoću matrice susjedstva. U ovoj implementaciji, vremenska složenost za metode koja vraća podatak jesu li dva čvora povezana je $O(1)$. No, u isto vrijeme, ova matrica ima prostornu složenost od $O(|V|^2)$. (section.io)

### 3.1.2. Matrica incidencije

Nadalje, kod matrice incidencije, za dva brida kažemo da su incidentni ako sadrže isti čvor. Na primjer, bridove $(A, B)$ i $(B, C)$ smatramo incidentnima jer dijele čvor $B$. Za čvor i brid kažemo da su incidentni ako je napomenuti čvor spojen sa drugim čvorom tim bridom. Na primjer, čvor $(A, B)$ i brid $B$ su incidentni. Na tablici žnj je graf na slici 3 predstavljen pomoću matrice incidencije.

Tablica 2: prikaz grafa sa slike 3 pomoću matrice incidencije

| | (A, B) | (A, D) | (B, C) | (C, D) | (C, E) | (E, E) |
|---|---|---|---|---|---|---|
| A | 1 | 1 | 0 | 0 | 0 | 0 |
| B | 1 | 0 | 1 | 0 | 0 | 0 |
| C | 0 | 0 | 1 | 1 | 1 | 0 |
| D | 0 | 1 | 0 | 1 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 1 | 1 |

U toj matrici redovi predstavljaju čvorove, a stupci bridove. Vrijednost u matici na poziciji $M_{i,j}$ je 0 u slučaju da čvor i i brid j nisu incidentni. U slučaju da jesu, vrijednost je 1. Takva matrica ima prostornu složenost $O(|V| \times |E|)$. Vremenska složenost za izradu takve matrice je $O(|E|)$. Poveznica matrice incidencije $C$ i matrice susjedstva $L$ se može prikazati u jednadžbi $L = C^T \times C - 2I$, gdje je $I$ jedinična matrica.

### 3.1.3. Lista susjedstva

Kod implementacije pomoću liste susjedstva graf predstavljamo kao niz vezanih lista. U toj implementaciji, svaki čvor grafa $A_i$ tvori jednu vezani listu koja sadrži vrijednosti svih čvorova koji se povezuju na čvor i. U tablici žnj možemo vidjeti prikaz grafa sa slike 3 pomoću liste susjedstva.

Tablica 3: prikaz grafa sa slike 3 pomoću liste susjedstva

| | |
|---|---|
| A | B, D |
| B | A, C |
| C | B, D, E |
| D | A, C |
| E | C, E |

Ovakva implementacija ima prostornu složenost $O(|V| + |E|)$. Ona je daleko manja od implementacije pomoću matrice sujedstva ili matrice incidencije. Stoga je i najčešći oblik implementacije. Zbog ovih podataka, izračunat ćemo vremensku složenost operacija za implementaciju grafa pomoću liste susjedstva.

## 3.2. Implementacija grafa pomoću liste susjedstva

Tablica 4: operacije implementirane u ATP-u graf

| Opservatori | |
|---|---|
| `isVertex(vertKey)` | Funkcija vraća **true** ako u grafu postoji čvor sa oznakom `vertKey`. U suprotnome vraća **false**. U slučaju da je oznaka navedenog čvora veća od dopuštene, program vraća pogrešku. |
| `isEdge(fromVert, toVert)` | Funkcija vraća **true** ako u grafu postoji brid između čvorova `fromVert` i `toVert`. U suprotnome vraća **false**. U slučaju da je oznaka navedenog čvora veća od dopuštene, program vraća pogrešku. |
| `printGraph()` | Funkcija koja ispisuje sve čvorove i koji su čvorovi njime povezani. |

| Transformatori | |
|---|---|
| `addEdge(fromVert, toVert)` | Operacija koja grafu dodaje neusmjereni brid. Taj brid povezuje vrhove `fromVert` i `toVert`.U slučaju da je oznaka jednog od vrhova veća od maksimalne, program vraća pogrešku. |
| `deleteEdge(fromVert, toVert)` | Operacija koja briše neusmjereni brid grafa. Taj brid se definira pomoću vrhova `fromVert` i `toVert`. U slučaju da naznačeni brid ne postoji u grafu, program vraća pogrešku. |

