Iskanje v globino in sorodni algoritmi

Luka Fürst

ponedeljek, 27. marca 2023

Vhod

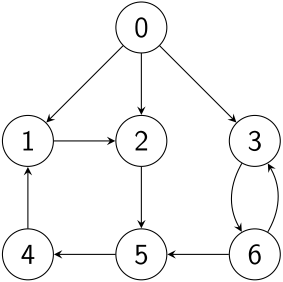
- enostaven usmerjen ali neusmerjen graf
- vozlišča $0, 1, \ldots, n-1$
- graf je predstavljen s seznamom sosednosti
    - `vector<vector<int>> graf;`
    - `graf[`u`]` = vektor sosedov vozlišča u

Iskanje v globino (DFS)

- algoritem, ki obišče vsa vozlišča in povezave
- obiščemo vozlišče, nato pa rekurzivno obiščemo vse njegove sosede
- da vsako vozlišče obiščemo natanko enkrat, vzdržujemo tabelo že obiskanih vozlišč
- $O(V + E)$

```cpp
void dfs(const vector<vector<int>>& graf, int vozlisce,
         vector<bool>& obiskano) {

    cout << vozlisce << endl;
    obiskano[vozlisce] = true;
    for (int sosed: graf[vozlisce]) {
        if (!obiskano[sosed]) {
            dfs(graf, sosed, obiskano);
        }
    }
}
```

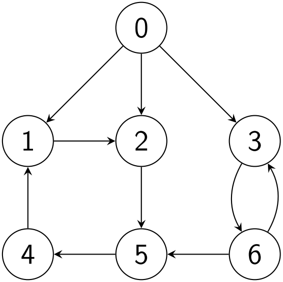
Iskanje v globino

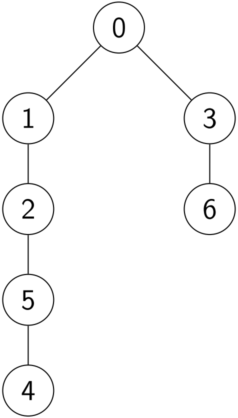



```
dfs(0)
    dfs(1)
        dfs(2)
            dfs(5)
                dfs(4)
    dfs(3)
        dfs(6)
```

DFS-drevo

- vozlišča DFS-drevesa so vozlišča grafa
- obstaja povezava (u, v) $\iff$ vozlišče v odkrijemo kot soseda vozlišča u





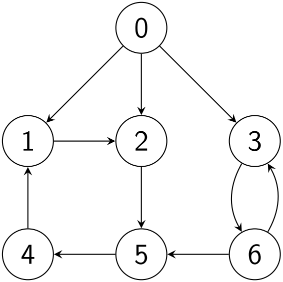
Stanja vozlišč in vrste povezav

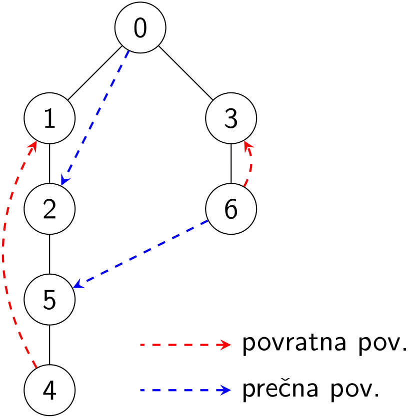
- Stanje vozlišča med iskanjem
  - neodkrito, če ga še nismo odkrili
  - obiskano, če smo ga odkrili, vendar pa še nismo obdelali vseh njegovih sosedov
  - obdelano, če smo ga obiskali in če smo obdelali vse njegove sosede
- Vrsta povezave
  - drevesna: med obiskanim in pravkar odkritim vozliščem
  - povratna: med obiskanim in obiskanim vozliščem
  - prečna: med obiskanim in obdelanim vozliščem (možna samo pri usmerjenih grafih)

Tipi vozlišč in povezav

```cpp
void dfs(const vector<vector<int>>& graf, int vozlisce,
         vector<int>& stanje, vector<int>& stars) {
    // stars[u]: starš vozlišča u v DFS-drevesu

    stanje[vozlisce] = OBISKANO;

    for (int sosed: graf[vozlisce]) {
        if (stanje[sosed] == OBISKANO) {
            // povratna povezava vozlisce-sosed
        } else if (stanje[sosed] == OBDELANO) {
            // prečna povezava vozlisce-sosed
        } else {
            // drevesna povezava vozlisce-sosed
            stars[sosed] = vozlisce;
            dfs(graf, sosed, stanje, stars);
        }
    }
    stanje[vozlisce] = OBDELANO;
}
```

Tipi vozlišč in povezav





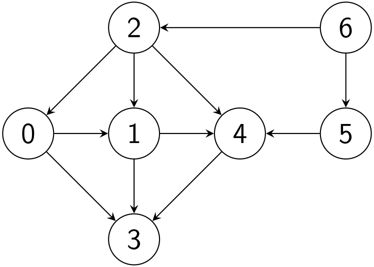
Preverjanje (a)cikličnosti grafa

- cikel: pot $v_1 \to v_2 \to \ldots \to v_k \to v_1$, kjer je $k \geq 3$
- povratna povezava $v \to u$ določa cikel, če u ni starš vozlišča v v DFS-drevesu

```
void dfs(...) {
    ...
    for (int sosed: graf[vozlisce]) {
        if (stanje[sosed] == OBISKANO) {
            // povratna povezava vozlisce-sosed
            if (stars[vozlisce] == sosed) {
                // dvosmerna povezava vozlisce-sosed
            } else {
                // cikel
            }
        }
        ...
    }
    ...
}
```

Topološko urejanje

- samo za usmerjene aciklične grafe (DAG)
- če obstaja pot $u \rightsquigarrow v$, mora biti u v topološkem vrstnem redu pred v



Eden od topoloških vrstnih redov: 6, 5, 2, 0, 1, 4, 3

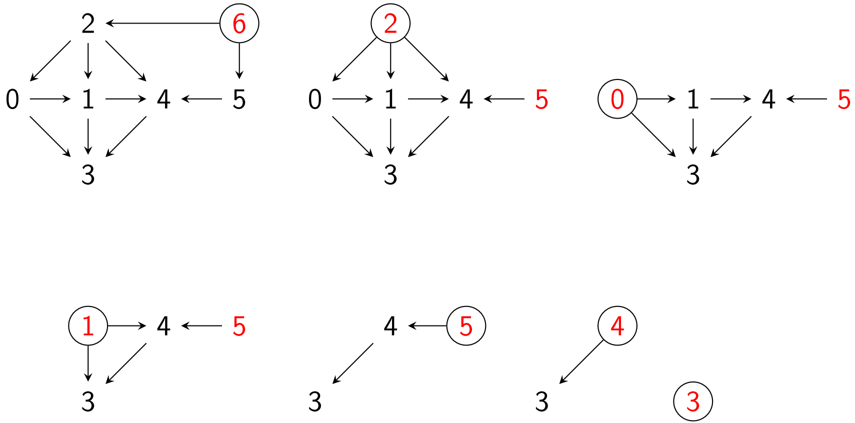
Topološko urejanje — Kahnov algoritem

- vstopna stopnja vozlišča = število povezav v vozlišče
- ponavljaj, dokler ne zmanjka vozlišč:
  - poišči vozlišče v z vstopno stopnjo 0
  - izpiši in odstrani vozlišče v
- vozlišča z vstopno stopnjo 0 lahko hranimo v (prioritetni) vrsti
  - na ta način bomo enakovredna vozlišča obravnavali po, recimo, naraščajočem indeksu

Topološko urejanje — Kahnov algoritem

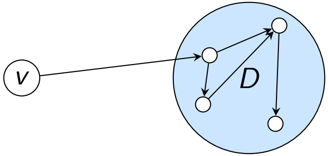
```
priority_queue<int, vector<int>, greater<int>> vrsta;
for (int v = 0; v < stVozlisc; v++) {
    if (stVstopnih[v] == 0) {
        vrsta.push(v);
    }
}

while (!vrsta.empty()) {
    int vozlisce = vrsta.top();
    cout << vozlisce << endl;
    vrsta.pop();
    for (int sosed: graf[vozlisce]) {
        if (--stVstopnih[sosed] == 0) {
            vrsta.push(sosed);
        }
    }
}
```

Topološko urejanje — Kahnov algoritem



Topološko urejanje — Tarjanov algoritem

- naj bo D množica vozlišč, dosegljivih iz vozlišča v



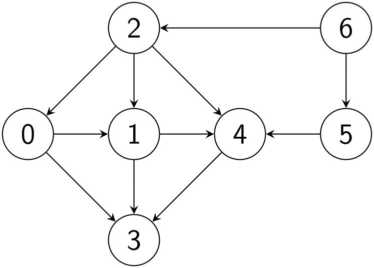
- če poženemo DFS iz vozlišča v, bomo celotno množico D obdelali pred vozliščem v
- vsa vozlišča v D sodijo po topološkem vrstnem redu za vozlišče v

Topološko urejanje — Tarjanov algoritem

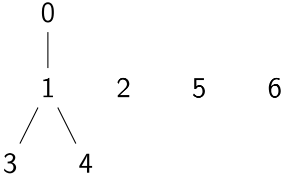
- ko vozlišče obdelamo, ga shranimo v vektor
- vektor na koncu samo obrnemo

```
void dfs(const vector<vector<int>>& graf, int vozlisce,
         vector<bool>& obiskano, vector<int>& vrstniRed) {

    obiskano[vozlisce] = true;
    for (int sosed: graf[vozlisce]) {
        if (!obiskano[sosed]) dfs(graf, sosed, obiskano, vrstniRed);
    }
    vrstniRed.push_back(vozlisce);
}

// main:
vector<int> vrstniRed;
for (int v = 0; v < stVozlisc; v++) {
    if (!obiskano[v]) dfs(graf, v, obiskano, vrstniRed);
}
reverse(vrstniRed.begin(), vrstniRed.end());
```