Na sljedećim slikama prikazana je implementacija ATP-a graf u programskom jeziku C++ koristeći listu susjedstva.

```cpp
#ifndef MAXV
#define MAXV 1000
#endif

#include <iostream>
#include <cstdlib>
using namespace std;

class Graph
{
    private:

        struct SlogCvor {
            int oznaka;
            struct SlogCvor* sljedeci;
        };


        struct SlogListaSusjd {
            struct SlogCvor *glavaL;
        };


        SlogListaSusjd* polje;

    public:

        Graph() {
            polje = new SlogListaSusjd[MAXV];
            for (int i = 0; i < MAXV; ++i)
                polje[i].glavaL = NULL;
        }

        void addEdge(int izvor, int odrediste) {
            if(izvor > MAXV || odrediste > MAXV){
                cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
                exit(EXIT_FAILURE);
            }else{


                SlogCvor* noviCvor = new SlogCvor;
                noviCvor->oznaka = odrediste;
                noviCvor->sljedeci = NULL;

                noviCvor->sljedeci = polje[izvor].glavaL;
                polje[izvor].glavaL = noviCvor;


                noviCvor = new SlogCvor;
                noviCvor->oznaka = izvor;
```

```
            noviCvor->sljedeci = NULL;

            noviCvor->sljedeci = polje[odrediste].glavaL;
            polje[odrediste].glavaL = noviCvor;
        }
    }

    void printGraph(){

        for (int i = 0; i < MAXV; ++i) {

            if(isVertex(i)){

                SlogCvor* priv = polje[i].glavaL;
                cout<<"\nCvorovi povezani sa cvorom "<<i<<": \n";

                while (priv) {

                    cout<<priv->oznaka;
                    if(priv->sljedeci != NULL){
                        cout<< ", ";
                    }

                    priv = priv->sljedeci;
                }
```

```
                cout<<endl;
            }
        }
    }

    bool isVertex(int vertKey){
        if(vertKey > MAXV){
            cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl;
            exit(EXIT_FAILURE);
        }else{

            return polje[vertKey].glavaL;
        }
    }

    bool isEdge(int fromVert, int toVert){
        if((fromVert > MAXV) || (toVert > MAXV)){
            cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
            exit(EXIT_FAILURE);

        }else{
            SlogCvor* priv = polje[fromVert].glavaL;

            while (priv) {
```

```
101
102                 if(priv->oznaka == toVert){
103                     return true;
104                 }
105
106                 priv = priv->sljedeci;
107             }
108
109             return false;
110         }
111     }
112
113
114     void deleteEdge(int fromVert, int toVert){
115
116         if((fromVert > MAXV) || (toVert > MAXV)){
117             cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
118             exit(EXIT_FAILURE);
119
120         }else{
121
122             if(isVertex(fromVert) && isVertex(toVert)){
123
124
125                 // brisanje zapisa odredista iz liste izvora:
126                 SlogCvor* prethodni =  polje[fromVert].glavaL;
127                 if(prethodni->sljedeci == NULL){
128                     if(prethodni->oznaka == toVert){
129
130                         delete prethodni;
131
132                         polje[fromVert].glavaL = NULL;
133
134                     } else{
135                         cout << "Brid ("<< fromVert << ", " << toVert << ") ne postoji." <<  endl;
136                         exit(EXIT_FAILURE);
137                     }
138
139                 } else {
140
141                     if(prethodni->oznaka == toVert){
142
143                         polje[fromVert].glavaL = prethodni->sljedeci;
144                         delete prethodni;
145
146
147                     } else{
148
149                         SlogCvor* tekuci = (*prethodni).sljedeci;
150
```

```
151                     while (tekuci){
152                         if (tekuci->oznaka == toVert){
153                             prethodni->sljedeci = tekuci->sljedeci;
154                             delete tekuci;
155                             return;
156                         }
157
158                         prethodni = tekuci;
159                         tekuci = tekuci->sljedeci;
160                     }
161                 }
162
163             }
164
165
166             // brisanje zapisa odredista iz liste izvora:
167             SlogCvor* prthodni =  polje[toVert].glavaL;
168             if(prthodni->sljedeci == NULL){
169                 if(prthodni->oznaka == fromVert){
170
171                     delete prthodni;
172
173                     polje[toVert].glavaL = NULL;
174
175                 } else{
176                     cout << "Brid ("<< toVert << ", " << fromVert << ") ne postoji." <<  endl;
177                     exit(EXIT_FAILURE);
178                 }
179
180             } else {
181
182                 if(prthodni->oznaka == fromVert){
183
184                     polje[toVert].glavaL = prthodni->sljedeci;
185                     delete prthodni;
186
187
188                 } else{
189
190                     SlogCvor* tekuci = (*prthodni).sljedeci;
191
192                     while (tekuci){
193                         if (tekuci->oznaka == fromVert){
194                             prthodni->sljedeci = tekuci->sljedeci;
195                             delete tekuci;
196                             return;
197                         }
198
199                         prthodni = tekuci;
200                         tekuci = tekuci->sljedeci;
```

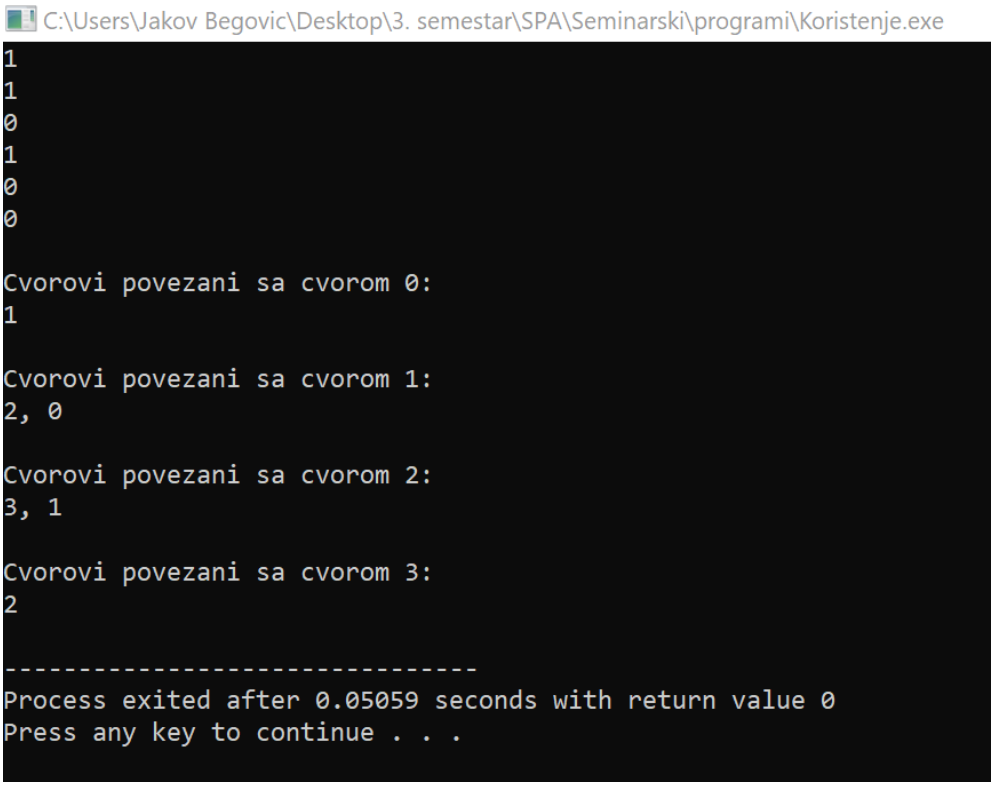
```
201                         }
202                     }
203
204                 }
205
206             }
207
208         }
209     }
210
211
212     ~Graph(){
213         for(int i=0; i<MAXV; i++){
214             SlogCvor* L =  polje[i].glavaL;
215             while(L != NULL){
216
217                 SlogCvor* p = L;
218                 L = (*L).sljedeci;
219                 delete p;
220
221             }
222         }
223     }
224
225
226 };
```