Topološko urejanje — Tarjanov algoritem



DFS-drevesa iz vozl. 0, 2, 5 in 6:

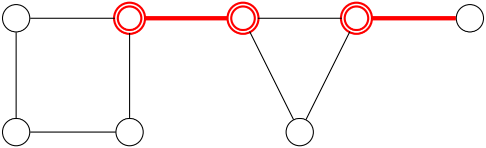


- obratni topološki vrstni red: 3, 4, 1, 0, 2, 5, 6
- topološki vrstni red: 6, 5, 2, 0, 1, 4, 3

Mostovi in prerezne točke

- podan je povezan neusmerjen graf
- če graf po odstranitvi povezave (u, v) razpade na dva dela, potem je povezava (u, v) most
- če graf po odstranitvi vozlišča u razpade na dva dela, potem je vozlišče u prerezna točka

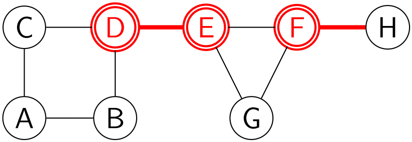
Mostovi in prerezne točke



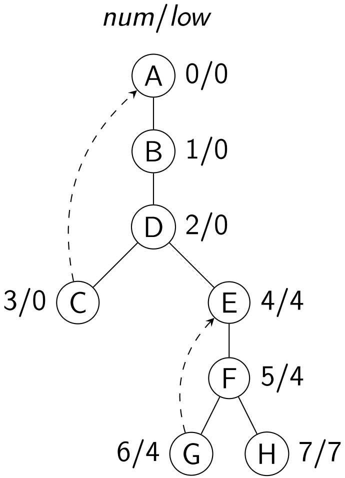
Mostovi in prerezne točke

- med izvajanjem DFS vzdržujemo še vektorja *num* in *low*
- *num*[u]: zaporedna številka vozlišča u v izvajanju DFS
- *low*[u]: najmanjši *num* v množici $M(u)$, ki jo sestavljajo:
  - vozlišča poddrevesa vozlišča u v DFS-drevesu
  - vozlišča, ki so iz poddrevesa vozlišča u dosegljiva prek največ ene povratne povezave

Mostovi in prerezne točke

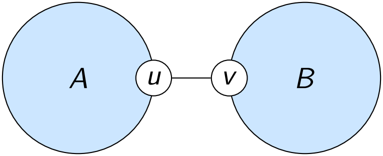


- $M(\mathrm{D}) = \{\mathrm{D}, \mathrm{C}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{A}\}$
- $M(\mathrm{F}) = \{\mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{E}\}$
- $M(\mathrm{G}) = \{\mathrm{G}, \mathrm{E}\}$
- $M(\mathrm{H}) = \{\mathrm{H}\}$



Most

- naj bo u–v most

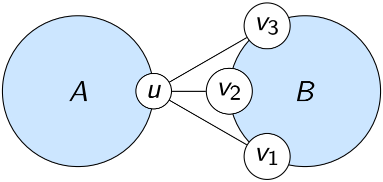


- recimo, da pričnemo v podgrafu A
- ko prečkamo most u–v, se v A brez sestopanja ne moremo več vrniti
- zato bo $low[v] = num[v]$
- ker je $num[v] > num[u]$, bo

$$\boxed{num[u] < low[v]}$$

Prerezna točka

- naj bo u prerezna točka



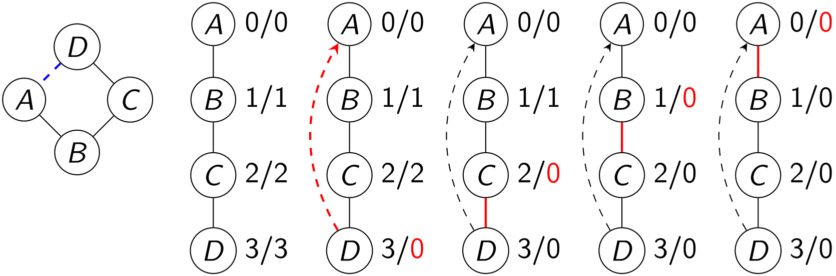
- recimo, da pričnemo v podgrafu A
- recimo, da v podgraf B vstopimo prek povezave u–v_1
- od B lahko brez sestopanja pridemo le še do u (prek v_2–u ali v_3–u), naprej v notranjost A pa ne moremo
- zato bo

$$\boxed{num[u] \leq low[v]}$$

za vse $v \in \{v_1, v_2, v_3\}$

Mostovi in prerezne točke

- ko obiščemo vozlišče u, nastavimo $low[u] := num[u]$
- če s povezavo u–v sklenemo cikel, nastavimo $low[u] := \min(low[u], num[v])$
- ko sestopamo, preverimo, ali je u prerezna točka in ali je u–v most, nato pa posodobimo $low[u] := \min(low[u], low[v])$



Mostovi in prerezne točke

```
void mostovi(int vozlisce, ...) {
    low[vozlisce] = num[vozlisce] = stevec++;

    for (int sosed: graf[vozlisce]) {
        if (num[sosed] == -1) {  // sosed neobiskan
            stars[sosed] = vozlisce;
            mostovi(sosed, ...);
            if (num[vozlisce] <= low[sosed]) {
                // vozlisce je prerezna točka
            }
            if (num[vozlisce] < low[sosed]) {
                // vozlisce-sosed je most
            }
            low[vozlisce] = min(low[vozlisce], low[sosed]);
        } else if (sosed != stars[vozlisce]) {  // cikel
            low[vozlisce] = min(low[vozlisce], num[sosed]);
        }
    }
}
```

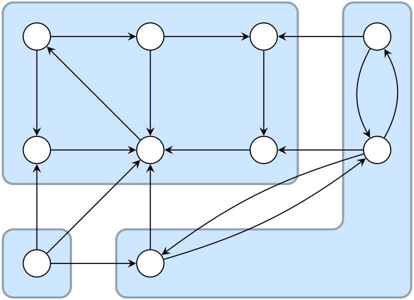
Mostovi in prerezne točke

- Izjema: vozlišče, v katerem sprožimo DFS, je prerezna točka natanko v primeru, če ima v DFS-drevesu vsaj dva otroka

```
void mostovi(int vozlisce, ...) {
    low[vozlisce] = num[vozlisce] = stevec++;

    for (int sosed: graf[vozlisce]) {
        if (num[sosed] == -1) {
            stars[sosed] = vozlisce;
            if (vozlisce == koren) stOtrokKorena++;

            mostovi(sosed, ...);
            if (low[sosed] >= num[vozlisce]) { ... }
            if (low[sosed] > num[vozlisce]) { ... }
            low[vozlisce] = min(low[vozlisce], low[sosed]);
        } else if (sosed != stars[vozlisce]) {
            low[vozlisce] = min(low[vozlisce], num[sosed]);
        }
    }
}
```

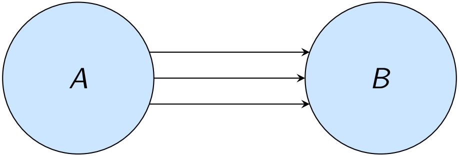
Krepko povezane komponente

- Množica vozlišč $K = \{v_1, \ldots, v_k\} \subseteq V(G)$ tvori krepko povezano komponento usmerjenega grafa G natanko tedaj, ko
  - obstaja pot $v_i \to v_j$ za vsak par $i, j \in \{1, \ldots, k\}$
  - nobeno vozlišče iz $V \setminus K$ ni dvosmerno dosegljivo iz K



Krepko povezane komponente — Kosarajujev algoritem

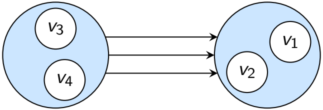
- naj bosta A in B krepko povezani komponenti
  - naj gredo povezave kvečjemu v smeri $A \to B$



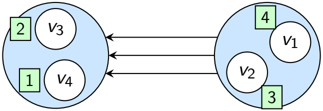
- če poženemo DFS iz komponente A, bodo vsa vozlišča v B obdelana pred katerimkoli vozliščem iz A
- če poženemo DFS iz komponente B, se zgodi enako
  - za komponento A potrebujemo dodaten klic DFS

Krepko povezane komponente — Kosarajujev algoritem

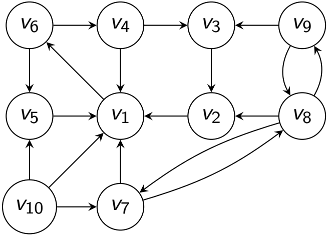
- poganjamo DFS, dokler ne obdelamo vseh vozlišč
- naj bo $v_1, v_2, \ldots, v_n$ vrstni red, v katerem so vozlišča obdelana (najprej v_1, nato v_2, ...)

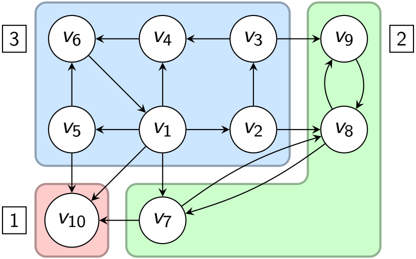


- obrnemo vse povezave grafa in po vrsti poženemo DFS na vozliščih $v_n, v_{n-1}, \ldots, v_1$



Krepko povezane komponente — Kosarajujev algoritem





- najprej poženemo DFS iz vozlišča v_{10}
  - komponenta $= \{v_{10}\}$
- nato poženemo DFS iz vozlišča v_9
  - komponenta $= \{v_9, v_8, v_7\}$
  - do vozlišča v_{10} ne pridemo, ker smo ga že obiskali
- vozlišče v_8 smo že obiskali
- ...