Slike 4-12: implementacija ATP-a graf u programskom jeziku C++ koristeći listu susjedstva

Na sljedećim slikama prikazan je program kojeg sam koristio kako bi testirao gore prikazane operacije.

```cpp
#include <iostream>
#include "Graph.h"
using namespace std;

int main() {

    Graph gh;
    gh.addEdge(0, 1);
    gh.addEdge(1, 2);
    gh.addEdge(0, 3);
    gh.addEdge(2, 3);

    cout << gh.isEdge(0, 3) << endl;
    cout << gh.isEdge(3, 0) << endl;
    cout << gh.isEdge(1, 3) << endl;

    cout << gh.isVertex(0) << endl;
    cout << gh.isVertex(4) << endl;

    gh.deleteEdge(0, 3);
    cout << gh.isEdge(0, 3) << endl;

    gh.printGraph();


    return 0;
}
```

Slika 13: program korišten za testiranje funkcionalnosti implementiranih operacija




Slika 14: rezultati pokretanja programa sa slike 13

## 3.3. Izračun vremenske složenosti operacija

```
#ifndef MAXV
#define MAXV 1000
#endif

#include <iostream>
#include <cstdlib>
using namespace std;

class Graph
{
    private:

        struct SlogCvor {
            int oznaka;
            struct SlogCvor* sljedeci;
        };


        struct SlogListaSusjd {
            struct SlogCvor *glavaL;
        };


        SlogListaSusjd* polje;

    public:

        Graph() {
            polje = new SlogListaSusjd[MAXV];        // c1
            for (int i = 0; i < MAXV; ++i)       // *(n), c2
                polje[i].glavaL = NULL;      // c3
        }

        bool isVertex(int vertKey){
            if(vertKey > MAXV){      // c1
                cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl;
                exit(EXIT_FAILURE);      // 36. i 37. linija b1
            }else{

                return polje[vertKey].glavaL;        // b2
            }
        }

        void addEdge(int izvor, int odrediste) {
            if(izvor > MAXV || odrediste > MAXV){        // c1
                cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
                exit(EXIT_FAILURE);      // od 46. do 47. linije b1
            }else{
```

```
51                  SlogCvor* noviCvor = new SlogCvor;
52                  noviCvor->oznaka = odrediste;
53                  noviCvor->sljedeci = NULL;
54
55                  noviCvor->sljedeci = polje[izvor].glavaL;
56                  polje[izvor].glavaL = noviCvor;
57
58
59                  noviCvor = new SlogCvor;
60                  noviCvor->oznaka = izvor;
61                  noviCvor->sljedeci = NULL;
62
63                  noviCvor->sljedeci = polje[odrediste].glavaL;
64                  polje[odrediste].glavaL = noviCvor;     // od 51. do 64. linije b2
65              }
66          }
67
68          void printGraph(){
69
70              for (int i = 0; i < MAXV; ++i) {        // *(n), <= c1
71
72                  if(isVertex(i)){    // b1
73
74                      SlogCvor* priv = polje[i].glavaL;
75                      cout<<"\nCvorovi povezani sa cvorom "<<i<<": \n";       // od 64. do 65. linije b2
76
77                      while (priv) {      // <= *(n), b3
78
79                          cout<<priv->oznaka;     // a1
80                          if(priv->sljedeci != NULL){     // a2
81                              cout<< ", ";        // a3
82                          }
83
84                          priv = priv->sljedeci;      // a4
85                      }
86
87                      cout<<endl;     // b5
88                  }
89              }
90          }
91
92          bool isEdge(int fromVert, int toVert){
93              if((fromVert > MAXV) || (toVert > MAXV)){       // c1
94                  cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
95                  exit(EXIT_FAILURE);
96
97              }else{
98                  SlogCvor* priv = polje[fromVert].glavaL;        // b1
99
100                 while (priv) {      // <= *(n), b2
```

```
101
102                 if(priv->oznaka == toVert){     // a1
103                     return true;        // a2
104                 }
105
106                 priv = priv->sljedeci;      // a3
107             }
108
109             return false;       // b4
110         }
111     }
112
113
114     void deleteEdge(int fromVert, int toVert){
115
116         if((fromVert > MAXV) || (toVert > MAXV)){       // c1
117             cout << "Oznaka cvora premasuje maksimalnu vrijednost." << endl ;
118             exit(EXIT_FAILURE);
119