Krepko povezane komponente — Kosarajujev algoritem

```
void dfs(const vector<vector<int>>& graf, int v,
         vector<bool>& obiskano, vector<int>& vrstniRed) {

    obiskano[v] = true;
    for (int sosed: graf[v]) {
        if (!obiskano[sosed]) {
            dfs(graf, sosed, obiskano, vrstniRed);
        }
    }
    vrstniRed.push_back(v);
}
```

Krepko povezane komponente — Kosarajujev algoritem

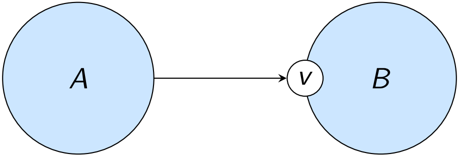
```
int main() {
    ...
    vector<bool> obiskano(stVozlisc);
    vector<int> vrstniRed;
    for (int v = 0; v < stVozlisc; v++) {
        if (!obiskano[v]) dfs(graf, v, obiskano, vrstniRed);
    }

    obiskano.assign(obiskano.size(), false);
    for (int i = stVozlisc - 1; i >= 0; i--) {
        int vozlisce = vrstniRed[i];
        if (!obiskano[vozlisce]) {
            vector<int> komponenta;
            dfs(obrGraf, vozlisce, obiskano, komponenta);
            // <komponenta> je krepko povezana komponenta
        }
    }
    ...
}
```

Krepko povezane komponente — Tarjanov algoritem

- podobna ideja kot pri iskanju mostov in prereznih točk
- $\mathit{num}[u]$
  - zaporedna številka vozlišča u v izvajanju DFS
- $\mathit{low}[u]$
  - ko vozlišče u obiščemo, nastavimo
    $\mathit{low}[u] := \mathit{num}[u]$
  - ko obdelamo soseda v vozlišča u ali ko odkrijemo povezavo od u do že obiskanega vozlišča v, nastavimo
    $\mathit{low}[u] := \min(\mathit{low}[u], \mathit{low}[v])$,
    če v pripada isti komponenti kot u (tj. če je $\mathit{delKomponente}[v]$ enako `true`)

Krepko povezane komponente — Tarjanov algoritem

- naj bosta A in B krepko povezani komponenti
- recimo, da pričnemo v A, v B pa vstopimo pri vozlišču v



- ko obdelamo v, velja $low[v] = num[v]$
  - za ostala vozlišča v B velja $low[v] < num[v]$
- podobno velja za komponento A

Krepko povezane komponente — Tarjanov algoritem

- ko vozlišče u obiščemo
  - ga postavimo na sklad
  - nastavimo $delKomponente[u]$ na `true`
- ko ga obdelamo, preverimo, ali velja $num[u] = low[u]$
- če to velja, pobiramo vozlišča s sklada, dokler ne pridemo do vozlišča u
  - za vsako pobrano vozlišče nastavimo *delKomponente* na `false`
- pobrana vozlišča tvorijo krepko povezano komponento

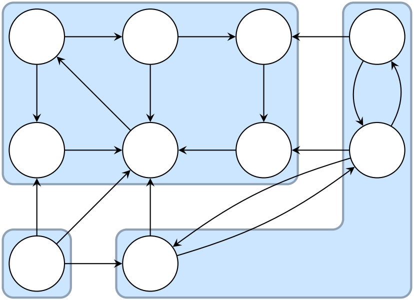
Krepko povezane komponente — Tarjanov algoritem

```
void povezaneKomponente(int vozlisce, ...) {
    low[vozlisce] = num[vozlisce] = stevec++;
    sklad.push(vozlisce);
    delKomponente[vozlisce] = true;
    for (int sosed: graf[vozlisce]) {
        if (num[sosed] == -1) povezaneKomponente(sosed, ...);
        if (delKomponente[sosed]) {
            low[vozlisce] = min(low[vozlisce], low[sosed]);
        }
    }
    if (num[vozlisce] == low[vozlisce]) {   // nova komponenta
        int v = 0;
        do {
            v = sklad.top();  // <v> pripada novi komponenti
            sklad.pop();
            delKomponente[v] = false;
        } while (v != vozlisce);
    }
}
```