120         }else{
121
122             if(isVertex(fromVert) && isVertex(toVert)){     // c2
123
124
125                 // brisanje zapisa odredista iz liste izvora:
126                 SlogCvor* prethodni =  polje[fromVert].glavaL;      // c3
127                 if(prethodni->sljedeci == NULL){        // c4
128                     if(prethodni->oznaka == toVert){
129
130                         delete prethodni;
131
132                         polje[fromVert].glavaL = NULL;
133
134                     } else{
135                         cout << "Brid ("<< fromVert << ", " << toVert << ") ne postoji." <<  endl;
136                         exit(EXIT_FAILURE);
137                     }
138
139                 } else { // 128 - 136 zanemarivo
140
141                     if(prethodni->oznaka == toVert){        // c5
142
143                         polje[fromVert].glavaL = prethodni->sljedeci;
144                         delete prethodni;
145
146
147                     }else{ // 143-144 zanemarivo
148
149                         SlogCvor* tekuci = (*prethodni).sljedeci;       // c6
150
151                         while (tekuci){     // <= *(n), c7
152                             if (tekuci->oznaka == toVert){      // b1
```

```
153                                 prethodni->sljedeci = tekuci->sljedeci;     // b2
154                                 delete tekuci;      // b3
155                                 return;       // b4
156                             }
157
158                             prethodni = tekuci;     // c8
159                             tekuci = tekuci->sljedeci;      // c9
160                         }
161                     }
162
163                 }
164
165
166                 // brisanje zapisa odredista iz liste izvora:
167                 SlogCvor* prthodni =  polje[toVert].glavaL;     // c10
168                 if(prthodni->sljedeci == NULL){     // c11
169                     if(prthodni->oznaka == fromVert){
170
171                         delete prthodni;
172
173                         polje[toVert].glavaL = NULL;
174
175                     } else{
176                         cout << "Brid ("<< toVert << ", " << fromVert << ") ne postoji." <<  endl;
177                         exit(EXIT_FAILURE);
178                     }
```

```
179
180                  } else { // 169-177 zanemarivo
181
182                      if(prthodni->oznaka == fromVert){       // c12
183
184                          polje[toVert].glavaL = prthodni->sljedeci;
185                          delete prthodni;
186
187
188                      } else{     // 185-184 zanemarivo
189
190                          SlogCvor* tekuci = (*prthodni).sljedeci;        // c13
191
192                          while (tekuci){     //  <= *(n), c14
193                              if (tekuci->oznaka == fromVert){        // b5
194                                  prthodni->sljedeci = tekuci->sljedeci;      // b6
195                                  delete tekuci;      // b7
196                                  return;     // b8
197                              }
198
199                              prthodni = tekuci;      // c15
200                              tekuci = tekuci->sljedeci;      // c16
201                          }
202                      }
203
204                  }
```

```
205
206                 }
207
208             }
209         }
210
211
212         ~Graph(){
213             for(int i=0; i<MAXV; i++){      // *(n), c1
214                 SlogCvor* L =  polje[i].glavaL;     // c2
215                 while(L != NULL){       // <= *(n), c3
216
217                     SlogCvor* p = L;        // b1
218                     L = (*L).sljedeci;      // b2
219                     delete p;       // b3
220
221                 }
222             }
223         }
224
225
226 };
227
228
```

Slike 15-23: izračun vremenske složenosti za pojedine linije koda u datoteci zaglavlja

### 3.3.1. Složenost za operaciju Graph:

$$T_{max}^{Graph}(n) = c_1 + (c_2 + c_3) \times n$$
$$= d_1 + d_2 \times n$$

Pa se može reći da je

$$T_{max}^{Graph}(n) = O(n)$$

### 3.3.2. Složenost za operaciju isVertex:

Izvest će se ili blok iz linija 36 – 37 ili linija 40. Neka je $c_2 = \max\{b_1, b_2\}$. Tada je složenost

$$T_{max}^{\text{isVertex}}(n) \leq c_1 + c_2$$
$$T_{max}^{\text{isVertex}}(n) = d_1$$

Pa se može reći da je

$$T_{max}^{\text{isVertex}}(n) = O(1)$$

### 3.3.3. Složenost za operaciju addEdge:

Izvest će se ili blok iz linija 46 - 47 ili blok linija 51 - 64. Neka je $c_2 = \max\{b_1, b_2\}$. Tada je složenost

$$T_{max}^{\text{addEdge}}(n) \leq c_1 + c_2$$
$$T_{max}^{\text{addEdge}}(n) = c_1 + c_2$$

Pa se može reći da je

$$T_{max}^{\text{addEdge}}(n) = O(1)$$

### 3.3.4. Složenost za operaciju printGraph:

While petlja će za najgori slučaj napraviti n ponavljanja. Svaki čvor kod netežinskog grafa može imati maksimalno n povezanih čvorova. Odnosno, svaki čvor može biti najviše jednom povezan sa svakim drugim čvorom grafa, kao i sa samim sobom. Također, polje čvorova sadrži n elemenata.

Za $c_2 \leq b_1 + b_2,\ \ b_4 \leq a_1 + a_2 + a_3 + a_4$ vrijedi

$$T_{max}^{\text{printGraph}}(n) \leq (c_1 + b_1 + b_2 + (b_3 + b_4) \times (n) + b_5) \times n$$
$$T_{max}^{\text{printGraph}}(n) = (c_1 + c_2 + (c_3) \times (n) + b_5) \times n$$
$$T_{max}^{\text{printGraph}}(n) = (d_1 + (c_3) \times (n)) \times n$$
$$T_{max}^{\text{printGraph}}(n) = n \times d_1 + c_3 \times n^2 - c_3 \times n$$

Pa se može reći da je

$$T_{max}^{\text{printGraph}}(n) = O(n^2)$$

### 3.3.5. Složenost za operaciju isEdge:

While petlja će napraviti n ponavljanja za najgori slučaj. Za $b_3 \leq a_1 + a_2 + a_3$ vrijedi

$$T_{max}^{\text{isEdge}}(n) \leq c_1 + b_1 + (b_2 + b_3) \times (n) + b_4$$

$$T_{max}^{\text{isEdge}}(n) = c_1 + c_2 + c_3 \times n$$

Pa se može reći da je

$$T_{max}^{\text{isEdge}}(n) = O(n)$$

### 3.3.6. Složenost za operaciju deleteEdge:

While petlja će napraviti n ponavljanja za najgori slučaj. Vrijedi

$$T_{max}^{\text{deleteEdge}}(n) \leq c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + (c_7 + b_1 + b_2 + b_3 + b_4) \times (n) + c_8 + c_9 + c_{10}$$
$$+ c_{11} + c_{12} + c_{13} + (c_{14} + b_5 + b_6 + b_7 + b_8) \times (n) + c_{15} + c_{16}$$
$$T_{max}^{\text{deleteEdge}}(n) = d_1 + (d_2) \times (n) + (d_3) \times (n)$$
$$T_{max}^{\text{deleteEdge}}(n) = d_1 + d_2 \times n + d_3 \times n$$

Pa se može reći da je

$$T_{max}^{\text{deleteEdge}}(n) = O(n)$$

### 3.3.7. Složenost za operaciju ~Graph:

$$T_{max}^{\sim Graph}(n) \leq (c_1 + c_2 + (c_3 + b_1 + b_2 + b_3) \times n) \times n$$
$$= (d_1 + d_2 \times n) \times n$$
$$= d_1 \times n + d_2 \times n^2$$

Pa se može reći da je

$$T_{max}^{\sim Graph}(n) = O(n^2)$$

### 3.3.8. Složenost operacija:

Tablica 5: složenost operacija ATP-a graf

| | |
|---|---|
| `Graph` | O(n) |
| `isVertex` | O(1) |
| `isEdge` | O(n) |
| `printGraph` | $O(n^2)$ |
| `addEdge` | O(1) |
| `deleteEdge` | O(n) |
| `~Graph` | $O(n^2)$ |

## 3.4. Izračun amortizirane složenosti strukture podataka

Kako bih izračunao amortiziranu složenost strukture podataka koristit ću metodu energetskih potencijala zbog njezine točnosti. Prvo moram izračunati stvarnu cijenu operacija.