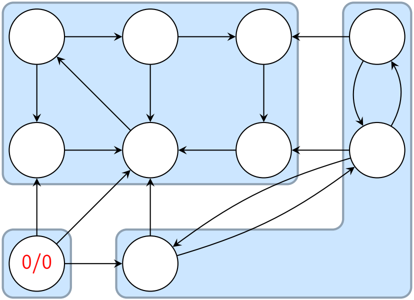
Krepko povezane komponente — Tarjanov algoritem

```
int main() {
    ...
    vector<int> num(stVozlisc, -1);
    for (int v = 0; v < stVozlisc; v++) {
        if (num[v] < 0) {
            povezaneKomponente(v, ...);
        }
    }
    ...
}
```

Krepko povezane komponente — Tarjanov algoritem

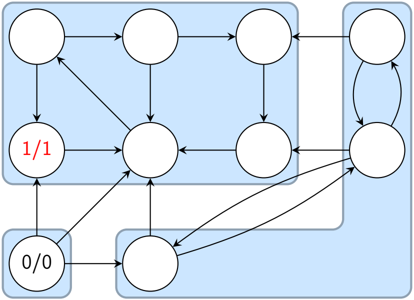


Krepko povezane komponente — Tarjanov algoritem



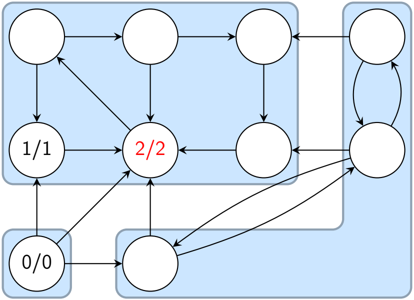
pričnemo v vozlišču 0
sklad: 0

Krepko povezane komponente — Tarjanov algoritem



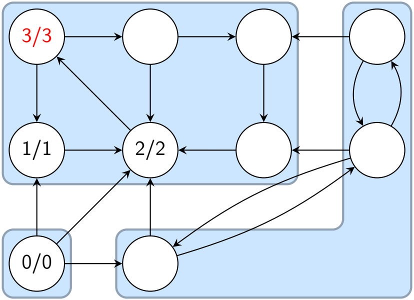
obiščemo vozlišče 1
sklad: 0, 1

Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 2
sklad: 0, 1, 2

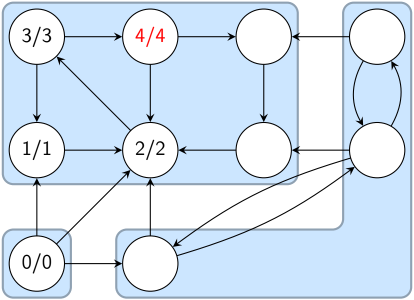
Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 3

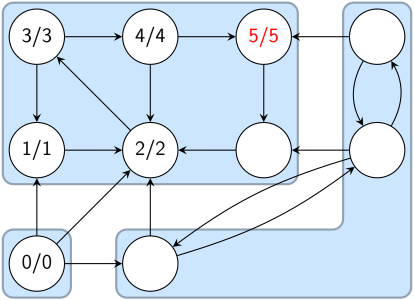
sklad: 0, 1, 2, 3

Krepko povezane komponente — Tarjanov algoritem



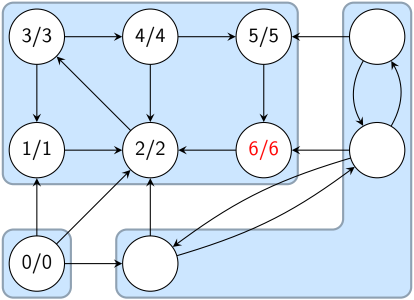
obiščemo vozlišče 4
sklad: 0, 1, 2, 3, 4

Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 5
sklad: 0, 1, 2, 3, 4, 5

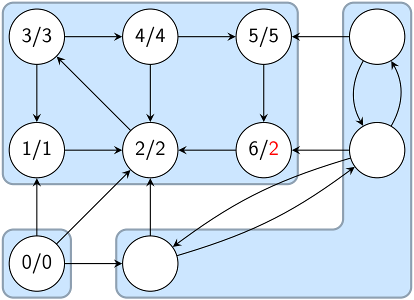
Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 6

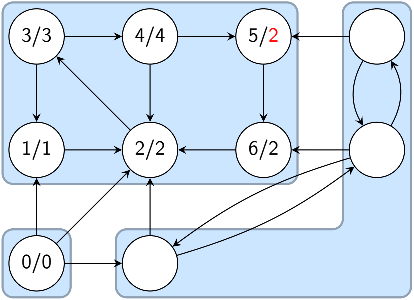
sklad: 0, 1, 2, 3, 4, 5, 6

Krepko povezane komponente — Tarjanov algoritem



vozlišče 6 je obdelano
sklad: 0, 1, 2, 3, 4, 5, 6

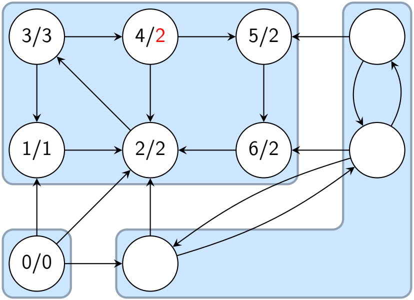
Krepko povezane komponente — Tarjanov algoritem



vozlišče 5 je obdelano

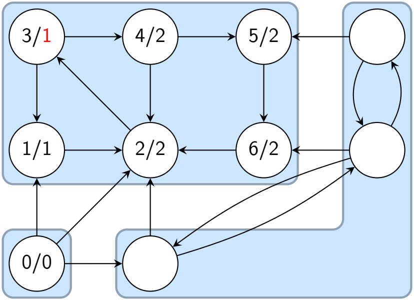
sklad: 0, 1, 2, 3, 4, 5, 6

Krepko povezane komponente — Tarjanov algoritem



vozlišče 4 je obdelano
sklad: 0, 1, 2, 3, 4, 5, 6

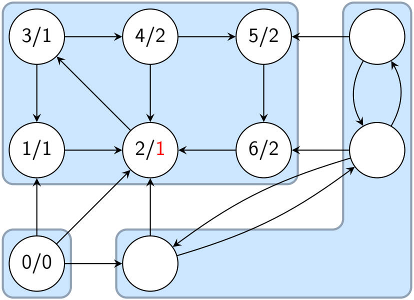
Krepko povezane komponente — Tarjanov algoritem



vozlišče 3 je obdelano

sklad: 0, 1, 2, 3, 4, 5, 6

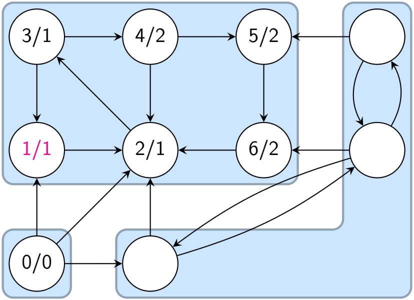
Krepko povezane komponente — Tarjanov algoritem



vozlišče 2 je obdelano

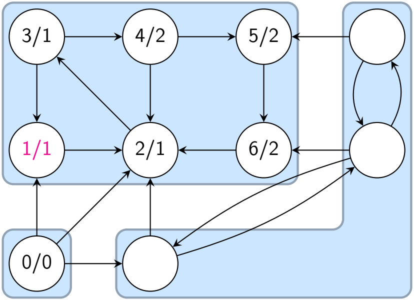
sklad: 0, 1, 2, 3, 4, 5, 6

Krepko povezane komponente — Tarjanov algoritem



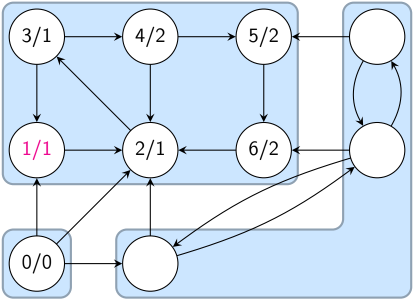
vozlišče 1 je obdelano
sklad: 0, 1, 2, 3, 4, 5, 6

Krepko povezane komponente — Tarjanov algoritem



num = *low*
sklad: 0, 1, 2, 3, 4, 5, 6

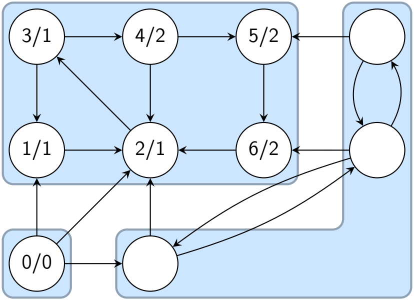
Krepko povezane komponente — Tarjanov algoritem



krepko povezana komponenta: {6, 5, 4, 3, 2, 1}

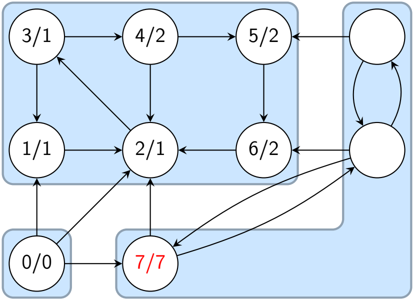
sklad: 0

Krepko povezane komponente — Tarjanov algoritem



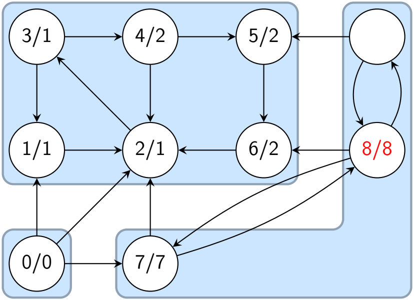
nadaljujemo z naslednjim sosedom vozlišča 0
sklad: 0