Tablica 6: stvarna cijena operacija

| | |
|---|---|
| `isVertex` | 1 |
| `isEdge` | i |
| `printGraph` | $i^2$ |
| `addEdge` | 1 |
| `deleteEdge` | i |

Nadalje, postavljamo da je $\Phi(D_0) = 0$ i izračunajmo amortiziranu cijenu operacija ove strukture.

### 3.4.1. Amortizirana cijena operacije isVertex

Ova operacija ne mijenja strukturu podataka, pa time ni ne mijenja njezin potencijal. Dakle, nakon njezina izvršenja bit će $\Phi(D_i) = \Phi(D_i - 1)$.

$$\hat{c}_i(isVertex) = c_i(isVertex) + \Phi(D_i) - \Phi(D_i - 1)$$
$$= c_i(isVertex)$$
$$= 1.$$

### 3.4.2. Amortizirana cijena operacije isEdge

Ova operacija ne mijenja strukturu podataka, pa time ne mijenja ni njezin potencijal. Dakle, nakon njezina izvršenja bit će $\Phi(D_i) = \Phi(D_i - 1)$.

$$\hat{c}_i(isEdge) = c_i(isEdge) + \Phi(D_i) - \Phi(D_i - 1)$$
$$= c_i(isEdge)$$
$$= i.$$

### 3.4.3. Amortizirana cijena operacije printGraph

Ova operacija ne mijenja strukturu podataka, pa time ni ne mijenja njezin potencijal. Dakle, nakon njezina izvršenja bit će $\Phi(D_i) = \Phi(D_i - 1)$.

$$\hat{c}_i(printGraph) = c_i(printGraph) + \Phi(D_i) - \Phi(D_i - 1)$$
$$= c_i(printGraph)$$
$$= i^2.$$

### 3.4.4. Amortizirana cijena operacije addEdge

Ova operacija povećava potencijal strukture podataka za 1. Dakle, nakon njezina izvršenja bit će $\Phi(D_i) = \Phi(D_i - 1) + 1$.

$$\begin{aligned}\hat{c}_i(addEdge) &= c_i(addEdge) + \Phi(D_i) - \Phi(D_i - 1) \\ &= c_i(addEdge) + \Phi(D_i - 1) + 1 - \Phi(D_i - 1) \\ &= 1 + 1 = 2.\end{aligned}$$

### 3.4.5. Amortizirana cijena operacije deleteEdge

Ova operacija umanjuje potencijal strukture podataka za 1. Dakle, nakon njezina izvršenja bit će $\Phi(D_i) = \Phi(D_i - 1) - 1$.

$$\begin{aligned}\hat{c}_i(deleteEdge) &= c_i(deleteEdge) + \Phi(D_i) - \Phi(D_i - 1) \\ &= c_i(deleteEdge) + \Phi(D_i - 1) - 1 - \Phi(D_i - 1) \\ &= i - 1.\end{aligned}$$

### 3.4.6. Amortizirane cijene operacija

Tablica 7: amortizirane cijene operacija

| | |
|---|---|
| `isVertex` | 1 |
| `isEdge` | i |
| `printGraph` | $i^2$ |
| `addEdge` | 2 |
| `deleteEdge` | i - 1 |

### 3.4.7. Amortizirana složenost strukture podataka

Najveću amortiziranu cijenu ima operacija printGraph, pa će najgori slučaj N uzastopnih operacija biti kada se radi o N uzastopnih operacija printGraph. Stoga je

$$\begin{aligned}\sum_{i=1}^{N} \hat{c}_i(O_i) &\leq \sum_{i=1}^{N} \hat{c}_i(printGraph) \\ &= \sum_{i=1}^{N} i^2 \\ &= \frac{N(N+1)(2N+1)}{6}\end{aligned}$$

Možemo zaključiti da je amortizirana složenost ove strukture podataka $T^{printGraph}(n) = O(n^3)$.

# 4. Zaključak

Struktura podataka graf korisna je pri modeliranju razno raznih veza između objekata. Te veze mogu biti međuljudske veze, kao i veze između gradova ili aerodroma. Svim tim vezama možemo pripisati razne karakteristike. Možemo opisati njihovo usmjerenje i težinu. Možemo okarakterizirati grafove prema njihovim karakteristikama. I pomoću jednog grafa možemo nalaziti optimalne puteve kroz njegove bridove. Možemo nalaziti puteve koji optimiziraju broj čvorova kroz koje se prolazi. Ili koji optimiziraju cijenu prijevoza. Ili vrijeme vožnje i slično.

Postoji tri različite implementacije te strukture podataka. U ovom seminarskom radu obradio sam implementaciju pomoću liste susjedstva. To je lista svih čvorova u grafu. Svaki čvor predstavlja jednu vezanu listu. Elementi vezane liste su čvorovi povezani s njime. Ova implementacija je široko korištena zbog relativno malog zauzeća prostornih, kao i vremenskih resursa. Matrica susjedstva ima prostornu složenost $O(|V|^2)$. Matrica incidencije ima prostornu složenost od $O(|V|+|E|)$, kao i lista susjedstva. Vremenska složenost inicijalizacije matrice incidencije iznosi $O(|E|)$, dok je to kod liste susjedstva $O(|V|)$.

U svojoj implementaciji tipa podatka graf razvio sam pet različitih operacija. Jedna koja provjerava postoji li čvor indeksa vertKey u grafu. To je operacija isVertex(vertKey). Nadalje, razvio sam operaciju koja provjerava jesu li dva čvora spojena bridom. To je operacija isEdge(fromVert, toVert). Operacija printGraph ispiše sve čvorove i čvorove koji su sa njima povezani. Operacija addEdge(fromVert, toVert) spaja čvorove fromVert i toVert, gdje ove varijable predstavljaju njihove indekse u listi susjedstva. Vrijednost čvora toVert dodaje u vezanu listu čvora fromVert. Također dodaje zapis čvora fromVert u vezanu listu čvora toVert. Zadnja operacija koju sam razvio je deleteEdge(fromVert, toVert). Ona iz vezane liste čvora fromVert briše zapis čvora toVert. Također, iz vezane liste čvora toVert briše zapis čvora fromVert.

U mojoj implementaciji liste susjedstva vremenski najkompleksnija operacija je printGraph sa vremenskom kompleksnosti od $O(n^2)$. Korisnik ne može sam razviti tu operaciju jer je lista susjedstva privatno inicijalizirana unutar datoteke strukture podataka. Isto tako, operacija printGraph vremenski je najkompleksnija dok se radi o amortiziranoj vremenskoj složenosti sa složenosti od $i^2$. Nakon nje, to je operacija deleteEdge. Ona ima vremensku složenost od $O(n)$, a amortiziranu vremensku složenost od $i - 1$.

# 5. Popis literature


[1] KUSALIĆ, D. (2011). Napredno programiranje i algoritmi u C-u i C++-u. Element.

[2] *Adjacency List (With Code in C, C++, Java and Python)*. (bez dat.). Preuzeto 25. siječanj 2023., od https://www.programiz.com/dsa/graph-adjacency-list

[3] *Graph Implementation in C++ Using Adjacency List*. (2022, prosinac 5). Software Testing Help. https://www.softwaretestinghelp.com/graph-implementation-cpp/

[4] *Graphs in Data structure (using C++)*. (bez dat.). Engineering Education (EngEd) Program | Section. Preuzeto 02. siječanj 2023., od https://www.section.io/engineering-education/graphs-in-data-structure-using-cplusplus/

[5] *Sum of n, $n^2$, or $n^3$ | Brilliant Math & Science Wiki*. (bez dat.). Preuzeto 25. siječanj 2023., od https://brilliant.org/wiki/sum-of-n-n2-or-n3/

[6] Wu, G. (2022, siječanj 11). *Graph Adjacency and Incidence | Baeldung on Computer Science*. https://www.baeldung.com/cs/graph-adjacency-and-incidence