Krepko povezane komponente — Tarjanov algoritem



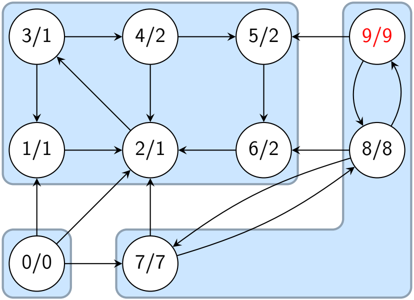
obiščemo vozlišče 7
sklad: 0, 7

Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 8
sklad: 0, 7, 8

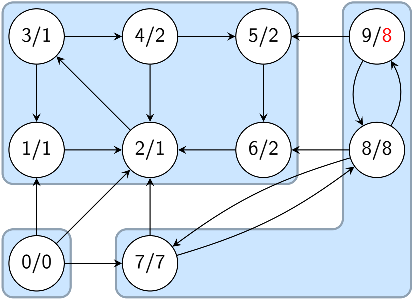
Krepko povezane komponente — Tarjanov algoritem



obiščemo vozlišče 9

sklad: 0, 7, 8, 9

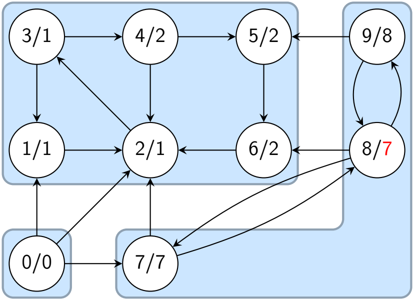
Krepko povezane komponente — Tarjanov algoritem



vozlišče 9 je obdelano

sklad: 0, 7, 8, 9

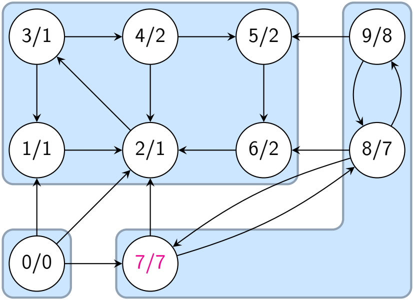
Krepko povezane komponente — Tarjanov algoritem



vozlišče 8 je obdelano

sklad: 0, 7, 8, 9

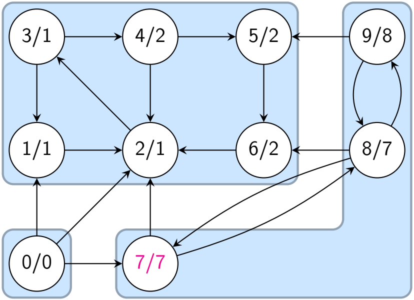
Krepko povezane komponente — Tarjanov algoritem



vozlišče 7 je obdelano

sklad: 0, 7, 8, 9

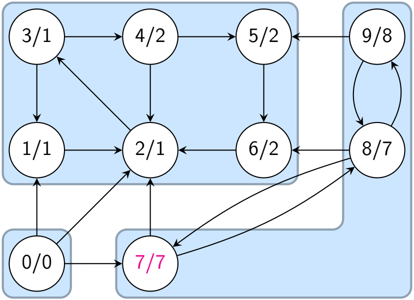
Krepko povezane komponente — Tarjanov algoritem



num = *low*

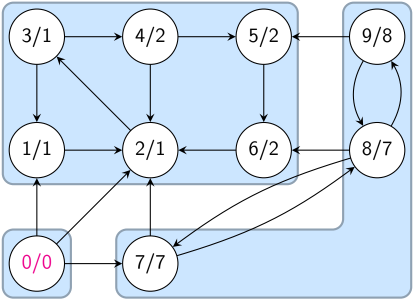
sklad: 0, 7, 8, 9

Krepko povezane komponente — Tarjanov algoritem



krepko povezana komponenta: {9, 8, 7}
sklad: 0

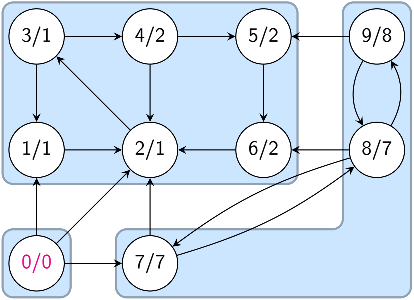
Krepko povezane komponente — Tarjanov algoritem



vozlišče 0 je obdelano

sklad: 0

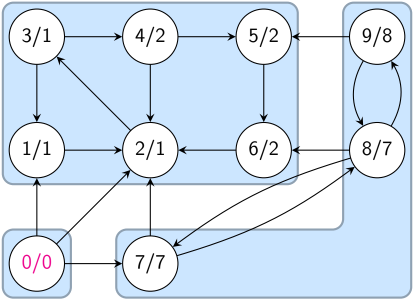
Krepko povezane komponente — Tarjanov algoritem



num = *low*

sklad: 0

Krepko povezane komponente — Tarjanov algoritem



krepko povezana komponenta: {0}
sklad: